

The McGraw-Hill Companies

Embedded Systems

Architecture, Programming and Design

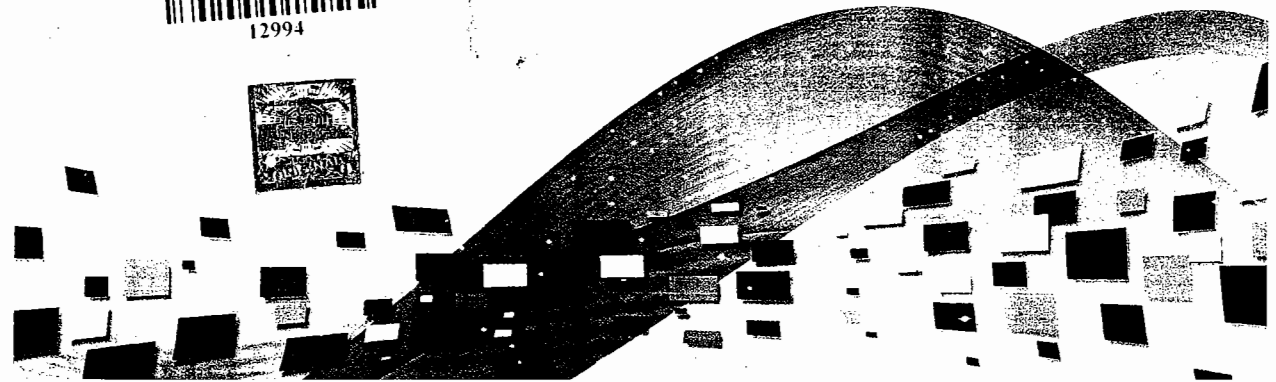
Second Edition

Raj Kamal

004.16 KAM



12994



Acknowledgements

I am immensely grateful to my teachers at the Indian Institute of Technology, Delhi (1966–72), and the University of Uppsala, Sweden (1978–79, 1984), for teaching me the importance of self-learning and the essence of keeping up with merging technology. I would like to thank Prof. MS Sodha, FNA, for his support and blessings throughout my academic life. I acknowledge my Indore colleagues—Dr PC Sharma, Dr PK Chande, Dr Sanjeev Tokekar, Mrs Vrinda Tokekar, Dr AK Jaisankar, Dr Maya Ingle, Dr Sanjay Tanwani, Ms Preeti Saxena, Ms Shraddha Masih, Ms Aparna Dev, and Ms Vasanti Gaurkar and other university academicians, Dr PS Grover (Delhi), Dr Harvinder Singh Saini (Hyderabad), Dr S Radhakrishnan (Chennai), Dr TV Gopal (Anna University) and Dr KM Mehata (Anna University)—for their constant encouragement and appreciation of my efforts. I am thankful to the editorial team at McGraw-Hill Education India for their reviews and suggestions. I acknowledge my late colleague Dr MK Sahu, Head, Computer Centre of the University, who will miss seeing his new edition as he passed away during the last phase of the preparation of this book.

I would also like to thank all those reviewers who took out time to go through the script and give me their feedback. Their names are listed below.

Ramanaryan Reddy

Dept. of Computer Science and Engineering,
Indira Gandhi Institute of Technology, New Delhi

Nilima Fulmare

Industan College of Science and Technology, Agra

Nikhil Kothari

Dept. of Electronics and Communication Engineering,
Shri Ram Singh Desai Institute of Technology, Gujarat

Umesh Kelkar

Ammins Institute of Engineering and Technology,
Tatyasaheb Kore University

Shilpa Adhikari

Dept. of Electronics and Communication Engineering,
Jalpaiguri Institute of Engineering and Management, Kolkata

Pratik Ghosh

Dept. of Electronics and Communication Engineering,
Jadavpur Institute of Technology, Kolkata

Debashish De

Dept. of Electronics and Communication Engineering,
Jagadgur Saha Institute of Technology, Kolkata

P Kabisatpathy

Dept. of Electronics and Instrumentation Engineering,
College of Engineering and Technology, Bhubaneswar

KK Mohapatra

Dept. of Electronics,
National Institute of Technology, Rourkela

JK Mendiratta

Dept. of Electronics and Communication and Engineering,
R.V. College of Engineering, Bangalore

Lyla B Das

Dept. of Electronics Engineering,
National Institute of Technology, Calicut

Santa Kumari

Dept. of Electronics and Communication Engineering,
Andhra University, Vishakhapatnam

V Murlidharan

Dept. of Computer Science and Engineering,
MS Ramaiah Institute of Technology, Bangalore

Josephine P Kumar

Dept. of Computer Science and Engineering,
MVI College of Engineering, Bangalore

Stanley Johnson

Invensys India Pvt. Ltd., Chennai

Finally, I acknowledge my wife, Sushil Mittal, and my family members—Shalin Mittal, Needhi Mittal, Dr Atul Kondaskar, Dr Shilpi Kondaskar, and Ms Arushi Kondaskar—for their immense love, understanding, and support during the writing of this revised edition.

Contents

Preface to the Second Edition

Preface to the First Edition

1. Introduction to Embedded Systems

- 1.1 Embedded Systems 3
- 1.2 Processor Embedded into a System 5
- 1.3 Embedded Hardware Units and Devices in a System 10
- 1.4 Embedded Software in a System 19
- 1.5 Examples of Embedded Systems 27
- 1.6 Embedded System-on-chip (Soc) and Use of VLSI Circuit Design Technology 29
- 1.7 Complex Systems Design and Processors 32
- 1.8 Design Process in Embedded System 37
- 1.9 Formalization of System Design 42
- 1.10 Design Process and Design Examples 43
- 1.11 Classification of Embedded Systems 52
- 1.12 Skills Required for an Embedded System Designer 53

2. 8051 and Advanced Processor Architectures, Memory Organization and Real-world Interfacing

- 2.1 8051 Architecture 62
- 2.2 Real World Interfacing 72
- 2.3 Introduction to Advanced Architectures 84
- 2.4 Processor and Memory Organization 96
- 2.5 Instruction-Level Parallelism 104
- 2.6 Performance Metrics 106
- 2.7 Memory-Types, Memory-Maps and Addresses 106
- 2.8 Processor Selection 113
- 2.9 Memory Selection 118

3. Devices and Communication Buses for Devices Network

- 3.1 IO Types and Examples 130
- 3.2 Serial Communication Devices 134
- 3.3 Parallel Device Ports 143
- 3.4 Sophisticated Interfacing Features in Device Ports 150
- 3.5 Wireless Devices 151
- 3.6 Timer and Counting Devices 152
- 3.7 Watchdog Timer 157
- 3.8 Real Time Clock 158
- 3.9 Networked Embedded Systems 159
- 3.10 Serial Bus Communication Protocols 160
- 3.11 Parallel Bus Device Protocols—Parallel Communication Network Using ISA, PCI, PCI-X and Advanced Buses 166
- 3.12 Internet Enabled Systems—Network Protocols 170
- 3.13 Wireless and Mobile System Protocols 175

4. Device Drivers and Interrupts Service Mechanism

- 4.1 Programmed-I/O Busy-wait Approach without Interrupt Service Mechanism 189

vii

ix

1

61

128

187

4.2	ISR Concept	192	
4.3	Interrupt Sources	200	
4.4	Interrupt Servicing (Handling) Mechanism	203	
4.5	Multiple Interrupts	209	
4.6	Context and the Periods for Context Switching, Interrupt Latency and Deadline	211	
4.7	Classification of Processors Interrupt Service Mechanism from Context-Saving Angle	217	
4.8	Direct Memory Access	218	
4.9	Device Driver Programming	220	
5.	Programming Concepts and Embedded Programming in C, C++ and Java		234
5.1	Software Programming in Assembly Language (ALP) and in High-Level Language 'C'	235	
5.2	C Program Elements: Header and Source Files and Preprocessor Directives	237	
5.3	Program Elements: Macros and Functions	239	
5.4	Program Elements: Data Types, Data Structures, Modifiers, Statements, Loops and Pointers	241	
5.5	Object-Oriented Programming	262	
5.6	Embedded Programming in C++	263	
5.7	Embedded Programming in Java	264	
6.	Program Modeling Concepts		273
6.1	Program Models	274	
6.2	DFG Models	277	
6.3	State Machine Programming Models for Event-controlled Program Flow	282	
6.4	Modeling of Multiprocessor Systems	288	
6.5	UML Modelling	295	
7.	Interprocess Communication and Synchronization of Processes, Threads and Tasks		303
7.1	Multiple Processes in an Application	305	
7.2	Multiple Threads in an Application	306	
7.3	Tasks	308	
7.4	Task States	308	
7.5	Task and Data	310	
7.6	Clear-cut Distinction between Functions, ISRS and Tasks by their Characteristics	311	
7.7	Concept of Semaphores	314	
7.8	Shared Data	326	
7.9	Interprocess Communication	330	
7.10	Signal Function	332	
7.11	Semaphore Functions	334	
7.12	Message Queue Functions	335	
7.13	Mailbox Functions	337	
7.14	Pipe Functions	339	
7.15	Socket Functions	341	
7.16	RPC Functions	345	
8.	Real-Time Operating Systems		350
8.1	OS Services	351	
8.2	Process Management	355	
8.3	Timer Functions	356	
8.4	Event Functions	358	
8.5	Memory Management	359	

8.6	Device, File and IO Subsystems Management	361	
8.7	Interrupt Routines in RTOS Environment and Handling of Interrupt Source Calls	366	
8.8	Real-time Operating Systems	370	
8.9	Basic Design Using an RTOS	372	
8.10	Rtos Task Scheduling Models, Interrupt Latency and Response of the Tasks as Performance Metrics	385	
8.11	OS Security Issues	401	
9.	Real-time Operating System Programming-I: Micro/OS-II and VxWorks		406
9.1	Basic Functions and Types of RTOSES	408	
9.2	RTOS mCOS-II	410	
9.3	RTOS VxWorks	453	
10.	Real-time Operating System Programming-ii: Windows CE, OSEK and Real-time Linux Functions		477
10.1	Windows CE	478	
10.2	OSEK	494	
10.3	Linux 2.6.x and RTLinux	496	
11.	Design Examples and Case Studies of Program Modeling and Programming with RTOS-I		511
11.1	Case Study of Embedded System Design and Coding for an Automatic Chocolate Vending Machine (ACVM) Using Mucos RTOS	512	
11.2	Case Study of Digital Camera Hardware and Software Architecture	531	
11.3	Case Study of Coding for Sending Application Layer Byte Streams on a TCP/IP Network Using RTOS Vxworks	537	
12.	Design Examples and Case Studies of Program Modeling and Programming with RTOS-2		566
12.1	Case Study of Communication Between Orchestra Robots	567	
12.2	Embedded Systems in Automobile	574	
12.3	Case Study of an Embedded System for an Adaptive Cruise Control (ACC) System in a Car	577	
12.4	Case Study of an Embedded System for a Smart Card	593	
12.5	Case Study of a Mobile Phone Software for Key Inputs	604	
13.	Embedded Software Development Process and Tools		618
13.1	Introduction to Embedded Software Development Process and Tools	620	
13.2	Host and Target Machines	623	
13.3	Linking and Locating Software	626	
13.4	Getting Embedded Software into the Target System	630	
13.5	Issues in Hardware-Software Design and Co-design	634	
14.	Testing, Simulation and Debugging Techniques and Tools		648
14.1	Testing on Host Machine	649	
14.2	Simulators	650	
14.3	Laboratory Tools	653	
	<i>Appendix 1: Roadmap for Various Course Studies</i>		662
	<i>Appendix 2: Select Bibliography</i>		663
	<i>Index</i>		668

Walkthrough

1.1 EMBEDDED SYSTEMS

1.1.1 System

A system is a way of working, organizing or doing one or more tasks according to a fixed plan, program, or set of rules. A system is also an arrangement in which all its units assemble and work together according to the plan or program.

Consider a watch. It is a time-display system. Its parts are its hardware, needles, and battery with the beautiful dial, hands and strap. These parts organize to show the real time every second and continuously update the time every second. The system program updates the display using three needles after each second. It follows a set of rules. Some of these rules are as follows: (i) All needles move only clockwise. (ii) A thin and long needle rotates every second such that it returns to same position after a minute. (iii) A long needle rotates every minute such that it returns to same position after an hour. (iv) A short needle rotates every hour such that it returns to same position after twelve hours. (v) All these needles return to the same inclination after twelve hours each day.

Consider a washing machine. It is an automatic clothes-washing system. The important hardware parts include its status display panel, the switches and dials for user defined programming, a motor to rotate or spin, its power supply and control unit, an inner water level sensor, a wireless valve for letting water in and another valve for letting water drain out. These parts organize to wash clothes automatically according to a program preset by a user. The system program is activated to wash the dirty clothes placed in a tank, which rotates or spins in preprogrammed steps and stages. It follows a set of rules. Some of these rules are as follows: (i) Follow the steps strictly in the following sequence: Step I: Wash by spinning the motor according to a programmed period. Step II: Rinse in fresh water after draining out the dirty water, and rinse a second time if the system is not programmed in water-saving mode. Step III: After draining out the water completely, spin the motor fast for a programmed period for drying by centrifuging out water from the clothes. Step IV: Show the wash-over status by a blinking display. Sound the alarm for a minute to signal that the wash cycle is complete. (ii) At each step, display the process stage of the system. (iii) In case of an interruption, execute only the remaining part of the program, starting from the position when the process was interrupted. There can be no repetition from step I unless the user resets the system by inserting another set of clothes and resets the program.

1.1.2 Embedded System

Definition One of the definitions of embedded system is as follows:

"An embedded system is a system that has embedded software and computer-hardware, which makes it a system dedicated for an application or specific part of an application or product or a part of a larger system."

Embedded systems have been defined in books published recently at several ways. Given below is a series of definitions from others in the field:

Wayne Wolf, author of *Computers as Components - Principles of Embedded Computing System Design*: "What is an embedded computing system? Loosely defined, it is any device that includes a programmable computer but is not itself intended to be a general-purpose computer" and "a fax machine or a clock built from a microprocessor is an embedded computing system"

Simple approach with interesting examples and figures

Simple approach with figures to explain complex topic of system on chip for a mobile phone

5. Discrete cosine transforms for signal processing applications.
6. Memories.
7. Multiple standard source solutions called IP (Intellectual Property) cores.
8. Programmable logic device and FPGA (Field Programmable Gate Array) cores.
9. Other logic and analog units.

An exemplary application of such an embedded SoC is the mobile phone. Single purpose processors, ASIPs and IPs on an SoC are configured to process encoding and decoding, analog, modulating, demodulating, interacting the key pad and multiple line LCD matrix displays or touch screens, saving data and recalling data from memory. Figure 1.10 shows an SoC that integrates internal ASIPs, internal processors (ASIPs), shared memories and peripheral interfaces on a common bus. Besides a processor, memories and digital circuits with embedded software for specific applications, the SoC may possess analog circuits as well.

