The McGraw-Hill Companies

Embedded Realtime Systems Programming

The McGraw-Hill Companies

Embedded Realtime Systems Programming

Sriram V Iyer

Pankaj Gupta

Philips Semiconductors Design
Competence Centre, Bangalore

Tata McGraw-Hill Publishing Company Limited

New Delhi

McGraw-Hill Offices

New Delhi New York St Louis San Francisco Auckland Bogotá Caracas Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal San Juan Santiago Singapore Sydney Tokyo Toronto

Tata McGraw-Hill

ISBN 0-07-048284-5

Published by Tata McGraw-Hill Company Limited,
7 West Patel Nagar, New Delhi 110 008, typeset by Devendra M Sharma,
#5/5, 1st Cross, Gowdanapalya Main Road, Bangalore–560 061 and printed at XXXXXX,
Address…….....

To my parents, Venkateswaran R & Lalitha V
my sister, Harini Deepak
–Sriram V Iyer

To my parents, PremChand & Omwati
my brothers Deepak & Vijay and my wife, Purnima
–Pankaj Gupta

And to the Philips 'family'—as our CEO
Dr Bob Hoekstra rightly calls our organisation

Preface

As we think of writing the preface, we are filled with thoughts on this book, which took over 20 months to reach this shape.

Embedded industry is now in its adolescent phase—Too young to be called mature and too mature to be called nascent. It is currently experiencing the 'growing pains'. The industry once restricted to a very small community is beginning to embrace more and more architects and developers into its fold.

When we started out in this industry after our college, a sense of mystery took over. We were familiar with programming in the colleges. We did do a course on programming (typically C/C++/Java), data structures/algorithms etc as a part of our curriculum. Programming in an embedded-realtime scenario gave a feeling of *deja vu* of our previous programming experiences though new experiences were definitely different.

As we patiently took notes, conversed wit senior colleagues and by sifting through a mammoth amount of information in the web, things started becoming clear. Finally, we could comprehend the rules that differentiate embedded programming from normal desktop/applications programming. It by no way means we have reached *nirvana* of embedded-realtime programming. We have just crossed the first few basic steps that we would like to share it with you.

This time, we were asked by Mr. K.M. Jayanth, the Department Manager of the department we work for, to compile our thoughts, experiences and information we had gathered and create a primer on embedded-realtime software for our department. We ventured into this activity without knowing what we were stepping into. As we started we saw that the project was indeed huge. After numerous sleepless nights, gallons of caffeine and countless pizzas we could finally constrain ourselves to writing a 100 odd pages for that primer. We clearly felt that we were "hungry" for a lot more but had to restrict the content to this level for it to remain a '*primer*'.

At the same time, we got a mail from Deepa (Manager, Professional Publishing, TMH) if a book on C++ could be written on seeing some C++ articles* in ITspace.com. We decided that a book on programming embedded-realtime systems would be more appropriate for us, than a book on C++.

This book is our collective wisdom distilled over hours of design, development, integration and debugging experiences. The contents were also developed with interaction with many of

*Sriram V Iyer was the community guide/moderator for the C++ section in ITspace.com

our colleagues and intelligent conversations during the series of a seminar on embedded-realtime systems that we conduct in our department.

Audience

The audience of the book could range from a novice developer who is curious to know more about embedded systems. It could be of immense use to a practicing embedded engineer to know more about the covered topics. It could also be used as an undergraduate text for CS, EE and courses in Embedded-Realtime systems.

The book is not intended to replace courses in microprocessors, algorithms, programming but can definitely complement this courses.

Though this book talks a lot about software interactions with hardware, the focus of the book is restricted only to software. Though some insights are provided in some chapters, this book definitely does not address the issue of embedded hardware design.

Organisation of the Book

The Section 1 "*Introduction*" is an introduction to the theory behind embedded-realtime systems. It discusses the fundamental differences between programming in desktop and embedded systems. It describes various attributes of these systems along with some of their limitations that make programming these systems interesting!

The Section 2 "*Embedded nitty-gritty*" delves into the details of embedded systems programming. Chapter 2 describes the elements of the build process with emphasis on features unique to embedded systems. Details of various steps like pre-processing, compiling, linking, locating etc are covered in length. Chapter 3 describes various types of memories. The knowledge of these types of memory and how they work is essential for any embedded engineer when designing his system. Chapter 4 delves into various techniques to handle the memory that is so precious for any embedded system. Internal details of various memory allocation schemes are described. It also lists some interesting bugs and their solutions. The Chapter 5 on Interrupts and ISRs covers in great depth the concept of interrupts. Various types of interrupts, ISRs are covered. The chapter ends with general guidelines for writing efficient ISRs and some interesting debugging tips.

Section 3 "*Correctness is not enough*" focuses on the 'realtime' aspect. Chapter 6 describes the mathematics of scheduling theory and covers the famous Rate Monotonic Analysis (RMA). This chapter should be interesting for engineers to know the basics of realtime theory and hopefully inspire them to go further. The Chapter 7 on RTOS is no doubt one of the most famous topics among the embedded community. The RTOS chapter describes various RTOS concepts like tasks, queues, events, timers and race-conditions, priority inversion etc. It also has interesting discussions on priority inversion & Mars Rover mission, history on nomenclature in semaphores, etc.

Section 4, aptly named "*But I know this already!*" discusses various Software Engineering aspects of designing embedded-realtime systems. Many of the books on embedded-realtime

theory do not include content on software engineering aspects of embedded system design though many of the principles remain the same across various software domains. Chapter 8 describes in detail the "Requirements Engineering" principles that need to be followed to create a system as intended by the customer/end-user. Following sound requirements engineering concepts will help us avoid nasty surprises during the delivery of the system. Chapter 9 "Architecture and Design of Embedded Systems" covers the high-level software architecture and design aspects of embedded systems. The topic of software architecture and design is huge and many independent books are available to address the topic. The chapter focuses on some of the issues including some "Architecture Patterns" that are relevant to embedded-realtime systems. The Chapter 10 "Implementation Aspects in Embedded Systems" discusses a plethora of techniques that can be applied in embedded systems ranging from bit-fields to endianness, callback systems to state machines. Chapter 11 "Embedded Software Estimation Modelling" describes some of the techniques in software estimation in embedded systems. Estimation, as its name indicates is still considered more as an art and a result of experience rather than an engineering discipline. This chapter helps in making estimation of software in embedded systems more accurate. The Chapter 12 "Debugging and Validation of Embedded Systems" discusses one of the most dreaded times of any embedded engineer—debugging. This chapter throws light on some of the concepts on debugging and various tools available for the same.

The book ends with a detailed bibliography and an appendix on "C++ for Embedded/ Realtime Systems", a paper presented by Sriram V Iyer and Manoj Kamath (Philips Semiconductors, Philips Software Centre) in Philips TechSym 2001, a technology symposium conducted in Philips Innovation Campus.

Acknowledgements

The authors would like to express their heart-felt gratitude to Philips for providing a congenial and nurturing environment for our thoughts on the subject. We especially owe a lot to Jayanth, Department Manager DSS for providing constant support and feedback during the entire duration of this big exercise. We also thank Dr Antonio, Director PS DCC-B, Philips for finding time out of his busy schedule to review the manuscript and write the beautiful preface of the book. The writers would like to say, "thanks buddies" to our teams in Philips who constantly pulled our legs and created perfect atmosphere for this creative endeavor.☺

Sriram V Iyer

Pankaj Gupta

Foreword

In times when the pace of change was slow, the variety of products and services was small, the channels of communication and distribution less explosive, and the consumer needs less sophisticated, engineering could enjoy prolonged periods of relative stability. The times when the customer can be held almost constant and the optimisation of other variables could have been optimised are over. These times have long gone.

Now, human beings live in times of choice. They are continuously bombarded with purchasing alternatives in every aspect of their lives. They demand more and more from their purchases and their suppliers. The markets are fragmented and their products can be tailored by design, programmability, service and variety. In the world of high technology such as Semiconductors, there is an analogy that can explain this process: behind the proliferation of electronic components, infiltrating our communication systems, entertainment centers, transport, homes, there are thousands of integrated circuits that are produced in high volume up to the last layer, which in turn is designed by the customer to add the final touch of personality needed for their specific products. Radical customization dramatically has shortened time-to-market and time-to-money. This exemplifies the remaking of our means of production to accommodate our ever-changing social and personal needs.

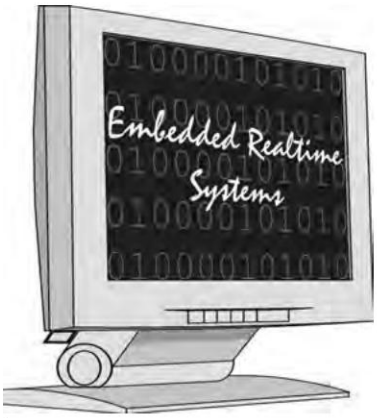
Programming embedded and realtime systems is not an exception to the rule—this also benefits from the flexibility the topic has to offer. It emphasizes the best-practice approach but allows the necessary customisation to shorten the time-to-market deliverables.

One of the functions of this book, and perhaps the most important one, is to open up the logic of applying the appropriate fundamentals on embedded software and realtime systems, so that everyone in the software industry can participate in the understanding of best practices. If prudence rather than brilliance is to be our guiding principle, then many fundamentals are far better than a series of sophisticated but isolated experiences. If embedded software is going to be the semiconductors driving force, and most of the semiconductor organisations are debating and insist this is their actual goal, then its fundamentals must be accessible to all players, and not as is sometimes the case, be reserved to an elect few who think that software engineering is just a part of the system approach.

Finally, I would like to make you think that this work is going to remain the basics of engineering despite the fast-paced, ever changing competitive world. This book will continue making you think about engineering basics and the way you think about engineering. But it will also make you think about fundamentals of software engineering.

I am confident the reader will enjoy and find Embedded-Realtime Systems Programming a useful experience.

Dr. Antonio M. Alvarez-Tinoco
Director, Philips Semiconductors
Design Competence Centre –Bangalore



Contents

Section Three: Correctness is not enough

Section Four: But, I know this already!

The McGraw·Hill Companies

SECTION ONE

Introduction

Main Entry: em•bed•ded
Pronunciation: im-be-ded
Type: *adjective*
: being a constituent within a similar surrounding.

Hardly convincing isn't it! That is why, we were motivated to write this book. Embedded systems seem to touch our lives everyday almost wherever we go. However, they still remain puddled inside a shroud of mystery, far from the normal world, being able to be unraveled only by elderly professors with flowing beards ☺.

No, embedded systems are not confined to these hallowed places alone. This section shall endeavor to introduce the reader to the common world applications of embedded systems. And it is our attempt to transform the reader's knowledge to an extent that (s)he looks at ordinary appliances at home or at work in a totally different light. Yes, we are talking of really ordinary appliances that have embedded systems inside them in some form or the other. We will then look at the unique challenges that lie in the path of engineers who create such systems and how it is different from normal systems. So, fasten your seat belts, here we make a take-off!

Chapter
1

Introduction to Embedded Realtime Systems

These are the days when the terms like embedded, ubiquitous and pervasive computing are increasingly becoming more and more popular in the world of programming. Embedded realtime programming was once looked upon as a niche skill that many programmers can keep themselves away from—but not any more. The focus is now on many smart and intelligent devices. The personal computer (PC)/workstation is moving away from the focal point of the computing/programming industry.

We are flooded with embedded systems that seem to be everywhere (ubiquitous) and inconspicuous. These systems should ideally communicate with each other (distributed) to achieve a feel of a complete system.

Before we delve further, we can define what an embedded system actually is. An embedded system is defined as "A microprocessor based system that does not look like a computer".*

If we look around, we will realise that there are a lot of devices with limited intelligence. Let us consider the good old washing machine. The main purpose of a washing machine is to wash clothes. But the modern world has extended it to include extra functions and give more control thereby optimising the actual process of washing clothes. Present day washing machines come complete with sensors, which maintain optimum water temperature, cloth dependent spin-speed, number of spins, etc. They take care of filling water, heating it to a particular temperature, mixing the optimum amount of detergent, soaking the clothes in water for just the right time, the soft tumble for

*Any correlation with any Zen quotation is purely coincidental.

extracting dirt, the aggressive tumble for removing stains, and excessive detergent from clothes, and finally the spin-dry. All this happens with minimum user intervention. The user may just have to select what kind of clothes are being put inside the machine and possibly how dirty they are!

This is not magic. All this is possible because somebody hit upon a brilliant idea that we can use a small microprocessor to automate a lot of the dreary process of washing. Since microprocessor cannot function in isolation, it needs inputs from sensors and other controlling devices so as to feel what is going around and then "decide" what actions need to be performed, which parts of the system have to run and in what order. The sensors detect that the quantity of water inside the machine is at a certain level and indicate this to the processor. The processor computes the required quantity of water that is necessary for the number of clothes and based on user settings. It then generates a control signal to stop the flow of water inside the machine and switch on the heater. The temperature detector keeps giving indications about the current temperature inside the washing machine compartment. At the optimum temperature for the kind of clothes to be washed, the processor generates a control signal to stop the heater. Then it gives a signal to start the soft tumble action to soak the clothes properly in water and mix the detergent. The processor will keep a watch on the amount of time the soft tumble action is going on. At the optimum time, it will stop the soft tumble action and start the aggressive tumble action to fight the stains. So, we can see that washing machine is an example of an embedded system. As illustrated, the seemingly simple task of washing clothes is a big exercise for the processor!

As embedded systems started progressing, they started becoming more and more complex. Additionally, new attributes that got added to these systems were smart and *intelligent*. Not only were the embedded devices able to do their jobs but also were able to do them smartly. What exactly do we mean by intelligence? Intelligence is one of the terms that cannot still be defined in a single concrete way (If it was indeed definable, we would have a laptop typing these pages on its own!). We can define a smart device as a device with the following attributes:

- ***Computational Power*** All these devices have some amount of computing power. This could be provided by a very simple 8-bit controller or a high-end 64-bit microprocessor.
- ***Memory*** The next requirement is memory. These devices possess some amount of memory that can be used by the processor and also some to remember user data and preferences.

- ***Realtime*** All the devices have to respond to user/environmental inputs within a specified period of time.

- ***Communication*** The device must be able to receive inputs given by other devices in the environment, process it and provide some tangible output to the other devices or users.

- ***Dynamic decisions*** The system should be able to change its next course of activity based on the change of input from its sensors or surroundings.

1.1 CHALLENGES FOR EMBEDDED SYSTEMS

Each of the attributes mentioned above is undergoing a series of transformations. The embedded processors are getting more and more powerful. It is not uncommon to find powerful 32-bit processors in high-end embedded systems like mobile (GSM/GPRS/3G) handsets, high-speed routers, bridges and even in seemingly small applications like a network interface card (NIC). Some applications like 802.11a (a faster version of normal Wireless LAN) require more million instructions per second (MIPS) than that can be provided by the advanced processors like the P4 2GHz processor.

Memory is getting cheaper and better. This is evident from the amount of memory being used in the small hand-held devices available today for wireless communication. In the times to come, wireless communication standards like Wireless LAN (802.11), Bluetooth, GPRS/3G will all help in providing inexpensive, untethered communication among devices.

Programming for these devices offers unique challenges not found in PC/workstation based applications. Some of these are listed below:

- ***Limited operating system (OS) support for programming*** Application programs for PCs/workstations are launched from the operating system. The tasks like scheduling, memory management, hardware abstractions and input/output from/ to peripherals are delegated to the OS. In embedded systems, the OS is part of application code and it closely co-ordinates with the OS to support a majority of the features that a desktop OS may provide.

- ***Limited secondary memory*** Many embedded systems do not boot from a hard disk. (A cell-phone with a hard disk? ☺). They depend on other types of non-volatile memory like read only memory (ROMs) and "FLASH memory" instead of secondary memory devices like the floppy disks or hard disks. Since we do not

talk about giga-bytes of flash (systems with 16 MB flash are considered premium), our code and data sizes must be small.

- ***Limited random access memory (RAM)*** Since embedded systems inherently operate with restrictions on resources, we do not usually have concepts of swapping, virtual memory, etc. in typical embedded systems. And, while programming for embedded systems, we must be very careful about memory leaks because, these programs tend to run forever. For example, a television goes to standby mode when it is switched off (unless the power is switched off). Some components take rest, while the program still runs anticipating commands from the remote commander. If the program ends when the television goes to standby mode, the television cannot be switched on again because there is no entity on television listening to the remote commander. These programs that run in embedded systems tend to run forever, and even a single byte leak in some path of execution will definitely bring the system to a grinding halt at a later point of time.

- ***Limited processing power*** We cannot afford to have a P4 2 GHz processor to power a microwave oven because of obvious cost considerations. We might have to work with microprocessors that clock 10–100 MHz or even with some microcontrollers with less powerful configurations. So, the code written must be efficient. We have to choose appropriate algorithms and cannot choose an algorithm with high computing requirement unnecessarily.

- ***Interaction with hardware*** This is the caveat. This factor singularly differentiates a normal application programming from embedded programming. An application programmer using the most modern OS can develop software blissfully unaware of the underlying hardware. Actually this is one of the underlying design principles and objectives of modern operating systems. But, an embedded programmer usually cannot afford this level of hardware independence since his code directly interacts with the underlying hardware. Embedded programmers usually have to work with realtime operating systems (RTOSes) that generally cannot provide such a high level of abstraction over hardware due to space and time restrictions and mind-boggling variety of hardware.

- ***Absence of standard Input/Output (I/O) devices*** A PC has standard I/O devices like keyboard, mouse and a display that can be used to peek into what's happening inside our program. But many of the embedded devices do not have such I/O devices. So, a programmer has no direct way of knowing what is happening within the system.

This seriously limits the amount of debugging possible on an embedded system working in the field.

1.2 FUNDAMENTAL COMPONENTS OF EMBEDDED SYSTEMS

Usually all embedded systems have a lot in common in terms of their components and their requirements. The following subsections introduce some of these requirements and components.

1.2.1 Computational/Processing power

This is one of the primary requirements of an embedded system. All systems take input from the user or the environment. The processing, however, can be done using microprocessors or a hydraulic circuit or a simple electrical/electronic circuit. The scope of this book is limited to systems that use a microprocessor with required hardware.

This processing power is required to translate the request from the user/changes in the environment to the output as desired by the end user (Fig. 1.1).

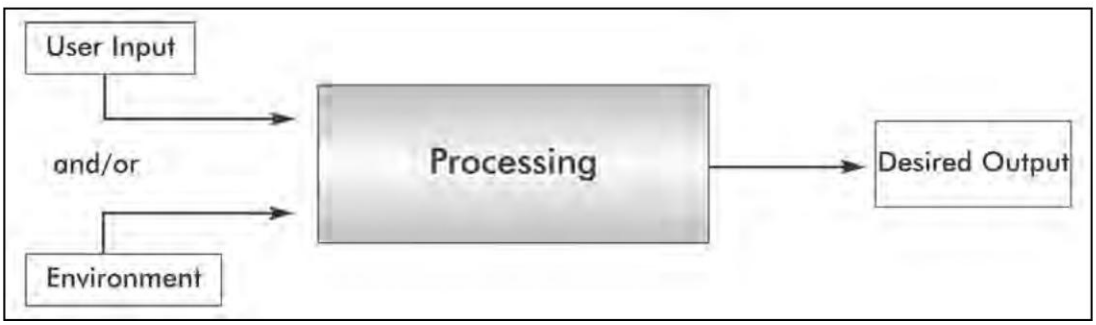


Fig. 1.1 *Processing Power*

This processing logic that used to be "hardwired" in a chip or other electrical circuits grew up exponentially and is so complex nowadays that many functionalities are simply unimaginable without software. The usual practice is to hardwire 'mature' features in hardware and use software to implement evolving features.

An embedded system can also take inputs from its environment. For example, consider a music system with preset options such as theatre, hall, rock, etc. A user can change the acoustic effect based on his requirements. In this case input is received from the user (Fig. 1.2).

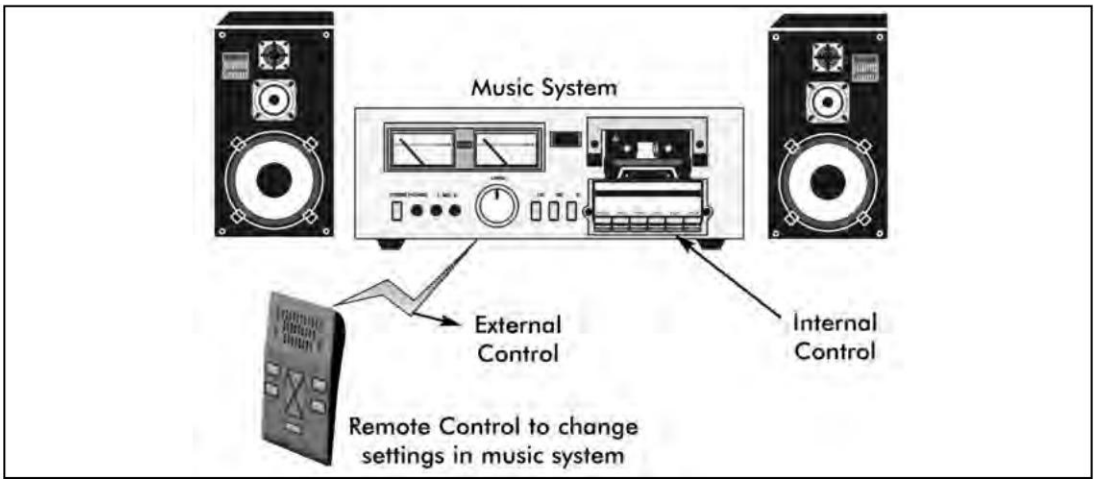


Fig. 1.2 *Input from the User of an embedded system*

For example, a refrigerator or an air-conditioner is more than just a compressor regulated by a thermostat. They have a control system that implements various functions like defrost, air circulation, apart from the seemingly dumb function of temperature control. Most of these systems come loaded with various settings. Some advanced refrigerators may have sensors to deodorize and detect inactivity. These control systems are usually implemented as software. They respond to environment apart from the user. For example, an air-conditioner will try to run the compressor for a longer duration if the ambient temperature is higher than the required level and based on the user preferences/presets he chooses.

To compute and regulate the various parameters a system may require various levels of computing power. The microcontroller can be chosen based on the required level of computing power.

1.2.2 Memory

Memory is a very precious resource and is always found wanting in many embedded systems (including human systems ☺).

It is indeed true that memory is becoming cheaper nowadays. But, in these days of intense price wars, every resource must be handled with extreme care. And, in many systems, some space has to be allocated for future expansions. Also, we cannot afford expansion slots as in PC for embedded systems due to cost constraints,

embedded-hardware design constraints and form-factor* restrictions. So, memory should be handled very carefully.

These constraints on memory will become self evident by looking at good embedded system designs and its software. Algorithms that use a huge amount of memory or copying of huge data structures are ignored unless it is an absolute necessity.

Much of embedded system software is such that it directly uses memory instead of high-level abstractions. Many RTOSes, however do provide routines that hide the complexity of memory management.

Many embedded systems do not carry hard disk or floppy disk drives with them. The usage of secondary storage is not possible in most embedded systems. So, these systems usually have some ROM and nonvolatile RAM where the code and user preferences are stored. (Various memories that are used and the programming techniques to handle memory efficiently are discussed separately in Chapter 4).

We have to remember that most of the programs do not terminate (when was the last time you "rebooted" your refrigerator?) and tend to run forever. In case of mission-critical systems, when emergency strikes or when some irrecoverable error occurs, embedded systems implement what are called *watchdog timers* which just reset the system.

1.2.3 Realtime

We can define a system as a collection of subsystems or components that respond to the inputs from the user or the environment or from itself (e.g. timers). Typically, there is a time lapse between the time at which the input is given and the time at which the system responds. In any system, it is quite natural to expect some response within a specific time interval. But, there are systems where, very strict (not necessarily short) deadlines have to be met. These systems are called realtime systems. These systems are characterised by the well-known one-liner:

P2P

"A late answer is a wrong answer".

Realtime systems can be classified in general as

- Hard Realtime Systems
- Soft Realtime Systems

*The size/form and shape of the appliance (or device). People will not buy a cell phone as big as a dumbbell just because it can be enhanced with more features in the future owing to its expansion slots.

- ***Hard realtime systems*** A realtime system where missing a deadline could cause drastic results that could lead to loss of life and/or property is called a hard realtime system.
  Examples are aircrafts, biomedical instruments (like pacemakers), nuclear reactors, etc.
  For example, fighter jets *have* to respond to the air force pilot immediately.

- ***Soft realtime systems*** A realtime system where a few missed deadlines may not cause any significant inconvenience to the user is known as a soft realtime system.
  Examples are televisions, multimedia streaming over Internet (where loss of some packets can be afforded).
  There is widespread confusion about 'hard and fast' realtime systems and soft and slow realtime systems.

The realtime systems can also be classified as fast and slow systems based on the time deadlines they operate with. Again, this is a very subjective definition. Typically, any system that works with subsecond response times can be classified as a 'fast' realtime system. The other systems that can take a second or more time to respond can be classified as 'slow' realtime systems.

Soft realtime systems can be fast. A few packets can be lost without loss to life and limb across a high-speed router that works with nanosecond deadlines. Similarly, hard realtime systems can also be slow—the cadmium rods inside a nuclear reactor need not be pulled out at lightening speed.

Closely associated with the concept of realtime is the concept of *determinism.* This is also a very important concept that differentiates realtime programming from normal application programming.

We have seen that a realtime system is one that behaves predictably—it responds within a particular amount of time. The time interval between the instant at which the input occurred to the time instance at which output occurs should be 'deterministic' or predictable. This does not necessarily require that the systems must be fast. It only requires that the system should always respond within a known period of time.

For e.g., Java though is highly acclaimed as portable that makes it ideal for embedded software systems that runs on various types of platforms, was found not ideally suited for realtime systems because, some of the calls are not deterministic.

Java implements the concept of *automatic garbage collection.** This means that the programmer need not remember to free any of his memory. The logic for collection of unused memory is built into the runtime. But this could cause problems.

Whenever memory for the objects are allocated from the heap, if there is not sufficient memory, then the garbage collection algorithm is executed till all the memory is reclaimed. This could stop our program from execution for an unknown period of time. As we will see later, this is a major issue in realtime systems wherever every function call must be deterministic.

1.2.4 Communication elements

Embedded devices and appliances can no longer remain as islands of information storage. They need to communicate with each other to perform any operation that is desired by the user. We simply cannot ask the user to carry a cable (rope?) to tether the device to an ethernet socket. These communications should typically occur using wireless networking protocols like Bluetooth™, Wireless LAN (WLAN), HiperLAN for short distances and 2.5G (GPRS), 3G (CDMA/WCDMA), 4G protocols over long distance. The communication element adds 'intelligence' to a simple embedded realtime system.

The other important 'soft' parameters that define an embedded system (and its software) are

- ***Cost*** Cost is often the major driving factor behind many embedded systems. This requires the designer to be extremely conscious about memory, peripherals, etc. This factor plays a key role in high volume products. But, some highly specific applications like avionics can be chosen to be expensive.
- ***Reliability*** Some products require a 99.999% uptime. Typical examples are routers, bridges, power systems, etc. But some may not require this kind of reliability (It is OK if your bread gets a bit overdone once in a while in

*Garbage collection is an oft-misused term. In many languages, the programmer has to take care of his/her 'garbage', i.e. the memory she/he no longer uses. She/he has to remember to free the memory manually. In some languages like Java, there is an option of the language-runtime (Java Virtual Machine—JVM) taking care of freeing unused memory. This is called 'automatic garbage collection'. We have to remember that garbage collection must be done—either manually by the programmer or automatically by the language runtime.

your microwave). Reliability may require the designer to opt for some level of *redundancy*.* This could make the system more expensive.

- ***Lifetime*** Products that have a longer lifetime must be built with robust and proven components.

- ***Power consumption*** This is becoming an important area of research in itself. With growing number of mobile instruments, power consumption has become a major concern. The problem was first encountered while designing laptops. The laptop seemed to siphon off the power in no time. Similarly, many of today's devices are mobile—like the cellular phone, PDA, to quote a popular few. The design of these devices is such that the power consumption is reduced to the minimum. Some of the popular tactics used include shutting down those peripherals which are not immediately required. These tactics are highly dependent on software. This factor has reached new dimensions with new processors being designed such that some of their parts can be shut down whenever not required. This requires a shift to a new era of programming with more dimensions being added to embedded software programming. The programmer for mobile devices is becoming increasingly aware of the power-saving features in his programming platform (peripherals and processors).

These are some of the soft factors that drive design of embedded systems and its software.

1.3 EXAMPLES OF EMBEDDED SYSTEMS

Let us see some of the typical embedded systems that surround us.

1.3.1 Music systems

Today's advanced music systems are very complex embedded devices. They contain a multitude of features such as the ability to play various types of media like the magnetic tape (cassettes), audio/video CDs, DVDs, etc. They also have support features for many kinds of presets for different types of music like jazz, rock, classical, vocal, etc. and the

*Having some duplicate peripherals that can be used when the main peripheral fails is called redundancy. This duplicate peripheral is not in use always. It is used only when the main device fails. There could be any level of redundancy. In highly critical systems more levels of redundancy can be provided. If the swapping between the failed main device and the redundant device can occur without a power-down or restart, then the device is said to be 'hot-swappable'. It is called 'cold-swappable' otherwise.

environment (hall, theatre, open-air, etc). These features are not hardwired in chips but are usually taken care of by the software that goes with these systems. The processors are typically 8-bit microprocessors for handling user inputs and display. Additionally, they have a high-end DSP 16-bit/32-bit microprocessor and/or MPEG2, MPEG4 decoders for decoding the input stream for various supported media. The RAM for these kinds of systems can vary a lot from 64KB to a few MB depending on how complex the system is.

On the realtime front, the media should be read, decoded and the stream must be sent to the speakers/video output at a predefined rate. Based on the media the requirements for this data throughput may vary. Imagine a Bluetooth™ network that takes care of playing your favourite music as you enter the house (by contacting your PDA). This system needs to interact with its components as well as other devices in realtime so that the desired functionality (playing of favourite music) is achieved.

1.3.2 Card reader

This is one of the systems that is often encountered in buildings that incorporate a security system. A person who wants to gain entry to the systems must flash his/her magnetic card in front of the reader. The reader then validates the card and may provide or deny access based on the privileges given to the card. The system does not have a very complex software.

On flashing of the card as detected by the magnetic sensor, the card identifier is looked upon in the access control list. If the card does have the access permit then the LEDs on the unit flashes and the door is unlocked for entry. Or, the system can emit a sound or display that the access is not permitted. The unit should just look up the access table and respond to the user.

However, this should happen sufficiently fast—typically in less than a second. We cannot allow even a few seconds lapse because, the user may assume that his access was not permitted or that the system is dysfunctional. The lists can be stored in a central server where the look up can be done. In this case, the authentication unit may not require storing of all the lists in its memory. Or, it can store the list only for the location for which it controls the access.

This is left entirely to the discretion of the designer of the system. The memory required for this system will depend on the method opted for its design. These units are connected with each other, usually using some kind of Ethernet connection.

1.4 LANGUAGES FOR PROGRAMMING EMBEDDED SYSTEMS

Assembly language was the *lingua franca* for programming embedded systems till recently. Nowadays there are many languages to program them—C,C++, Ada, Forth and … Java together with its new *avatar* J2ME. Embedded software is coming of age and it is fast catching up with application software. The presence of tools to model the software in UML, SDL is sufficient to indicate the maturity of embedded software programming.

But, the majority of software for embedded systems is still done in C. Recent survey indicates that approximately 45% of the embedded software is still being done in C. C++ is also increasing its presence in embedded systems. C++ is based on C, and helps the programmer to pace his transition to OO methodology and reap the benefits of such an approach.

C is very close to assembly programming and it allows very easy access to underlying hardware. A huge number of high quality compilers and debugging tools are available for C. Though C++ is theoretically as efficient as C, some of its compilers are buggy due to the huge size of the language. These compilers may create a buggy executable in some situations. C can definitely claim to have more mature compilers than C++. And in C++, some of the features do cause a lot of code to bloat. Actually there is an ongoing effort to identify a subset of C++ that can be used in embedded systems. This subset is called the *Embedded C++*.*

In this book, we concentrate on C and we use C++ wherever applicable. The myths that surround C++ and implications of using it in embedded systems are discussed in Appendix A.

1.5 ORGANISATION OF THE BOOK

In this book, we will explore what an embedded system is, various types of embedded systems, techniques to program them, and major concepts that are required to be mastered for efficient design and implementation of embedded system software. We will also take a peek into the process of developing efficient embedded software and its potential pitfalls.

*For more information look into the site for Embedded C++ - http://www.caravan.net/ec2plus/.

However, the book is NOT

- An ASIC developer's guide
- Guide for board design/board layout
- Detailed guide to semiconductor/digital techniques
- User guide to any embedded programming language

This book aims to explore the basic concepts that underline an embedded system and its software.

This book has been divided into following sections

- Introduction to Embedded Systems
- Embedded System and Microprocessor concepts relevant to building software for embedded systems
- Software Engineering practices: requirements, design, implementation, estimation and testing aspects

In Part I, the basic functionalities of an embedded system are defined. This part shall be the springboard for the rest of the book, by giving useful insights into the importance and organisation of rest of the sections.

A microprocessor or a microcontroller has loads of features. In Part II, we discuss concepts like interrupts, hardware timers, memory types and its management.

Embedded software has a lot of components that can be found in many systems. For example, state machines/state charts are used in many communication protocols. Task-based design is also one of the ubiquitous programming practice. In part III some of the common design paradigms that a programmer can immediately benefit from are described.

Software is complete only with its corresponding engineering practices. Some of the software engineering issues are dealt with in Part IV. Estimation, requirements gathering, architecture definition, design, implementation and testing of embedded systems should be familiar to embedded programmers. This helps in improving over-all quality of the system. All the case studies and examples discussed in the chapters above are used to create a complete case study to help a programmer understand the complete Software Development Lifecycle (SDLC).

The content of the book in all the chapters is based on the Learning Pyramid™ as indicated in ancient Indian texts (Vedas) (Fig. 1.3).

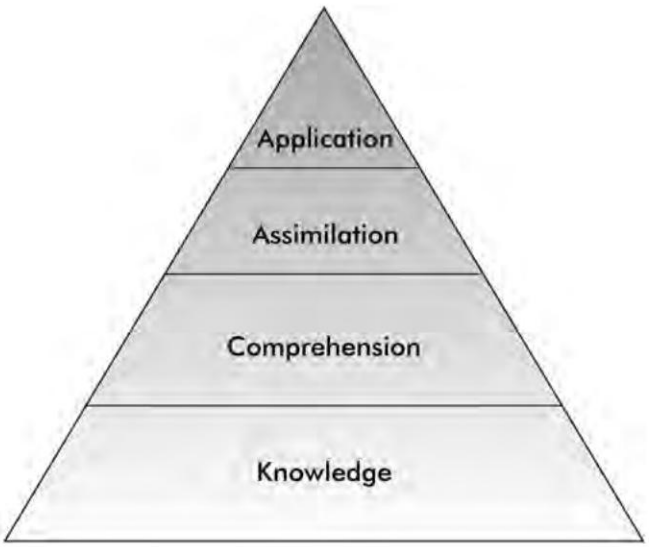


Fig. 1.3 *The Learning pyramid*

Learning is usually completed in four steps that are described below:

The first stage is *knowledge*—This consists of the definitions and the theory that go along. Knowledge as information is the foundation of learning cycle. At the beginning of any learning cycle, knowledge is not usually exhaustive.

The next stage is *comprehension*—This consists of explanation of the theory and how the things happen the way they happen. At this stage, the learner is able to generalise information.

Note

Watchdog Timers: *Watchdog timer is a mechanism to monitor the activity (rather inactivity) in a system. A watchdog timer periodically checks for a predefined activity. (It could be looking for a specific value to be updated periodically in a particular memory location). If the watchdog senses that the value has been updated as expected, then, it concludes that the system is an irrecoverable state (either the code is stuck in a loop, or it has crashed, etc.) and resets the system.*

The third stage is *assimilation*—This consists of insights and examples to get deep understanding of the system and interrelation between various information groups.

The final stage is *application*—This stage includes real case studies that would help the reader exercise all that he has assimilated. This involves use of knowledge, after comprehension and assimilation, production of an experience by using it to solve a problem in real life.

Thus the Learning Pyramid™ helps in complete coverage of any subject, and specifically here "Embedded realtime systems" and its software.

Learning step	Mechanism used in this book
Knowledge	Definitions, theory
Comprehension	Explanation of definitions and theory
Assimilation	Tips, warnings, examples
Application	Case studies, exercises

Table 1.1: ***Characteristics of Embedded systems***

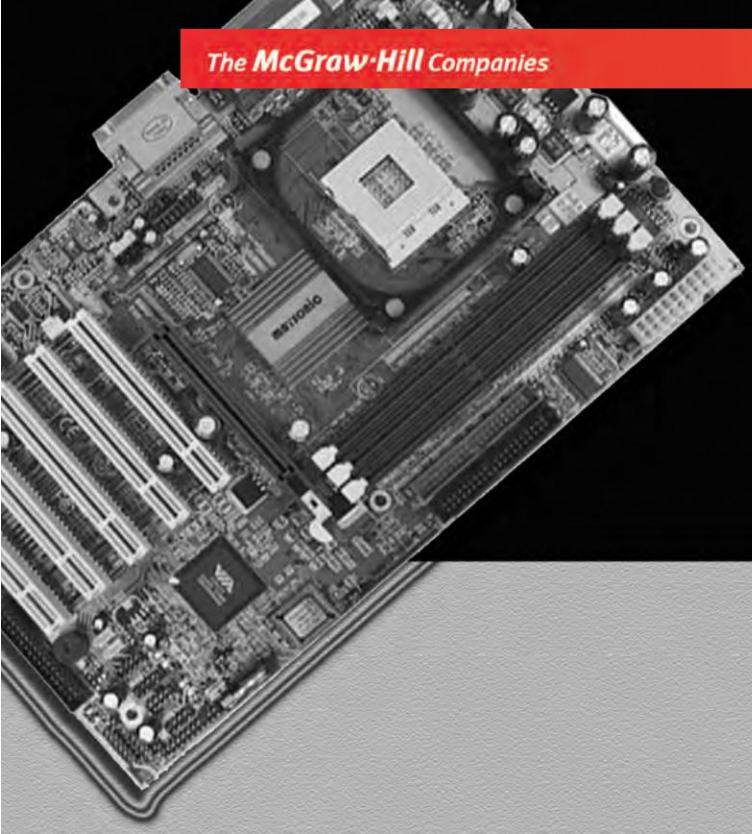
1.6 LESSONS LEARNT

In this chapter, we learnt about typical characteristics and features of embedded systems. Embedded systems can be found all around us—in washing machines, in music systems, remote controls and the like. Embedded systems have been increasing in complexity and intelligence constantly. It is a challenge to balance the strict restriction on memory and size of embedded systems with more computational power. In addition, programming for embedded devices has its own problems such as limited OS support, lack of standard I/O devices, limited memory and interaction with hardware devices in realtime. C replaced Assembly as the most commonly used language for programming embedded systems as of today because of ease of programming and its compact code generation. Other languages such as C++ and Java are picking up too.

1.7 REVIEW QUESTIONS

- What is meant by a smart device? What are its usual attributes?
- What are the typical challenges of programming for embedded systems?
- What is meant by realtime? What are the different categories of realtime?
- What is meant by determinism?
- Why is power consumption a serious issue in embedded systems?
- What is meant by garbage collection? Why does it render Java (in its normal version) in its current form nonsuitable as a language to implement embedded systems?

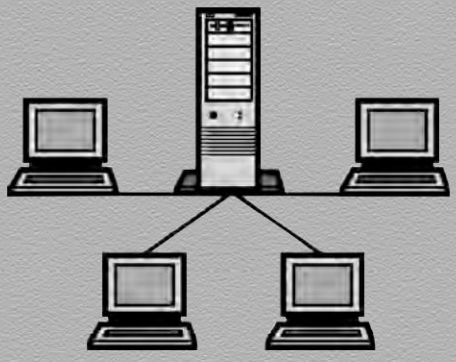
The McGraw-Hill Companies



SECTION TWO

Embedded nitty-gritty

Chapter 2 gives an insight into the build process associated with embedded systems, and we understand why and how it is different from a traditional compilation process. Memory is one of the most important parts of an embedded system. So it always pays for an embedded programmer to know something about its organisation, access methods and the associated circuitry in order to get a feel of things. We take a look at them in Chapters 3 and 4. Also, most embedded systems interact with the external world using the 'interrupt' mechanism.



Chapter
2

The Build Process for Embedded Systems

The process of translating the code that is written by humans to the code that is understandable by the microprocessor is called the *build process* (Fig. 2.1).

Fig. 2.1 ***The Build Process***

An embedded programmer must understand build process deeper than an application developer. This chapter describes the build process in detail. The process in embedded systems is almost the same as PC based development, but for some subtle changes that will be indicated when appropriate.

Some of the topics discussed here may not be specific to embedded systems per se. But, the discussions common to both embedded and applications (nonembedded) are added in this chapter to make the chapter complete and the book, a stand alone entity.

For C/C++ programs, the initiation of the build process varies from using a simple command from the command line (mostly in case of trivial programs) to huge makefiles and sophisticated build tools.

The steps that are involved in transforming the source code (the code created by the programmer) to the final executable format are listed below:

i. Preprocessing

ii. Compiling

iii. Linking

iv. Locating

Let us consider development of a program, say—foo.c

To build the corresponding executable, we give a command (in Unix) like

```
$ cc foo.c
```

Or, in Windows command line

```
E:\> bcc32 foo.c
```

to invoke the Borland compiler. If you are using an integrated development environment (IDE), then you could start building by pressing some special keys directly from the IDE editor.

If everything goes well, then we can execute the program by

```
$ ./a.out (or)
E:\> foo
```

There are various steps involved in translation of foo.c to a.out (or foo.exe). Each of the steps is described below in the following sections.

2.1 PREPROCESSING

This is the first step in the build process. The whole of the build process is accomplished by a set of executables (not just cc/bcc32). Preprocessing is accomplished using an executable usually called the `cpp` (C Pre-Processor) in the case of Unix* and `cpp32.exe`/`cpp.exe` in the case of Windows™ machines.

The pre-processor is automatically invoked by 'cc' (or cl or bcc32 or gcc, etc.) and its output is a temporary file, which is deleted at the end of the compilation process.

*When we mean Unix it is usually *nix (The '*' symbol is to represent a wild card and not to swear at Unix). Unix in this book, is Unix and its clones (HP-UX, Solaris, Linux and our favourite–AIX).

The preprocessor does the following jobs:

i. Strips the comments

ii. Expands include files

iii. Expands MACROs and replaces symbolic constants (simply put, the #defines ☺)

Including files is a great mechanism to maintain code modularity thought upon years before. Though other mechanisms have evolved, they are based on similar concepts and we still cannot do away with including files. Let us see a sample header file and see what is done while including header files.

P2P

Can we include .c (source) files?

```
// foo.h

#ifndef MY_FOO_H
#define MY_FOO_H

/*
    Preprocessor does not bother typedefs. typedef is a
    compiler feature. So, note the semicolon in the end.
*/

typedef       int         MY_INT;

/*
     PI is a symbolic constant. We can use PI wherever we
     can use 3.14159
*/

#define       PI          3.14159

/*
  SQR is a macro. Though preprocessor is a text replacement
  program, macros can be used effectively to improve our
  coding. (though they have a tendency to introduce bugs.)
*/

#define       SQR(x)      (x)*(x)

void    myFoo ( MY_INT );

#endif
```

Listing 2.1: *foo.h*

```
// foo.c

#include "foo.h"

int main (void)
{
     int i = SQR(2);
     float r = 1.0,
            area = PI * SQR(r);

     myFoo(i);

     return 0;
}

void myFoo (int i)
{
}
```

Listing 2.2: *foo.c*

Now, let us look at `foo.c` after it has been preprocessed. (Please note that during the normal compilation process, this file is not generated. This file is a temporary file that gets deleted after the compiler uses it. However, preprocessors can be used directly and have options to dump the preprocessed file in the screen or be redirected to a file. We can use the corresponding preprocessor available in our platform to generate the preprocessed file). We used the preprocessor cpp32.exe freely available Borland C/C++ compiler provided by Borland™ Inc. The comments are inserted by the preprocessor to improve our readability and will not be present in the actual output given to the compiler (i.e. in the temporary file created and given to the compiler).

Note

To create this preprocessed output, we used the C *Preprocessor* (`cpp32.exe`) *provided by Borland free command line tools.*

```
cpp32   -Ic:\Borland\bcc55\Include foo.c
```

This produced the following file (`foo.i`)

```
/* foo.c 1: */
/* foo.c 2: */
/* foo.c 3: */

/* foo.h 1: */

/* foo.h 2: */

/* foo.h 3: */

/* foo.h 4: */
/* foo.h 5: */
/* foo.h 6: */
/* foo.h 7: */
/* foo.h 8: */
/* foo.h 9: */
/* foo.h 10: */
/* foo.h 11: */typedef  int
MY_INT;
/* foo.h 12: */
/* foo.h 13: */
/* foo.h 14: */
/* foo.h 15: */
/* foo.h 16: */
/* foo.h 17: */
/* foo.h 18: */
/* foo.h 19: */
/* foo.h 20: */
/* foo.h 21: */
/* foo.h 22: */
```

```
/* foo.h 23: */
/* foo.h 24: */
/* foo.h 25: */
/* foo.h 26: */
/* foo.h 27: */void     myFoo (
int );
/* foo.h 28: */
/* foo.h 29: */
/* foo.h 30: */
/* foo.h 31: */
/* foo.h 32: */
/* foo.c 4: */
/* foo.c 5: */int main (void)
/* foo.c 6: */{
/* foo.c 7: */int i = (2)*(2);
/* foo.c 8: */float r = 1.0,
/* foo.c 9: */area = 3.14159 *
(r)*(r);
/* foo.c 10: */
/* foo.c 11: */myFoo(i);
/* foo.c 12: */
/* foo.c 13: */return 0;
/* foo.c 14: */}
/* foo.c 15: */
/* foo.c 16: */void myFoo
(MY_INT i)
/* foo.c 17: */{
/* foo.c 18: */}
/* foo.c 19: */
/* foo.c 20: */
```

After preprocessing foo.c will look like above. Note that the header file foo.h is expanded (foo.c lines are in bold). Look at the lines foo.c #5 and foo.c #9 respectively. The preprocessor has expanded the macros SQR and replaced PI.

It should be noted that the compiler sees the .c file only after preprocessing. So, the compiler *cannot* see if we had manually typed in 3.14159 or the preprocessor replaced

PI with 3.14159. The compiler/linker usually create entries in the final executable to add debugging information. Since the compiler cannot identify preprocessor symbols (because they have been removed already when it comes to the compiler), it cannot add any debug information about preprocessor symbols in the executable. So, debugging preprocessor macros is extremely difficult (if not impossible).

2.2 COMPILING

This is one of the most important steps in the build process where the object code is created from the source code. The name for this step has become synonymous with the entire build process.

In the compiling parlance, the code that the user creates is called the source code and the output of the compiler is called object code.

In this step, the code written by the user is converted to machine understandable code. The steps involved in compilation process can be split as

i. Parsing

ii. Object code generation

In the *parsing* step, the compiler parses the source file to validate the use of variables and checks that the language grammar/semantics are not violated. The parsing step also makes a note of the external variables that are used from other modules and the variables exported to other modules.

In the next step, the object code is generated from the corresponding high-level language statements. Some compilers choose to create an intermediate assembly file and invoke the assembler to create object code from the assembly listing produced.

The object code created cannot be executed yet. One of the important points to be observed is that the compiler works on a single source file at a time. In compiler parlance, each source file is called a '*translation unit*' (TU). An object file (.o or .obj) is created for every translation unit. A TU is typically the preprocessed file produced by the preprocessor.

P2P

Can a .obj file be created for a .h file?

There exists a one to one relation between every source file and object file (At least, for every compiler that we know of replace a .c extension with a .o/.obj extension for the object file name).

2.3 CROSS COMPILING

Cross compiling is a term unique to embedded environments. To really understand what it is, let us look at a typical application build/execute process in a workstation.

The user types in the code in an editor (or in some sophisticated integrated development environment (IDE)). Then he invokes the compiler. We should note that the compiler is also an executable program that runs on the workstation. The compiler compiles the source code written by the programmer into machine language code '*understandable*' by the processor in the workstation.

If the build is successful, the developer can execute the program in his workstation.

Now, let us carefully observe the following points:

i. The compiler itself is a program that runs in the (processor of the) workstation.

ii. The compiler creates object code that is understandable by the processor in the workstation.

If we look at the second point, it is really not necessary for a compiler to produce object code that is executable in the same workstation it runs. Both are logically independent. A compiler can also produce object code that can be executed on a different processor. (The catch here is that the build code cannot be executed immediately in the same workstation in the absence of emulators)

Before venturing into the details of cross compiling, we have to know briefly, how software development is carried out for embedded systems.

The embedded software is finally executed on boards like the one shown in Fig. 2.2.

A board has a processor (ARM/MIPS/x86), some RAM, ROM, Flash and some interconnect devices like Ethernet, UARTs etc. They usually don't have any display or keyboard attached. So, it is not possible to do any development of software for the board on the board. If you have worked on an 8085 board, then you would remember that the programs used to be typed in directly using a small numeric keyboard directly in machine language.

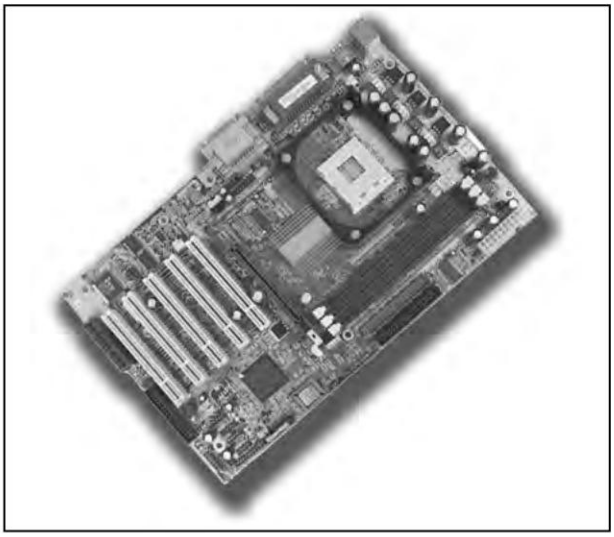


Fig. 2.2 *Typical development platform*

So, wouldn't it be nice if we can build the code for the board in the comfort of development environment in a PC/workstation, but execute the code in the target? A cross compiler helps us do exactly this. Though a cross compiler is run on the host like a Unix/Windows workstation, the object code cannot usually be executed in the same workstation.

Definition

A cross compiler is defined as a *compiler* that produces object code for the *processor in the target* rather than the host in which the compiler is executing.*

Analysis:

Compiler The cross compiler is also a compiler because it converts the source code to object code.

Processor in the target The targets are usually boards/platforms that do not have a display and keyboard. So, the compilation is done on a PC/Unix workstation. But the object code produced by the cross compiler executes in the target and not on the host.

*Not to mention that both the host and target processors can be same in which case the executable can be run on the host also.

2.4 LINKING

The process of compilation ends after creating object files for every source file (translation unit). We still do not have a single executable. The object files though in machine understandable format, are incomplete. Some of the incomplete parts could be:

a. ***References to external variables***: Consider the case when a project consists of two files. The t2.c declares a global variable foo that is an integer and t1.c refers to it by an external reference. Since the compiler works only on a single translation unit at a time, while compiling t1.c, the compiler can only assume about the existence of foo and does not know about exact location of foo. In the case of t1.c, the compiler adds the name of foo to the list of 'imported' symbols when it adds it to list of 'exported' symbols while compiling t2.c. It is the duty of the linker to resolve these external symbols (now does the linker error 'unresolved external XXXX' mean more to you?). This is not limited to variables. The linker also links the references to functions that are defined in other source files and libraries.

```
//t1.c

extern int foo;

// ...
```

```
//t2.c

int foo;

// ...
```

Fig. 2.3 *Extern references*

b. ***No binding to real address***: Again, due to the nature of the compiler (working with one file at a time), the addresses of different variables in different files will be assigned relative to the particular file. When all the object files are linked together to a single executable file, the address assigned to all these variables must be unique. The linker takes care of assigning correct addresses to these variables. Code in a file may also refer to a function in some other file. The linker fills these addresses of the functions. Then, the code for a particular function may be in a library. The linker will search the library and link appropriate code with the application.

2.5 LOCATING

The output of the linker is a single executable. The linker would have resolved all the external references among the object files, linked up the code from library, etc. Still, the executable is not ready to run on the target!

The final step (at last) that should be done is called 'locating'. This step is unheard of among developers who work with abstraction levels of COM, .net, etc. Though this is done for every executable, this step is of high importance to embedded systems because, this step requires a lot of input from the programmer.

This step finally adds the target specific information into the executable. This was not required in the case of application programs because the OS takes care of most of the locating issues. Typically, the locating is done at load time in many operating systems. This is not the case in embedded systems, where locating must be done because development is done on hardware platforms unique to each project.

A dedicated OS on the target that loads the executable to the memory is unheard of in embedded systems. The programmer must explicitly provide the hardware specific information of location of ROM, RAM, etc. and their sizes.

This information can be provided either as a command line option or inside linker scripts. The linker (ld) in GNU GCC compiler collection has a sophisticated linker script language that can be used to finely control the final executable. ARM linker provided with the ARM developer suite (ADS 2.2) also has a good support for these scripts.

The locator may be available as a separate program or be bundled together with the linker. Please refer to the toolset documentation on specific information for controlling the linker/locator.

2.6 COMPILER DRIVER

By this time we are fairly comfortable with the various steps in the build process. The entire build process is not done by a single huge monolithic executable. It is instead accomplished by a set of executables. For instance, we saw that the preprocessing in our case was done by cpp32.exe. But, who takes care of invoking the right component at the right time?

The answer to question is: *The compiler driver.*

The compiler driver is one of the least known terms among the developer community. We usually think all the work of building is done by cc / bcc32/cl. But, this is not true. cc is simply a compiler driver that takes care of invoking the right component at the right time and passing the correct inputs to them.

The functions of a compiler driver can be listed as below:

- Determine the type of the file and invoke the appropriate component: In an embedded application we typically have a list of C and assembly language files (.c and .asm). In some cases, we might need to build an application using a mix of C++/C and assembly language files. The compiler driver takes care of invoking the C compiler or C++ compiler or the assembler based on the extension of the appropriate file.
- Sequence the build process: The compiler driver not only takes care of calling the right component, but also at the right time. The entire build process is sequenced as preprocessing, compiling, assembling, linking, etc. by the compiler driver.
- Pass the arguments: The user can pass command line arguments and parameters to various modules using the compiler driver. The driver takes care of passing the arguments to the appropriate module. (For example, linker commands to ld and compiler command to cc (in the case of Unix)).
- Pass default arguments: It could be too cumbersome to pass every default argument of the compiler and linker. So, the compiler driver relieves the burden of passing the default options to the various components.

2.7 LINKER MAP FILES

So far, we were only talking about providing various inputs to the linker/compiler, etc. At the same time, we can request the linker to provide some output too (other than the executable, obviously ☺).

At this stage, before continuing, the reader may require a small deviation to look at section 2.10 in order to understand the concept of segments. At this stage of build process, we do not know the exact location /address of (initialised) global variables, various functions, etc. We also do not know the size of each function (in bytes), the amount of debug information, etc. These are decided by the linker and known only to the linker. To know these details, we can request the linker to generate MAP files.

The generated MAP file is huge but some of the contents are showed in Fig. 2.4 for explanatory purposes. (The map file was generated using Microsoft cl compiler (32-bit C/C++ Standard Compiler Version 13.00.9466 for 80x86)).

```
     Start          Length     Name                    Class
 0001:00000000    000050faH    .text                    CODE
 ...
 0003:00000030    00000590H    .data                    DATA
 0003:000005c0    00000594H    .bss                     DATA

0001:00000000       _main                        00401000 f   foo.obj
0001:0000002b       _myFoo                       0040102b f   foo.obj

0003:00000030       _myvar_init                  00408030     foo.obj
```

Fig. 2.4 *Sample MAP file excerpt*

From the excerpt above, we can see that the .text segment (containing the executable code) has segment number '0001'. The data segments (the initialised and un-initialised data) belong to segment 3 (0003). The C program has 3 symbols described in the map file. The first two are functions (foo and main) that belong to the text segment. (indicated by symbol 'f' in the MAP file)

Note

MAP file can be generated using the –M option provided for the Borland compiler.

```
bcc32 -Ic:\Borland\bcc55\Include
-Lc:\Borland\bcc55\Lib -M foo .c
```

We had two global variables. As indicated above, only the initialised variable is assigned an address in the map file. The symbol for uninitialised variable `myvar_un_init` is not present since it has been moved to bss section.

2.7.1 Startup code and board support packages (BSPs)

We have always been taught that main() is the first function to be executed in C. Surprise! This is not true. There is some startup code that gets executed before the execution reaches function main(). The steps are usually standard across platforms (obviously, their implementation will vary). So, the compiler vendors usually provide appropriate startup code with their compiler toolset for the particular platform. This

startup code gets executed before the main() is reached. A typical flowchart for the startup code is given in Fig. 2.5.

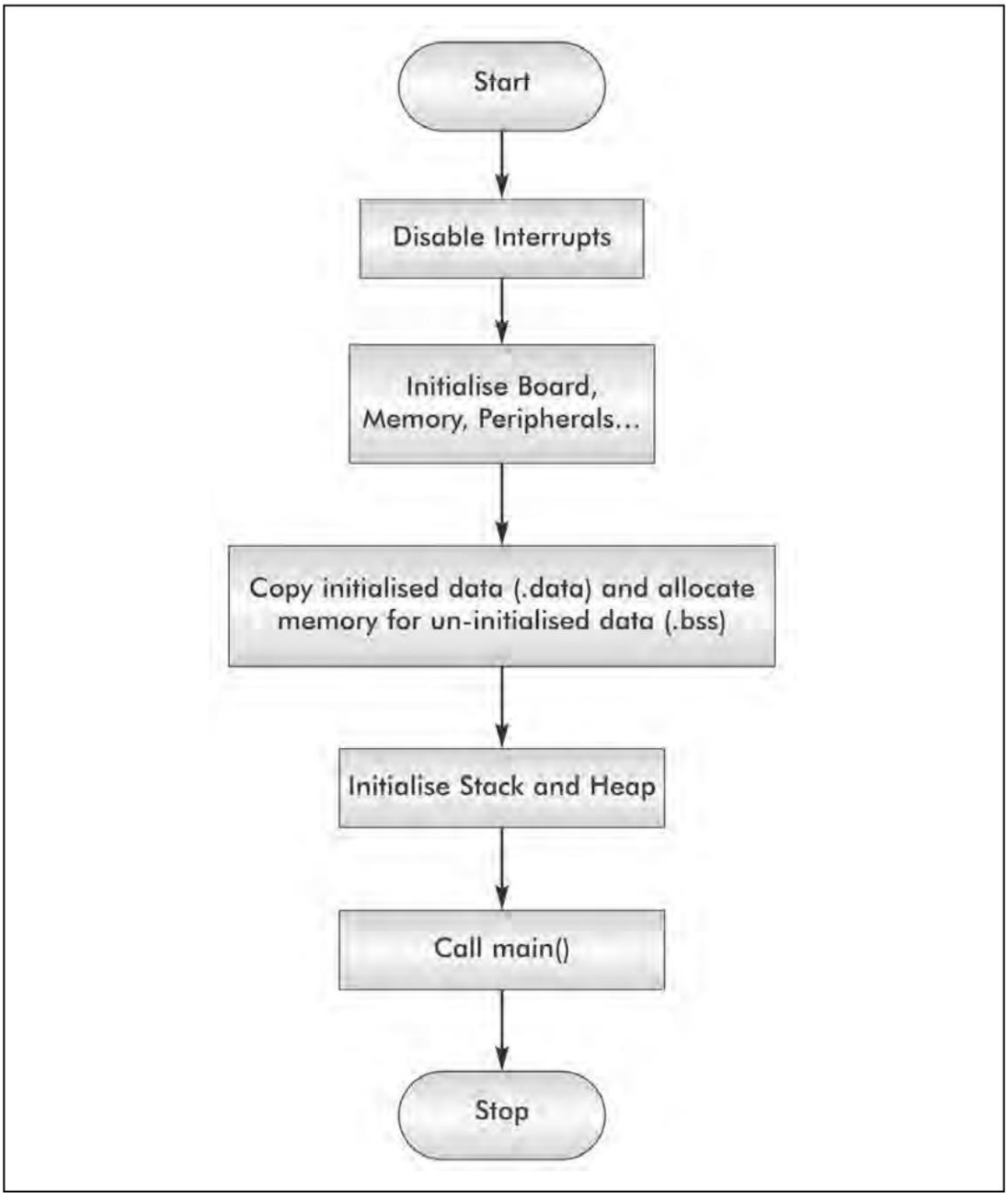


Fig. 2.5 ***Startup code — steps***

We know very well that all global/static variables are initialised to zero. If in doubt, execute the following program and check it out!)

```
#include <stdio.h>

int a;

int main(void)
{
     int b;
     static int c;

     printf ("%d\t%d\t%d\n", a, b, c);

     return 0;
}
```

Listing 2.3: *glob_val.c*

The output of the program when compiled and run in our PC was:

```
0       575       0
```

The 575 is a garbage value and can change based on the memory location the variable finds itself. The point to be taken is that the variables a, c are initialised to zero even before main is entered. This is done during the initialisation of the bss segment. This is sufficient proof that `main()` is not the first function that gets executed.

2.8 LINKER SCRIPTS AND SCATTER LOADING

Now, embedded systems are becoming complex and have various types of RAM. Designers can choose to load different parts of program to different types of RAM (say SRAM, SDRAM) during the execution of the program to meet performance and cost requirements.

For this we need to instruct the linker to place various modules in various locations in the memory. So, we need to create linker scripts.

A sample linker script is given below:

```
ROM 0x08000000 0x2000 ; "Region" "Base Address" "Maximum Size"
{
  ROM 0x08000000 ; "Region" "Base Address
  {
        foo.o (+RO)
  }
  RAM 0x0600000 ; "Region" "Base Address"
  {
        foo.o ( +RW )
        foo.o ( +ZI )
  }
}
```

Here, we are placing the read-only content (.text section in ROM and the rest in RAM) during execution. RO, RW are Read Only and Read Write respectively. The ZI is called the Zero Init section, which corresponds to BSS.

In the startup script, the memory for ZI section must be allocated and the contents must be zeroed.

The syntax of these files are linker specific. The format given above is based on ARM™ linker (provided with ARM developer suite). 'ld' of the GNU GCC compiler collection also supports linker scripts.

2.9 LOADING ON THE TARGET

In order to finish the discussion related to compilation, let us briefly touch upon how this code is executed. We will take this process further when we describe memory types and we will explain available tools in Chapter 11.

As we mentioned before, the typical process of code development for embedded systems happens on a host machine. This host machine may have an integrated development environment and a cross compiler in order to produce machine code understandable by the embedded hardware. Figure 2.6 illustrates this concept. The developer works on the host system and "loads" the machine code on the target hardware memory (usually Flash or some kind of ROM—we will discuss this in Chapter 3) using a serial link or Ethernet connection.

P2P

Why are host and target called so?

As we will see in Chapter 11, Fig. 2.6 is a simplified version of the actual setup; usually there are more tools available to help the developer.

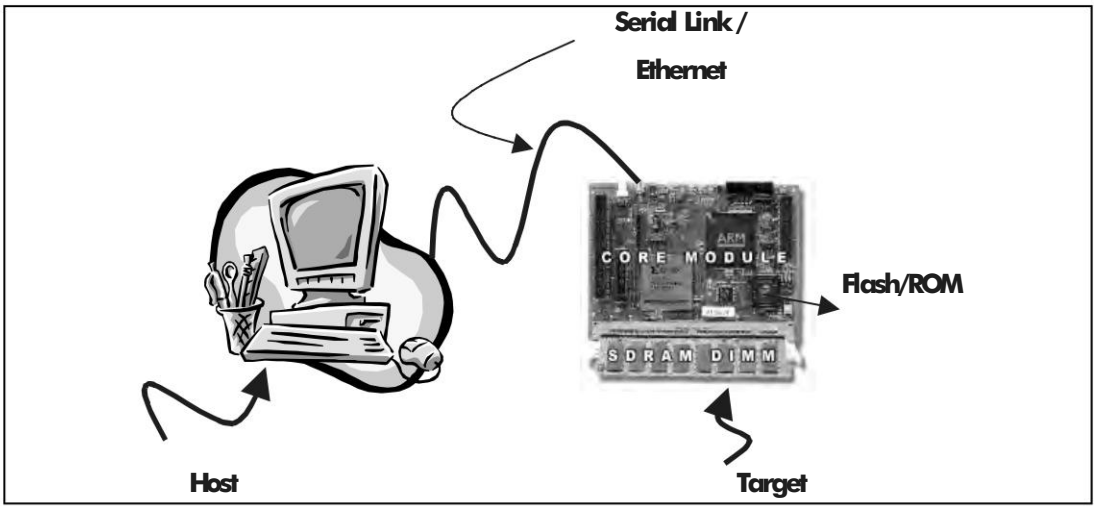


Fig. 2.6 *Development of embedded platforms*

2.10 PROGRAM SEGMENTS

In compiling parlance, a segment is usually a collection of related/similar data.

When we ask a programmer to define a program, the usual reply is that a program is a collection of code and data. So, based on this, every program has the following segments: (some of the segments can be empty based on the nature of the program)

i. data
ii. bss/zeroinit
iii. stack
iv. text/code

The above four are common segments found in most of the programs. Though others (e.g. TLS—thread local storage) also exist, we will be restricting our discussion to the above four only.

2.10.1 Text/Code segment

Consider the following program:

```
int i = 1; // Initialized global integer variable
int j;     // Un-initialized global integer variable

int a[100];// Un-Initialized global integer array
int b[10] = { 0, 1, 2 }; // Partially initialize array

int main (void)
{
    int x;           // local variable
    int y[50];       // same

    static int z;    // static variable with local scope

    for (i = 0; i < 50; i++)
        y[i] = i;

    return 0;
}
```

Listing 2.4: *Program segments*

The above program does nothing useful, but is taken for demonstration of various program segments. This program exists in the text segment. The program has some data (the variables) and some code (the for loop) that operates on the data.

The data can be classified as:

i. Global Variables
   - Initialised
   - Uninitialised

ii. Local Variables

Note

Global variables, global variables with file scope (global variables defined with static qualifier) and static variables defined inside a function scope are variables that are supposed to have a static linkage. Usually programmers get carried over by this static linkage notation. All the above three kinds of variables have space allocated at the compile/link time. So, their addresses are a constant, i.e. static throughout the lifetime of the program. This is unlike local variables that get created on the stack and have varying addresses for every invocation. (In fact, to enable recursion, we require each instance of the local variable of the same function to have different addresses).

2.10.2 Data segment

This consists of global* variables that have been initialised to some value. The space for these initialised variables is assigned at compile time. So, having more initialised global variables leads to an increase in size of the final executable. This is not usually desirable.

We all know that if we have an array (global or local) with initialisers less than that of the size of the array, then the rest of the elements are initialised to zero. For e.g., consider

```
int a[100] = {0, 1, 2};
```

This means that `a[0] = 0, a[1] = 1, a[2] = 2` and the rest `a[3]` to `a[99]` are initialised to 0.

But, the danger of above initialisation is that, for the sake of initialising 3 members of the array, we have added the space needed for 100 integers to the size of the final executable. If the integer size in your machine is 4 bytes, then, the code size increases by 400 bytes. That is quite a lot of memory in embedded systems.

Sometimes, global variables are initialised to zero for the sake of clarity.

```
int i = 0;
int j = 0;

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

Listing 2.5: *Global variables initialised to zero*

But, this initialisation to zero is superfluous and is not required. But, as a result of initialisation, the final code size will increase by a few bytes. Uninitialised global variables are automatically initialised to zero. But, the main objective of this section is to bring home the point that initialised global variables are part of the data segment (that take space in the final executable).

*Henceforth, in this chapter, whenever a reference is made to global variables, it is also applicable to static variables unless explicitly mentioned otherwise.

2.10.3 BSS segment

This is the place for uninitialised global variables. Adding uninitialised global variables does not increase the code size. A note of the size of the bss segment is kept in the executable. During the loading of the executable, the size for the bss segment is allocated at runtime. Then, the memory allocated for the BSS segment is zeroed. (Now we know how global variables are initialised to zero).

Tips

If you have global integer/array variables initialized to zero, then, you may very well remove the initializers to save object code size.

Consider the following code listings and their corresponding sizes:

```
// bssarray.c
#include <stdio.h>

int a[100]; // This aray will be placed in BSS

int main (void)
{
 printf ("This program has a global array in bss segment\n");
 return 0;
}
```

```
//dataarray.c
#include <stdio.h>

int a[100] = {0, 1, 2}; // This aray will be placed in data
segment

int main (void)
{
 printf ("This program has a global array in bss segment\n");
 return 0;
}
```

In our system, when we compiled the above programs with the Borland compiler, we got the following output:

```
C:\Sriram\code>dir *.obj

 Volume in drive C has no label
 Volume Serial Number is 1380-0FE3
 Directory of C:\Sriram\code

BSSARRAY OBJ             540  01-01-03 11:01a bssarray.obj
DATAAR~1 OBJ             926  01-01-03 11:01a dataarray.obj
         2 file(s)          1,466 bytes
         0 dir(s)       3,724.56 MB free
```

Fig. 2.7: *Size of object codes (Borland)*

We can see that the size varies approximately by 400 bytes (size of 100 integers in our machine). (The same code when compiled with Microsoft cl compiler produced the following output. Here, the difference is exactly 400 bytes as expected. This is not a benchmark test to compare two compilers. We just want to show that different compilers could produce different outputs and that around 400 bytes get added to the code size when we try to have an initialised array instead of uninitialised one).

```
C:\Sriram\code>dir *.obj

 Volume in drive C has no label
 Volume Serial Number is 1380-0FE3
 Directory of C:\Sriram\code

BSSARRAY OBJ             503  01-01-03 11:13a bssarray.obj
DATAAR~1 OBJ             903  01-01-03 11:13a dataarray.obj
         2 file(s)          1,406 bytes
         0 dir(s)       3,723.29 MB free
```

Fig. 2.8: *Size of object codes (Microsoft)*

The standards (C(C9X) and C++ (ISO 14882)) say that the BSS segments be filled with zeroes. But, the numbers may not have a logical value zero and can have their own representation. (For e.g., for a global floating point number, if all its bytes are initialised to zero, then it can take a value decided by the compiler implementers. But, in most cases, we can assume that the value is zero)

2.10.4 Stack segment

The stack segment provides the space for data when the program is executing. So, the stack segment does not add space to the size of the executable.

The stack segment is required only when the program is running. The stack is used to implement function-calling mechanism in C/C++ (and a lot other languages too).

When a function is called, the basic steps that can be observed are:

i. The function call is made and the arguments are passed to the called function.

ii. The function uses the variables local to the function to perform its desired operation.

iii. The function returns (with or without a return value) to the function caller.

It should be observed that the function may in turn call other functions or may call itself.* If we observe carefully, memory is required for all the three cases:

Step	Memory required for
Function call	Space for arguments
Computation	Local variables
Return	Save the context of the caller function and returned values.

Table 2.1: ***Memory requirements for a function call***

These steps are explained below.

Function call

Usually a function is called with some arguments. The function performs some operation based on the arguments and may return a value to the function that called it. The arguments that need to be passed to the function occupy memory.

Note

The class of functions that do not take any argument, but return a value are called 'generators' *or* 'sources'. *E.g. rand() in math library that returns a random value. The classes of functions that take input but do not return any value are known as* 'sinks'. *Sinks are not to be confused with procedures that used to perform a specific operation based on its arguments.*
While designing systems we must be careful in avoiding sources and sinks, since they are not natural in systems.

*In which case it is called '*recursive*'.

Computation

A function typically uses local variables to store intermediate values. These local variables are created during runtime and require memory.

Return to the caller

In order to return to the caller, the original context must have been saved. Simply put, we must know where to return after the function completes.*[1] And, if the function is to return a value to the caller, memory will be required for this also.

2.11 UNDER THE HOOD — FUNCTION CALLING MECHANISM

The mechanism that takes care of these dynamic data structures used to implement these stack structure actually form part of C-Runtime.*[2] The data structure that is used to handle the above memory requirements is called a stack frame.

Consider the following code:

```
// ...
int main()
{
    foo(a)
    // ...
}

int foo (long i)
{
    // ...
     foobar();
}
```

Listing 2.6: ***Function invocation***

The main() calls foo()which in turn calls foobar(). Considering that no other functions are called in between, the stack would look like this:

*[1] Discussed elaborately in the Interrupts chapter.

*[2] This is another huge topic that deserves a lot of detail. Beyond scope of this book.

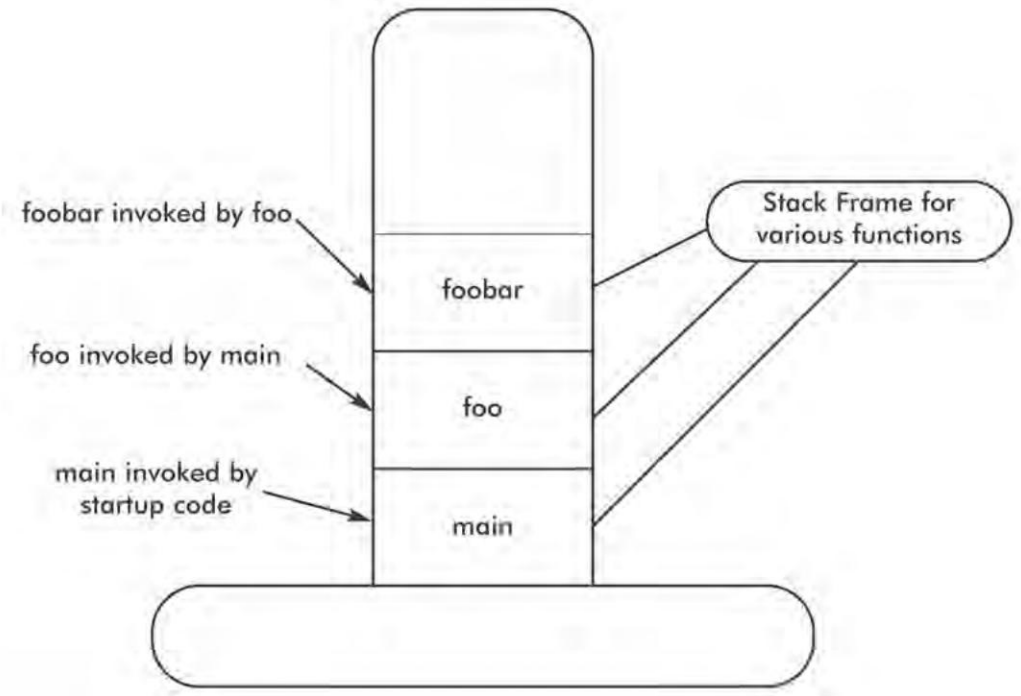


Fig. 2.9 ***Stack frames when a program is running***

A schematic representation of a single stack frame in the above diagram could be as in the following figure.

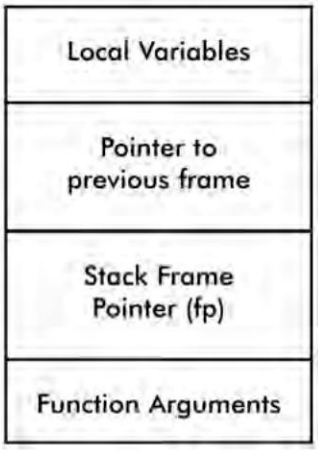


Fig. 2.10 ***A typical stack frame***


Note

This is just a sample representation of stack frame. For the exact structure (in Unix), we can refer to frame.h file.

We can now see that both passing of arguments and the return value take space. So, whenever a big structure needs to be passed to a function, it is advisable to pass the pointer to the function rather than the structure itself directly because, it takes a lot of space in the stack. Moreover, time is also spent in copying the big structure into the memory allocated for the local variable. Similarly, while returning a structure, it is preferable to pass back a pointer rather than the entire structure. We just have to make sure that we don't pass the pointer of structure created in local memory (since that will create a dangling pointer).


Tips

Whenever you pass a structure/class as an argument or return a structure from a function, it is always preferable to pass their pointers (or references (in C++)) rather than the actual structures themselves. This will save a lot of stack space.


Note

The order in which the arguments are pushed into the stack gives rise to an interesting situation. Consider the following function call:

```
foo(a, b, c); // Let a, b, c be 3 integers – 4 bytes each
```

Assembly language code for pushing the arguments into the stack can be:

```
push a into stack
push b into stack
push c into stack
```

So, at the called side (i.e. inside foo), the arguments are got back as

```
pop c from stack
pop b from stack
pop a from stack
```

Note that these are in reverse order of the arguments that are pushed because stack is a LIFO (last in first out) structure.

So far, it seems good. But now, we can question ourselves, "*Why push 'a' in the stack first? Why not c?*"

The answer is: either way is OK so long as both the caller and callee agree to the convention. The two types of conventions are called 'Pascal' calling convention and 'C' calling convention, respectively.

In the pascal calling convention, the leftmost argument is pushed first (in this case, a) and the rightmost last (c, in this case). So, c is the first argument popped by the callee.

In C, the rightmost argument is pushed first and the leftmost (i.e., the first argument) last into the stack. This enables C to handle function with variable number of arguments (The classic example being printf).

The first argument can provide data on the arguments that follow it. In the case of classic printf, the first argument tells about the size of the arguments that follow (%d => integer – 4 bytes (in 32 bit machines), %c => character etc – 1 byte).

Consider

```
printf ("%d %c", a, c);
printf ("%c %c %c %c %c", x, y, z, t);
```

In both cases, 5 bytes are pushed into the stack. These five bytes can be seen as 5 characters, an integer and a character, two short integers and a character and so on. To identify the exact arguments passed, the printf takes help from the first argument which is standardized as a string. For e.g., in the format string, if printf sees a %d, it infers that the data in the byte stream is an integer and pops 4 bytes. If it sees a %c, it infers that the byte in the stack is a character and pops only one byte. This is possible because, the first argument—format string is pushed last. So, functions with C calling conventions can have variable number of arguments while those with pascal calling conventions cannot.

Usually, it is possible for the programmer to control the way the arguments are pushed by using compiler specific options.

```
int __cdecl foo(int a, float b);
int pascal foobar (char a, int b);
```

Windows programmers would have been familiar with int pascal WinMain(). In Intel architectures, it was found that functions that use pascal conventions took less space. So, windows programmers use the PASCAL prefix to function that do not use variable number of arguments.

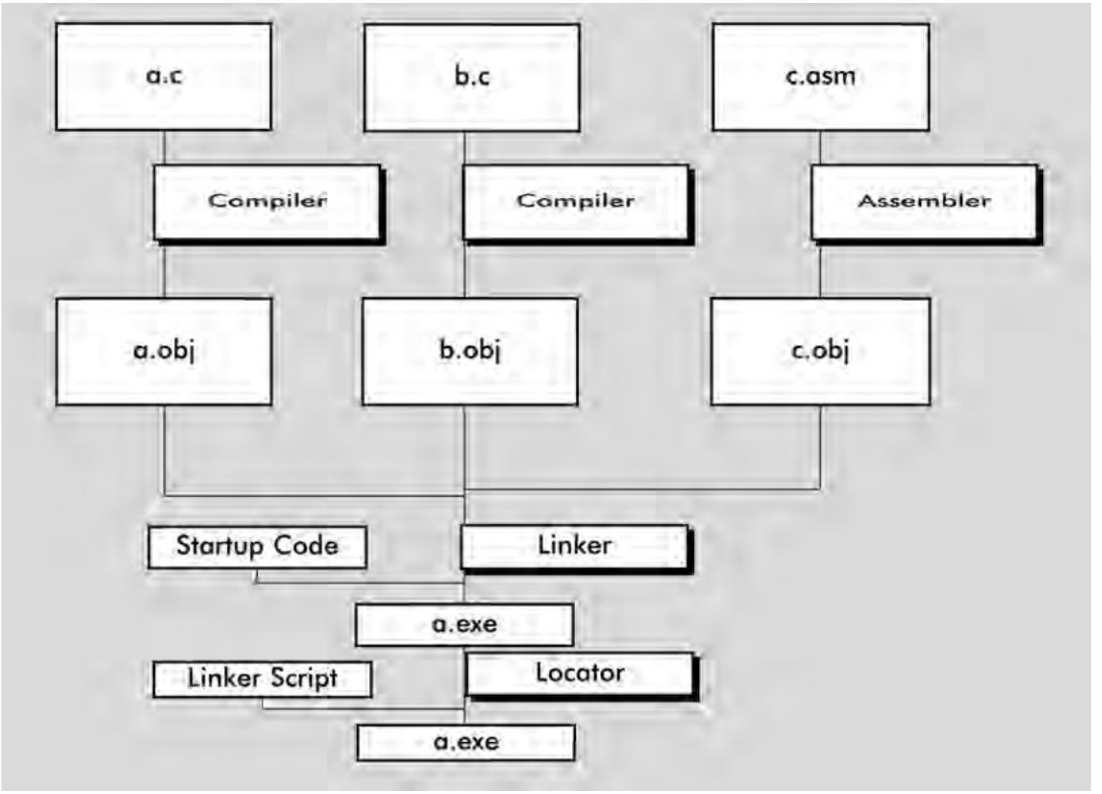


Fig. 2.11 *Build process overview*

Answers to P2P Questions

1. Can we include .c (source) files?

Ans. For the preprocessor it makes no difference if the file to be included is a C file, or a CPP file or a header file or even a fortran source file. But, it is a bad practice to include .c files. This is because, if the resulting file (after inclusion of the .c file) is compiled and linked in a project that also includes the original included file, then link errors will occur because, the global variables and the functions will be defined multiply. To avoid these errors, it is always better not to include C source files. But, the answer to this question is "Yes. You can include C source files". However, it is not the proper mechanism to build a project using multiple files. The preprocessor must be used only to include header files.

2. Can an object file be created for a header file?

Ans. Yes and no. Some compiler drivers will not let you compile a header file. But it is possible to compile them using some compilers. But, even if an object file is created, it

will not contain any data. Let us see some components of a header file and see if we can generate any code for them:

- ***Symbolic Constants*** These are #defines. The symbolic constants are replaced in the code wherever they are found. But, their actual definition does not cause the compiler to generate code.

- ***Preprocessor Macros*** These are #define macros like `"#define SQR(x) (x)*(x)"`. Again, these are expanded only in the places where they are called and hence, these are simply replacement texts. So, these too do not generate code.

- ***Typedefs*** Typedefs are compiler's way to know about a new user defined type. For e.g., we can have a typedef like `"typedef unsigned long ULONG"`. These are used by the compiler to know about the type of variables used. These do not cause any computation overload. So, these too, do not generate code.

- ***Inline functions*** Inline functions are a feature provided by C and C++ that helps us avoid preprocessor macros, providing us the same level of performance benefits with much higher type and call safety. Usually on seeing functions, we will expect that these will lead to generation of code. But, inline functions are expanded directly at the place of their call. So, unless called, inline functions cannot generate code. However, making a function inline is just a request to the compiler. The compiler is free to ignore the request if the function is huge or if the function has loops, etc. So, if the function is not treated inline, the function may generate some code. But, beware! This will cause the same problems indicated in including .c files.

- ***Extern variables*** A header file may also contain extern references to the global variables used by a .c file, defined in other modules. These are instructions to the compiler not to produce "variable not defined" errors.

3. Why are host and target called so?

Ans. Host is called so because it provides the facility to develop, store and manage the source code usually in a high level language. Target is called so because it is the final platform on which the (cross!) compiled code executes.

2.12 LESSONS LEARNT

The build process converts a human understandable code to a machine executable format. The build process consists of preprocessing, compiling, linking and locating. Preprocessor strips comments, expands include files and macros. Compilation parses

the pre-processed code and generates object code for a particular file. Embedded systems usually use a special type of compilation called cross compilation since the code is supposed to execute on a target platform different from the host. After compilation, the linker is called to resolve external references and perform binding to real addresses. The final step in the build process is locating, which means giving details of positioning and size of RAM/ROM and segments. The steps of build process are taken care of by a compiler driver that resolves all arguments and ensures invoking of the right component at the right time.

A program consists of four segments: data, bss, stack and text. Data segment stores initialised global variables. Bss segment stores uninitialised global variables. The stack segment stores parameters and data while the program is executing. Its most important job is to keep track of arguments, local variables and return addresses when functions are called. Text segment stores the actual code.

2.13 REVIEW QUESTIONS

- What role does a preprocessor play in the build process for an embedded system?
- What is meant by a compiler driver? What are its functions?
- If a compiler generates object code already, why do we need to call the linker?
- What portion of build process is different in embedded systems and applications? In what way?
- Is it a good practice to initialise all global variables? Why? Why not?
- Where and how are the local variables of a function stored?
- What is meant by a target in the context of an embedded system?
- What is a board support package? What are the typical steps inside a startup code?
- What is a cross compiler?
- Why do we need 'extern' references in a file?
- What is a TU?
- What are the two basic actions taken in the linking step?

- Why do we need board support packages?
- Write a linker script with the following features:
  There are four object files:
  a. main.o
  b. isrs.o
  c. rtos.o
  d. bsp.o

  There are two RAM locations. One at 0×1000000 and another at 0×2000000. All the read write data must be present in 0×2000000. All the code except the one in main.o must also be present in 0×2000000. The rest must be located in 0×1000000.
- What problem can occur if we have:
  a. Large partially initialised global array
  b. Large arrays as local variables
- What could be the problems in using recursion?
- What are the typical contents of a stack frame?
- What are the two famous function calling conventions?

Chapter
3

Types of Memory

Most embedded systems use memories in various ways. Memory is required inside embedded systems for a variety of reasons: to store the executable code, to load the instructions to be executed as well as the associated data, to save important information that can be changed by the user during sessions. We will introduce these causes of memory usage inside embedded systems in this section. Then we will deal with the techniques used to satisfy these requirements inside embedded systems.

Let us begin this chapter by giving an introduction to the types of memory used by embedded systems. As embedded system engineers, it is always advantageous to peek into this domain so that we can appreciate the usage of different types of memory. We can understand the type of memory, which should be used for a specific activity. Also, this gives us an understanding of the capabilities of these memories and the pros and cons in using a specific type of memory. We will discuss about ROM and its kinds, RAM and its types, as well as the FLASH memory. We will necessarily keep this discussion short. The interested reader is advised to consult the exhaustive literature available for memory.

Basically embedded systems require memory to store the following classes of data:

Data related to executable code in machine instruction format: This is usually burnt* while the device is being manufactured. This kind of data is typically not changed during the lifetime of the device. This kind of data requires a write-protected or read only memory—that is, once filled, this memory will not be changed during the lifetime of the product.

*We will soon see how it is "burnt".

Data storing the current context of execution: This data is usually the variables being used by programs and their stacks. This memory is very volatile. Since the variables and stacks make sense only when a program is executing, it is expected that the contents of this memory can be lost when power is turned off. However, it is also expected that this kind of memory is fast to access for reading as well as writing because the realtime behaviour of a device will also be governed by this factor. This type of memory should be fast, erasable and volatile—or in technical jargon, random access memory. We will see later that this name is a misnomer by the way. Embedded systems use different types of random access memory based on amount of data used, cost of device and requirements on speed.

P2P
Does access to RAM need to be deterministic?

Configuration data: This data relates to configuration of the device. For example, in DECT* phones we can store the phonebook in the form of name-telephone number pairs. Now this phonebook is expected to remain intact if the phone is switched off and back on again. However, it is also expected that entries can be added, deleted and changed over and over again. This kind of memory should be capable of being altered. It should not be volatile so that the data is not lost at switch off. It is a sort of mixed type of the earlier memories. This memory does not have very stringent requirements on speed though. So this memory should be non-volatile and changeable.

3.1 MEMORY ACCESS PROCEDURE

A memory access is the procedure used to read and write into memory. This procedure used to control each access to memory involves the memory controller to generate correct signals to specify which memory location needs to be accessed. This is done based on the access of data by the program. The data shows up on the data bus connected to the processor or any other device that requested it.

Memory chips are organised as rows and columns of data. For example, a 16MB chip can be accessed as a 4M×4 block. This means that there are 4M (4,194,304) addresses with 4 bit each; so there are 4,194,304 different memory locations—sometimes called cells—each of which contains 4 bits of data. 4,194,304 is equal to 2^22, which means 22 bits are required to uniquely address that number of memory locations. Thus, in theory 22 address lines are required.

*DECT—Digitally Enhanced Cordless Telephone.

However, in practice, memory chips do not have these many address lines. They are instead logically organised as a "square" of rows and columns—sometimes called bitlines and wordlines respectively. The low-order 11 bits are considered the "row" and the high-order 11 bits the "column". First the row address is sent to the chip, and then the column address. For example, let's suppose that we want to access memory location 3,780,514 in this chip. This corresponds to a binary address of "11100110101111110100010". First, "11110100010" would be sent to select the "row", and then "11100110101" would be sent to select the column. This combination selects the unique location of memory address 3,780,514. The active row and column then sends its data out over the data bus by the data interface.

Figure 3.1 shows at a conceptual level, an example of memory access with an 8*8 row and column grid. Note that the grid does not have to be square, and in fact in real life it's usually a rectangle where the number of rows is less than the number of columns.

P2P
What is the advantage of using a rectangular address grid?

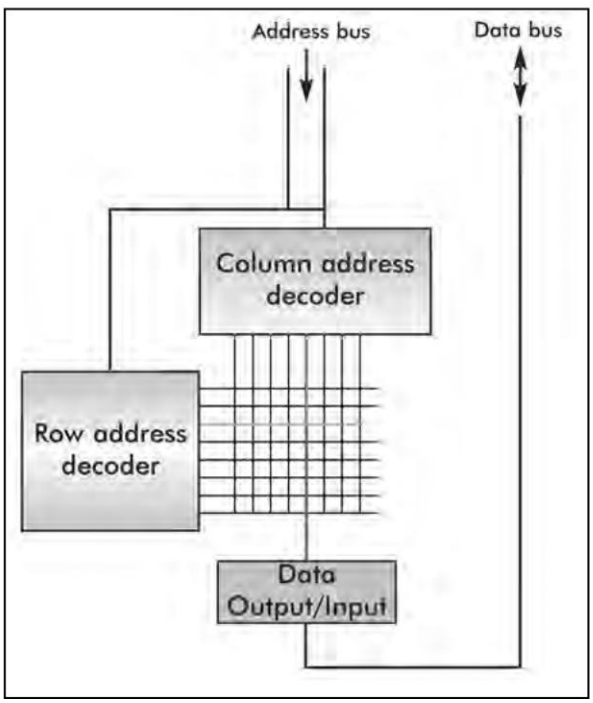


Fig. 3.1 *Memory access*

This is analogous to how a particular cell on a spreadsheet is selected and set: row #34, say, and then look at column "J" to find cell "J34". Similarly, for example, how do chess connoisseurs track the moves made by Vishwanathan Anand yesterday in New York? Elementary, my dear Watson, just label the chessboard by rows 1 to 8, and column a to g. Now, all moves can be represented by a series of digit-alphabet combinations.

Let us now get back to the world of memory chips ☺. If we apply common sense to this theory, we can argue that designing memory chips in this manner is both more complex and slower than just putting one address pin on the chip for each address line required to uniquely address the chip—why not just put 22 address pins on the chip? The answer may not surprise many people: it is all about cost finally. Especially when so many embedded systems do not have hard real time constraints, we can still live by with a few memory-access delays if it makes the system simpler and cheaper. By using the row/column method, it is possible to greatly reduce the number of pins on the DRAM chip (We will explain how this "D" came before RAM very soon ☺). Here, 11 address pins are required instead of 22. However it should be noted that additional signaling is required so that the memory chip and the accessing device are always synchronised about what they are expecting. This signaling pin is usually called probe or chip select. One thing is for sure: everything else remaining constant, having to send the address in two "chunks" slows down the addressing process, but by keeping the chip smaller and with fewer inputs we gain in terms of power consumption and space (because of reduction in number of pins). The reduction in power consumption further leads to an increase in the speed of the chip, partially offsetting the loss in access speed. Figure 3.2 shows a typical memory chip with 8 address lines and two data lines.

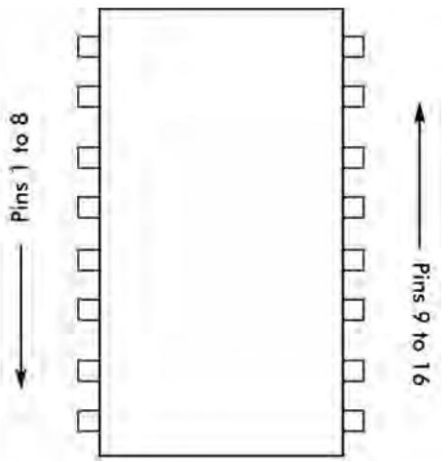


Fig. 3.2 *A memory chip*

Let us try to understand the different parts of this memory chip with the aid of Table 3.1.

Pin number	Name	Description
1 to 5	A0 to A4	Address lines
6	WE	Write Enable
7	CS	Chip Select
8	Dout	DATA Out
9	GND	Ground
10	Din	DATA IN
11	Vcc	Power Source
12	RAS	Row Address Select
13	CAS	Column Address Select
14 to 16	A5 to A7	Address lines

Table 3.1: ***Explanation of Fig. 3.2***

The functions of various pins are described as following:

- ***Vcc*** This is the pin through which the chip receives power for functioning.
- ***GND*** This is the Ground pin required for any electric circuit.
- ***WE*** Write Enable. When this pin is asserted, this means that the chip has been selected for writing. This also means that the data will be sent through the Din pin and pins A0 to A7 specify the location of this data. When the WE pin is low, this means that the chip has been selected for reading.* In this case, the location of the data is again specified through address pins A0 to A7. The data flows out through the Dout pin.
- ***RAS and CAS*** As we noted before, the rows and columns inside the memory area are addressed one at a time. Hence there must be a way of telling the chip that at this moment, the address lines specify a row number or a column number. When RAS is set high, a row address is specified, otherwise a column address is assumed. As we mentioned before, these are the additional pins required for synchronisation in order to reduce the number of pins for address bus.

*This is not entirely true though. Some chips have other mechanisms to distinguish between a read and a write operation. However for simplicity reasons, and to make the concept clear, let us reuse the same pin for our illustration.

With the aid of Figs. 3.3 and 3.4 respectively, let us trace the steps for read and write operations through this chip.

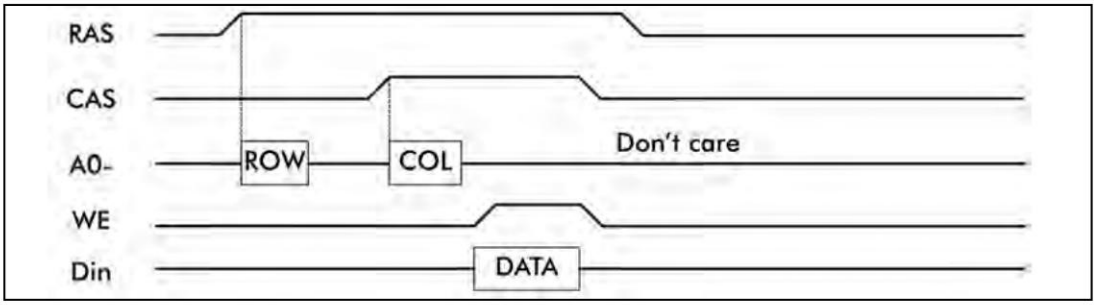


Fig. 3.3 *Write operation in a memory chip*

For writing:

i. *The address of the cell to be written to is placed on the address pins via the address bus*: This is done by first setting the RAS and putting appropriate row number followed by setting the CAS and putting appropriate column number on the address bus.

ii. *Set the Write Enable pin*: The bit that needs to be stored in the chip is sent on the Data In pin via the data bus.

iii. *Chip select is activated to select the memory chip*: When all these operations are performed simultaneously, a bit on Din pin is written inside the chip at the address specified by the address bus.

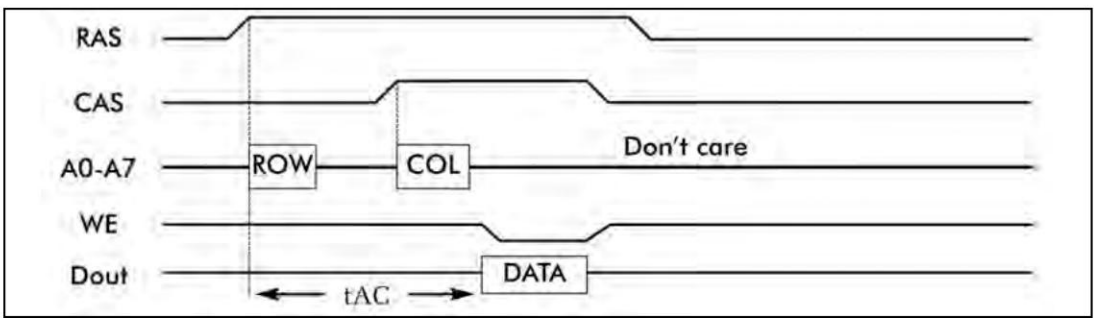


Fig. 3.4 *Read operation in a memory chip*

For reading:

i. The address of the bit to be read is put on the address pins via the address bus. RAS and CAS pins are used appropriately.

ii. Chip select is turned high to select the memory chip.

iii. Write Enable pin is turned low so that the chip knows it's being read from.

iv. When all these conditions are active simultaneously, data appears on the Dout pin from the address specified by the address bus.

In actual practice, memory is accessed at least a byte at a time, and not a bit at a time. This is accomplished by stacking each such chip into blocks of eight and combining the bit-data streams from these eight chips. When these chips need to be addressed, the Chip select is enabled on all of them, and the same address is specified on all address lines. Depending on the capacity of the data bus, each such block can again be stacked to make mega-blocks that can service data in multiples of a byte.

Definition
The amount of time that it takes for the memory to produce the data required, from the start of the access until when the valid data is available for use, is called the memory's access time, abbreviated tAC (See Fig. 3.4).

Access time is normally measured in nanoseconds (ns). Memory available today normally has access time ranging from 5 to 70 nanoseconds.

3.2 TYPES OF MEMORY

Figure 3.5 identifies the different types of memory. Let us look at each below.

3.2.1 RAM

Random access memory (RAM) is a read-write memory. RAM is considered "random access" because any memory location can be accessed directly instead of a sequential operation from the beginning of the memory. Being random access does not define this kind of memory completely. It is sufficient to distinguish it from its opposite, **serial access memory** (SAM). SAM stores data as a series of memory cells that can only be accessed sequentially (like a cassette tape). Data is searched from the beginning of the memory until it is found or end-of-memory is reached. SAM works very well for memory buffers, where the data is normally stored in the order in which it will be used (a good example is the texture **buffer** memory on a video card). A random access

memory on the other hand can directly address particular portions of memory. This makes it fast and expensive as compared to SAM.

RAM is the place in embedded systems where usually, the program, its stack and data in current use are kept so that the processor can quickly reach it. At the beginning of execution (switch on of the system), these initial values are loaded inside RAM. RAM is used since it is much faster to read from and write to than its distant cousin: the ROM. However, the data in RAM is volatile and stays there only as long as the system is powered up. When the system is turned off, RAM loses its data. When the system is switched on again, the binary image is again loaded from ROM and all stack and data is again initialised.

Tips

Who loads this program and data inside RAM? It is the job of your program to know the amount of data and its required space inside RAM.

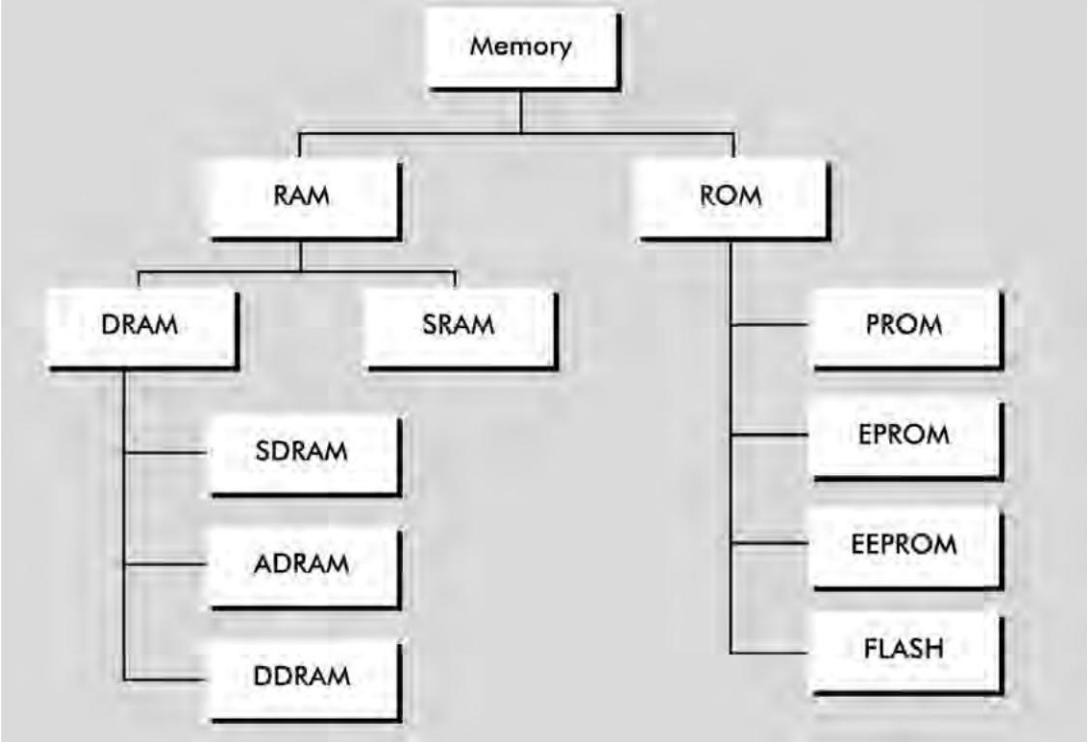


Fig. 3.5 *Types of memory*

DRAM

DRAM or the Dynamic RAM is a very volatile memory. It is not able to store information even for a few milliseconds even if power is available to the chip. The storage medium keeps on forgetting the data stored on it. It periodically requires a refresh circuit in order to prevent loss of data. Here we need to address two questions:

Why use a memory that is not able to memorize? ☺ and;
How do we make sure that the DRAM indeed stores the data correctly?

The answer to the first question is 'cost'. DRAM is one of the cheapest forms of memory. Typically, it completely forgets any stored data in 5 or 6 hundredths of a second if a mechanism does not exist to retain it.

The answer to the second question is a subtler one. To understand it, we need to introduce a little bit of techniques about electric charge and capacitors.

DRAM is implemented in the form of a series of microcapacitor-transistor pairs that store charge inside them in order to 'remember' the data stored. The capacitor holds one bit of information—either 1 or 0. The transistor is meant to manage the operation of the capacitor—reading, writing, changing state, etc.

Now, capacitors have a curious property that they cannot store charge for a long time. They tend to get discharged over time unless refreshed by a source of power. In that sense, a capacitor is a sort of leaky bucket with a certain rate of outflow of charge while storing the data. Therefore, we need a source of power and a 'refresh circuit' that keeps on feeding energy in order to prevent the capacitor from discharging. The refresh circuit simply restores a full charge to those bits that were remembering to be 1s. Hence typically several hundred times per second, the capacitors require reminding in order to keep the memory intact.

This reminding process is called 'refreshing' the memory, and DRAMs use a wide variety of sophisticated refresh circuits which do nothing but cruise through the vast DRAM memory array polling each bit for its content before it fully forgets, and retelling the memory what it is supposed to remember. The disadvantage of this type of ephemeral memory is offset by its very tiny size and simple structure. Having a memory cell so small allows us to put more of them on any given device, thus reducing cost considerably.

Figure 3.6 shows a block diagram containing the actual memory array and the associated circuitry to perform the refresh operation. The left part of the figure is the DRAM controller. Now, Reading from or writing to a DRAM cell refreshes its charge,

so the DRAM controller just keeps on reading periodically from each cell. For that reason there is a refresh timer inside the figure. The DRAM controller takes care of scheduling the refreshes and making sure that they don't interfere with regular reads and writes generated by the processor or some other device. DRAM controller periodically sweeps through all of the rows by cycling RAS repeatedly and placing a series of row addresses on the address bus. The upside of a DRAM is that since it is so simple, it is small in size and less expensive. The downside is that all this refreshing takes time and slows down the memory, particularly as compared to its sister—the SRAM.

Note

Earlier in the chapter, we had pointed out that a DRAM chip is usually in the form of a rectangle instead of being a square. Now is the time to explain this fact. Since DRAM uses RAS to periodically sweep through the entire RAM area, this operation will be faster if the number of rows is less because the fewer rows the chip has, the less time it takes to refresh all the rows. Consequently, DRAM makers design DRAMs with fewer rows than columns thus resulting in a rectangular layout.

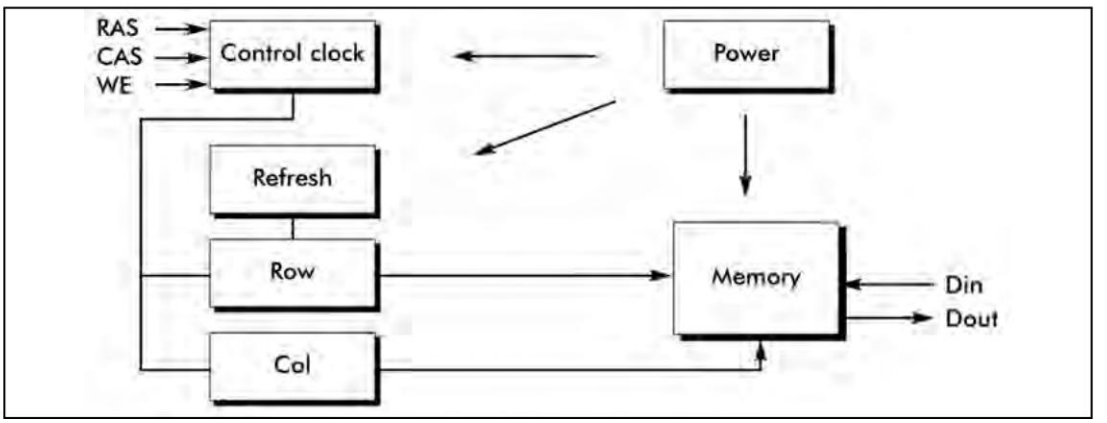


Fig. 3.6 *Block diagram for DRAM*

As DRAMs became more sophisticated, it became common to put this refresh circuit from the system board directly onto the DRAM chip itself. When this is done, from the outside, it appears that the memory is behaving statically since it does not require any refresh circuit from outside. In reality, however, it is still a DRAM since each memory cell is being constantly refreshed on the chip; only the position of the source of refresh operation has changed. When the refresh circuit is integrated with the DRAM chip, the device is called a *Pseudostatic DRAM.*

DRAM is of two kinds: asynchronous and synchronous. An asynchronous DRAM has the freedom to start its operations irrespective of the clock. However, this requires some time for co-ordination between the different pins in order to judge the change of configurations on the pins. SDRAM or the synchronous dynamic RAM is called so because this memory marches in step with the system clock instead of allowing itself the asynchronous freedom to respond at its own pace and on its own schedule. SDRAM "locks" (synchronises) the memory access to the CPU clock. This way we get faster data transfer. While one portion of data is transported to the CPU another may be being prepared for transfer. Additionally, it stays on the row containing the requested bit and moves rapidly through the columns, reading each bit as it goes. The idea is that most of the time, the data asked for from the device will be in consecutive locations inside memory. To understand why, let us take a look at Fig. 3.7, where we show an asynchronous operation of reading two bits.

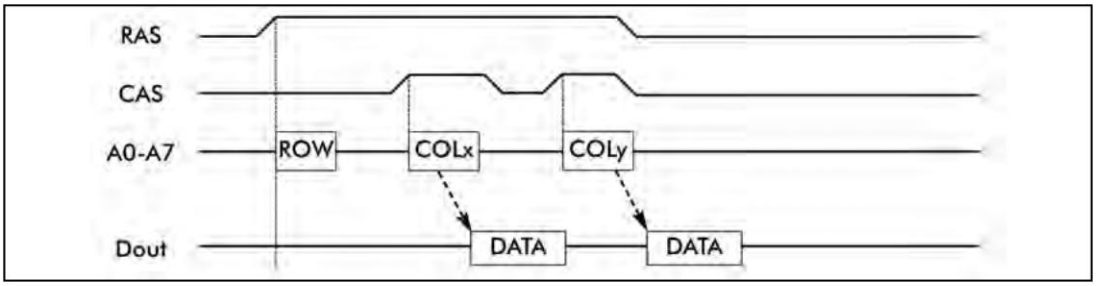


Fig. 3.7: ***Operation of asynchronous DRAM***

For each operation, synchronisation has to be maintained between RAS, CAS, etc. In Fig. 3.8, a corresponding operation for SDRAM has been illustrated. Data starts to be read from contiguous memory locations after the bitline and wordline have been specified. Each clock tick initiates a read operation from the next wordline.

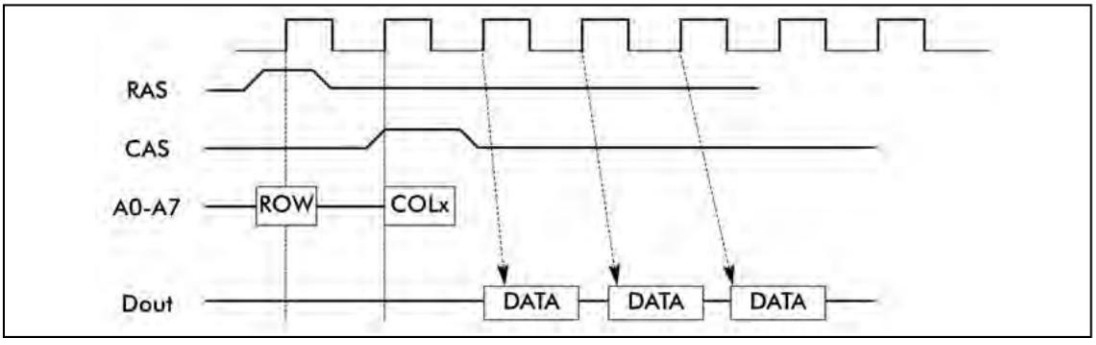


Fig. 3.8: ***Read operation in SDRAM***

SDRAM typically has an access time of only 6–12 ns. Another variant of SDRAM is called DDR RAM or a double density RAM. It is a new technology and is a clock-doubled version of SDRAM, which is replacing SDRAM nowadays.

SRAM

SRAM or static RAM is so called because it retains any information stored in it, as long as power is maintained. The data just sits there, calmly awaiting retrieval by the system command. Upon receiving an order to over-write the data or to provide some data being retained, the SRAM is very fast to respond. That's one of its endearing qualities.

SRAM uses a completely different mechanism for storage of information. An SRAM cell is usually made up of a flip-flop gate which further comprises of about 4 to 6 transistors, arranged in a configuration that traps either a binary 1 or a binary 0 in between them until that value is either written over with a new value or the power goes out. This configuration never needs refreshing unless power is switched off. This makes SRAM much faster in response time than DRAM and very power efficient. SRAM can be made with a rise time as short as 4 ns. However, because it has more cells, each cell of the SRAM takes more space as compared to a DRAM cell. This means that a chip cannot hold as many cells as that of DRAM. This makes SRAM more expensive as compared to DRAM. SRAM is normally used in places that require a very quick response time, like for example cache.

3.2.2 ROM

ROM, or the read only memory, as the name suggests is a memory in which we can only read from. This means that ROM cannot be written again and again. This memory will retain its contents even when the power is switched off. Hence this memory is used to store anything that needs to be used after the system has been switched off and on again. What kind of information can this be? This is usually the actual program that will be executed on the embedded system. Since this memory does not get erased at switch-off, it is also called nonvolatile memory.*

Because of the way it stores information (as we will see soon), ROM is much slower compared to RAM, typically having double the access time of RAM or more. However,

*One term that often confuses people is that RAM is the "opposite" of ROM because RAM is read-write and ROM is read-only memory. Hence, since RAM stands for "random access memory", ROM is not random access. This is not true. ROM is also a random access memory. Inside ROM, any location can be read in any order, it is just not writeable. RAM gets its name because primitive read-write memories introduced in the beginning were sequential, and did not allow random access. The name stays with RAM even though it is no longer relevant. ☺

it is expected that ROM need not be accessed as frequently as RAM, so this limitation can be lived with. This, combined with the fact that ROM is considerably cheaper as compared to RAM per byte, definitely has its own advantage.

Definition

RAM is often used to shadow parameters stored in EEPROM (RAM is mapped to ROM's memory space) to improve performance. This technique is called 'ROM shadowing'.

While the purpose of a ROM is that its contents cannot be changed, there are times when being able to change the contents can be very useful. Sometimes it is desirable that the memory remains read-only for all normal circumstances and it should be possible to over-write it by specially defined processes. For example, in a mobile phone, it will be worthwhile to store a specific type of ringer tone into such memory that cannot be erased when the phone is switched off. However, it should also be possible to update this tone from time to time.

Similarly, the user settings inside a washing machine need to be nonvolatile from the switch off point of view. However, it should be possible to set different setting for different clothes (cotton, wool…) and mode of operation (spin dry, double wash…). Hence, there exist a lot of ROM variants that can be changed under certain circumstances; these can be thought of as "writeable nonvolatile memory". The following sections describe the different types of ROMs with a description of their relative modifiability.

Regular ROM

A regular ROM is constructed from hardwired logic, encoded in the silicon itself, much the way that a processor is. It is designed to perform a specific function and cannot be changed. This is inflexible and so regular ROMs are generally used only for programs that are static (not changing often) and mass-produced. ROM is analogous to a commercial software CD-ROM that can be purchased in a store.

While RAM uses a transistor and capacitor combination to store data, ROM uses an electric diode at the junction of row-column to determine whether a 1 or a 0 has been stored at the location. When the ROM chips are "burned", the cells where a 1 has to be stored have a connected diode. The cells where a 0 has to be stored have an unconnected electric circuit. The diodes in these intersections allow the flow of current in only one direction. Like all diodes, a voltage above the break over voltage (of the order of 600 mVolts) is needed so that the diode passes current. To determine whether a cell

has a 0 or a 1, a voltage above 600 mV is applied to a column while keeping the row grounded. If the diode is connected, the current will be conducted to the ground. Using this method, the status of each cell can be read.

Obviously, since there is physical presence of a diode to indicate a 1 in a cell, this kind of memory cannot be changed and reused. Once the ROM is manufactured, it can only be read. And if there are some bugs in the values, well, unfortunately, the whole chip has to be thrown. This makes the chip design process long and cumbersome. On the upside, ROM chips are very cheap for mass-production, have high reliability over a long duration and they consume very less power.

Programmable ROM (PROM)

The limitation of a regular ROM is that it can be burned only once. PROM is a type of ROM that can be programmed using special equipment; it can be written to, but only once. This is useful for companies that make their own ROMs from software they write, because when they change their code they can create new PROMs without requiring expensive equipment. This is similar to the way a CD-ROM recorder works by 'burning' programs onto blanks once and then providing the facility to read from them many times.

Just like regular ROM, a PROM chip has a grid of rows and columns. In order to make it changeable once, each cell has a small fuse in it, (logically the same as we have in our homes for checking electric malfunction). In normal circumstance when the PROM chip is created, all cells have a value of 1. An electric charge can pass through the fuse. To change the value of a cell to 0, a high voltage is sent to the cell thus breaking the fuse. The electric circuit is left open and a 0 gets created at this position.

This process makes it possible to change the PROM chip, but only once. PROM chips are very fragile but cheap. As discussed above, they are generally used for prototyping data for a regular ROM before committing to the costly fabrication of ROM chip.

Erasable programmable ROM (EPROM)

As the name suggests this kind of ROM is nonvolatile but erasable by a special process. Once again, it is a mesh of rows and columns and this time inside each cell there are two transistors technically called the control gate and floating gate. The two transistors are separated by a thin oxide layer. At an erased state, electricity can pass through the

oxide layer and link the two transistors (thus linking the row and column) and a value of 1 is indicated. A process called *Fowler-Nordheim tunneling* is used to program a value of 0 inside the cell. An electric charge, usually 10 to 13 volts, is applied to the floating gate. This high voltage causes the floating gate to act like an electron gun and the electrons are pushed through the oxide layer where they get trapped. At this time, the two gates are insulated from each other and when the voltage is removed, electric circuit becomes open and a value of 0 is indicated. A cell censor detects the level of charge passing through the floating gate, based on which it indicates a 0 or a 1.

The erasing procedure takes care of returning the electrons back to the floating gate. The erasing process for this kind of ROM is also interesting and worth mentioning. To erase the EPROM, a UV light source of wavelength 253.7 nanometre is shown through a quartz window on top of the EPROM chip. The light source is kept about an inch from the chip resulting in erasing of all cells in the chip. This process is strictly specified in terms of intensity of UV light, duration of the process and positioning of light source, any deviations may result in inappropriate erasing.

Electrically erasable programmable ROM (EEPROM)

The next level of erasability is the EEPROM, which can be erased under software control. This is one of the most flexible types of ROM, and is now commonly used for holding configuration and status data meant to be retained across sessions of the embedded device. Here we are blurring the line a bit between what "read-only" really means, but it should be kept in mind that this rewriting is done maybe once a day or so, compared to real read-write memory (RAM) where rewriting is done often many times per second!

EEPROM uses the same setup as EPROM. However, in EEPROM, it is possible to erase only a portion of the chip or all at once and it is possible to perform this operation by software control. UV light is not required for this process. EEPROM uses electric charge to return the trapped electrons to their original state. No special equipment is required and the chip need not be removed. This electric charge can be targeted at any data point at any time. Operationally, EEPROM chips change data one byte at a time, which makes them versatile but slow.

3.2.3 Flash

Flash memory is similar to EEPROM in design. The difference is that it can be erased and reprogrammed in blocks instead of one byte at a time. In-circuit wiring is used to apply electric charge to an entire chip or to specific sections called blocks, each usually of size 512 bytes. Being light, compact and energy-efficient, typical uses of

FLASH are in CompactFlash, SmartMedia, Memory Stick (most often found in digital cameras), PCMCIA type I and type II memory cards (used as solid-state disks in laptops). The original intended usage of FLASH memory was to replace mass storage devices like disk drives and tapes. Flash memory in the form of a card or stick is very versatile and can be used across devices if a standard file system is used to represent data inside it. This is the concept of so-called linear flash.

There is another kind of FLASH called the ATA flash. An ATA flash memory module interfaces with the rest of the system using the de facto "AT Attachment" standard. The FLASH gives the illusion that it is made up of sectors like on a hard disk and the same APIs can be used as for accessing a disk drive. The main advantages of ATA flash, from the embedded system developer's perspective, are flexibility and interchangeability with hard disks. While linear flash modules aren't 100% interchangeable between devices, ATA flash overcomes this limitation by using a standard AT interface for accessing it. ATA flash can be accessed using an operating system's standard disk access code and the same file system APIs. This aids in cross compatibility.

For example, a memory card inside a digital camera, equipped with flash memory uses a format to store data that is compatible with the way PC Card stores it. Hence, the card can just be inserted into a PC card slot and can be read directly by the computer. Not only does this promote cross compatibility, it aids in debugging as well since the limitation of an embedded system (lack of screen and input device) are easily surmounted.

There are additional advantages. The built-in file system is robust enough to perform some housekeeping tasks. For example, it can detect areas of memory that are defective. It can then forbid access to these regions for read-write purposes. It can have a mechanism by which it can create virtual sectors which point to physical sectors in memory in such a way that read and write accesses to these sectors is evenly spread on the chip thus preventing heavy usage and associated wear of a particular portion of the chip. As expected, everything in this world comes with a price ☺. ATA Flash has so many advantages, but all these features make it more expensive and power-hungry. Because of speed limitations, flash memories incorporate built-in SRAM buffers, duplicating the contents of a block of memory from the flash array for fast access.

3.3 LESSONS LEARNT

Memory is used inside embedded systems in order to store executable code, to load the instructions to be executed together with their data, to store information that can change between sessions (for example user preferences). This gives rise to different

types of memory inside embedded systems. Memory chips are usually arranged in the form of rectangles as rows and columns of data. Typically, memory chips use the same address lines for row and column address and reduce the number of lines significantly. The data is read or written based on the RAS and CAS pins.

There are various classes of memory based on their usage. The random access memory has the ability to be accessed in a non-serial way and loses its contents if the power supply is disconnected. It is of two types: the dynamic RAM needs constant charging typically many times a second to prevent it from forgetting its data. The static RAM does not need constant charging for storing its contents. The Read-Only memory is a form of 'not-easily-erasable' memory. The regular ROM can be used only once and its contents cannot be changed after burning it once. The programmable ROM is capable of being recharged once through electric current. The EPROM uses the Fowler-Nordheim technique to erase its contents any number of times. EEPROM is similar to EPROM in operation except that its contents can be erased through software control through electric charge. Flash is a type of EEPROM that can be reprogrammed in blocks instead of one byte at a time. Flash memory has enhanced cross-compatibility of memory across embedded devices through the usage of a standard AT interface.

3.4 REVIEW QUESTIONS

- What are the different types of memory inside embedded systems? What are they used for?
- Describe the memory access procedure. What is the use of chip select function?
- Why are memory chips usually arranged as rectangles and not squares?
- How is DRAM different from SRAM?
- What are the advantages and disadvantages of FLASH over other memories?
- In order to store user preferences, which memory would you use and why?
- What is meant by an asynchronous RAM?

Chapter

4

Memory Management in Embedded Realtime Systems

Memory is a very precious resource in embedded systems. As we have seen in the chapter 'Introduction to embedded/realtime systems', price is one of the critical factors in designing embedded systems especially while targeting high volume products. We have seen the rapid fall in prices of memory in the desktop systems, we have seen that current desktop configurations have 256 to 512 MB RAM as minimum configuration. But, in embedded systems, even 8Mbit Flash and few MB RAM can be considered a premium. Desktop systems also have the luxury of having huge hard disks, which can be used as supplementary virtual memory. Few embedded systems have hard disks. So, the concept of virtual memory is absent in most of the embedded systems. Since memory is scarce, it should be managed properly.

4.1 MEMORY MANAGEMENT METHODS

Before jumping into memory management schemes for embedded systems, let us see how memory was managed before.

4.1.1 Machine language

In the good old days processors like 8085 were state of art and there was no MMU (Memory Management Unit). Programming was done using machine language, and there was no concept of variables (as we have now). For e.g., if we wanted to store a value, then, we would manually allocate a memory location—say, 0×8000 for it. And whenever that value in that location was required to be changed, then it could be done directly by referencing its memory location. There were no compilers. So machine language translation was done by hand using two pages of lookup tables for assembly of

machine language translations. (Gurus were those people who knew the op-codes for 50–60 assembly instructions)

When variables were absent, there is no point talking about arrays. An array by definition is a "*finite set of homogenous data stored in contiguous memory locations, each location accessed by the array variable and its index/indices*".

With this definition in mind, we used to allocate contiguous memory locations on paper in our memory map and then calculate array locations.

4.1.2 Assembly language and high level languages

Assemblers alleviated some of the pains when compared to machine languages. Then came BASIC and FORTRAN. BASIC was an interpreted language. Initially, BASIC had procedure calls (I remember an instruction called GOSUB. But, it was nowhere similar to a function call. It was similar to a jump statement. But, the address from where the GOSUB call was made need not be remembered). And, in BASIC, there was no need to declare variables and all the code was a single module. However some versions of BASIC supported some level of multi-file programming by a command known as 'CHAIN'. We forcefully abstain ourselves from describing how it worked. (And its merits (?)).

FORTRAN was better and it was a compiler-based language. That is, unlike BASIC, there was no need to carry a BASIC interpreter along to execute a BASIC program. FOTRAN (II) did allow procedure calls and functions, but had no notion of stacks. So, addresses of every local variable and arrays (local to functions) were required to be resolved during the compile/link time. Sizes of arrays were fixed to the maximum value that could be expected at runtime. And because of the absence of stacks, there could be no recursion. If a function would call itself, its return address was lost (since it was returned in the function's return link register). With these disadvantages, it would now seem that it could not have been possible to program with FORTRAN. But, later versions of FORTRAN (77) were highly successful and were used to create large number of mathematical applications.

Then, Ritchie and Kernighan in Bell Labs developed C as a language for programming UNIX. Little did they know that this language with a cryptic name would revolutionise the programming world. Every modern language would have some syntax or concept based on this language. C was followed by modern languages like C++, Java, C#, etc. In the rest of the chapter, we will discuss memory management as relevant in C since it is the most widely used language for programming embedded systems. Some pointers to C++ are also given where relevant.

In C (and C++), there are 3 kinds of allocation of memory to variables as described below.

Static allocation

This consists of global variables and the variables are defined with the 'static' qualifier. The address or the memory from these locations are provided at compile/link time and do not vary when the program is executing.*[1]

Automatic allocation

This kind of allocation is done at runtime for variables defined in every function (or scope). This allocation of memory for local variables is done automatically during the execution of the program (by C runtime*[2]). Hence it is christened as 'automatic' allocation. This is the reason for existence of a keyword called 'auto' (ever heard of it?) in C that is never used.

Keyword 'auto' is a qualifier for a variable defined in a scope local to a function.

In the listing below, memory for the variables a, b and c have been allocated as automatic. In the listing below, the keyword auto is used to explicitly qualify variable c for automatic allocation. Memory for all the variables are allocated on the runtime stack (as explained in the chapter "build process for embedded systems".

```
void foo ( int a)
{
     int b;
     auto int c;

     // . . .
}
```

Dynamic allocation

The third way to allocate memory for variables is to demand some amount of memory from a component called the heap. In the earlier two cases, the compiler is responsible for allocating memory for the variables and, so it is its duty to take care of de-allocating the memory.

---

*[1]Refer to the discussion on .bss, .data in Chapter 2.

*[2]The term C runtime conjures up images of complex runtimes like the ones of Java, .net etc. C runtime is fairly simple and will be present in a file called crt.s (assembly) and linked with our program.

The user can use variables that are statically or automatically allocated without bothering much about memory leaks. He should bother if he uses a large number of initialised static variables since it will cause the code size to increase.

The first two methods of allocation are explained in Chapter 2. This chapter deals with third kind of allocation (dynamic allocation) and the runtime data structures.

C Programmers will be familiar with good old `malloc` and `free` (and the C++ counterparts with `new` and `delete`). However, there is a huge variety of heap management functions available especially for the embedded/realtime space. These heap management functions can be classified as

- Variable Buffer size routines
- Fixed buffer size routines

Before going deep into these functions, we need to know the typical implementations of these memory allocation functions and their effects on the realtime behaviour of our program.

Note

Some of the popular myths in the common programming community are listed below:

i. The size of memory allocated by malloc is exactly equal to the size requested

ii. There is no harm in freeing a memory twice

iii. `Malloc/Free` *request and return memory from the OS*

iv. `Malloc/Free` *do not take much time (i.e. they are pretty fast)*

Hopefully, the discussions below on the various heap management implementations should clear the myths.

We all know about the widely used idiom* for memory allocation in C:

```
T* p = (T*) malloc ( sizeof(T) );
```

Here, T can be any type (A predefined type such as int or user defined type like a struct).

*Programming idioms are constructs (like the one described here) that have been used repeatedly that programmers use them even without consciously thinking about them.

Novice programmers generally feel that malloc allocates space exactly equal to the size of type T. To understand why this doesn't happen mostly, we must understand how malloc works.

Malloc (and other allocation functions) request a block of memory during the program startup. This memory can be used by malloc to provide memory to its callers. This memory is organised by malloc and is termed as '*heap*'.

We should understand that the entire memory of the heap is not available to the program. Some part of the total memory is required to maintain the heap. So, the total memory of the heap can be divided as

- The memory required to maintain the heap
- The memory available for the program

Why do we need memory to maintain the heap? To answer the question let us remind ourselves of how malloc and free are used.