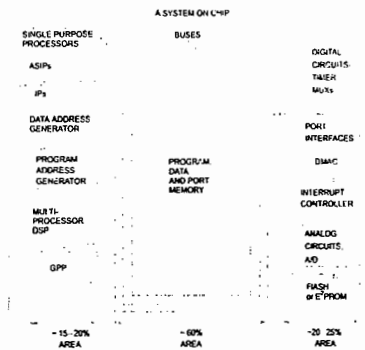


Fig. 1.10 A SoC embedded system and its common bus with internal ASIPs, internal processors, IPs shared memories and peripheral interfaces

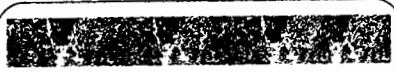
- 17. End of Code for displayTimeOne function */
- 18. ISR code for posting mailbox message to Task_Display */
- 19. ISR_Throttle ()
- 20. Codes for creating a message for time and date after each IRQ interrupt from the system RTC tick. */
- 21. OSMDemoPanic (MainTimeDateSetting, timeDate);
- 22. End of ISR_TimeDate code*/

11.2 CASE STUDY OF DIGITAL CAMERA HARDWARE AND SOFTWARE ARCHITECTURE

The digital camera was introduced earlier in Section 1.10.4. A digital camera is an example of SoC. (Section 1.6) Section 1.10.4 listed the camera functions, hardware and software units. Figure 1.14 showed the hardware and software components in a simple digital camera. Sections 11.2.1 and 11.2.2 give the design steps of a digital camera. Sections 11.2.3 and 11.2.4 describe hardware and software architecture.

11.2.1 Requirements

- Requirements of the digital camera can be understood through a requirement table given in Table 11.1. The detailed functions inside the camera are as follows:
- A set of controllers control shutter, flash, (for example, for peripheral, driver memory access, and focus, auto focus and eye-ball image control. GUI consists of the LCD display for graphics, and various and buttons for input to camera. A touchscreen is another alternative for LCD and keypad. The user gives commands for switching on the camera, flash, shutter, adjust, brightness, contrast, color, save and transfer. The user commands are in the form of on-screen signals. Each signal generates from a user input from an external switch or button. When a button is pressed, the shutter is released, a flash lamp glows and a self-timer circuit switches off the lamp automatically.
 - The picture generation light, which falls on the CCD array, which through an A/D converter the bit for each pixel in each row in the frame, and also for the dark area pixels in each row in a vertical strip. The other correction in CCD signaled light intensities for each row. The strip is in adjacent frame.
 - A picture consists of a number of pixels. The number of pixels used for a picture determines resolution. 2092 x 1484 is a 3000-dot sets of cells. Each set of pixel has three cells, for the red, green and blue components in a pixel. Each cell gets exposed to a picture when the shutter of camera opens on user command. The camera records the picture using a charge-coupled device (CCD) array. The array consists of a large number of CCD cells, three at each pixel.



We will learn embedded systems design from three case studies discussed in this chapter and four in the next chapter. The first case study is of an automatic chocolate vending machine (ACVM) which was introduced earlier in Section 1.10.2. The second study is of a digital camera, introduced earlier in Section 1.10.4. The third study is of a TCP/IP stack, which was introduced earlier in Section 1.11. The objectives of these case studies are as follows:

- To learn how the requirements are stated and specifications of a system are listed.
- To learn how UML modeling is used to model the design of the system.
- To learn to define hardware architecture using microprocessors or microcontrollers or ASICs or DSPs and devices.
- To learn how to define the software architecture for software external modules and related systems and define the decomposition of software into modules, components, appropriate protection strategies and means of software.
- To learn coding for design implementation using MUCOS and VxWorks RTOSes, and also the use of IFCs for task synchronization and communication processing.

11.1 CASE STUDY OF EMBEDDED SYSTEM DESIGN AND CODING FOR AN AUTOMATIC CHOCOLATE VENDING MACHINE (ACVM) USING MUCOS RTOS

ACVM was introduced earlier in Section 1.10.2. It listed ACVM functions, hardware and software units. Figure 1.12 showed a diagrammatic representation of ACVM. Sections 11.1.1 to 11.1.6 give the design steps of an ACVM. Section 9.2 described MUCOS RTOS. It has a portable, ROMable, scalable, preemptive, real time and multitasking kernel. Section 11.1.7 describes coding for ACVM using MUCOS RTOS (Section 9.2) programming environment.

11.1.1 Requirements

The requirements of the machine can be understood through a requirement table given in Table 11.1.

11.1.2 Specifications

- The ACVM specifications are brief as follows:
- It has an alphanumeric keypad on the top of the machine. This enables a child to interact with a robot having a character. The screen can also command and interact with the machine.

Explains modeling of programs and software engineering practices for system design by case studies of systems for automatic chocolate vending machine, digital camera, TCP/IP stack creation, robot orchestra, automatic cruise control, smart card and mobile phone



We will learn the following in this chapter:

- Basic functions and types of RTOSes.
- RTOS mCOS-II (referred as MUCOS in the text) through 20 examples—Example 9.1 to 9.20. What operations are performed and how values are returned for each given MUCOS function will be explained. Emphasis on the use of functions in the MUCOS is important for a reader even if another RTOS is used later. This will help greatly in understanding the advanced, sophisticated and embedded RTOSes later on.
- VxWorks from Wind River® Systems is also an RTOS for sophisticated embedded systems with powerful tool support through seven examples—Example 9.21 to 9.27. Differences between the VxWorks semaphores, mailboxes and queues with respect to that of MUCOS will be made clear. Chapter 10 will describe the Windows CE, OSEK and real-time Linux (RTOSes).

9.1 BASIC FUNCTIONS AND TYPES OF RTOSes

- A complex multitasking embedded system design requires the development of thoroughly tested bug-free codes for
- Integrated development environment.
 - Real functions in embedded C or embedded C++.
 - Real-time clock-based hardware and software timers.
 - Scheduler.
 - Device drivers and device manager.
 - Functions for IPCs using the signals, event flag group, semaphore-handling functions and functions for the queues, mailboxes, pipe and wickets.
 - Additional functions, for example, TCP/IP or USB or Bluetooth or WiFi or ZigBee and GPRS.
 - Error and exception handling functions.
 - Testing and system debugging software for testing RTOSes as well as developed embedded application.

Figure 9.1(a) shows the basic functions expected from the kernel of an RTOS. The RTOSs have the following features in general:

- Basic kernel functions and scheduling pre-emptive or pre-emptive plus time sharing.
- Priority inheritance feature or option of priority ceiling feature.
- Limits for number of tasks.
- Task synchronization and IPC functions.
- IDE consisting of editor, platform holder, GUI and graphics software, compiler, debugger and host target support tools.
- Device manager tool and device drivers.

- Change PRIOE queue: Important events in using the RTOSs queue are as follows:
 - The function `mqPriority` is used to set the priority of a PRIOE queue.
 - The function `mqGet` is used to check if a task is available and remove a signal queue.
 - The function `mqSet` is used to set the priority of a PRIOE queue.
 - The function `mqSend` is used to send a signal to a task and lock a queue.
 - The function `mqWait` is used to wait for a signal until the message is available. The message is available for a single task, which has been recognized for a notification (in extended version) on take into account, ready. This provision is extremely useful for a server task. A server task receives the notification from a client task through a signal-handling function (like an ISR).
 - The function `mqWait` retrieves the address of a PRIOE queue.
 - The RTOSs queue function `mqWait` does not destroy the queue immediately but prevents the other task from using the queue. The queue will get destroyed only if the last task destroys the queue. In any case, the user should be aware of the memory associated with queue IFCs.

VxWorks queues have the additional following features: (i) Time out option can be used. (ii) Two options, FIFO and task priority, for queue wait by multiple tasks. PRIOE queues have the additional following feature: task priority. In case a single waiting task is available and there can be 32 message priority levels in place of one priority level in LIFO or FIFO. When a task exists, a task ID allocates (Section 9.1.1). When using the task-related functions, the number facilitates task identity. Similarly, a pipe (Section 7.14) or wicket (Section 7.15) or file (Section 8.6.2) when created, a file descriptor data structure is created and a number, for example, fd, is assigned to identify the device created. The number is assigned after examining a set of the numbers already allocated. When using the device-related functions, the number facilitates the device identity. The device function examples are open or read or write or get attribute or set or attribute close (Section 8.6). A pipe in VxWorks is a FIFO queue, which is managed by the queue IPC functions but by the device driver functions. VxWorks has management functions for a pipe-driver (like a device driver) priority. This is analogous to the named pipe driver in Unix. Pipes also implement the unidirectional link between a set of tasks. Function `pipeOpen` (Section 8.6.2) creates a pipe-driver named `pipeName` for maximum number of messages. Each message can be of maximum size `maxMsgSize`. It returns (in a file of device or create or dev) a function retrieves the list of devices with the device number allotted to each device including pipe devices. Consider an example for creating a pipe named `pipeUserIn`. Assume that it can have a maximum of four messages per name, password, telephone number and e-mail ID. Each of these can be of a maximum size of 32 bytes only. A global variable `fd` is an integer number for a file descriptor that identifies a device among a number of devices at the RT system. The device can be a file or pipe or socket or other device. Example 9.26 explains the codes for creating, writing and reading.

```

Example 9.26
1. #include "StdLib.h" /* Include the IO library functions. */
pipeDev ( ) /* Create a pipe device. */
2. /* Declare file descriptor. */
int fd;
3. /* Mode refers to the permission as an NFS (Network File Server). Mode is read as 0 for unrestricted permission.
int mode;
4. /* Create pipe named as pipeUserIn for 4 messages, each 32 bytes maximum. */

```

- 24. /* Write the array elements after encryption. */
- ApplyE in SmartOS (in type in smartApp, DHS).
- SmartUSQ (in SmartStart, AppSW).
- 25. /* Resume Detached task. */
- SmartUSQ (in SmartStart, AppSW).
- 26. /* End of While loop. */
- 27. /* End of task_App1 Code. */

12.5 CASE STUDY OF A MOBILE PHONE SOFTWARE FOR KEY INPUTS

Mobile phones are smart. Each phone has many APIs. Example of APIs are phone, SMS (short message service), MMS (multimedia messaging service), e-mail, address book, web-browsing, calendar, talk-to-us (VoIP), WordPad, PocketWord, PocketExcel, one-step for memos, PocketPDF, slide show and cameras. Mobile phone with a large touchscreen uses a virtual keypad. Mobile phone with a small screen uses a physical keypad. The present case study refers to SMS using application in a mobile phone with a physical keypad. Section 12.5.1 gives the requirements of SMS create and send application. Section 12.5.2 gives the classes and class diagrams. Section 12.5.3 gives the state diagram and Section 12.5.4 gives communication hardware. Section 12.5.5 describes software architecture. Section 12.5.6 describes the software task- and synchronization Model for the application.

12.5.1 Requirements

A processor, keypad, screen, scratch pad memory, persistence memory and communication units are required for SMS create and send application. Scratch pad memory addresses are used for temporary storage of characters (bytes) during the application. Persistence memory addresses are used to store such data as when a change is made in the byte, it persists even after the power switches off. Further, when there is a change, there is an identical change is reflected in other correlated objects. For example, a name is added in a file for the Contacts, the same change takes in the file for Address book for sending the e-mails. Figure 12.50 shows specific units, which are used for the SMS create application. The screen is used for displaying the menu. Figure 12.50 shows that there are four cursor keys (up, down, left and right) denoted by C1, C2, C3, and C4. In navigation keypads, four different cursor keys are used. The mobile cursor key in the keypad is such that it is for up, or left or right or down. When the key is pressed towards the left the cursor moves left (←) which is denoted by C1. Similarly, when the right the cursor moves right (→), and so on for up (↑) or down (↓). In addition, there are four command keys (right-corner second-row, left-corner second-row, right-corner first-row and left-corner first-row) denoted by KeyRow2, KeyRow1, KeyRow3 and KeyRow4. Also, there are six TR keys for numbers 1 to 9 as well as for alphabets a to z (A to Z). There are two mode-keys (KeyM1 and KeyM2) and one key for keying in a text character number 0 to 9. Alphabetic text in small case or capital case is controlled by a mode-key 1 state. Text character entered on keying depends on state of the TR key. (Refer to Examples 2.6 and 4.8 and Section 6.3.)

internal communication. Extension does not permit creation and deletion of task during run. Extension defines each task of different priority and activates it only once as the codes. Extension does not use message queues and uses semaphores as event flag only with no task having run-time deletion or creation of these.

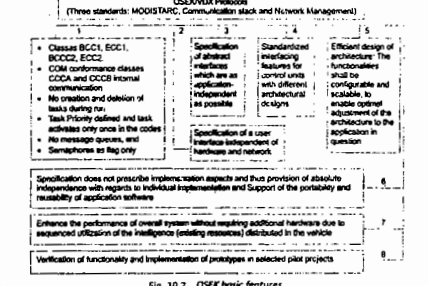


Fig. 10.2 OSEK basic features

Appendix 2: Select Bibliography

A. PRINTED BOOKS

1. Graham Phillips, Bill Pines and John Martin, "Linux Appliance Design: A Hands-On Guide to Building Linux Appliances", BS Search Press, 2002.
2. Giuseppe Borzinger and Fred Veeninger (Eds.), "Lessons on Embedded Systems: European Educational Forum School on Embedded Systems, Waltham", Springer, Mar 2000.
3. Michael Gier and Andrew James, "Programming Embedded Systems: With C and GNU Development Tools", 2nd Edition, O'Reilly, Oct 2006.
4. Nandan Gopal and Raj Kamal (eds.), "Computer Architecture", Samsam Series Texts Edition, May 2008.
5. Peter Marwedel, "Embedded System Design", Springer Verlag, New York 2005.
6. Rajeev P. Laxmi and Suresh Nishikumar, "Embedded Linux System Design and Development", Auerbach Publications, Taylor and Francis, Dec 2002.
7. Brian Bornerman and Joseph Stabile, "Embedded System Design: The Art and Science of Research and Development", Springer, 2005.
8. Tammy Noergard, "Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers", Newnes, Boston with Hingham, MA, USA, 2002.
9. Raj Kamal, "Microcontroller Architecture, Programming, Interfacing and System Design", Pearson Education, Singapore, 2001.
10. Jack Casade (Ed.), "The Firmware Handbook", Newnes, Boston with Hingham, MA, USA, 2004.
11. Jack Casade and Michael Bart, "The Embedded Systems Dictionary", CMP Books, 2003.
12. Prasad K. V. K., "Embedded Real Time Systems: Concepts, Design and Programming - The Ultimate Reference", Dreamtech, 2003.
13. Douglas Beatty "Programming Microsoft WINDOWS CE.NET", Microsoft, USA, 2001.
14. John Casade, "Designing Embedded Hardware", 2nd Edition, Elsevier, 2003.
15. Prasad K. V. K., "Microcontroller Design, Interfacing, Programming for Embedded Systems - Crossing the Code", Wiley, New Delhi, 2002.
16. Houshuo W. Yin, "Embedded Microcontroller Systems: Real Time Interfacing", Pearson, Stamford, 2002.
17. Stephen Palmer and John Pilling, "A Practical Guide to Firmware Development", Prentice Hall, 2002.
18. Stuart R. Ball, "Embedded Microprocessor Systems: Real World Design", Butterworth Heinemann, Boston, MA, USA, 1996. (3rd Edition, May 2002).
19. Pithy A. Lakshmi, "Real Time System Design and Analysis - An Engineer's Handbook", 2nd Edition, IEE Press, USA, 1997 (Prentice Hall India, Third Indian Reprint, April, 2001).