```
T* p = (T*) malloc ( sizeof(T) );
free ( p );
```

We see that we pass the amount of memory required to malloc, but not to free the memory. This information is stored somewhere else in the heap. This space for storage of the size of pointer is said to be used for maintenance of the heap.

The entire available memory with the heap cannot be looked upon in units of one but as chunks of blocks with unit sizes of 16, 32, etc. (usually in powers of 2).

The concepts discussed above are illustrated in the example below:

Consider we have 2K (2 * 1024 = 2048 bytes) of memory and we analyse two cases where the memory is split as 16 and 32 byte blocks. (We should remind ourselves that the entire 2K memory may not be available for the program. But for the sake of making this discussion simpler, we are considering entire memory as available to programs).

Memory can be acquired or released only in the units in which the memory is divided (i.e. 16 or 32) as shown in Fig. 4.1.

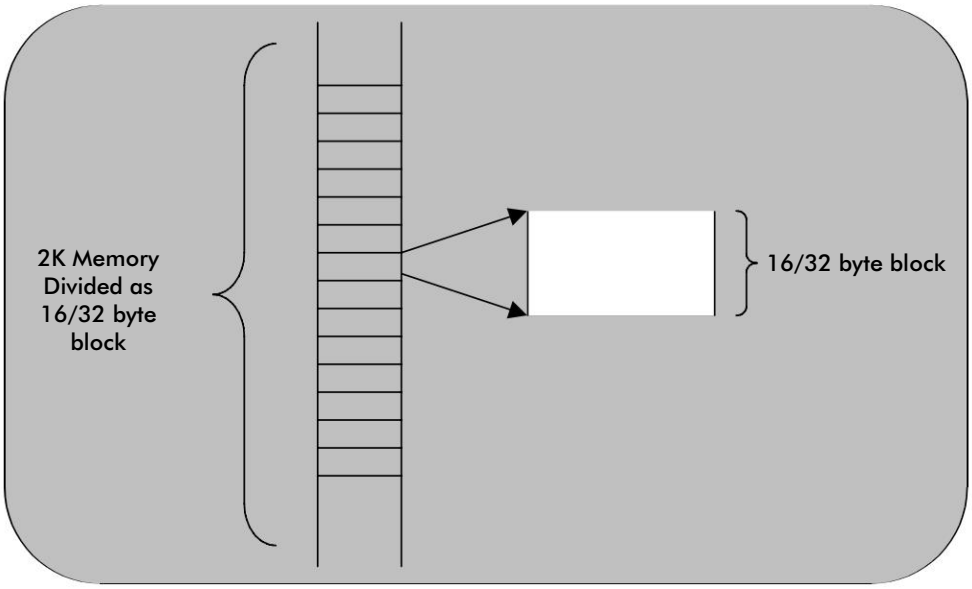


Fig. 4.1 *Memory divided as blocks*

Say, a request for 70 bytes is made, then:

In the case where unit size is 32,

Number of blocks required $= \lceil 70/32 \rceil = 3$.

(Where $y = \lceil x \rceil$ is the ceiling operator, that indicates the smallest integer y that is greater than or equal to x)

So, total memory allocated = 3 * 32 = 96 bytes.

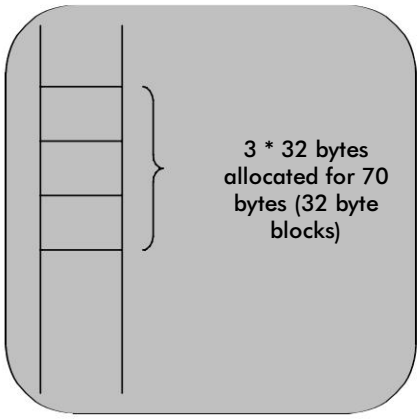


Fig. 4.2 *96 bytes allocated for 70-byte request (32-byte blocks)*

In the case where the unit size is 16,

Number of blocks required = $\lceil 70/16 \rceil = 5$.

So, total memory allocated = 5 * 16 = 80 bytes.

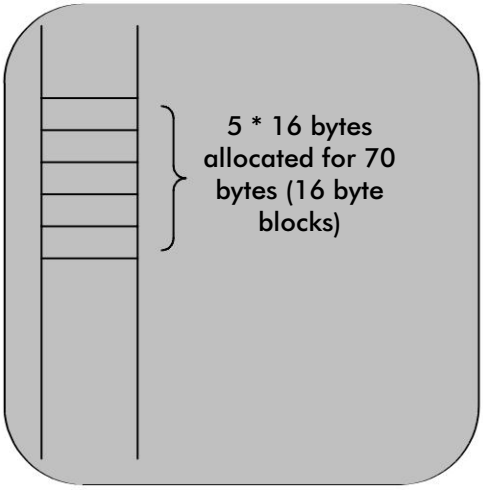


Fig. 4.3 *80 bytes allocated for 70-byte request (16-byte blocks)*

Here we can see that for a 70-byte request, the memory allocated is 96 and 80 respectively for the cases where the unit sizes are 32 and 16. We can infer that by reducing the unit size we can reduce the wastage of memory, the ideal case being unit size = 1. To see why this is not done, we can look up some implementations of heap. In some RTOSes, the beginning few blocks of a memory chunk is reserved for maintaining the data regarding allocation of blocks (Fig. 4.4).

The amount of memory required to maintain the data is inversely proportional to the unit size. If, the unit size is small, then the number of units (total memory/unit size) will be high and hence the memory needed to maintain the data regarding those units becomes high (thus reducing total available memory).

Thus, by decreasing the unit size, we may actually waste more memory than what we may think we are saving. This can be considered as a classical maxima/minima problem in calculus.

Usually in desktop implementation of heap managers usually the problem of determining and assigning unit sizes for heaps does not arise. But, in RTOS, the burden of determining unit sizes rests with the programmer.

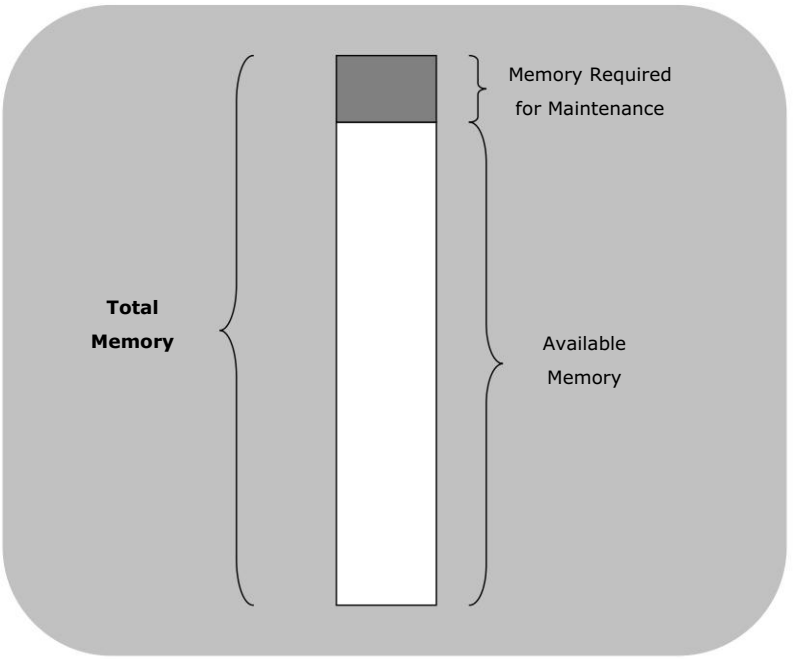


Fig. 4.4 *Memory required for maintenance*

Note

If you need to determine the unit size for the heap, then don't make random guesses to suit the situation. Instead, study and prepare a table of all the structures that are used in the program and that require dynamic memory management. Then choose a unit size based on the table, I know this is tough. But, who said embedded programming was easy? ☺

Now I know that you'll swear that `malloc` will not give you the exact number of bytes you had requested for. But still, before we move to clear the next myth, let us see how memory allocation works.

Usually the heap management keeps a list called the "free list" which is a list of blocks containing memory that is free. These free blocks can be arranged in various orders:

- Decreasing sizes of memory (largest free block first)
- Increasing sizes of memory (Smallest free block first)
- Memory Location (Physically adjacent blocks)

In the first case, `malloc` will work real fast. If memory is requested, it is checked if the request can be satisfied with the memory available in the first block. If available, the requested memory is allocated. Else, there is no point traversing the list because, later blocks will only be smaller. This is called the first fit mechanism.

In the second case, where the smallest block is the first bloc, the heap manager will traverse the list until a block that is large enough to satisfy the request is found. This is called the 'best fit' allocation. It is definitely slower than the first fit approach.

The advantage of the third arrangement (where the blocks are arranged in an order that is physically adjacent) is that adjacent free blocks can be easily identified and merged (or in memory management parlance, '*coalesced*').

Now, we can tackle the next myth: "*There is no harm in freeing a memory twice*". Freeing a memory twice can happen if multiple copies of the pointer are stored in different places. This can happen when the pointer is saved in lists or when passed to functions. Sometimes, it could happen that a pointer is freed in multiple places.

This usually results in unpredictable behaviour. Heap functions have the right to validate the pointer being freed. They have the right to raise software exceptions. This usually happens in desktop applications.

A more dangerous thing can happen in the case where a pointer freed could be allocated for some other purpose. Freeing the memory by an invalid previous reference will free the memory allocated to newly allocated pointer. In this case, the behaviour of the system will be inexplicable. So, freeing a pointer twice is definitely not *safe*.

In Chapter 2, we discussed about the startup code that gets linked up with the main application. One of the tasks of the startup code is to initialise the heap so that the application can start using the heap as soon as it comes up. In desktop based applications, during the startup, a chunk of memory is requested during the startup by the heap manager. Any requests by the application to allocate and free memory is handled by the heap manager using the memory chunk requested during the startup. So, the OS has no role to play in heap management during the execution of the program. (Except in cases where the heap size may be extended by the OS then heap space runs out). This information should clear the third myth—"*Malloc/Free request and return memory to the OS*".

The fourth myth—"*Malloc/Free do not take much time (i.e. they are real fast)*" is a myth that could affect realtime programmers. We have seen that heap is organised as lists and some search occurs whenever memory is requested or even freed (to merge adjacent

blocks). The duration of this search is unpredictable. The problem with this delay is that they make the system non-deterministic. Non-determinism is the greatest enemy in determining realtime responses of a system.

To really understand the realtime issues and why they arise, we have to appreciate two processes that are associated with any heap:

- Heap Fragmentation
- Heap Compaction

Heap fragmentation: As the name indicates, heap fragmentation is a condition where the available memory is scattered throughout the heap in such a way that it is not possible to service a request for memory even if the collectively available memory (scattered throughout the heap) is more than the memory requested. The condition is illustrated in Fig. 4.6.

In the hypothetical condition described above, if a request for 35 K is made, then, it cannot be serviced (even if total available memory is 45K) because the heap is fragmented and no single block that can satisfy the request is available.

Fragmentation can be classified as:

i. Internal fragmentation

ii. External fragmentation

Obviously, both of them lead to wastage of memory.

Internal fragmentation occurs if a huge block of memory is given for even a small request. The two figures (Figs. 4.5 and 4.6) indicate how a request for 40 bytes is satisfied in cases where unit sizes of a heap are 16 and 32 respectively.

In the first case, 2 blocks are required and 64 bytes are allocated and hence 24 bytes are wasted. In the second case, 3 blocks are allocated. So, 8 bytes are wasted (48–40). This memory is not available for use and is wasted. This is called internal fragmentation.

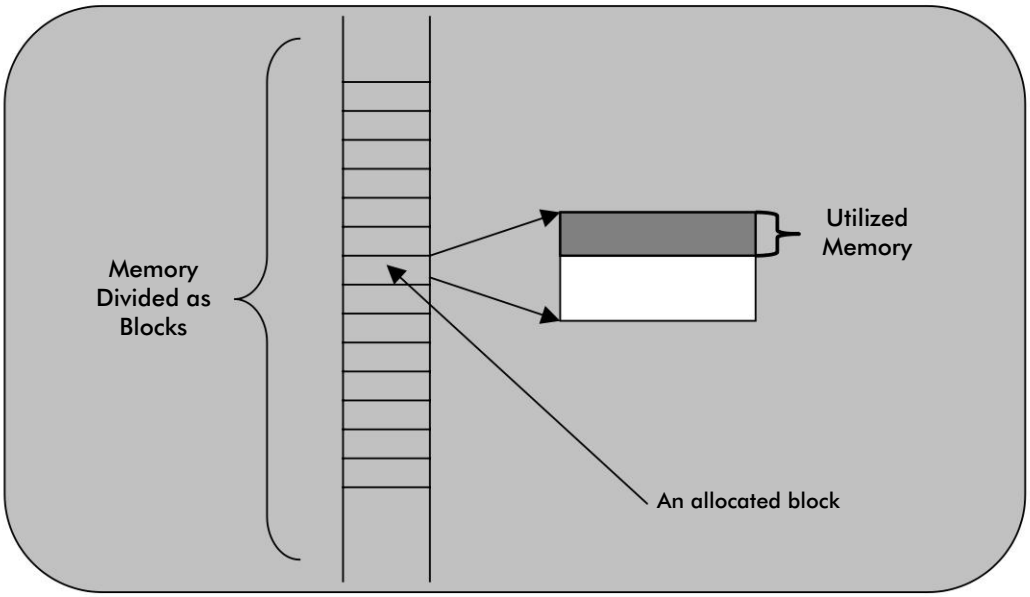


Fig. 4.5 *Internal fragmentation*

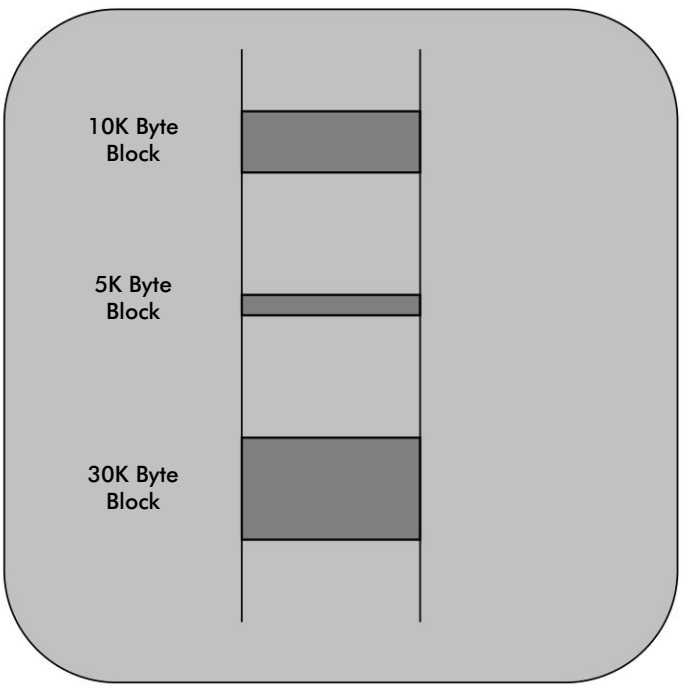


Fig. 4.6 *External heap fragmentation*

External fragmentation occurs when the available memory is distributed throughout the heap. In this case, memory blocks are not available continuously; we may not be able to service a request for memory greater than the largest available block. Consider Fig. 4.6. In this instance, though 45K bytes are available, it will not be able to service even a 35 K byte request.

Heap compaction: We have read before that the heap can be arranged/structured in various ways. One popular approach is that the heap manager maintains a list of available memory locations (known as *free list*).

If a request is made, and no sufficient memory is available, then a process known as heap compaction is carried out. In this case, the entire list of free locations is scanned and adjacent ones are merged (coalesced) to see if we can still service the request.

The problem, which occurs here, is that the compaction can take an unknown period of time based on the fragmentation level of the heap. This will make the system nondeterministic. Long compaction times can wreak havoc while considering the fact that realtime deadlines have to be met. For e.g. think of a case of a flight controller task that started executing heap compaction for 20 seconds during the landing of a flight.

There are various ways of working around this problem. The heap manager can be made to run as a separate thread carrying out problems in installments. This still introduces some level of non-determinism into the system.

Another alternative case is that, compaction can be done whenever a pointer is freed. But, again this could cause a condition where the adjacent blocks again need to be merged and the list restored. This again introduces nondeterminism.

It should now be appreciated as to why we dwelt so long on this concept. The reasons are unique to embedded realtime systems:

i. Realtime

ii. Run for ever

Realtime: In realtime applications, systems not meeting the deadlines can cause results ranging from minor discomfort to huge loss of life and property. In the case of desktop applications, users are usually tolerant when the system appears to have crashed or '*hang*'. These embedded realtime systems can be used by people who have no introduction to computing (e.g. your vegetable vendor). If consumers finds that their

mobile phone is not responding in a way that they want it to, then they will return it back—and no amount of explaining that it was a rare case of realtime deadline miss will help.

So, timeliness is a very important criterion for realtime systems. A designer/developer targeting realtime systems must know that these issues exist and tackle them.

Run forever: Unlike desktop applications that exit after sometime*, embedded systems tend to run forever. (When did you last "reboot" your television/mobile phone?). Memory is a finite resource and the leakage of even a single byte will bring the system to a grinding halt though it may have giga bytes of memory available. So, memory issues must be tacked carefully in embedded systems.

Solution of dynamic memory problems: It might now seem to you that with so many perils lurking around, it is safer not to use dynamic memory management at all and simply allocate all memory statically or in stacks (local variables) during runtime. These approaches are not without their disadvantages:

Statically allocating memory for all the variables may increase the code size and hence the RAM size in the final system.

Lot of memory in the system will be wasted if all the local variables are statically allocated. It will lead to a condition where a lot of memory is badly utilised. That is, some variables require storage only for a small period of time but they occupy memory permanently.

It is not always possible to predict memory requirement that may occur during runtime. Since, memory is really expensive, we cannot have oversized arrays eating up expensive memory. So, we have to live with dynamic memory management.

But, there are some schemes that offer deterministic or even constant time operations for allocation and freeing of memory.

Pools can force realtime behaviour inside the system because they do not take time to allocate and free memory. This is because, memory is not actually allocated and freed. Only the pointers no longer point to the memory segments. They still remain allocated from the system point of view.

*Daemons and services that execute in servers are exception to this. But they are not usually realtime.

The pools divide the available memory space into linked lists of memory segments. Hence, dirty mini-fragments as we discussed above are not created. And, no complicated algorithms to optimise the memory space are required. A first fit algorithm is all that is required at allocation time. And memory freed is not fragmented further. So it is instantly available for next allocation.

On the downside, pools have the following disadvantages:

- At startup time, a lot of effort and time is spent in initialising the pool and allocating chunks of memory of different sizes and making linked lists.

  However, a real time system usually requires the least real time behaviour at start up time. If it is not really time-bound like a mobile phone, which is expected to respond to an incoming call 30 seconds after startup.

- A prudent decision has to be made to define the size of memory chunks inside the pool. Usually, as we discussed earlier, more than one lists of different sizes are available.

  The size of chunks inside these lists is usually configurable inside all RTOSes. The sizes should be decided based on typical memory usage characteristics of the system.

- All said and done, pools may still introduce fragmentation and wastage of memory. If we have 20 byte chunks available and our application requires 10 bytes, I will still be given a pointer to point a 20 byte chunk and 10 bytes will be wasted in this chunk.

  But this does not make the memory space dirty and it is a small price to pay for the elegance and simplicity of the approach.

Some of the most common problems associated with dynamic memory usage are memory leak and dangling pointers. This section will address memory leaks, its causes and possible solutions. The next section will take a look at dangling pointers.

Definition

Memory leak is an amount of memory that has been rendered useless and inaccessible to the system.

What are memory leaks: Memory leak easily seems to be one of the most famous and destructive forms of problems related to dynamic memory usage. How can a part of memory become inaccessible to the system? And what is the significance of the term "memory leak"? Take a look at Listing 4.1. We will use the malloc and free function calls in order to illustrate the concept. The same problem exists for a pool-based system too.

```
#include <malloc.h>
#include <stdio.h>
/*
This program will allocate some memory. Then the memory will
not be freed by the program. Hence the memory will remain
allocated but it will be not be used by the program. The rest
of the system will not be able to use the memory anyway. This
condition is called memory leak because the total amount of
memory available in the system decreases.
*/



int main( void )
{
    int i = 0;
    int* p;

    for (i = 0; i < 10; i++) {
        p = (int*) malloc(sizeof( int ));
        *p = 10;
    }

    return 0;
}
```

Listing 4.1: ***A memory leak***

Listing 4.1 shows an obvious memory leak.

What we see in Listing 4.2 is that some memory is allocated by pointer `lp_pointer` in the form of a linked list. The final block of the linked list has been terminated with a NULL. Now, after the usage, only the memory pointed to by `lp_pointer` has been freed. This however frees only the block pointed to by the pointer. The rest of the blocks pointed to by the next pointer have not been freed. And, now it is not possible to access the memory of the next blocks. There is no means to free this memory now and it practically becomes inaccessible for the rest of the system. This situation is akin to a small hole in the petrol tank of your car. As you are driving the car, some of it is leaking. The leak is not big enough to be noticed immediately, but it is consuming more petrol than it should have done in an ordinary situation. Now, if you had planned to go on a long drive and estimated that the car normally takes

20 litres of petrol to get to the destination, this time around, there are chances that you will become stranded mid-way with no clue as to why your car did not cover the appropriate number of kilometres. This is because the petrol tank of the car had a leak. Similar is the situation in software.

```
#include <malloc.h>

/*
  This program will allocate some memory. Then the memory
  will not be freed by the program. Hence the memory will
  remain allocated but it will be not be used by the program.
  The rest of the system will not be able to use the memory
  anyway. This condition is called memory leak because the
  total amount of memory available in the system decreases.
*/
/* Following is the structure defined as an example */

typedef struct
{
    u8 ld;              // u8, u16 are user define types
    u16 number;
    t_MyStruct* next;
} t_MyStruct;

int main( void )
{
    /* Defined pointers to t_MyStruct */
    t_MyStruct* lp_pointer;
    t_MyStruct* lp_temp_pointer;
    u8           i;
    /* . . . . */
    while(1) {
        if(eventlist.handleDeviceA) { // some condition
            for(i = 0; i <= MAX_NUM_DATA; i++) {
          /* Allocate memory */
          lp_temp_pointer = malloc(sizeof(t_MyStruct));
          if (lp_temp_pointer == NULL) {
              WRITE_ERROR("Memory not available. Exiting…");
              return(FAILURE);
          } //if
          if(i == 0)
              pl_pointer = lp_temp_pointer;
```

```
                /*
                  Code that does some manipulation and computation
                */
                fillData(lp_temp_pointer);
                lp_temp_pointer = lp_temp_pointer-> next;
             } // for
          } // if
          free(lp_pointer);
          return(SUCCESS);
          /*
           A memory leak has been created here since all blocks of
           data pointed to by next have not been freed.
          */
       } // while
    } // main
```

Listing 4.2: ***A memory leak***

Since a portion of the code keeps on taking a chunk of memory and never return it, at some point in time, memory will not be enough to conduct the normal business. Even though tools are available in the market to detect and fix them, it is not easy to detect memory leaks. Thus due to the nature of the problem, the bug is mysterious and usually difficult to trace. You may get statements like the following from the customer:

- System was running fine in the field, though it appeared it was becoming slower. Finally on the 53rd day, all of a sudden it crashed.
- When the user tried to browse a 10 MB file, the system crashed midway.
- To our pleasant surprise, this time, the system did not crash on the 53rd day. It crashed only on the 62nd day.

All this usually points to the same direction. It will be futile to look at the millions of lines of trace statements of the last 52 days because the problem will not be visible there. The problem is that in some corner of the software, somebody is allocating memory and it is somehow not getting freed. Hence over a period of time, the system becomes helpless. A question that comes to the mind here is as to why the system took more than 53 days to crash. Well, we know that embedded systems are event-driven machines. An embedded system probably will not do much unless some external or internal event happens and introduces some activity inside the system. And, if the leak

is inside a particular portion of software that gets executed whenever event E happens, it entirely depends on the number of times the event E happens, in order for the system to stop functioning properly. Second point to be mentioned in this regard is that the memory leak may go on undetected for some time inside the system. In our previous listing, for example, if the program keeps on inadvertently accessing the next blocks via the lp_pointer->next pointer, there may be no problem till the point of time that the block pointed to by lp_pointer is reassigned to some other part of the system.

Typical causes of memory leaks inside realtime systems: Memory leaks arise since the programmers are usually unaware about sections of code—whether a particular pointer to a memory block can be freed as it is not clearly known whether it is required any longer.

Following are some of the typical problems:

- Usually in realtime systems, messages are framed by the sender process by allocating memory from the heap. It is the job of the receiving task to free the memory when it is no longer required. As long as we make this sure, we have plugged all memory leaks between tasks related to intertask communication. Here, a guideline may be required to ensure that all data inside the message is copied into local variables and then the dynamic memory is freed in the first function called inside the receiving task.
- Most of the time, especially in large teams, typical situations happen in which pointers to memory are moved in such a way that the memory is no longer accessible. This happens particularly when a pointer is to be reused for some other location and the memory pointed to by this pointer is not freed beforehand since it is still required.
- Lack of sound programming principles may create leaks. If a memory is allocated inside a function and a local variable is used to access it, its scope will no longer be valid when the function returns. If the memory has not been freed, it will create a leak. Refer Listing 4.1 and 4.2.
- As we saw in the above example, the allocation and freeing routines should be robust. Usually, it is possible to catch such memory leaks by doing effective code reviews. Testing may not be a quick solution to detect leaks.
- If we buy and integrate software from outside, there are chances that memory leaks creep in. This code may be in the form of a module or it may be a library.

The author of this software does not have full knowledge of how it is going to be used. For example, a library may allow the application code to allocate memory in some way, but the library may not be able to free the item since the library cannot know when the application has finished using the memory block. One option is for the library to provide a free routine to be used to free the memory. This does not solve the problem since the application anyway has to remember to free the memory. The second solution is for the library to use static allocation instead of dynamic so that this issue of freeing of memory does not arise in the first place. However, a static allocation works fine for application. It can wreak havoc for a library since it no longer guarantee reentrancy of the calling function. (We will delve on this issue in Chapter 7.) So, the only solution for the application is to free the memory allocated by the library. However, in this case, the developer of the library needs to mention this explicitly in the interface specification.

Even though clear guidelines are formed in order to identify who owns the dynamic memory and who has to free it, the solution is not always easy. For example, imagine that one task is receiving messages from a communications channel. It allocates space for the message and that space will not be freed until the message is completely processed. Since the messages may not be processed in the order in which they were received, some may live longer than others. All pending messages live on a list whose length varies according to the number of messages being processed at any given time. Let's say that this embedded system must forward the messages to another device, and the message cannot be deleted until delivery is confirmed. Since the messages are going to many different destinations, and some destinations may have some down-time, causing retries, it would not be feasible to process the messages in a first-in-first-out manner.

In problem areas such as these, dynamic memory management enables more efficient use of the available RAM than a predefined number of buffers. When the memory is not being used by one message queue, it can be used by another queue, or by a completely different part of the program.

Dangling pointers: The problem of dangling pointers often arises when there is more than one pointer to a specific block. If the first entity owns the memory and wants to free it, then it must first consider whether any other pointers point at that location. If any do, and the first entity frees the memory, those other pointers become dangling pointers — pointers that point to space that is no longer valid. When the dangling pointer is used, you may happen to get the correct data, but eventually the memory will be reused (via another call to malloc()) leading to unpleasant interactions between the

dangling pointer and the new owner of that piece of memory. Consider the following code listing for a demonstration of dangling pointer.

```
#include <malloc.h>

u8 * createMem(void)
{
     u8 * lp_data ;
     u8  vl_index ;

     lp_data = & vl_index ;
     /* do some operation with vl_index and lp_data */
     return lp_data ;
}
```

Listing 4.3: *Creation of Dangling Pointer*

In this listing, lp_data and vl_index are local variables inside the function. Since lp_data does not point to the heap its scope is limited to the stack of the function createMem. When the function returns its value, the memory location pointed to by lp_data is no longer valid. If this value is assigned to another pointer in the calling function, that pointer will become a dangling pointer.

Tips

A leak occurs when you fail to free something; a dangling pointer occurs when you free something that was not yet ready to be freed.

Memory leaks and dangling pointers are similar to race conditions in a number of ways. The misbehaviour they cause may occur far from where the bug was caused. As a result, these problems are difficult to resolve by stepping through the code with a debugger. For both memory leaks and race conditions, code inspections sometimes catch these problems more quickly than any technical solution.

Adding debug code to generate output is often a better alternative than a source code debugger, but in the case of race conditions, it could alter the behaviour enough to disguise the problem. With memory leaks, adding debug code can change the shape of the memory layout, which means that dangling pointer bugs may exhibit different

behaviour. Another drawback is that if the debugging code consumes memory, you may run out of RAM sooner in the debug version than you would in the production version. Still, a leak is a leak and should remain detectable regardless of these side effects of the debug code.

4.2 GENERAL OBSERVATIONS ON POINTER-RELATED BUGS

This section tries to document some of the most common pointer-related bugs observed by us during our tryst with embedded systems. Through this section, we can arrive at some general guidelines in order to troubleshoot pointer-related bugs. However, the bottom line is: better safe than sorry. ☺

4.2.1 Symptoms of pointer bugs

Embedded systems provide no control checks on the values of pointers and memory inside the system. So, it can lead to a situation where pointers can become a run-away horse. If proper care is not taken, it can lead to corruption of executable code or system stack. Memory related bugs are difficult to debug. And, if some trace statements are put and the program compiled again, the source of bug may shift location because of the change in memory layout of the program.

- The program starts to crash inside a well-tested function or inside a library function:

  Cause: A pointer may have become corrupted and written into this function code space. This makes the execution to crash at this place.

  Solution: No direct solution exists without a debugger. Try to put watchpoint on the memory area of the function to monitor when the memory is "written". This will lead us to the code that is doing the illegal memory write.

- The program sometimes works and sometimes crashes.

  Cause: An uninitialised variable pointer is potentially used inside the program and is being used to write to the memory. Since this pointer contains a random value, it may be corrupting different portions of the memory at different times. Hence this intermittent crash of the program at different places.

  Solution: Make sure that all pointers are used properly, that is, they are pointing to proper memory addresses.

4.2.2 How to Find and Fix pointer-related bugs

Fixing pointer-related bugs is a two-step process. First, all the observable quantities are studied and a pattern is created. Then the source of the bug is traced through these consistent observations. Here are some guidelines:

- While debugging, it helps to document everything inside the program: all arguments passed, environment variables, etc. In this way, reliable logs can be generated and tracked based on these standard values.

  Many times, we can make two and two together by looking at a list of such parameter value-output pairs. The best way to do it is to maintain a testing report log and running tests from a batch file that contains enough comments to map the values of the input and erroneous output values.

- During implementation, it is advisable to introduce probing of the trace statements inside the code. It generally helps if a trace statement is present at the top of each function and inside each impossible switch statement.

  These trace statements can help significantly during debugging process since they can pick the trail of the program like Sherlock Holmes! We just need to backtrack from the final executed statement in order to hook the bug.

- Debuggers are very helpful in solving pointer-related problems. They provide tools to watch the values of memory pointed by these pointers. Certain values should immediately raise suspicion.

  For example, if you see a pointer with a small negative value (e.g., FFFFFE hex), it is possible that the pointer has either been corrupted or was never initialised in the program.

4.3 LESSONS LEARNT

Memory is a very precious resource in embedded systems. The memory management in early embedded systems used to be hard-wired. The advent of high level languages brought in their memory management procedures. C and C++ provide three kinds of allocation of memory: static, automatic and dynamic. Static and automatic variables occupy memory throughout the duration of the program. Hence, dynamic memory is sometimes needed in order to reuse heap at different times. Heap management can be done through variable sized or fixed sized buffers. In both methods there are trade-offs and invariably some memory is wasted inside the buffer, and in order to maintain the heap.

Heap fragmentation is a big problem if efficient heap management routines are not in place. This problem can render the embedded system nonrealtime. Usage of pools solves this problem at the cost of wasting some memory.

However, as long as dynamic memory is used, there is always a possibility to introduce two kind of pointer-related bugs: memory leaks and dangling pointers. It is very difficult to catch such bugs because of their inconsistent occurrence.

4.4 REVIEW QUESTIONS

- Which are the different ways of memory allocation in C?
- How does size of allocation block affect the effectiveness of memory allocation?
- What is heap management? What are the different ways of heap management?
- What is heap fragmentation? Why does it happen? What are the ways to prevent it?
- How does heap fragmentation reduce the realtime behaviour of an embedded system?
- What is memory leak? How can it be prevented and debugged?
- What is dangling pointer? How is it different from memory leak?

Chapter
5

Interrupts and ISRs

Interrupts are to embedded systems what pointers are to C/C++ programmers. Most embedded programmers shudder when they hear about interrupts. The interrupts are inarguably one of those components of embedded system programming that are toughest to comprehend. This is because interrupts form the boundary between hardware and software. The inputs from the external world enter the software realm usually by interrupts. This chapter explains the basic concepts around interrupts and the ways to program them.

5.1 WHAT IS AN INTERRUPT?

Every embedded system typically takes input from its environment or its user. The *interrupt* mechanism is one of the common ways to interact with the user.

Consider a situation in which the microprocessor has to process inputs from three peripheral devices:

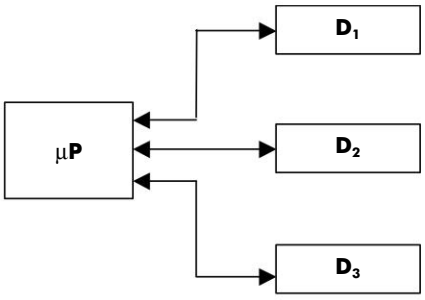


Fig. 5.1 *A Microprocessor interacting with three devices*

One way to accomplish servicing of the three devices is by polling (technically sounding synonym for '*asking*') each device for input. The algorithm for this could be:

```
while the program is running {
    1.  Poll Device D1 for input
    2.  If input is present, ProcessInputFromD1
    3.  Poll Device D2 for input
    4.  If input is present, ProcessInputFromD2
    5.  Poll Device D3 for input
    6.  If input is present, ProcessInputFromD3
}
```

Listing 5.1 ***Algorithm for three polled devices***

This keeps the microprocessor always busy. It is either polling for input or processing a polled input. This method has more cons than pros with one definite drawback of appearing wicked to the microprocessor for forcing it to work forever.

The other mechanism is the interrupting mechanism. In this mechanism, the device informs, i.e. *interrupts* the microprocessor whenever it has some input for the processor. The clear advantage of this method over the polling method is that the processor is free to do other work(running other applications ☺) when there is no input from these devices.

Definition

An interrupt is a way to *asynchronously request* the processor to *perform a desired service.*

Analysis:

Asynchronous: Because, the interrupt might come from another device that may not be clocked by the system clock. The interrupt might arrive independent of the microprocessor clock. (However, the interrupts are presented to microprocessor only synchronously—the processing is synchronous to the system clock)

Request: Interrupts (except the one called non maskable interrupt (NMI)) are requests. The processor is free to defer servicing the interrupt when it is working on a higher priority code (could be the service routine for a higher priority interrupt).

Perform Desired Service: Nobody likes to be interrupted for fun (especially for others') and this applies to microprocessors too. A device interrupts the microprocessor to request it to perform a defined service or to pass on some data to the processor.

5.2 POLLING Vs INTERRUPTS

One evident disadvantage of polling is that it consumes a lot of processing power. And, since all the devices are polled in a sequential manner, there is no explicit priority mechanism in the polling method. Let us consider the loop described in Listing 5.2.

```
while (1) {
     if (input_from_D1)
          Process_D1();
     if (input_from_D2)
          Process_D2();
     if (input_from_D3)
          Process_D3();
}
```

Listing 5.2: *Code for a polled loop*

Consider that D1 and D3 have inputs for the microprocessor (i.e. they require the service of the processor and the processor has begun work on input from D1). So, input from the D3 has to wait for the processor to finish its work with D1, and query D2 before it can start to work with input from D3. Even if device D3 has a high priority, the processor cannot be made to process D3 before it finishes its work with D1 and its query with D2.

What if D3 is an input port and its buffer could overflow if not attended for a specific time? We may be able to finetune the loop (and the functions that service D1 and D2) in such a way that D3 will be attended before it overflows.

A simple way is to change the main loop is described in Listing 5.3.

```
while (1) {
     if (input_from_D1)
          Process_D1();
     if (input_from_D3)
          Process_D3();
     if (input_from_D2)
          Process_D2();
     if (input_from_D3)
          Process_D3();
}
```

Listing 5.3: *A tuned loop*

In this case, if processing of D1 and D2 are within the overflow time of D3, device D3 is safe. Sometime, we may be required to finely tune/optimise the functions that process inputs from D1 and D2 such that the input from D3 does not overflow.

So far so good. It may now seem to work without any problem. But, what if another device is added or due to change in requirements (the only requirements that do not change are that of dead software), priority of a device changes, then the situation becomes very awkward. All the fine-tuning is lost and we have to embark on another attempt to adjust the code and the priorities. The coding for this case reduces to mere jugglery. Even after a second tuning, we might have to anxiously wait for next change in requirements with crossed fingers. Simply put, *this solution is not elegant.*

5.3 THE JARGON SOUP: TYPES OF INTERRUPTS

The area of interrupts and servicing them is filled with many terms. This jargon soup is explained below.

There are many types of interrupts. The general nomenclature observed is

- Hardware interrupts
- Software interrupts

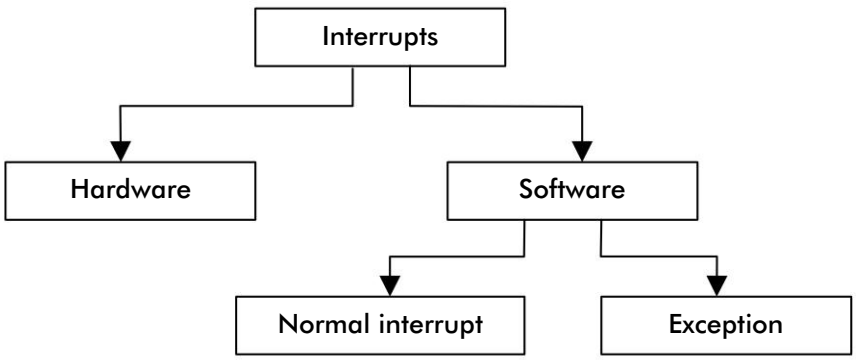


Fig. 5.2 ***Interrupt nomenclature***

The interrupts can be classified on the basis of the source of the interrupt.

Hardware interrupt: If the microprocessor is interrupted by external device/hardware, then the interrupt can be classified as hardware interrupt.

Software interrupt: Surprise! All interrupts need not be raised by external hardware. They can be raised by executing some software instructions (depending on the specific microprocessor). These interrupts are known as software interrupts.*[1]

The software interrupts can be further classified as either interrupts or exceptions. If an interrupt is planned (or occurs deterministically or is anticipated or is raised intentionally), then it can be classified as a 'software interrupt'. If it is unplanned, then it is classified as an 'exception'.*[2]

There is yet another way of classifying interrupts based on their periodicity:

- **Periodic interrupts**: If the interrupts occur at fixed intervals in timeline, then the interrupts can be termed as *periodic.*
- **Aperiodic interrupts**: If the interrupts can occur at any point of time, then they are called *aperiodic* interrupts. (for e.g. anticipating a key press from user)

The interrupts can also be classified based on their temporal relationship with the system clock of the processor:

- **Synchronous interrupts**: If the source of the interrupt is aligned exactly in phase with the system clock (the clock used by the microprocessor), then the interrupt source is said to be synchronous (Fig. 5.3). E.g., a timer service that uses the system clock.
- **Asynchronous interrupts**: If the interrupt source is not in phase with the system clock it is termed as asynchronous (Fig. 5.4).

This might seem contradicting the definition of interrupt stated in the beginning of the chapter, but the definition covers the hardware interrupts that arise from the external devices. The synchronous interrupts occur because of software interrupts or devices that are driven by the system clock and typically form a small subset of interrupts arising from external devices in a typical embedded system.

*[1]Programmers who worked in DOS age would immediately remember the famous INT33H and INT 21H that are some of the oft-used software interrupts—for mouse and other general purposes.

*[2]C++/Java programmers should not to confuse this with the exception handling mechanism available with these languages.

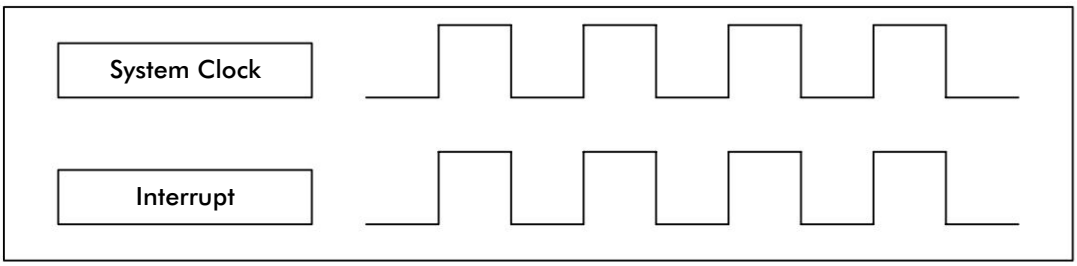


Fig. 5.3 *An example of a synchronous interrupt*

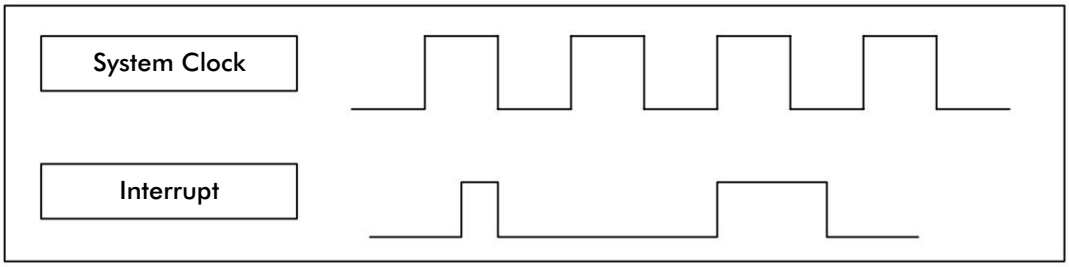


Fig. 5.4 *An example of an asynchronous interrupt*

5.4 INTERRUPT LATENCY

Interrupt latency is defined as the time that elapses from the instant the interrupt was raised to the first instruction of the interrupt service routine being executed.

Greater the latency, greater is the response time. So, the user will feel that the system is not responsive enough. We will use this term in the context of interrupt handlers very soon.

5.5 RE-ENTRANCY

Re-entrancy is a very important concept to be understood while studying concurrent systems. To explain what re-entrant code is, we must first learn to appreciate the concurrency present in the system that runs several jobs simultaneously.

In the context of a realtime system, these jobs are called tasks, which are similar to processes in general operating systems. These tasks execute concurrently and share resources including global variables.*

*The concept of tasks is covered in detail in RealTime Operating system chapter.

The concurrency of tasks implies that a particular piece of code (could be a function) may be used by many tasks at the same time.

Consider the case where two tasks are using a common code — say, a function call. To put it simpler, consider that the two tasks are concurrently executing for the same function. Then, the code that is executed is common to both the tasks.

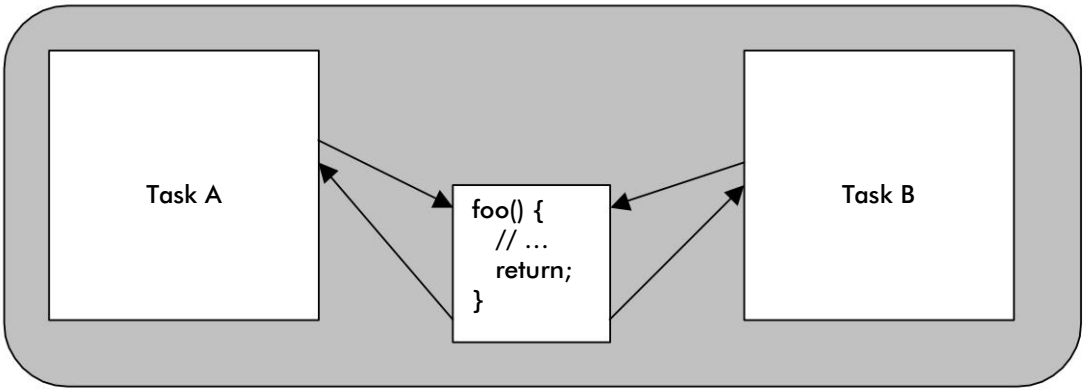


Fig. 5.5 *Two tasks calling a function simultaneously*

In Fig. 5.5, foo() is executed in the context of A and B separately, i.e. in their own stacks. This means that the local variables defined by the function foo() and the arguments passed to function foo() are allocated in the stacks of respective tasks. This is illustrated in Fig. 5.6.

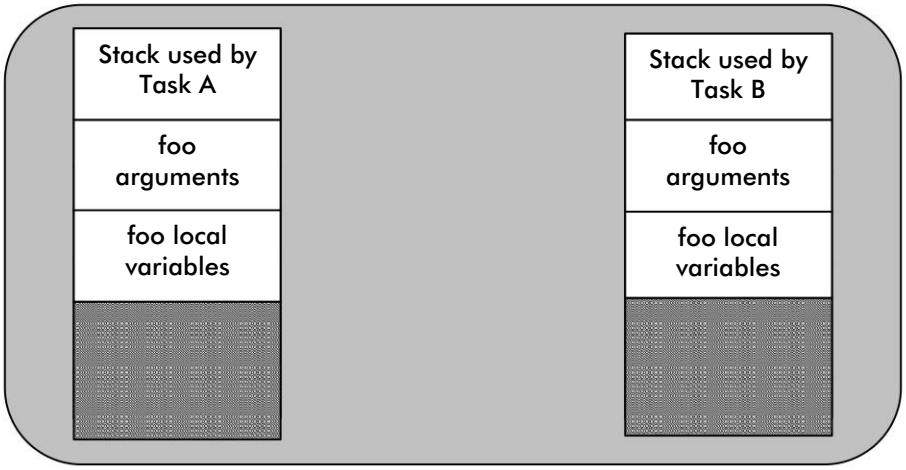


Fig. 5.6 *Stack utilization of Tasks after calling foo*

Say, the function foo() has a local variable a.

```
int foo( int b )
{
      int a;
      /* Rest of processing */
}
```

Listing 5.4: *Sample foo*

The typical layout in memory is illustrated in the Fig. 5.7. From this picture it should be clear that changing of local variable 'a' of foo() called in the context of task A does NOT affect local variable 'a' of foo called in the context of task B. (since they are in different memory locations independent of each other).

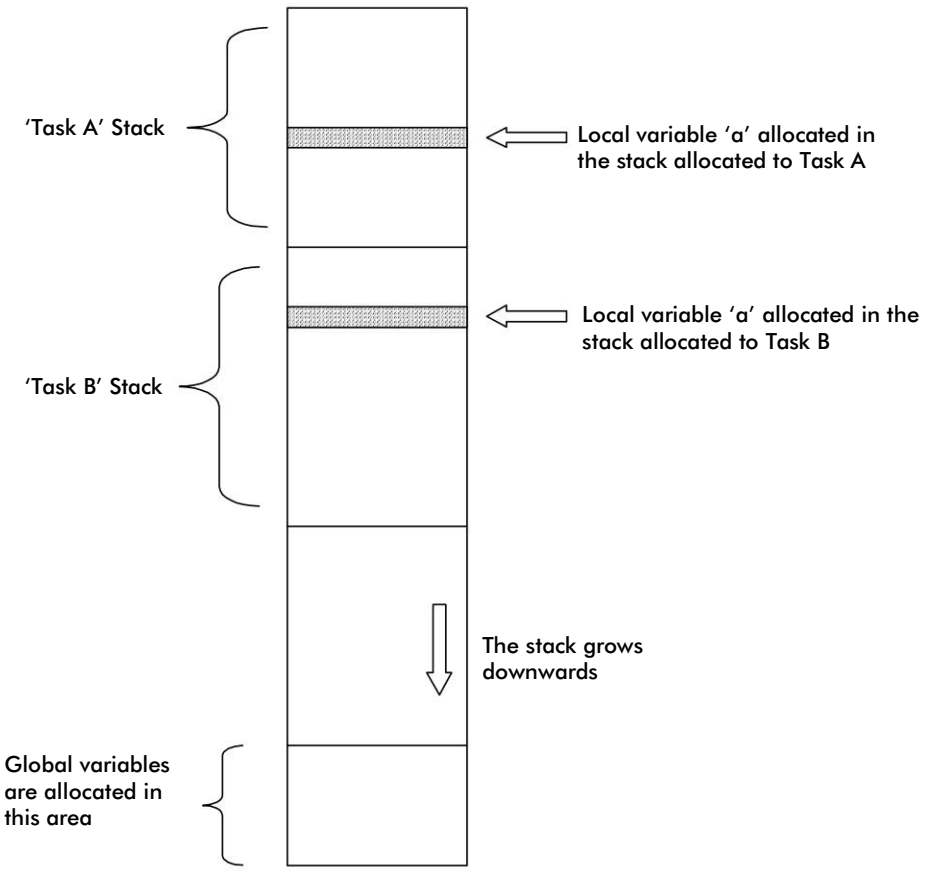


Fig. 5.7 *Memory layout of stacks of tasks A and B*

We can see that the local variable 'a' of foo() when called by tasks A and B gets allocated in the stack of tasks A and B respectively. So, any changes to local variable 'a' in the context of task A does not affect the instance of 'a' in the context of task B.

However, this is not the case with global variables. If a variable has been defined as in following listing, this variable can be accessed by any other task in the system.

```
int a;
int foo( int b )
{
  a = 1;
  /* Rest of processing */
}
```

Listing 5.5: ***Global variable in a task***

In other words, when foo() accesses global/static data,*[1] clashes can occur between instances of foo() called by the two tasks.

The Peterson's solution for mutual exclusion between two tasks that share a common resource is given below: (Don't worry if you don't even recognise it. This algorithm is in the scope of theory of operating systems. This is given just to explain re-entrancy.)

```
// ...
turn = pid;  // process id
while ( turn == pid && require[other] == true )
        ;
```

Listing 5.6: ***Peterson's Solution***

The comparison of turn = = pid in the above code may seem unnecessary because the variable turn is not changed before the comparison (or turn never becomes lvalue of any expression).

The comparison may sound irrelevant in a von-Neumann*[2] model of computing which is a sequential execution model.

---

*[1]Static variables declared within a function is IDENTICAL to a global variable in terms of storage except that the access is limited to function that defines it. Otherwise it gets stored along with other global variables.

*[2]Von Neumann can be considered the father of existing computing model that is widely used in which the CPU executes the instructions of a program sequentially. If you can pardon us for adding to the confusion, many concurrent programs are abstractions over the von Neumann model where the OS provides virtual concurrency over a single processor that executes instructions sequentially.

But we should note that in this solution, turn is a global variable. The task that runs the above code might have been pre-empted after executing the statement turn = pid. The higher priority task may now change the value of turn. So, when the execution returns to the first task, the value of turn would have changed. Now the validation turn = = pid immediately after the assignment turn = pid makes sense.

The use of shared global variables amidst multiple tasks without any synchronisation may lead to race conditions.*[1]

So, we can conclude that if a function uses only its local variables and does not use any variable with static storage (includes global variables and variables that have a static storage specification) are safe to call any number of times. This kind of code is called '*re-entrant*' code.*[2] It should be observed that any set of code (not necessarily a function) could be classified as re-entrant.

Summarising, a code can be called re-entrant if and only if:

- It uses only local variables (i.e. the variables allocated on the stack)
- It does not use any variable that has static storage (global variables and local variables that are specified with static storage)
- It calls only functions that are re-entrant. (i.e. these three rules must be applied to all the function that the code calls)
- If global/static variables are used, they should be accessed in a mutually exclusive way. The mechanism to create a mutually exclusive access to such variables is described in the chapter—realtime operating systems.

Note

Interrupt Latencies

After fixing some major fixes, once we decided to analyze the performance of the system. We attached the logic analyzer to measure the interrupt latencies. To our astonishment, we found that the latencies were much larger than we had even anticipated. This was heavily affecting the throughput of the system. So, we decided to analyze what the problem was. We found that we had two sets of memories one that was

*[1]Explained in the RTOS chapter.

*[2]There is another camp that likes to define a reentrant function as a function that returns the same value independent of the time it is called.

internal to the processor and one that was external. When the code was built and loaded (we had used the scatter loading technique described in Chapter 2) we had designed that some code be loaded in the internal RAM and some in the external. During the path in which the interrupt was serviced, some of the code was in internal and some in external. We noticed that the jump between these locations caused the most of the performance penalty. So, we changed the loader file to put all the code needed by the ISR in internal RAM to avoid expensive jumps. After this the latency was reduced to a desired value.

5.6 INTERRUPT PRIORITY AND PROGRAMMABLE INTERRUPT CONTROLLERS

Microprocessors and microcontrollers usually have only a few interrupt pins.* This number is typically two to four (with one allocated for NMI (nonmaskable interrupt)). This number is normally smaller than the number of devices that the processor must interface.

So, in a situation where multiple devices need to interrupt the microprocessor, a hardware called "programmable interrupt controller" is used.

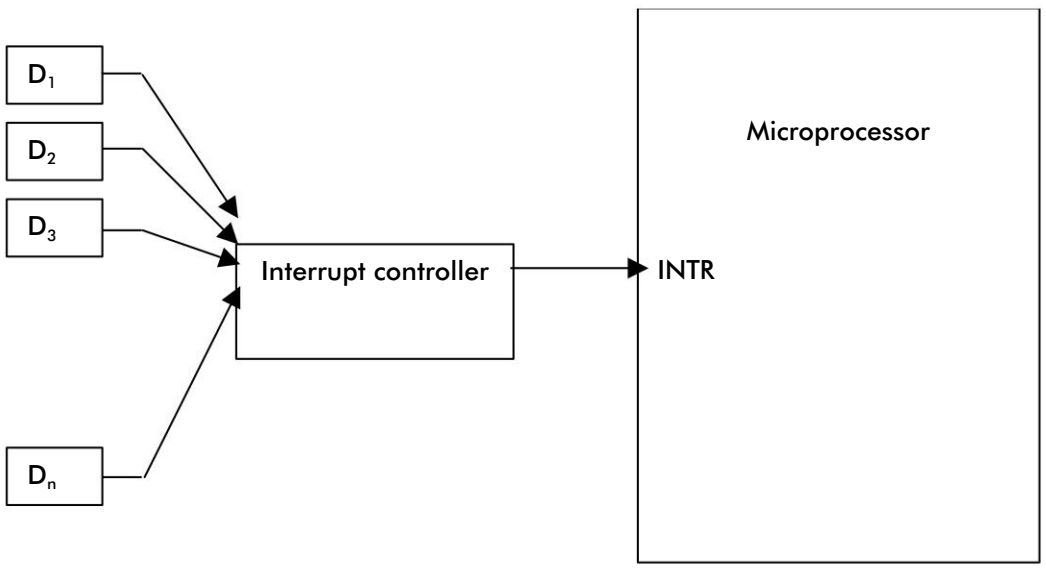


Fig. 5.8 ***Multiple devices connected to a sing INTR using an interrupt controller***

*This is a typical situation. Chips designed with specific applications may have a number of interrupt pins.

The diagram above describes the position of the interrupt controller. Some people tend to imagine that the controller is a highly intelligent* hardware. (*Look ma!* This device arbitrates the interrupts raised by many devices). In its simplest *avatar*, this controller can be as dumb as an 8-3 encoder.

A case of this simple 8-3 encoder used to map multiple addresses is given below. (Old-timers will be reminded of their happy times with 8085)

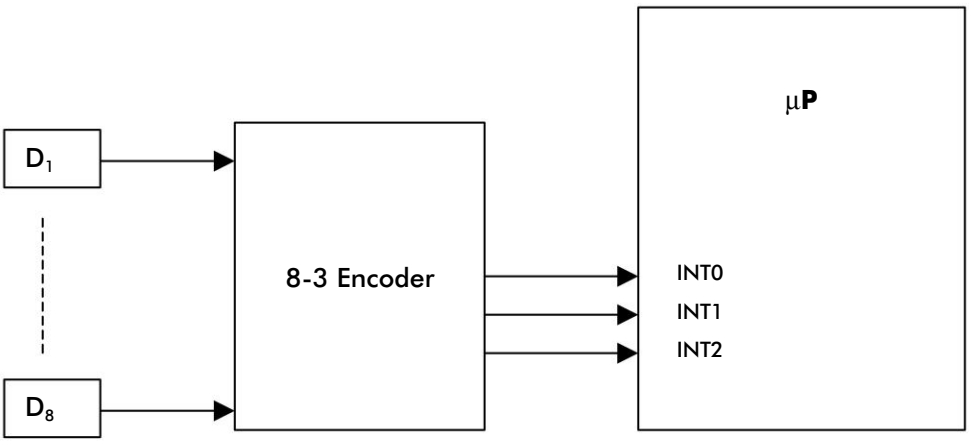


Fig. 5.9 *An 8-3 encoder used to connect 8 devices to a microprocessor with 3 interrupt pins*

An interrupt controller is usually equipped with much more sophisticated features. But, only a subset of these features is needed for most of the common operations. I always compare these features to features in a DVD/TV remote control. (I never use anything other than PLAY and STOP).

The interrupt controller can be used to assign different priorities to different interrupts. The device also takes care of providing information to vector the interrupt to its service routine.

5.6.1 Masking of interrupts

Applying a mask to an interrupt source does not endow an interrupt with superpowers and make it strong, flexible and agile (except for Jim Carey interrupt if one exists)—it disables it! When the microprocessor is working on a very high priority interrupt, the least we can do is to disturb it with a low priority interrupt (a microprocessor has its

*Students in Indian engineering colleges like to associate a word 'fundoo'.

right to spend its time in a responsible manner). This can be compared to us programmers switching the phone to 'Don't disturb mode' when we are doing some very important work.*

So, when the microprocessor is working on a higher priority interrupt, lower priority interrupts are masked.

5.6.2 Interrupt service routines

So far, we have always talked about microprocessors doing some work on being interrupted. The code that gets executed on raising an interrupt is called an interrupt service routine (ISR) of the corresponding interrupt. ISR is a very important word in the parlance of embedded programming.

Now there are a few questions to be answered:

- How do we know which interrupt occurred?
- How does the processor know which ISR to execute for the interrupt that was raised?
- How do we pass an argument to the ISR?

These will be clear in the following sections.

5.7 TYPES OF ISRs

We should remember that the interrupts do not occur in a controlled environment. They can come in any order at any rate and in any sequence. Our software must be able to handle all of them in a way that the system meets its requirements. The ISRs can be classified simply as:

- **Non-nested handlers**: When this interrupt service routine is executing, it cannot be pre-empted by another interrupt.
- **Nested handlers**: This handler can be pre-empted by ISR of another interrupt. So, this kind of interrupt handler should be coded with special care to avoid race conditions.

*Like reading email ☺.

5.7.1 Non-nested handlers

These are the simplest types of handlers. When an interrupt is being serviced, all the other interrupts are blocked. So, all the interrupts that were raised during the execution of this ISR will be pending till the completion of the ISR that is executing.

The sequence of events that occur after the interrupt is raised is illustrated using the flowchart in Fig. 5.10.

The flowchart is explained as follows:

1. The interrupt source*[1] raises the interrupt. This interrupt could also be raised by an exceptional software condition such as Data Abort.
2. Usually the processors (e.g. ARM cores) disable the interrupts. The interrupts remain disabled till the ISR re-enables it.
3. Based on the information provided by the device, the processor jumps to the corresponding ISR indicated in the interrupt vector table (IVT) explained later in Section 5.8.2.
4. The ISR saves the context (explained later)
5. The ISR now gets executed.
6. The ISR re-enables interrupts.
7. The context is restored.
8. On return from the ISR, the control of execution is transferred to the task that was being executed when the interrupt was raised.*[2]

Debug Tips

Sometimes it might seem that the system does not respond after the execution of some interrupt service routines. In case of processors we can check if the interrupts are re-enabled in the ISR. If not the interrupts will be permanently disabled and the system would appear non-responsive. The system would be up once we re-enable the interrupts in the ISR.

*[1]It could be an external device connected to the microprocessor or could be a device integrated within the microprocessor as in case of SoCs (System on Chips).

*[2]It is strictly not necessary that execution returns to the same task that was executing when the interrupt was raised. This scenario is explained more in the RTOS chapter.

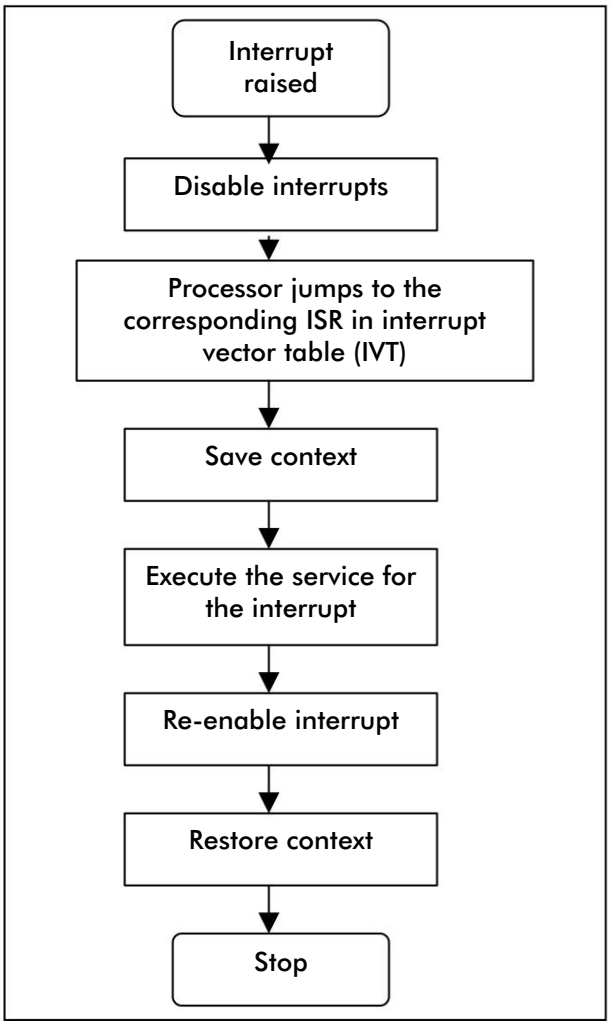


Fig. 5.10 ***Flowchart for simple non-nested interrupts***

These are the generic steps that are executed while processing an interrupt. These steps are not rigid. For e.g. steps (vi) and (vii) can be interchanged. And, the process of handling interrupts is highly processor specific. In some processors (8085) we have to explicitly disable interrupts in an ISR. But in case of some processors, the interrupts are disabled once an interrupt is raised.

Saving of context

We have seen that an interrupt can be raised when a task is executing. The processor now jumps to the corresponding interrupt service routine and returns back to the task

once it is finished. In a single processor system (i.e. a system with a single CPU/microprocessor), the path of the instruction pointer (or PC—program counter) is given by the following diagram:

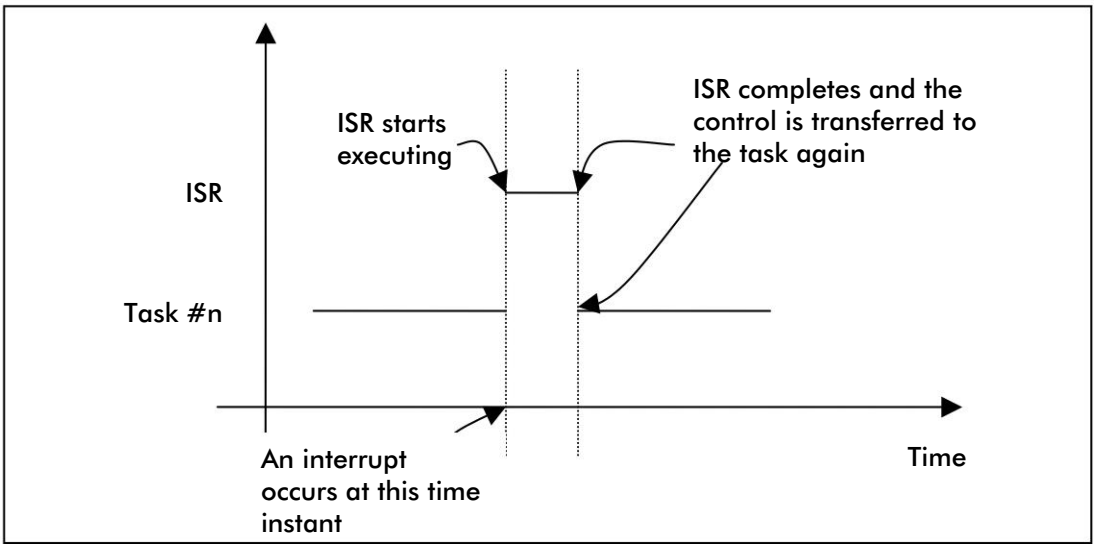


Fig. 5.11 *An ISR interrupting a task*

We all know that many calculations (and variables) are stored in registers. For e.g. 8085 has eight registers and all additions can be performed only with the register A (accumulator). Say, for e.g. the task that was being executed before the interrupt was raised (indicated by 'Task #n' in the figure) stored some value in the accumulator (or as a matter of fact in any register). Now, the interrupt occurred and the ISR is executed. The ISR may need some registers for its computational purposes. So, if it uses the register A for some adding purposes, the previous value stored in register A will be lost. So, when the Task #n resumes, the data it would have stored in its registers would have been corrupted. This is not acceptable.

So, before the ISR uses any of the registers, the ISR usually saves the entire set of registers in the system stack. So, on completion of the ISR, the context that was saved is *restored*. Now, the Task #n can continue without its data being corrupted. Usually microprocessors provide a single instruction using which the entire context can be saved/restored instead of saving/restoring all the register contents one after another.

Context saving in modern processors

Many modern processors (RISC and CISC processors) have *register banks*. This means that there are few sets of registers. The processor operates in a few modes (say Normal, Supervisor, etc.). And each mode has a register bank, i.e. its own set of registers. So a task that operates in a normal mode can use its set of registers. When it is interrupted, the processor switches to another mode that has its own set of registers. Because of this there is no corruption of registers since the general task and ISR operate on their own set of registers.

The visible advantage of this method is that the time taken to save and restore the context is saved. This increases the speed at which the interrupt is processed.

The disadvantage is that the microprocessor requires a set of registers for each mode.

Tips

A microprocessor may choose to bank only a subset of registers instead of the entire set of registers. In this case we can speed up programming by using only the registers that are banked. A good optimising compiler can assign variables to registers that are banked in a particular mode.

The transition from single tasked environment model (like DOS) to multitasked environment is complicated because, it takes time for a programmer to get used to the effects of concurrency in the system. The effects are still worsened in a system where interrupts are enabled.

For e.g., in an 8085 system, we could write a delay routine based on the number of clock cycles required by an instruction (typically NOP) and looping a particular number of times.