Appendix 2: Select Bibliography

20. <http://www.embedded.com> (For subscribing to a popular embedded system magazine)
21. <http://www.ETSystems.com> (For subscribing to a popular embedded system magazine)
22. <http://www.cet.com/etbooks> (For subscribing to a popular embedded system magazine)
23. <http://www.embedded-computing.com> (For subscribing to a popular embedded computing system magazine)
24. <http://www.embedded.com> (A popular embedded system site)
25. <http://www.wikipedia.com> (A popular world wide web encyclopedia on website to clarify the key terms in various topics)

C. PRINTED JOURNAL PAPER REFERENCES

1. Huan Lu, Krishna Ramasubramanian, Prabhat Shrivastava, Indrak A. Gupta and John D. Sweeney, "Resource Management for Real Time Tasks in Mobile Robots", *Journal of Systems and Software*, 80(7), 962-971, 2006.
2. G. R. Gadh, N. Sharma, K. Ramasubramanian, S. Mahapatra, "Efficient Real Time Support for Autonomous Applications: A case Study", *Proc. of 12th IEEE International Conference on Embedded Real Time Computing Systems and Applications*, 2006.
3. Lari-Berni, Frohlichmann and Kriener, "CAN for Critical Embedded Assymetric Network", *IEEE Mem*, 23(4), 28-33, 2002.
4. Wayne Wolf, Benik Chet and Tachin Lu, "Smart Camera as Embedded System", *IEEE Micro*, 22(5), 48-55, 2002.
5. Sam Louis Bredt, "Enabling Hardware Software Co-design with Verilog Pro FPGA", *ICVJ Journal*, pp 24-29, Summer issue, 2002.
6. Che-Ying Liang and Hsin-Ping, "Optimal Adaptive Cruise Control with Geometric Strong Stability", *Journal Vehicle System Dynamics*, 38, pp 313-330, 1999.
7. Tadas Mikenas, "Fuel Area, Properties, Analysis and Application", *Proc. IEEE*, 77(4), 541-550, 1989.

Detailed selected bibliography of books, journal references and important web links at end of the book to facilitate building a startup library for references and further studies in Embedded Systems

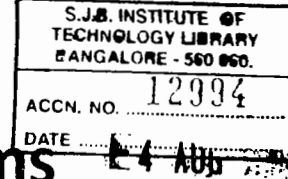
666 Embedded Systems

85. Cary F. M., *Microcontrollers and Microcomputers—Principles of Software and Hardware Engineering*, Oxford University Press, New York, 1999.
86. Rajeev T. M. Choudh, A. Venkatesh, N. Huan, A. E. Larygin, C. Passerone, A. Sanguineti, V. Vincenzi, S. Seniveth, K. Suresh, and R. Tahara, *Embedded Software Co-Design of Embedded Systems: A Field Approach*, Morgan, USA, Elsevier Academic Publishers, June 1997.
87. Jane Ferrara Borner, *Embedded System Programming in C and Assembly*, Van Nostrand, Reinhold, New York, USA, 1996.
88. Peter Sipson, *Microcontroller Technology: The ARMCH1, 2nd Edition*, Prentice Hall, Englewood Cliffs, NJ, 1996.
89. Fred Mitchell, *Data Communication, Computer Networks and Open Systems*, 4th Edition, Pearson Education, 1996 (Harcourt Indian Reprint, 2001).
90. Subbathal and PB Gulati, *Operating Systems*, Addison Wesley, Reading, MA, USA, 1996.
91. Peter Marwedel and Carl Ginet, *Code Generation for Embedded Processors*, Kluwer Academic Publishers, Dordrecht, 1993.
92. Daniel Tabak, *Advanced Microprocessors*, McGraw-Hill, USA, 1995.
93. Gayle, Daniel D., *From Value Stream Mapping to the Core: An Overview and Overview of Embedded Systems*, Englewood Cliffs, NJ, Prentice Hall, 1992.
94. Franklin G. F., J. D. Pirelli and A. E. Khamis, *Real-time Control of Dynamic Systems*, 2nd Ed., Addison Wesley, Reading, MA, USA, 1994.
95. Steven J. W., *The ARM Microcontroller—Hardware, Software and Interfacing*, Prentice Hall, 1993.
96. Walter J. Gonzalez and Thomas L. Vincent, *Advanced Control Systems—Analysis and Design*, John Wiley, 1993.
97. Huan Lu, J. and Daniel Tabak, *Microprocessors—Architecture, Implementation and Programming*, McGraw-Hill, 1992.
98. Jack G. Casade, *Art of Programming Embedded Systems*, Academic, USA, 1992.
99. Greenfield C. D., *The ARMCH1 Microcontroller*, Saunders College Publishing, 1991.
100. Rajeev T. M., *Design with Microcontrollers and Microprocessors*, McGraw-Hill, 1992.

B. WEBSITE REFERENCES

1. <http://www.armv7tts.com> (For the ARMv7TTS Processor, Section 1.2.4, 2.3.6)
2. <http://www.armv7tts.com/armv7tts> (For Section 1.4)
3. <http://www.armv7tts.com/armv7tts/armv7tts.html> and <http://www.armv7tts.com> (For Section 2.3.3)
4. <http://www.armv7tts.com> (Section 2.3.3, For ARM Processor)
5. <http://www.armv7tts.com/armv7tts/armv7tts.html> (For Section 2.3.6)
6. <http://www.armv7tts.com> (For benchmarking of performance of embedded systems, Section 2.6 and 13.5.3)
7. <http://www.armv7tts.com> (For Section 5.7.5)
8. <http://www.armv7tts.com> (For Section 8.1)
9. <http://www.armv7tts.com> (For Section 9.3)
10. <http://www.armv7tts.com> (For Section 10.1)
11. <http://www.armv7tts.com> (For Section 11.1)
12. <http://www.armv7tts.com> (For Section 11.2)
13. <http://www.armv7tts.com> (For Section 11.3)
14. <http://www.armv7tts.com> (For Section 12.1)
15. <http://www.armv7tts.com> (For Section 12.2)
16. <http://www.armv7tts.com> (For Section 12.3)
17. <http://www.armv7tts.com> (For Section 12.4)
18. <http://www.armv7tts.com> (For Section 12.5)
19. <http://www.armv7tts.com> (For Section 12.6)
20. <http://www.armv7tts.com> (For Section 12.7)
21. <http://www.armv7tts.com> (For Section 12.8)
22. <http://www.armv7tts.com> (For Section 12.9)
23. <http://www.armv7tts.com> (For Section 12.10)
24. <http://www.armv7tts.com> (For Section 12.11)
25. <http://www.armv7tts.com> (For Section 12.12)
26. <http://www.armv7tts.com> (For Section 12.13)
27. <http://www.armv7tts.com> (For Section 12.14)
28. <http://www.armv7tts.com> (For Section 12.15)
29. <http://www.armv7tts.com> (For Section 12.16)
30. <http://www.armv7tts.com> (For Section 12.17)
31. <http://www.armv7tts.com> (For Section 12.18)
32. <http://www.armv7tts.com> (For Section 12.19)
33. <http://www.armv7tts.com> (For Section 12.20)
34. <http://www.armv7tts.com> (For Section 12.21)
35. <http://www.armv7tts.com> (For Section 12.22)
36. <http://www.armv7tts.com> (For Section 12.23)
37. <http://www.armv7tts.com> (For Section 12.24)
38. <http://www.armv7tts.com> (For Section 12.25)
39. <http://www.armv7tts.com> (For Section 12.26)
40. <http://www.armv7tts.com> (For Section 12.27)
41. <http://www.armv7tts.com> (For Section 12.28)
42. <http://www.armv7tts.com> (For Section 12.29)
43. <http://www.armv7tts.com> (For Section 12.30)
44. <http://www.armv7tts.com> (For Section 12.31)
45. <http://www.armv7tts.com> (For Section 12.32)
46. <http://www.armv7tts.com> (For Section 12.33)
47. <http://www.armv7tts.com> (For Section 12.34)
48. <http://www.armv7tts.com> (For Section 12.35)
49. <http://www.armv7tts.com> (For Section 12.36)
50. <http://www.armv7tts.com> (For Section 12.37)
51. <http://www.armv7tts.com> (For Section 12.38)
52. <http://www.armv7tts.com> (For Section 12.39)
53. <http://www.armv7tts.com> (For Section 12.40)
54. <http://www.armv7tts.com> (For Section 12.41)
55. <http://www.armv7tts.com> (For Section 12.42)
56. <http://www.armv7tts.com> (For Section 12.43)
57. <http://www.armv7tts.com> (For Section 12.44)
58. <http://www.armv7tts.com> (For Section 12.45)
59. <http://www.armv7tts.com> (For Section 12.46)
60. <http://www.armv7tts.com> (For Section 12.47)
61. <http://www.armv7tts.com> (For Section 12.48)
62. <http://www.armv7tts.com> (For Section 12.49)
63. <http://www.armv7tts.com> (For Section 12.50)
64. <http://www.armv7tts.com> (For Section 12.51)
65. <http://www.armv7tts.com> (For Section 12.52)
66. <http://www.armv7tts.com> (For Section 12.53)
67. <http://www.armv7tts.com> (For Section 12.54)
68. <http://www.armv7tts.com> (For Section 12.55)
69. <http://www.armv7tts.com> (For Section 12.56)
70. <http://www.armv7tts.com> (For Section 12.57)
71. <http://www.armv7tts.com> (For Section 12.58)
72. <http://www.armv7tts.com> (For Section 12.59)
73. <http://www.armv7tts.com> (For Section 12.60)
74. <http://www.armv7tts.com> (For Section 12.61)
75. <http://www.armv7tts.com> (For Section 12.62)
76. <http://www.armv7tts.com> (For Section 12.63)
77. <http://www.armv7tts.com> (For Section 12.64)
78. <http://www.armv7tts.com> (For Section 12.65)
79. <http://www.armv7tts.com> (For Section 12.66)
80. <http://www.armv7tts.com> (For Section 12.67)
81. <http://www.armv7tts.com> (For Section 12.68)
82. <http://www.armv7tts.com> (For Section 12.69)
83. <http://www.armv7tts.com> (For Section 12.70)
84. <http://www.armv7tts.com> (For Section 12.71)
85. <http://www.armv7tts.com> (For Section 12.72)
86. <http://www.armv7tts.com> (For Section 12.73)
87. <http://www.armv7tts.com> (For Section 12.74)
88. <http://www.armv7tts.com> (For Section 12.75)
89. <http://www.armv7tts.com> (For Section 12.76)
90. <http://www.armv7tts.com> (For Section 12.77)
91. <http://www.armv7tts.com> (For Section 12.78)
92. <http://www.armv7tts.com> (For Section 12.79)
93. <http://www.armv7tts.com> (For Section 12.80)
94. <http://www.armv7tts.com> (For Section 12.81)
95. <http://www.armv7tts.com> (For Section 12.82)
96. <http://www.armv7tts.com> (For Section 12.83)
97. <http://www.armv7tts.com> (For Section 12.84)
98. <http://www.armv7tts.com> (For Section 12.85)
99. <http://www.armv7tts.com> (For Section 12.86)
100. <http://www.armv7tts.com> (For Section 12.87)
101. <http://www.armv7tts.com> (For Section 12.88)
102. <http://www.armv7tts.com> (For Section 12.89)
103. <http://www.armv7tts.com> (For Section 12.90)
104. <http://www.armv7tts.com> (For Section 12.91)
105. <http://www.armv7tts.com> (For Section 12.92)
106. <http://www.armv7tts.com> (For Section 12.93)
107. <http://www.armv7tts.com> (For Section 12.94)
108. <http://www.armv7tts.com> (For Section 12.95)
109. <http://www.armv7tts.com> (For Section 12.96)
110. <http://www.armv7tts.com> (For Section 12.97)
111. <http://www.armv7tts.com> (For Section 12.98)
112. <http://www.armv7tts.com> (For Section 12.99)
113. <http://www.armv7tts.com> (For Section 13.00)

Introduction to Embedded Systems



1

L
E
A
R
N
I
N
G
O
B
J
E
C
T
I
V
E
S

Section 1.1

Definitions of system and embedded system

Section 1.2

The processing unit of an embedded system consists of

1. A processor
2. Commonly used microprocessors
3. Application-specific instruction set processors (ASIPs), microcontrollers, DSPs and others
4. Single purpose processors

Section 1.3

The hardware unit of an embedded system consists of

1. An embedded system power source with controlled power-dissipation
2. A clock oscillator circuit and clocking unit that lets a processor execute instructions
3. Timers and a real time clock (RTC) for various timing needs of the system
4. Reset circuit and watchdog timer
5. System and external memories
6. System input output (IO) ports, serial, parallel and wireless communication, serial Universal Asynchronous Receiver and Transmitter (UART) and other port protocols and buses
7. Devices such as Digital to Analog Converter (DAC) using Pulse Width Modulation (PWM), Analog to Digital Converter (ADC), Light Emitting Diode (LED) and Liquid Crystal Display (LCD) units, keypad and keyboard, touch screen, pulse dialer, modem and transceiver
8. Multiplexers, demultiplexers, decoder for interfacing of the devices and buses

L

9. Interrupt controller (handler)

E

Section 1.4

1. Languages that are used to develop *embedded software* for a system
2. Program models
3. Multitasking using an operating system (OS), system device drivers, device management and real time operating system (RTOS)
4. Software tools for system design

A

R

N

Section 1.5

Examples of applications of embedded systems

I

Section 1.6

Designing an embedded system on a VLSI chip

N

1. Embedded SoC (System on Chip) and examples of its applications
2. Uses of Application Specific Instruction Set Processor (ASIP) and Intellectual Property (IP) core
3. Field Programmable Gate Array (FPGA) core with single or multiple processor units on an ASIC chip

G

Section 1.7

The complex system consists of

O

1. Embedded microprocessors or GPPs in complex systems
2. Embedding ASIPs, microcontrollers, DSPs, media and network processors
3. Embedding application-specific system processors (ASSIPs)
4. Embedding multiple processors in systems

B

J

Section 1.8

The design process has

E

1. Challenges in embedded system design
2. Design metrics optimization
3. Co-design of hardware and software components

C

T

Section 1.9

The system design formalism is defined

I

Section 1.10

The design of embedded hardware and software in an automatic chocolate vending machine, smart card, digital-camera, mobile phone, mobile computer and robot are given as examples

V

E

Section 1.11

Classification of embedded systems into three types

S

Section 1.12

Skills needed to design an embedded system

1.1 EMBEDDED SYSTEMS**1.1.1 System**

A system is a way of working, organizing or doing one or many tasks according to a fixed plan, program, or set of rules. A system is also an arrangement in which all its units assemble and work together according to the plan or program.

Consider a watch. It is a time-display system. Its parts are its hardware, needles and battery with the beautiful dial, chassis and strap. *These parts organize to show* the real time every second and continuously update the time every second. The system-program updates the display using three needles after each second. *It follows a set of rules.* Some of these rules are as follows: (i) All needles move only clockwise. (ii) A thin and long needle rotates every second such that it returns to same position after a minute. (iii) A long needle rotates every minute such that it returns to same position after an hour. (iv) A short needle rotates every hour such that it returns to same position after twelve hours. (v) All three needles return to the same inclination after twelve hours each day.

Consider a washing machine. It is an automatic clothes-washing system. The important hardware parts include its status display panel, the switches and dials for user-defined programming, a motor to rotate or spin, its power supply and control unit, an inner water-level sensor, a solenoid valve for letting water in and another valve for letting water drain out. *These parts organize to wash clothes automatically according to a program preset by a user.* The *system-program* is activated to wash the dirty clothes placed in a tank, which rotates or spins in preprogrammed steps and stages. *It follows a set of rules.* Some of these rules are as follows: (i) Follow the steps strictly in the following sequence. Step I: Wash by spinning the motor according to a programmed period. Step II: Rinse in fresh water after draining out the dirty water, and rinse a second time if the system is not programmed in water-saving mode. Step III: After draining out the water completely, spin the motor fast for a programmed period for drying by centrifuging out water from the clothes. Step IV: Show the wash-over status by a blinking display. Sound the alarm for a minute to signal that the wash cycle is complete. (ii) At each step, display the process stage of the system. (iii) In case of an interruption, execute only the remaining part of the program, starting from the position when the process was interrupted. There can be no repetition from Step I unless the user resets the system by inserting another set of clothes and resets the program.