```
;Delay Routine
      MOV A, 0xD4 ;Some number based on the delay required
LOOP: NOP
      DCR A               ;decrement register A
      JNZ LOOP            ;check if loop one
      RET                 ;return
```

In the delay routine above, we know the number of cycles required by the loop (the NOP, DCR A and JNZ instructions). We also know the clock speed at which

the microprocessor executes. So, we can determine delay based on the required delay value.

Loop value = (Required delay time)/(Time for one loop)

This worked fine in single threaded applications as in 8085.* But, this will not work in any multi-tasked environment or in an environment that supports interrupts. This is because, these delay routines work with the assumption that the CPU time is always available to them.

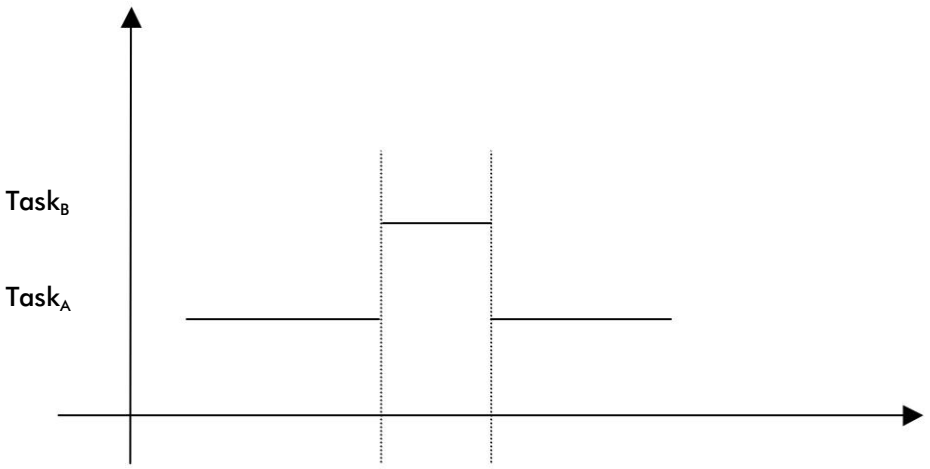


Fig. 5.12 *Tasks deprived of CPU time because of preemption by other tasks*

As illustrated in the above figure, we can see that no task can have all the CPU time for itself. It could be pre-empted by other tasks. So, carefully calculated loops like the one shown above will not work. We will typically need help from the hardware and the OS. Many OS (and RTOSes) provide APIs to start and stop timers without bothering about these issues of pre-emption by other tasks/processes.

We can see that the advantage of the simple non-nested interrupt handler lies in its simplicity of implementation. It is also very easy to debug, since it cannot be pre-empted by other interrupts.

But this simplicity comes with a cost—High *Interrupt Latency*. If we give another careful look at Fig. 5.12, then we will notice that during the execution of a simple non-nested ISR, the interrupts are disabled till the completion of the ISR. During this period, the

*We should remember this could work only when interrupts were disabled.

microprocessor will be effectively isolated from the environment. So, the system does not respond to any other interrupt during this period. This implies that disabling interrupts for a particular period of time will increase interrupt latency of other interrupts (including another instance of the same interrupt) for the same amount of time. This kind of addition to already existing system latency may not be acceptable in all cases.

In many cases (as we will see later) disabling the interrupts for such a long period is not required at all.

The comfort of coding and debugging a non-nested ISR comes with a heavy price tag—increase in latency of interrupts. As indicated earlier, increase in latency of interrupts beyond a certain threshold value may not be acceptable (i.e. will not satisfy the system requirements).

5.7.2 Nested interrupt handlers

We saw in Section 5.5 that re-entrant code can be executed any number of times to give the same result. So, the trick is to write to an ISR that separates its code into two parts:

- Non re-entrant code
- Re-entrant code

The flow chart for a nested interrupt handler is given in Fig. 5.13.

So, in this case the interrupts do not remain disabled for the entire duration of the ISR, but only for the period for which the non re-entrant code executes. This reduces the interrupt latency.

Warning

If you find that most of the code in an ISR is non re-entrant and this technique does not save much time (especially when the ISR is long), still we have a problem. This means that the system uses many global/static variables. We should try to redesign the system in such a way that we minimise the number of global variables.

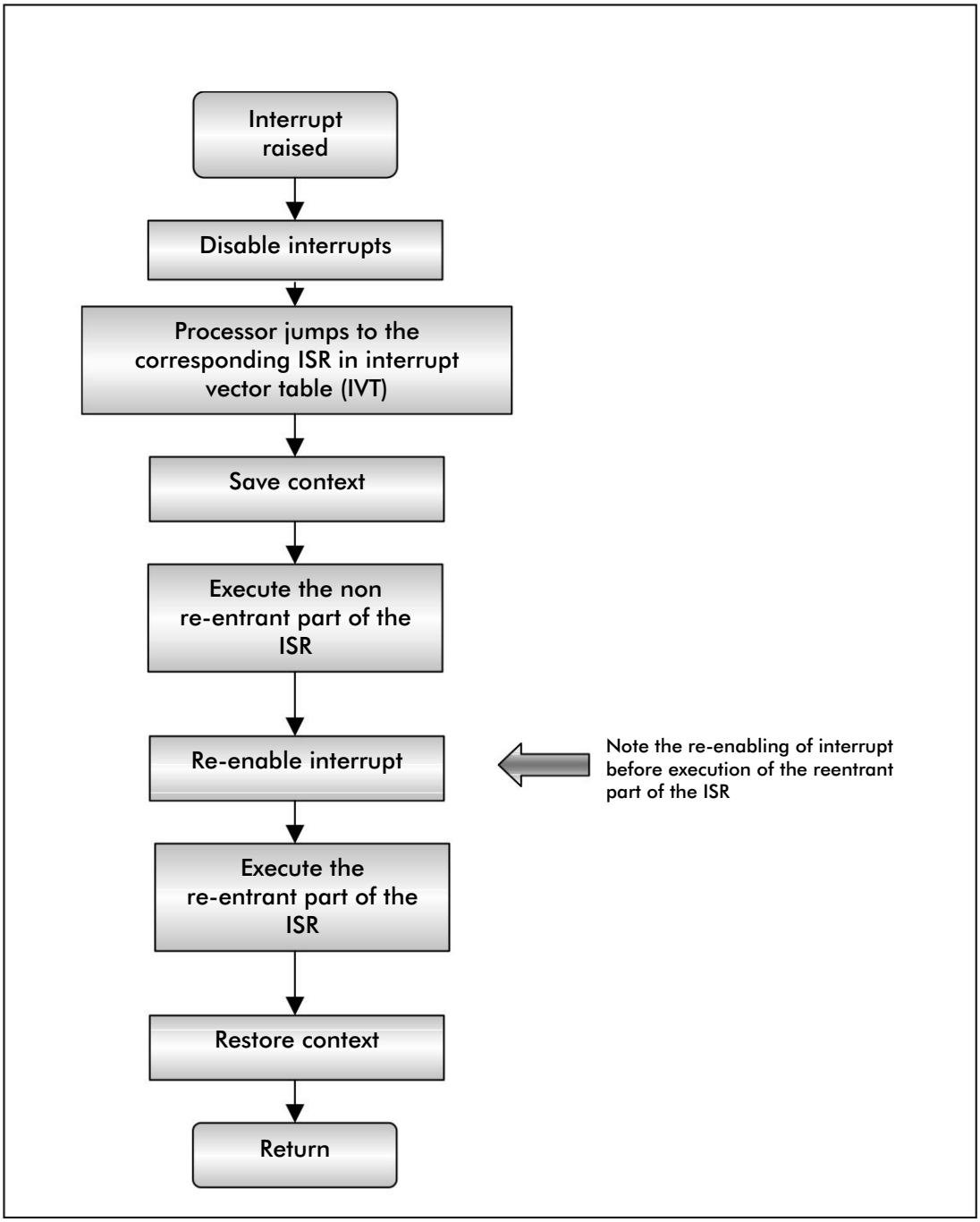


Fig. 5.13 *Flow chart for nested interrupt handler*

5.8 LOOKING UNDER THE HOOD ON HOW INTERRUPTS WORK

The first question that pops up in the minds of novice embedded programmers is how the processor knows that the interrupt occurred? Surprise! The answer to this question is that the processor polls for it. But, in this case the processor polls the interrupt pins during every system cycle for a fraction of the cycle. This is different from the device polling that we discussed earlier. In this case, reading of memory is not required and it does not involve reading from the registers of these interrupts.*[1]

Note

In the case of device polling, the processor typically executes some instructions to find out if the device requires service. In the case of interrupts, the processor polls its interrupt pins for input during a fraction of its (every) system cycle.

5.8.1 The case of missing and extra interrupts

This is an extension of the debug tip given on page 106. Interrupts can still be missed even if interrupts are re-enabled in the ISR*[2]. A worse problem is spurious interrupts because of signal noise. To solve these problems, we have to first note which problem occurs the most. Usually in the case of level sensitive interrupts, the interrupt pin must be asserted for a minimum duration for the processor to detect the interrupt. So, if the pin is not asserted for this minimum period, the interrupt will be missed. Similarly, if the interrupt pin is configured as edge sensitive, it is possible that any spurious signal may interrupt the processor. The choice of configuring as edge sensitive or level sensitive must be done bearing in mind the above discussed criteria and as mandated by the device and microprocessor requirements.

Programming Tips

If the Interrupt Vector Table (IVT) can hold ten entries and of which only four are used, then usually people do not fill the rest of the entries or fill it with NULL. This doesn't solve the problem. And, could actually worsen it. The program will crash once it gets a spurious interrupt and we will not be able to track the problem. (Unless we use some kind of realtime trace program or a trace program like Introspect of gdb of GNU/GCC compiler collection). To make things simpler we can have a single handler for all spurious interrupts, say, `UnexpectedInterruptHandler` and have a breakpoint on this handler. The execution would stop when a spurious interrupt is detected and we would know where it occurred.

*[1]This is just a sample case. Internal implementation may vary.
*[2]From these tips, we get a feeling that there are more ways to miss an interrupt than to trap one correctly. Sadly this is true.

Nowadays, the processors can be configured in such a way that the interrupt is edge/level sensitive.

So, by watching the INTR pins, the microprocessor knows if an interrupt occurred. The next question is "How does the processor know which ISR to execute when the interrupt is raised?". The answer depends on the processor.

If the controller is a simple controller such as a 8051 controller, the address to which the jump is to be made is hardwired.

5.8.2 Vectoring

A more prevalent and widely used scheme is called '*vectoring*'. In the case of vectoring, the device/interrupt controller asserts the interrupt (INTR) pin of the microprocessor and waits for the 'interrupt acknowledgement' (INTA). On receiving the INTA, the device places a 8-bit or 16-bit data in the data bus of the processor. Each device (interrupt source) is assigned a unique number. This data is used to branch to an appropriate ISR. This array/vector of interrupt handlers is known as the 'interrupt vector table' (IVT)

The steps can be summarised as follows:

i. The device or the interrupt controller asserts the interrupt pin (INTR).

ii. The processor detects the interrupt and acknowledges it (INTA).

iii. The device places 8/16 bit data in the data bus.

iv. The CPU sends EOI (end of interrupt) to the device/interrupt controller.

v. The CPU branches to the ISR corresponding to the data.

Then, based on the ISR type (non-nested/nested), the flow continues as described in Fig. 5.10 or Fig. 5.13 respectively.

5.8.3 How to pass an argument to an ISR?

This final question remains unanswered so far. The answer should be pretty clear from the above discussions… there is no direct way!☺

When devices communicate with the microprocessor, they can share data in many ways. One way is to use a shared memory concept. In this case, both the device and the microprocessor can see a common address space. This does not mean the memory as seen by the processor and the device is identical. This only means an area in memory

can be seen both by the processor and device that may map to different addresses. The device can write the data in a predefined and previously agreed format and raise an interrupt. On receiving the interrupt, the ISR can read from the shared memory and process it.

Another way could be using a DMA (direct memory access). And, the device and processor can have a non-overlapping address space. In this case, a device such as a DMA can be used to write the shared data into the address space of the processor. The ISR can be invoked on the 'DMA Transfer Complete' interrupt.

Keyword–Volatile

The keyword volatile is used to instruct the compiler not to 'aggressively' optimise the code regarding a particular variable. For any compiler (C++, C, FORTRAN etc.) , the optimiser plays a very important role in final object code generation.

The optimiser in a compiler usually does many things such as

- Removing invariant code from a loop such as Listing 8-1

```
// Listing 8-1
for ( int i = 0; i < 10; i++ ) {
   doSomething();
   a = 2; // invariant code - This can be removed from the
          //loop and  be brought 'outside'. Time is saved while
          // iterating
   doSomethingElse();
 }
```

- inlining small functions, to save on jumping to a function and returning from it
- moving frequently used variables to registers. (In old compilers, keyword register was used to request a particular variable to be put on one of the microprocessor's internal register to aid frequent access. It was a request and compilers could ignore it. Only problem was that you could not access the address of a register variable)
- removing multiple assignments, where a single variable is initialised at two or more places and so on.

These are just a hint of some of the optimisations that usually a compiler does. There is a lot more to it. (Probably some compiler writer can send in an article on object code optimisations. It would be very interesting!)

Sometimes, in certain situations it is desirable not to have optimisations. In such cases, where variables should not be optimised, keyword volatile is used. Some of the applications of this keyword are in

- Multithreading / Multitasking
- Hardware related programs such as memory mapped input/output each of which are explained below.

Multithreading/Multitasking: If we use a shared global variable technique to share data or synchronise between threads, and if compiler optimises based on that shared variable, then we could be running into serious problem. For e.g. we might have a part of a code like Listing 8-2. (which is used in Peterson's solution for mutual exclusion of a resource among two processes. Here turn is a variable global to the processes)

```
// Listing 8-2
// ...
turn = pid;  // process id
while ( turn == pid && require[other] == true )
        ;
```

Here, we use `turn = pid` and immediately check if `turn == pid`, because a context switch (the OS may schedule another process) might have occurred after turn = pid, and some other process that shares the variable turn might have changed the value of turn. But compiler might see this validation of variable turn as redundant and may evaluate the condition to be true always. In this case the solution fails. So, we instruct the compiler not to optimise code based on variable turn. For this, we declare `turn` as

```
volatile int turn;
```

This will make the compiler wary of `turn` and will not perform optimisations based on turn.

Memory mapped input/output: In memory mapped input/output (i/o) we could map a device to a particular memory location. Then writing into and reading from that memory location become synonymous to sending input and obtaining output from the device. For example, de-referencing a pointer can mean reading a byte (or word of predefined size) from the device (or port). In Listing 8-3,

```
// Listing 8-3
/* read first byte/word : equivalent
of char c = *((char*) (0xFF4E321A)) in C */
char c = * ( reinterpret_cast(0xFF4E321A) );
/* read second byte/word - compiler may feel
this is redundant */

c = *(reinterpret_cast (0xFF4E321A));
if ( c == 0x4C ) { /* if second byte/word equals 0x4C */
   doSomething();
}
```

the compiler may feel that the second line is redundant and may remove that code. So, the if statement will actually compare the first byte with a predefined value and not the second byte. Again, to turn off the optimisations, we declare c as

```
volatile char c;
```

So that no optimisations are carried out on c.

There was another interesting incident when the use of volatile turned out to be useful. When one of my friends (a beginner) was using an IDE, he ran into problems while debugging—the 'watch' window displayed something like—'Unable to display optimised variable'. I asked him to recompile using DEBUG switch on. But he could not immediately identify location of that option in his IDE. I then asked him to put volatile before the variables he wanted to 'watch' and it worked fine! So, some keywords find some applications the language designers would not have even thought of!

5.9 GUIDELINES FOR WRITING ISRs

5.9.1 Keep it short

This is the one of the key factors while writing an ISR. Typical rule of thumb is that it should not exceed 25–40 lines. If you have a longer ISR, you have a problem tagging along.

ISRs are meant for executing and not for debugging. Moreover debugging ISRs is not always possible (more in Debugging ISRs section). A longer ISR usually also means a higher latency (though not always). So, do away with long ISRs. The best way to avoid long ISRs is to make ISRs to spawn off a new task that will take care of non-time critical stuff. This is where a commercial RTOS would come helpful. It can take care of bookkeeping information that is required to create and maintain a task, while we can concentrate on problem logic.

The ISR should perform only and only the time-critical stuff and nothing else. This will reduce the interrupt latency.

Once, our team was in charge of a (now very famous) wireless networking protocol. We found that the system could not get data at higher rates because it took some time to get the data from hardware into the software. To solve the problem we had two options:

- Go for a faster processor
- Rewrite the ISRs

The first one was difficult, since the higher end processor was more expensive (obviously) and did not fit into our budget. So, it was decided that we should check all the ISRs that were causing the problems.

Fortunately we could identify the problem soon. The problem was not in the transmission part as one would immediately expect but on the reception side.

One of the main tasks for reception expected a packet in a specific format. But, the format of the packet received from the air was different (since it was just raw information). The ISR tried to format the received data into the required format before passing* it on to the task responsible for reception. This took valuable time during a packet burst.

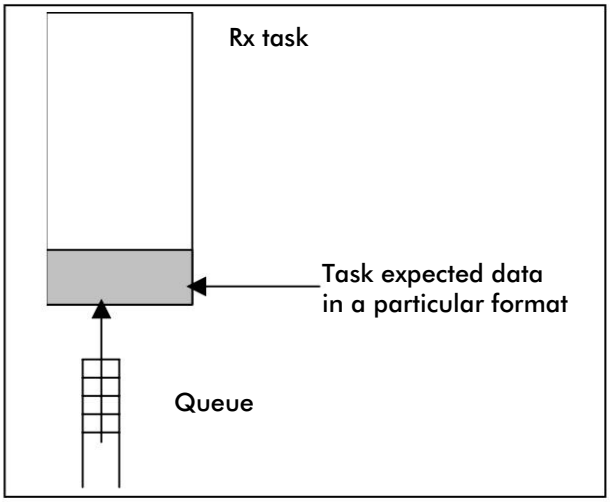


Fig. 5.14 *Lower Interface of Initial Rx Task*

The initial Receive ISR pseudocode is as follows:

```
Rx_ISR()
{
  retrieve pointer from register;
  enable hardware to get the next
packet;

  format data to queue it to task;
  post the data top the task;
}
```

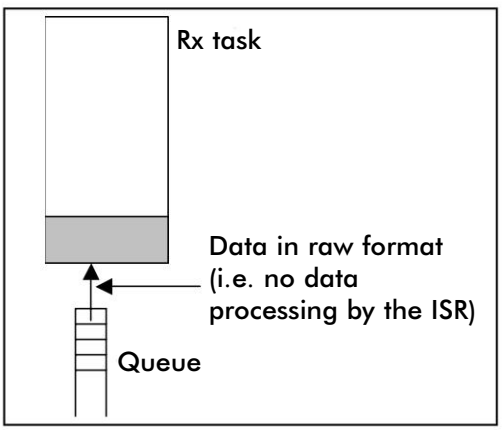
Listing 5.7: *Pseudocode for Receive ISR*

*To pass the data from ISR to the task IPC (Inter Process Communication) mechanisms were used. These are covered in detail in the RTOS (RealTime Operating Systems) chapter.

Of these, the first two and the last step were very small. The third step took the longest time since it had to peek into some of the fields of the header to format the received data.

Actually, it was a mistake that the ISR was designed this way. Any way it is always better to be late than never, and we changed the lower interface of the reception task in such a way that it could accept raw information. The ISR would just receive the data and pass on the pointer to the task.

The lower of the interface of the receive task now look like the illustration below:



Now, the ISR just received the data and posted it to the task and the system could handle the bursts easily.

This is again the appropriate time to discuss the difference between normal desktop programming and realtime programs. In the above example we saw that the program was logically correct. It performed the operation that was logically correct. Still, the necessary functionality was not achieved because, we have to add one more dimension—time, for correctness of realtime programs. This makes the life of a realtime programmer really interesting.

The pseudocode of the new ISR is given below:

```
Rx_ISR()
{
  retrieve pointer from register;
  enable hardware to get the next packet;

  post the data top the task;
}
```

Listing 5.8: ***Pseudocode for Changed Receive ISR***

Note that the step that was used to format the data is removed (and the functionality is added to the receive task).

P2P

This solves the latency problem. But there exists more problems in this area. One of the important topics not discussed above is the problem of the size of buffers. The question that arises in one's mind is *"How much memory should I reserve to buffer the packets before they get processed completely?"*

Since the packets will be processed by the task and the ISR queues the packets rapidly during the burst, the calculation of buffer required at the receiving side must be carefully calculated. This is a classic example of application of "queuing theory".

5.9.2 Keep it simple

Evaluation of complex trigonometric equations and integral calculus expressions are a big NO in ISRs. Avoid nested ifs, complex switch within switch statements. Also avoid long iterations.

This rule is again an offshoot of the 'debuggability' of ISRs. Since debugging ISRs is extremely complex and perhaps impossible in many cases, we cannot write an ISR that embodies complex logic. We may not have a chance to step through the ISR. So, as much as possible design a system that has ISRs that do truly simple tasks like copying from a shared memory, updating global variables, etc. Never let it perform complex mathematics and statistics.

5.9.3 Protect the critical section from interrupts

The critical section of the code should be executed when the interrupts are disabled to avoid data corruption problems. The critical section means that the part of the code accesses global variables, i.e. the non re-entrant part of the code.

Beware of operations that might seemingly look simple enough needing no protection. Microsoft compiler produces the following code when we try to increment a variable i using statement `i++`:

```
mov   eax, DWORD PTR _i$[ebp]
add   eax, 1
mov   DWORD PTR _i$[ebp], eax
```

This is not a single instruction ☺, as seen from the assembly, i.e. this is NOT an atomic operation. If i is a global variable, and if the code gets pre-empted after one or two of the above instructions, it may produce unexpected results.

Note

An atomic operation is one that should be done without being preempted. The above example describes that even a simple operation like i++ need not be atomic. So never assume anything about the output produced by the compiler. Either make sure that the operation is indeed atomic by checking the assembly output. Or protect them by disabling interrupts. While disabling interrupts, we must remember that it increases the latency of the system.

5.9.4 Re-enable interrupts as soon as possible

As discussed in the "nested interrupt handlers", interrupts need not be enabled only at the end of the ISR. They can be enabled after all the work in the critical section is over. So, care should be taken such that the critical section code is not distributed throughout the ISR. (If it is possible), the ISR should be designed in such a away that all the critical section code is bundled together in the beginning of the ISR. This will help us re-enable interrupts at the earliest.

5.10 DEBUGGING ISRs

There could be many issues while dealing with interrupts. This section describes various scenarios that could arise due to incorrect handling of interrupts and ways to identify and solve the issues.

A golden rule of debugging ISRs is that, "Don't write complex ISRs that need to be debugged". ISRs are asynchronous so, we may never know when they occur during the execution of the program. Another problem regarding debugging of ISRs is they cannot usually be debugged by using break points. In case of networks, we might get the next packet when we step through the code and the program will fail. In some cases in systems using robots and huge servomotors, we can cause damage to the machinery if we stop them abruptly. So, the best way to write ISRs is to keep them as simple as possible.

5.10.1 System does not respond after some time

As indicated earlier, in most systems, interrupts are the only way by which a system can interact with the external environment. So, if the system does not respond to the inputs from the external environment, it usually means that the interrupts are not enabled. In

case of some processors, the interrupts are disabled at power-up. In these cases, interrupts must be explicitly enabled in the startup code (or boot-loader) of the system to receive interrupts.

So, we must check the processor/board documentation on enabling interrupts and make sure that interrupts are enabled when the software begins executing.

The other reason could be that one (or more) ISR has some bug. Interrupts are disabled when an ISR begins execution. The ISR should enable interrupts explicitly before it returns. If not, the interrupts that occur later will not be serviced. (These kinds of errors can easily be identified in the code review sessions. If not, these kinds of bugs take much more time in debugging efforts later).

This situation can also occur because of configuration of the interrupt controller/interrupt pin. In the beginning of the chapter, various kinds of interrupts were discussed. One of the types were "Edge/Level Sensitive" ones. Nowadays, all these pins are configurable. But still, we need to configure them right.

Let us consider the following situation: If, the interrupt pin is configured as level sensitive and if the INTR line is low,

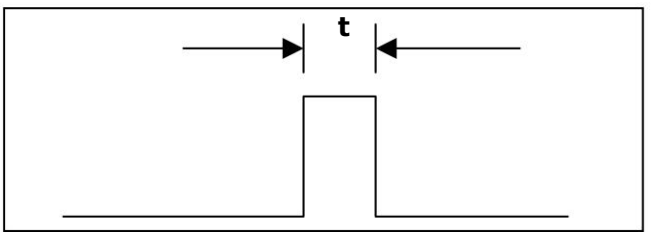


Fig. 5.15 *Level sensitive interrupt being asserted for 't'*

When an interrupt is raised, it should be high for sufficiently long period for the microprocessor to recognise the interrupt. If this time period 't' for which the interrupt signal is asserted is less than the time required for the processor to recognise the interrupt, then the processor will not service it.

Another extreme is that the interrupt is configured as edge sensitive and the processor recognises spurious interrupts that are caused by even mild electrical disturbances. The usual practice among novice programmers is to fill only the IVT entries that correspond to the interrupts that are used by the system. The rest are left unfilled. This would cause the processor to jump to undefined locations when a spurious interrupt

occurs. Another practice that is equally dangerous is filling of all the unused IVT entries as NULL (0). This would cause unexplained behaviour when a spurious interrupt arrives. So, during the development period (before release), we should ideally write a handler for unexpected interrupt and place a breakpoint over it. This will help us identify a spurious interrupt when it occurs.

A typical handler is given in Listing 5.9.

```
void UnexpectedInterruptHandler (void)
{

#ifdef DEBUG
  DB_PRINT("Unexpected Interrupt");

#endif
}
```

Listing 5.9: ***A sample handler for unexpected interrupts***

If the code is to be shifted for field-testing, it is advisable to add a print statement that will write in flash* (or some other nonvolatile memory). We can use this trace to identify occurrences of these interrupts.

5.10.2 The system starts behaving unpredictably after some time

There are many reasons why this behaviour could occur. The discussion here is limited to interrupts.

In some processors, there are two kinds of return statements–one to return from functions/subroutines and another to return from the ISR. If proper return statement is not used, then the context will not be restored properly and the system will behave in an unexpected manner.

Sometimes people wonder why different return statements are needed since they both are utilised to return from a single function. In many cases, when an ISR is executed, the processor might change its state. Say, in ATM7, it changes from USER to SUPERVISOR mode. So, returning from the ISR may need to switch back the mode and not just restore the context. This could be the reason for different return statements.

*More in "Type of Memories" chapter.

For e.g., the following is a listing (pseudocode) for a normal procedure:

```
; Save Context
;

; Restore Context
RET ; return from procedure
```

Listing 5.10: *Pseudocode for returning from a procedure/function*

Assembly listing for an ISR would be:

```
; Save Context
;

; Restore Context
IRET ; return from ISR
```

Listing 5.11: *Pseudocode for returning from an ISR*

In all but extremely small systems, C can be used to code the ISRs. Many compilers especially those targeted at embedded systems usually provide a keyword (like INTERRUPT, __interrupt etc.) to specify if a function is an ISR. Then the compiler will take care of inserting the correct return statement. Nowadays, the C compilers optimise so much that in all but very small systems (where no C compiler is available) it is always preferable to use C compiler to generate code instead of assembly.

5.10.3 Value of variables change unpredictable

Whenever something similar happens first make sure that no ghosts (especially those that are computer literate) are around. And either way (whether you found a ghost or not), start a prayer since this is one problem that is very difficult to solve.

One hopes that you would have checked if your memory (RAM, flash etc.) works fine before jumping into further diagnosis. This could be caused by spurious interrupts (if not handled properly as discussed earlier) explained in detail* in point #2. It could also be caused by dangling pointers.

*BIG DETAIL

(If you don't follow the instructions given above after lengthy warnings, you may very well deserve the bug ☺. So make sure that all the handlers for the spurious interrupts are filled).

The problem could be in '*memory writes*' that are triggered by the interrupts. Check all the writes that happen in the period during which the variable changes unpredictably.

We remember a nightmare of a debugging experience during the development of a wireless networking protocol. We always received a packet that would not be accepted by the system because it failed in some validation. What was puzzling was that the packet sent was a valid packet and the same packet was received (i.e. No transmission/reception errors — in wireless protocols there could be errors here also).

And the validation routine was checked again and was found to be perfect. After some time, we narrowed down to the area where the validation was failing. It was in the length of a particular field.

The length of a particular field received was compared to value stored in a global variable. It was here that the validation kept failing. We found that the length against which the received length was validated kept changing mysteriously. Once we found this, we put break points in all locations where this length could be changed (i.e. wherever gu16CorrectLength was on the left hand side of any expression (lvalue)). But it was of no use since the value was not changed in the breakpoints.

Now we removed all the breakpoints and set up a '*watchpoint*'. A watchpoint also breaks the execution of a program not when the execution reaches a particular line, but when a condition becomes true. We can use watchpoint to stop the execution when a write occurs at a memory location. We set a watchpoint to break when a write occurs on gu16CorrectLength. And finally, the program stopped during a `memcpy` (memory copy) in the Receive ISR. The problem was finally solved.

The receive ISR was given a pointer where it could write the received data.

The ISR was supposed to receive a small packet (ACK) after receiving the packet. The transmitter had correctly transmitted the packet, but had transmitted the ACK packet in an incorrect format. The ACK packet was longer. It so overshot its length and

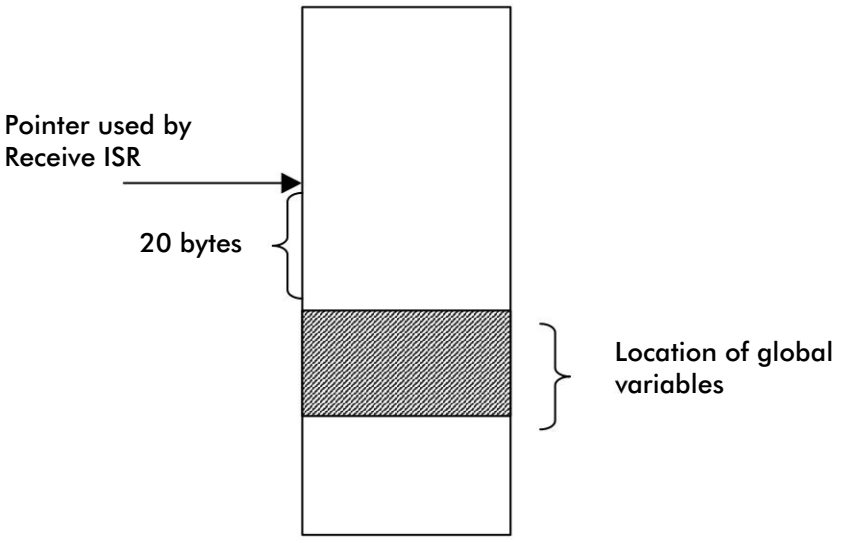


Fig. 5.16 *Bug in receive ISR*

wrote over the global variable space. (One should remember that RTOS' are not UNIX. They don't dump the core whenever an illegal memory write occurs. We can usually write anywhere we want. Actually some other variables had changed too, but they were not observed because they were not in use in the receive path.

So, when we had checked the received packet it was fine. But before the received packet could be processed, the ACK was received and it had corrupted the global variable space. So, once the length of the ACK was corrected, the problem vanished. And, the person in-charge of the board was extremely happy because he was afraid that the board was dysfunctional.

5.11 LESSONS LEARNT

In order to access and process peripheral devices, polling and interrupts are the two most common methods. Polling in a sequential way is not an effective way of handling these devices. Instead, interrupts can be used effectively by the processor while still performing other jobs when the devices are not ready.

Interrupts can be classified in a variety of ways: hardware and software interrupts, periodic and aperiodic, synchronous and asynchronous. When a lot of interrupts are expected, it is wise to use an interrupt controller.

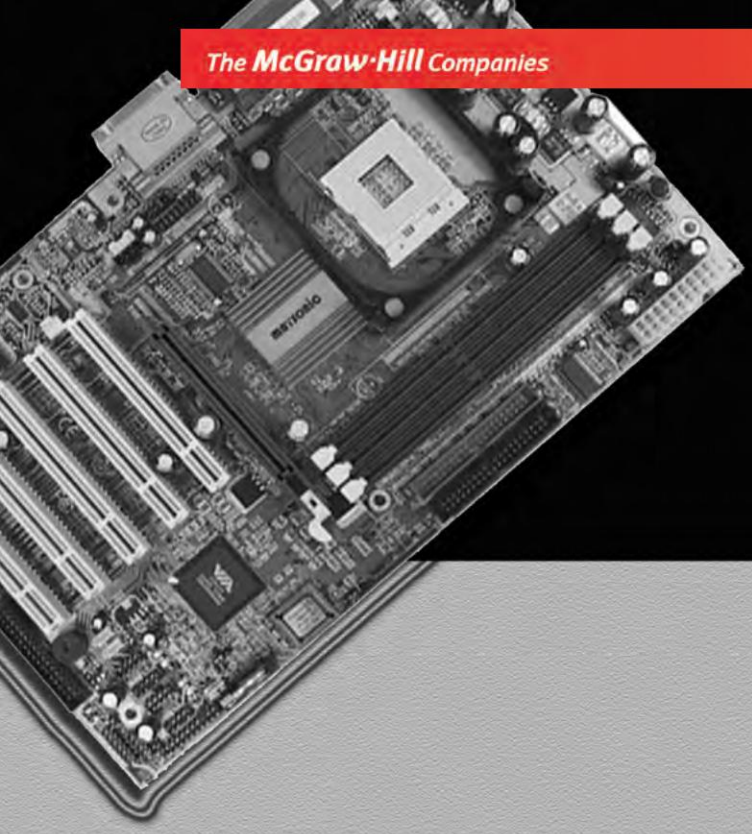
Interrupt Service Routines are executed when an interrupt is executed. They can be nested as well as non-nested. When an interrupt is raised, the current context of the task needs to be saved and the control needs to be given to the ISR.

Non-nested interrupts tend to have higher interrupt latency if proper care is not taken to design them. ISRs should be designed with care so that they are short, simple, efficient and re-entrant. Re-entrancy can be achieved by using only local variables, guarding global variables through mutual exclusion and calling re-entrant functions.

5.12 REVIEW QUESTIONS

- How are interrupts different from polling? Under what conditions polling is better than interrupts?
- What is an interrupt controller? Design an interrupt controller to connect 16 devices to a microprocessor with 4 interrupt-pins.
- What is re-entrancy? How can we ensure re-entrancy while writing ISRs?
- What steps are taken when ISRs are called during the execution of a task? How are nested and non-nested ISRs different in this regard?
- What is interrupt vector table? What is its relevance in embedded systems?
- In what context can we use the volatile construct?
- What are the two general ways that can be used to interface a system to the external world?
- What is an interrupt?
- How can the interrupts be classified?
- How can a code be made re-entrant?
- What are priority interrupt controllers?
- When and why do you mask an interrupt?
- What is saving of context? What is the need? When is it done?
- What are the advantages of using register banks? What are the drawbacks?
- What are the requirements to write a nested interrupt handler?
- What are the guidelines to be followed while writing an ISR?
- How do we handle spurious interrupts?

The McGraw·Hill Companies



SECTION THREE

Correctness is not enough

Most embedded systems are realtime. In this way, embedded systems differ from traditional systems. Chapter 6, introduces realtime theory that can be used to analyze the realtime behavior of systems. (not necessarily embedded). This section describes 'Rate Monotonic Scheduling' algorithm, popularly known as RMS. The RTOS chapter describes the fundamentals of various components of an RTOS like tasks, queues, semaphores etc. It also provides some interesting design insights by discussing some real-life application design problems involving RTOSes.

Chapter
6

Introduction to Realtime Theory

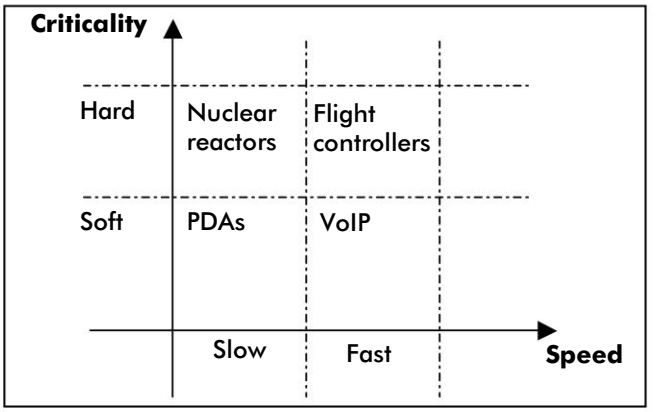
This chapter is one of the prime motivating factors behind the writing of this book. Books on embedded programming shy away from realtime theory at best describing APIs of some RTOS. Books on realtime theory revel in mathematics without thought on practical applications much to the agony of an average engineer/programmer. We honestly believe programming is fun only when the theory behind it is clear. The chapter introduces some aspects of realtime theory (the topic deserves an entire book). After this the reader is encouraged to study other material, which we hope will be easier to comprehend with this introduction.

As described in the introduction of the book, in realtime systems, providing the result within a deadline is as important as providing the correct answer. An oft-quoted saying in realtime theory is, "*A late answer is a wrong answer.*" This can be compared to a quiz program. A late answer is usually not accepted (and your chance may have been passed on to your competitor). Sometimes the deadlines are very small, i.e. the system must respond rapidly. But, there could be instances when the response can be slow, but deadlines are critical. So, based on these characteristics, a realtime system can be classified as illustrated in next page.

This is a very important classification since it is common to find people interpreting a fast realtime system as a hard realtime system.

A *hard* realtime system is one where missing of a deadline can cause a great loss to life and property. Aeroplane/Space navigation systems and nuclear power plants are some examples of this kind of system.

A *soft* realtime system is one where the system is resilient to missing a few deadlines.



Examples are DVD players and music systems. The user usually tolerates an occasional glitch.

If we carefully observe the two definitions of hard and soft realtime systems, they do not include a notion of the speed with which the system must respond to. They simply describe the criticality of meeting the deadlines. If the system has to meet the deadline in a few microseconds (to few milliseconds), then the system is categorised as a fast realtime system.

If you are watching video on a broadband network, most probably your system is receiving data at a few mbps. So, this is a fast realtime system. But, this is not a hard realtime system because a rare miss in audio/video is tolerated and does not make you lose your life or property (unless you were watching world cup soccer ☺). Similarly hard realtime systems need not be fast.

It is this timeliness factor which distinguishes realtime software from normal application software targeted at desktop computers. In realtime software, we might have to ascertain that all the deadlines are met before deploying the software. In desktop software, usually ensuring correctness is sufficient, but not in the case of realtime systems. So, while developing realtime software, we should do '*Performance Analysis*' during the design phase.

6.1 SCHEDULING THEORY

Definition

The scheduling theory deals with *schedulability* of *concurrent tasks* with *deadlines* and *priorities*.

Analysis:

Schedulability: The processing power of a given CPU is fixed. So, if various tasks are present in the system, in the worst case, can all tasks be scheduled in such a way that all the deadlines are met? This is called schedulability.

Concurrent tasks: Because, if the tasks are sequential, (it is known as batch processing), there is no need for complex performance analysis. Since, in sequential processing, a task can begin only after all its predecessors have completed. So, scheduling theory here, deals with scheduling of concurrent tasks.

Deadlines: All (concurrent) tasks have a deadline to meet.

Priority: Different tasks, though they run concurrently are differentiated based on their priorities.

The theory of scheduling is vast and has been around for a long time (1960s, 1970s). It has matured over years. The only sad part of the story is that its application is usually limited to academic interests. Mostly, they are outside the scope of a common engineer. But, the theory has wide practical implications and is immensely useful in mathematical validation of systems. '*Rate Monotonic Scheduling*' is chosen for discussion in this chapter because of its popularity and wide variety of applications.

6.2 RATE MONOTONIC SCHEDULING

As indicated earlier, this is one of the most popular and widely used scheduling mechanisms. This method has also been implemented in Ada* 9X. This theory began its rapid growth from a paper on rate monotonic scheduling published by Liu and Layland in 1973 in ACM.

We would have studied about monotonic sequences, i.e. a sequence of numbers that are either arranged in an increasing or decreasing manner.

So, the numbers

2, 5, 12, 27, 99

form a monotonic sequence but the following series does not.

3, 5, 8, 6, 10, 9

*Ada is a programming language that has inbuilt support for tasks and scheduling of tasks.

6.2.1 Definition

The rate monotonic scheduling is a way of scheduling in which increasing priorities are assigned to tasks in decreasing order of their periods.

i.e. the tasks are arranged in monotonic series of their periods, e.g. if we have 4 tasks:

Task	Period	Priority
1	100	2
2	250	4
3	150	3
4	80	1

Now the tasks are arranged in increasing order of their periods and priority is assigned.

80,	100,	150,	250
Task#4	Task#1	Task#3	Task#2
1	2	3	4

Here Task #4 has the highest priority and Task #2 the lowest.

Before proceeding to intricacies of RMS, let us make some assumptions:

6.2.2 Assumptions

Let T_i be the period of the periodic task.

Let C_i be the time the processor would be required by the task

Let D_i be the deadline of the task. Initially, the D_i is assumed to be equal to T_i (i.e. the task should complete before its period).

Now, let us define a useful term called the utilization ratio (U_i)

$$U_i = C_i / T_i$$

U_i is defined as the ratio of the execution time of task i to its period T_i. Obviously the acceptable limit of U_i is 1.

6.2.3 Necessary conditions

The following describe the necessary conditions required for a given set of tasks to be schedulable. They are necessary but are not sufficient conditions for schedulability.

Individual utilisation

$$\forall i : T_i : C_i \leq T_i$$

This is the first necessary condition. If a periodic task takes more time to complete than its own period it cannot be scheduled (even if it is the only task in the system). If this is the case, then the processing power of the processor can be increased to reduce C_i. Or, the algorithm can be improved or changed, or some part of the task can be moved to hardware.

Total utilisation

$$\sum_{i=1}^{n} (C_i / T_i) \leq 1$$

This rule states that the sum of all utilisation ratios cannot exceed 1. The previous result said that individual utilisations couldn't exceed 1. This is because if $\sum U_i = 1$, then it means that CPU is 100% time loaded. It cannot do any extra task. So, the sum of all utilisation ratios must be less than or equal to 1. Again, this is a necessary test and not a sufficiency test.

Now, let us discuss one more important concept—Deadlines—before discussing the next conditions. Deadlines are points in time from the arrival of event to that a task needs to complete its work. But, now let us assume that the deadline of a task is equal to its period, i.e. a task can complete its work before its next period

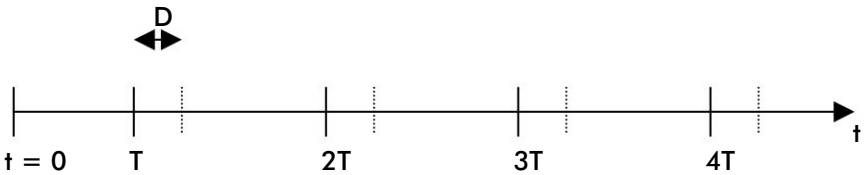


Fig. 6.1 *Period and deadline*

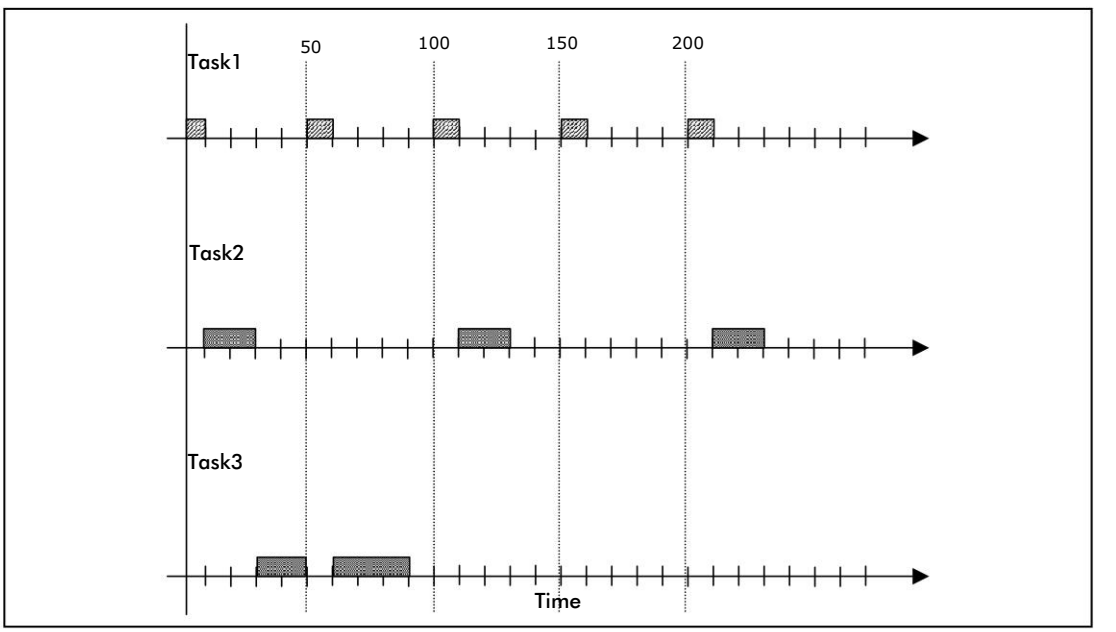
In the above picture, we see that an event occurs periodically with period T. The deadline is the time within which the response must be given, in our case the task must be accomplished. The deadline and the period of a task are two independent entities. For the sake of the following discussion, let us assume that the Deadline = Period.

Consider the set of following tasks:

Task#	Period	C_i	U_i	RMP*	C_i (cum)
1	50	10	0.20	1	0.20
2	100	20	0.20	2	0.40
3	200	50	0.25	3	0.65

RMP= Rate Monotonic Priority*

We have to make sure that tasks are listed in the order (as above) of their Rate Monotonic Priority. Let us draw a diagram that depicts the state of the system when all the tasks are started at the same instant ($t = 0$).



In a pre-emptive system, the ready task with the highest priority will be active at any point of time. (There are other kinds of systems where like time-sharing and fairness scheduling policies are adopted*).

*More later in this chapter.

Let us see how these 3 different tasks are scheduled in the system. Since RMS uses a strictly pre-emptive scheduling, only the task with highest priority is executed by the system at any time instant.

We also assume the worst case situation that all the tasks are started simultaneously (at $t = 0$). Since the task #1, (with $T = 50$) has the highest RMS priority, it is always executed whenever its period occurs.

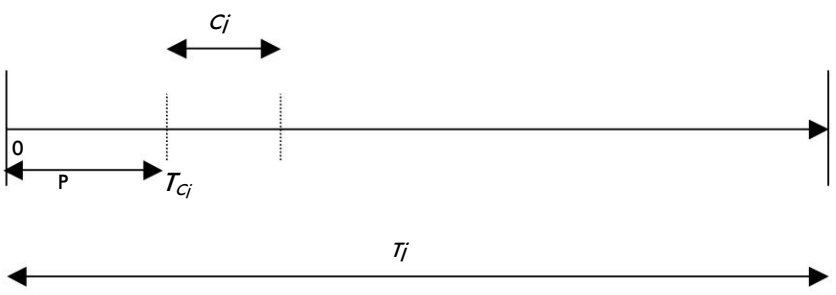
When task #1 finishes at $T = 10$, task #2 begins its execution. Now, in a pre-emptive system, only the highest priority task will be executed. Let us see how these three different tasks are getting scheduled. RMS uses a strictly pre-emptive scheduling scheme. From the diagram we see that task #1, gets scheduled exactly at $t = 50$, 100 etc. Task #1 takes a period of 10 from the CPU.

Though all the tasks are ready to execute at $t = 0$, only the highest priority task is executed. After this completes, the task with next higher priority is executed (task #2 with T–100. This executes from time 10–30 in the timeline.

Now, after task #2 completes, task #3 can execute. Task #3 requires a time of 30 units to finish its job. It starts at $t = 30$ and should complete by 80 if it is uninterrupted. But at $t = 50$, the highest priority task—task #1 gets ready to execute. So, task #3 gets pre-empted at $t = 50$. This is called a *scheduling point*. Now, task #1 completes at t = 60. Since task #2 is not ready to execute now, task #3 can continue. Task #3 completes by $t = 90$. So, all the three tasks get scheduled.

Therefore, for any task, Task #j to complete its execution, all the higher priority tasks within the period Tj cannot take time more than Cj.

Please take some breath and read the above conclusion before going to the explanation below. Any task Tj takes Cj time to complete. So, the time it can spare* to higher priority tasks is $Tj - Cj$.



*Is there any other option ☺? This is like 'forced volunteering'.

Consider a task with period *Tj*. The task takes time *Cj* to execute. The period of execution of task *j* may not coincide with beginning of the period unless it is the highest priority task. So, there will be a phase delay *P* before a task starts executing.

For task *j* to complete, the phase delay can be utmost *Tj* – *Cj*. Otherwise it cannot complete before its next period starts.

Similarly, even if the phase delay *P* is less than (*Tj* – *Cj*), if higher priority tasks pre-empt the task *j* for a period more than *Tj* – *Cj*, then task *j* cannot complete.

Consider two tasks *i*, *j*, *Ti* > *Tj*. Hence, priority of task *j* is greater than task *i*. Task *j* will pre-empt task *i* (maximum) *Ti* / *Tj* times. (In our previous example, during the period of task #3, task #1 is scheduled 200/50 = 4 times).

So, time taken by task *j* from task *i* is,

$$(T_i / T_j) * C_j$$

(Because Task #*j* takes *Cj* times to execute)

Therefore, for a task #*i*, all the higher priority tasks pre-empt for

$$\sum_{j=1}^{i-1} (T_i / T_j) * C_j \leq (T_i - C_i)$$

To explain this, consider the previous diagram. Let us consider task T_3. (Period = 200). T_1 pre-empts it $\lceil 200/50 \rceil = 4$ times during the period of task T_3. So, the time taken by task T_1 during the period of T_3 is:

$$\lceil 200/50 \rceil * 10 = 40$$

Similarly for task T_2, time taken by it during the period of task T_3 is

$$\lceil 200/100 \rceil * 20 = 40$$

So, out of 200 units for T_3, 40 + 40 = 80 units have already been taken by higher priority tasks. So, the time left for $T_3 = 200 - 80 = 120$.

The time needed for T_3 is 50 units, which can be comfortably accommodated in the available 120 units. So, the set of tasks T_1, T_2 and T_3 are schedulable.

This is the third sufficient, but not a necessary condition for schedulability.

Note

This is a pessimistic test because we take the ceiling operator (e.g. $\lceil 3/2 \rceil = 2$) while doing this calculation. It must be noted that a higher priority task, in this case, T_1 may not pre-empt T_3 four times as indicated in the calculations. In the example, T_1 pre-empts T_3 only once. This test does not consider the execution time of T_3. It only takes into consideration the period of T_3. So, this is a pessimistic test.

After these three conditions are passed, we can move to the next test, which is also a pessimistic test. The theorem on which the test is based is called the "Utilisation Bound Theorem".

6.3 UTILIZATION BOUND THEOREM

Definition

Consider a set of *n independent periodic* tasks. They are *schedulable* (under the priority assignment of RMS) if

$$\sum_{i=1}^{n} C_i / T_i \leq n(2^{1/n} - 1)$$

Independent—Because for RMS, we assume that the tasks do not interact (neither synchronisation nor communication) with each other.

Periodic—The tasks are periodic, i.e. they have a certain frequency. They are not sporadic or event triggered.

Schedulable—All the tasks can meet their deadlines.

Since the proof is not as simple as the ones described before, it is beyond the scope of this book.*

The expression in the RHS of the equation above may look pretty complicated. On the first look, people feel that evaluation of these expressions can be time consuming.

*We feel that the engineers need to remember the basic theory behind any topic and most importantly remember the results than spending huge efforts in proving them. We are not suggesting that we should be dumb to only apply the results, but we shouldn't be too engrossed only in the academics.

If we carefully look at the expression, we can see that it is a constant for various values of n.

So, we can make a lookup table for various values of n.

Number of Tasks	$U(n)$
1	1.000
2	0.828
3	0.779
4	0.756
∞	0.690

So, whenever we are required to do performance analysis, we can create a table like the one below:

i	C_i	T_i	$U_i = C_i / T_i$	$U_{i(\text{cum})}$	UB(i)
1	100	40	0.400	0.400	1.000
2	150	40	0.267	0.667	0.828
3	350	100	0.286	0.953	0.779

The data we have at the start is usually C_i and T_i only. The fourth column is U_i which defines the utilisation ratio of the task to the available CPU time. $U_{i(\text{cum})}$ as the name suggests is the cumulative utilisation of the tasks so far. For e.g., if we consider only the two tasks, the total utilisation is 0.667. $U_{i(\text{cum})}$ should not exceed the limit set by UB theorem. In this case, we see that by addition of the third task, $U_{i(\text{cum})}$ becomes greater than UB(i). So, these three tasks are not schedulable according to UB theorem.

We should remember that UBT is also a pessimistic test. (Actually, the set of three tasks are indeed schedulable). We may have to perform more tests (like the Response Time test) to ascertain if the set of tasks is schedulable. Readers are encouraged to delve deeper into realtime theory. (Some recommended books are given in the Bibliography section in the end).

6.4 LESSONS LEARNT

Real time systems can be classified in two ways. First based on their speed (fast and slow). Second, based on the criticality of their deadlines (hard and soft). There can even

be an overlap between these two classifications. Real time theory deals with assessing the schedulability of more than one concurrent tasks with given deadlines and priorities in such a way that all deadlines are strictly met. In this chapter, we used a simplified version of Rate Monotonic Analysis in order to arrive at a mathematical formula for their schedulability. The utilisation bound theorem in its simplest form, gives a formula of schedulability for independent periodic tasks.

6.5 REVIEW QUESTIONS

- How do we classify a system as hard-real time and fast-real time? Can both of them be true for a system at the same time?
- What is the definition of Rate Monotonic scheduling theorem? What are its basic assumptions?
- The example in the chapter assumes that C_i of task1 as 10. If we increase it to 40, will tasks still remain schedulable? Prove this using Rate Monotonic scheduling.
- What is scheduling theory?
- What is rate monotonic scheduling?
- According to Rate Monotonic Scheduling, which takes higher priority-Tasks with higher importance or higher rate?
- What are the three necessary conditions for a set of tasks to be schedulable?
- What is Utilisation Bound Theorem?

Chapter

7

Realtime Operating Systems

7.1 INTRODUCTION

The term realtime operating system (RTOS) usually triggers an image of a highly complicated OS in the minds of a programmer who is new to embedded systems. It is usually felt that only the experts can program with RTOS. This is totally untrue. In reality, an RTOS is not usually such a complex piece of software (in comparison to some of the mammoth size OSes' currently available). Though current RTOSes provide a huge variety of features, a basic RTOS is just small enough to provide some scheduling, memory management and a decent level of hardware abstraction.

7.2 DESKTOP OS vs RTOS

An RTOS should not be compared to a desktop OS.* A desktop OS is usually huge in size. It is not uncommon to see these OS' using 300–500 MB just for their installation. But, in embedded systems, even 8–16MB of memory is considered to be luxury. A desktop OS usually has huge libraries for UI management, support for numerous networking/interconnectivity protocols and fancy features like Plug 'n' Play. They also implement complex policies for network communication (like Microsoft NDIS) and facilities for binary reusability (DLLs, Shared Objects, COM/DCOM, .net, etc.).

An RTOS may not provide these features. But simply, every RTOS will provide at least the following features:

- Task Creation/Management
- Scheduling

*Nowadays, the embedded (not necessarily realtime) OSes are as big and complex as their desktop counterparts. For example, Windows XP is available as an 'embedded' version.

- Intertask communication/synchronisation
- Memory Management
- Timers
- Support for ISRs

Many good RTOSes also provide support for protocols like TCP/IP and some applications like telnet, tftp etc.

In a desktop development environment, a programmer opens his IDE/favourite editor*[1] and types in his code. Then, he builds it using his compiler and then executes his program. The point to be noted is that the OS is already running and it 'loads' the executable program. The program then makes uses of the OS services. The OS takes care of scheduling it (sometimes hanging in the process ☺). When the program completes, it exits. The OS can run other programs even while our program is running*[2]. The important point to observe is that programs have a definite exit point and programs that contain infinite loops are considered bad. The following diagram (Fig. 7.1) describes the steps during the lifetime of a desktop OS:

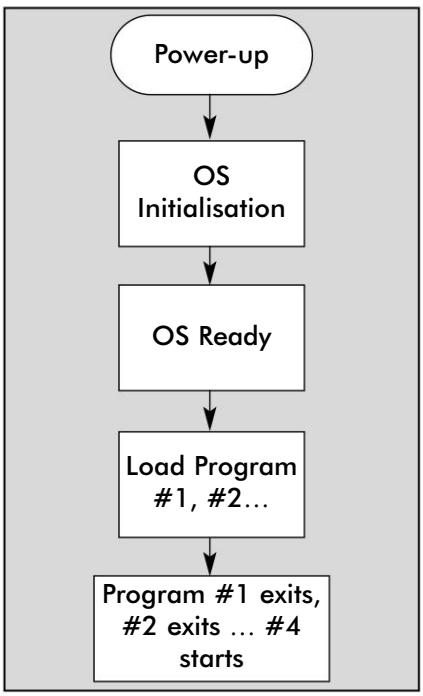


Fig. 7.1 *Operation of a desktop OS*

*[1]Some Unix guys still will swear by vi/vim.

*[2]Long ago, there was an OS called MS DOS that could run only one program.

But, this is not the case in most of the embedded systems.* Usually, there will be no dedicated OS already running on the target to load our program.

In an embedded system the software written by us, the RTOS (usually in the form of some library) and a special component called the BSP is bundled into a single executable file. This executable is burnt into a Boot-ROM or Flash.

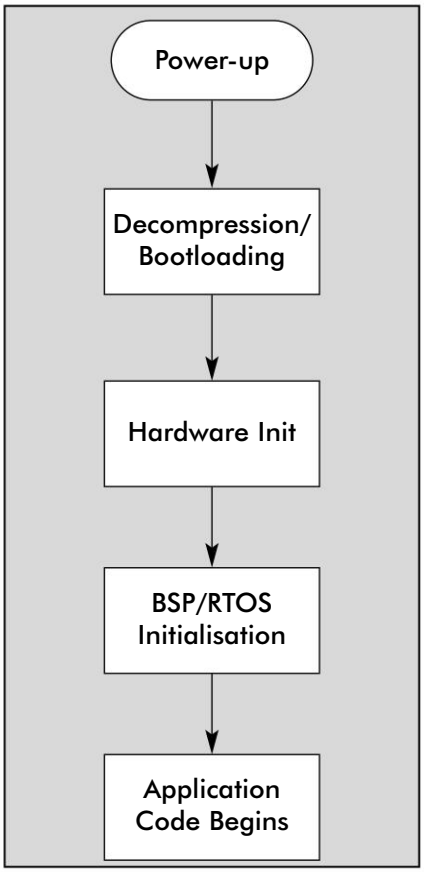


Fig. 7.2 *Operation in an RTOS Environment*

It is not necessary that a boot-ROM/Flash should always be present in the system. There could be cases where the code is obtained over a network on being powered up. This is because, usually embedded systems cannot afford the luxury of having hard disks. In the above flowchart, the decompression step is optional.

*With even Windows XP getting embedded, the barriers are slowly decreasing.

In this case, we can see that there is no OS during the startup. There is only a single executable that contains:

1. OS Code
2. Board Support Package (BSP)
3. Application code

BSP is a part of OS code in the sense that it is used by the OS to talk to different hardware in the board. We should remember that the software that we write is run on the target board.

Fig. 7.3 *A Typical development environment on the target*

The OS vendor has no idea of how the board is organised. He does not know the components of the board in most cases. (For e.g. what is the Ethernet controller? UART controller? etc.)

In some cases, the software is developed over some standard boards that can be bought off the shelf. If the board is quite common, then the OS vendors may provide BSP for those boards. But, if the board is custom built, we have to write a BSP for the board.

7.3 NEED FOR BSP IN EMBEDDED SYSTEMS

The OS needs to interact with the hardware on the board. For e.g. if we have a TCP/IP stack (provided by an OS vendor) integrated with our software, it should finally talk to the Ethernet controller on our board to finally put the data in the networking medium. The TCP/IP package has no clue about the Ethernet controller (say 3Com, Intel) and the location of the Ethernet controller and ways to program it. Similarly, if there is a debug agent that runs as a component in the OS that may want to use the serial port, then we should provide the details and the driver for the UART* controller used in our board.

*Universal Asynchronous Receiver/Transmitter.

Note

BSP is a component that is used to provide board/hardware specific details to the OS, for the OS to provide hardware abstractions to the tasks that use its services.

We should understand that the BSP is highly specific to both the board and the RTOS for which the BSP is written.

The BSP/startup code will be the first to be executed at the start of the system. The BSP code usually does the following:

- Initialisation of processor (Usually processors can operate in various modes. BSP code initialises the mode of operation of the processor. Then, it sets various parameters required by the processor.)
- Memory initialisation
- Clock setup
- Setting up of various components such as cache

Usually BSP can be classified into two parts:

- The first part does real low-level stuff like the one specified above. This is usually done in assembly language.
- The next part usually consists of drivers that are required by the RTOS to use some peripherals. (E.g. Ethernet driver, video, etc., UART).

Coding BSP is considered as 'real' embedded work since it requires mastery of both hardware and software. A BSP engineer must be able to code real optimised programs and he must be extremely capable with the hardware.

The BSP gets bundled with the software that gets written and hence, any increase in BSP code size will lead to increase in the size of the total software. (We should remember that in embedded systems there is only a single executable that includes the code written by the programmers, the RTOS and the BSP code). So, the BSP must be written in a way such that it provides all the required abstractions but nothing more.

Note

BSP engineers usually assume that the hardware is correct. But, these days when hardware as well as the pressures on the engineers gets complex by the day, the hardware too gets as buggy as software. (We are not beaming with pride when we say this. Some day we'll be able to produce cleaner implementation of hardware and software). So, it is a good idea to search for errata/known bugs section in hardware documentation rather than to waste inappropriate time in searching for the bug)

Now, let us see the various components of an RTOS and how these functionalities are achieved.

7.4 TASK MANAGEMENT

Task management in an RTOS consists of the following:

- Task creation
- Task scheduling
- Task deletion

7.4.1 Tasks

The concept of a task is fundamental to understanding an RTOS.

> A task is the *atomic unit* of execution that can be *scheduled* by an RTOS to use the *system resources*.

Analysis

'Atomic Unit'—A task is considered as an atomic unit because, any other entity smaller than a task cannot compete for system resources.

'Scheduled'—In a system, there could be many tasks competing for system resources. But, the duty of the RTOS is to schedule tasks such that requirements of tasks are met in such a way that the system objectives are met.*

'System Resources'—All tasks compete for resources like CPU, memory, input/output devices etc.

A task first needs to be '*created*'. A task is usually characterised by the following parameters: (sample parameters are given in brackets)

1. A task name ("TxTask")
2. Priority (100)
3. Stack size (0×4000)
4. OS specific options

*What this means is that, some task can be blocked for a long period of time and another can use the same resource for a long time. There is no question of fairness as long as the system objective is met. In an aeroplane, a task that controls the air-conditioning system is less important than a task that controls the flight.

These parameters are used to create a task. A typical call might look like as in Listing 7.1.

```
result = task_create ( "TxTask", // Task name
                       100,
                       0x4000,
                       OS_PREEMPTABLE);

if (result == OS_SUCCESS) {
    // task successfully created…
}
```

Listing 7.1: *Task creation*

Now, the task can be considered to be in an embryonic state (*Dormant*). It still does not have the code to execute. But by now, a task control block (TCB) would have been allocated by the RTOS.

7.4.2 Concept of control blocks

The use of control blocks is not limited to a task. During the course of this chapter, we would see various control blocks for memory, synchronisation, etc. The control blocks are internal to RTOS. There is absolutely no need for a programmer to access these blocks. In many cases, the access to these structures is restricted to the programmer. But, to comprehend the working of an RTOS, we need to know how these control blocks are used by the RTOS.

The RTOS usually reserves a portion of available memory for itself during startup. This chunk of memory is used to maintain structures such as the TCB. The TCB will usually consist of the state of the task, i.e. the value of registers when the task was pre-empted earlier, its priority and its RTOS specific parameters (say scheduling policy). When a task is blocked, the value of the registers is saved in its context in TCB. When a task is scheduled again, the system registers are restored with these saved values, so that the task will not know even if it was pre-empted. An RTOS uses the TCB of a task to store all the relevant information regarding the task.

7.4.3 Task states

At any point of time, a task can be in one of the following states:

1. Dormant
2. Ready

3. Running
4. Blocked

- ***Dormant*** When the task is created, but not yet added to RTOS for scheduling.
- ***Ready*** The task is ready to run. But, cannot do so currently because, a higher priority task is being executed.
- ***Running*** The task is currently using the CPU
- ***Blocked*** The task is waiting for some resource/input.

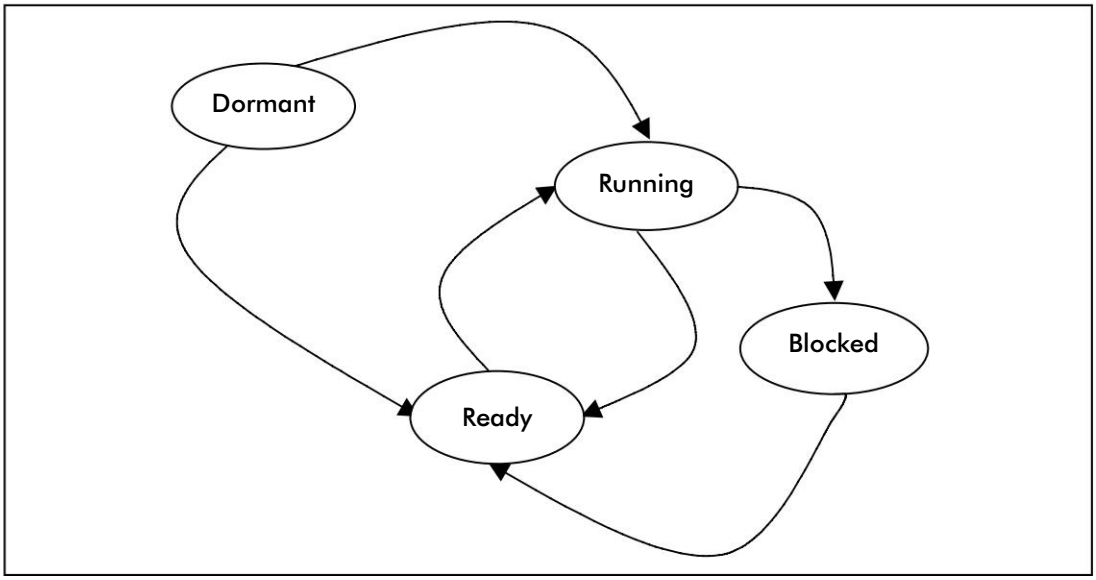


Fig. 7.4 *Task State Transition Diagram*

The stages of a task and its transitions are illustrated in Fig. 7.4.

i. When a task is created, it is in a 'Dormant' state

ii. When it is added to the RTOS for scheduling, it usually arrives in the ready state. But, if it is the highest priority task, it could begin executing right away.

iii. When a task is running and if another higher priority task becomes ready, the task that is running is pre-empted and the highest priority task is scheduled for execution.

iv. During the course of execution of a task, it may require a resource or input. In this case, if the resource/input is not immediately available, the task gets blocked.

Note

What happens if a lower priority task starts a higher priority task? It gets pre-empted immediately. So, the root task that creates all the tasks should typically have the highest priority and if it has any task higher than its own priority, it should be started last.

Other transitions

There are transition stages in the execution of the tasks. Some of them are as follows:

- *Ready to running*: For a particular task, if all the higher priority tasks are blocked, then the task is scheduled for execution. It then changes its state from ready to running.
- *Blocked to ready*: When a higher priority task releases a resource required by a lower priority task, then the lower priority task cannot begin execution. The higher priority task will continue to run. But, the state of the lower priority task will change from 'blocked' to 'ready'.
- *Blocked to running*: Sometimes, it could happen that a higher priority task is blocked on some resource/input. When that resource is available again, then the task begins execution, pre-empting the lower priority task that was executing.

P2P

Impossible transition: Of the transitions considered above, the transition "Ready to Blocked" is not possible. A task can block itself only when it is running. So, this transition is not possible.

7.4.4 The IDLE task

So far we have seen how a task pre-empts another. But what if no task is ready to run and all of them are blocked? The RTOS will be in deep trouble. So, an RTOS will usually execute a task called *idle** task. An idle task does nothing. A sample idle task code could look like:

*Nomenclature may vary across RTOS'.

```
void IdleTask (void)
{
    while(1);
}
```

Listing 7.2: *Idle task*

You can see that it has no system calls… in fact no code except an infinite loop. The idle task in an RTOS is the task with the lowest priority. And, many RTOS' reserve a few lowest and highest priority tasks for themselves. For e.g., if an RTOS can provide 256 tasks, it may reserve the lowest 10 and the highest 10, leaving the user with 226 tasks of priorities in the range (10–246).

CPU loading

Though an idle task does nothing, we can use it to determine the CPU loading—the average utilisation ratio of the CPU. This can be done by making the idle task writing the system clock in some memory location whenever it gets scheduled. So, an idle task need not be '*idle*' after all.

P2P

Is an ISR also a task?
The answer is no. A task is a standalone executable entity. An ISR is a routine that is called by system in response to an interrupt event. (However some newer RTOSes model ISRs as high priority threads are schedulable by the OS kernel).

7.4.5 Task scheduling

Task scheduling is one of the primary reasons for choosing an RTOS. The programmer can assign priorities to various tasks and rest assured that the RTOS would do the needed scheduling.

There are a huge variety of scheduling policies available. Some of the most used ones are described below:

Strictly pre-emptive scheduling

This is one of the most widely used scheduling policies in an RTOS. However, we should know that this is not a preferred scheme in a desktop OS. In this policy, at any instance of time, only the highest priority task that is ready to run executes. If a task T_i runs, it means that all tasks T_j with priorities lesser than T_i are blocked.

This scheduling policy makes sure that important*[1] tasks are handled first and the less important later. For e.g., in an aircraft cruise control system, the flight controller task will have more priority than a task that controls the air-conditioning system.

Pros:

Once the priorities are set properly, we can rest assured that only the important things are handled first.

Cons:

It is possible that one or more of the lower priority tasks do not get to execute at all. So, to avoid this, a proper analysis should be done in the design phase.

Note

There could be a slight deviation in the implementation of this scheduling algorithm. In almost all systems, ISRs will have highest priority irrespective of the priorities assigned to the tasks. This is usually necessary too. So, this deviation from the normal behaviour is acceptable.

Time slicing

In this kind of scheduling policy, the CPU time is shared between all the tasks. Each task gets a fraction of CPU time. There is no notion of priority here. This scheduling is also known as round robin scheduling. The scheduling is not used in its original form.

However, this can be used in conjunction with pre-emptive scheduling. In a pre-emptive system, if two or more tasks have same priority,*[2] we can make the scheduler use time slicing for those tasks with the same priority.

Pros:

i. No need for complex analysis of system
ii. This kind of kernel is relatively easily to implement
iii. The pre-emption time of a task is deterministic i.e. if a task is pre-empted, we will know exactly the time after which the task will be scheduled (if the number of tasks in the system do not vary with time)

---

*[1] i.e. higher priority.

*[2] Some RTOSes will not allow two tasks to have same priority while using a pre-emptive kernel.

Cons:

This is a very rigid scheduling policy (and exactly that is what it is meant to be — there is no notion of priority).

Fairness scheduling

In this kind of scheduling, every task is given an opportunity to execute. Unlike preemptive scheduling, in which a lower priority task may not get an opportunity to execute, in this case, every task will be given a '*fair*' chance to execute. Though some kind of priority mechanism could be incorporated here, it is not strict. Priority of a task, which has not executed for some period will gradually be increased by the RTOS and will finally get a chance to execute.

This scheduling policy is complex (how to vary priority of tasks in such a way as to achieve fairness?). And, it does not fit right in realtime systems.

This kind of scheduling is widely available in desktop OS'. (We still listen to music while compiling our programs).

Pros:

i. Every task will get an opportunity to execute

Cons:

i. Introduces nondeterminism into the system

7.4.6 Task synchronisation

We have so far talked about many tasks executing in the RTOS. In all but trivial systems, these tasks need to interact with each other—i.e. they must synchronise and communicate with each other.

We can illustrate this using an oft-used road example.

When the tasks are independent—i.e. there is no communication between the tasks, they do not share any resources* between them. This can be compared to two roads that run parallel to each other and hence, do not meet.

Vehicles can ply on these roads safely without colliding with ones in the parallel road. But the case becomes different when we have two intersecting roads. (Refer Fig. 7.6).

*Except the CPU of course!

In this case, we need an explicit mechanism like a traffic signal to make sure that the vehicles ply without getting into any mishaps.

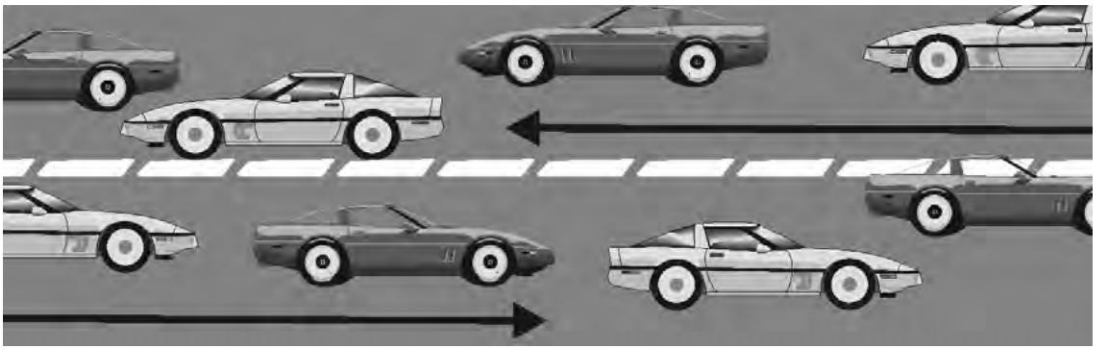


Fig. 7.5 *Two roads that run parallel to each other*

This situation is different because, there is a shared region between the roads. So, traffic on the two roads need explicit synchronisation. Another point to be observed is that traffic signal is required only at the region of intersection. There is no need for this synchronisation either before or after this region.

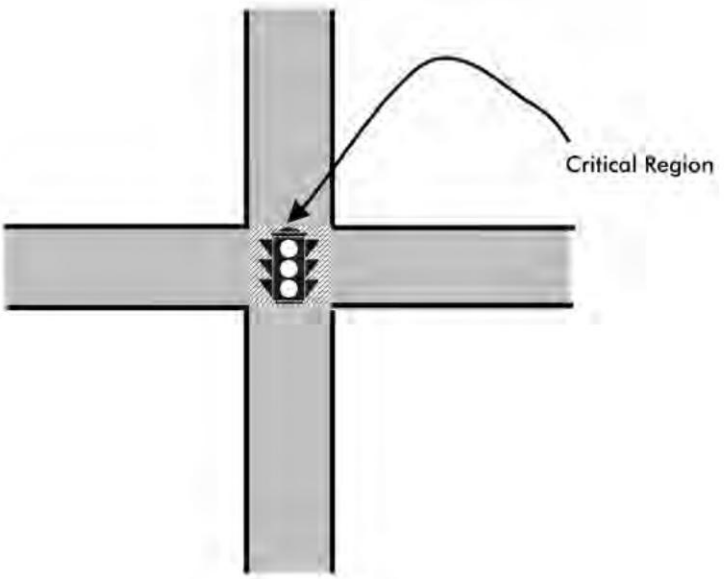


Fig. 7.6 *Two intersecting roads*

You'll be surprised by how this example translates perfectly to inter-task communication.

There are two ways of achieving synchronisation with the tasks:

- Task synchronisation using mutexes
- Task synchronisation using semaphores

Both the methods are discussed in detail below.

Task synchronisation using mutexes

In continuation of the discussion above, we can summarise the situation in the following points:

i. Problems occur only when a resource is shared among tasks.

ii. Synchronisation needs to be done only during resource acquisition and release.

For e.g., consider that two tasks want to share a printer. Let task *A* want to print the following sequence:

1 2 3

And, let task *B* want to print

A B C

If these tasks are scheduled in a round robin (time slicing) method, then the printout appearing on paper could be

1 2 A B 3 C (or any other junk)

It is also possible that the output is perfect because, the two tasks had sufficient time to queue their requests.

But, since such an error situation could arise, and can cause undesirable output from the system, this problem should be addressed. The immediate solution that occurs to mind is that one of the tasks can acquire the printer resource, use it and then release it.

To implement this solution, we need to use a mutex—a short and very common name for "Mutual Exclusion".

As the name indicates, it is a mechanism to exclude other tasks to use a resource when a specific task has acquired it.

For e.g., task *A* can be coded as:

```
// Task A code
// . . .
mutex_acquire( printer_mutex );
print( 1 );
print( 2 );
print( 3 );
mutex_release( printer_mutex );
```

Listing 7.3: *Task* A *Code*

Similarly, task *B* can be coded as

```
// Task B code
// . . .
mutex_acquire( printer_mutex );
print( 'A' );
print( 'B' );
print( 'C' );
mutex_release( printer_mutex );
```

Listing 7.4: *Task* B *Code*

At any point of time if both the tasks want to use the printer, they first try to acquire the mutex. Since, we are considering only a single processor model, the task, which makes the first attempt will acquire it.

Let us consider a case where task *A* has acquired the `printer_mutex`. (Refer Listing 7.5.)

```
// Task A code
// . . .
mutex_acquire( printer_mutex );
print( 1 );
print( 2 );
```

Listing 7.5: *Task* A *pre-emption*

Let us now consider that the task *B* has a higher priority and it gets scheduled after `print( 1 )`. And now, let task *B* also want to print something. It will now try to acquire the `printer_mutex`. But it cannot, since task *A* has already acquired the mutex.

```
// Task B code
// . . .
mutex_acquire( printer_mutex );   <-- Blocked here
print( 'A' );
print( 'B' );
```

Listing 7.6: *Task* B *pre-emption*

The task *B* will now be blocked. (It is not necessary that the task *B* be blocked. We say *B* is blocked on resource.)

Since task *B* is blocked, task *A* gets to resume again and completes its printing. It then releases the mutex. Now, task *B* can resume and continue with its printing.

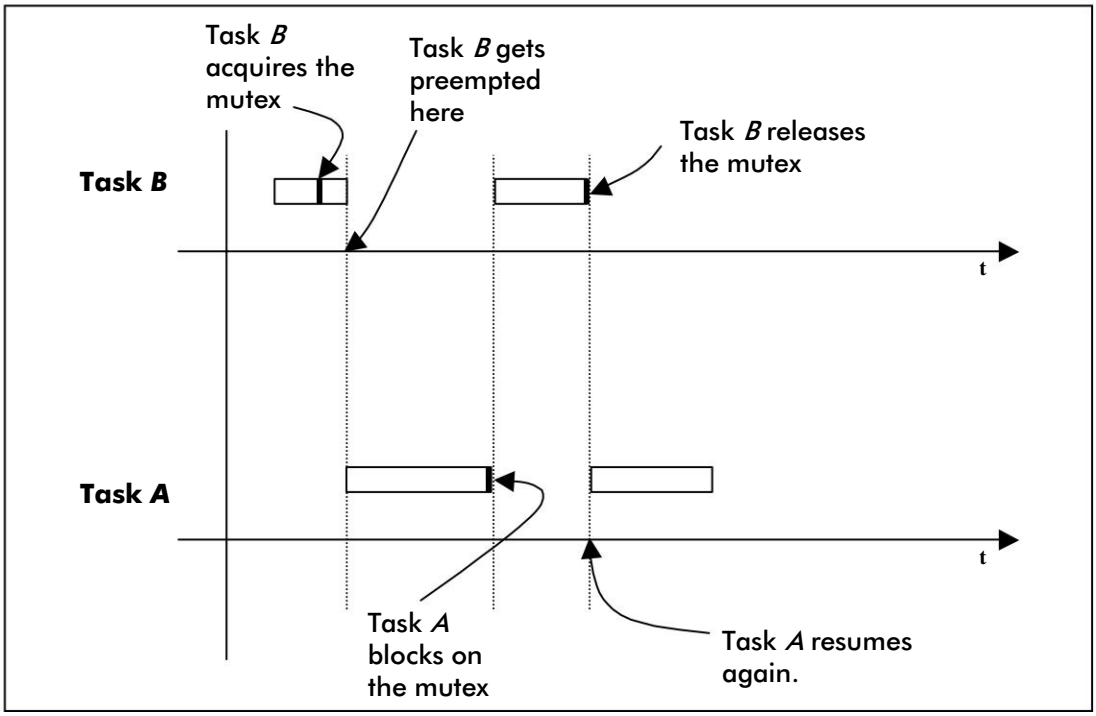


Fig. 7.7 *Task dynamics*

We should remember that since task *B* is a higher priority task, the execution would shift to task *B* *immediately* after task *A* releases the mutex.

Consider the following code of task *A*:

```
// . . .
print ( 3 );
mutex_release ( printer_mutex );
my_foo(); // some other function called from task A
```

Listing 7.7: *Mutex release by task* A

In a truly pre-emptive system, the execution will be transferred to task 'B' immediately after execution of `mutex_release`. Statement `my_foo` will be executed only after task A is scheduled again.

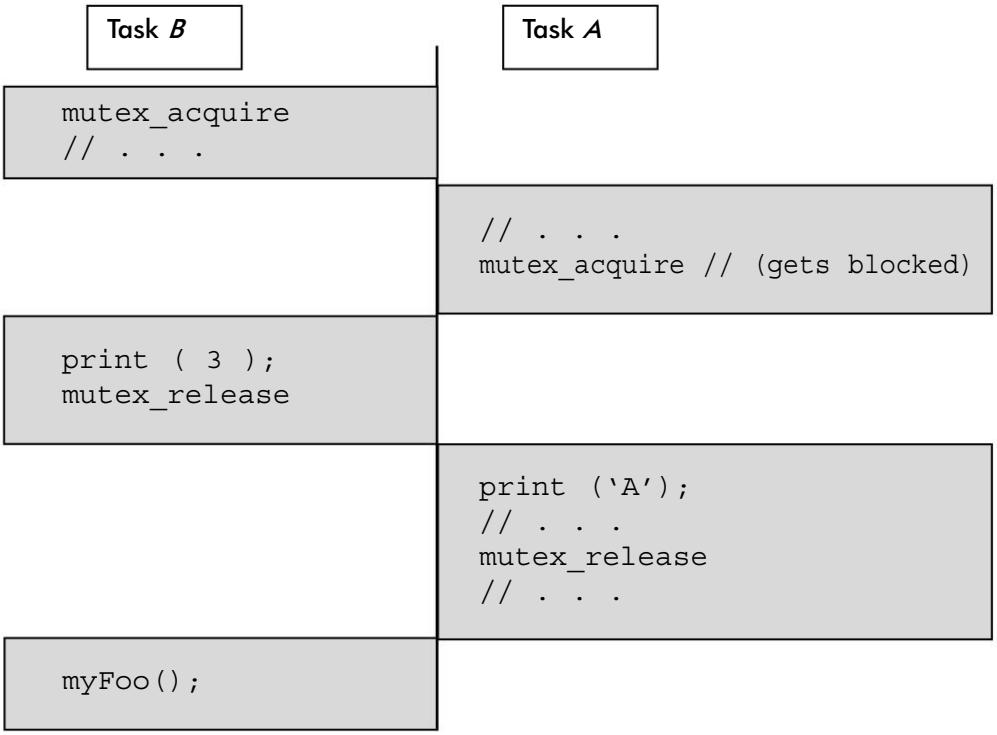


Fig. 7.8 *Transfer of execution between Task* A *and Task* B

The above figure describes how execution is transferred between tasks *A* and *B* during a sample run. The shaded boxes indicate the code that gets executed.

Pseudocode for the entire program is given in Listing 7.8.

```
#include <my_os.h>

// The printer mutex is a global variable so that both the
tasks
// can access it.
mutex printer_mutex;

int main()
{
        // . . .
  task_create ( TaskA );
  task_create ( TaskB );
  // . . .
}

void TaskA ( Params )
{
}

void TaskB ( Params )
{
}
```

Listing 7.8: *Source file for Task* A *and* B *with mutex sharing*

7.5 RACE CONDITIONS

Mutexes are also required when two tasks share data using global variables.

Let us consider a case where two tasks are writing into contiguous memory locations and another task uses these values produced by the two tasks. In concurrent programming parlance, the first two tasks that generate the values are called '*producers*' and the

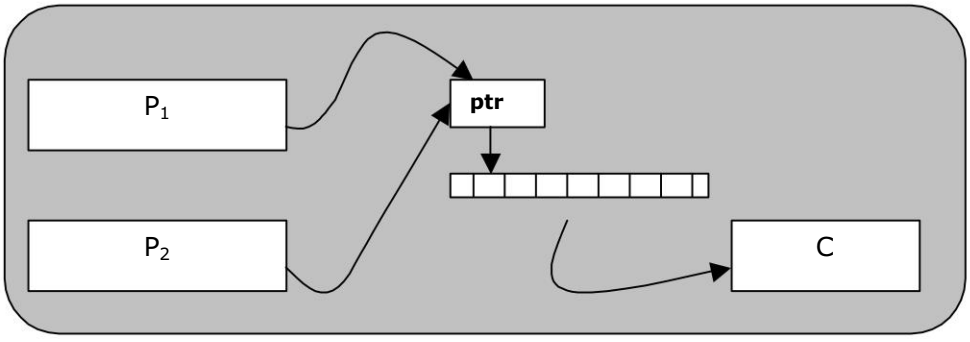


Fig. 7.9 *Two producers and a customer*

one that uses these values is called the '*consumer*'.*[1] (Actually, a task can be a producer and a consumer at the same time.)

Here, we see that the consumer task *C* reads and uses the values in list L. The pointer `ptr` is used to write into the list. To write, we can use `*ptr = 8;`

`ptr` is a global variable and both the tasks P_1 and P_2 can access it. Let us hypothetically assume that ptr points to memory location 0×4000.*[2] Consider the following situation:

P_1 reads the value of `ptr`. After P_1 reads the value of ptr from the memory, it gets pre-empted by P_2 that writes say `ptr = 12`; Now, the contents of memory location 0×4000 will be changed to 12. Before P_2 increments the pointer, task P_1 is scheduled again. Now, the contents of P_2 are lost when P_1 writes 8 into the same memory location.

We'll observe the behaviour as consumer task will not produce the output corresponding to the pointer value = 12. We should also note that this can occur only when the tasks are scheduled in the sequence described above. So, the problem will manifest itself only rarely, and most important, inconsistently.

This condition, where data consistency is lost because of lack of synchronisation of the component tasks of a system is called a '*race condition*'.

And, this belongs to one of the worst categories of bugs—nonrepeatable bugs. Even if the customer reports the problem, we cannot easily reproduce the problem in the lab.

Ice-cream and logic*[3]

A complaint was received by the Pontiac Division of General Motors:

"This is the second time I have written you, and I don't blame you for not answering me, because I kind of sounded crazy, but it is a fact that we have a tradition in our family of ice cream for dessert after dinner each night. But the kind of ice-cream varies so, every night, after we've eaten, the whole family votes on which kind of ice-cream we should have and I drive down to the store to get it. It's also a fact that I recently purchased a new Pontiac and since then my trips to the store have created a problem."

*[1]This can be roughly correlated to concepts of 'source' and 'sink'.

*[2]The number 4000 is used as a simple address rather than something like 0x4BC45D20, which would make our life horrible when writing it every time. ☺

*[3]This is quoted 'as is' from a listing in www.joke-archive.net

"You see, every time I buy vanilla ice-cream, when I start back from the store my car won't start. If I get any other kind of ice-cream, the car starts just fine. I want you to know I'm serious about this question, no matter how silly it sounds: 'What is there about a Pontiac that makes it not start when I get vanilla ice-cream, and easy to start whenever I get any other kind?'"

The Pontiac President was understandably skeptical about the letter, but sent an engineer to check it out anyway. The latter was surprised to be greeted by a successful, obviously well-educated man in a fine neighbourhood. He had arranged to meet the man just after dinner time, so the two hopped into the car and drove to the ice-cream store. It was vanilla ice-cream that night and, sure enough, after they came back to the car, it wouldn't start.

The engineer returned for three more nights. The first night, the man got chocolate. The car started. The second night, he got strawberry. The car started. The third night he ordered vanilla. The car failed to start.

Now the engineer, being a logical man, refused to believe that this man's car was allergic to vanilla ice-cream. He arranged, therefore, to continue his visits for as long as it took to solve the problem. And toward this end he began to take notes: he jotted down all sorts of data, time of day, type of gas used, time to drive back and forth, etc.

In a short time, he had a clue: the man took less time to buy vanilla than any other flavour. Why? The answer was in the layout of the store.

Vanilla, being the most popular flavour, was in a separate case at the front of the store for quick pick-up. All the other flavours were kept in the back of the store at a different counter where it took considerably longer to find the flavour and get checked out.

Now the question for the engineer was why the car wouldn't start when it took less time. Once time became the problem — not the vanilla ice-cream — the engineer quickly came up with the answer: vapour lock. It was happening every night, but the extra time taken to get the other flavours allowed the engine to cool down sufficiently to start. When the man got vanilla, the engine was still too hot for the vapour lock to dissipate.

Moral of the story: Even insane-looking problems are sometimes real.

To avoid this problem, shared global variables must be used only with synchronisation.

There is another interesting solution called the Peterson's solution described in the sidebar on keyword `volatile`.

7.6 PRIORITY INVERSION

Priority inversion is one of the issues that must be addressed during the analysis and design of realtime systems.

We discussed that, in a pre-emptive system, at any point of time, only the task with the highest priority executes. But, due to some reasons, if a higher priority task is

blocked because of some lower priority task, then a '*Priority Inversion*' is said to have occurred.

It can happen in two ways:

- Bounded priority inversion
- Unbounded priority inversion

Let us discuss them one-by-one.

7.6.1 Bounded priority inversion

Let us consider a system with two tasks *A* (T_A) and B (T_B). Let priority of T_A be higher than that of T_B.

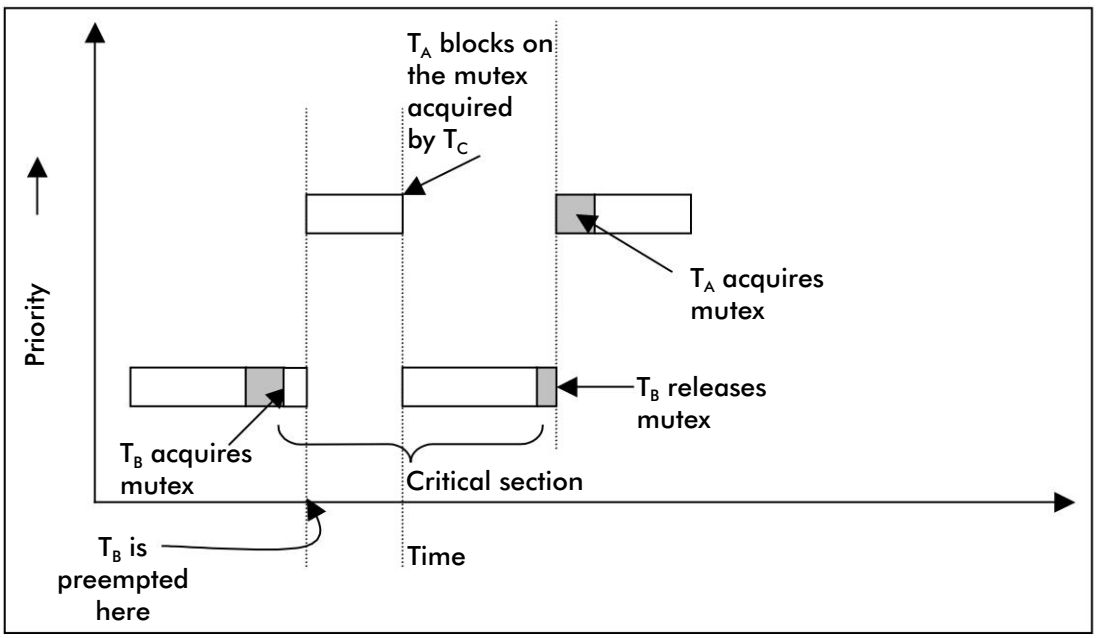


Fig. 7.10 ***Bounded priority inversion***

Initially, let T_A be executing and after sometime, T_A gets blocked and T_B scheduled. Now, let T_B acquire a mutex corresponding to a resource shared between T_A and T_B. After sometime, before T_B gets to finish its critical section code, T_A gets scheduled (since T_A's priority is higher).

After sometime, T_A tries to acquire the mutex for the resource shared between T_A and T_B. But, it cannot acquire the mutex because, it has already been acquired by T_B.

Because of this T_A is blocked. Now, T_B runs till it completes its critical section code and releases the mutex. Once the mutex is released, T_A begins execution.

Here, we see that T_A gets blocked for a period, because of lower priority than task T_B in acquiring a shared resource. So, in this case priority inversion is said to have occurred.

Now, let us look at the question, "How long is T_A blocked?" The answer is, in worst case, T_A will be blocked for the period equal to the critical section of T_B (i.e. if T_B is pre-empted immediately after acquiring the mutex)

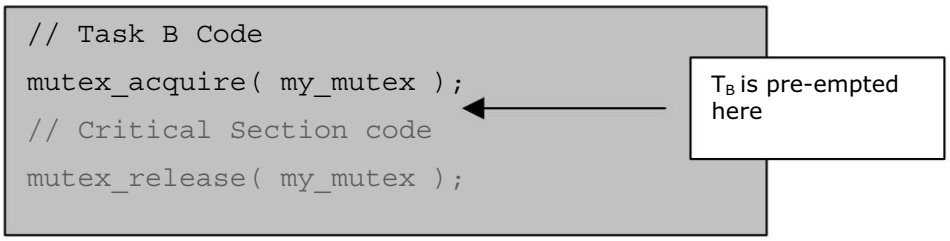


Fig. 7.11 *Task B pre-emption–worst case*

Here we see that the period for which the priority inversion occurs is '*bounded*'. The worst case is that the priority inversion occurs for the period equal to complete T_B critical section. So, this is called 'bounded priority inversion'.

In summary, a 'bounded priority inversion' is said to occur when a higher priority task is blocked for a deterministic period of time within a limit (bound).

7.6.2 Unbounded priority inversion

More dreaded than bounded priority inversion is the 'Unbounded Priority Inversion'. As the name suggests, it is a case when the time for which priority inversion is unbounded—i.e. we cannot fix how long the priority inversion will occur as we did in the previous case. This should be a serious cause of concern for system designers since the higher priority task will not be able to provide its services for a unknown period of time. This could cause failure of the entire system. A famous example is its occurrence during the Mars Rover exploration of Mars by NASA. (Explained later in the chapter).

To illustrate unbounded priority inversion, let us consider a system with three tasks.*

*A system with any number of tasks > 2 can be used. To make the illustration of unbounded priority inversion easier, a system with 3 tasks is considered.

Let the three tasks in the system be T_a, T_b, T_c in decreasing order of priority (T_a has highest priority)

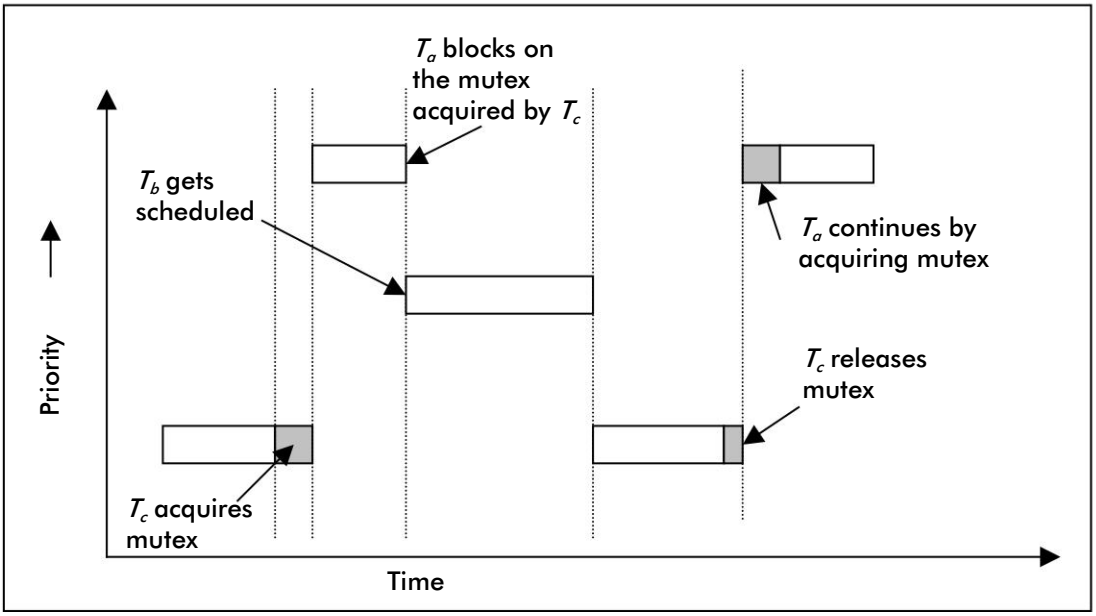


Fig. 7.12 *Unbounded priority inversion*

Initially, let us assume that the highest priority task (T_a) is running and gets blocked (Refer Figure 7.12). Now T_c starts running. (Assuming T_b is also blocked because of some reason). The task T_c acquires the mutex for the resource shared between T_a and T_c and enters the critical region.

Now, it gets pre-empted by T_b, which gets pre-empted again by task T_a.

After sometime, T_a tries to acquire the mutex for the shared resource. But, T_c had already taken the mutex. Once T_a gets blocked, T_b starts running. Now, T_c is still blocked and cannot release the mutex required.

Unlike the previous case, we cannot say how long it will be before the lower priority task releases the resource needed by higher priority task.

We'll have to wait for the intermediate priority task(s) to complete before the lower priority task will release the resource. So, this is called and 'unbounded priority inversion'.

7.6.3 Preventing priority inversion

To avoid occurrence of priority inversion, some schemes have been developed. The schemes aim at reducing the unbounded priority inversion to a bounded one. Two prominent schemes are:

i. Priority Inheritance Protocol (PIP)
ii. Priority Ceiling Protocol (PCP)

Priority Inheritance Protocol (PIP)

In this protocol, the priority of a task using a shared resource shall be made equal to the priority of the highest priority task that is blocked for the resource at the current instant. This is done so that the priority of a lower priority task is boosted in such a way that the priority inversion gets bounded.

Illustration: Consider an RTOS where 1 is the highest priority. Let 10 tasks, with priorities 1–10* execute in the system. Let us consider that the tasks with priority 2 and 7 share a resource R.

Let T_7 (lower priority) acquire the shared resource R and get pre-empted by task T_5 before releasing it. Later, let T_2 get scheduled and get blocked for the resource R. Immediately priority of T_7 is boosted to 2. Now, T_7 with boosted priority will be able to complete its critical section and release the mutex. Once the mutex is released, its previous priority is restored so that the actual high priority task can continue its execution and enter its critical section.

As we can see, the priority inheritance protocol requires the support of the RTOS to be implemented.

Priority Ceiling Protocol (PCP)

Here, we associate a priority with the resource during design time. During runtime, any task, which wants to use the resource will acquire the priority associated with the resource. The priority associated with the resource is the priority of the highest priority task associated with the resource.

Illustration: Let us assume that the system has three tasks and two resources that are shared. Like the above example, let us assume 1 is the highest priority in the system. Let tasks T_1, T_2, T_3 have priorities 1, 2 and 3 respectively.

Then we can form a table mapping resources to the tasks.

*It is not necessary that the range of priorities must be 1–1 0. Usually, it is not desired because, we cannot introduce a new task without affecting the priority of other tasks.

Resource	Sharing tasks	Priority*
R_1	T_1, T_2, T_3	1
R_2	T_2, T_3	2

*Priority associated with the resource

Table 7.1: ***Priority ceiling protocol — sample table***

Any task that wants to use R_1, then it has to do the following

```
oldPriority = task_getPriority();
           // This call returns the priority
          // of the current task

newPriority = R1_PRIORITY;
task_setPriority ( newPriority );
  // Now, use the resource
task_setPriority ( oldPriority );
```

Listing 7.9: *PCP without OS support*

Say, if T_3 wants to use R_1, it sets its priority to 1 and then access the resource. Now, task T_1 cannot pre-empt T_3 because its priority has increased to 1. After using the resource, the task restores its own priority.

The advantage of this system is that it does not require any explicit support from the RTOS. Another significant change is that there is no mutex/semaphore is required. We just use the priority changing mechanism provided by the RTOS.

However, many RTOS' help by adding support for PCP in the mutexes. Whenever a mutex is created, it is associated with a corresponding priority. Any task that takes the mutex will take up the priority associated with the mutex. The greatest advantage of this is that the boosting and restoring of priority is automated.

This method has the following disadvantages:

i. This method is manual i.e. the priority is associated with the resource manually. So, in case of large systems, maintaining priorities associated with resources can be error prone.

ii. Manual set/reset of priorities: In its original form (i.e. without mutexes), after using the resource, if tasks do not reset their priority, it could cause havoc. In

this aspect, using PCP provided by RTOS with mutexes is a preferable way of using PCP.

iii. Time-slicing not allowed: While using PCP, we have to adopt only a strict pre-emptive scheduling policy. PCP will fail if we mix pre-emptive and time-slicing.

It is now up to the designers to use either or none or both of the protocols appropriately for their system after weighing the benefits of all the schemes.

Mars Rover

The concepts like priority inversion and preventive measures described earlier are not limited to academic research. They can and should be applied to solve practical issues. Importance of these analysis, once believed to be a cosmetic addition to analysis of systems, came to light during the Mars Rover failure. Mars Rover was a project by NASA to explore the surface of MARS. It consisted of two major blocks—landing software and land mission software. Landing software was very critical because, any fault in this part would make the rover crash on the mars surface. The land mission software was used by the controller to analyse the environment in Mars, collect data and transmit them back to earth.

The landing of the Rover was perfect. But, later, during the execution of the land mission software, the system started resetting itself mysteriously. So, data could not be collected and sent to earth.

Usually, there are two versions of any software—the debug and release versions. The debug versions are used during the test phases of the software. The executable in this case is usually big because, it contains debug information also. After testing and the software is declared bug-free (!), a release version is made. This version is much leaner than the debug version, but contains no information for the programmer if the system fails.

Fortunately, the land mission software in the Rover was the debug version. The RTOS used was VxWorks™, which offered features like saving the collection of events till the system was reset. After a long analysis the bug was found. (Actually, it seems that it was not possible to reproduce the bug on earth. Engineers had left for the day and one engineer had stayed back and he could reproduce the problem).

There was an information bus, which was to be used by a high priority task. Another low priority task also required the bus. (The bus was a shared resource). Hence, a mutex was used to implement task synchronisation. Whenever the reset occurred, the lower

priority task had acquired the mutex and was later pre-empted by the higher priority task. But, it could not use the bus because, the mutex was acquired by the lower priority task. In between, an intermediate priority task used to run pre-empting the lower priority task. So, the higher priority task could not get the mutex.

Meantime, the system had a watchdog timer. As indicated before, a watchdog timer is used to reset a system if '*hangs*' for sometime. Here, the watchdog timer noticed that the high priority task could not access the bus for a long time and hence reset the entire system.

Mutexes, when being created were created as '*plain vanilla*' mutexes. So, it was decided to enable PCP feature of the mutex. Then, remotely, using one of the debug support feature of VxWorks, the mutexes when being created, the PCP flag was set to true instead of false, and the problem never happened again.

So, realtime theory finds practical uses too.

7.6.4 Task synchronization using semaphores

Semaphores are another mechanism similar to mutexes that can be used to achieve synchronisation of tasks and resources.

The concept of semaphores was invented by Edgar Dijkstra (1930–2002). This is explained in his paper (http://www.cs.utexas.edu/users/EWD/welcome.html). Unfortunately, the document is in Dutch.

A semaphore can be used where there is more than one resource of the same type. (Say there are 4 pumps in a reservoir).

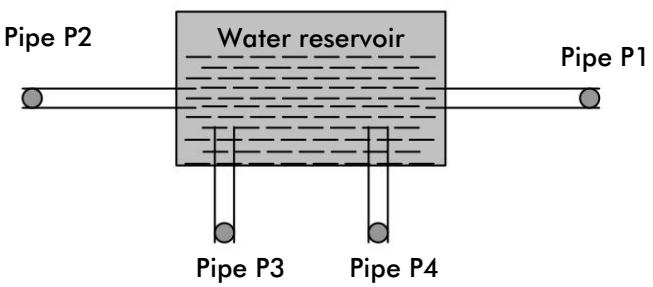


Fig. 7.13 *A Reservoir with 4 Pumps*

When a semaphore is created, it is associated with a specific number of tokens. (In the pump example above, a semaphore with 4 token needs to be created). Once a semaphore is created, tasks can request a specific number of tokens (less than the

maximum number of tokens) from the semaphore. When the requested number of tokens are available, the number of tokens in the semaphore is decremented, and given to the task. If the number of tokens exceed the number of available tokens, the task blocks. (Some RTOS' can provide the options of blocking, non-blocking or blocking with a timeout). Once some other task releases the tokens, the blocking task is released.

Types of Semaphores: If more tasks are blocking on a semaphore, and some tokens are released, which task must acquire the tokens? The answer is that the RTOS usually provides options to configure this.

There could be a FIFO (First In First Out) arrangement where the task that first blocked on the semaphore is given the tokens. If the semaphore is created with priority options set, the tokens are acquired by the blocking task with the highest priority.

Illustration: Consider a system with 3 tasks. Let us also assume it is the system which tries to control the 4 pumps in the reservoir. Let a task then acquire 3 pumps (tokens).

```
semaphore_acquire( pump_semaphore, 3 ); // Task B
```

If some other task requires 2 tokens, it will try to acquire it by

```
semaphore_acquire( pump_semaphore, 2 ); // Task C
```

But, since the required number of tokens is not available, the task will block. (As indicated with mutexes, a task can also choose not to block or to block with a timeout). Let us assume task *A* also requires 3 tokens.

```
semaphore_acquire( pump_semaphore, 2 ); // Task A
```

Let task *B* execute, complete its critical section and release its three tokens.

```
semaphore_release( pump_semaphore, 2 ); // Task C
```

Now, if the pump semaphore was a FIFO semaphore, then, task *C* will acquire the two tokens since it had blocked first on the semaphore. But, if the pump semaphore was a priority semaphore, then, task *A* would have acquired its 3 tokens, since it has a priority higher than that of *C*.

Binary Semaphores: We know that a semaphore can take on any positive integral value (including 0) at any point of time. If a semaphore is created with maximum value = 1, then its value will toggle between 0 and 1.

Hence, such a semaphore is called a binary semaphore. A binary semaphore has properties identical to mutex. Such a semaphore can be used where no explicit mutex feature is provided by the RTOS.

The story of semaphores

Dijkstra was once thinking hard about task synchronisation—about tasks synchronising before entering their critical sections.* There was a railway station in front of his house, where he could see lots of trains waiting for their signals.

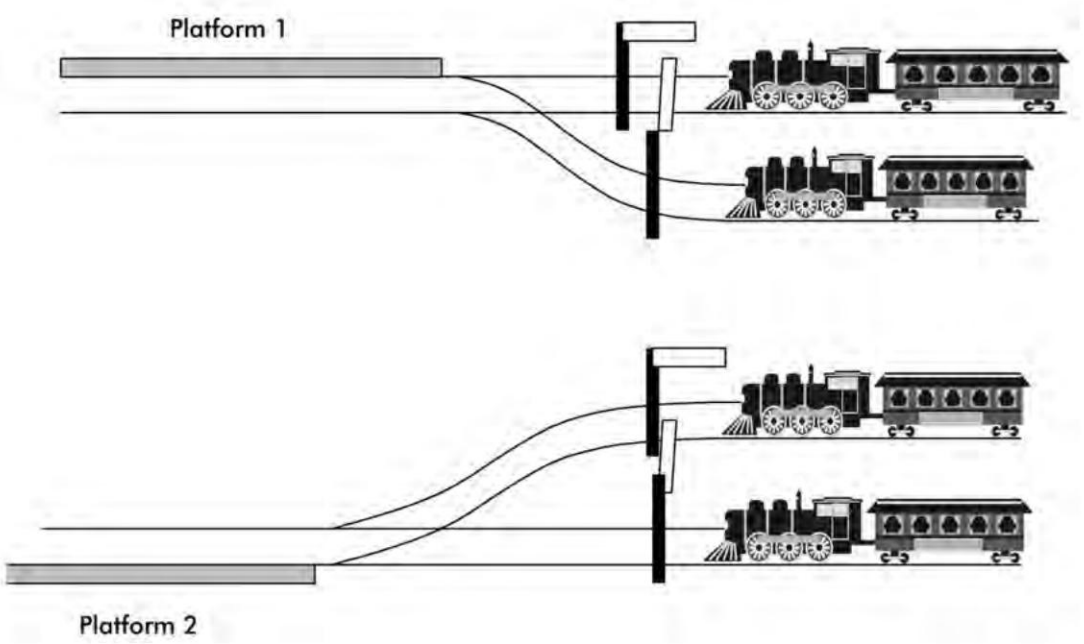


Fig. 7.14 *Railway Station*

(Obviously there were more trains than there were platforms). He saw that signals were used to control the movement of trains.

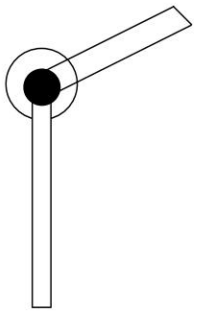


Fig. 7.15 *Train can proceed*

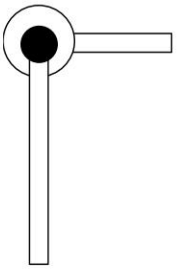


Fig. 7.16 *Train should wait*

*With a little dramatisation. ☺

Now, Dijkstra could see that the station/platforms were similar to shared resources and the trains were similar to tasks. The trains wanted to use the platforms as the tasks wanted to use the shared resources. The one thing missing in RTOS was the signal pole. So, he added it to the software system. He first used the term '*seinpaal*' (signal-pole in Dutch). When applied specifically in parlance of trains, it becomes' 'semafoor'* in Dutch.

Nowadays we use terms like acquiring and releasing semaphores (`semaphore_acquire, semaphore_release`), older documentation (even some newer ones) will have APIs like `semaphore_p, semaphore_v` (or P(s), V(s)). Almost every OS book will define P and V operations on a semaphore. The letter 'p' is taken from Dutch word 'proberen' which means 'try to' (acquire the semaphore). The letter 'v' is taken from the Dutch word 'vrijgeven', which means 'release' (the semaphore).

We guess that the complex Dutch words were reduced to one-letter acronyms by distraught non-Dutch programmers to avoid typing complete Dutch words.

7.7 RTOS—UNDER THE HOOD

We hope you are not surprised to suddenly come across theory on how RTOS work, midway through the chapter. We had earlier discussed the differences between a desktop OS and an RTOS. There is no separate OS running that can be used to load and execute programs. Also, there is only a single executable file that includes both the application code and OS code.

The working of RTOS is a mystery for novice programmers. In this section we discuss how scheduling works in an RTOS.

As discussed in the 'Build Process' chapter, any program can be split into two components:

i. Code

ii. Data

Code is non-modifiable, i.e. contents of the text section do not change and the contents of the data section of the program keep changing as the system is running. Initially, the entire application is stored in the ROM/Flash.

*Truly, no prizes will be given for guessing semaphore is derived from semafoor.

After a program is setup and is executing, the lower memory sections are usually required by the RTOS/hardware for board specific software and interrupt vector table entries. It is followed by the code section. Since the code is non-modifiable, we can choose to keep it in ROM (if not compressed and slow access time of ROM is not a problem).

Each task then requires a stack to create its local variables and store the arguments passed to functions. Each task is associated with its own stack of memory.

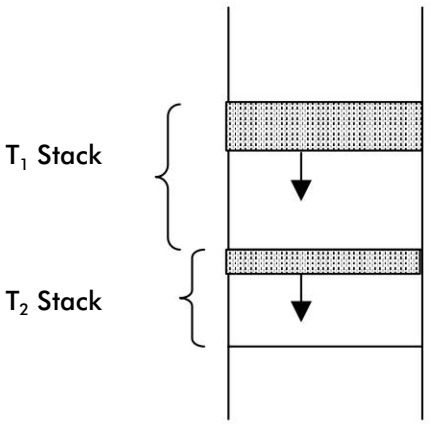


Fig. 7.17 ***Stack grows downwards***

Note that the stack grows downwards. In basic RTOS', there will be no memory protections. Unix programmers will be familiar with *core* dumps whenever there is an illegal memory access. But, an RTOS does not have such features.* In some cases, stack of a task can grow beyond its limits. For e.g., if stack size of T_1 is insufficient, the stack may overflow into T_2 stack area. This will corrupt the values stored by task T_2 in its stack. We'll never know that this had happened until the system does something interestingly out of the normal. ☺

So far, we saw how tasks come to life and execute. The next part is to know how the OS works. First, let us consider a strictly pre-emptive system. (And, let us restrict the discussion only to a uniprocessor system). Let us also consider that some task is running.

*Statutory Warning: In computing parlance it is true that 'If it works, it is obsolete'. We recently saw a demo of an RTOS will all these features. So, this statement can become antiquated. (We are bored of telling, 'usually does not have', and 'typically does not happen' to avoid prying eyes of our critics.

OS is scheduled only when a task executes a system call. System calls are the RTOS API that get linked with the application for task creation, semaphore/mutex operations, etc.

In a strictly pre-emptive system, a task can change its state (running, ready, blocked) only after executing a system call. The only way a running task can block itself is by executing a system call typically of a semaphore/mutex operation.

For e.g., if a task is requesting a semaphore or a mutex that is not available, then a task could block. If a low priority task releases a semaphore on which a higher priority task is blocked, then, the lower priority task will be blocked.

Conceptually, a semaphore release implementation could be

```
semaphore_release( int semaphore_id, int tokens )
{
  // get corresponding semaphore control block
  semphore_control_block* scb = sm_list[semaphore_id];

  // increment available number of tokens
  scb->tokens += tokens;

  release_tokens_if_someoneelse_is_blocking();

  invoke_scheduler();
}
```

Fig. 7.18: *RTOS Scheduler invoked from a System Call*

The invoke scheduler function should conceptually be a part of every system call. This call should check if any task needs to be scheduled by updating and checking the state of all tasks. If say, some other task needs to be scheduled, then the scheduler will save the context (registers) of the task to its TCB, restores the context of the task that needs to be scheduled from its corresponding TCB and schedules the new task.

7.8 ISRs AND SCHEDULING

Another way scheduling a task is by raising an interrupt when the corresponding ISR executes a system call. For e.g. if some data arrives in the system and the ISR posts the data (using system calls) to a higher priority task, then, after the ISR completes, the higher priority task will be scheduled.

The important points to be noted are as follows:

- Task switching does not happen immediately after the ISR executes a system call. It happens only after the ISR completes.
- Once the ISR completes, it need not return to the task that was running when the ISR was triggered.

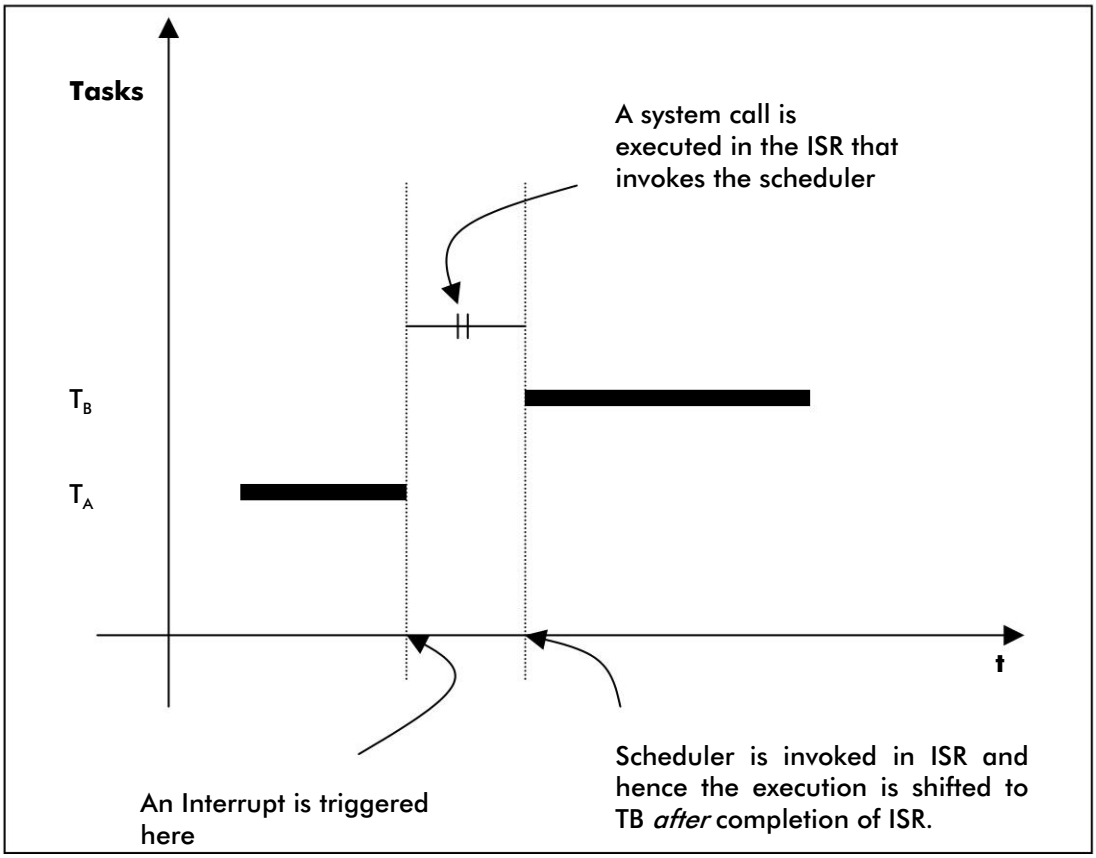


Fig. 7.19 ***Interrupts and Scheduling***

Another way of scheduling is time slicing. Here, the OS initialises an ISR called the '*tick*' ISR that gets executed whenever a tick (a fixed period of time) occurs. The scheduler could be invoked in the tick ISR. Usually any RTOS will initialise a tick routine for implementing RTOS timers. This tick ISR can also be used for scheduling.

7.9 INTER-TASK COMMUNICATION

So far, we have seen how different tasks synchronise to access shared resources. This section describes how these tasks communicate. Two mechanisms that are provided by various RTOS' are

- Message Queues
- Signals/Events

In addition, we will discuss the following mechanisms that can be used in specific situations (not recommended normally!):

- Function calls
- Accessing of variables

Message queues are used to pass data, while signals/events are used to signal other tasks.

7.9.1 Message Queues

Message queues are FIFO structures.

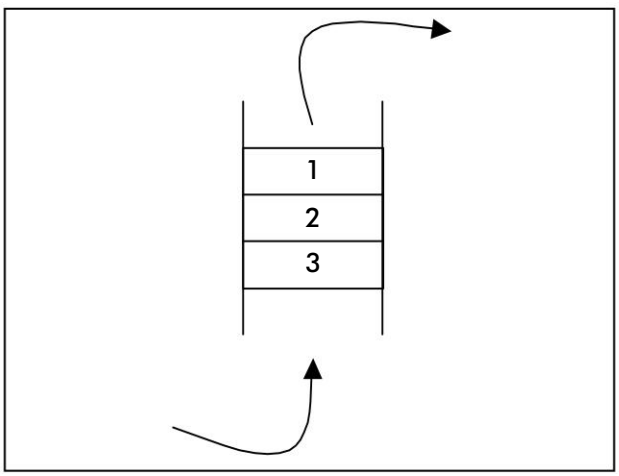


Fig. 7.20 *Queue—A FIFO structure*

These act as buffers between two tasks. A task need not consume data passed by another immediately—i.e. the processing need not be synchronous. (Whenever a synchronous processing is required, it is better to use a function call mechanism).

To use a message queue, the first step is to create one. Creation of a queue returns a queue ID. So, if any task wants to post some message to a task, it should use its queue ID.

```
qid = queue_create ( "MyQueue", // Some queue name
                     QUEUE_OPTIONS ); // OS Specific Options
```

Each queue can usually be configured as fixed size/variable size. Most RTOS will provide at the least, fixed size entries.

Though the queue post operation varies in methods across RTOS', the idea is usually to pass a pointer to some data. This pointer will point to a structure previously agreed between the tasks that use the queue.

Say there are two tasks that try to communicate using queues. The transmit task want to send a message `my_message` to the receiving task.

```
typedef struct queue_message {
    int       msg_id;
    void*     pMessage;
} queue_message;

typedef struct return my_message {
    int  a;
    char c;
} my_message;
```

Listing 7.10: ***Declarations to be common between transmitter and the sender***

The above declarations are common to both the transmitting and receiving task. So, these are put in a common header file to be included in both files implementing both the tasks.

```
// Sender side

queue_message* qmsg;
my_message * rmsg;

qmsg = (queue_message*) malloc(queue_message );
rmsg = (my_message *) malloc(my_message);

rmsg->a = 2;
```

contd…

contd...

```
    rmsg->c = 'r';

    qmsg->msg_id = RX_MESSAGE;
    qmsg->pMessage = (void*) rmsg;

    queue_send (qid, qmsg);
```

Listing 7.11: *Code for the sending (transmitting) task*

```
// Receiver side

queue_message* qmsg;
my_message * rmsg;

queue_receive (qid, qmsg); // block here for message

if(qmsg->msg_id == RX_MESSAGE)
rmsg = (my_message *) qmsg->pMessage;

// . . .

free( qmsg );
free( rmsg );
```

Listing 7.12: *Code for the receiving side*

On the transmitting side, we can see that, the transmitting task allocates memory for the queue message and the message to be transmitted. It then passes the pointer to `my_message` in the `pMessage` field of the queue message structure.

On the receiving side, the queue message pointer is passed and the receiver extracts the message from the `pMessage` field based on the message ID. It should be noted that there could be many messages passed between two tasks and message ID is one of the common ways to distinguish between the messages.

The usual programming practice is to use structures to pass data between two tasks even if it is a single character. Because, in future, if we want to pass more data, we can add them on to the structure and the interface (i.e. pointer to the structure) will remain the same.

FIFO/Priority queues

We learnt that queues are FIFO. But these FIFO or priority queue options can be used if multiple tasks are blocking on a single queue and a message is posted in the queue.

The queue could be configured to deliver the task that blocked the queue first or the task that has the highest priority. In the former case, it is called a FIFO queue and a priority queue in the latter.

Some RTOS' also provide some operations to add messages to the queue in the order of priority.

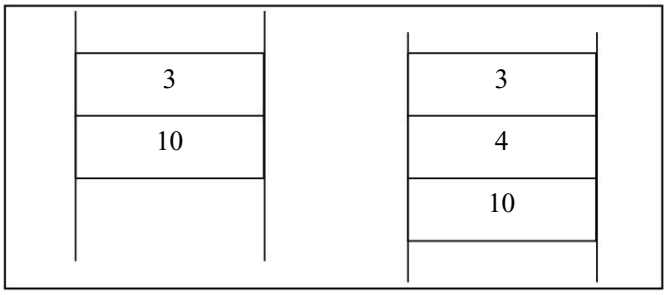


Fig. 7.21: ***Addition of messages to a priority based queue***

Some provide operations to insert some (say, an important) message to the top of the queue.

Using queues

It is a good practice to associate a queue with a task. This queue can then be addressed through its unique mailbox id. This is illustrated in the following diagram:

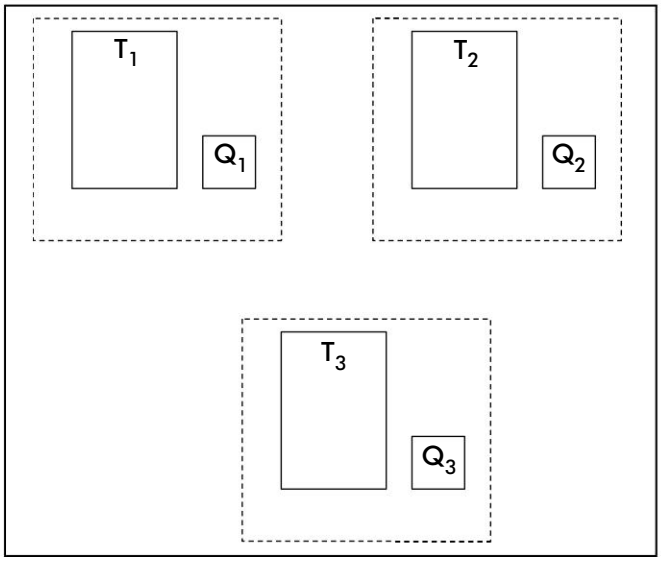


Fig. 7.22 ***Each task associated with a queue***

Whenever a message needs to be posted to a task, it should be posted to the corresponding queue (say Q_{T1} for T_1). When we create multiple queues for the same task, we might end up blocking inappropriately.

For e.g., consider a case where there are two queues, Q_1 and Q_2 associated with a task T_A.

```
// Task A code
queue_receive (qid_1, pMessage1);
queue_receive (qid_2, pMessage2);
```

Let us assume Q_1 is empty and T_A blocks on it. Meanwhile if some message is posted to Q_2, it will remain unattended till some message is posted in Q_1. Unix programmers usually have features like select call for a process that blocks multiple tasks. There are no such mechanisms in most RTOSes. So, it is best to avoid these kinds of constructs.

We burnt our fingers once during one of our early projects…

We were to design and implement a task as a part of a networking protocol subsystem. This task that was supposed to receive packets from a lower layer (let's call it LL) and pass it to Upper layer (UL) after some processing. We could also receive some packets to be transmitted to lower layer.

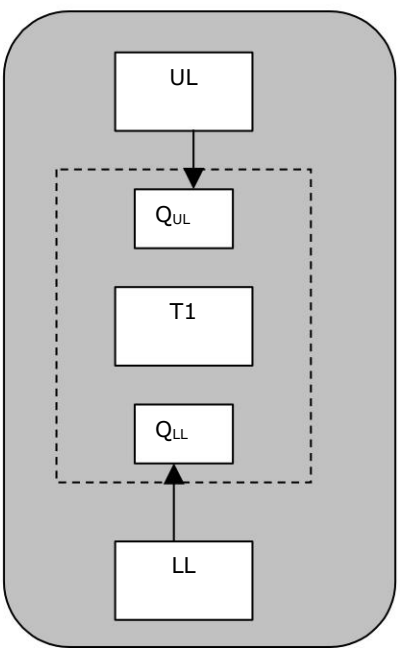


Fig. 7.23 *Model of two input queues per task*

We created two queues such that the messages from UL would go to Q_{UL} and the messages from LL would go to Q_{LL}. (Refer Fig. 7.23) We were then struck by the problem described above. When the task was blocking on Q_{LL}, messages posted to Q_{UL} could not be attended to.

So, we removed one of the queues and redesigned in such a way that both the tasks posted messages to the same queue (Refer Fig. 7.24). The differentiation between the messages was achieved using the message ID mechanism described earlier.

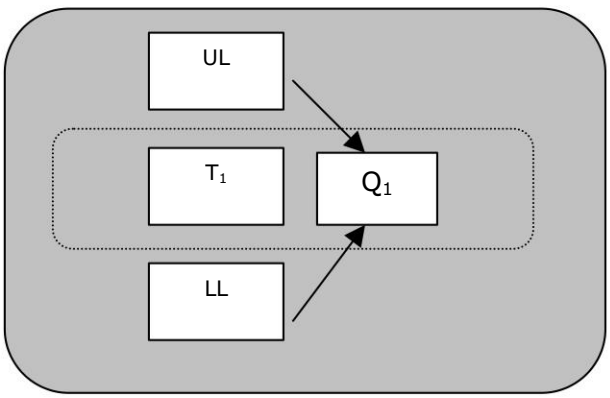


Fig. 7.24 *Model of one input queue per task*

But, ideally speaking, the design could have been such that, there are two tasks that do the interfacing between the two layers. It is better to assign different actions that could be concurrent to different tasks. However, it is up to the discretion of the system designer, based on the system and software constraints.

7.9.2 Events

Events, as mentioned earlier, are also known as signals. These cannot be used to pass data between tasks, but can be used to signal occurrence of some activity to another task. Events are supported by many RTOS'. An event is encoded as an integer. Let us see how events can be used. Consider a machine where an integer is 32 bits long. Events can only be used between tasks that have mutually agreed upon using events. Otherwise, miscommunication could occur.

B5	B4	B3	B2	B1	B0
0	1	0	0	0	1

Fig. 7.25 *A sample event flag*

The last 6 bits of an event flag is given in the above diagram. (Fig. 7.25)

Let us consider two tasks T_1 and T_2 that want to communicate using events. The first step is that they have to agree upon the events that they'll use to communicate. The events are integers with only one of their bits set to 1.

The values of numbers that can be used as events are numbers that can be expressed in 2^n where $n = 0$ (or more as restricted by the machine word size). Let one of the tasks be a producer task that signals that data has arrived and let the consumer task indicate that it has completed its processing. So, the two events that can be agreed upon can be

```
#define MY_PROJECT_DATA_READY                (0x01 << 0)
#define MY_PROJECT_PROCESSING_COMPLETE       (0x01 << 1)
```

Moreover, if the consumer is overloaded, it can ask the producer to wait before it produces more data. And, the consumer can indicate that it has sent all the data it wants to send:

```
#define MY_PROJECT_WAIT                      (0x01 << 2)
#define MY_PROJECT_DATA_COMPLETE             (0x01 << 2)
```

The values assigned to the above three symbolic constants are 1, 2 and 4 respectively. Now, the consumer task has to wait for the event

```
event_receive( MY_PROJECT_DATA_READY | MY_PROJECT_DATA_COMPLETE,
               ANY_EVENT, // Some OS specific flag
               &received_events);
```

In the above example, the first argument is the events that the consumer task must wait on. In this case, it waits for signal DATA_READY and DATA_COMPLETE. The second argument is an OS specific flag. In some OS' it is possible to block the occurrence of any event or occurrence of all events. The third argument is that the events have arrived in a bit encoded form. The third argument is used to find out which event has occurred. Typical usage will be

```
if (received_events & MY_PROJECT_DATA_READY) {
    // Means data is ready. So, process the data
}
else if (received_events & MY_PROJECT_DATA_COMPLETE)
     // Data reception is complete. Cleanup or do something else
}
```

Note the use of bitwise ORing (| operator) for combining events and bitwise ANDing (& operator) for checking of individual events.*

*Check Chapter 10 for implementation aspects in embedded systems.

Similarly, some RTOS' also give an option to block with time-outs to avoid conditions where the system hangs if the necessary event does not occur. The time-out is passed as an argument to the `event_receive` call.

To send an event, the event or the events to be sent and the task ID to be sent are required:

```
event_send (MY_PROJECT_DATA_READY, consumer_task_id);
```

We should note that unlike queues, where messages stack up, in the case of events, it does not happen. If an event is sent to task and before it processes it, if the same event is sent again, the second signal can be considered 'lost', because the OS will not remember that the event is sent twice.

Events: Looking under the hood

The events are implemented using bitwise integer operations. The event flags for a task are stored in event control block. If an event occurs, the value of the event is ORed bitwise with the event control block. Needless to say, on startup, the value of the flag will be initialised to zero. Once, a task consumes an event by event_receive, the corresponding bit is cleared.

Initial value of event flag for consumer task = 0×0000

For e.g., consider that the producer task sends MY_PROJECT_DATA_READY to the consumer task. The value of the event is 0×0001. This value is ORed with the event flag for the task

$$\text{Event Flag} = 0\times0000 \mid 0\times0001$$
$$= 0\times0001$$

Now, if the consumer task blocks the event MY_PROJECT_DATA_READY, then the OS checks if the bit is already set. In our case, it will already be set and the consumer task can begin processing the data. And, now, the event is '*consumed*', i.e., the flag value is reset.

$$\text{Event Flag} = 0\times0001 \;\&\; !(0\times0001)$$
$$= 0\times0000$$

If some other flag is set, the above operation will not clear those flags. From this we can understand the case of missing signals. If MY_PROJECT_DATA_READY event is already set, sending the same event does not alter the value of the event flag for the task.

Event Flag = 0×0001 | 0×0001

= 0×0001

So sending the same event to a task before it uses the previous signals will result in loss of signals.

7.9.3 Events Vs Queues

i. The obvious difference, (from the discussions above) is that queues can buffer messages while the events cannot.

ii. Events cannot carry data.

iii. A task can wait on multiple events, while the tasks wait only on a message from a single queue. (This is discussed in detail in the queues section).

iv. The communication takes place directly between the involved tasks. There is no component (like queues) in-between component.

So, the general rule is that if a task has to block multiple events, then, it is preferable to use events. Else, queues are a better option since they have a buffering mechanism.

Another point to be remembered is that sending events can cause scheduling to occur. So, if a lower priority task sends an event to a higher priority task that is blocking on that event, then the lower priority task may be pre-empted and the higher priority task will resume execution.

7.9.4 Function calls and accessing global variables across tasks

The mechanism of function call involves a task *A* to call a function from task *B* directly. So, this function executes in the context of task *A*. If during the execution of the function, task *B* gets scheduled and starts execution of the same function, the stack of task *B* shall create its own context for this function.

This means that the function is safe as long as it accesses only local variables or it establishes any of the mechanisms of protecting critical section of memory accessible by other tasks.

However, by our personal experience, we have seen that such an inter-task communication leads to a clumsy implementation, so such mechanisms should be handled with care, and avoided if possible. Calling functions directly across tasks and accessing global variables defined in another task directly may be difficult to trace, reproduce and debug.

7.10 TIMERS

Almost every RTOS will provide features to use a timer. We have already seen that we cannot use delay loops to implement timers in an environment that supports preemptive multitasking. So, we have to use either some of the hardware timers that could be available or use the abstractions of timers provided by the RTOS.

Various RTOSes provide various kinds of APIs to use timers. But some of the concepts remain the same irrespective of the OS:

- **Tick ISR**—This is an ISR that gets executed at every *tick* of the clock. The resolution of timers that an RTOS can provide is decided by the clock that is used for the ticks and the tick ISR.
- **Starting of Timer**—There is usually an API provided to start a timer. The arguments are usually the duration of the timer (may be in micro/milli seconds or in counts of ticks of the system clock) and the function to be called at the expiry of the timer. For e.g.

```
tLED = timer_start ( 1000,          // 1000 milliseconds
                     ToggleLED );   // Call a function to toggle
                                    // ON/OFF an LED
                                    // (void ToggleLED(void); )
```

Usually, APIs that are used to start timer usually return an ID to the timer that can be used to either cancel or query the current status of the timer.

The above call will call the function ToggleLED() after expiry of 1000 milliseconds. The RTOS that you may be using may provide a different interface. Some may post an event or a message in the queue after the expiry of a timer.

Cancelling (Stopping) a Timer: There are usually APIs that can be used to stop a timer before the timer expires. This can be done in cases when we are waiting for an input but we do not want to be stuck in the case when input does not arrive or is lost. In those cases, we start a timer before we request input and we cancel the timer on arrival of input. If the input does not arrive, then in the timer expiry function we handle the situation. A timer ID (like tLED in the code above) is required to cancel a timer.

This is one of the common patterns that is used when we interact with the external environment for inputs/synchronisation.