1.1.2 Embedded System

Definition One of the definitions of *embedded system* is as follows:

"An embedded system is a system that has embedded software and computer-hardware, which makes it a system dedicated for an application(s) or specific part of an application or product or a part of a larger system."

Embedded systems have been defined in books published recently in several ways. Given below is a series of definitions from others in the field:

Wayne Wolf author of *Computers as Components – Principles of Embedded Computing System Design*: "What is an *embedded computing system*? Loosely defined, it is any device that includes a programmable computer but is not itself intended to be a general-purpose computer" and "a fax machine or a clock built from a microprocessor is an embedded computing system".

Todd D. Morton author of *Embedded Microcontrollers*: "Embedded Systems are electronic systems that contain a microprocessor or microcontroller, but we do not think of them as computers—the computer is hidden or embedded in the system."

David E. Simon author of *An Embedded Software Primer*: "People use the term *embedded system* to mean any computer system hidden in any of these products."

Tim Wilmshurst author of *An Introduction to the Design of Small Scale Embedded Systems* with examples from PIC, 80C51 and 68HC05/08 microcontrollers: (1) "An embedded system is a system whose principal function is not computational, but which is controlled by a computer embedded within it. The computer is likely to be a microprocessor or microcontroller. The word embedded implies that it lies inside the overall system, hidden from view, forming an integral part of [the] greater whole". (2) "An embedded system is a microcontroller-based, software-driven, reliable, real time control system, autonomous, or human- or network-interactive, operating on diverse physical variables and in diverse environments, and sold into a competitive and cost-conscious market".

A computer is a system that has the following or more components.

1. A microprocessor
2. A large memory of the following two kinds:
 - (a) Primary memory (*semiconductor* memories: Random Access Memory (RAM), Read Only Memory (ROM) and fast accessible caches)
 - (b) Secondary memory [*magnetic* memory located in hard disks, diskettes and cartridge tapes, *optical* memory in CD-ROMs or memory sticks (in mobile computers)] using which different user programs can be loaded into the primary memory and run
3. I/O units such as touch screen, modem, fax cum modem, etc.
4. Input units such as keyboard, mice, digitizer, scanner, etc.
5. Output units such as an LCD screen, video monitor, printer, etc.
6. Networking units such as an Ethernet card, front-end processor-based server, bus drivers, etc.
7. An operating system (OS) that has general purpose user and application software in the secondary memory

An embedded system is a system that has three main components embedded into it:

1. It embeds hardware similar to a computer. Figure 1.1 shows the units in the hardware of an embedded system. As its software usually embeds in the ROM or flash memory, it usually do not need a secondary hard disk and CD memory as in a computer
2. It embeds main application software. The application software may concurrently perform a series of tasks or processes or threads
3. It embeds a real-time operating system (RTOS) that supervises the application software running on hardware and organizes access to a resource according to the priorities of tasks in the system. It provides a mechanism to let the processor run a process as scheduled and context-switch between the various processes. (The concept of process, thread and task explained later in Sections 7.1 to 7.3.) It sets the rules during the execution of the application software. (A small-scale embedded system may not embed the RTOS.)

Characteristics An embedded system is characterized by the following: (1) Real-time and multirate operations define the ways in which the system works, reacts to events, interrupts and schedules the system's functioning in real time. It does so by following a plan to control latencies and to meet deadlines. (Latency refers to the waiting period between running the codes of a task or interrupt service routine and the instance at which the need for the task or interrupt from an event arises). The different operations may take place at

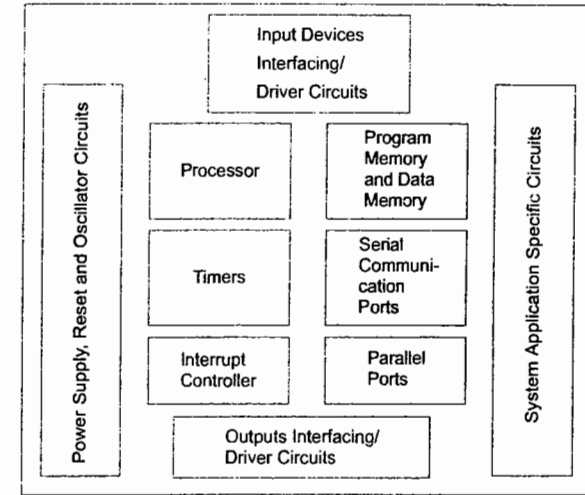


Fig. 1.1 The components of embedded system hardware

distinct rates. For example, audio, video, data, network stream and events have different rates and time constraints. (2) Complex algorithms. (3) Complex graphic user interfaces (GUIs) and other user interfaces. (4) Dedicated functions.

Constraints An embedded system is designed keeping in view three constraints: (1) available system-memory, (2) available processor speed, (3) the need to limit power dissipation when running the system continuously in cycles of 'wait for events', 'run', 'stop', 'wake-up' and 'sleep'.

The system design or an embedded system has constraints with regard to performance, power, size and design and manufacturing costs.

1.2 PROCESSOR EMBEDDED INTO A SYSTEM

A processor is an important unit in the embedded system hardware. It is the heart of the embedded system. Knowledge of basic concept of microprocessors and microcontrollers is must for an embedded system designer. A reader may refer to a standard text or the texts listed in the 'References' at the end of this book for an in-depth understanding of microprocessors, microcontrollers and DSPs that are incorporated in embedded system design. Chapter 2 will explain 8051 and a few processors.

1.2.1 Embedded Processors in a System

A processor has two essential units: Program Flow Control Unit (CU) and Execution Unit (EU). The CU includes a fetch unit for fetching instructions from the memory. The EU has circuits that implement the instructions pertaining to data transfer operations and data conversion from one form to another. The EU includes the Arithmetic and Logical Unit (ALU) and also the circuits that execute instructions for a program

control task, say, halt, interrupt, or jump to another set of instructions. It can also execute instructions for a call or branch to another program and for a call to a function.

A processor runs the cycles of fetch-and-execute. The instructions, defined in the processor instruction set, are executed in the sequence that they are fetched from the memory. A processor is in the form of an IC chip; alternatively, it could be in core form in an Application Specific Integrated Circuit (ASIC) or System on Chip (SoC). Core means a part of the functional circuit on the Very Large Scale Integrated (VLSI) chip.

An embedded system processor chip or core can be one of the following.

1. General Purpose Processor (GPP): A GPP is a general-purpose processor with instruction set designed not specific to the applications.
 - (a) Microprocessor. [Section 1.2.2]
 - (b) Embedded Processor [Section 1.7.7]
2. Application Specific Instruction-Set Processor (ASIP). An ASIP is a processor with an instruction set designed for specific applications on a VLSI chip.
 - (a) Microcontroller [Section 1.2.3]
 - (b) Embedded microcontroller [Section 1.7.7]
 - (c) Digital Signal Processor (DSP) and media processor [Section 1.7.3]
 - (d) Network processor, IO processor or domain-specific programmable processor
3. Single Purpose Processors as additional processors: Single purpose processor examples are as follows: (1) Coprocessor (e.g., as used for graphic processing, floating point processing, encrypting, deciphering, discrete cosine transformation and inverse transformation or TCP/IP protocol stacking and network connecting functions). (2) Accelerator (e.g., Java codes accelerator). (3) Controllers (e.g., for peripherals, direct memory accesses and buses). [Section 1.7.7]
4. GPP or ASIP cores integrated into either an ASIC or a VLSI circuit or a Field Programmable Gate Array (FPGA) core integrated with processor units in a VLSI (ASIC) chip. [Sections 1.6 and 1.7]
5. Application Specific System Processor (ASSP). [Section 1.7.9]
6. Multicore processors or multiprocessor [Section 1.7]

For a system designer, the following are important considerations when selecting a processor:

1. Instruction set
2. Maximum bits in an operand (8 or 16 or 32) in a single arithmetic or logical operation
3. Clock frequency in MHz and processing speed in Million Instructions Per Second (MIPS) or in an alternate metric *Dhrystone* for measuring processing performance [Section 2.6]
4. Processor ability to solve complex algorithms while meeting deadlines for their processing

A microprocessor or GPP is used because: (i) processing based on the instructions available in a predefined general purpose instruction set results in quick system development. (ii) Once the board and I/O interfaces are designed for a GPP, these can be used for a new system by just changing the embedded software in the ROM. (iii) Ready availability of a compiler facilitates embedded software development in high-level languages. (iv) Ready availability of well-tested and debugged processor-specific APIs (Application Program Interfaces) and codes previously designed for other applications results in new systems developed quickly.

1.2.2 Microprocessor

The CPU is a unit that centrally fetches and processes a set of general-purpose instructions. The CPU instruction set includes instructions for data transfer operations, ALU operations, stack operations, IO operations and

program control, sequencing and supervising operations. The general-purpose instruction set is always specific to a specific CPU. Any CPU must possess the following basic functional units:

1. A control unit that fetches and controls the sequential processing of a given command or instruction and communicates with the rest of the system.
2. An ALU that undertakes arithmetic and logical operations on bytes or words. It may be capable of processing 8, 16, 32 or 64-bit words at an instant.

A microprocessor is a single VLSI chip that has a CPU and may also have some other units (e.g., caches, floating point processing arithmetic unit, pipelining and superscaling units) that are additionally present and that result in faster processing of instructions.

The earlier generation microprocessor's fetch-and-execute cycle was guided by a clock frequency of the order of ~4 MHz. Processors now operate at a clock frequency of 4 GHz and even have multiple cores. In early 2002, it became possible to design Gbps (Giga bit per second) transceiver and encryption engines in a few highly sophisticated embedded systems using processors that operate on GHz frequencies. A transceiver is a transmitting cum receiving circuit that has appropriate processing and controls units, for example, for controlling bus-collisions. An encryption engine is a system that encrypts the data to be transmitted on the network.

Intel 80x86 (also referred as x86) processors are the 32-bit successors of 8086. [The x here refers to an 8086 extended for 32 bits.] Examples of 32-bit processors in 80x86 series are Intel 80386, 80486 and Pentiums (a new generation of 32- and 64-bit microprocessors is the classic Pentium series). IBM PCs use 80x86 series and the embedded systems incorporated inside the PC for specific tasks (like graphic accelerator, disk controllers, network interface card) use these microprocessors.

High performance processors have pipeline and superscalar architecture, fast ALUs and Floating Point Processing Units (FLPUs). [A pipeline architecture means that the instructions have between 3 and 9 stages. Different instructions are at different stages of the pipeline at any given instance. A superscalar architecture refers to two or more sets of instructions executing in parallel pipelines.]

The important microprocessors used in the embedded systems are ARM, 68HCxxx, 80x86 and SPARC family of microprocessors.

Section 1.7 will describe the embedding of a microprocessor GPP in complex systems.

A microprocessor is used as general-purpose processor when large embedded software has to be located in the external memory chips.

1.2.3 Microcontroller

A microcontroller is an integrated chip that has processor, memory and several other hardware units in it; these form the microcomputer part of the embedded system. Figure 1.2 shows the functional circuits present (in solid boundary boxes) in a microcontroller. It also shows the application-specific units (in dashed boundary boxes) in a specific version of a given microcontroller family.

Just as a microprocessor is the most essential part of a computing system, a microcontroller is the most essential component of a control or communication circuit. A microcontroller is a single-chip VLSI unit (also called 'microcomputer'), which, though having limited computational capabilities, possesses enhanced input-output capabilities and a number of on-chip functional units. [Refer to Section 1.3 for various functional hardware units.] Microcontrollers are particularly suited for use in embedded systems for real-time control applications with on-chip program memory and devices.

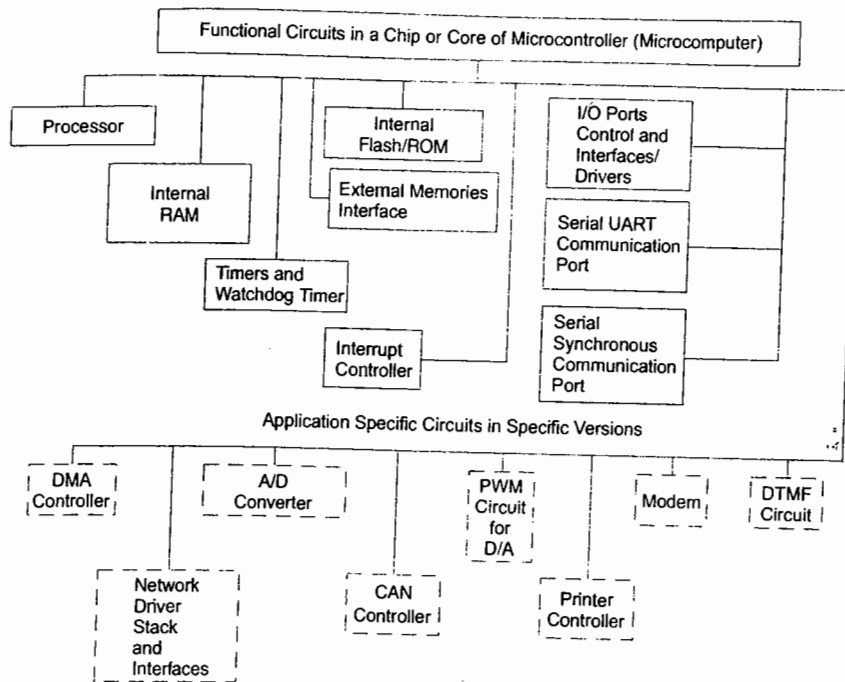


Fig. 1.2 Various functional circuits (solid boundary boxes) in a microcontroller chip or core in an embedded system. Also shown are the application-specific units (dashed boundary boxes) in a specific version of a microcontroller

A few of the latest microcontrollers also have dual core and high computational and superscalar processing capabilities. Important microcontroller chips for embedded systems are 8051, 8051MX, 68HC11xx, HC12xx, HC16xx, PIC 16F84 or 16C76, 16F876 and PIC18, microcontroller enhancements of ARM9/ARM7 from ARM, Intel, Philips, Samsung and ST microelectronics.

Figure 1.3 shows commonly used microcontrollers in small-, medium- and large-scale embedded systems. Choosing a microcontroller as a processing unit depends upon the application-specific features in it.

A microcontroller is used when a small or part of the embedded software has to be located in the internal memory and when on-chip functional units such as the interrupt-handler, port, timer, ADC, PWM and CAN controller are required.