```
timer_cancel ( tLED ); // Stop/Cancel the LED timer
```

Warning

Strange situations can occur in cases when the input occurs and at the same time the timer also expires. So, we might need to handle these kinds of issues also. I ran into a similar situation recently. *Positive Acknowledgement Scheme* is a technique in wireless networks, where we assume that the packet transmitted over the medium is considered successful only if the acknowledgment for the transmitted packet is received. If the acknowledgment (ACK) is not received, we may have to indicate that the packet was not transmitted successfully. I had a system in which results of both successful and unsuccessful transmissions were communicated to a higher level entity. I suddenly realised that I was getting more results (either successful/unsuccessful) than the packets I actually had to transmit. I then realised that due to some complex processing in the ISR, even after the ACK was received, the timer that started to stop waiting for the ACK was not immediately cancelled. So, the timeout indicated an unsuccessful transmission and the receive ISR indicated a successful transmission for the same packet. (The receive ISR cancelled the timer AFTER it had expired and the result of the cancellation unfortunately, was not checked). So, I was getting more results than the packets and hence the transmit state machine was going awry! The problem was solved by increasing the timeout and optimising the reception ISR.

7.11 LESSONS LEARNT

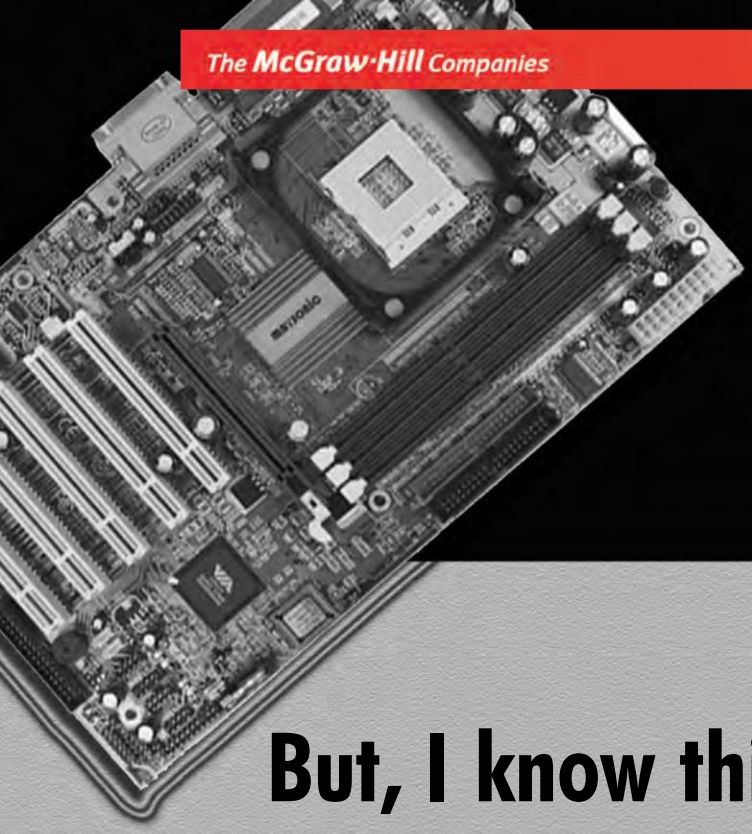
A realtime operating system is different in many ways from a conventional desktop operating system. In a RTOS environment, the typical steps, which take place on startup are bootloading, initialisation of hardware, BSP and RTOS. The most important job of an RTOS is management of tasks, which includes creation, scheduling and deletion. Tasks can be in different states throughout their existence — dormant, ready, running and blocked. Each task has a TCB associated with it in order to store its status and associated data.

Tasks in a realtime system can be scheduled using different schemes. Most popular schemes include time slicing, strictly pre-emptive, or combinations of these. During the execution of tasks, they may need to synchronise with other tasks. Mutexes and semaphores are popular methods for synchronisation. The designer should be aware of the problems of race condition and priority inversion associated with task synchronisation.

Tasks need to communicate with each other in order to share information. Message queues and events are important methods used for inter-task communication.

7.12 REVIEW QUESTIONS

- What are the most important jobs of an RTOS in an embedded device?
- Describe the boot-up sequence with RTOS in an embedded device.
- What is a task? How is it started, scheduled and deleted? What is the significance of the idle task?
- What are the different scheduling methods?
- How does priority inversion occur? How can we solve it?
- What are the various intertask communication mechanisms in real time systems?
- What is BSP? Why it is needed?
- What does the BSP code usually do?
- What is a Control Block?
- What are the states that a task can be in?
- What is the state transition that is not possible in a task?
- What is 'CPU loading'?
- When do we need task synchronisation?
- What is a 'Race Condition'?
- What are two kinds of priority inversion? Which is more dangerous?
- What are the techniques that are available to tackle the problem of unbounded priority inversion?
- What is a semaphore? How are they used?
- Is it strictly necessary that after an ISR is executed, execution is returned to the task that was executing when the ISR was raised? Why?
- What are mechanisms available to pass data between tasks?
- What are events? How are they different from message queues?



SECTION FOUR

But, I know this already!

Typical Software Development Process

A typical software development process consists of the following steps:

1. Requirements collection/analysis
2. Architecture
3. Design
4. Implementation
5. Testing

The steps described above are just for guidance and may not be followed strictly in the above order. Software engineering is evolving fast to tackle current issues in software. So, with advent of 'Test-First' methodologies, we may actually begin writing test code even before we write the actual implementation code!

This section tries to give a brief overview of the general development process. Each of the step is explained in detail in the following chapters. The process starts with creation of a requirements document that describes the nature and functionalities of the software that needs to be developed. Once the requirements are completed, then the high

level architecture and design of the system are created. Then the code is written based on the design. The software is then tested and delivered. (I only wish software development can be as easy as described in this paragraph…)

It could happen that each of the steps is completed in iterations. Or, it could be that some of the steps are changed to suit the development environment.

In a realtime environment, this is not sufficient. We usually describe *realtime* as *"When correctness in not enough"*. Even though the steps above will try to ensure that the correctness of software is achieved, we can notice that there is no step to ensure timeliness of the software. So, we must introduce another step —*'Performance Analysis'* to ensure realtime behaviour of the system. "So, in the case of realtime systems development, the additional step of temporal —performance analysis is required."

1. Requirements collection/analysis
2. Architecture
3. Design
4. Performance Analysis
5. Implementation
6. Testing

The topic of performance analysis has been covered briefly in the "Introduction to Realtime Theory" Chapter.

Chapter

8

Requirement Engineering

8.1 INTRODUCTION

In a typical development process, the sequence of events is as follows:

i. A statement of need arising out of predevelopment process
ii. Requirements gathering and specification
iii. Architecture definition
iv. Design
v. Implementation
vi. Testing

The first two activities trigger the rest of the events. For example, it is perfectly fine to embark on testing activity after a requirements document has been created. This chapter will look briefly at the activities and issues related to gathering and specification of requirements.

8.2 REQUIREMENTS OF AN EMBEDDED SYSTEM

The development of requirements of an embedded system starts with the availability of a statement of need. Based on this initial statement, the requirements engineering process is used to define and describe what a system should do to satisfy the informal requirements specified by the statement of need.

8.2.1 Why should we spend time on requirements?

One of the classic doubts that arise among novices in software industry is whether we should 'waste' time in documentation. They believe, since embedded systems are usually small, and have common constraints such as memory, processing speed etc. it is not worth the effort and time to prepare detailed specifications and design for an embedded system. It should suffice to read some requirements for a similar system and then tailor them for our system.

It is true that all embedded systems have a lot in common. They usually have common constraints. They are bound by realtime limitations. So an experienced developer in the embedded domain will understand another system very quickly. However, understanding a system and designing a new system are totally different things. The world is not altruistic. Some of the common factors that necessitate a formal requirement specification are:

- There may be communication gaps when requirements are outlined in the beginning.
- Requirements may change during the development.
- It may not be possible to remember all the minute details that are discussed and decided in the meeting with the customers. It is hard to remember one's own code after a few weeks, let alone something as big and complex as the entire specification and behaviour of the system.
- Though the embedded systems are small, they are usually very complicated. For example, embedded systems related to communication typically have to take care of many data ports simultaneously, process them with realtime limitations, use variables and memory in an effective way and usually some other systems in the neighbourhood are waiting desperately for an output from the embedded system!

Hence the basic steps of requirements, architecture and design deserve a lot of effort before jumping into the world of coding. As a matter of fact, the standard of embedded industry is that the complete product development of an embedded system is 40% design, 40% testing and only 20% coding. Secondly, it is advisable to catch bugs creeping into the system as early as possible. Data shows that bugs caught in the specifications phase are the cheapest to solve. They become more and more expensive and effort sensitive as the product goes through the later stages of its lifecycle. It is advisable to spend some time in the beginning of product development to understand and

document the requirements and design of the system before embarking on the implementation of the product. Implementation has its own issues and they are not in any way trivial. However, if we do not follow this school of thought, we will burden the implementation stage with issues that could easily have been solved at requirements and design stages. In later sections, we will analyse the issues that affect different stages of the development lifecycle and the cost of such issues. One thing, however, has been observed over a long period of time:

P2P

The quality, clarity and completeness of specification description of a product have a lot of role in driving the eventual quality of the product being developed.

Figure 8.1 illustrates the increase in the complexity and cost of bugs in different stages of the development.

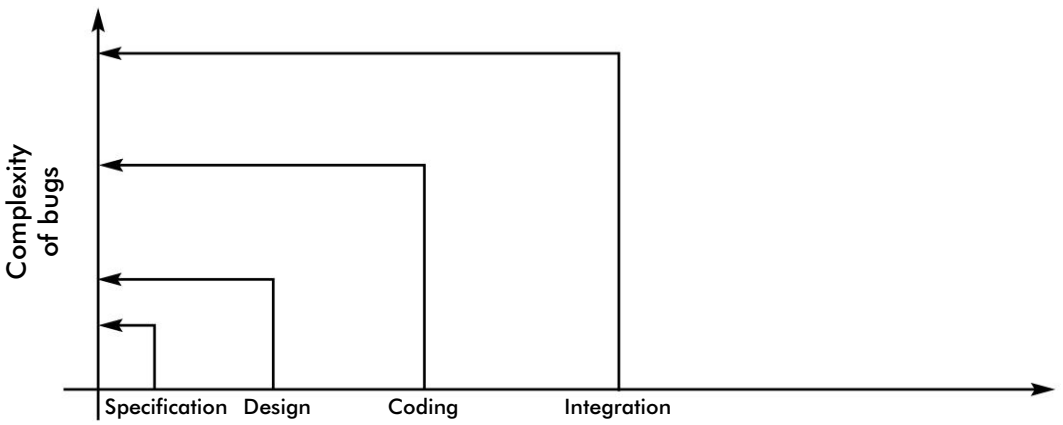


Fig. 8.1 *Increase in cost of bugs vs time*

8.3 CONCEPTUALISATION OF A PRODUCT

As we saw in the previous section, the statement of need is the first stage of conceptualisation. This may include, at a very high level, the functions the product is expected to perform, the stimulus-behaviour sets of some actions from the system in plain English. Sometimes, this document is also referred to as the 'feature list'. Alternatively,

the requirement document can be written by a senior member of the implementation team, or by a system architect, in conjunction with the customer. The definition of the customer is rather vague in this context.

The customer can be the marketing department inside the same organisation. They may have done research on the current trends in the market and come out with a proposal about the kind of product the company should develop to remain competitive in the market. The definition of this product might be very hazy at this stage, since nobody has visualised it at a technical level. Secondly, the specification of the document might just be a culmination of different kind of features with tags like 'necessary', 'good to have' and 'desirable'. This specification generally will not give a clue to the technical aspects of the system, such as amount of memory required, or any estimates associated with development of the system. The specification of expected behaviour of the system may undergo huge changes while the system is being developed as the system is better understood and the desirable features change. In many cases these specifications are known to take a complete 180 degrees shift too.

A customer might be another department inside the same organisation who wants to implement a huge system. However, they may want a small portion of the system to be implemented from outside because of lack of resources or expertise.

A customer can also be another organisation that gives a sort of 'contract' to this organisation for the amount of job done. In this case, the specifications are provided with a much higher degree of detail (since the actual design of the complete system has already been done and a small component of the system has to be developed from outside). The external interfaces of the small subsystem being developed usually would have been defined to a great detail. In other words, the specification for this component is fairly complete and frozen.

As we can see in the previous paragraph, there is an entire gamut of specifications potentially available that range in the amount of detail and completeness as well as propensity to change. Hence the embedded system being so designed has to take into account the factors associated with the applicable scenario before choosing a design and implementation methodology. The statement of need is an informal specification of the deliverable, and is usually written by somebody not directly involved in the development of that product. So, in most cases, this statement of need would not be expected to contain the nitty-gritty of internals of product behaviour.

Thus jumpeth into the fray the requirements gatherer! (S)he is the catalyst who converts the vague and incomplete statement of need into a blueprint for the development

of the product, commonly called the requirement specification in computer science parlance. The requirement specification is expected to contain a description of principal features of the product, its attributes and behaviour. The requirement definition contains a precise, complete and clear description of:

- What the product is expected to perform?
- Why it is to be done?
- What should not be done by the product?
- What are the constraints and expectations?

The inputs for the requirement definition in addition to customers, may be provided by system level definitions, other competing products in the market, well recognised standards such as those produced by IEEE, etc.

Consider an example of the bar code reader described in Chapter 1. It is expected to send its output signal on a serial line to a printer. This may be the core requirement of an embedded device. However, the display of the actual cost on a screen may be an add-on feature for the product. The customer and architect come out with the various scenarios after looking into the different uses, the product will potentially be put into. It is usually not possible to envisage exhaustively all the inputs the system will receive during its real life usage. However, to a fair degree of accuracy, each small scenario can be broken down into multiple cases to detail the situations the product will find itself into. This step may mistakenly be taken as a design step. However, this is still a step into understanding the expected behaviour of the system. Since the embedded systems typically work because of external events, usually this behaviour is shown or understood through what are known as use-case diagrams. The use case diagrams illustrate a system as a black box, and the user of the system (may be human or another system) together with the kind of interaction possible.

8.4 REQUIREMENT ENGINEERING PROCESS

The requirements engineering process consists of following steps:

- Requirements development
- Requirements management

Requirement development implies arriving at conclusive requirements as described in previous paragraph.

Requirement management implies managing change in requirements, impact analysis on the current development.

8.4.1 Requirements development

Requirement development involves the following steps:

- Elicitation
- Analysis
- Specification
- Inspection

Let us discuss these steps one by one.

Requirement elicitation

Elicitation means gathering of requirements from the customer (the customer is defined in Section 8.3). This means, *listening* to the customer, *sending appropriate stimuli* to the customer so that accurate responses are received, understanding the *needs of the customer* other than the ones being stated, asking *meaningful questions* in order to arrive at better details, *summarising* the points under discussion from time to time in order to erase any misconceptions or communication gaps, etc. In that sense of the word, requirement elicitation procedure is more of an art than a science since it goes into the realm of non-technical aspects of engineering.

The most popular methods of requirement elicitation are interviews, brainstorming sessions, questionnaires and use cases. Many rounds of these and combination of these methods are usually used to elicit requirements.

Requirement analysis

Requirement analysis involves estimation of the cost based on requirement elicitation process and classifying them into categories like: mandatory, optional and good to have. The visualised solution during the requirement elicitation process may then be scaled down into a workable solution, based on real life constraints.

Requirement analysis may identify dependencies among requirements, assumptions to be made during development and any reuse possible from an existing product. Based on real life problems associated with insufficient resources, insufficient time, change in requirements, imperfect communication and lack of proper financial support, a workable solution is then arrived at. (See Fig. 8.2).

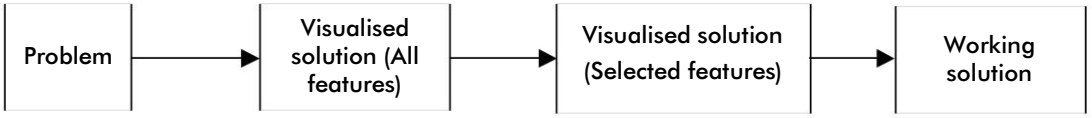


Fig. 8.2 ***Steps in requirement analysis***

Requirement specification

Now comes the time when the requirements are written in black and white based on the elicitation and analysis phase of requirement gathering. If the previous two steps have been followed properly, chances are that all requirements have at least been captured. The step of converting these requirements into a human readable and understandable form remains.

An ideal requirement specification is:

Correct: A requirement is correct if it helps to satisfy the needs of all the persons/parties affected by the outcome of the system development.

Unambiguous: A requirement is unambiguous if it has only one interpretation. Models, pictures and brief, precise statements with good grammar help in making unambiguous specification.

Complete: A requirement is complete if all the needs of the persons/parties affected by the outcome of the system development are satisfied. Definitions of responses by the system for all realisable inputs, in all realisable situations are specified.

Consistent: Requirements should not contradict each other and all inputs for the requirements are coherent.

Verifiable: A requirement is said to be verifiable if:

> There exists some finite cost effective process with which a person or a machine can check that the software product meets the requirement, to a degree sufficient to convince all the persons/parties affected by the outcome of the product development.

Ambiguity implies non-verifiability

Modifiable: A requirement specification is modifiable if it is easy to make changes to it. Usually, this is possible if the specification is organised in a modular way, sections and sentences are not too much burdened with many specifications, and care is taken to avoid what is called over-specification.

Traceable: A requirement document should be forward and backward traceable. It is backward traceable if all requirements can be traced back to the statement of need or/and other input document. The requirement specification is forward traceable if each requirement can be uniquely referenced from all documents spawned by the requirement specification.

Effective traceability can be achieved through:

- Unique numbers/identifiers/labels
- Logical organisation in presentation
- Traceability matrix between statement of need, specification, design and test documents.

The IEEE 830–1993 provides a description of good requirement definition. By and large, a requirement definition needs to take care of following issues:

Function: The actual function of the product in the form of stimuli-response pairs or use cases. For example, *when a mobile station is paged by the network, it shall respond with a paging response message*: is a function in the requirement definition of a mobile phone. Such requirements may be provided by standard bodies and/or business requirements of the product and organisation. Business requirements may define the product strategy based on its vision and scope and may explain where the product fits in the market.

Interfaces: All the external interfaces and environment in which the product is expected to perform. The interfaces and environment may include software, hardware and human beings. *The mobile phone shall provide the CLEAR key to the user in order to kill an editor or cancel an outgoing call*: is an interface requirement for the mobile phone.

Performance: The realtime characteristics and timing constraints of the product. *The mobile phone after getting switched on, shall be capable of responding to a paging within 30 seconds, under ideal radio conditions*: is a realtime requirement for a mobile phone.

Non-functional requirements: These requirements are not associated directly with the product behaviour, but more with the overall characteristics such as maintainability, scalability, availability, portability, testability, size, security, etc. *The ROM size shall not exceed 5MB*: is a quantitative nonfunctional requirement.

Quality requirements: These are related to aspects of product development, such as, development environment (use of structured language vs object oriented design), available budget and resources, etc.

Requirement inspection

After the requirements are ready, it is time to inspect them for verification of all components listed in the specification. A body constituted by affected parties and neutral third persons should carry out inspection. The purpose of inspection is to check that—

- Requirement definition is complete
- It is error free and clear
- It is understood well and agreeable to all parties affected by it

Usually, it is not possible to create a complete requirement specification in the beginning. The iterative process of specification, modelling, refinement, discussion with customer, change in specification, change in modelling…, goes on till the requirements have been frozen to the satisfaction of the developers and the customer.

8.4.2 Requirement management

One of the major complicated issues in the embedded (as in any software) industry is the near-constant change of requirements of the product. So, at the time of requirement definition, the author should plan and prepare for it. If the requirement definition has all the good qualities described earlier, it is easy to manage such changes.

Requirement definition may also change based on errors noticed by its users on the field. Such errors are easier to fix since they are related usually to specific section or statement, and the correction can be made satisfactorily.

However, because of a change in requirement, or a new requirement for the product, complications can arise. The first step in this stage is to perform an *impact analysis*. A good and up-to-date traceability document comes handy in this situation. For example, if it is possible to find out all changes in the design, test and requirement specification from the statement-of-need document, any change can also be traced easily. The impact of a new requirement can be gauged from the same traceability matrix. Another practice followed in combination with a traceability matrix is to base-line a requirement specification document once a certain level of clarity, maturity and stability has been achieved. A base lined requirement definition provides a clear launching pad for developing more activities like design and testing, besides providing a base for comparison of future changes.

To conclude, requirement specification process is iterative in nature. The requirements phase is referred to more than once during the project life cycle.

8.5 COMMON PROBLEMS IN REQUIREMENT ENGINEERING

Common problems experienced by authors during requirement engineering can be listed as follows.

Incomplete information: Usually when requirement specifications are written, many details may not be available to the authors, possibly because the information does not exist or they may not have access to the information. So they need to make assumptions and go on. Bad assumptions may create damage in some aspects of the system. Now, this situation may not be entirely avoidable since requirements are not usually clear in the beginning. The best we can do in such a situation is to highlight all assumptions made at the beginning of the specification document, together with their impacts.

Mixing design with Requirement: Being engineers by heart, authors of requirement specification tend to get carried away and start specifying *how* the system is implemented instead of *what* the system is supposed to do! To avoid this, the author should ask *why* the requirement is needed. If this takes the author back to a real need statement, then a requirement is being specified.

Proper usage of words: Appropriate syntax should be followed so that the readers understand the statements well.

For statements that relate to requirements, the normal syntax used is SHALL.

For statements of fact, usually WILL is used.

Consistent usage of these terms will make sure that the specification is not open to multiple interpretations.

Quantitative statements: Quantitative statements should be provided as far as possible. This is because they are easy to test and verify. Examples of testable specifications

The software shall compute aircraft position within the following accuracies:

- + or − 50 ft in the horizontal plane
- + or − 20 ft in the vertical plane

The system shall respond to:

- Type A queries in <= 2 sec
- Type B queries in <= 10 sec
- Type C queries in <= 2 min

Where Type A, B, and C queries are defined in the specification.

Incorrect syntax or bad grammar: creates problems in interpretation and understanding the intent.

Ambiguous statements like minimise, maximise, fast, easy, sufficient, adequate, small, etc. are an absolute no-no in requirement definition.

Missing requirements: Authors may miss some aspects of requirement definition such as quality. This can be solved by using a standard template for requirement definition.

Over specification leads to wastage of resource. It creates unnecessary requirements that are difficult to follow.

8.6 REQUIREMENTS OF CARD VERIFIER

Let us take an example of a card verifier that controls the access of the visitor of a building through his card. This system will have a small sensor to detect the card and read the bar code to identify the card. Then the system will search the code inside a list of codes that are authorised and have the access inside the building. If the code matches, it will generate a signal to release the lock for the gate. Otherwise it will show possibly a red light on the front monitor and a user-friendly message such as "Card not authorised". If an unauthorised card is put inside the slot for a maximum number of times, the system will generate an alarm. Figure 8.3 shows an illustration of such a system.

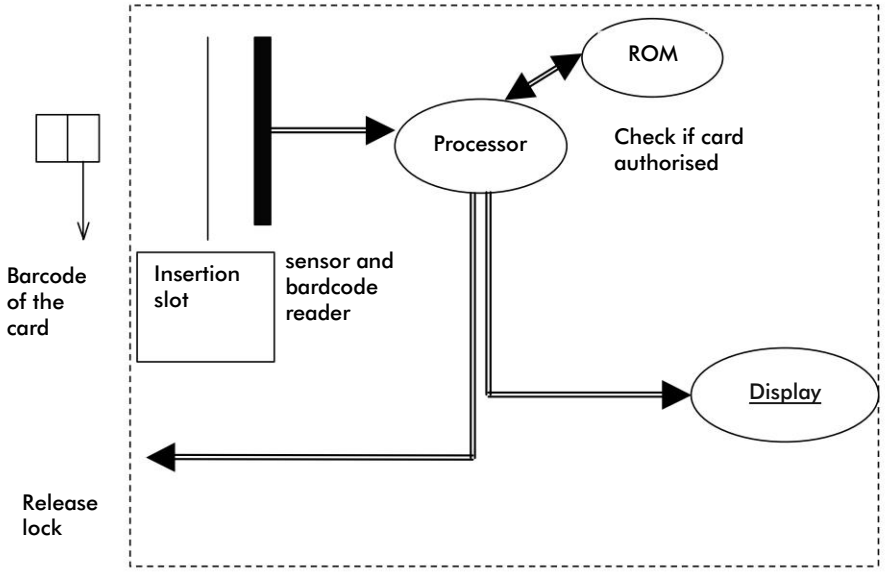


Fig. 8.3 *A card verification system*

The system has three external interfaces. One interface is with the card that is entered inside the slot. The other one is the lock of the gate. The third is the user interface that displays a message.

The previous two paragraphs can be taken as statement of need from the customer. As usually is the case, this statement of need is vague and very general from an engineering point of view. It does not give insight into a lot of specifics that may be very important for the engineering community. It is silent about the non-functional requirements of the system—its reliability, testability, ease of future development, etc.

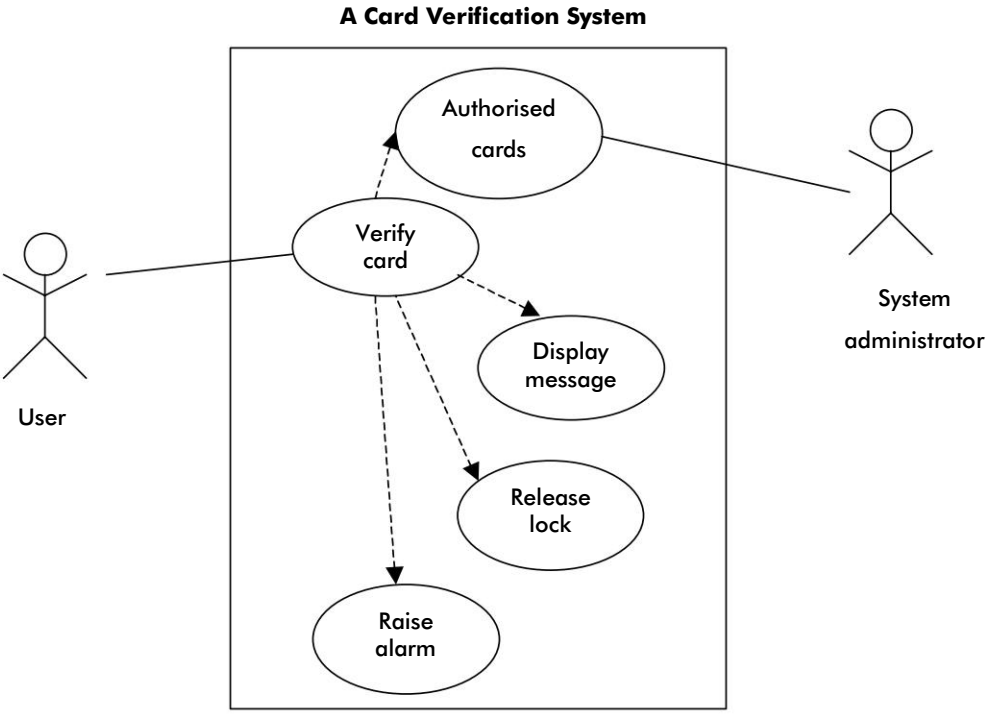


Fig. 8.4 *Top level use case for card verification system*

As shown in Fig. 8.4, this system has a very simple use case diagram. The users of the system are the human users or more specifically the card that is inserted into the system. Input to the system is a card-in event and a card-out event. The card-in event causes the system to read the barcode number of the card. The card-out event is an indication that another card can potentially be inserted very soon so that the system should get ready for the next input. The output of this system is unlocking of the gate, a user-friendly message or a small alarm and red light.

This system is a realtime system, since the output of the system, door lock release and red light have to be generated in a 'reasonable' amount of time. The amount of time that is reasonable is anybody's guess and is not generally specified. It is measured in few milliseconds accuracy. Hence, this system is not a hard realtime system. And, it is a soft realtime system since a few milliseconds delay will not cause damage to life and limb. The specifications for this system will finally give an approximate best and worst time behaviour of this system. The timing issues of the system can then be derived from the functional requirements given by the customer and these best and worst case scenarios.

Once, the use-case diagrams identify the external behaviour of the system on a higher level and list the kind of interactions between the system and the external world, it is time to judge the different events in terms of their time relationship. One of the most commonly used tools for this job is a message sequence chart (MSC). As the name suggests, message sequence charts are basically a representation of components of the system as a whole, a stimulus, and the representation of other events and message exchange in time domain. MSCs give a feeling about the impact of different events on a system, as also on the state of the system.

For example, MSCs can be made for situations related to an authorised card, that has not been inserted properly and which has been left inside the reader, etc. These three situations will possibly elicit similar responses as a whole but may be associated with additional actions as well. The additional action may be a beep sound to indicate that the visitor has left the card inside the machine.

MSCs operate in the realm of the system design, and within the specifications. They usually throw light into the different ways in which a system can be broken and developed. However, a requirement definition in terms of its use-case and the MSC defining its break up are closely linked and drive each other in the beginning of the project. The requirement definition drives the MSC generation. In turn, any missing requirements can easily be detected based on additional insight received with the help of MSCs and realtime behaviour modelling of its components. We will describe the MSCs for this system in the next chapter. However, a basic introduction in this chapter would not be out of place.

Detailing requirements at this stage facilitates verification and change without impacting the development cycle. In the initial stages of development, only the top level of the system is aware about the services it is expected to deliver. When we write MSCs, we force ourselves to think about components of the system and also to

brainstorm into all the possible events the system may have to manage. This effectively presents new requirements for the system that previously the customer may not have thought of. Thus a greater degree of control over the development cycle is achieved since the customer requirements are met by the efforts of implementation team. Any communication gaps left because of insufficient specification at the early stage of conceptualisation of the system are filled here.

8.7 POINTS TO REMEMBER

While do my requirement specification, the following point deserve to be remembered.

8.7.1 Focus on requirement only

Care should be maintained so as not to involve design issues at this stage since they can easily bias the specification process. Secondly, since the idea is in its infancy, any steps in the direction of implementation may prove futile if the concept may have been misunderstood. At this stage, we need to look at the system from the point of view of the user and understand the components. It should be our endeavour to enlist and identify all possible stimuli to the system and their impacts on different parts of the system. For some stimulus, it is possible that the system may not have any impact other than doing some household chores or starting/stopping of some timers. Some stimulus may result in a lot of actions based on the timing and kind of stimulus. Since the system to be developed is embedded, a realtime constraint is usually inherent. As we have seen in previous chapters, the most important constraint in making a system realtime is the simultaneous execution of a lot of code because of external or internal events happening together rather than in a serial way. When we make MSCs identifying the different actions the embedded system will perform based on the different stimulus, we are trying to nail down and quantify the system behaviour. It becomes possible to identify the best and worst case behaviour of the system if an exhaustive list of MSCs have been prepared.

8.7.2 Change is the only constant!

The next thing to be kept in mind is the potential change in specifications of the system. Though people are usually uncomfortable whenever the requirements change, we should remember that the only software, whose requirements do not change, is the software that is not used ☺. The amount of change may vary from system to system but since the embedded world is very dynamic, we should be prepared to undergo a few changes in the specification itself before the product sees the light of the day. When the specifications change, the complete development cycle has to be repeated. Here MSCs and use-case diagrams help in a big way since they are the tools that help to

quantify the system. Any change in specifications of the system will finally be gauged from the amount of change it has on the output of the system based on a stimulus. New MSCs or use-cases may have to be added to take care of additional requirements and the existing ones may need to be changed to understand the effect of change in current requirements. It is very important to keep the specifications document up to date and make it a 'running' document since it is the very basis of all activity in the development of the product. Specifications are the corner stone of final product delivery.

8.7.3 Focus on early testing

Another offshoot from requirements is the test procedure. Since the requirements document gives an insight into what the system is expected to do under different stimulii, the test document has almost a one-to-one mapping with the requirement. The test procedure should be able to perform testing of all requirements listed in the requirement document. There is no other qualitative test of the system. This test procedure will be able to catch a lot of bugs on the host machine by simulation of external environment of the system. Every bug caught from the host machine is an investment in saving time, energy and cost in the field.

8.8 REQUIREMENTS CHECKLIST

The following list defines the components of a good requirement document for embedded systems.

- ***Author (list):*** This comprises of the author(s) of the document. One author should be identified as the owner of the document. (S)he updates the document and maintains it throughout the duration of the project.
- ***Technical control or review team:*** This identifies the architects who review the document for the sole intention of checking the technical suitability of the document.
- ***Distribution list:*** This is a list of teams or people who get affected by the requirement document.
- ***Status of the document:*** Status refers to whether the document is a draft, or under review, or released.
- ***Project Name:*** The name identifies the collection of activities for a particular goal.

- ***Date:*** The date mentions the last time the document was changed.
- ***Version:*** It identifies the evolution.
- ***History:*** It lists the changes that happened in the past with an overview of their impact and the author of these changes.
- ***Abbreviation list:*** It lists the abbreviations used in the document. It may give a reference to another document.
- ***Definitions:*** It gives an introduction to the terms used in the document. They may be standard or particular to the document.
- ***References:*** Any related information can be found in the list of references.
- ***Perspective of the system:*** Defines the external interfaces of the system and the kind of inputs that are expected in the system.
- ***Functional requirements:*** Defines what the system is expected to do based on the different inputs or change of internal conditions.
- ***Performance requirements:*** If the system is expected to respond in a particular time or if it has timing constraints for some processing, they are identified here.
- ***Use-cases:*** This is becoming an increasingly popular way to capture requirements.
- ***MSCs:*** To relate the system with the external world in the time domain.
- ***Other issues:*** related to testing, future maintenance, change control, constraints with the environment, etc. to keep track of future activities.

Notice in the requirement document that the system has been seen from a black box perspective. The requirements document has tried to create stimulus-behaviour pairs in different situations only. No attempt has been made to think in terms of how (or whether) it is possible to implement this specific kind of behaviour. When requirements change, they have an impact on this relationship between this input-output system. This specific change in input-output pair results in change in only specific portions of design. The bottom line of this approach is to quantify the system in terms of stimulus and actions. It is also possible to create specifications for parts of the system such as card reader and database search, if being developed by different teams. In that case, a specification will exist for each such part or component of the system.

8.9 CHARACTERISTICS OF A GOOD REQUIREMENT DOCUMENT

To summarise, a good requirement document for an embedded system should have the following characteristics.

8.9.1 Optimum detail

It should be detailed enough so as to identify all possible scenarios that affect the system behaviour. These scenarios will list external events as well as internal events such as timers or indications about failure in some parts of system.

The specifications should not be too detailed so as to cloud the understandability of the document from a top view. The software development lifecycle uses three views: conceptual view, during requirements analysis phase, specification view during the design phase and implementation view during coding phase.

8.9.2 Cross-references and change history

It should have appropriate cross-references to the original feature list or functional requirements document that was the basis of the current requirement document.

It should have a change history and version control identifying the changes in the document and the reason for the change. If some requirements have to be undone at a later stage of time, it should be possible to just extract the document from some version control system instead of re-inventing the wheel.

8.9.3 Conditional requirements

It is possible that the same product is being made for several customers with minor modifications. In this case, it is possible and advisable to write requirements document for everybody on a broad level and then define conditional requirements for different customers. This can ease the task of design and implementation since they have to relate to only one source of requirements, and since only one requirements document is being maintained, the extra overhead of keeping all requirements in order, becomes simpler. It also avoids the overhead of maintaining synchronisation between two copies of slightly different requirements.

8.9.4 Mention constraints and assumptions

Besides specifying the behaviour of the system, the requirements document should identify the constraints on the system related to speed, memory or power. For example, a bar code reader may have a constraint on the battery power consumption. This

constraint may not have been explicitly stated in the functional requirement document since it may have been assumed as 'understood'. On the other hand, a card verification system may have a constraint on the available memory since it has to store a huge number of authorised card codes for comparison. It is possible that these constraints are not quantifiable. However, they can be told in relative terms. For example, for the card verification system, the system is expected to complete its processing of the card in the time that the current card is taken out and the next card is inserted inside the slot. Since only humans are expected to interact with the system, and there are physical limitations on the speed with which this procedure can be performed, this constraint gives a fairly good idea about the processing speed required for each request.

If the system has a bearing on the behaviour of other embedded systems it interacts with, or if it assumes a particular interface with some systems, the requirements document is the best place to identify and detail them.

8.9.5 Keep it correct, complete and up to date

Requirements document forms the input based on which the product is finally rolled out for delivery. A detailed, complete, correct and up to date requirements document is necessary for the healthy delivery of a product because of the following factors.

Requirements document is a statement from the customer about the product that needs to be delivered. The document may be written by someone from the implementation team or the customer himself. The requirements document is the first statement about the product that is seen and accepted by both parties involved. The requirements document forms the basis for judging the quality of the product since it should satisfy all requirements identified in this document.

Requirements document forms the basis for all subsequent changes in the product. This is the starting point for judging the impact of change.

Requirements document gives rise to the design of the product. If the requirements document is vague and does not identify the scenarios that the product is likely to go through, the design for the product will be weak, incomplete and incorrect. It will be prone to changes since the behaviour may not have been understood in the same way as originally intended by the customer.

Requirements document defines the testability of the product. As mentioned before, the quality of the product is gauged from the number of requirements it is able to satisfy. These requirements are tested through some means to judge the quality of the system.

8.10 LESSONS LEARNT

Requirements engineering is a very important stage of an embedded development activity. Requirements specification process should be done properly and a lot of time should be spent on it since it formally states all the things the system is expected to perform. Hence, requirements should be detailed yet specific. Requirements engineering consists of two steps: development and management. Development of requirements concerns with all the steps needed to produce a concise, specific, unambiguous set of requirements. Management process relates to performing impact analysis of change in design based on changed requirements.

There are some universal guidelines about what constitutes a good requirement specification. And, based on the experience collected over a long period of time, a lot of do's and don'ts regarding requirement specification process exist. It is worthwhile to get familiar with these.

8.11 REVIEW QUESTIONS

- Why do you think requirement specification is a very important stage of embedded development?
- Which are the steps involved in developing requirement specification?
- What are the qualities of a good requirement document?
- Why is it necessary to keep a requirement document up to date?
- What impact does ambiguous requirements such as those below may have on:
  i. Design
  ii. Testing
    - The embedded system should be fast.
    - The TV should switch itself on within reasonable amount of time after the button is pressed in the remote.
    - We do not have limitation of ROM. However, it should be reasonable.

Write good requirements corresponding to these.

Chapter
9

Architecture and Design of an Embedded System

When specifications are available, it is time to define the architecture and design of the system. The architecture of a system identifies its components, the interfaces between them in a static and dynamic way. Each component will have a design potentially in a hierarchical way. The design will determine factors such as what data structures to use, how to distribute functionality according to their priority and their order of calling, whether to use messages or mailboxes for communication, any imported or exported APIs, etc.

9.1 GENERAL

An embedded system is usually divided into two parts:

- A hardware system which consists of processor RAM, ROM, peripherals and actual circuitry of the system and takes control of external events, fetching and execution of instructions from memory.
- A software system that 'drives' this hardware based on timing of events, user actions, and other requirements of the system.

From an architectural point of view, a lot of thought has to be given so as to find the right balance of jobs done in hardware as well as in software.

P2P

The process of developing hardware and software simultaneously by delegating issues between hardware and software, frequently, is called **co-design**.

The following factors play a pivotal role in deciding the overall architecture of the system.

Cost of developing the hardware: It typically takes more money to develop a part of hardware dedicated to a specific task. And, it is more difficult later on to change that hardware owing to a request for change in behaviour of the system. However, once hardware has been developed, it is usually free from bugs and less prone to introduce problems in the rest of the system.* And anything being executed in hardware is more efficient and saves a lot of memory space that can be used by software applications. When the embedded system is being budgeted, the cost of developing different hardware plays a key role in defining the break-up of functions in software and hardware.

Change in behaviour or requirements of system: In a system that is expected to change frequently, implementation inside hardware may not be a prudent choice. This is because of two factors:

- Firstly, hardware typically takes more time to develop.
- Secondly, facilities to develop specific pieces of hardware may not be available at all places. So, it involves additional delays.

When software and hardware teams sit at different physical locations, significant delays in hardware can retard the overall integration of hardware and software.

Complexity of the hardware: The architect has to take into account the complexity of the existing hardware and the cost of maintaining it in the future. In the embedded world, usually, complexity is avoided at all costs since it makes future changes very difficult. Usually, hardware is kept for only those jobs, which are very expensive (in terms of time or complexity) to be performed in software.

Timing constraints: In a lot of embedded systems, there are specific timing constraints. For example, mobile phones have to listen to information broadcast over the air periodically. This period is measured in microseconds. For example, every 577 microseconds, a mobile station may have to tune to a particular frequency and time slot, send 62 bits of data to a particular base station, and then do some other tasks. Now, this accuracy of 577 microseconds cannot usually be guaranteed by software. These timing constraints require that some hardware circuitry is working in close co-ordination with the

*But, nowadays, as mentioned earlier, hardware too gets buggy. So the embedded engineer must be careful enough to identify issues with hardware before spending inappropriate time in finding elusive bugs in software.

clock of the system and ensuring that the timing requirement of execution of such actions are met. This also gives rise to a requirement that this strict timing constraint is an important activity in the system and other jobs being executed inside the system may be pre-empted when this situation arises.

Specific requirements of the system: Sometimes systems have specific requirements that require some actions to be performed in hardware. For example, a mobile phone needing to perform access for the establishment of a call, needs to use the "slotted Aloha"* mechanism for channel access. This requires generation of a random number. If the first attempt is not successful because of a collision, another random number should be generated after a random duration. All these random numbers should be completely independent from one another. Now, implementation of such a series of random numbers in such close proximity is very difficult (to say the least) in software. It would be much easier to implement it in hardware (e.g. by using the random noise being received by the receiver of the mobile phone) and it will also guarantee that the numbers so generated are indeed random in value, unlike software which can produce only pseudo-random values.

Synchronisation needs with the external world: An embedded system needs to interact with the external world. The system and the external entity need to follow the same clock so that they are able to understand each other all the time. So, before any transmission takes place, both the systems decide who is the master for the communication happening between them. Then the slave has to synchronise its clock with that of the master. Usually, such kinds of requirements exist for embedded systems being used for communication protocols. For example, when a wireless device is switched on, it needs to synchronise itself to the beacon of a base station in order to get information about timing and synchronisation. Different beacons may have different clocks.

Change of configuration: Any system having the ability and requirement to change the configuration parameters should have some software to interact with external environment to use it and possibly store it. This change of configuration usually introduces a different path of execution inside the software. This configuration implementation is best done in software. As we saw, earlier, these parameters may exist in EEPROM or FLASH.

*Slotted Aloha is a contention-based protocol used when a single channel for communication exists and there are many data sources. Any data source transmits some information and waits for a reply. In case of a clash from another data source, a random delay is introduced before the next transmission and this process continues till contention is resolved.

9.2 ARCHITECTURE STYLES

All the above mentioned requirements are fairly common across the embedded system domain, what differs is the relative importance of each driving factor behind it. So, it is advisable to look at the architecture styles prevalent among the software engineering community and compare them with respect to their focus and area of usage. This will give us a feeling of identifying our system with that of some tested systems. We can take the advantages of a lot of architecture styles and build a composite architecture for the unique mix of driving factors for our system. This section together with the next section shall provide us with a lot of information gathered from the varied experience of embedded system architects.

There exist basically four broad architecture styles:

- Data flow
- Data centric
- Virtual machine
- Call and Return

9.2.1 Data Flow architecture style

The data flow architecture looks at the system as a series of transformations on successive pieces of input data. Data enters the system and flows through the components one at a time until they are assigned to a final destination in or outside the system. Like the Unix pipes and filters, this architectural style may be used in DSP applications where a complete stream of input data needs to be processed in many steps. This architecture pattern is modular, easy to change and can be reused easily. Figure 9.1 gives an example of such an architecture style.

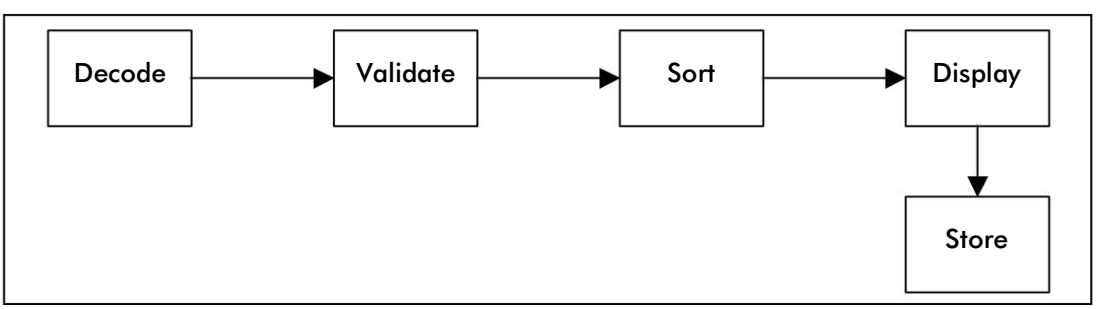


Fig. 9.1 *Example of data flow architecture style*

9.2.2 Data centered architecture style

The data centered architectural style creates all the data of the system as the core and lets the components of the system access this data. In its easiest manifestation, all data may be created as global and functions access it. This data needs to be created as a critical section if the situation demands. The advantages offered by the data centered approach are easy integration of all components since all data is integrated anyway. The components are relatively independent of one another, and the common data does not depend on how the components have been designed internally. New components can be easily added without affecting the existing components. Many embedded systems built from legacy components use this approach in order to reduce time for integration and because this style provides scalability in a very simple way.

The database of the data centered architecture can be implemented as an active or passive database. An active database is able to identify any changes in the contents of data and can notify the components affected by this change. Depending on implementation, an active database may not be required since it may add a lot of overhead of data update mechanisms, and if all updates are not used by components, it wastes energy of the embedded device. In that case, the database can be implanted as a passive repository. The repository updates data but the components are expected to contact the database in case they require the data. This may need the mechanism of critical section described in previous chapters. Figure 9.2 gives an example of data centered architectural style.

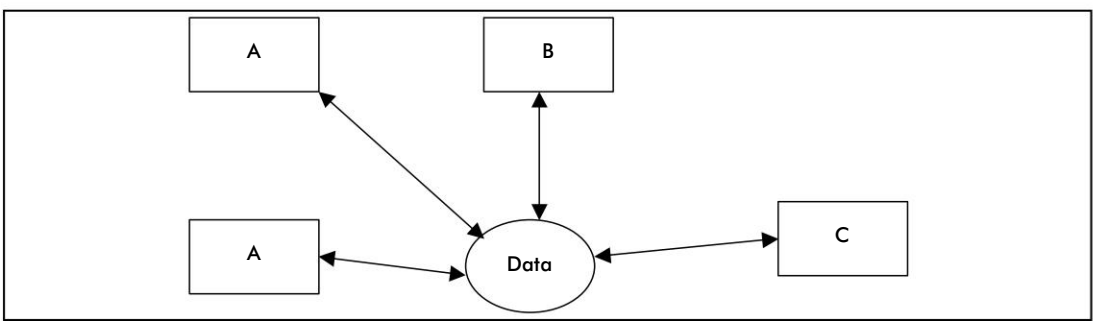


Fig. 9.2 *Example of data centered architecture style*

9.2.3 Virtual machine architecture style

Virtual machines are used when a given component should be made independent of the underlying software or hardware. In common parlance, it is also called a 'wrapper module'. The objective of this style of architecture is to minimise the cost of change in software if the underlying infrastructure changes. Second need of such an architecture arises when the underlying infrastructure is not available and the wrapper module is

then used to simulate it in order to develop or test the other components. As is evident, this style can provide portability easily. In embedded systems parlance, this architecture style is especially useful since in many cases the underlying hardware is either not available or it is simpler or cheaper to develop and test software with a simulated version. Usually, the software for embedded systems is tested on a host to find all the major problems of implementation. In such a case, a virtual machine architecture is useful in providing the relevant abstraction for development and testing environment.

9.2.4 Call and Return architecture style

Call and return architectures are the most widely used style in some form or the other. The two most common forms of call and return architectures are procedure calls and layered architectures. Procedure call style involves decomposition of the system into procedures that are called from a main module. Control is present in the main module first and then handed over to the procedure called by it and so on. Each procedure can potentially call another procedure. This architecture style aids modification of the system. A subtype in this architecture style has emerged based on distributed programming: Remote procedure calls are procedures that are not necessarily executed on the same processor as the calling procedure. Figure 9.3 shows an example of a procedure call architecture style.

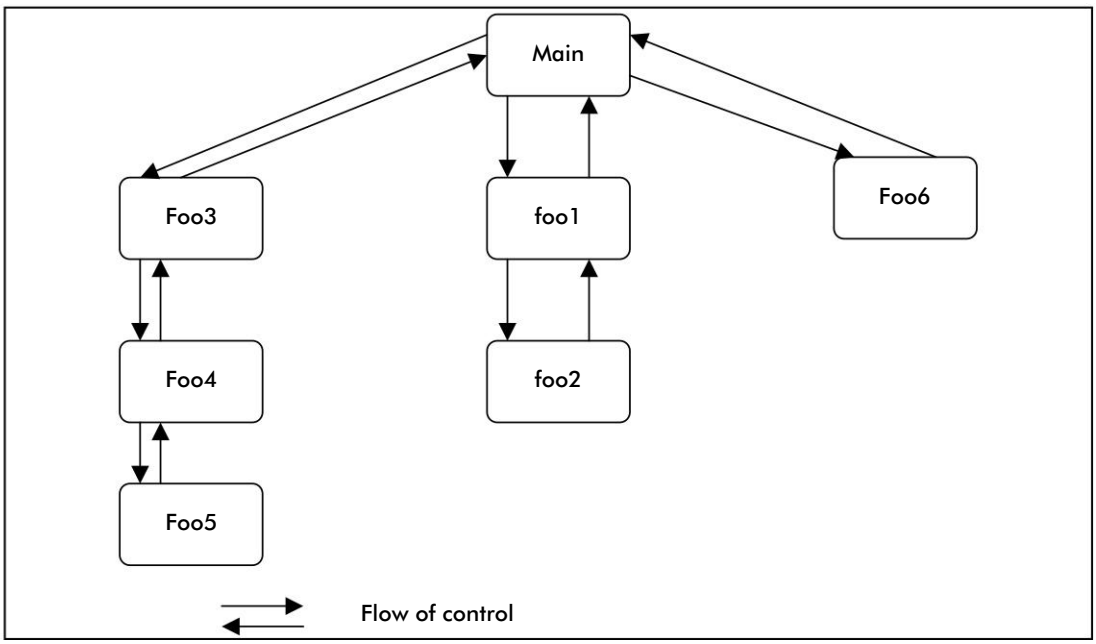


Fig. 9.3 *Procedure call architecture style*

Layered architecture decomposes the system into independent components that communicate with each other in a specific way. Usually, the components are defined as layers and each layer can then talk to its adjacent layer only. This architecture style is prevalent especially in embedded systems that implement communication protocols. In such systems, the lowest layers implements hardware specific functionality, each higher layer uses the services of the layer below and provides service to the layer above it. This style aids maintainability, scalability and portability. An astute reader shall recognise that the layered architecture is similar to virtual machine style in many ways. In fact, the underlying principle is the same in the two architecture styles, that is, communication control between components. The most common example of a layered architecture is the seven layered ISO OSI stack.

9.2.5 Independent components

Components can also be implemented using an independent architecture style. This means that the components are not aware of adjacent components or even the recipient of messages sent by them. A registry process or a message manager is implemented that notes down all components and their desired inputs. When a component sends a message, the message manager sends it to all components desirous of receiving it. This style aids portability, scalability and independent development of components.

P2P

In real practice however, systems are implemented through a mix of architecture styles mentioned above.

For example, many communication-related embedded systems decompose the system into layers, and the layers are implemented as call-and-return or further decomposed into virtual machines. The idea is to borrow the best features of all architecture styles and create a judicious mix of advantages.

9.3 ARCHITECTURE PATTERNS

In order to arrive at suitable architectures, the designers can use the experience and expertise of architectures developed previously in similar situations. This existing well-proven experience in software development can be used to create a software with specific properties. In fact, while designing a system, very often than not, most designers try to relate the properties of the new system with some system designed by them

in the past and then reuse or tailor it so that it can be applied in the current context. This is basically true and advisable for two reasons. One, a working system designed earlier for a similar problem gives more confidence that the new design will work. Second, it saves time if an idea, model or implementation can be reused.

Such an idea is an architecture pattern. An intuitive definition for architecture patterns can be given as follows:

Tips

Given a context, an architecture pattern is a generic solution to a recurring problem.

By no means, an architecture pattern can be just lifted and copied into the current system. This may be possible if the systems are very similar and they have to deal with similar driving forces, but generally, an architecture pattern is associated with a context, for example, a distributed system or interactive system or embedded system or a combination of some of these. Second, the architecture pattern provides a way of solving a common recurring problem in this context. For example, CORBA is an architecture pattern for distributed object based systems.

This section shall describe some patterns available for embedded systems. These patterns are discussed below together with their context.

9.3.1 Layered patterns

Layered patterns decompose a system into components and define mechanisms and rules for communication among these components. Two most common among them are:

- Router pattern
- Microkernel

Router pattern

Router pattern creates independent components such that the communication among them is transparent to the sender and the receiver. There is a dynamic binding of destination component for each message inside the router component. So, all components indicate the messages they want to receive to the router, henceforth, the component

sends all messages to the router. The router then sends the message to the destination component. If the message was a request primitive, the router takes care to send the corresponding confirm or reject primitive back to the sender component when the request has reached the destination. This pattern is especially useful if, the components have to be independent of each other, and when, components have instances. When components have instances, each instance can communicate with its corresponding instance on another component in a simple way, based on the routing table maintained by the router. When instances get destroyed, the router is updated.

An example of this pattern inside dual mode 3G phones shall be useful here. Dual mode 3G phones should be capable of connecting to a 3G network, as well as, the older 2.5G GPRS networks, depending on their availability. So, a link layer can be developed for both technologies, and a router decides which layer is active based on the registration performed by that component. The higher layer components become independent of which process and component they need to send their messages to.

Microkernel pattern

Microkernel pattern is used to create a basic set of essential services and a mechanism to develop additional applications and extensions in an independent way based on the kernel core. In a way, this pattern is very similar to the Unix kernel mechanism. The shell hides the applications from the kernel specifics and vice-versa. The kernel is generic and need not be modified for any application. Applications receive a standard interface from the shell and use the services of the shell to perform their jobs. This results in architectures where slightly different applications based on a core set of operations need to be developed, or in cases where the life span of applications is not large and need to be enhanced without touching the core of the kernel.

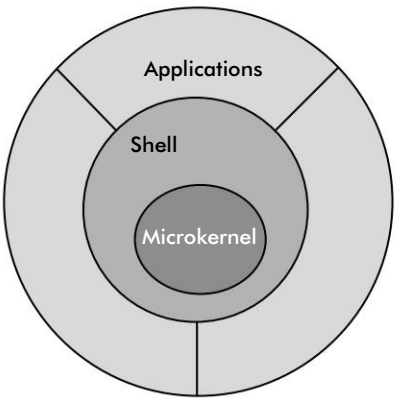


Fig. 9.4 *Micro-kernel architecture pattern*

9.3.2 Distributed architecture patterns

Distributed systems consist of independent (possibly heterogeneous) components executing on different processors separated physically, performing a job that involves some level of interaction between them. Embedded industry is rapidly embracing the distributed system approach because of its numerous advantages in terms of efficiency, portability and implementation independence. There are numerous architecture patterns for distributed systems; we will discuss three of them here:

- Client-server
- Proxy
- Broker

Client-server pattern

The client-server pattern is used in cases where one component needs to access service from another component. Usually, these components exist on physically separate nodes; however, conceptually they may co-exist as well. The server provides a particular service and starts listening for requests coming from clients. In this way, the implementation of service becomes independent of the request. The server and client only need to establish the protocol with which they communicate. The client must know the physical address of the server in order to connect to the destination. Also, the server and the client should use common protocol mechanism in order to understand each other, and, they should implement recovery and acknowledgement procedures possibly. The sequence of events in case of a client server pattern happens as follows:

- Server starts its service and waits for a request in a predefined format on a well-known logical location such as port.
- The client sends a request to the server using its physical address and port number known before hand.
- The request is routed to the server.
- Server sends a response to this request back to the client using the physical address of the client mentioned in the request message.
- The client may send more requests or the server may send more data.

The client and server are independent of each other, so this pattern aids in implementation-independence, portability and scalability. However, there are two serious drawbacks. First, the client should know before the communication, the physical

address of the server. If the server changes its location or there is another server providing the same service, some changes are required on the client side to make alternate arrangements.

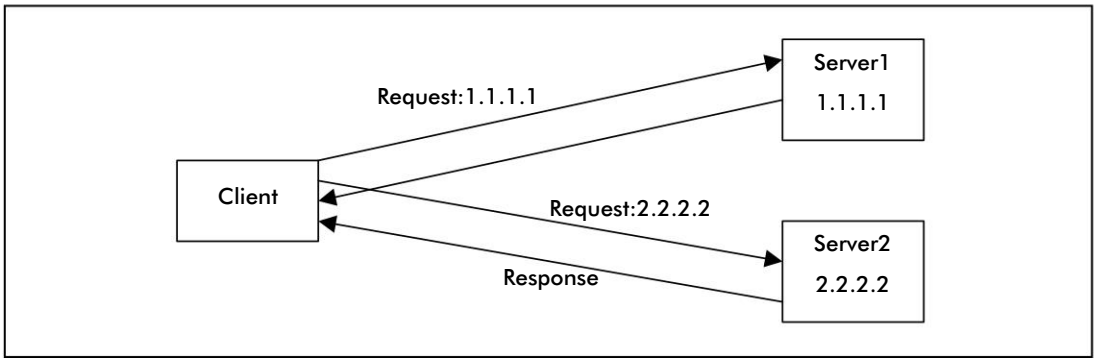


Fig. 9.5 *Client-server architecture pattern*

Second, the client and server should use a mutually agreed protocol. This means that if a client wants to connect to servers A and B, that use different protocols for communication, the client needs to implement both these protocols in order to use their services.

The Proxy pattern

The proxy pattern solves the first problem of the client server pattern. The proxy pattern provides a standard physical address to all clients such that it creates a new pseudo server that takes requests from all clients destined to the original servers and then routes them appropriately. The advantages of this approach are two-fold: in case multiple servers are present for a service, the proxy server can perform load balancing. Second, the clients need not know the physical location of server, they only need to know the service being requested, and then the proxy takes care of mapping it to a server address and communicating with it. However, proxy servers are not configurable at run time, and the clients and servers are not completely de-coupled, client and server need to still use the same protocol. Proxies act as *passive* servers.

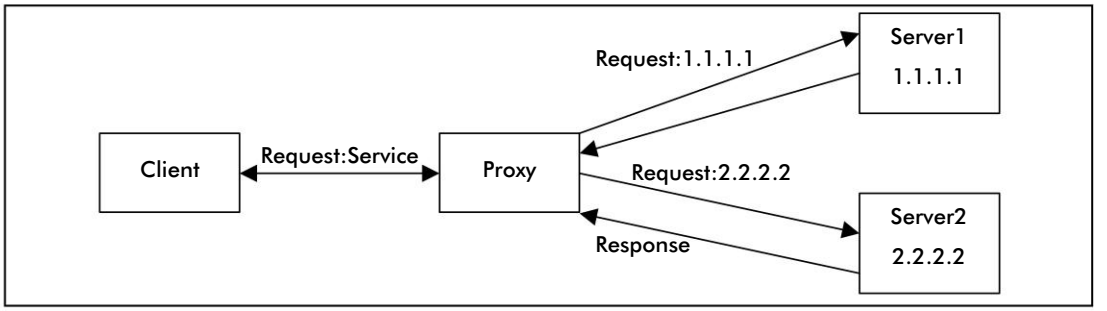


Fig. 9.6 *Proxy architecture pattern*

The broker pattern

The broker pattern removes the disadvantages of the proxy pattern as well. This pattern creates an active server called a broker. The server sends configuration data to brokers in order to advertise their services. Brokers create a run-time mapping of servers and their services, possibly together with some routing information. Clients request for services from the broker. Broker decides the best server and the best possible path to reach the server. So, client and server are completely de-coupled. They need not follow the same protocol as long as they follow the standard protocol understood by the broker. Broker pattern has been a hit among the multiprocessing architecture using symmetric multiprocessing since this pattern eases the job of the OS in scheduling jobs on different processors.

9.4 ARCHITECTURE OF CARD VERIFICATION SYSTEM

The card verification system introduced in the previous chapter can be understood first by drawing MSCs that relate the components of the system and their interaction. The system has basically four physical components: card slot, door lock, card verifier and screen.

The MSC in Fig. 9.7 introduces a correct path of execution in which the card is authorised and is inserted into the slot properly.

This case is known as the 'Happy Path', i.e. a situation where nothing unanticipated has happened. This is the path that is supposed to be executed very often. But, a robust software design must also anticipate some error conditions. Two of such conditions that can be thought of are

i. Insertion of an unauthorised card

ii. Removing the card too quickly

In each of these cases, the user is alerted of the error.

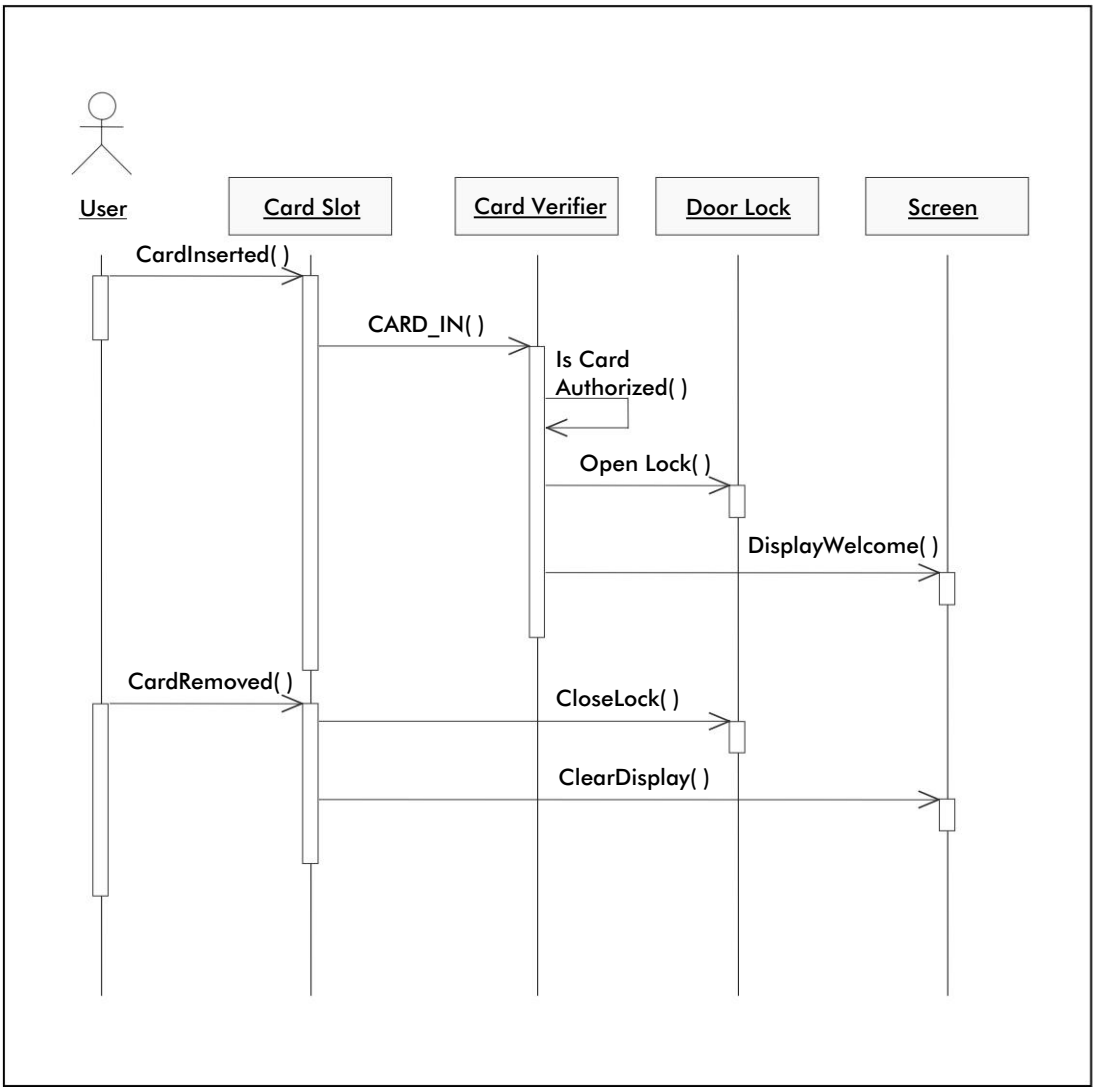


Fig. 9.7 *MSC for correct case of card insertion*

The following MSC (Fig. 9.8) describes the situation where an unauthorised card was inserted.

This MSC (in Fig. 9.9) takes a look at a scenario where the card has not been inserted into the slot for the minimum duration.

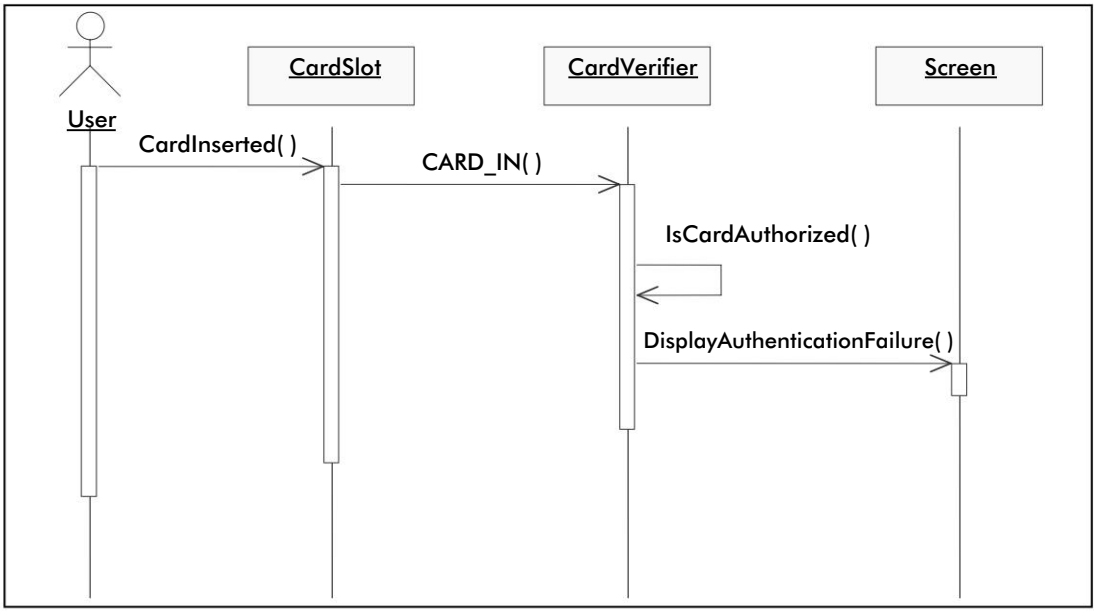


Fig. 9.8 *Unauthorised card inserted*

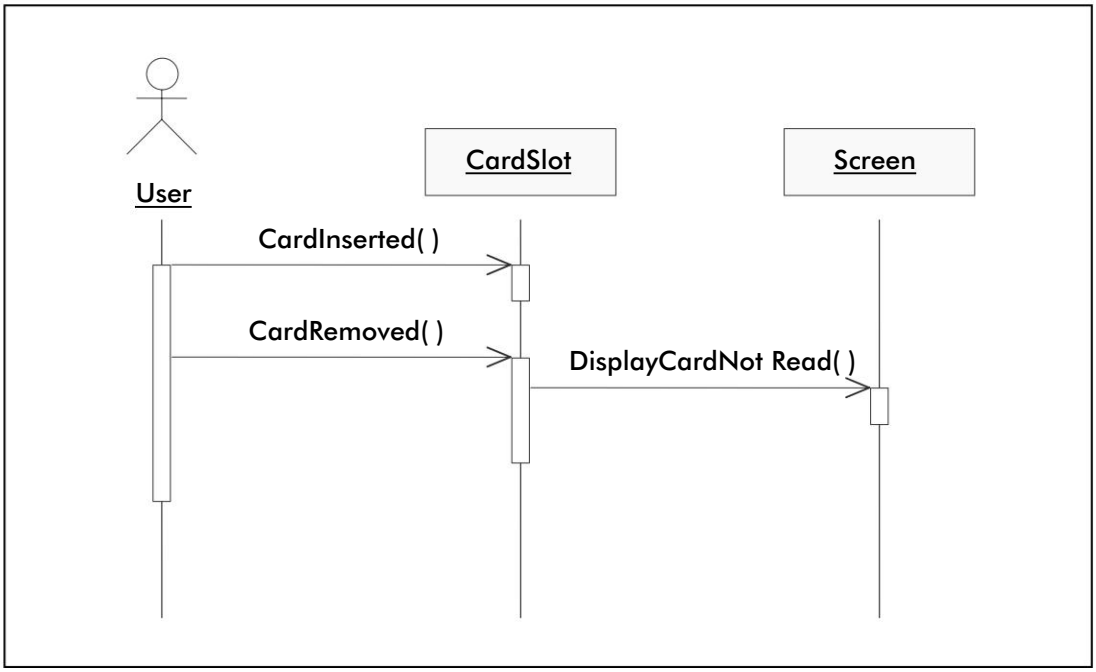


Fig. 9.9 *Card not read*

In order to arrive at an architecture, the card verification system can be divided into a number of components:

- The **Read Card** task reads the code on the card and then generates an interrupt. It sends an interrupt when a card is taken out. The system can then have a task dedicated to this input.
- The **Manage Door** task keeps track of opening the door. It may have a sensor to gauge if the door has not been closed for a long time. This may raise a small alarm.
- The **Comp Card** task handles comparison of input card code with the list of authorised codes stored in the system.
- There should also be some mechanism to update this list. Most customers will prefer this activity to be done through some software, which has the option of deletion, addition and alteration of codes in the database. This software should have the capability to check the super-user card so that this option is not given to everybody. So the CompCard task compares super-user card number with the card code received from ReadCard and performs updating functions if they match.

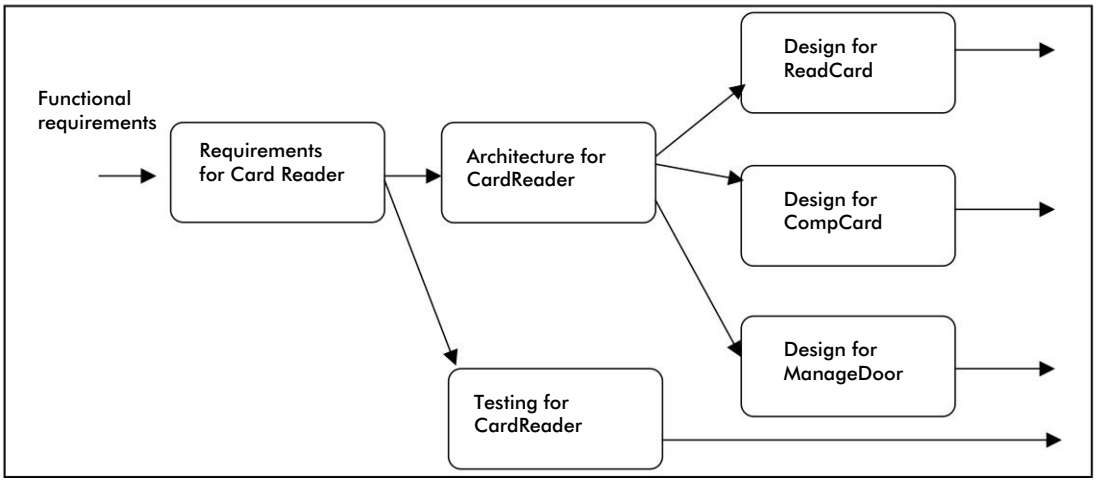


Fig. 9.10 *Stages of development of card reader*

Figure 9.10 illustrates how the various stages of development of this system are connected to one another. Figure 9.11 gives a possible architecture of this system.

As we can see in this diagram, the interfaces have been identified from the point of view of implementation. Once, this arbitration is achieved, each of these tasks can be

designed independently and in parallel and integrated after implementation. The figure shows some of the major types of interface prevalent in embedded systems: interrupt, message, function-call and callback.

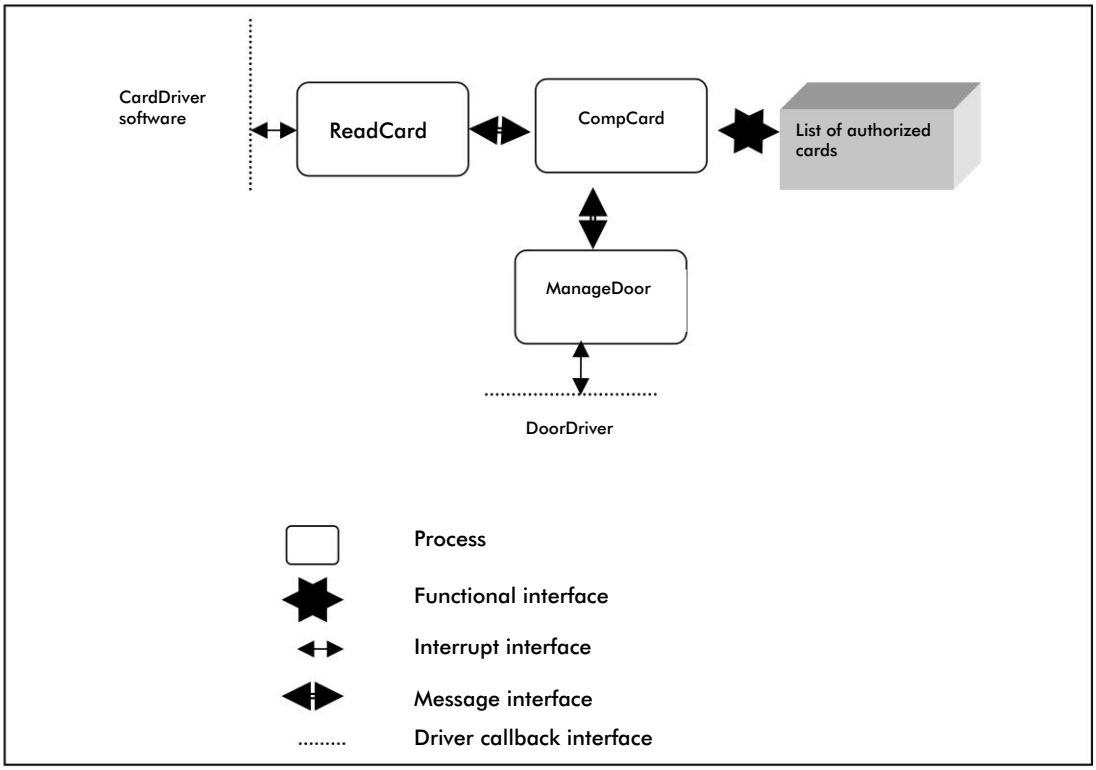


Fig. 9.11 *Architecture of card verifier system*

An architectural diagram shows potentially the top to down decomposition of a large project. Each element inside this architectural diagram is presented in the form of an architecture document and a detailed design document. These architecture and detailed design documents may, in turn, be further divided into subelements. These are then presented in the same way in a hierarchical fashion.

9.5 PRACTICES FOLLOWED IN DESIGN

9.5.1 Design approach

The architecture and design of a system can be done in two approaches. The data oriented approach focuses on the data structures of the entire system and their management. The process-based approach decomposes the system into units and processes,

and identifies the relationship and interfaces between them. These components may further be divided the same way. It is seen that the combination of both approaches results in a good design of the system.

9.5.2 Don't be complacent

A very common mistake seen among embedded designers is the tendency to throw all caution to winds if a fast processor or more memory is available. This is probably because these things are items of luxury for normal embedded systems and any excess is basically removing the restriction of the system being too "embedded". However, the architecture and design should be optimised from the word-go. It would take much more manpower and cost to optimise the system once it starts operating and we identify that the system is crashing at sporadic times because of memory overrun or if the performance of the system is not adequate. This is especially true if there are any chances of porting the embedded system to another processor. An advice in this direction would be to start developing architecture and design over a slow processor and small amount of memory, make the design optimised and then try to extract the benefits of fast processing speed over this design. Otherwise, embedded programmers have a tendency to be lazy when luxuries are available!

9.5.3 Analysis of the worst cases

The design should take into account the worst and best case timing of all the scenarios identified in the requirements phase. This will have an impact on the way we are breaking up the system. Some tasks may be made high priority if they are required for an urgent response and are prevented from running because some other processes hog the processor time.

9.5.4 Processing of data

The design of an embedded system takes a look at the data of the system. The data may be received from outside. This is true typically in communication related protocols running on embedded systems where the majority of data the system has to work on is received from outside. For example, a mobile phone is expected to periodically read information through its receiver from the network on the air interface*[1] and act on this information to decide about its area of service,*[2] among other things. Other systems may have majority of data in-built. For example, the card verification system has most

*[1]A mobile phone accesses the network over the air. For this it needs to get synchronised to the base station in time so that the information being broadcast is accessible and readable.

*[2]The coverage area of the base station to which the mobile station is listening governs an area of service.

of its data related to authorised cards built-in. This data is not expected to change while the system is being accessed. This data has its own limitation that it is potentially huge and secondly, what matters is the time in which a new card number is searched in this list since, that is what governs the availability of the card verification system for the next user. In case of mobile phone, the limitation is to read the data completely and correctly as it occurs on the air interface, and then take actions based on this data in a realtime manner before the processor time is allocated to other. Therefore the module that is receiving this data needs to be given high priority so that the data can be read at proper times and no other trivial actions halt this operation. Then the modules that act on this data need to run uninterrupted (for example by application modules).

9.5.5 Decomposition

A typical embedded system is first divided into layers and/or tasks. Sometimes these layers are defined in the system by governing standard bodies. This is in case the embedded system is supposed to interwork with other systems from other vendors. Bluetooth, Wireless LAN and other wireless mobile phones are examples of such standard systems. These layers are present to perform specific actions and are usually based on the ISO OSI mechanism for defining a protocol stack (See Fig. 9.12).

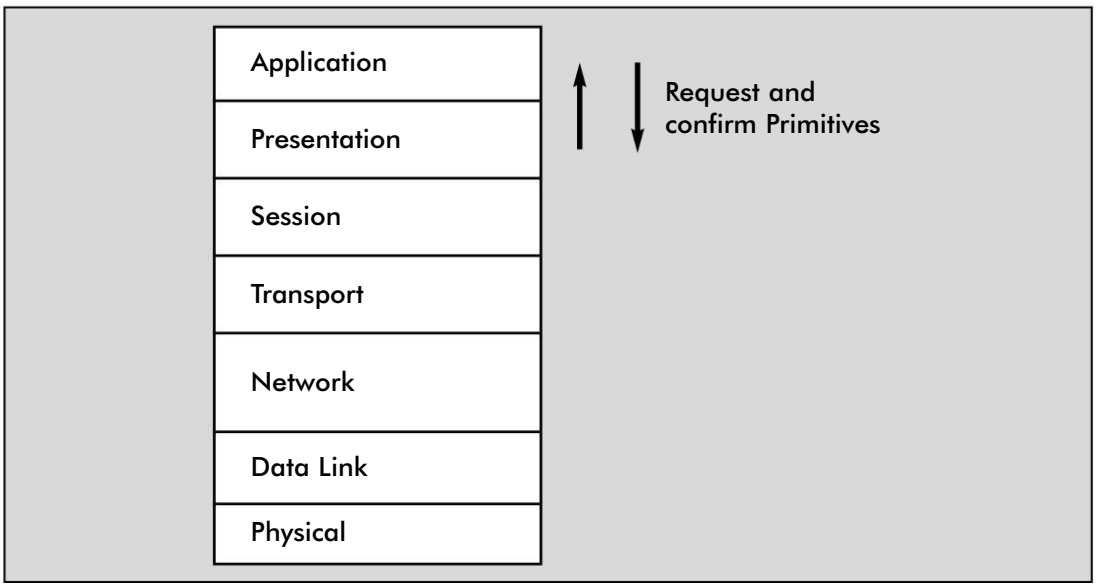


Fig. 9.12 *OSI layered architecture*

One layer provides services to the layer above it and receives service from the layer below. One or more layers are implemented in one task inside the system. These tasks are then given priority based on the criticality and realtime behaviour of the actions performed by them.

Typically, lower layers are implemented in hardware, however, that may depend on the factors listed in the beginning of this chapter. These layers perform the very basic hardware functions related to electrical characteristics of data bits, checksum, error detection, etc. Basically, they are responsible for providing a transport mechanism to the upper layers that then implement a hand shaking mechanism and other flow control algorithms to take care of application data.

9.5.6 Heart-beat in distributed systems

Since communication systems have to deal with the outside world that is not in direct contact with the system, the lower layers typically employ some heart-beat mechanism to indicate to the adjacent node about the fact that the system is running smoothly. An example in this approach is the use of FISU* messages in SS7 MTP layer. These messages are sent to the adjacent nodes whenever there is nothing else to send so that the other node gets to know about the health of this node.

Similarly, many times the system itself is divided into parts that run physically apart from each other, the heart-beat mechanism is required to keep track of health of the complete system.

For example, in Fig. 9.13, the system is divided into three parts: node A, node B and node C. If node B crashes down because of overload or other factors, there is no way that the other parts will come to know about it on their own. They will keep on pumping data to that part unnecessarily. This will introduce undesirable behaviour into the system. Instead, if all three nodes are designed in such a way that they send some data on all interfaces at a given minimum rate, the overall health of the system is available everywhere. The parts may also give some other statistics to each other like system load that may be used to perform flow control to avoid a potential congestion condition. These same messages can be piggybacked with acknowledgements of received packets. Another advantage of using heart-beat mechanism is utilised inside switching networks in telecommunication (SS7 for example). In such switching networks, there is a high degree of redundancy with respect to routing of packets. For example, node B

*FISU: Fill-In Signalling Unit messages. In SS7, this heartbeat mechanism is used at MTP layer level. All nodes send this empty message to each other whenever nothing else is present to be sent.

and node C are providing two routes to the same destination node D. So at any given time, node A can decide that the health of node B is not particularly good, and hence can change the routing to node C and help the node B in recovering from congestion.

In wireless system, the mechanism of heart-beat is used for a different purpose. Since the resources on the air are very precious, they cannot be allocated to all users all the time. The air interface, however, is particularly prone to bad radio conditions and potential drop in quality of channel. Hence if heart-beat is not received from a particular user, the resources assigned to that user are released and then they can be potentially allocated to other users. The heart-beat and associated procedures are typically a part of lower layers inside the protocol stack. Once the quality of link is guaranteed, the higher protocol layers use more sophisticated methods to deal with selective retransmissions, or retransmissions after a particular number of packets inside a huge stream of data (including ARQ and selective repeats).

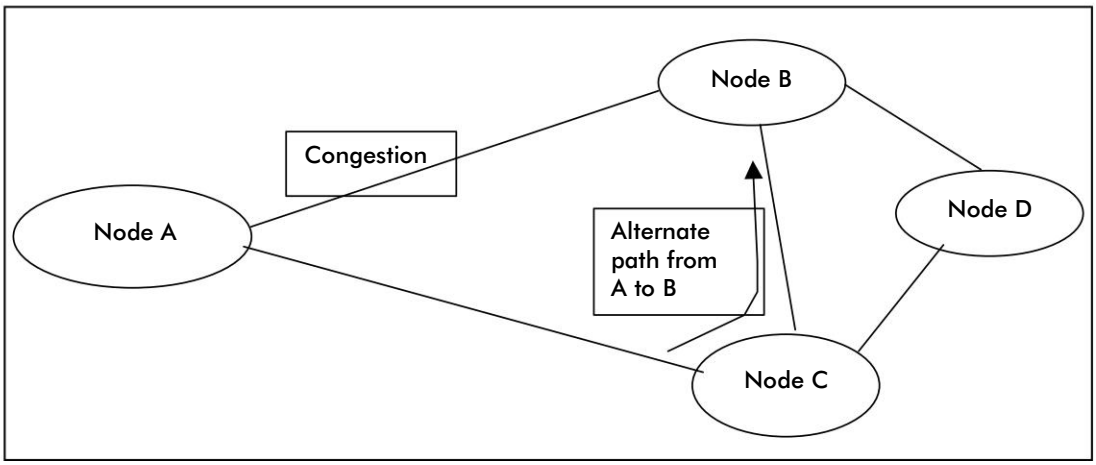


Fig. 9.13 *A typical network*

9.5.7 Assignment of priorities

Once the system has been broken down into tasks, these tasks are then assigned system priorities so that the RTOS understands the relative importance of each task. Each task is assigned a stack space for its internal use. How much stack space should be assigned is just about anybody's guess☺. It takes the combination of experience and intelligence to assign some space to a task. However, it should be kept in mind that there are some factors that result in a lot of stack space. For example, if recursion is being used, there are chances that the space may be a little under strain. In embedded systems, it is usually a bad practice to have recursion since it can easily become a runaway horse out of

control. Many RTOS keep a limit on the number of messages that can be placed in the mail-box of a task. Many RTOS also keep a maximum limit on the number of tasks in the system. This limit however, is usually very large.

9.5.8 Impact of software development method on design

The design methodology depends on the kind of software development model. One is called the waterfall model and the other is the iterative approach.

Waterfall model

The waterfall model is a stage-based approach to implementing a system. First a requirement is taken, and then a design is provided for this requirement, following which it is reviewed and finally the implementation begins, followed by testing of the module. Figure 9.14 gives these different stages of a waterfall model.

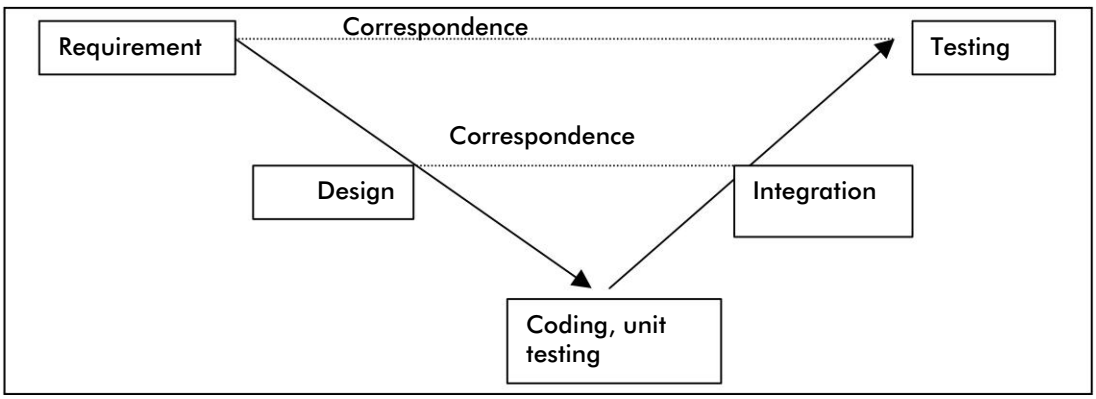


Fig. 9.14 *Waterfall model of development*

Iterative approach

In the iterative approach, on the other hand, the module is looked at from all perspectives in the beginning itself. So we have use-cases that specify requirements and architecture. The class diagrams throw light on the design, etc. In this way, it is possible to analyse the system at a much greater level in the beginning itself. Since most of the problems in the field occur due to unforeseen errors during design and unfortunate crossover of events, the UML approach seems to be better in countering this threat. However, the proponents of waterfall model argue that UML based design sometimes becomes too detailed in the beginning itself. As usually is the case, the combination of both approaches should generate better results. The waterfall model should be used while keeping the advantages of UML related to thinking in terms of use-cases, class diagrams, etc. to visualise and represent parts of the system.

9.5.9 What is the best architecture?

One case in the point while designing a system is the tendency on the part of designers to jump at the first reasonably well designed architecture and start implementing it head-on. While this approach is good since the system may work reasonably well, it is advised that alternate architecture be thought of and compared. This gives an idea about the benefits and drawbacks of each approach. Then, looking at the factors that govern the design alternate approaches can be weighed and compared. Essentially, most of the approaches will work. However, they may vary on merit on a scale of the relative importance of factors, which govern the design and architecture. We will consider an example. Consider a communication system requiring a task that needs to take care of some link management while no communication is running. Additionally there are some tasks related to receipt and discharge of data during communication and some jobs related to making checks related to quality of service link and sending these reports to the other node. Here we can potentially have a lot of architecture ideas. We may have the factors that govern the design and their relative importance as high, medium and low, according to a given implementation. Table 9.1 may be one such example.

Factor	Weight
ROM	Medium
RAM	Medium
Maintainability	High
Ability to evolve or get extended	High
Portability	Low
Effort	Medium
Efficiency	High

Table 9.1: ***Weighted factors for architecture evaluation***

What constitutes a good architecture is rather debatable. Each architecture may be good or bad depending on what combination of its driving factors are satisfied and what trade-offs have been used. The driving factors such as the ones above decide the suitability of a proposed architecture. The weights attached to each factor weighs the architecture propositions and identifies the trade offs involved.

Now we can look at three different ways of breaking the tasks of the system.

- Approach 1, in which we have a separate task for everything. That means three tasks, one for housekeeping during idle mode, one for data transfer during the

connection, and one for measuring quality of service, and then initiating the connection process for sending a report to the other node.

- Approach 2, in which there is only one task handling everything.
- Approach 3, in which there is one task for handling quality of service issues, and one more task, which takes care of everything else.

All these approaches may be appropriate in the light of the factors listed above.

9.5.10 Better safe than sorry!

When the state machines are being developed for each task of the system, the designer should identify spurious messages or events in each state. Since we are dealing with an asynchronous system, the task may have changed its state before another input meant for the previous state arrives. This input may not be valid for the current state. For example, in a mobile phone system, it is possible that the network starts paging a mobile phone and at a slightly later instance, the user of the phone initiates a call. In this case, since the phone is already "occupied" while processing information and performing activities related to receiving the incoming call, the request for an outgoing call is not valid and hence trashed at this particular moment. The same request was perfectly valid when the incoming call was not received. This is not a problem with state machine. This is not a problem with the embedded system. This is not a problem with RTOS. This is just a case of unfortunate crossover of messages. At the design stage itself, such crossover cases and spurious inputs possible in different states of the system should be identified and some proper processing should be performed. In the case of the mobile phone for example, it may suffice to just give an indication to the user that the phone is busy or an outgoing call is not possible at this time. This error handling is required and recommended at all levels of design and implementation. The system should be capable of handling interrupts at uncertain and uneasy times. The system should take into account, corrupted data bytes received, especially when the sender of information is physically removed from the destination. A lot of effort is especially reserved in wireless systems for this since they are particularly prone to corruption on the air. However, this applies to any two systems separated by a distance. The LAPD protocol used in ISDN lines is another case in point.

9.5.11 Watch the memory usage

At the beginning of testing of an embedded system, the norm is to keep a close watch on the memory being used by the system. This memory can easily become a pain in the neck if at some creepy corner of the code, it is going out of bounds. In this case,

some modules may have to be optimised with regards to their usage of memory. This is also required to judge the optimum allocation of stack space to different tasks. When the RTOS starts execution, all stack space is initialised to some constant number say, 0×55. During its execution, the realtime system uses varying amount of stack space. After a lot of trials, the unused space that still contains 0×55 is freed for some other use.

9.5.12 Polling or interrupt?

We looked at this question in Chapter 7 as well. A major decision at the firmware and hardware level is to use either the interrupt mechanism or polling to identify when some data has arrived. The choice stems from two different approaches based on the kind of system we are dealing with. In the card verification system, the system is doing the job of reading a card, searching for it in the database, displaying a message for a given amount of time, or unlocking the door lock. And it is expected that the next user has to wait till any of these actions are in progress since the device has only one slot. Hence it would suffice to keep a polling mechanism in this system. The system runs in almost a serial order, once a loop of action has been finished it checks for the polling bit set. If so, it performs another cycle of the loop. If not, it waits for the bit again.

Alternatively, the system can be implemented such that it expects to be interrupted when something arrives. In that case, the system continues its execution and then waits for something to happen; in this case, it is the interrupt. In systems which expect a lot of inputs from a lot of channels, it would make sense to make them interrupt driven if the data coming in can potentially be overwritten by more data.

Once the system has been broken into tasks having fairly independent jobs, it is time to start designing the task itself. Each task is made up of executable code that determines how a system will behave on the occurrence of external events. Usually tasks in realtime systems are implemented as state machines because most embedded systems are inherently of the following kind:

- Expect an internal or external event in the form of an interrupt, message or timer
- Perform an action
- Generate an event for the same or some other part of the system
- Again wait for another event.

Also the subset of events that the system is waiting for is not the same during its entire execution. So the different tasks are generally organised in a series of states expecting specific inputs, and ignoring the rest.

9.6 DESIGN CHECKLIST

A good design document contains the following:

- ***Author (list):*** Author(s) of the document. As for requirements, one author should be identified as the owner of the document. S/he updates the document and maintains it throughout the duration of the project.
- ***Technical control or review team:*** Identifies the team, which reviews the document for the sole intention of checking the technical suitability of the document. This review takes special focus on effect of external interfaces on the module.
- ***Distribution list:*** List of teams or people who get affected by the design document. These possibly will be other modules that have interface with this module.
- ***Status of the document:*** Whether it is a draft, or under review, or released.
- ***Project Name:*** This identifies the collection of activities for a particular goal.
- ***Date:*** mentions the last time the document was changed.
- ***Version:*** identifies the evolution.
- ***History:*** lists the changes that happened in the past with an overview of their impact and the author of these changes.
- ***Abbreviation list:*** lists the abbreviations used in the document. It may give a reference to another document. This may give a reference to architecture document or requirements document.
- ***Definitions:*** Introduction to the terms used in the document. They may be standard or particular to the document. This may give reference to requirement document or architecture document.
- ***References:*** Any related information can be found in the list of references.
- ***Perspective of the system:*** Defines the external interfaces of the system and the kind of inputs that are expected in the system.
- ***Data structures:*** Based on the interface and expected behaviour of the system, the set of data structures are defined. These data structures are the core of information processing for the module. It is advised to provide this section with attributes like initial values of variables, their scope, conditions under which they will change, etc.
- ***State diagram:*** The high level state diagram of the system identifies the static design of the system. This gives all pairs of stimulus-response expected in the system.

- ***SDL:*** The dynamic behaviour is provided by the representation of the system as current state->stimulus->action->response->final state mapping. The output messages may be to other modules inside the system or to the same module.
- ***Timers, buffers:*** The design document lists the timers used by the module, the conditions under which they are set, reset and when they expire. This document gives the values of buffer lengths and constants used inside the module.
- ***Pseudo code:*** A detailed design will usually contain pseudo code, which is a semi-English series of statements and comments to explain the flow of execution of control of the module.
- ***Exported and imported procedures/interface:*** The module may be using procedures defined in other modules. These should be documented here. In addition, some procedures from this module may be used by other modules. All such interface to the external world should be documented so that the user of the interface gets it from one place.
- ***Callbacks, APIs:*** If the module uses or provides callback functions and APIs, design document should identify them.
- ***Error Handling:*** Most embedded systems have the peculiarity of being 'always-on'. This may give rise to stringent needs related to unforeseen errors. Also, as we saw in the last chapter, special cross-over cases may give rise to race conditions. Such cases should be carefully thought of and documented inside the design document.
- ***Cross-reference:*** Finally, all requirements inside the requirements document need to be implemented. To track this, the design documents usually contain a section, which relates the sections of requirement document being addressed by sections of design document. Alternatively, a separate cross-reference document can be made which gives the relation between requirements, design, implementation and testing.

9.7 LESSONS LEARNT

Architecture of a system is like a blueprint of the system. It concerns itself with identifying the components of the system, interactions between them and managing trade-offs based on priorities. Each of the components can then be individually designed (possibly) independent of each other. A lot of architecture styles are available, each focusing on some priorities over others. While architecture has evolved over the years, the architect community has documented architectural patterns that have been known to solve

some typical problems that occur in architecture definition. The design of the system heavily depends on its priorities. This chapter gave examples of different choices available to the designer based on the kind of system being developed. In the end, the design document should be traced from the requirements and should clearly define static and dynamic behaviour of the system through SDLs/UMLs and handle possible erroneous/cross-over cases.

9.8 REVIEW QUESTIONS

- Compare the architecture styles based on their ability to create portable software.
- What is an architecture pattern? How is it different from architecture style?
- Heart-beat is a common mechanism used in distributed systems. Can it be used for non-distributed systems as well? If so, how?
- How do we arrive at the best architecture for a system? How do we justify it?
- How does UML-approach differ from traditional design approach?

Chapter
10

Implementation Aspects in Embedded Systems

10.1 INTRODUCTION

Now this is one topic that all embedded engineers love. The smell of wires and sound of data inside them makes many a heart flutter with joy ☺. Implementation has its own history. In the not so old world, assembly language used to rule the roost in embedded systems. With the advent of efficient compilers in C, it became possible to get the same power of assembly language while retaining the ease of coding and understanding of a high level language. C came like a boon for the programmers especially in the embedded world and has since its introduction remained as the queen (or is king?☺) of the embedded world. Object oriented methods and languages are making heavy inroads into the world of embedded world of late, however, even today most programmers feel comfortable and secure with this language. Hence, this chapter has been written with C in mind even though most of the things are valid for any other similar high-level language.

Ideally speaking, implementation deserves a complete book by itself. However, it would not be out of place to document some tips about implementation aspects in embedded systems in this chapter. These implementation tips are usually encountered during the daily life of an embedded engineer. This chapter should be read from the perspective of good and bad practices during implementation. Like the proverbial stitch in time, good programming habits save lot of time later.

10.2 READABILITY

The code written by an engineer should be readable and understandable by the rest of the community. In this direction, the following are helpful.

10.2.1 Proper indentation

Indentation is done to identify the flow of a program properly. Indentation helps to see the depth of a statement. Only spaces should be used for indentation. Tabs should not be used. When code is ported from one system to another, the tab settings of the second environment will most likely be incompatible with the tab settings used from the first environment (according to Murphy's laws). This turns source code into an indentation nightmare. Indenting should be used on a line which is not blank after an open brace and control statement (case, for, if, else, while). Usually four blanks are used as an indent.

For example:

```
for (var = 0 ; var < MAX ; var ++)
{
      do_something ;
      if (check_a_condition)
      {
           this_condition_is_satisfied;
      }
      else
      {
           this_is_unfortunate;
      }
}
continue_working;
exit_from_program;
```

10.2.2 Comments

Comments are very useful for readability of code. The comments should be given in sentence form, with correct spelling, grammar, and punctuation (although the terminating period is not extremely important). Good code comments should strive to tell the reader *why*, as opposed to *what* the code is doing.

Bad comment example:

```
/* Assign the value of 4 to the variable x. */

x = 4;
```

Good comment example:

```
/* Loop four times: once for each corner of the rectangle. */

x = 4;
```

Here is another example with clear, simple and useful comments.

```
/*
       All ports except the MAX_PORT port are initialised
       since the last port is reserved for use by the debugger.
*/

for (i = 0; i < MAX_PORTS - 1; i++) {
     /* Set baud rate, parity, data bits */
     InitialisePort ( i );
}
```

10.2.3 Paraphrasing of functionality

It is advisable to paraphrase the purpose of a group of lines usually contained within braces. This paraphrasing should be done in the beginning of the block. This serves two purposes. First, the purpose of the block is shown at a glance. Secondly, it serves as a micro-review of the C statements following the comments. Since the behaviour of statements should match the comments. So this helps as an early warning system for any inconsistencies in the code. This is shown in the following example:

```
  }    /* lv_len_var > LEN */
}    /* lv_brdth_var > BRDTH   */
else
{
      /* Here lv_brdth_var < BRDTH. Calculate the area of all
         rectangles and store them in global variables for
         later use. If an area is less than MIN_AREA, don't
         store it.
     */

    int lv_index ;
    for(lv_index = 0; lv_index < NUM_RECTANGLES; lv_index ++)
    {
          ga_Areas[lv_index] = CalculateArea( );
    }

}       /* else lv_brdth > BRDTH */
```

Now, because of the block comment in the beginning of else block, it becomes immediately clear what this block as a whole is supposed to do. And, it becomes clear that the author has missed out the condition to check the `MIN_AREA`.

These block comments should be written in the beginning of the development of code so that only the conditional statements and block comments are written. This skeleton is then later filled. This helps in organising the code better and there are few loopholes in the code.

10.2.4 Magic numbers

Magic numbers inside code should be avoided. They should be replaced by meaningful symbolic constants. This is because, in any subsequent change in value, the macro needs to be changed at one place thus eliminating probability of human error. Secondly, the C statements become more understandable since the macro name will identify some meaning to the value. Additionally, the same value may be referring to different logical entities. For example, 10 may be the maximum number of ports and also the maximum sockets that can be opened.

```
for (i=0; i < MAX_VAL; i++)
{
     if (gv_length > MIN_LENGTH)
     {
          SendErrorMessage(MIN_LENGTH);
          PutBuffer(MAX_VAL);
          break;
     }
     PutBuffer(i) ;
}
```

10.2.5 Positive booleans

We should use booleans with positive names. Usually the name of a boolean variable or symbol denotes some condition that is on/off, enabled/disabled, etc. Code is easier to understand if the name of a boolean variable or symbol denotes a positive condition–such as “mode enabled”—versus a negative—such as “mode disabled”. If you are testing for the boolean being FALSE, then a negative meaning causes a double negative, which is more difficult to understand. For example:

Suppose we have a boolean variable `TempNotGreater`, which has the following meanings:

`TempNotGreater == TRUE`, implies temperature is less than a threshold.

`TempNotGreater == FALSE`, implies temperature is more than a threshold.

To execute code that uses this variable, we would write the following fragment:

```
if ( ! TempNotGreater )
{
    /* Temp > threshold code here  */
}
```

The predicate inside the if statement is hard to understand. We have to say to ourselves, "OK. *Since TempNotGreater is false means negative of negative of this condition means temperature is higher*". Instead, it's clearer to name the boolean 'TempLesser', with these meanings:

```
if ( TempLesser )
{
    /* Temp < threshold code here */
}
```

Even the contrary test is easy to understand:

```
if ( ! TempLesser )
{
    /* Temp > threshold */
}
```

10.2.6 Explicit computation of macros

Let us consider the following definition:

```
#define TWO_PI 6.283185306
```

There is no reason to calculate this value by hand when the compiler can calculate possibly more accurately than us. Instead let us see this definition:

```
#define PI 3.141592653
#define TWO_PI (2.0*PI)
```

It is easier to understand the intent and be more accurate. Let us say we are defining the center of a rectangle, consider these two possible definitions:

```
#define RECT_LEFT 100
#define RECT_RIGHT 200
#define RECT_MIDDLE 150
```

Instead, if it is defined as:

```
#define RECT_LEFT 100
#define RECT_RIGHT 200
#define RECT_MIDDLE ((RECT_LEFT+RECT_RIGHT)/2)
```

The latter definition helps us understand the meaning of `RECT_MIDDLE`. Also, the latter definition enables us to change `RECT_LEFT` and `RECT_RIGHT` without explicitly recalculating `RECT_MIDDLE`.

10.2.7 Documentation of bugs

There could be compiler bugs that could have been overcome by some way. For e.g. some compilers provide `__int64` as an implementation for a 64-bit integer. (To use such compiler provided features or not could be a separate discussion). In some compilers, ++, += operator does not work with them.

It is always advisable to comment on such bugs so that some other programmer does not change it back. (After a few months even we may forget why we did that!)

```
__int64 timer_register_value = 0;


/* timer_register_value++ does not work ! Compiler bug!
*/
timer_register_value = timer_register_value + 1;
```

10.3 FUTURE MAINTENANCE

In order to aid future maintenance of the code, some points are noteworthy.

10.3.1 Global variables

While designing and implementing a system, it is advisable to minimise the use of global variables. Global variables are difficult to manage because of their scope in the system. It is difficult to limit the usage of global variables in the system thus exposing the system to more possible errors. Even if global variables are used, any usage and

change of the variables in a function should be documented at the beginning of the function inside the function header.

10.3.2 Use and misuse of macros

The classical debate while writing the code is the relative merit of using macros and functions for particular tasks. While functions are slower than macros, the macros are difficult to debug and understand. Hence, it is advisable to create macros with small functionality inside them. The macros should not change global variables in general. They should not be used for doing actions. They should typically be used for returning a value based on some conditions. Macros are suited ideally for making small calculations the results of which can be assigned to some variables from the place where the macro is called. For anything extra, it should be kept in mind that macros are difficult to debug and hence can possibly wreak havoc with the system.

We should enclose macros in parentheses. Also, inside each macro, we should enclose parameters in parentheses. Unless our macro is EXTREMELY simple, we should put parentheses around the body of the entire macro. (Or braces "{}" if the macro is a statement.) The only exceptions are strings that are numeric, character, or string constants. If we don't use parentheses, operator precedence may cause the macro to produce unexpected results. For example, refer to the following macro definition:

```
/* perimeter of a rectangle */
#define MYCAL(var1, var2)        3*var1 + 2*var2
```

There is a problem in the definition of this macro. If we call this macro as MYCAL(v1, v2)/2 or as MYCAL(v1+v2, 30), operator precedence will create different result than expected.

The best way of expressing the macro above is the following:

```
/* perimeter of a rectangle */
#define RECTPERIM(width, height) (2*(width) + 2*(height))
```

10.3.3 Variable reduction

In the embedded world, memory is at a premium. Consider the following function:

```
int Mycompare1( char * testarray )
{
  int return_status = 0;

  if( strcmp(testarray, "TEST" ) == 0 )
    return_status = 1;
```

```
    return return_status;
}
```

The result of the expression enclosed within an `if` statement must be either true or false. Thus the code can be simplified:

```
int Mycompare2( char * testarray )
{
  int return_status = (strcmp(testarray, "TEST" ) == 0);
  return return_status;
}
```

One more simplification can be done as below:

```
int Mycompare3( char * testarray )
{
  return (strcmp(testarray, "TEST" ) == 0);
}
```

This looks more like an embedded function ☺.

10.4 PERFORMANCE

Though compilers perform a lot of optimisation, the way we write code can also affect the performance of code.

10.4.1 Use of bit-shifting

Let us look at the following code:

```
for( index = 0; index < MAX_PORT; index += 5 )
{
  x = index * 8;
  y = index * 11;
}
```

The above code can execute much faster using the bitshift operator, as follows:

```
for( index = 0; index < MAX_PORT; index += 5 )
{
  /* x = (index * 8); y = (index * 11) */
  /* multiplication by 11 = multiplication by 8
     plus multiplcation by 2 plus x
  */
```

```
    x = index << 3;
    y = (index << 3) + (index << 1) + index;
}
```

Division can be implemented using the right shift operator and modulus by a power of two can be obtained by performing the *binary* and operation (&) by the same number, less one, as follows:

```
for(index = MIN; index < MAX; index += INCREMENT)
{
    /* x = (index % 8) */
    /* Y = (index % 32) */

    x = index & 7;
    y = index & 31;
}
```

Usually bitshifting is faster than multiplication, division and modulus. The comments inside the code are useful to describe the result to fellow developers.

10.5 MISCELLANEOUS TIPS

10.5.1 Byte stuffing

Generally compilers store data in alignment with even byte boundaries. Any extra bytes in between are allocated and remain unused. This process is called byte stuffing. This is done because it is quicker to address data on even address boundaries. Consider the following definition.

```
typedef struct my_struct
{
    char first_char;
    int  first_int;
    int  second_int;
    char second_char;
}
```

If char is one byte and int is two bytes, a total of 8 bytes will be allocated for this structure, one extra byte after first_char and one more at the end of second_char. This structure could have been optimised in the following way.

```
typedef struct my_struct
{
     char first_char;
     char second_char;
     int  first_int;
     int  second_int;
}
```

In this way, the byte stuffing is avoided and only 6 bytes are allocated for the structure. It is advisable to look at the documentation of the compiler to optimise RAM further.

10.5.2 Bit fields

Since embedded systems are forever short of RAM, we can make use of the facility of bit fields provided in C to optimum use. A lot of flags and variables do not take the maximum values for which static allocation is provided for them. So, only specific number of bits can be allocated for these flags thus saving valuable RAM. The optimisation is tremendous if these flags are a part of an array. Consider the following example.

```
typedef struct bit_field_struct
{
     u8 one;
     u8 two;
     u8 three;
     u8 four;
     u8 five;
     u8 six;
     u8 seven;
     u8 eight;
}

bit_field_struct wasted_array[MAX_NUM];
```

If the flags above are used to store some status values or similar operations, we can save seven (because of byte stuffing) bytes by creating them using a bit each. We can use more bits for flags that have more possible values too. Since we are wasting memory in structure definition itself, any array like `wasted_array` above created through `bit_field_struct` will replicate this wastage. Consider the following code instead.

```
typedef struct bit_field_struct
{
     u8 one:1;
     u8 two:1;
     u8 three:1;
     u8 four:1;
     u8 five:1;
     u8 six:1;
     u8 seven:1;
     u8 eight:1;
}

bit_field_struct wasted_array[MAX_NUM];
```

In this way, we allocate a total of `MAX_NUM` bytes instead of 8 * `MAX_NUM` in the previous listing.

10.5.3 Optimum stack sizes

Stack sizes must be calculated and validated. In many cases, to meet tight schedules, the stack size required per task/process is not put in sufficient scrutiny. So, to be on the '*safer*' (!) side a large buffer is added to the stack sizes so that the program does not crash in realtime.

Since memory is expensive, the stack sizes MUST be calculated approximately even in the worst case. Though theoretical calculations are preferred sometimes to get accurate theoretical calculations might be extremely tough or even unnecessary. We can use some techniques that are more on the practical side:

- ***Use Processor Options:*** Some processors come with the option of software stack size checking. Set stack sizes to minimum expectation. Whenever stack overflow exception occurs, increase task stack size by say 1K and repeat the experiment. This method should take a day or two depending on the complexity of the system and the number of tasks in the system. This method should be used only when the software is stable and when not much of the coding is required.

- ***Watch points:*** Watch points are features provided by the debugger that will stop execution of the program whenever a variable or content of a memory location changes. (Though this could be used as a valuable debugging technique, we'll see how it can be used to determine stack sizes).

The MAP file provided by the linker (or some tools provided by the RTOS vendors (e.g. pSOS Awareness)) can be used to find out the location where the stack of a particular task begins. We must remember that the stack grows downwards.

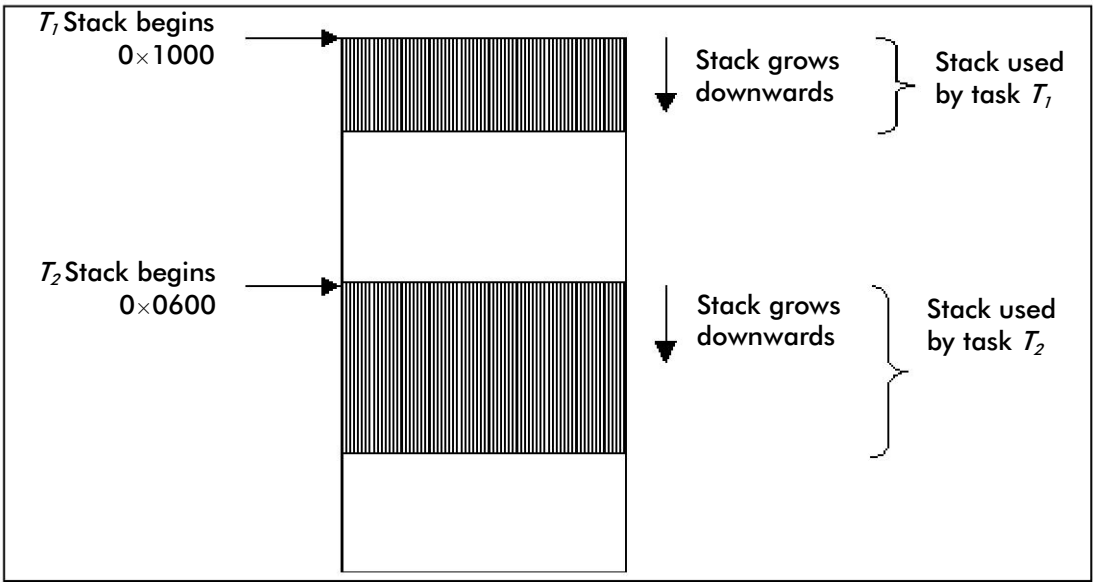


Fig. 10.1 *System stack*

We can have watch points where the task stack ends. (For e.g. 0×0601 in the above picture). The program execution will stop whenever stack grows to its limit.

- Another way is to fill the entire stack area with a pattern e.g. 0×DEAD or similar. Now let the program run for some time. Now if we examine the memory we can see how much the stack was unused by seeing the memory dump of stack region. Based on this the stack size can be either increased or decreased.

The best way is to use a combination of all the three methods.

10.5.4 Endianness

Endianness refers to the representation of multibyte variables inside embedded system as low-order first (called little endian), or high-order first (called big endian).

Care should be taken to maintain the endianness throughout. An endianness difference can cause problems if the processor tries to read binary data written in the opposite format from a shared memory location or file.

Consider the following example of a little endian processor:

```
short x = 1 ;
short z = 0 ;
short y = 0×FE;
```

In case of a little endian architecture the memory location will look like the following:

```
Memory address      Contents
0×1000              01 00 00 00
0×1004              FE 00 00 00
```

Notice that the value is stored contrary to how humans tend to read the two bytes. So, the programmer should be a little careful when accessing the contents of this memory directly through a pointer since it may result in an erroneous interpretation of memory.

10.5.5 Compiler-optimiser

Compilation is followed by an optimisation step in which redundant and useless code is removed. Normally, it works in the benefit of the programmer. For example, consider the code in Listing 10.1.

```
{
    int a = MAX_VAL ;

    if( (a == MAX_VAL) && (status == TRUE))
    {
        do_something_please( );
    }
    // … Continue doing something
```

Listing 10.1: ***Redundant code***

Here "a" has been defined and assigned a value of MAX_VAL. The program compares the same value again in the next statement. This is a perfect candidate for optimisation and most optimisers will remove the comparison statement a == MAX_VAL. Makes sense. The problem comes if "a" is a shared variable and can be potentially changed by another task. So after "a" is assigned a value of MAX_VAL above, an unfortunate task switch happens and the other task changes its value. Now, we want to call

the function do_something_please( ) only if a is equal to MAX_VAL, however, the desired action fails here. Optimiser had good intentions but could not be out pal here.

```
{
     volatile int a = MAX_VAL ;

     if( (a == MAX_VAL) && (status == TRUE))
     {
          do_something_please( );
     }
     // … Continue doing something
```

Listing 10.2: *Use of volatile keyword*

In such cases, we can use the volatile keywork (see Listing 10.2). This keyword tells the optimiser that the variable associated with this keyword should not be optimised at any cost since it is "volatile" to optimisation and can create undesired results.

Another place where we should disable the optimiser is when we are reading from say a memory where data is changing, (for example a memory-mapped I/O). So we will be constantly accessing the same memory location through a pointer possibly in a loop. Most optimisers will disable reading the memory again and return the contents read for the first time. Making the pointer volatile solves this problem.

10.5.6 Different implementation choices

Embedded systems have been implemented in a lot of ways. It depends on various factors like ROM requirements, efficiency, modularity, source code transparency, up-gradation needs in the future, etc. Since embedded systems are usually message based, they qualify for being Mealy machines. The implementation depends on the where messages are handled and what actions are required based on the messages.

Callback systems

Callback systems are the forefathers of today's application frameworks. They work on the principle of "*inversion of control*" i.e. "*we will call you, don't call us*".

In this kind of system, we implement a few functions as specified by the callback system and these functions are called whenever required. A classic example to callback (not call back system) is the comparison function we pass to quicksort (`qsort()`) routine. (quicksort what?)

One real world example could be when you implement a Network Interface Card (NIC). The NIC usually implements the MAC (Media Access Control) of Data Link

layer and PHY (Physical) layers of OSI stack. This should integrate with the LLC (Logical Link Control) already available in the host.

For e.g., if the host wants to transmit a packet using the NIC, what does it do? The NIC MAC layer can implement a callback exported by the LLC. And LLC would call this function whenever it wants to transmit a packet. This makes the software in the driver independent of the actual conditions under which the function will be called by the application. It is important to note that even if we implement the callback function, we don't call it directly. It is called as and when required by the systems (or the application framework).

State machines

State machines seem to be ubiquitous today and their applications are wide and varied. It might seem strange that these state machines are not an old concept in computer science! State machines are spin-offs from finite state automata theory.

This theory has revolutionised the field of compilers. It should be interesting to observe that, the earlier compilers did not allow nesting of expressions more than a certain level because they could not parse those expressions. The compilers grew tremendously powerful after the introduction of automata theory.

State machines rule the protocol world. Almost every protocol in telecommunications and networking fields uses state machines. The extensive use of state machines has led to creation of a language called SDL (specification and description language). SDL extends the functionality of state machines and provides a lot of additional features.

In some operating systems there is direct support for implementing state machines. This is OS specific. But for other RTOS' we have to implement our own state machines. In this section we will explore ways of implementing them.

Simply put, state machines can be considered to be a collection of:

i. States
ii. Transitions
iii. Actions

Consider a networking protocol where two models of networking are possible, say peer to peer (P2P) and client server (CS). And at any instance only one mode operation is allowed.

We can represent this using the following state diagram:

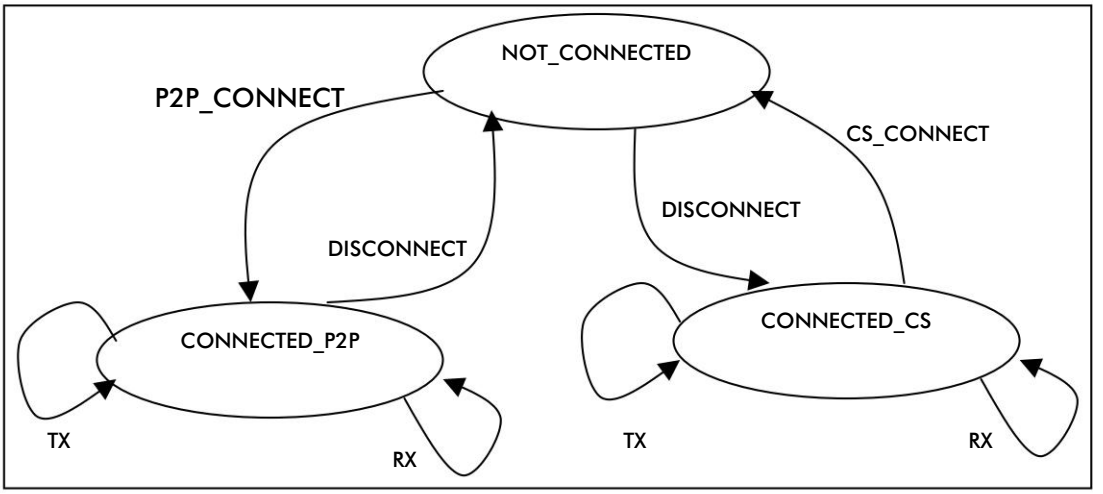


Fig. 10.2 *Sample state machine*

At any point of time, the protocol could be in any of the following states:

i. NOT_CONNECTED
ii. CONNECTED_P2P
iii. CONNECTED_CS

The state transitions occur due to events that could be triggered by the environment or could be triggered internally. (e.g. timeouts)

State transitions could also occur because of setting/resetting of a flag. This is illustrated by the following example.

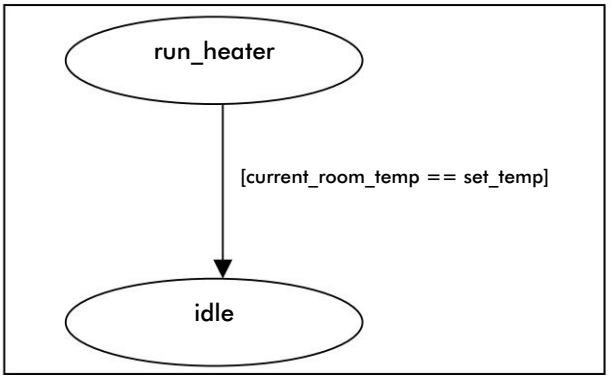


Fig. 10.3 *State machine transition based on conditions*

The above state transition (that could be part of temperature controller in a house) is caused when the condition `current_room_temp == set_temp` evaluates to true. But, in most of the cases, the transitions are caused by events or by signals as explained before.

Each signal is associated with a signal handler that performs the action to be done on receipt of a signal. A signal may or may not cause a transition.

It is very important to note that the transition (internal or external) occurs only AFTER performing the action. The temporal sequence is as follows:

i. Event occurs

ii. Action is performed

iii. Transition occurs (internal or external)

This is explicitly noted here because, many people either associate the action with the transition or assume that it happens after transition. This ambiguity can be removed by thinking of transition as the final step of action.

Now, we will explore two ways of implementing state machines. The two ways described here are:

i. Using switch-case construct

ii. Using function pointers

Implementing state machines using the switch-case construct: Here, we create various states and signals using `enum` or `#define` and use a switch case construct to implement a state machine.

Let us consider the case of networking protocol illustrated in Fig. 10.2.

The three states can be defined in a header file (StateMachine.h) as:

```
typedef enum ProtocolState_ {
      PS_NOT_CONNECTED,
      PS_CONNECTED_P2P,
      PS_CONNECTED_CS
} ProtocolState;
```

PS_ prefix is added as the acronym for ProtocolState. This can be replaced by the project name.

The signals are also defined in the same header file:

```
typedef enum Signals_ {
     P2P_CONNECT,
     CS_CONNECT,
     DISCONNECT
} Signals;
```

Let us assume that these signals are appropriately defined in StateMachine.h after sufficient precautions for multiple inclusion.

There are two ways of looking at this state machine implementation:

i. In every state, various signals are handled

ii. Each signal behaves differently in different states

We will choose the first option that is widely used because of many reasons like extensibility and reusability.

```
/* static global variable that stores the current state
   of the state machine */
static UINT16 u16State = PS_NOT_CONNECTED

void StateMachine ( int SIGNAL, void* pvMessage )
{
   /* Somehow we get the message (signal) and the data
      associated with it. (Usually a pointer to a struct) */

   switch (u16State)
   {

   case PS_NOT_CONNECTED:
   {

   switch ( signal )
   {

   case P2P_CONNECT:
   {
        /* take appropriate action … */
```

```
    }
    break;

    case CS_CONNECT:
    {
        /* take appropriate action … */
    }
    break;

    } /* end of switch(signal) */

    } /* end case PS_NOT_CONNECTED */

    case DISCONNECT:
    {
       /* … */
    }
    break;

    case PS_CONNECTED_P2P:
    {
       /* … */
    }
    break;

    default:
    {
      DB_PRINT ("Unknown State");
    }
    break;

    } /* end of switch (state) */

    } /* end of state machine code */
```

Thus we can implement sate machines using the switch-case construct.

Pros:

Very simple to implement.

Cons:

If each case becomes big, the handler must be made into a separate function. Otherwise, its readability goes down drastically.

Implementing state machines using function pointers: Another way of implementing state machines is by using function pointers and state/signal matrix.

The state signal matrix for the above protocol example can look like:

State/signal	P2P_REQ	CS_REQ	TX	RX	Disconnect
NOT_CONNECTED	OnP2PReqInNC ()	OnCSReqInNC ( )	error()	...	...
CONNECTED_P2P	.....	.....	.....	...	...
CONNECTED_CS	.....	.....	.....	...	...

A typical function name could be:

```
OnTxRequestinConnectedP2P ( );
```

Signal State

It is important to note that all the signals are not handled in all the states. It is better to fill in a common error function in all unexpected signals in all the states rather than leaving them undefined. Then, we'll never know if a spurious signal occurred. Our code will crash unexpectedly.

Pros:

This system is easy to debug.

Cons:

As the states and the signals grow, the matrix may become sparse and may occupy much more space than necessary.

The designers/implementers should weigh the pros and cons of both the methods before choosing one over the other.

10.6 LESSONS LEARNT

Sound programming practices are like the proverbial stitch in time. Coding guidelines are helpful in creating consistent quality code across the team. Proper care should be taken to make the code readable, robust, maintainable and efficient. Macros are very helpful in creating well-written code, however, the programmer should use them properly, else, they have the potential to introduce bugs.

Based on the qualities of real time systems, usually they are implemented using state machines.

10.7 REVIEW QUESTIONS

- What is the pitfall while using magic numbers in code?
- Which are the ways of calculating appropriate stack sizes for tasks?
- What are far pointers? Where are they used?
- Explain the implementation of a state machine using function pointers.
- Why should there be a need to document bugs in software?
- What is meant by little endian and big endian notation?

Chapter
11

Estimation Modelling in Embedded System

11.1 INTRODUCTION

An embedded project is usually executed by a team. This team will have a goal (in terms of fulfilling the functional requirements from the customer). So, it becomes inherently important to take stock before beginning the project and try to create a picture of cost and time required.

The first step in this direction is to try to calculate the amount of effort required to convert the set of functional requirements into working code, unit test it and integrate it. This procedure is called *effort estimation.* This chapter provides an introduction to this procedure. We try to focus on the factors that make estimation inherently difficult and inaccurate. We will highlight the reasons for keeping the estimations recursive and up to date.

Though this chapter introduces estimation as an integral activity of software development, the information in this chapter is far from being exhaustive. The discussion in this chapter has been intentionally kept brief and introductory. The reader is advised to look at references given at appropriate places inside the chapter for more information.

11.2 WHAT IS ESTIMATION?

Suppose you are invited to your friend's place for a dinner party. Your friend gives you a map of the part of town he resides in, writes down his postal address and telephone number and leaves. You are supposed to be there at 8 PM. It is a nice evening. You take

a good bath, wear cologne and set off — cool breeze in the hair and not a worry in the world. Not a worry till you realise that you do not know what time to start in order to reach there at 8.

If you are in a situation like this, what thoughts will come to your mind? You may start thinking, "well, let me make a smart guess based on the facts I have at hand". You may take into account the traffic patterns in that part of the city at this time of the day, you may ask other people or your friend to give you a feeling of the time it takes to reach there. You may like to consider the status of your car before applying some alteration on the duration given by your friend. If you own a Porsche, you may want to change the duration he has specified (for example ☺). This complete process of time duration arrived at by you based on historical data and expert advice, to use the jargon, is called *estimation*. **By its very definition, estimation is not accurate**. This is because we have not taken into consideration what is actually happening on ground for arriving at this estimate, or what events will have to be considered in future to arrive at an accurate figure. We have taken only past experience into account. And past experience is based on past events, which may not be valid now. So, if you are not Nostradamus, it will be difficult for you to predict future events and take them into account ☺.

11.3 ESTIMATION IS NOT SIMPLE

It is theoretically impossible to arrive at an accurate estimate before the beginning of an activity. On the other hand, an inaccurate estimate leads to problems later. If the estimate is very optimistic, most probably, you will reach late to your friend's place. Your hosts may have been waiting for you, for quite sometime. If you make this habit, you will start enjoying quite a reputation in your social circles ☺. The estimate may have been optimistic because it did not take into account some risks or factors arising out of extraordinary situations. You did not anticipate a traffic jam, or the fact that you are new to the area so there is a possibility to miss a turn or two. Such factors may lead to an unrealistic estimate. An optimistic estimate may create frustration and loss of motivation. On the other hand, a pessimistic estimate leads to wastage of energy and time (in management jargon, wastage of *resources* ☺). You may consider the fact that there is some probability of a car breakdown, so you are prepared for the worst. This situation may never occur and you land at your friend's place at 6 p.m. itself, while they are still sweating it out in the kitchen ☺. To conclude, estimation is necessary before embarking on even such a mundane activity like searching a new route to a friend's place, what to talk of an embedded project involving possibly millions! However, it is wrought with

uncertainty and lots of dynamically changing factors. On the other hand, it should be accurate enough for proper planning. Seems like the proverbial Catch 22 isn't it? ☺

11.4 ESTIMATION IN SOFTWARE

What we saw in the last section applies perfectly to software. Suppose your customer comes out with a project and a set of requirements. You need to estimate these requirements based on three factors, namely:

- how much effort is required for it, i.e. man days,
- how many people and resources are required for its execution, i.e. cost;
- what will be the basis of acceptance of the project by the customer, i.e. quality.

11.4.1 Estimation is not accurate

Inherently, as we saw earlier, the process of estimation is not accurate. Estimation of effort for the project depends on an accurate definition of the project itself to begin with, and it will not take any dynamically changing factors into consideration. Usually, the liberty of an accurate project definition is not available, especially in these difficult days of marketing. On the other hand, it is quintessential to estimate size and effort before the beginning of the project, since it will be taken as a guiding factor for getting resources for the project. Herein lies the trap. If the estimate is overly optimistic, deadlines will be missed, customer will not be happy and team will be frustrated. If the estimate is very pessimistic, then Parkinson's law—that work expands to fill the available time—comes into play. The project will still take as long as estimated even if the project could have been finished early. This will lead to wastage of resources and money. The team may not like to be underutilised. It will also lead to blocking of resources for the next project.

11.4.2 Overly incorrect estimation is costly

Available industry data illustrates that the cost of incorrect estimation is enormous. However, due to the complex, unpredictable nature of software development, software estimation (how long, how much code, how much testing, etc.) is a very challenging task. It is a skill that requires a great deal of experience. Estimation can never be fully accurate. The hard part is that the first estimate is drawn up in the beginning of the project when the requirements are not particularly frozen or understood. And this estimate is very crucial since resource plan and scheduling depends on it.

Secondly, clients often ask for an estimate without giving an accurate description of what they want. (E.g. "I would like a vending machine program. How much will that cost?") This drastically undermines the ability to determine how long a task will take. Usually, most programmers heavily underestimate the time required, sometimes in order to get the job and sometimes from inexperience. And, as we all know, in the programmer's world, everything can be done in a few weeks ☺. Secondly, usually in practical situations, if I am in competition with other contractors bidding for the same project and the client is looking for the best bargain, I am bound to lose the bid if our estimate is larger than others'. **It is a trade-off between high cost being expensive and low cost being risky.**

11.4.3 Estimations become better over time

Software development is a process of gradual refinement. The development team begins with an unclear picture of requirements and tries to build a system based on them. As requirements become clearer, so does the implementation maturity. Because the requirements of the software are unclear in the beginning, the estimate of the time and effort needed for the job cannot be better either. As the project progresses, the estimates for the job iteratively become better. Estimations done in the beginning of a project are usually not correct and difficult to bank on.

However, it is essential for a project manager to know the effort, schedule and functionality of a project before embarking on the actual development. This knowledge is essential in order to plan and organise the resources for the rest of the project activity. If a project manager knows at the beginning that there is not enough time to complete it by a given due date, or if there is not enough money or resources in order to execute it, or if the time is too short in order to arrive at a proper quality for the product, the project may not be started at all. This can save a lot of money and time. To reiterate, this is the eternal paradox of software development—lack of accurate estimate in the beginning versus the essential dependency of project planning over it.

11.4.4 External factors

In the real world, there are additional factors. Requirements keep on changing continuously. Since the software industry is very dynamic, the current requirements are valid only for a short time, and if the product is not available in that time window, these requirements have to be altered in order to cater to the changed market. Requirements and other product factors change during the duration of the project, and they may change a lot. The worse thing is that one can seldom predict how they will change, yet we need to know all these before we start! (Where is Nostradamus?☺)

11.4.5 Conclusion

This is the reason why we *estimate* software projects. There is no mathematical way to calculate the exact effort in advance and it would be naïve to expect the initial values to be correct and accurate. However, this does not render the process of estimation useless. On the contrary, this is an indication that we need to develop better estimation techniques, which give accurate results early enough in the software project life cycle and are useful in proper planning of projects.

Definition

Software estimation is the technique of predicting the duration and cost of a project.

It is a complex process with a lot of factors and dependencies. Software estimation is neither accurate nor constant. Being not constant has implications over two forms:

If I ask for an estimate from two experts about the size of a deliverable, they will most probably give different answers, and they will have reasons to prove their estimates. Second, if I ask the same experts later in the project life cycle for estimation, the new estimates may be different from the previous ones.

Even with these constraints, estimation activity is not entirely useless. Estimates usually lead to a lot of insight into the resources needed later on—the pre-condition being that the estimation is done properly, by taking a lot of factors into account. An estimate done by intuition may not work out as an advantage and should be avoided as far as possible.

Note

The estimation process does not finish until the project finishes.

This is the typical statement of the project manager because of the ever changing conditions of the project (requirements, attrition, money, risks etc.). It is very normal for the initial project estimates to be way off the final figures and become better as the project progresses. Industry data shows that in the early stages of a project, feasibility study, the size of a project, etc. may be underestimated or overestimated by as much as 4 times its final size.

During the duration of the project, there is a gradual convergence of the estimation figures and the width of uncertainty reduces a lot. Also see Fig. 11.1 for a graphic representation of the above.

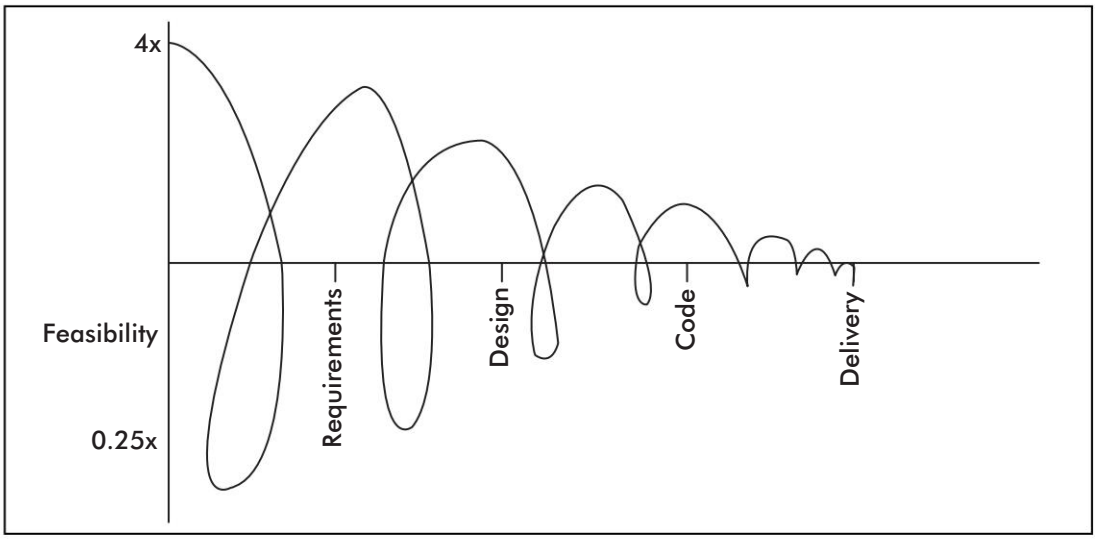


Fig. 11.1 *Uncertainty in estimates through a project's lifetime*

11.5 FACTORS AFFECTING ESTIMATION

When a project is being estimated, usually the required information is not completely available and a lot of ambiguities exist.

11.5.1 What is the definition of complexity?

First, the system has to be classified based on complexity. Or, may be parts of the system have to be classified. The problem at hand is how to measure and quantify the complexity of software. Other than the opinion of an expert, there are no concrete ways of measurement. We can take into account the kind of cohesion in the parts of the system. This will give us an idea about how well the different parts of the system interact with each other. Inside a software component, we can have other methods of classifying the complexity of a system. For example, we can measure the number of unique paths in the software. This will give us an idea of how difficult this software is to test and maintain, thus giving a measure of complexity. We can draw the flowchart of the system and evaluate the number of different conditions (or diamond boxes). This gives us a feeling of how difficult it will be to code the software. This kind of complexity of the software is denoted as "cyclomatic" complexity. We can look at the number of paths inside the flowchart for reaching a particular node from the top node. This gives an idea of the complexity of algorithm. We can also have a measure of nesting levels based on loops or function calls. This nesting level may indicate stack boundaries and difficulty

in coding. All these factors give us a feeling about the complexity of the code being written. However, no method gives us a complete view of complexity of the software. All these methods need to be combined in order to arrive at a better conclusion. Also, at the beginning of the project, it is difficult to make flowcharts of the system. An avid architect may well have a tentative block diagram and high level interactions between these blocks. So, one can look at cohesion as a criterion which identifies complexity of software based on the number and complexity of interactions between its component blocks.

11.5.2 Feature creep

The task of estimation definitely gets affected by a number of external factors, such as *feature creep*, which means addition of new requirements while the project is being executed. In fact, this factor has so much impact on the execution of a software project and is widely prevalent in the industry that new software development life cycles have been proposed and practiced in order to counter the ill effects of this feature. For example, the classic waterfall model of software development, which was very appropriate for the software development activity of late 80s, has not proved effective against this problem. Hence new models such as incremental model and clean-room engineering have been developed which try to mitigate the effects of change of requirements.

11.6 THE BASIC STEPS OF ESTIMATION

Estimation is performed in a number of steps. At any point of time, the estimator has to perform these steps in order to arrive at the final estimate for the deliverable. This section will provide an introduction to these steps. The next sections will describe how to perform these steps.

11.6.1 Step 1: Estimation of size

The first step is the estimation of the size of the software in measurable units. The two most popular measures of size of software are SLOC (Source Lines of Code) and FP (Function Point). The basis of arriving at the size of the deliverable in the beginning is derived from a formal specification such as the customer's requirement document, functional requirement document, system specification, etc. Even when such a formal document is not available, as we saw earlier, an initial estimate is always needed, and is useful for planning.

Estimates are governed by two factors: *accuracy* and *precision.*

Accuracy means how close the estimate is to the actual figure. For example, if the final effort is 100 staff hours, an estimate of 90 hours is more accurate than 80 hours.

Precision defines the level of uncertainty for that estimate. For example, 90 plus or minus 40 is less precise than 80 plus or minus 20.

As we saw in Fig. 11.1, the uncertainty cone is very wide at the beginning of the project. Hence the accuracy and precision are not very high at this time, and so cannot be relied upon heavily. The estimation of size should invariably be refined (made precise and more accurate) during the entire duration of the project, periodically, or in different stages (requirements to design to implementation, etc.), or at the update of requirements, etc. It is often easier for an estimator to propose an estimate in SLOC. This is because the final product will be measured in this unit and also, because of experience of working with C by embedded engineers. The unit of measurement may be convenient; however, it is very difficult to predict a size in terms of lines of code when not a single line has been written. First, it heavily depends on coding styles of individuals. Second, it depends on the language being used for implementation. FORTRAN is known to use less SLOC for mathematical operations as compared to C++, but fails utterly when it comes to performing file operations!

Hence, if the organisation has very strict programming guidelines as proposed in previous chapters and they are followed religiously this basic inconsistency may be nullified. Otherwise, these factors may bring an inherent inaccuracy in the estimation process.

11.6.2 Step 2: Conversion into effort

Once the size of the product has been estimated, it needs to be converted into an effort in man days. This conversion from total software size to total project effort is again intuitive. At best, we can get an empirical formula based on past projects data from the software industry. This conversion depends heavily on the maturity of software development process (involving requirements, design, implementation, testing and integration) of the organisation. A software development project involves far more than simply coding the software. In fact, coding is often the smallest part of overall effort. Writing and reviewing documentation, implementing prototypes, designing deliverables, reviewing and testing code take up the larger portion of overall project effort. The project effort estimate requires the estimator to identify and estimate all the activities that must be performed to build a product of the estimated size. The total effort should remain the same irrespective of whether the size has been specified in SLOC or FP.

11.6.3 Step 3: Estimate the schedule

The third step in estimating a software development project is to determine the project schedule from the effort estimate. This generally involves estimating the number of people who will work on the project, their assignments, availability, how much holidays, travel, and training, besides other risks. Once we have this information, we need to lay it out into a calendar schedule. Now, there are no specific tools or methods available for doing this since a schedule is highly specific to a project. At best some tools exist to help the estimator organise these ideas. Past data available with the organisation is the most useful guide to work out the number of people appropriate for a given size and complexity of a project.

11.6.4 Step 4: Estimate the cost

The final step is to estimate the cost of the project. A project usually has a lot of expenditure other than the salaries of people ☺. Cost includes expenses on infrastructure, tools, training to employees, travel, telephone bills, equipment for testing, office space, etc. This is also highly specific to projects and hence no standard formula can be given in order to generate the final cost based on a schedule and effort. Figure 11.2 explains these steps.

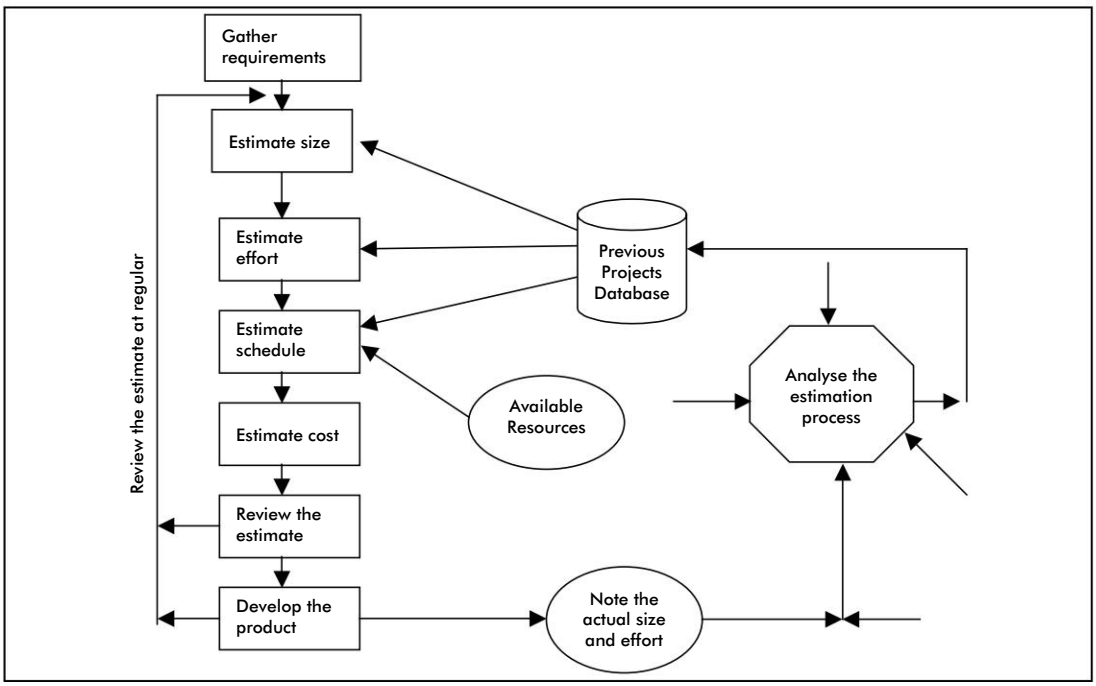


Fig. 11.2 *Steps in a project estimation process*

The estimator should keep the availability of staff in mind and the level of expertise available in executing similar projects in the past. The amount of code reuse possible also has a bearing on total estimated effort. The size of code that has to be written for adapting the code for reuse definitely brings in overheads. In addition, organisational factors like attrition, holiday, etc. and effort for risk management create an impact on the estimated schedule. All these little estimates are themselves inaccurate and they tend to add on to the inherent inaccuracy of an estimation process.

11.7 HOW TO PERFORM ESTIMATION

11.7.1 Estimation of size

There are several methods to estimate size.

Expert opinion

One of the first methods that comes to mind in case of estimation of a particular software is to get the expert's opinion. The expert would have experienced a similar system in the past and would have come across similar code. He or she is best equipped to perform an extrapolation from the past and apply it to the new estimation. However, this needs to be taken with a pinch of salt. First, usually such an estimate does not have a quantitative analysis. Hence it is difficult to review it in a quantitative way. Second, the estimate so done may differ from expert to expert in the perception of complexity of the problem, the foreseen problems and risks are based on different possible implementations. Hence it is very difficult to come to a conclusion based on tentative subjective analysis. Third, no two experiences are the same. If an expert has an experience in developing a project in the past, it is near to impossible that the new software will be the same. In fact, very few parts of the code will be similar to the old one. Hence, the estimate will be mostly based on the factors affecting the current project. When a lot of money and resources are involved, we would certainly like to use something more than just intuition! Please note that this estimation can be done in any unit be it SLOC, FP or something else: it does not really affect the estimation process and the associated difficulty of accuracy and precision.

Strengths of this method:

- The domain expert is the best person to estimate.

Weaknesses of this method:

- Generally it is qualitative.
- Unavailability of previous data to bank on since no two projects are the same.

- Usually the factors affecting estimation in previous projects get diluted over time, or the same factors may not be valid in the new project and the expert is unable to apply the same in subsequent projects.
- Depends heavily on expert's judgement and objectivity and is qualitative to a very large extent.

Historical data

Another way of doing it is to use historical data of similar projects and apply it to the current activity. This will give close results if historical data is available in the organisation in the first place. Secondly, this data should correspond to the kind of system being developed. An embedded system is doomed for disaster if data from a GUI system is taken for estimation. Thirdly, factors under which the previous system was developed should be taken into consideration.

Strengths of this method:

- Useful when a systematic database exists that captures estimates and actual effort for previous similar projects.
- The actual experience on similar projects can be averaged for better estimation.
- Quantitative, repeatable and self-generating. The accurate numbers can then be used to perform better estimations in the future.

Weaknesses of this method:

- Qualitative arguments are needed to understand the differences between the factors of the previous project and the current one.
- Historical data can never be complete in identifying all constraints, techniques, personnel, and activities of the new project.
- The historical data needs to be accurate so that it helps in yielding accurate estimations in the current activity.

Delphi technique

In order to arrive at an analysis, certain **parametric** or **algorithmic** models can be used. The most famous among these models is the **Delphi technique**. This technique uses a group of experts who are asked to estimate the software individually and in isolation. The coordinator without a group discussion then averages the

experts' opinion. A variation to this technique called the **wide band Delphi** works as follows:

i. Co-ordinator provides each expert with an estimation sheet.

ii. Each expert fills out the form individually.

iii. Co-ordinator collates all estimates and marks the points where the estimates differ widely.

iv. Co-ordinator calls a group meeting; where the experts discuss these points and understand from one another the basis of arriving at the estimation figures.

v. Based on the discussion, experts review and submit the estimates again.

vi. The co-ordinator and the experts go through iterations in order to arrive at a consensus.

Strengths of this approach:

- The "judgment" factor present in earlier approaches gets averaged out.
- Experts can resolve qualitative estimates based on insufficient recall and bias through the iterative review and negotiation process.

Weaknesses:

- Since experts do the estimates based on their experience, there is no way to judge the accuracy of an estimate.
- Estimate arrived at group consensus may be more accurate, however, it may involve a lot of time. A quick estimate from an expert may be quick but not quantifiable.
- The estimate is not always exactly repeatable.
- Arriving at a consensus of different estimates may be difficult. There is no clarity on whether to find the average or median of these different estimates.

There are two ways to refine this approach. If a person is asked to provide an estimate about the card verification system, he will most probably reply, "Mm, may be 2000 SLOC". This is possible because, usually while giving estimates from the top of the mind, the system is viewed in its entirety and many times the complexity involved in the interfaces among the components are not taken into account. This is called the top-down method of estimation.

The estimation can be arrived at in a different way if the expert tries to break the system into its components to get a feel of the internal behaviour of the system. The expert bases this on the knowledge and experience of working for similar system. This is called the bottom-up method of estimation.

The expert can then provide an estimate based on the two approaches. Conceptually both top-down and bottom-up approaches usually arrive at the same results. It depends on the ease of use for the expert.

Top down approach

In this technique, the overall cost estimate for the project is derived from the global and common properties of the software project. Estimates are made on the whole system taking the external interfaces of the system into account. Additional factors such as the overall system integration and configuration management are also taken into account.

Strengths:

- System level focus.
- Takes care of overall internal and external factors like system integration into account.
- Puts emphasis on the coupling or interaction among the system constituents.

Weaknesses:

- Can be less accurate because of lack of detail at the level of constituent blocks.
- No focus on the complexities of individual components of the system. So their estimation may not prove to be accurate.

Bottom up estimating

In this technique, the cost of each software component is estimated and these costs are then added to arrive at an estimated cost for the overall product. As can be seen easily, this approach provides the estimator with what the top-down approach fails in—the detail about individual components.

Strengths:

- All components are individually estimated, so there is a better estimation basis.
- Interaction between the components can be better estimated once the individual components are understood well.

Weaknesses:

- May overlook overall common effort related to system integration, configuration management associated with software development.

11.7.2 Estimation of effort and schedule

Estimation of effort and schedule based on the estimation of size, too, is highly intuitive and empirical. Examples of methods to perform effort estimations are:

COCOMO in its versions [1] and Function Point Analysis [5].

11.8 DO's AND DON'Ts OF ESTIMATION

Based on the experience of the software industry, there are some lessons to be learned regarding estimation and the associated planning.

Change and addition in requirements: The possibility of new requirements during the development of software and the associated changes need to be kept in mind while doing estimation. COCOMOII [1] manages this fact well while doing estimation.

Estimation inaccuracy: By its very definition, estimation is inaccurate. It needs to be refined at each stage of development.

Management of the project: Even with accurate estimation, a mismanaged project can lend itself into trouble with respect to effort and schedule. Hence, wastage should be eliminated at all costs.

Don't over-promise: Human limits should never be reached while planning. When all estimates are pointing otherwise, it is better not to over-promise. It finally boils down to the basics of doing too many things wrong or doing few things right.

Beware of excessive multitasking: Humans cannot perform a lot of jobs at the same time. While performing the planning for a project, it is vital to look at this tendency on the part of software engineers. When the number of jobs performed simultaneously by a software engineer increases, the quality of deliverable invariably suffers.

Precision is not possible: As we saw early in the chapter, estimation should be done in ranges owing to the inherent lack of precision in estimation. These ranges become narrower and narrower as the project proceeds until at delivery time when they converge on the actual delivered size of the product.

Use several methods in parallel: It is advisable to use a lot of techniques for estimation in parallel thus verifying the final figure against each other.

Proper planning: While making a schedule, it is imperative to consider that people work x days a week and for y weeks in a year. They are not evenly spread across the whole year. People take holidays. There are other distractions such as travel, recreation and organisational indulgences. So the schedule needs to take these factors into account.

Proper review: It is advisable not to be in a hurry while doing estimation. It is always good to sit down and review the estimates made in the first round. People are known to be overly optimistic when they want to make estimates. Hence, some time should definitely be spent in revising the analysis and factors affecting the estimations.

Who should do the estimation? Two sets of people are indispensable for doing estimates. First, veterans of estimation, these people have battled hard and know the general issues affecting the project. The other people are the developers themselves. Developers usually have strong focus in technical details and it helps in highlighting problems in implementation.

P2P

Involvement of developers during estimation brings in a feeling of ownership that garners commitment to the project.

Ask as many questions as possible: When doing estimates, we should feel free to ask questions. These questions will narrow down the assumptions we have made and hence lead us towards a more focused and less hazy analysis.

Store project data: Organisations that keep database of projects they completed in the past can reuse this knowledge when planning for the next projects. This is true even when the projects of the past were different or involved different factors, or were performed under different conditions. Past records give us a hint of achievable targets and a basis can be arrived regarding the new project. It may vary slightly depending on the extent of changes; however, data collected over a period of time can be really accurate.

11.9 LESSONS LEARNT

In this chapter, we learned that estimation is a necessity for proper planning of an embedded project in terms of cost, resources and time. However, by the very definition, estimation is not an accurate and precise activity. Usually, in the beginning of a project, estimates are wide off the mark with respect to the actual figures. Hence, it is necessary to review the estimate during different stages of a project, or when there are changes in requirements.

Estimation of size, effort and schedule are the prerequisites for a good project plan. Estimation of size can be done on the basis of complexity of the code and other factors such as coupling between components, and can be expressed as SLOC or Function points. This size can be converted into effort based on the empirical formulae available for different projects from a study conducted by Boehm.

All said and done, estimation is a team activity, and all members of a team need to be involved during initial estimation, review and planning.

11.10 REVIEW QUESTIONS

- Give your arguments on why estimation is an important activity during embedded project execution.
- What are the steps involved in estimation?
- "Estimation is a continuous process and does not finish until project closure." If so, then estimation should not serve any purpose. Do you agree with this statement? Justify.
- How does bottom-up approach of estimation differ from top-down approach? What are their strengths and weaknesses?
- Why is it difficult to estimate the size of a product at the beginning of the project?

Chapter
12

Validation and Debugging of Embedded Systems

12.1 INTRODUCTION

As is true with all systems, embedded systems need to be validated before they are shipped for delivery. Testing of embedded systems is done at various levels: first the developers perform unit testing of lowest level code, module level testing to test a group of units forming a logical entity, and finally system testing to validate the entire system. Another way of looking at testing is the kind of errors being looked for, that is, the scope of testing. It is interesting to note how embedded system validation differs from validation of applications. Also, the kind of challenges that exist in validating software written for embedded systems are quite interesting. This chapter begins with this discussion followed by a brief introduction of the different kinds of testing performed on embedded systems, validation tools available, testing strategies and tactics, some troubleshooting tips and finally, we will look into some of the most famous embedded system faults in the human history together with the reasons behind them.

12.2 WHY IS SOFTWARE TESTING DIFFICULT?

In general, any software needs to be tested using the following three basic steps:

i. Create an input for the software in some form.

ii. Receive an output and compare it with the expected output.

iii. Repeat it till it can be safely concluded that the major paths of execution and decision have been tested.

We need to keep in mind, that for a reasonably big software, whether it is application or embedded, it is highly difficult, if not impossible to test each and every part of it. That is why the last item in the above list states that the test should be concluded after all major paths have been checked. The definition of what the major path constitutes, need to be defined before the test execution.

Let us then list out what makes testing of software so very difficult.

- ***Sensitivity to errors:*** Software is inherently sensitive to errors. A near correct value is no better than a completely wrong value. The percentage of software executed in the correct way does not matter if the output is not correct. The only difference is the amount of time that needs to be spent in debugging may vary based on how easily the error can be tracked. Even then, mostly, a completely wrong answer is easier to track than an error that does not occur all the time or the one that gives a slightly wrong answer, because both of these conditions will generally get executed in some obscure portion of the code.

- ***Complexity:*** All said and done, software is complex. If we take the instruction set of a language, there are seemingly infinite ways in which they can be combined in order to arrive at a code.

- ***Testing issues:*** Even a simple program cannot be tested completely based on all combination of possible inputs and comparing against all possible outputs. Testing all execution paths and decision statements with all possible values is a really hard job. This, together with the real life constraints of aggressive schedules and limited resources, makes it a real losing battle. So, at the end of the day, what is important is to realize that software shipped for delivery can never be claimed for zero defects. We can only give a feeling of the quality of software based on defects found, and how the number of defects has gone down over a period of time. And most importantly:

P2P

Testing can only unearth the presence of bugs; it cannot lay any claim on the absence of them.

- ***Regression:*** Software is a unique field in that, the concept of regression, or reappearance of a previous bug in a subsequent change of code, is unheard of in any other discipline. This makes the job of the developer harder by another few degrees. In typical embedded projects, software is not written from scratch, but

handed down from a legacy system, which is then enhanced according to new requirements. The introduction of regression in a legacy system is even simpler! This means that the software testing should be such that it not only tries to find new bugs introduced on the system, but also keep an eye on the previous ones resurfacing because of an inappropriate patch.*

- ***Simulation of actual environment:*** The most challenging job for the tester of the software is to think of all possible uses the software shall be put into during its lifetime. This is as difficult as writing a blank cheque to a stranger! You never know what happens next ☺. What happens if the user presses the key of remote control too fast? What happens if the user presses a particular combination of keys on a mobile phone to launch an application while an incoming call is detected? What happens if the sensor is operated in an unusual temperature or while driving at an unusual speed? Many embedded systems are used in safety critical systems or in systems where the simulation of this atmosphere may be expensive or dangerous. The bottom line is that limitation of the validation of software needs to be accepted.

Keeping this basic mechanism of testing in mind, we can arrive at the differences between testing an application software and embedded software.

Warning

Hardware could be wrong…

It is normal for software engineers to assume hardware is perfect and break their heads over solving some bug. We learnt this the hard way. We were working on the first set of chips that had arrived recently from the fab. After a few days, when we were testing some functionality, we realised that the system was not performing as expected. So, the only thing we began doubting was the software. After two days of frantic debugging, we were still nowhere. Then we were helped by a senior architect who sat with us to identify the problem. Then we realised that the problem occurred when the software/hardware worked in a way to provide a time-critical response. We always saw a wrong response and hence were trying to debug how the software produced the wrong result. But the problem was that even if the software provided the correct response, it got corrupted in the hardware transmission engine. We changed the chip in the board and voila! The problem vanished. The project would have been in serious trouble if this problem was not identified. So, nowadays it makes perfect sense to suspect bugs in the hardware and not just software.

*Patch is a change made in an existing system to solve a bug.

12.3 DIFFERENCES BETWEEN APPLICATION AND EMBEDDED TESTING

- ***Embeddedness:*** Most embedded software are, well embedded. So they do not generally have a keyboard, hard disk and monitor attached to them. In that case, special mechanism needs to be created to validate the software and the complete system before actual usage.

- ***Realtime behaviour:*** Most embedded systems are realtime. It is important that the output received from the system needs to be correct and also that it arrives within a specific time. A late response is a wrong response.

- ***Difficult to simulate:*** Due to the conditions under which embedded systems are used, the simulation of its actual environment may be expensive, difficult or dangerous. A simple embedded device such as an electric or smoke detector creates problems since the validation system needs to circumvent the actual sensor and send a signal directly to the software executed because of this signal.

- ***Difficulty in seeing outputs:*** Since embedded systems are usually connected to devices, their generated outputs are not in the form of a message on the screen, but may be a command to handle a device or write something in memory. So the test suite needs to explore the internal parts of the embedded system in order to verify if the desired operation was performed.

Most consumers do not seem to mind an occasional glitch in the execution of application software. However, there may be a great price to pay, if even the simplest of embedded systems such as a bar code reader crashes during operation or worse, gives a wrong output, or if a washing machine gets programmed in a different mode than desired.

- ***No downtime:*** Unlike application software, many embedded systems are expected to run continuously. This poses its own problems, and they are very difficult to detect during validation. Problems such as memory leaks, recovery from incorrect states, hardware malfunction, etc. usually are difficult to simulate.

There are numerous similarities. However, the extent of differences between the two makes testing deserve a second look by the embedded software developer. And as is evident, the realm of testing embedded systems spans across many frontiers.

12.4 VALIDATION TYPES AND METHODS

Armed with this background of Section 12.3, it would now be worthwhile to know how best all these problems can be minimised. On a broad level, testing of embedded systems can be divided into two categories based on their platform. Embedded are one of the very few kinds of systems that are developed on one kind of platform (generally Unix, however, Windows too), but need to be cross-compiled. The first platform is called the host and the second platform is called the target. Host is used for development because of limitations of editing and compiling code on the target system. So, testing needs to be performed on both the host and target level.

12.4.1 Target testing

This testing is performed on the actual embedded systems as a whole under either actual conditions or a very similar replica of these conditions under realtime. This means, for example, that a TV remote control is tested by actually connecting it to a radio and performing sanity check through the remote control.

12.4.2 Host testing

As the name implies, the software of embedded system is detached from its surroundings and it can be tested in a limited way on a host machine by simulation of all neighbouring environment.

12.4.3 Target testing is good but difficult

Obviously, target testing is the more comprehensive testing method. Since it is performed under near to real conditions, it can detect errors quickly and in possibly larger numbers. On the other hand it is relatively more difficult to actually perform target testing throughout the development period due to the following restrictions.

Incomplete software

Testing on target hardware means that the complete software should be available for validation. This means that the target testing can be performed only after the entire development has finished. This is a major limitation since target testing cannot check the quality and validity of individual components of embedded software. So, it is more expensive since for each small bug in a piece of software (that may not actually relate to being called “realtime”), the complete software needs to be executed, and the long race of chasing of the mysterious bug begins. It is much more cost effective and time

saving to perform sanity checks of individual components of the software before approaching a target station.

Incomplete hardware

Not only does the software need to be complete, but also the hardware should be up and running before the actual target testing is done. Now, it is a project planning issue, but in most cases, both the hardware and software cannot be delivered on exactly the same day. This means that at least one of the teams has to wait. While hardware is getting ready, it makes more sense to perform testing on the software in a limited way on the host.

Regression

Due to realtime and actual usage of the complete environment, it is much harder to simulate a bug again in target platform later in order to check for regression. Once a bug has been detected, it can be simulated easily on a host and can be effectively used later as part of standard test suite to contain regression.

Incomplete testing

Target testing cannot test all portions of the code. The reason is simple. A good part of embedded systems code relates to catching exceptions triggered by rare failures. The rest of it deals with realtime characteristics of input signals and depends on what "state" of processing the system was at that time. So, it is not possible to claim that all code has been tested. It is much easier to simulate such situations on the host and test code.

Lack of good tracking methods

Due to the inherent nature of realtime embedded systems, many a time, a bug detected once may not appear again. So, at each cycle of test, it is essential to capture all status values in a trace dump. However, embedded systems usually are devoid of any storage mechanism. This makes the target platform useless for tracking purpose unless some other mechanism is found to circumvent the problem. For this need, special hardware and software is needed.

12.4.4 Limitations of host testing

However, there are some problems that can be found only in target testing.

No realtime

All problems that relate to the realtime behaviour of the system are very difficult to test on the host. Even though simulators are used, they can help only to a limited extent

since they perform some kind of overlay over the existing software, thus changing the real time behaviour of the code.

Access of peripherals and memory

Shared data problems cannot be detected on the host because in realtime systems, they may occur because of an interrupt. Problems related to accessing of peripherals, actual field conditions like an increase in the temperature of the chip, spurious interrupts can be detected on the target much easily.

Watchdogs can be tested easily and conclusively on the target. Watchdog is a module that regularly receives the status and health-check signals from all parts of the system. It waits for a certain predefined interval. If such signal(s) is(are) not received, it just resets the processor.

12.4.5 Host-based testing setup

The concept of host-based testing involves a simulation of the complete environment of the system under test (SUT). Since the hardware dependent portions of the code cannot be tested effectively on host, they need to be simulated as well. Figures 12.1 and 12.2 illustrates the above fact.

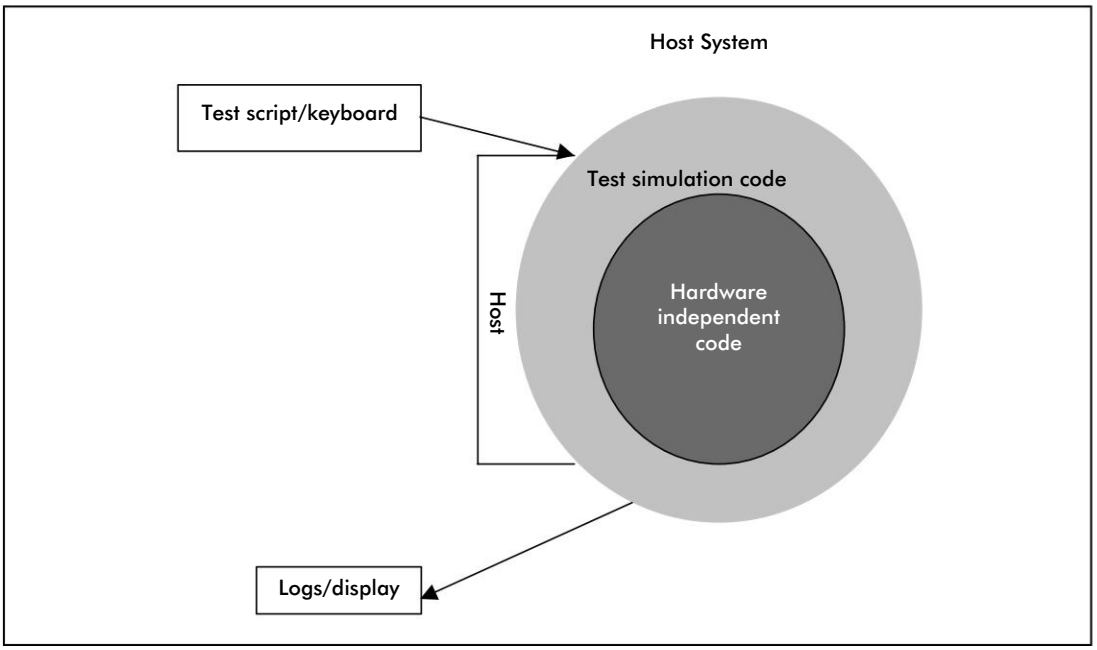


Fig. 12.1 *Host test setup*

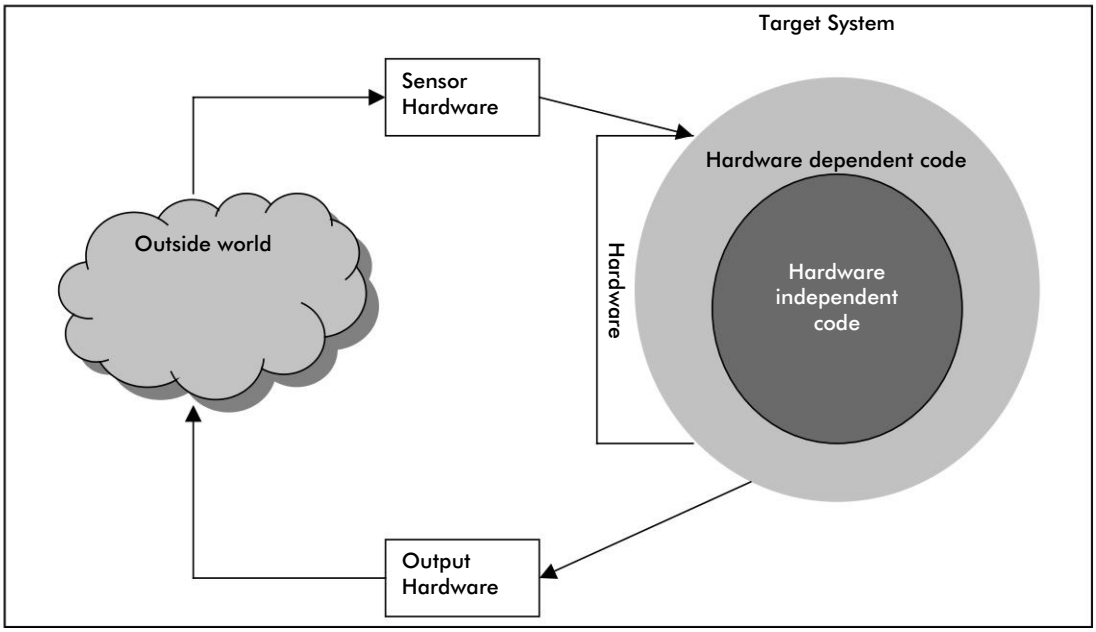


Fig. 12.2 *The organisation of target system*

There is a part of code inside the system that is completely hardware independent, and a piece of code that depends on the hardware. The idea is to replace this hardware dependent code with a test setup that simulates this environment. The simulation test code needs to be written for ISRs, timing interrupts, direct access to memory and devices, other modules, etc. Usually, such test code is called a "stub".

12.4.6 Manual versus automated testing

A very interesting debate that goes on in embedded systems testing fraternity is whether manual testing or automatic testing is more effective. Well, we cannot say which one is more powerful and hence eliminate the other one. It is just that, a combination of both can yield strong gains.

Test automation can improve embedded development by obtaining better test coverage through repeatable tests, performed consistently, by automating the test and verification process. Automated testing can provide faster, more complete product verification faster, once effort has been made to create the automated testing environment. In many instances, an automated environment can perform testing that manual testing could not accomplish or would take too long to perform. Another significant benefit of performing test automation is the accumulation of tests over time. These tests can be

used to check regression and to keep an eye over the overall quality of the system. As systems get developed, they become more and more complicated. With each cycle of development, the effort required for manual testing may grow exponentially. Lastly, in cases where embedded systems need to take certificate of quality from some standard organisations (for example communication devices), the logs generated by automated test tools are a must.

Advantages of automated testing include reduced test time and reduced staff effort in executing the tests. The time thus saved, can then be used for development of more tests. This in turn, results in increased test coverage quality and earlier defect detection and correction because of more extensive testing by running a greater number of test cases. In a nutshell, it augments the overall quality of the product.

There are important and significant differences between manual and automated testing. Manual testing of embedded systems is most useful in situations where the results of a specific and limited set of test cases are needed relatively quickly. Automated testing, on the other hand, requires a greater initial effort to plan, organise, and produce the tests. One of the reasons manual testing is performed, is because of the relatively quick feedback it produces. Automation requires an initial investment of time. However, it produces repeatable tests that can be run possibly in a batch and the results are logged and compared automatically.

Let us take a typical example. Let us assume that we are developing a target system that takes n different types of input values: 1 to n and generates an encryption key based on these values. The ideal way to test such a function would be to create inputs that span across the entire gamut. So, if input can range from say, 1 to 100.

This means: values of 0, 1, 2, 100/2, 100-1, 100 and 100+1.

For the case of n equal to 4, this means 7*7*7*7 = 2401 combinations.

It would be hard to opt for a manual testing of all these combinations.

Even if we do, in the best of conditions, we estimate that each manual input takes one minute. In that case, this manual testing needs more than 40 hours to finish. Not to mention the human error associated with such a scheme. Even though the probability of finding a lot of bugs in this function is less, we still need to spend time in testing it in order to make sure. If the project goes through three build cycles for a release, we have spent more than 15 man days on a single test!

Let us now automate this test. The execution time of each test drops from 40 hours to a far lower value. After some initial setup, the tester is free to spend the rest of the

time performing manual testing over and above the well-defined tests in automated testing. Chances are that the manual testing also unearths bugs. So, the automated testing didn't replace the manual testing, the advantages of manual and automated testing created a synergy and made the testing process much more effective.

Let us look at it in a different way. Let us assume that the project has planned for 20 days of testing effort. If 80% of the test plan can be automated, only 4 days are spent on the automated testing, and at the end of it, the result is the same as that had been achieved by performing manual testing for 20 days. If manual testing is continued in the rest of 16 days, any other bug found could not have been found if the automated testing had not been available.

To conclude:

> We usually cannot find new bugs by performing automated testing but we can reduce testing effort tremendously, while still ensuring quality of the product by performing a judicious mix of manual and automated testing.

12.4.7 Regression testing

Regression testing is used to test previously observed bugs in the code. This testing is best performed in an automated manner whenever a new baseline for the software is created or after a bug has been fixed. The purpose of regression testing is twofold:

- ***Sanity check:*** Regression testing performs effective sanity checks for the system. After going through a regression testing phase, the minimum quality of the software is guaranteed such that basic minimum paths and operations have been tested.
- ***Old ghosts:*** It is common for software developers to introduce new bugs while fixing a problem, or to fix the problem partially in the first place. Regression testing makes it possible to detect any old ghosts returning. ☺

As is evident, regression test is more to control re-occurrence of past defects. Hence, they can be created only after faults have been detected in the system. Alternatively, all tests created for a system in the beginning can renegade to the status of being regression tests later in the next versions.

Usually, regressions tests are written on the basis of reports from the field, or from a past error. So, the regression tester can run his imagination wild and try to execute similar conditions and improve overall quality of software. Unless, a very remote and complicated path of software has been found to be faulty, software bugs usually exist in

groups. If a portion of software has been found to be faulty, chances are that more problems can be unearthed by changing the parameters slightly and checking other border conditions.

12.4.8 White box testing

White box testing is performed at a system or module level by a team in order to exercise the most important paths of the source code. This necessitates that the tester knows about the organisation and functionality of the code. Usually, white box testing is performed after developers have done unit testing. This testing can be performed on the host or target. White box testing tests the coverage of the code. It tests the software from the code point of view. This means that no efforts are made to verify if the code matches the requirements, or if the requirements are complete and correct in the first place. The focus of white box testing is to see if the way code has been written matches its expectations. White box testing is unlikely to detect missing code faults, and particularly in embedded systems, some parts of the code may be unreachable through this testing.

As is evident, white box testing lays its entire focus on the source code being tested. This means that it is heavily dependent on the way the code has been written. In case of slight change in code, because of change in requirements, design or fixing of a major bug, all such tests that depend on the changed code get affected as well.

Secondly, this testing is more difficult to perform in isolation from the development team since the current code needs to be understood and a *heuristic* needs to be found about what are the most important places in the code that need evaluation. In this task, testers need to interact heavily with the developers so that useful insight can be given.

White box testing is also inadequate in testing for systems issues such as systems timing requirements, hardware interfaces, and load or stress testing.

12.4.9 Functional testing

Functional testing or black box testing looks at the system from a requirements point of view. It checks to see if the system is able to satisfy all the requirements expected out of it. There is no focus on code coverage or past regression—just the current requirements and whether the system satisfies these requirements.

As is evident, functional testing is completely independent of the way the system has been designed and implemented. This is the greatest strength of functional testing. Functional testing and development teams perform their jobs independent

of each other. The test suites can be developed as soon as the requirements for a system are ready and much of the work can be done parallel to development of source code. And because functional testing looks at the system from a requirement point of view, it is best suited for testing of nonfunctional requirements as well. These features allow it to find problems that are not detectable by regression testing or code-based testing.

However, functional testing is heavily dependent on the quality of requirements. If the specification for the software is not good, or the specification changes rapidly, the functional testing needs to keep pace with it, thus limiting its effectiveness.

Because the primary focus of functional testing is requirements, it can never be expected to perform analysis of code coverage. Once a requirement needs to be coded, it gets blown up by the multitude of subtle conditions that must be considered. And it is much more cost-effective to perform code coverage analysis through white box testing. This is why functional testing is not a good substitute for code-based testing.

12.4.10 Conclusion of testing methods

All testing methods discussed above compliment each other in some way. Hence, an embedded project can get the maximum benefit by judiciously performing all methods in a suitable way. Since the focus of each kind of testing is different, they unearth different level of bugs. Usually, the pattern followed in test execution is, unit testing of individual units, followed by white box testing of modules, later performing functional testing of the complete system. And, topped by regression testing after each update in the system code.

12.5 TARGET TESTING

Once host testing is finished target testing can commence. As we have noticed before, it is a bit more difficult to test on the target platform because of the limited visibility of what goes on inside the target, in realtime.

12.5.1 PROM programmer

Usually, it is not advisable to directly burn the software image into a ROM and then start testing it on the target. Any small modifications in the software shall render the ROM useless and a new one needs to be burned. It is expensive and more time

consuming. As we saw in Chapter 3, PROMs are a useful alternative for ROMs when the software is not stable. It is simple to create the image inside the PROM using a PROM programmer. This software loads the image inside the PROM. If the software needs to be changed, there are two options. If PROM is E-PROM, then the PROM can be put inside an eraser, and then reprogrammed. If it is not erasable, then the PROM needs to be thrown and a new PROM programmed and inserted inside the target. The main hurdle to be crossed here is to create a compatible version of the image created by the locator that can be understood by the (E) PROM. This usually needs to be done in an adhoc way.

12.5.2 ROM emulator

A slightly better way to handle the frequent programming of PROM is to use a ROM emulator instead of a PROM. (See Fig. 12.3: Setup for ROM emulator). ROM emulator is an electronic circuit having two external interfaces to connect to the host and the target systems. The cross-compiled image is loaded on the ROM-emulator, and the emulator connects to the target system through a bus and gets plugged inside the memory socket of the target system. In this way, the emulator replaces the ROM completely. Now, any changes in the code can be managed much easily since the new code needs to be just compiled and loaded to the emulator.

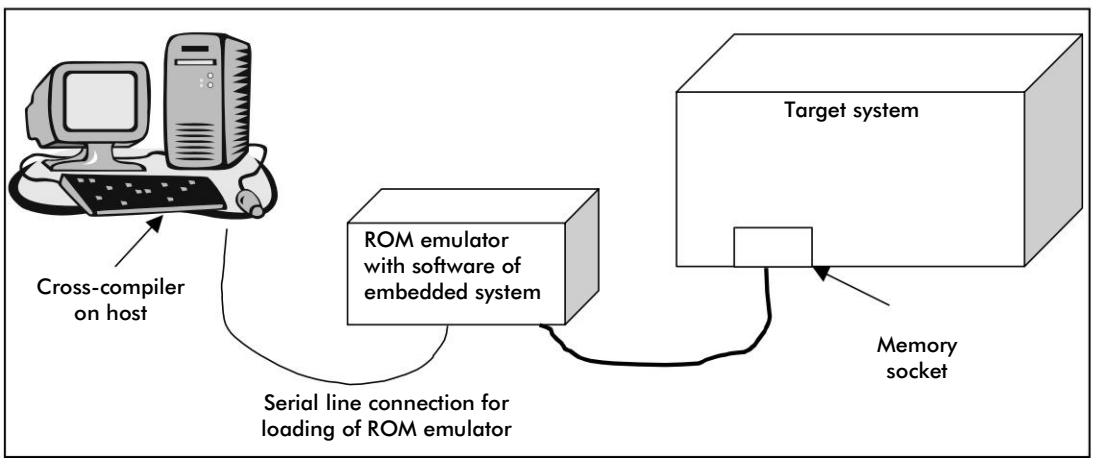


Fig. 12.3 *Setup for ROM emulator*

12.5.3 Source-level debugger

Source-level debugger is one of the most fundamental tools available on the target system for debugging. The concept of a source level debugger (or debug monitor as it is called) is to connect a host machine with the actual target machine. The part of debugger running on the host is similar to a standard debugger and provides user-interaction and display facilities. The part of software on the target performs the job of loading the image to RAM and executing it. (See Fig. 12.4).

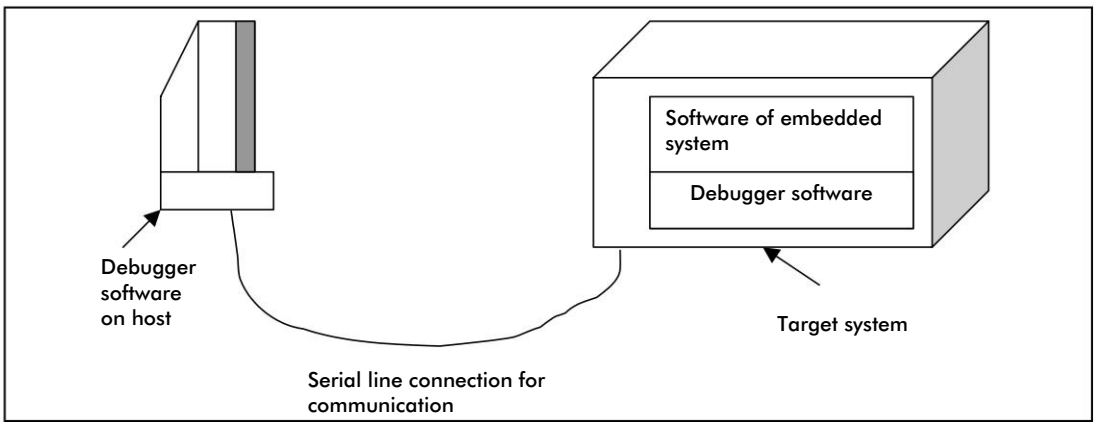


Fig. 12.4 *Setup of source-level debugger*

As is evident, the source program needs to be cross-compiled together with the small debugger software and then loaded on the target.

Advantages of this approach are:

- Source level debuggers are cheaper as compared to other advanced tools like emulators discussed later in this chapter.
- The testing and debugging can be performed in a more realtime way as compared to host based testing. This is because the software is running on the actual target. We have been able to use the same debugger we used previously on host, on the target as well.
- Many debuggers provide the facility to create breakpoints, single step, stop, watch registers and memory locations.

Among the disadvantages:

- This is only a software simulation. The only benefit is that the user can set software break points at instruction fetch time in the target. So, debugging can be trifle more realtime as compared to host debugging. Usually, the debugger on the target does not have access to anything internal to the processor. So more rigorous break points and tracing is not usually possible.
- The code of the debugger inserted in the target is not the code to be shipped with delivery. So, after getting satisfied with the quality of the product, this code needs to be stripped off and the software then needs to be tested on the target without a debugger.
- The debug monitor and the serial line driver code needs to be ported to the target system before they can be used for debugging.

12.5.4 Logic Analysers

They are used to check the logical level of input pins in realtime. Usually it is possible to connect a number of inputs pins for smart analysis by programming the logical analyser: Start tracing pins C and D when the inputs on pins A and B are 1. Usually, logic analysers are used to debug hardware circuits in conjunction with other methods described in this section. Logic analysers in a sense are a smarter version of oscilloscopes with a flexible event system. They also have displays showing different data values observed on different pins as programmed by the user. The logic analysers are actually specialised oscilloscopes for embedded systems, but unlike oscilloscopes, can measure and report voltages as either logical high or low, nothing in between. This limitation, however, suits most embedded systems perfectly since they anyway measure voltages in that way.

12.5.5 JTAG

JTAG is a hardware tool that can control and observe boundary pins of a device for verification of their operation via software control. The reason that a special tool had to be created for this purpose is because of the proliferation of number of pins in a given area on a chip. A JTAG (Joint Test Action Group) consortium exists that caters to the requirements and standardisation of this testing procedure. IEEE 1149.1 standard, known as IEEE Standard Test Access and Boundary Scan Architecture provides complete detail of this procedure.

For a boundary scan to be possible, the device should be compliant to JTAG, which means that processor provides what is known as a JTAG port. A cable connects the host to the JTAG port on the target system and software on the host controls the target microprocessor through it.

12.5.6 Incircuit emulator

One of the most extensive tools used while performing testing of embedded systems is the use of Incircuit emulator (popularly called ICE). ICE is a piece of electronic test equipment that replaces the actual target processor of your system. A debugger runs on the host system and the ICE runs on the target, with a bus using UART connecting the two in order to exchange information. The debugger usually has a graphical user interface and shows the current state of execution, messages in a trace, status of registers, etc. ICE is a very powerful tool in order to test and debug the system at the basic realtime level. The ICE is capable of setting hardware break points or setting of conditions that are normally very difficult to check by using source-level debuggers.

Conditions like:

(if my_var == 1 AND contents of certain memory location equal 0×5555) then stop.

A certain address is accessed but only if a data value of 0×1111 is written

When some data is accessed, and register value is 1.

When code fetch is performed with specific data patterns can be checked easily with an ICE setup.

After the break point is reached, the ICE gives access to the contents of memory and registers for analysis. The code can then be single stepped.

ICE supports tracing in a realtime environment. Since ICE runs the actual system on its processor, it has the capacity to trace all information about processor cycles and timing information together with message flows and their contents. This information can be transmitted by the special bus connected between the host and the target and can be read by a debugger or trace window for simple analysis. Some emulators provide the capability of selective tracing as well.

Many in-circuit emulators nowadays contain overlay memory too. This creates a big advantage because then, the ICE becomes a ROM emulator too. All the powerful featuress of ICE are combined with the simplicity of loading software patches directly on the overlay ROM.

The Trace32-ICE provided by Lauterbach Cop. is a state of the art emulator that supports 8 to 32 bit microprocessors and possesses 16 MB emulation memory.

12.6 THE LAST WORD ABOUT SOURCE CODE

Validation activity is necessary in order to find out problems in the software. However, as we will see some guidelines about programming in the later chapters, here are some tips about programming embedded systems:

- ***KISS:*** Write simple code that can be easily understood and changed by a third person if required. Efficiency is a major factor in embedded systems. But it brings in complicated optimised code. A balance needs to be maintained between the two. In other words, Keep It Short and Simple.

- ***Build tracing mechanisms*** in the code early enough such that a compile switch or a similar method can enable them. Traces can vary from knowing the execution flow of functions down to the values of variables.

- ***Document your interfaces carefully:*** Any interface used by your module should be well documented. Any assumptions in this regard can prove detrimental.

- ***Do not assume anything about realtime behaviour:*** While designing the system, it is advisable not to make any assumptions like:

  This message will always be received in this state since it normally takes 5 msec for response to travel from task *B* to task *A*.

  After sending a request to task *B*, task *A* has sufficient time to process internal conditions. So, this function need not be optimised.
  Such assumptions can prove costly.

- ***Create contingency plan:*** The software should be designed with significant distance from deadlines. If the software has been found to be working perilously near to its deadlines while validation, Murphy's law is bound to haunt you in the field.

12.7 A FEW WELL-KNOWN ERRORS AND THEIR CAUSES

This section provides case studies of some famous faults in embedded systems together with an analysis of the causes behind these failures.

- ***Therac-25:*** Therac-25 was a medical linear accelerator. It overdosed six radiation therapy patients over a two year period leading to deaths of three of them.

The cause that was found related to a total lack of formal software product life cycle, insufficient time allocated for testing, little documentation and an adhoc approach to implementation and testing of software.

- ***Ariane-5:*** Ariane-5 was a $500 million rocket designed to launch satellites in 1996. This rocket flew for a little more than 40 seconds before self-destructing.

The cause was found to be a software error, tracking launch data that was not even relevant to the execution of flight when the error occurred. It triggered a chain of events resulting in finally a self-destruct.

An improper software reuse from Ariane-4, an unnecessary 64-bit floating point datum, a horizontal launch velocity vector, was forcefully converted into a 16-bit signed integer causing an overflow. Incidentally, this data was relevant to the system only when it was on the launch platform and was misleading 30 seconds after the launch. The system decided to self-destruct.

- ***Mars Mission in 1998:*** Mars orbiter was supposed to orbit around Mars as the first interplanetary weather satellite. However, it lost communication with NASA due to either entering orbit too sharply and getting destroyed, or with a small orbit.

Cause: Failure to approach orbit at the right angle because of an inconsistency in the units of measure used by two separate modules developed by separate software groups.

- ***Mars Lander 1999:*** Mars Lander was supposed to land on the surface of the planet and perform experiments for 90 days. Communication was lost after entering into the atmosphere.

Cause: Spurious signals generated when the lander's legs were deployed during descent, giving an indication that it had landed even before it had actually done so, thus crashing it into the surface of Mars.

Warning

Watchdog and very long leash!

As we saw earlier, watchdogs are used to monitor the proper functioning of tasks inside a system. This reminds us of a problem we faced while we were designing the task structure for the software inside an embedded system.

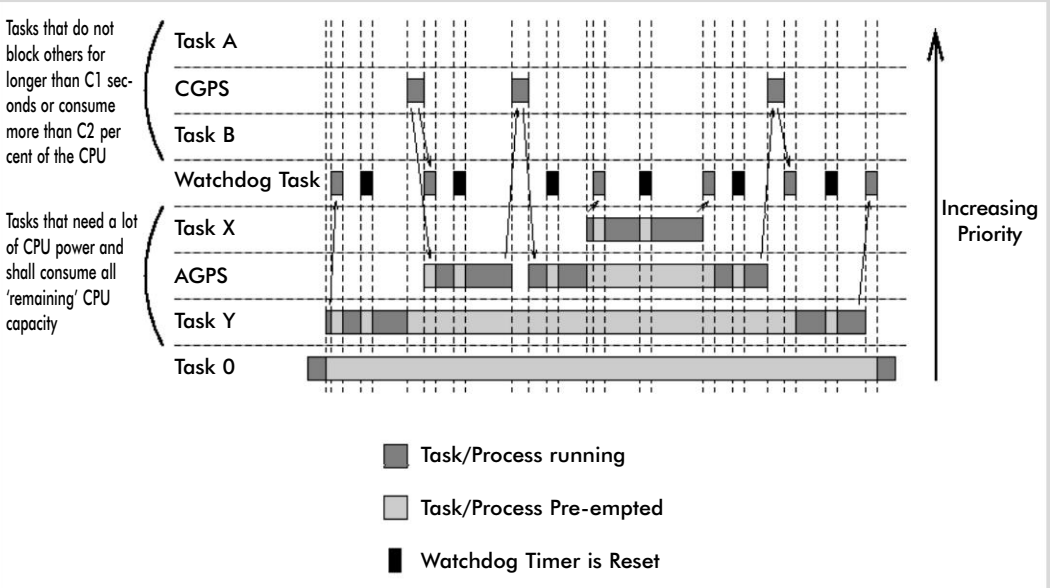
There was a collection of tasks that used to take few milliseconds in the worst case, and they used to be called based on asynchronous events and periodic timer events. There was a watchdog implemented as the last instruction inside the main loop such that it used to get executed if no other task was running.

The watchdog value had been set as one second and everything was working according to plan.

When we were developing a new feature inside our embedded system, we needed to include a task (code purchased from a third party) that used to take several seconds to execute. And this time was not deterministic since it also depended on the quality of signals received on the air interface. Its execution time used to range from 1 second to 60 seconds on our processor. So, the problem arose: how should we include this task such that the watchdog gets a chance to reset its counter before 1 second. The leash of the watchdog became really long ☺.

First the reader should agree that we should put this task (called AGPS) as the lowest priority inside the system otherwise the normal operation of other tasks will no longer remain real-time.

Second problem is that of resetting the watchdog. Various options were discussed. We can reset the watchdog inside the new task as well as inside the mail loop. In this way, if the new task is running, we still take care of resetting it. However, this is not a good design since the control of watchdog is at various places inside the system. Secondly, this solution is not foolproof since if we need to add another similar task in the future, our code will become messy. Second option is to create a watchdog task whose priority is kept the lowest except for the new task. In this way, the normal behaviour of the system shall be kept. For taking care of the low priority task, it needs to register with the watchdog task, giving an estimation of the total duration of its execution. The watchdog task collaborates with the scheduler whenever this low priority task is scheduled out and keeps a count of time. If results are not available by the end of the estimated time of execution of the low priority task, watchdog shall re-initialise this task. There are two problems to this approach: first it is very difficult to accurately estimate the duration of a job. Second, if there is a problem inside the low-priority task, it will not be detected inside the watchdog task until the end of the time estimated by the low priority task—this time may be several seconds! So we hit upon an idea to do it the other way. We implemented it using a heart-beat mechanism. We still created a watchdog task but the lower priority task now was supposed to provide periodic heart-beats to the watchdog task. In this way, the problem of estimation was avoided and any fatal loops inside the lower priority task could be detected within the duration of one heart-beat. Voila!



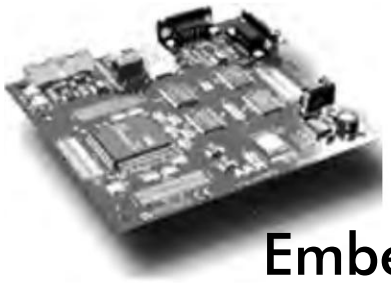
12.8 LESSONS LEARNT

Testing of embedded systems involves unique challenges. Since development is done on the host and execution performed on target, it usually pays to test on host to capture non-realtime bugs. Target system should be used to capture bugs related to peripheral devices, crossover events because the target hardware simulates near-to-field conditions. Testing can be done manually, however, it usually pays to create automated tests and also perform manual testing. Automated tests can then be used for regression testing after each upgrade in the software. White box testing involves active participation from the developer in order to test major paths in the software. Functional testing verifies if all requirements have been satisfied.

A host of tools are available in the market in order to perform efficient testing on the target. PROM programmer burns the code in ROM. ROM emulator is a better way of executing source code and it usually comes complete with a source level debugger too. Logical analyser is used to check the logical level of input in the pins in real time. JTAG can be used for verification of pins through software control. ICE replaces the actual target processor of the embedded system and has powerful features such as setting hardware breakpoints, access to contents of memory and registers.

12.9 REVIEW QUESTIONS

- What is the difference in testing for application and embedded systems?
- What are the benefits of doing white box testing?
- If target testing is powerful, why should we spend time on host testing?
- Describe a typical target testing setup for embedded systems. What tools will you use to perform testing?
- What is the motivation to automate our tests?
- What is the best way to check if any regression has been introduced during a new baseline?



Appendix-A

Embedded/Realtime Systems—Benefits of Using Object Oriented Design and C++

This paper was presented in Philips Technical Symposium, 2001 by Sriram V Iyer and Manoj Kamath

15.1 INTRODUCTION

C++ has always been thought of as a 'big' and 'lavish' language loaded with tones of features, consuming space and slowing down the execution. This led to the perception that, C++ cannot fit in 'small' systems. This idea was seconded by the older compilers producing 2MB executables for a small 'Hello world' program. But the language is now much mature and a large number of very good compilers are available. It is time to give a serious thought about C++ for powering Embedded/Realtime Systems.

This paper is divided into 4 parts as following

i. The Myths regarding C++.
ii. C++, OO, Component Based Development, ORBs, …
iii. A perspective on Embedded C++.
iv. Transition to C++.

15.2 PURPOSE

C++ was mainly thought of as a language that was 'distant' from embedded systems. The paper challenges that notion by showing the effectiveness of the language.

15.3 THE MYTHS REGARDING C++

Myths regarding C++ exist because of the lack of knowledge of "what happens under the hood". The basic constraints of any language are code size and speed of execution.

There are many facts contributing to these constraints. In C++ point of view, the general misconceptions and their reasons are described in this section.

15.3.1 C++ is big

This perception arose mainly because of earlier C++ compilers. Some of the older compilers produced huge (and highly non-optimised) object codes. We should note that, the size of object code entirely depends on the compiler used to produce the code. The language as such does not impose any overheads. Also, because C++ is superset of C, the C code fragment compiled using a C++ compiler will produce exactly same amount of code as a C compiler produces.

As we analyse internals of the C++ compilers, it is clear that, most features of C++ are strictly Compile-Time provisions, with least effect in Code generation. A good example is the use of keywords like **const**, **private**, **protected** and **public**. No code-wise difference exists between **class** members of these types. Function Name Overloading is based on another powerful compile-time mechanism, known as: ***Name Mangling*** or ***Name Decoration***. The function overloading mechanism, as implemented in C++ and as in C are described in code snippets below.

```
// C++ Function Overloading
void myFn()
{      …….
}
void myFn(char* s)
{      ……
}
void main(void)
{      myFn();
       myFn("Hello World")
}
```

```
// Implementation of Function Overloading in C
void myFn_void()
{      …….
}
void myFn_string(char* s)
{      ……
}
void main(void)
{      myFn_void ();
       myFn_string ("Hello World")
}
```

Listing 15.1: ***Function overloading in C and C++***

From this, it is quite clear that, in C++, name Mangling constructs are available, but in C, it is the responsibility of the programmer. Note that, there is no size difference.

Another argument will be the size of **class**. Note that, except the protection and scoping, C++ **class** is almost the same as a C **struct** in size perspective. The member functions are equivalent to functions taking pointer to the **struct** as an argument except that this is done transparent to the user. This is illustrated using equivalent pieces of both C++ and C code. Note that here also code size is the same.

```
// C++ Class Example
class myClass       {
private:
        int i;
public:
        void setVal(int j)
} ;
void myClass:: setVal(int j)
{
  i = j;
}
```

```
// C Equivalent of above Class Example
struct myClass_c
{
  int i;
} ;
void setVal(struct myClass_c *myCls, int j)
{ myCls →i = j;
}
```

Listing 15.2: *Equivalence of class in C and C++*

The **inline functions** of C++ is another powerful concept, which is always blamed because of misuse, accounting to code bloat. **Inline functions** provide understandability and safety of normal functions. Appropriate use of **inline functions** will improve both size and speed.

Now the **inheritance** mechanism of C++, is nothing but "**Struct-inside-Struct**" in C. But in C, as we need to manually implement this, will cause introduction of more and more functions.

15.3.2 C++ was NOT designed for system programming

This is another popular myth. Even though C++ is *lingua franca* for programming complex database systems and mammoth applications, for its ability to scale beautifully,

C++ was originally conceived as a system programming language as a superior alternative to C. That is why C++ inherited most of the features of C, to support the legacy code available. Also through the Object Oriented Concept, it is really easy to communicate with devices and provide synchronisation easily.

15.3.3 C++ is slow

This concept spread because of the '*Code Bloat*' myth. The C++ objects 'arrive' in a predefined state; that is they are already initialised as they come into existence. This takes time, but gives the programmer more control over the code, with less leakage of resources. The C variables are (mostly) uninitialised and it is done by a separate function, which the programmer should remember to call. An example code snippet is given below.

```
// C++ Class Constructor/Destructor Example
class myClass       {
private:
        int *i;
public:
        myClass()
        {     i = new int[MAX_CLS];   }
~myClass()
        {     delete [] i;            }
} ;
void fn(int j)
{ myClass cls;
  ……
// as this scope is exited, the destructor is called
//automatically and memory will be freed
}
```

```
// C Equivalent of above Example
struct myStruct
{ int *i;
} ;

void fn(int j)
{
  struct myStruct mySt;
  mySt.i = (int *)malloc(MAX_CLS *sizeof (int));
  …..

  free(mySt.i);     /* usually programmers fail to do*/
}
```

Listing 15.3: ***Constructors and destructors***

The constructors and destructors in C++ automate the process of acquiring and releasing resources (not just memory). The technique of having a constructor acquire a resource and a destructor release can be phrased as '*resource acquisition is initialisation*'.

15.3.4 C++, OO, component based development, embedded ORBs, …

Another term that is grossly misunderstood in conjunction with C++ is that it '*is*' an 'object oriented' language. The fact is that C++ greatly aids OO based software development. This only means that the language provides constructs to write software based on the OO paradigm. Period.

This does not mean that all software written in C++ is object oriented or that only good OO software can be written in C++. (Horribly bad OO implementations that are available stand to prove this claim)

Objects/classes, abstraction, encapsulation, inheritance are some of the fundamental concepts of object-oriented technology. C++ provides language constructs to implement the above.

OO aims in making software more

- Understandable
- Changeable
- Reusable

C++ when used effectively can be used to achieve the above goals.

Let us really see some code to find how C++ helps in achieving some these goals.

15.3.5 4.1 Code size reduction, improvement in readability

Consider a case for a network protocol, where two kinds of packets say

- Data Packets: Sent from higher layers for transmission
- Control Packets: Used for link establishment, timing synchronisation, etc.

Let them have two kinds of headers, HeaderData, HeaderCtrl. And say, we have a function that processes header of the packet received at some buffer.

In C, we could have to write two functions like this: ProcessHeaderData and ProcessHeaderCtrl.

Let us see how we can do this in C++.

```
class Header {
public:
      virtual int length ( void ) = 0;
      // …
};

class HeaderData : public Header {
public:
      virtual int length ( ) {
        return DATA_HEADER_LENGTH; // return Header length of
                                   // packet 'Data'
      }
      // …

private:
      char* szBuf;
      // …
};

class HeaderCtrl : public Header {
public:
      virtual int length ( ) {
        return CTRL_HEADER_LENGTH; // return Header length of
                                   // packet 'Control'
      }
      // …

private:
      char* szBuf;
      // …
};
```

Listing 15.4: ***Classes arranged hierchically to use polymorphism***

Here we create an abstract base class ('*interface*') specifying general '*behaviour*' of a packet header. HeaderMgmt and HeaderData '*are*' headers. They both '*implement*' common behaviour of a packet header. Our C++ implementation of the ProcessHeader can be like this:

```
void vProcessHeader ( Header* pHeader )
{
       // …
       memcpy ( szTargetBuffer, pHeader->data(),
       pHeader->length() );
       // pHeader->length( ) returns appropriate length for
       // packets
}
```

Listing 15.5: *Using polymorphism*

More than just elegance, simplicity and improvement in readability, this causes considerable reduction in code size by $1/n$ where n is the number of types of packets (in this case).

15.3.6 Maintainability

Our vProcessHeader will work even if a new class of packets with a new type of header is introduced.

```
class HeaderNew : public Header {
public:
      virtual int length ( ) {
          return NEW_HEADER_LENGTH;
      }
//…
};
```

Listing 15.6: *Ease of addition of new class of packets*

15.3.7 Reusability

Reusability was achieved in C using libraries (using the Structured Analysis and Design paradigm). These approaches though quite suitable for smaller projects, they cannot be used to construct large software systems. Any change in their interface broke the build. They were the main cause of increase in fragility of large systems built this way.

Let us consider a code that could be used to write on a display terminal in C. This can be called vDisplayOnScreen (char* szText);

Now if we wanted to add a background colour, we need to rewrite vDisplayOnScreen (char* szText) as vDisplayOnScreen (char* szText, enColor color).

This obviously breaks the build. This causes ripples to be caused throughout the project and obviously the code has to be recompiled. Let us see how we can do this in C++.

```
class DisplayWriter {
public:
       virtual void display ( );
       // …

private:
       string text;
};
```

Listing 15.7: *Reusability in C++ - 1*

A function that uses a DisplayWriter object to write to display can be

```
void  vDisplayOnScreen ( DisplayWriter& dw )
{
     dw.display ( );
}
```

Say, now we want to add a background color, we can write a class such as BGDisplayWriter.

```
class BGDisplayWriter : public DisplayWriter {
public:
       virtual void display ( ) {
            DisplayWriter::display ( ); // reuse existing-
                                        // DisplayWriter code
            fillBackGround ( color );
       }
// …

private:
       enColor color;
};
```

Here in the 'display' function of BGDisplayWriter, reuses the code in DisplayWriter. And, an object of BGDisplayWriter can be passed to vDisplayOnScreen without breaking the build. (This principle of being able to use a derived class instead of base class is called 'Liskov's Substitution Principle')

Thus we also see that C++ really helps in writing reusable code. Some of the most reused code in the PC world is written in C++. (Heard MFC?)

Now, as the software in embedded systems get only larger (and exponentially), we move towards embedded ORBs. (Companies like ObjectTime™ already market embedded ORBs). Thus to derive major objectives like faster time to market, more code reuse etc., shifting to C++ becomes imperative.

15.4 A PERSPECTIVE ON EMBEDDED C++ (EC++)

Embedded C++ specification is a subset of C++ that aims to reduce code size and improve speed of C++ code by excluding certain features of C++ that cause code and time overheads.

Zero overhead rule in design of C++: *What you don't use, you don't pay for it.* This simply means that the language features that you do not use do not cause overheads, runtime or code size. Most of the good compilers go a long way in implementing this. But RTTI, exception handling, etc. inevitably cause some increase in code size. But most compilers give you an option of disabling these features when you don't use them.

What is embedded C++?: Embedded C++ is a scaled down version of ANSI C++, with the following features removed:

- Multiple Inheritance/Virtual base classes
- Run Time Type Identification (RTTI)
- Templates
- Exception Handling
- Namespaces
- New style casts (static_cast, dynamic_cast, const_cast, reinterpret_cast)

Exception and RTTI are some of the major features that cause quite some code bloat without the user being aware of it. In theory, a good implementation of templates by the compiler causes no code bloat. The '*standard template library*' (STL) is one great reason to shift to C++. But, by removing templates, this advantage is nullified. Nowadays, good C++ compilers also give an option of EC++ with templates. (For e.g. Green Hills™ C++ compiler comes with a dialect called ETC++ which is EC++ with template support). Even though, namespaces and new type casts do not cause any increase in code size they were not included in EC++ because those were relatively new features then.

Using EC++ causes quite some reduction in object code produced from C++, especially in large software systems. As compilers mature, the overheads also will reduce considerably. The EC++ standardisation was done by http://www.caravan.net/ec2plus.

15.5 TRANSITION TO C++

The language constructs are so that, their appropriate usage will provide good space and time saving measures, but careless usage may cause problems. So, the following facts should be taken into account during migration to C++.

15.5.1 Virtual functions

Virtual functions provide the base for 'Polymorphism' in C++, with a little, but a justified cost. Virtual functions are implemented by array of pointers called ***vTable*** for each class with virtual functions. Each object of such class will contain a pointer to the class's ***vTable***, which is put by the compiler and is used by the generated code. This makes the objects slightly bigger (typically 4 bytes per virtual function), because every object of a class with virtual functions will contain ***vTable*** Pointer. Another cost is the ***vTable*** lookup for a function call, rather then direct call. But note that, this is typically less than providing another parameter to the function for the purpose to implement similar functionality in C.

15.5.2 Templates

A class template is rather like macro expanding to an entire class. Older compilers had expanded the templates to classes every time it is encountered which was having devastating effect on the code size. But the newer compilers and linkers find duplicates and produce at most, one expansion of a given template for a given parameter class.

15.5.3 Exceptions

Exceptions are abnormal condition handlers, which will really help the programmer to handle such conditions, also facilitating prevention of resource leakage. Support for exception results in small performance penalty for each function call. This is to record information to ensure that, destructor calls are made when an exception is thrown. But usage wise, the exceptions are worth this cost, if used properly.

15.5.4 Runtime type information

For keeping this information, a little space is used for each Class. But for type identification, RTTI exploits the vTable pointer in an object [if it is present] or provides

sensible default values. Compiler options can be used to disable Run Time Type Information storage, to avoid the cost. Though useful, exception and RTTI do not justify their cost in embedded systems in near future.

15.6 CONCLUSION

We see that C++ has come a long way since the first Cfront compiler was written for C++ by Bjarne Stroustrup in AT&T Bell (then) Labs. The compilers do a fantastic job in producing code with very little overheads. It took 20 years for C to enter embedded systems because it was thought to be a language that was huge for embedded systems. Now C is the most widely used language for programming embedded systems. Through this paper we state the facts regarding using C++ in embedded/realtime systems. We see that the benefits clearly overweigh the defects. Let us not delay the usage of C++ in embedded systems for next 20 years. The language is mature enough now and it is the right time to shift to C++ to reap the benefits of OO-based software development.

15.7 REFERENCES

1. C++ *FAQs*, Marshall Cline *et al.*, Addison Wesley, 1999.
2. *Realtime Systems Design and Analysis*—An Engineer's Handbook, Phillip A Laplante, Prentice Hall India.
3. *Developing Object Oriented Software for Real-Time Systems*, Maher Awad *et al.*, Embedded Systems Programming, September 1996.
4. *Strategies for Real-Time System Specification*, Derek Hatley & Imtiaz Pirbhai, Dorset House, 1988.
5. *Software Design Methods for Concurrent and Realtime Systems*, Hassan Gomaa, Addison Wesley, 1993.
6. *Abstract Base Classes*, Sriram V Iyer, ITSpace.com, 2001.

Bibliography

- *Programming Embedded Systems in C and C++*. Micheal Barr, O'Reilly Publications, 1999.
- *Real-Time UML: Development of Efficient Objects for Embedded Systems*, Douglas, Bruce Powel, Addison-Wesly, 1998.
- *The Art of Programming Embedded Systems*, Jack G. Ganssle, Academic Press, 1991.
- *Embedded Real-Time Systems: A Specification and Design*, Jean Paul Calvez, John Wiley, 1993.
- *Reusable Software Components: Object-oriented Embedded Programming in C*, Ted Van Sickle, Prentice Hall, 1997.
- *Real-Time Programming: A Guide to 32-bit Embedded Development*, Rick Greham, Robert Moote and Ingo Cyliax, Addison Wesley, 1998.
- *Arithmatic Built-in Self-test: For Embedded System*, Janusz Rajski, Jerzy Tyszer, Prentice Hall PTR, 1998.
- *Debugging Embedded Microprocessor System*, Stuart R Ball, Newnes-Butterworth & Heinenmann, 1998.
- *Embedded Microprocessor System: Real World Design*, Stuart R Ball, Newnes-Butterworth & Heinenmann, 1996.
- *Realtime Systems*, Nimal Nissanke, Prentice Hall, 1997.
- *An Embedded Software Primer*, David E Simon, Addison Wesley, 1999.

- *Embedded Systems Programming* (ESP), www.embedded.com
- *C/C++ Users Journal*, www.cuj.com
- *Dr Dobb's Journal*, www.ddj.com
- *Various Discussions in* www.slashdot.org
- Boehm, Barry W. *Software Engineering Economics*, Prentice Hall, 1981.
- Data and Analysis Center for Software of US Department of Defense Information Analysis Center: www.dacs.dtic.mil
- Boehm, C Abts, A W Brown, S Chulani, B K Clark, E Horowitz, R Madachy, D Reifer and B Steece, *Software cost estimation with COCOMO II*. Prentice Hall PTR, 2000.
- *The common software measurement Metric Consortium manual:* http://www.lrgl.uqam.ca/publications/private/446.pdf
- *R Boehm, Function point FAQ:* http://ourworld.compuserve.com/homepages/softcomp/

Exercises

The card verification system was introduced in Chapter 1. The requirements of the system have been defined in Chapter 8.

1 Create a requirements document for the system.

Design all components shown in Figure 8.3

Write the pseudocode for these components and simulate this device.

I want to automate my house in the following way: My mobile phone sends a message to my PC at home that controls all household devices. So, when I leave my place of work in the evening, I just send a message to my PC. My PC then switches on my geyser so that I have hot water ready for a bath. My coffee maker starts
2 preparing coffee and the music system plays the kind of music I like when I enter my house. Also, the refrigerator detects if any groceries are missing and sends me a message via the PC.

Create a simulation of this intelligent network that satisfies the above requirements. Make suitable assumptions wherever required while giving appropriate explanations.

Assuming the following code is run on a system that allocates two bytes for an unsigned int and one byte for unsigned char. Calculate the data segment of the following code:

3

```
#define TRUE 1
#define FALSE 0
unsigned int my_int;
unsigned int another_int=0;
unsigned int third_int=1;
int main(void)
{
  unsigned int local_int=3;
  unsigned char local_char;
  int i;

  my_int = local_int;

  for(i=0, i<third_int; i++) {
        if(third_int == FALSE) third_int=TRUE;
  }
  return 0;
}
```

The following code uses a function called GetBits that performs the following function:

Takes argument an offset byte off_byte, offset bit off_bit and number of bits num
Operates on a global array buff of length 24 bytes.
Returns a long int by filling least significant bits of buff with num bits from the Position off_byte & off_bit.

4 The code takes an input buffer stream and starts decoding bits and fills the structure global_struct based on the values of some parameters. What different problems/bugs can you spot in the code? Imagine char occupies 1 byte and unsigned long 4 bytes.

```
typedef struct
{
      char ncc;
      char bcc;
      char power;
```

```
        char chn;
        char c1;
        char beacon;
} global_struct ;

int decode_struct( )
{
        off_byte = 4 ;
        off_bit = 2 ;

        if(GetBits(off_byte, off_bit, 1) == 1)
        {
                off_byte += 21 ;
                off_bit = 0 ;
        }
        else
        {
                off_bit ++;
        }
        if(GetBits(off_byte, off_bit, 3) == 0x2)
        {
          off_bit += 3 ;
          global_struct.ncc = GetBits(off_byte, off_bit, 3);
          global_struct.bcc = GetBits(off_byte, off_bit, 4);
          global_struct.power = GetBits(off_byte, off_bit, 6);
        }
        else if(GetBits(off_byte, off_bit, 3) == 0x1)
        {
          global_struct.chn = GetBits(off_byte, off_bit, 3);
          global_struct.c1 = GetBits(off_byte, off_bit, 4);
          global_struct.beacon = GetBits(off_byte, off_bit, 9);
        }
}
```

4

---

5 In the previous question, is it possible to optimize the usage of RAM? What about ROM? For achieving the same functionality, how is it possible to write more efficient code?

---

6 As we saw inside the chapter, memory pools are an efficient way of removing the non-real-time effects of dynamic memory allocation. The total amount of memory that can be allocated inside a system is 20 KB. It is desired to create memory

pools of sizes 100 bytes, 1K and 2K. Create a best-fit pool allocation and de-allocation mechanism taking care of any external interrupts entering the system.

---

7 Write a driver to take care of a DMA channel of size 1024 bytes. The DMA channel generates an interrupt when the buffer is half full as well as when it is full. The data should be stored in a circular buffer and pointers should be used to keep track of new and old data.

---

8 My system is connected to the following devices:

Battery
Clock
Keyboard
Display unit

Battery gives periodic indications about the remaining charge that needs to be displayed on the display unit, which is memory mapped. Clock gives periodic one-second ticks to update the time on the display unit. Keyboard is pressed by the user and can trigger events based on the following combination of digits:

*123#
+123*
$123+

Any other combination shall lead to the display of an error message in the display unit. Write source code for such a system. Make appropriate assumptions where appropriate.

Hint: This system shall need to implement ISRs for battery, clock and keyboard. It should provide a minimal editor that displays the keys entered by the user together with usage of keys to undo some actions.

---

9 Two devices A and B are connected by a UART that can support 9.6 baud. Device A takes input from a user or a stored file and starts transmission of data. The speed of this transmission may be more than 9.6 baud. Design a protocol between these devices such that data is not lost mid-way.

---

10 Many operating systems now provide "plug-and-play" functionality. For example, it is possible to plug another device to a running system. The same functionality is also found in ad-hoc networks of small devices such as WirelessLANs and Bluetooth™. How do these devices detect each other? Would polling be suitable here?

---

11 Design a queuing mailbox mechanism between two tasks inside a system. It should provide the following features:

a. Two levels of priority of messages based on different function calls
b. Ability to look inside the queue to check for a particular message
c. Save a message instead of consuming it, which means that the message goes to the end of the queue
d. Sending messages to multiple recipients

---

12 In previous question, we also want to implement an automatic garbage collection of messages. The RTOS should create the lowest priority task and de-allocate the memory used by messages. The messages are allocated using the pool mechanism.

INDEX