1.2.4 Single Purpose Processors

Single purpose processors used in embedded systems include:

1. Coprocessor (for example, for floating point processing).

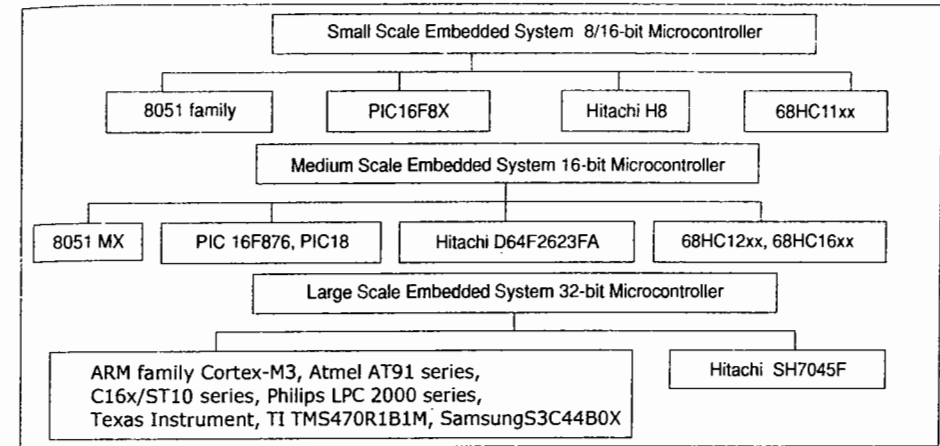


Fig. 1.3 Commonly used microcontrollers in small-, medium- and large-scale embedded systems

2. Graphics processor: An image consists of a number of pixels. For example, Quarter common intermediate format—Quarter-CIF images have 144×176 (horizontal x-axis \times vertical y-axis) pixels. Video frames have 525×625 pixels. The video graphic adapter (VGA) format of e-mailing and web pages has $640 \times 480 = 307,200$ pixels. A separate graphics processor is required for functions such as, for example, gaming, display from graphics memory buffers and to move (translate on screen) and rotate an image or its segments.
3. Pixel coprocessor: High-resolution pictures have formats: 2592×1944 pixels = 5,038,848 pixels; $2592 \times 1728 = 3.2$ M; $2048 \times 1536 = 3$ M and $1280 \times 960 = 1$ M. A pixel coprocessor is required in digital cameras for displaying images directly or after operations such as rotate right, rotate-left, rotate-up, rotate-down, shift to next, shift to previous.
4. Encryption engine: A suitable algorithm runs in this processor to encrypt data for secure transmission.
5. Decryption engine: A suitable algorithm runs in this processor to decrypt the encrypted data at receiver's end.
6. A discrete cosine transformation (DCT) and inverse transformation (DCIT) processor is required in speech and video processing.
7. Protocol stack processor: A protocol stack, which has a number of header words, is prepared before an application data is sent to a network. At the receiver's end, the protocol stack is received and application data is accepted accordingly. A TCP/IP protocol stack processor processes TCP/IP network data.
8. Network processor: A network processor's functions are to establish a connection, finish, send and receive acknowledgements, send and receive retransmission requests and check and correct received data frame errors. The network processor's functions include all protocol stack-processing functions.
9. Accelerator (for example, Java codes accelerator). The accelerator is a coprocessor that accelerates computations by taking advance actions that are just-in-time computations of the next object in Java programs.
10. CODEC (Coder and Decoder): A CODEC is a processor circuit that encodes input and decodes the encoded information or bits or signals into a complete set of bits or original signal. Voice, speech,

image, video signals and bits are encoded for storing or transmission and decoded from the stored or received bits or signal for display or playing. The CODEC functions as a compression and decompression unit for voice, speech, image or video signals.

11. JPEG CODEC: This is a processor for jpg compression and decompression. The Joint Photographic Experts Group (JPEG) is an International Telecommunication Union for Telecom (ITU-T) and International Standards Organisation (ISO) committee.
12. MPEG CODEC: The Motion Pictures Experts Group (MPEG) recommends CODEC standards for video. MPEG3 CODEC is a processor for mp3 compression and decompression. MPEG 2 or 3 or 4 compression of audio/video data streams is done before storing or transmitting, and decompression is done before retrieving or playing files. For MPEG compression and decompression algorithms, if GPP-embedded software is run, then separate DSPs are required to achieve real-time processing.
13. Controller (e.g., for peripheral, direct memory access or bus).

Single purpose processors are used for specific applications or computations or as controllers for peripherals, direct memory accesses and buses.

1.3 EMBEDDED HARDWARE UNITS AND DEVICES IN A SYSTEM

1.3.1 Power Source

Most systems have a power supply of their own. The Network Interface Card (NIC) and Graphic Accelerator are examples of embedded systems that do not have their own power supply and connect to PC power-supply lines. The supply has a specific operation range or a range of voltages. Various units in an embedded system operate in one of the following four power ranges: $5.0\text{ V} \pm 0.25\text{ V}$, $3.3\text{ V} \pm 0.3\text{ V}$, $2.0\text{ V} \pm 0.2\text{ V}$ and $1.5\text{ V} \pm 0.2\text{ V}$. There is generally an inverse relationship between propagation delay in the gates and operational voltage. Therefore, the 5 V system processor and units are used in most high performance systems.

Certain systems do not have a power source of their own: they connect to external power supply or are powered by the use of charge pumps (made up of a circuit of diode and capacitor that accumulate charge from the bus signals through which they connect or network to the host or from wireless radiation).

Low voltage operations

1. In portable or hand-held devices such as a cellular phone [when compared to 5 V, a CMOS 2 V circuit power dissipation reduces by one-sixth, $\sim (2/5\text{ V})^2$. This also increases the time intervals needed for recharging a battery by a factor of six.].
2. In a system with smaller overall geometry, low voltage system processors and IO circuits generate lesser heat and thus can be packed into a smaller space.

A power supply source or a charge pump is essential in every system.

1.3.2 Clock Oscillator Circuit and Clocking Units

The clock controls the time for executing an instruction. After the power supply, the clock is the basic unit of a system. A processor needs a clock oscillator circuit. The clock controls the various clocking requirements of

the CPU, of the system timers and the CPU machine cycles. The machine cycles are for fetching codes and data from memory and then decoding and executing them at the processor and for transferring the results to memory.

For processing units, a highly stable oscillator is required and the processor clock-out signal provides the clock for synchronizing all system units with the processor.

1.3.3 System Timers and Real-time Clocks

A timer circuit is suitably configured as the system-clock, which ticks and generates system interrupts periodically; for example, 60 times in 1s. The interrupt service routines then perform the required operation.

A timer circuit is suitably configured as the real-time clock (RTC) that generates system interrupts periodically for the schedulers, real-time programs and for periodic saving of time and date in the system.

The RTC or system timer is also used to obtain software-controlled delays and time-outs. An RTC functions as driver for software timers (SWTs). [Sections 3.6 and 3.8]

Microcontrollers also provide internal timer circuits for counting and timing devices.

To schedule the various tasks and for real-time programming, an RTC or system clock is needed. The clock also drives the timers for various timing and counting needs in a system.

1.3.4 Reset Circuit, Power-up Reset and Watchdog-Timer Reset

The program counter (PC) holds the address from where the instruction is to be fetched for execution. In 80x86 processors, the instruction pointer (IP) holds that address. A code segment register (CS) holds the base address of the code memory segment. The CS address equals the code starting address when the IP = 0 at the start of a code segment. The IP increments when the program executes the codes.

Reset means that the processor begins the processing of instructions from a starting address. That address is one that is set by default in the processor PC (or IP and CS in x86 processors) on a power-up. From that address in memory, program-instructions are fetched following the reset of the processor. A program that is reset and runs on a power-up can be one of the following: (i) A system program that executes from the beginning. (ii) A system boot-up program. (iii) A system initialization program.

In certain processors, for example, 68HC11 and HC12, there are two start-up addresses. One is based on the power-up reset vector and the other on the reset vector after the reset instruction or after a time-out (for example, from a watchdog timer). The processor fetches the bytes for the PC from the first power-up reset vector on power-up. The processor fetches the bytes for the PC from the second reset vector on the watchdog timer timing out or on executing the reset instruction.

The reset circuit activates for a fixed period (a few clock cycles) and then deactivates. The processor circuit keeps the reset pin active and then deactivates to let the program proceed from a default beginning address. The reset pin or the internal reset signal, if connected to the other units (for example, the IO interface or the serial interface) in the system, is activated again by the processor; it becomes an outgoing pin to enforce a reset state in other sister units of the system. On deactivation of the reset that succeeds the processor activation, a program executes from a start-up address.

Reset can be activated by an external reset circuit that activates on power-up, on switching-on reset of the system or on detection of a low voltage (e.g. <4.5 V when what is required is 5 V on the system

supply rails). This circuit output connects to a pin called the reset pin of the processor. This circuit may be a simple RC circuit, an external IC circuit or a custom-built IC. Examples of ICs are MAX 6314 and Motorola MC 34064.

Alternatively, it can also be activated by any one of the following: (i) software instruction; (ii) time-out by a programmed timer known as a watchdog timer (or on an internal signal called COP in 68HC11 and 68HC12 families); (iii) a clock monitor detecting a slowdown below certain frequencies.

The watchdog timer is a timing device that resets the system after a predefined timeout. It is activated within the first few clock cycles after power-up. It has a number of applications. In many embedded systems reset by a watchdog timer is very essential because it helps in rescuing the system if a fault develops and the program gets stuck. On restart, the system can function normally. Most microcontrollers have on-chip watchdog timers. The watchdog timer device is described in detail in Section 3.7.

Consider a system controlling temperature. Assume that when the program starts executing, the sensor inputs work all right. However, before the desired temperature is achieved, the sensor circuit develops some fault. The controller will continue delivering the current nonstop if the system is not reset. Consider another example of a system for controlling a robot. Assume that the interfacing motor control circuit in the robot arm develops a fault during the run. In such cases, the robot arm may continue to move unless there is a watchdog timer control. Otherwise, the robot will break its own arm!

When a program executes the program counter increments or changes. An important circuit that associates a system is its reset circuit that can change the program counter to a power-up default value. A program that is reset and runs on a power-up can be one of the following: (i) A system program that executes from the beginning. (ii) A system boot-up program. (iii) A system initialization program.

The watchdog timer reset is a required feature in control applications.

1.3.5 Memory

In a system, there are various types of memory. Figure 1.4 shows a chart for various forms of memory that are present in systems. These are as follows:

1. Internal RAM of 256 or 512 bytes in a microcontroller for registers, temporary data and stack.
2. Internal ROM/PROM/E²PROM for about 4 kB to 64 kB of program (in the case of microcontrollers).
3. External RAM for the temporary data and stack (in most systems) or internal caches (in the case of certain microprocessors).

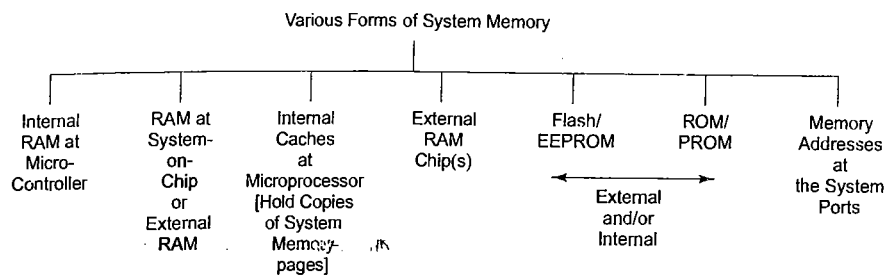


Fig. 1.4 The various forms of memories in the system

4. Internal flash (in many systems the results of processing can be saved in nonvolatile memory: for example, system status periodically and images, songs, or speeches after suitable format compression).
5. Memory stick (or card): video, images, songs, or speeches and large storage in digital camera and mobile systems. Sony memory stick Micro (M2) is of size 15×12.5×1.2 mm and has a flash memory of 2 GB. It has a data transfer rate of 160 Mbps (mega bit per second) and PRO-HG 480 Mbps and 120 Mbps write [since Dec. 2006.]
6. External ROM or PROM for embedding software (in almost all systems other than microcontroller-based systems).
7. RAM memory buffers at ports.
8. Caches (in pipelined and superscalar microprocessors).

Table 1.1 details the functions assigned in embedded systems to the memories. ROM or PROM or EPROM embeds the software specific to the system.

Table 1.1 Functions assigned to the memories in a system

Memory Needed	Functions
ROM or EPROM or flash	Storing application programs from where the processor fetches the instruction codes. Storing codes for system booting, initializing, initial input data and strings. Codes for RTOS. Pointers (addresses) of various interrupt service routines (ISRs).
RAM (internal and external) and RAM for buffer	Storing the variables during program run and storing the stack. Storing input or output buffers, for example, for speech or image.
Memory stick	A flash memory stick is inserted in mobile computing system or digital-camera. It stores high definition video, images, songs, or speeches after a suitable format compression and stores large persistent data.
EEPROM or Flash	Storing nonvolatile results of processing.
Cache	Storing copies of instructions and data in advance from external primary memory and storing the results temporarily during processing.

A system embeds (locates) the following either in the internal flash or ROM, PROM or in an external flash or ROM or PROM of the microcontroller: boot-up program, initialization data, strings or pictogram for screen-display or initial state of the system, programs for various tasks, ISRs and operating system kernel. The system has RAMs for saving temporary data, stack and buffers that are needed during a program run. The system uses flash for storing nonvolatile results.

1.3.6 Input, Output and IO Ports, IO Buses and IO Interfaces

The system gets inputs from physical devices through the input ports. Examples are as follows:

1. A system gets inputs from the touch screen, keys in a keypad or keyboard, sensors and transducer circuits.
2. A controller circuit in a system gets inputs from the sensor and transducer circuits.
3. A receiver of signals or a network card gets the input from a communication system. [A communication system could be a fax or modem, or a broadcasting service.]
4. Ports receives inputs from a network or peripheral.

Consider the system in an Automatic Chocolate Vending Machine. It gets inputs from a port that collects the coins that a child inserts.

Consider the system in a mobile phone. A user inputs the mobile number through the buttons, directly or indirectly (through recall of the number from its memory). Keypad keys connect to the system through an input port.

A processor identifies each input port by its memory buffer addresses, called port addresses. Just as a memory location holding a byte or word is identified by an address, each input port is also identified by the address. The system gets the inputs by the read operations at the port addresses.

The system has output ports through which it sends output bytes to the real world. Examples are as follows:

1. Output may be sent to an light emitting diode (LED), liquid crystal display (LCD) or touch screen display panel. For example, a calculator or mobile phone system sends the output-numbers or an SMS message to the LCD display.
2. A system may send the output to a printer.
3. Output may be sent to a communication system or network.
4. A control system sends the outputs to alarms, actuators, furnaces or boilers.
5. A robot is sent output for its various motors.

Each output port is identified by its memory-buffer addresses (called port addresses). The system sends the output by a write operation to the port address.

There are also general-purpose ports for both the input and output (IO) operations. For example, a mobile phone system sends output as well as gets input through a wireless communication channel. A mobile computing system touch screen system sends output as well as gets input when a user touches the menu displayed or key on the screen.

Each IO port is also identified by an address to which the read and write operations both take place.

Ports can have serial or parallel communication with the system address and data buses. In serial communication a one-bit data line is used and bits are sent serially in successive time slots. Universal Asynchronous Receiver and Transmitter (UART) is a popular communication protocol for serial communication. In parallel communication, several data lines are used and bits are sent in parallel.

A system port may have to send output to multiple channels. A demultiplexer or multiplexer circuit is then used.

A demultiplexer is a digital circuit that sends digital outputs at any instance to one of the provided channels. The channel to which the output is sent is the one that is addressed by the channel address bits at the demultiplexer input. A demultiplexer takes the input and transfers it to a select channel output among the multiple output channels.

A multiplexer is a digital circuit that receives digital inputs at any instance from multiple channels, and sends data output only from a specific channel at an instance. The channel address bits are at multiplexer input. A multiplexer takes the input from one among the multiple input channels and transfers a selected channel input to the output.

A system unit (for example, memory unit or IO port or device) may have to be selected from among the multiple units in the system and activated. A decoder circuit when used as an address decoder decodes the input addresses and activates the selected output channel from among the many outputs. For example, there are 8 units of which one has to be selected. An address-select input of 3 bits is input to the decoder. Based on the input address, the output select line among the 8 activates. If the input address bit is 000, then the 0th output is active and the 0th unit activates. If the input address bit is 111, then the 7th output is active and the 7th unit activates.

Bus A system might have to be connected to a number of other devices and systems. A bus consists of a common set of lines to connect multiple devices, hardware units and systems for communication between any

two of these at any given instance. A bus communication protocol specifies how signals communicate on the bus. A bus may be a serial or parallel bus that transfers one or multiple data bits at an instance, respectively. The protocol also specifies the following: (i) ways of arbitration when several devices need to communicate through the bus; (ii) ways of polling bus requirement from each device at an instance; (iii) ways of daisy chaining the devices so that bus is granted to a device according to the device-priority in the chain.

For networking the distributed units or systems, there are different types of serial and parallel bus protocols: I²C, CAN, USB, ISA, EISA and PCI. For wireless networking of systems there are 802.11, IrDA, Bluetooth and ZigBee protocols.

Chapter 3 will describe the ports, devices, buses and protocols in detail.

A system connects to external physical devices and systems through parallel or serial I/O ports. Demultiplexers and multiplexers facilitate communication of signals from multiple channels through a common path. A system often networks to the other devices and systems through an I/O bus: for example, I²C, CAN, USB, ISA, EISA and PCI bus.

1.3.7 DAC Using a PWM and an ADC

DAC is a circuit that converts digital 8 or 10 or 12 bits to the analog output. The analog output is with respect to the reference voltage. When all input bits are equal to 1, then the analog output is the difference between the positive and negative reference pin voltages; when all input bits equal 0, then the analog output equals -ve reference pin voltage (usually 0 V).

Suppose a system needs to give the analog output of a control circuit for automation. The analog output may be to a power system for d.c. motor or furnace.

A pulse width modulator (PWM) with an integrator circuit is used for the DAC. A PWM unit in the microcontroller operates as follows: Pulse width is made proportional to the analog-output needed. PWM inputs are from 00000000 to 11111111 for an 8-bit DAC operation. The PWM unit outputs to an external integrator, which provides the desired analog output. From this information, the formula to obtain the analog output from the bits in a given PWM register with bits ranging from 00000000 to 11111111 is as follows: Analog output $V = K \cdot pw$, where K is constant and pw is the pulse width.

Suppose a circuit (external to the microcontroller) gives an output of 1.024 V when the pulse width is 50% of the total pulse time period, and 2.047 V when the width is 100%. When the width is made 25%, by reducing by half the value in the PWM output control-register, the integrator output will become 0.512 V. The constant K depends on integrator amplifier gain.

Assume that the integrator operates with a dual (plus-minus) supply. The PWM unit in the microcontroller operates by another method, which is as follows. Assume that when an integrator circuit gives an output of 1.023 V, the pulse width is 100% of the total pulse time period and of -1.024 V when the width is 0%. When the width is made 25% by reducing by half the value in an output control register, the integrator output will be 0.512 V; at 50% the output will be 0.0 V. From this information, the formula to obtain the analog output from the bits in a given PWM register ranging from 00000000 to 11111111 in both situations is as follows: Analog output $V = 0.01 \cdot K' \cdot (pw - 50)$, where K' is constant and pw is pulse width time in percentage with respect to pulse time period. K' depends on integrator amplifier gain.

Analog to Digital Converter ADC is a circuit that converts the analog input to digital 4, 8, 10 or 12 bits. The analog input is applied between the positive and negative pins and is converted with respect to the reference voltage. When input is equal to difference of reference positive and negative voltages, then all output bits equal 1; when equals negative reference voltage (usually 0 V), then all output bits equal 0.

The ADC in the system microcontroller can be used in many applications such as data acquisition systems (DAS), digital cameras, analog control systems and voice digitizing systems. Suppose a system gets the analog inputs from sensors of temperature, pressure, heart-beats and other sources in a DAS. Suppose a system gets the analog inputs from a digital camera. It has CCD (Charge Couple Device) which has tiny pixels that charge up on exposure to light. The charging of each pixel depends upon the light intensity at that point in the image. The analog inputs to the system generate from each pixel. Each pixel's analog input has to be converted into bits to enable processing in the next stage.

Suppose a system needs to read an analog input from a sensor or transducer circuit. If converted to bits by the ADC unit in the system, then these bits, after processing, can also give an output. This provides a control for automation by a combined use of ADC and DAC features.

The converted bits can be given to the port meant for digital display. The bits may be transferred to a memory address, a serial port or a parallel port.

A processor may process the converted bits and generate a Pulse Code Modulated (PCM) output. PCM signals are used to digitize voice into a digital format.

Important points about the ADC are as follows.

1. Either a single or dual analog reference voltage-source is required in the ADC. It sets either the analog input's upper limit or the lower and upper limits both. For a single reference source, the lower limit is set to 0 V (ground potential). When the analog input equals the lower limit, the ADC generates all bits as 0s, and when it equals the upper limit it generates all bits as 1s. [As an example, suppose in an ADC the upper limit or reference voltage is set to 2.255 V. Therefore, the resolution will be 2/256 V. If the 8-bit ADC analog-input is 0.255 V, the converted 8 bits will be 00000000. When the input is $0.255 \text{ V} + 1.000 \text{ V} = 1.255 \text{ V}$, the bits will be 10000000. When the analog input is $0.255 \text{ V} + 0.50 \text{ V}$, the converted bits will be 01000000. From this information, finding a formula to obtain converted bits for a given analog input = v volt is as follows: Binary number n bits after conversion in an n -bit ADC corresponds to decimal number N . Then $N = v \cdot (V_{\text{ref}+} - V_{\text{ref}-})/2^n$. Here, $V_{\text{ref}+}$ is the reference voltage that gives all the bits that are equal to 1 and $V_{\text{ref}-}$ is the reference voltage that gives all the bits that are equal to 0.]
2. An ADC may be of 8, 10, 12, or 16 bits depending upon the resolution needed for conversion.
3. The start of the conversion (STC) signal or input initiates the conversion to 8 bits. In a system, an instruction or a timer signals the STC.
4. There is an end of conversion (EOC) signal. A flag in a register is set to indicate the end of conversion and the ADC generates an interrupt; the ISR reads the ADC bits and saves them in the memory buffer.
5. There is a conversion time limit in which the conversion is definite.
6. A Sample and Hold (S/H) unit is used to sample the input for a fixed time and hold till conversion is over.

An ADC unit can be repeatedly used after the intervals equal to the conversion time. Therefore, one can digitizes the DAS sensor signals, CCD signals, voice, music or video signals, or heart beat sensor signals in different systems. An ADC unit in an embedded system microcontroller may have multichannels. It can then take the inputs in succession from the various pins interconnected to different analog sources.

For automatic control and signal processing applications, a system provides necessary interfacing circuit and software for the Digital to Analog Conversion (DAC) unit and Analog to Digital Conversion (ADC) unit. A DAC operation is done with the help of a combination of a PWM unit in the microcontroller and an external integrator chip. ADC operations are required for data acquisition, image processing, voice processing, video processing, instrumentation and automatic control systems.

1.3.8 LCD, LED and Touchscreen Displays

A system requires an interfacing circuit and software to display the status or message for a line, for multiline displays, or for flashing displays. An LCD screen may show up a multiline display of characters or also show a small graph or icon (called a pictogram). A recent innovation in the mobile phone system turns the screen blue to indicate an incoming call. Third generation system phones have both image and graphic displays. An LCD needs little power. A supply or battery (a solar panel in the calculator) powers it. The LCD is a diode that absorbs or emits light and 3 to 4 V and 50 or 60 Hz voltage-pulses with currents less than $\sim 50 \mu\text{A}$ are required. The pulses are applied with the same polarity on the crystal front and back plane for no light, and with opposite polarity for light. Here, polarity means logic '1' or '0'. A display-controller is often used in case of matrix displays.

To indicate the ON status of the system, there may be an LED that glows. A flashing LED may indicate that a specific task is under completion or is running or in wait status. The LED is a diode that emits yellow, green, red or infrared light in a remote controller on application of a forward voltage of between 1.6–2 V. It needs current up to 12 mA above 5 mA (less in flashing display mode). It is much brighter than the LCD, making it suitable for flashing displays and for displays limited to a few digits.

A touchscreen is an input as well as an output device, which can be used to enter a command, a chosen menu or to give a reply. The information is input by physically touching at a screen position using a finger or a stylus. A stylus is thin pencil-shaped object. It is held between the fingers and used just as a pen. The screen displays the choices or commands, menus, dialog boxes and icons. The display-screen display is similar to a computer video display unit screen. Newer touch screen senses the fingers even from proximity, for example, in Apple iPhone.

Sections 3.3.4 and 3.3.5 describe the LCD and touchscreen devices and their connections to the system.

The system may need the necessary interfacing circuit and software for the output to the LCD display controller and the LED interfacing ports or for the I/Os with the touchscreen.

1.3.9 Keypad/Keyboard

The keypad or keyboard is an important device for getting user inputs. The system provides the necessary interfacing and key-debouncing circuit as well as the software for the system to receive input from a set of keys, from a keyboard, keypad or virtual keypad. A touchscreen provides for a virtual keypad in a mobile computing system. A virtual keypad is a keypad displayed on the touch screen where the user can enter the keys using a stylus or finger.

A keypad has upto a maximum of 32 keys. A keyboard may have 104 keys or more. The keypad or keyboard may interface serially or parallelly to the processor directly through ports or through a controller. Mobile phones may have a T9 keypad. A T9 keypad has 16 keys and four up-down right-left menu keys. Using 0 to 9 keys text messages, such as SMS messages, are generated.

For inputs, a keypad or board may interface to a system. The system provides necessary interfacing circuit and software to receive inputs directly from the keys or through a controller.

1.3.10 Pulse Dialer, Modem and Transceiver

For user connectivity through the telephone line, wireless or a network, a system provides the necessary interfacing and circuits. It also provides the software for pulse dialing through the telephone line, for modem

interconnection for fax, for Internet packets routing and for transmitting and connecting to a wireless cellular system or personal area wireless network. A *transceiver* is a circuit that can transmit as well as receive byte streams.

In communication system, a pulse dialer, modem or transceiver is used. A system provides the necessary interfacing circuit and software for dialing and for the modem and transceiver, directly or through a controller.

1.3.11 Interrupt Handler

A timing device sends a time-out interrupt when a preset time elapses or sends a compare interrupt when the present-time equals the preset time. Assume that data have to be transferred from a keyboard to a printer. A port peripheral generates an interrupt on receiving the input data or when the transmitting buffer becomes empty. Each action generates an interrupt. A system may possess a number of devices and the system processor has to control and handle the requirements of each device by running an appropriate ISR (interrupt service routine) for each. *An interrupts-handling mechanism must exist in each system to handle interrupts from various processes and for handling multiple interrupts simultaneously pending for service.* Chapter 4 describes in detail the interrupts, ISRs, and their handling mechanisms in a system. Important points regarding the interrupts and their handling by the program are as follows.

1. There can be a number of interrupt sources and groups of interrupt sources in a processor. [Section 4.3] An interrupt may be a hardware signal that indicates the occurrence of an event. [For example, a real-time clock continuously updates a value at a specified memory address; the transition of that value is an event that causes an interrupt.] An interrupt may also occur through timers, through an interrupting instruction of the processor program or through an error during processing. The error may arise due to an illegal op-code fetch, a division by zero result or an overflow or underflow during an ALU operation. An interrupt can also arise through a software timer. A software interrupt may arise in an exceptional condition that may have developed while running a program.
2. The system may prioritize sources and service them accordingly. [Section 4.5.]
3. Certain sources are not maskable and cannot be disabled. Some are assigned the highest priority during processing.
4. The processor's current program has to divert to a service routine to complete that task on the occurrence of the interrupt. For example, if a key is pressed, then an ISR reads the key and stores the key value in the processor memory address. If a sequence of keys is pressed, for instance in a mobile phone, then an ISR reads the keys and also calls a task to dial the mobile number.
5. There is a programmable unit on-chip for the interrupt handling mechanism in a microcontroller.
6. The operating system is expected to control the handling of interrupts and running of routines for the interrupts in a particular application. The system always gives priority to the ISRs over the tasks of an application.

A system provides an interrupt handling mechanism for executing the ISRs in case of the interrupts from physical devices, systems, software instructions and software exceptions.

1.4 EMBEDDED SOFTWARE IN A SYSTEM

The software is like the brain of the embedded system.

1.4.1 Final Machine Implementable Software for a System

An embedded system processor executes software that is specific to a given application of that system. The instruction codes and data in the final phase are placed in the ROM or flash memory for all the tasks that are executed when the system runs. The software is also called ROM image. Why? Just as an image is a unique sequence and arrangement of pixels, embedded software is also a unique placement and arrangement of bytes for instructions and data.

Each code or datum is available only in the bits and bytes format. The system requires bytes at each ROM address, according to the tasks being executed. A *machine implementable software file is therefore like a table having in each rows the address and bytes.* The bytes are saved at each address of the system memory. The table has to be readied as a ROM image for the targeted hardware. Figure 1.5 shows the ROM image in a system memory. The image consists of the boot up program, stacks address pointers, program counter address pointers, application programs, ISRs, RTOS, input data and vector addresses.

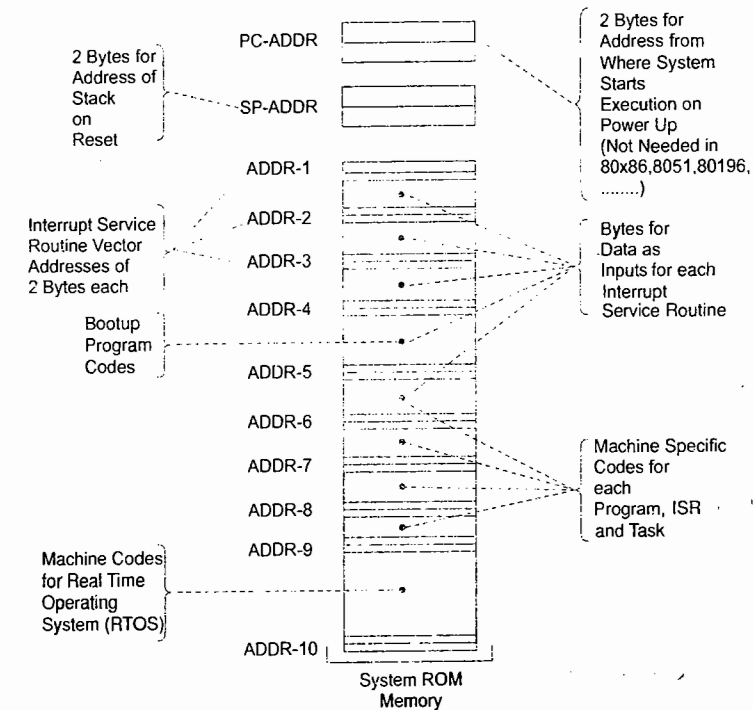


Fig. 1.5 System ROM memory embedding the software, RTOS, data and vector addresses

Final stage software is also called the ROM image. The final machine implementable software for a product embeds in the once programmable flash or ROM (or PROM) as an image in a frame. Bytes at each address must be defined to create the ROM image. By changing this image, the same hardware platform will work differently and can be used for entirely different applications or for new upgrades of the same system.

1.4.2 Coding of Software in Machine Codes

During coding in this format, the programmer defines the addresses and the corresponding bytes or bits at each address. In configuring some specific physical device or subsystem, machine code-based coding is used. For example, in a transceiver, placing certain machine code and bits can configure it to transmit at specific megabytes per second or gigabytes per second, using specific bus and networking protocols. Another example is using certain codes for configuring a control register with the processor. During a specific code-section processing, the register can be configured to enable or disable use of its internal cache. However, coding in machine implementable codes is done only in specific situations because it is time consuming and the programmer must first have to understand the processor instructions set and then memorize the instructions and their machine codes.

1.4.3 Software in Processor Specific Assembly Language

A program or a small specific part can be coded in *assembly language* using an assembler after understanding the processor and its instruction set. Assembler is software used for developing codes in *assembly*.

Assembly language coding is extremely useful for configuring physical devices like ports, a line-display interface, ADC and DAC and reading into or transmitting from a buffer. These codes are also called low-level codes for the device driver functions. [Sections 1.4.7 and 4.2.4.] They are useful to run the processor or device-specific features and provide an optimal coding solution.

Lack of knowledge of writing device driver codes or codes that utilize the processor-specific features-invoking codes in an embedded system design team can cost a lot. A vendor may charge for the APIs and also charge intellectual property fees for each system shipped out of the company.

To make all the codes in *assembly language* may, however, be very time consuming. Full coding in assembly may be done only for a few simple, small-scale systems, such as toys, automatic chocolate vending machines, robots or data acquisition systems.

Figure 1.6 shows the process of converting an *assembly language program* into machine implementable software file and then finally obtaining a ROM image file.

1. An *assembler* translates the assembly software into the machine codes using a step called *assembling*.
2. In the next step, called *linking*, a *linker* links these codes with the other codes required. Linking is necessary because of the number of codes to be linked for the final binary file. For example, there are the standard codes to program a delay task for which there is a reference in the assembly language program. The codes for the delay must link with the assembled codes. The delay code is sequential from a certain beginning address. The assembly software code is also sequential from a certain beginning address. Both the codes have to be linked at the distinct addresses as well as at the available addresses in the system. The linked file in binary for *run* on a computer is commonly known as an executable file or simply an '.exe' file. After linking, there has to be reallocation of the sequences of placing the codes before actually placing the codes in memory.

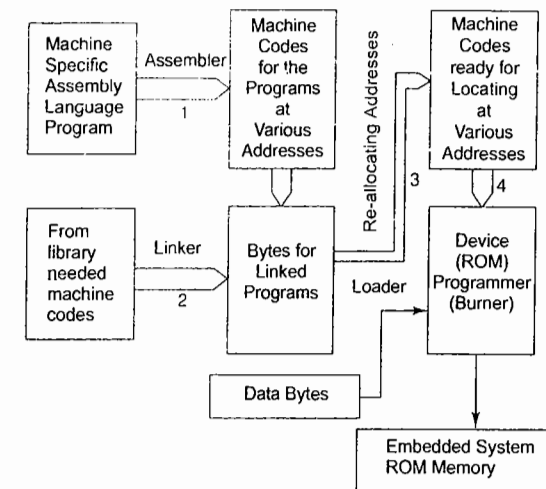


Fig. 1.6 The process of converting an assembly language program into the machine codes and finally obtaining the ROM image

3. In the next step, the *loader* program performs the task of *reallocating* the codes after finding the physical memory addresses available at a given instant. The loader is a part of the operating system and places codes into the memory after reading the '.exe' file. This step is necessary because the available memory addresses may not start from 0x0000, and binary codes have to be loaded at different addresses during the run. The loader finds the appropriate start address. In a computer, after the loader loads into a section of RAM, the program is ready to run.
4. The final step of the system design process is *locating* these codes as a ROM image. The codes are permanently placed at the addresses actually available in the ROM. In embedded systems, there is no separate program to keep track of the available addresses at different times during the run, as in a computer. In embedded systems, therefore, the next step instead of loader after linking is the use of a *locator*, which locates the IO tasks and hardware device driver codes at fixed addresses. Port and device addresses are fixed for a given system as per the interfacing circuit between the system buses and ports or devices. The *locator* program reallocates the linked file and creates a file for a permanent location of the codes in a standard format. The file format may be in the Intel Hex file format or Motorola S-record format. The designer has to define the available addresses to locate and create files to permanently locate the codes.
5. Lastly, either (i) a laboratory system, called *device programmer*, takes as input the ROM image file and finally *burns* the image into the PROM or flash or (ii) at a foundry, a mask is created for the ROM of the embedded system from the ROM image file. [The process of placing the codes in PROM or flash is also called burning.] The mask created from the image gives the ROM in IC chip form.

To configure some specific physical device or subsystem such as the transceiver, machine codes can be used straightaway. For physical device driver codes or codes that utilize processor-specific features-invoking codes, 'processor-specific' assembly language is used. A file is then created in three steps using an Assembler, Linker and Locator. The file has the ROM image in a standard format. A device programmer finally burns the image in the PROM or EPROM. A mask created from the image gives the ROM in IC chip form.

1.4.4 Software in High Level Language

Since the coding in *assembly language* is very time consuming in most cases, software is developed in a high-level language, 'C' or 'C++' or visual C++ or 'Java' in most cases. 'C' is usually the preferred language. The programmer needs to understand only the hardware organization when coding in high level language. As an example, consider the following problem.

Example 1.1

Add 127, 29 and 40 and print the square root.

An exemplary C language program for all the processors is as follows. (i) `#include <stdio.h>`
 (ii) `#include <math.h>` (iii) `void main (void) {` (iv) `int i1, i2, i3, a; float result;` (v) `i1 = 127;`
`i2 = 29; i3 = 40; a = i1 + i2 + i3; result = sqrt (a);` (vi) `printf (result);`}

The coding for square root will need many lines of code and can be done only by an expert assembly language programmer. To write the program in a high level language is very simple compared to writing it in assembly language. 'C' programs have a feature that adds the assembly instructions when using certain processor-specific features and coding for a specific section, for example, a port device driver. Figure 1.7 shows the different programming layers in a typical embedded 'C' software. These layers are as follows. (i) Processor Commands. (ii) Main Function. (iii) Interrupt Service Routine. (iv) Multiple tasks, say, 1 to N. (v) Kernel and Scheduler. (vi) Standard library functions, protocol handling and stack functions.

Figure 1.8 shows the process of converting a C program into the ROM image file. A compiler generates the object codes. It assembles the codes according to the processor instruction set and other specifications. The C compiler for embedded systems must, as a final step of compilation, use a code-optimizer that optimizes the codes before linking. After compilation, the linker links the object codes with other needed codes. For example, the linker includes the codes for the functions *printf* and *sqrt* codes. Codes for device and driver (device control codes) management also link at this stage: for example, printer device management and driver codes. After linking, the other steps for creating a file for ROM image are the same as shown earlier in Figure 1.6.

C, C++, Java, Visual C++ are the languages used for software development. A C program has various layers: processor commands, main function, task and library functions, interrupt service routines and kernel (scheduler). The compiler generates an object file. Using a linker and locator, the file for the ROM image is created for the targeted hardware.

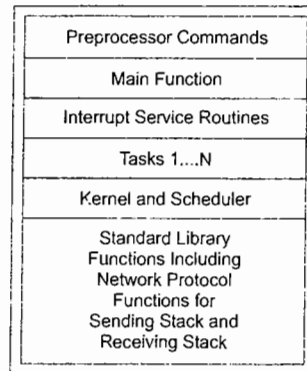


Fig. 1.7 The different program layers in the embedded software in C

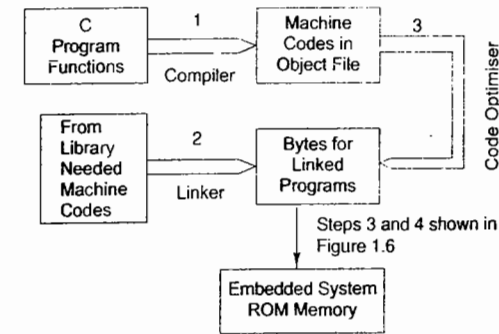


Fig. 1.8 The process of converting a C program into the file for ROM image

1.4.5 Program Models for Software Designing

The program design task is simplified if a program is modeled.

The different models that are employed during the design processes of the embedded software are as follows:

1. Sequential Program Model
2. Object Oriented Program Model
3. Control and Data flow graph or Synchronous Data Flow (SDF) Graph or Multi Thread Graph (MTG) Model
4. Finite State Machine for data path
5. Multithreaded Model for concurrent processing of processes or threads or tasks

UML (Universal Modeling language) is a modeling language for object oriented programming.

These models are explained Chapter 6.

1.4.6 Software for Concurrent Processing and Scheduling of Multiple Tasks and ISRs Using an RTOS

An embedded system program is most often designed using multiple processes or multitasks or a multithreads. [Refer to Sections 7.1 to 7.3 for definitions and understanding of the processes, threads and tasks.] The multiple tasks are processed most often by the OS not sequentially but concurrently. Concurrent processing tasks can be interrupted for running the ISRs, and a higher priority task preempts the running of lower priority tasks.

An OS provides for process, memory, devices, IOs and file system management. A file system specifies the ways in which a file is created, called, named, used, copied, saved or deleted, opened and closed. File system is the software for using the files on a disk, flash memory, memory card or memory stick.

OS software have scheduling functions for all the processes (tasks, ISRs and device drivers) in the system. Since the running of the tasks and ISRs may have real time constraints and deadlines for finishing the tasks, an RTOS is required in an embedded system. The RTOS provides the OS functions for coding the system, provides interprocess communication functions and controls the passing of messages and signals to a task.

RTOS functions are highly complex. There are a number of popular and readily available RTOSs.

Chapters 8 to 12 describes the RTOS functions and examples of applications in the embedded systems.

RTOS is used in most embedded systems and the system does concurrent processing of multiple tasks when the tasks have real time constraints and deadlines.

1.4.7 Software for Device Drivers and Device Management in an Operating System

An embedded system is designed to perform multiple functions and has to control multiple physical and virtual devices. In an embedded system, there may be number of *physical devices*. Exemplary physical devices are timers, keyboards, display, flash memory, parallel ports and network cards.

A program is also developed using the concept of *virtual devices*. Examples of virtual devices are as follows.

1. A file (of records opened, read, written and closed, and saved as a stream of bytes or words)
2. A pipe (for sending and receiving a stream of bytes from a source to destination)
3. A socket (for sending and receiving a stream of bytes between the client and server software and between source and destination computing systems)
4. A RAM disk (for using the RAM in a way similar to files on the disk)

A file is a data structure (or virtual device) which sends the records (characters or words) to a data sink (for example, a program function) and which stores the data from the data source (for example, a program function). A file in a computer may also be stored in the hard disk and in flash memory in embedded system.

The term virtual device follows from the analogy that just as a keyboard gives an input to the processor for a *read*, a file also gives an input to the processor. The processor gives an output to a printer for a *write*. Similarly, the processor writes an output to the file.

A device for the purpose of control, handling, reading and writing actions can be taken as consisting of three components. (i) A control register or word that stores the bits that, on setting or resetting by a device driver, control device actions. (ii) A status register or word that provides the flags (bits) to show the device status to the device driver. (iii) A device mechanism that controls the device actions. There may be input and output data buffers in a device, which may be written or read by a device driver. Device driver actions are to get input into or send output from the control registers, input data buffers, output data buffers and status registers of the device.

A device driver is software for opening, connecting or binding, reading, writing and closing or controlling actions of the device. It is software written in a high level language. It controls functions for device open (configure), connect, bind, listen, read or write or close. The device driver executes after the programming of the control register (or word) of a peripheral or virtual device. The programming is called device initialisation or registration or attachment. The driver reads the status register, gets the inputs and writes the outputs. It executes on an interrupt to or from the device.

A *driver* controls three functions. (i) Initializing, which is activated by placing appropriate bits at the control register or word. (ii) Calling an ISR on interrupt or on setting a status flag in the status register and running (driving) the ISR (Interrupt Handler Routine). (iii) Resetting the status flag after an interrupt service. A driver may be designed for asynchronous operations (multiple use by tasks one after another) or synchronous operations (concurrent use by the tasks).

Using the functions of the OS, a device driver coding can be made such that the underlying hardware is hidden as much as possible. An API then defines the hardware separately. This makes the driver usable when the device hardware changes in a system.

A device driver accesses a parallel or serial port, keyboard, mice, disk, network, display, file, pipe and socket at specific addresses. An OS also provides device driver codes for system-port addresses and for hardware access mechanisms.

A device manager software provide codes for detecting the presence of devices, for initializing these and for testing the devices that are present. The manager includes software for allocating and registering port (in fact, it may be a register or memory) addresses for the various devices at distinctly different addresses, including codes for detecting any collision between these, if any. It ensures that any device accesses to one task only at any given instant. It takes into account that virtual devices may also have addresses that are allocated by the manager.

An OS also provides and executes modules for managing devices that associate with an embedded system. The underlying principle is that at an instant, only one physical or virtual device should get access to or from one task only.

Sections 4.2.4 and 8.6.1 will describe device drivers and device management in detail. The OS also provides and manages virtual devices such as pipes and sockets. Sections 7.14 and 7.15 describe these in detail.

For designing embedded-software, two types of devices are considered: physical and virtual. Physical devices include keypad, printer or display unit. A virtual device could be a file or pipe or socket or RAM disk. Device drivers and device manager software are needed in the system. The RTOS includes device-drivers and a device manager to control and facilitates the use of the number of physical and virtual devices in the system.

1.4.8 Software Tools for Designing an Embedded System

Table 1.2 lists the applications of software tools for assembly language programming, high level language programming, RTOS, debugging and system integration.

Table 1.2 Software modules and tools for designing of an embedded system

Software Tools	Application
Editor	For writing C codes or assembly mnemonics using the keyboard of the PC for entering the program. Allows the entry, addition, deletion, insert, appending previously written lines or files, merging record and files at the specific positions. Creates a source file that stores the edited file. It also has an appropriate name [provided by the programmer].
Interpreter	For expression-by-expression (line-by-line) translation to machine-executable codes.
Compiler	Uses the complete set of codes. It may also include codes, functions and expressions from the library routines. It creates a file called object file.
Assembler	For translating assembly mnemonics into binary opcodes (instructions), that is, into an executable file called binary file and for making a list file that can be printed. The list file has address, source code (assembly language mnemonics) and hexadecimal object codes. The file has addresses that reallocate during the actual run of the assembly language program.

(Contd)

Software Tools	Application
Cross assembler	For converting object codes or executable codes for a processor to other codes for another processor and vice versa. The cross-assembler assembles the assembly codes of the target processor as the assembly codes of the processor of the PC used in system development. Later, it provides the object codes for the target processor. These codes will be the ones actually needed in the final developed system.
Simulator	To simulate all functions of an embedded system circuit including that of additional memory and peripherals. It is independent of a particular target system. It also simulates the processes that will execute when the codes of a particular processor execute.
Source-code engineering software	For source code comprehension, navigation and browsing, editing, debugging, configuring (disabling and enabling the C++ features) and compiling.
RTOS	Refer Chapters 8 to 10.
Stethoscope	For dynamically tracking the changes in any program variable or parameter. It demonstrates the sequence of multiple processes (tasks, threads, service routines) that execute and also records the entire time history.
Trace scope	To help in tracing the changes in modules and tasks with time on the X-axis. A list of actions also produces the desired time scales and the time expected to be taken for different tasks.
Integrated development environment	This is a development software and hardware environment that consists of simulators with editors, compilers, assemblers, RTOS, debuggers, stethoscope, tracer, emulators, logic analyzers, and application code burners in PROM or flash.
Prototyper	This simulates and does source code engineering including compiling, debugging and browsing and summarizing the complete status of the final target system during the development phase.
Locator [#]	This uses a cross-assembler output and a memory allocation map and provides the locator program output as a hex-file. It is the final step of the software design process or an embedded system.

[#] The locator program output is in the Intel hex file or Motorola S-record format.

Software tools are used to develop software for designing an embedded system. Debugging tools, such as a stethoscope, trace scope, and sophisticated tools such as an integrated development environment and prototype development tools, are needed for the integrated development of system software and hardware.

1.4.9 Software Tools Required in Exemplary Cases

Table 1.3 gives the various tools needed to design exemplary systems.

RTOS is essential in most embedded systems to process multiple tasks and ISRs. Embedded systems for medium scale and sophisticated applications need a number of sophisticated software and debugging tools.

Table 1.3 Software tools required in exemplary systems

Software Tools	Automatic Chocolate Vending Machine ^g	Data Acquisition System	Robot	Mobile Phone	Adaptive Cruise Control System with String Stability ^h	Voice Processor
Editor	Yes	Yes	Yes	Yes	Yes	NR
Interpreter	Yes	NR	Yes	NR	NR	NR
Compiler	Yes	Yes	Yes	Yes	Yes	Yes
Assembler	Yes	Yes	Yes	No	No	No
Cross Assembler	NR	Yes	Yes	No	No	No
Locator	Yes	Yes	Yes	Yes	Yes	Yes
Simulator	NR	Yes	Yes	Yes	Yes	Yes
Source code engineering software	NR	NR	NR	Yes	Yes	Yes
RTOS	Yes	MR	Yes	Yes	Yes	Yes
Stethoscope	NR	NR	NR	Yes	Yes	Yes
Trace scope	NR	NR	NR	Yes	Yes	Yes
Integrated development environment	NR	Yes	Yes	Yes	Yes	Yes
Prototyper	NR	No	No	Yes	Yes	Yes

Note: NR means not required. MR means may be required in a specific complex system but not compulsorily needed.

1.5 EXAMPLES OF EMBEDDED SYSTEMS

Embedded systems have very diversified applications. A few select application areas of embedded systems are telecommunications, smart cards, missiles and satellites, computer networking, digital consumer electronics, and automotives. Figure 1.9 shows the applications of embedded systems in these areas.

A few examples of *small scale embedded system* applications are as follows:

1. Point of sales terminals: automatic chocolate vending machine
2. Stepper motor controllers for a robotics system
3. Washing or cooking systems
4. Multitasking toys
5. Microcontroller-based single or multidisplay digital panel meter for voltage, current, resistance and frequency
6. Keyboard controller
7. SD, MMI and network access cards
8. CD drive or hard disk drive controller

9. The peripheral controllers of a computer, for example, a CRT display controller, a keyboard controller, a DRAM controller, a DMA controller, a printer controller, a laser printer controller, a LAN controller, a disk drive controller
10. Fax or photocopy or printer or scanner machine
11. Remote (controller) of TV
12. Telephone with memory, display and other sophisticated features

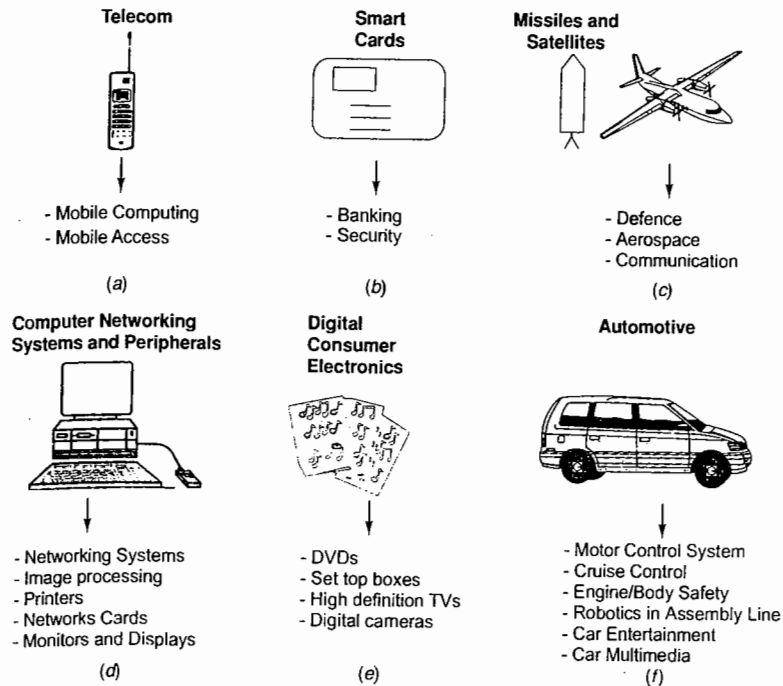


Fig. 1.9 Applications of the embedded systems in various areas

13. Motor controls systems—for example, an accurate control of speed and position of the d.c. motor, robot and CNC machine; automotive applications such as closed loop engine control, dynamic ride control, and an antilock braking system monitor
14. Electronic data acquisition and supervisory control system
15. Electronic instruments, such as an industrial process controller
16. Electronic smart weight display system and an industrial moisture recorder cum controller
17. Digital storage system for a signal wave form or for electric or water meter reading system
18. Spectrum analyzer
19. Biomedical systems such as an ECG LCD display cum recorder, a blood-cell recorder cum analyzer, and a patient monitor system

Some examples of *medium scale embedded systems* are as follows:

20. Computer networking systems, for example, a router, a front-end processor in a server, a switch, a bridge, a hub and a gateway
21. For Internet appliances, there are numerous application systems (i) An intelligent operation, administration and maintenance router (IOAMR) in a distributed network and (ii) Mail client card to store e-mail and personal addresses and to smartly connect to a modem or server
22. Entertainment systems such as a video game and a music system
23. Banking systems, for example, bank ATM and credit card transactions
24. Signal tracking systems, for example, an automatic signal tracker and a target tracker
25. Communication systems such as a mobile communication SIM card, a numeric pager, a cellular phone, a cable TV terminal and a FAX transceiver with or without a graphic accelerator
26. Image filtering, image processing, pattern recognizer, speech processing and video processing
27. Video games
28. A system that connects a pocket PC to the automobile driver mobile phone and a wireless receiver. The system then connects to a remote server for Internet or e-mail or to a remote computer at an ASP (application service provider)
29. A personal information manager using frame buffers in handheld devices
30. Thin client [A thin client provides disk-less nodes with remote boot capability]. Application of thin-client accesses to a data centre from a number of nodes; in an Internet laboratory accesses to the Internet leased line through a remote server.
31. Embedded firewall / router using ARM7/ multiprocessor with two Ethernet interfaces and interfaces support to PPP, TCP/IP and UDP protocols.
Examples of *sophisticated embedded systems* are as follows:
32. Mobile smart phones and computing systems
33. Mobile computer
34. Embedded systems for wireless LAN and for convergent technology devices
35. Embedded systems for video, interactive video, broadband IPv6 (Internet Protocol version 6) Internet and other products, real time video and speech or multimedia processing systems
36. Embedded interface and networking systems using high speed (400 MHz plus), ultra high speed (10 Gbps) and a large bandwidth: Routers, LANs, switches and gateways. SANs (Storage Area Networks), WANs (Wide Area Networks)
37. Security products and high-speed Network security. Gigabit rate encryption rate products

1.6 EMBEDDED SYSTEM-ON-CHIP (SoC) AND USE OF VLSI CIRCUIT DESIGN TECHNOLOGY

Lately, embedded systems are being designed on a single silicon chip, called *System on chip (SoC)*, a *design innovation*. SoC is a system on a VLSI chip that has all the necessary analog as well as digital circuits, processors and software.

A SoC may be embedded with the following components:

1. Embedded processor GPP or ASIP core,
2. Single purpose processing cores or multiple processors,
3. A network bus protocol core,
4. An encryption function unit,

5. Discrete cosine transforms for signal processing applications,
6. Memories,
7. Multiple standard source solutions, called IP (Intellectual Property) cores,
8. Programmable logic device and FPGA (Field Programmable Gate Array) cores,
9. Other logic and analog units.

An exemplary application of such an embedded SoC is the mobile phone. Single purpose processors, ASIPs and IPs on an SoC are configured to process encoding and deciphering, dialing, modulating, demodulating, interfacing the key pad and multiple line LCD matrix displays or touch screen, storing data input and recalling data from memory. Figure 1.10 shows an SoC that integrates internal ASICs, internal processors (ASIPs), shared memories and peripheral interfaces on a common bus. Besides a processor, memories and digital circuits with embedded software for specific applications, the SoC may possess analog circuits as well.

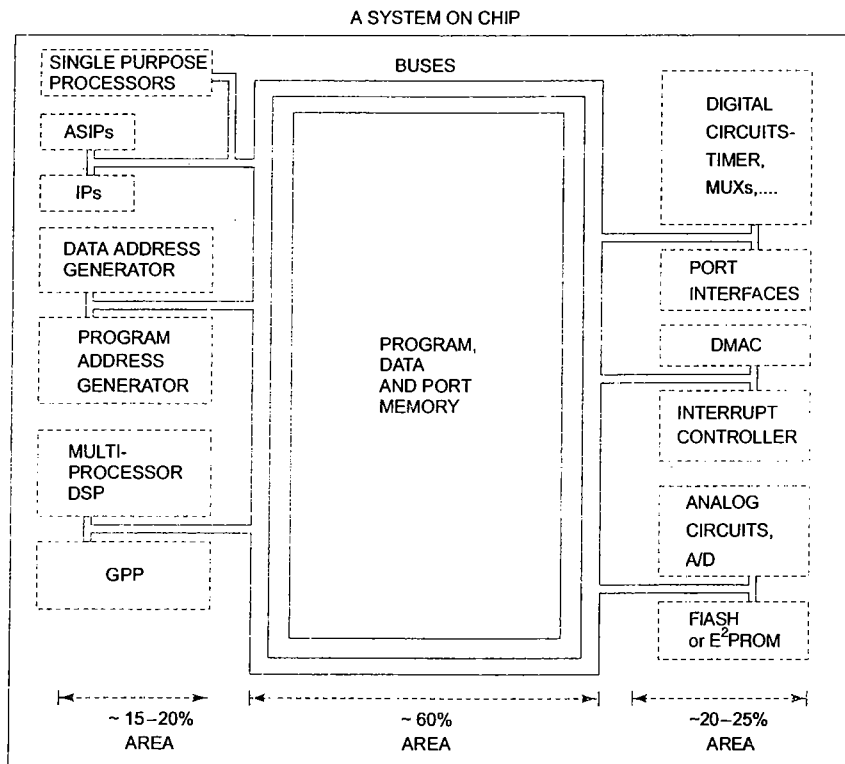


Fig. 1.10 A SoC embedded system and its common bus with internal ASIPs, internal processors, IPs, shared memories and peripheral interfaces

1.6.1 Application Specific IC (ASIC)

ASICs are designed using the VLSI design tools with the processor GPP or ASIP and analog circuits embedded into the design. The designing is done using the Electronic Design Automation (EDA) tool. [For design of an ASIC, a High-level Design Language (HDL) is used].

1.6.2 IP Core

On a VLSI chip, there may be integration of high-level components. These components possess gate-level sophistication in circuits above that of the counter, register, multiplier, floating point operation unit and ALU. A standard source solution for synthesizing a higher-level component by configuring an FPGA core or a core of VLSI circuit may be available as an Intellectual Property, called (IP). The designer or the designing company holds the copyright for the synthesized design of a higher-level component for gate-level implementation of an IP. One might have to pay royalty for every chip shipped. An embedded system may incorporate several IPs.

- An IP may provide hardwired implementable design of a *transform*, an *encryption algorithm* or a *deciphering algorithm*.
- An IP may provide a design for *adaptive filtering* of a signal.
- An IP may provide a design for implementing Hyper Text Transfer Protocol (HTTP) or File Transfer Protocol (FTP) or Bluetooth protocol to transmit a web page or a file on the Internet.
- An IP may be designed for a USB or PCI bus controller. [Sections 3.10.3 and 3.12.2]

1.6.3 FPGA Core with Single or Multiple Processors

Suppose an embedded system is designed with a view to enhancing functionalities in future. An FPGA core is then used in the circuits. It consists of a large number of programmable gates on a VLSI chip. There is a set of gates in each FPGA cell, called macro cell. Each cell has several inputs and outputs. All cells interconnect like an array (matrix). Each interconnection is programmable through the associated RAM in an FPGA programming tool. An FPGA core can be used with a single or multiple processor.

Consider the algorithms for the following: Fourier transform (FT) and its inverse (IFT), DFT or Laplace transform and its inverse, compression or decompression, encrypting or deciphering, specific pattern recognition (for recognizing a signature or finger print or DNA sequence). We can configure an algorithm into the logic gates of FPGA. It gives hardwired implementation for a processing unit. It is specific to the needs of the embedded system. An algorithm of the embedded software can implement in one of the FPGA sections and another algorithm in its other section.

FPGA cores with a single or multiple processor units on chip are used. One example of such core is Xilinx Virtex-II Pro FPGA XC2VP125. XC2VP125 from Xilinx has 125136 logic cells in the FPGA core with four IBM PowerPCs. It has been used as a data security solution with encryption engine and data rate of 1.5 Gbps. Other examples of embedded systems integrated with logic FPGA arrays are DSP-enabled, real-time video processing systems and line echo eliminators for the Public Switched Telecommunication Networks (PSTN) and packet switched networks. [A packet is a unit of a message or a flowing data such that it can follow a programmable route among the number of optional open routes available at an instance.]

1.7 COMPLEX SYSTEMS DESIGN AND PROCESSORS

1.7.1 Embedding a Microprocessor

A General Purpose Processor microprocessor can be embedded on a VLSI chip. Table 1.4 lists different streams of microprocessors embedded in a complex system design.

Table 1.4 Important microprocessors used in embedded systems

Stream	Microprocessor Family	Source	CISC or RISC or Both features
Stream 1	68HCxxx	Motorola	CISC
Stream 2	80x86	Intel	CISC
Stream 3	SPARC	Sun	RISC
Stream 4	ARM	ARM	RISC with CISC functionality

1.7.2 Embedding a Microcontroller

Microcontroller VLSI cores or chips for embedded systems are usually among the five streams of families given in Table 1.5.

Table 1.5 Major microcontrollers[@] used in the embedded systems

Stream	Microcontroller Family	Source	CISC or RISC or Both
Stream 1	68HC11xx, HC12xx, HC16xx	Motorola	CISC
Stream 2	8051, 8051MX	Intel, Philips	CISC
Stream 3	PIC 16F84 or 16C76, 16F876 and PIC18	Microchip	CISC
Stream 4 [*]	Microcontroller Enhancements of CORTEX-M3 ARM9/ARM7 from Philips, Samsung and ST Microelectronics	ARM, Texas, Philips, Samsung and ST Microelectronics etc.	RISC Core with CISC functionality

[@] Other popular microcontrollers are as follows. (i) Hitachi H8x family and SuperH 7xxx. (ii) Mitsubishi 740, 7700, M16C and M32C families. (iii) National Semiconductor COP8 and CR16/16C. (iv) Toshiba TLCS 900S (v) Texas Instruments MSP 430 for low voltage battery based system. (vi) Samsung SAM8. (vii) Ziglog Z80 and eZ80

1.7.3 Embedding a DSP

A digital signal processor (DSP) is a processor core or chip for the applications that process digital signals. [For example, filtering, noise cancellation, echo elimination, compression and encryption applications.] Just as a microprocessor is the most essential unit of a computing system, a DSP is essential unit of an embedded

system in a large number of applications needing processing of signals. Exemplary applications are in image processing, multimedia, audio, video, HDTV, DSP modem and telecommunication processing systems. DSPs also find use in systems for recognizing image pattern or DNA sequence.

DSP as an ASIP is a single chip or core in a VLSI unit. It includes the computational capabilities of a microprocessor and Multiply and Accumulate (MAC) units. A typical MAC has a 16×32 MAC unit.

DSP executes discrete-time, signal-processing instructions. It has Very Large Instruction Word (VLIW) processing capabilities; it processes Single Instruction Multiple Data (SIMD) instructions; it processes Discrete Cosine Transformations (DCT) and inverse DCT (IDCT) functions. The latter are used in algorithms for signal analyzing, coding, filtering, noise cancellation, echo elimination, compressing and decompressing, etc. Major DSPs for embedded systems are from the three streams given in Table 1.6.

Table 1.6 Important digital signal processor[@] used in the embedded systems

Stream	DSP Family	Source
Stream 1	TMS320Cxx, OMAP ¹	Texas
Stream 2	Tiger SHARC	Analog Device
Stream 3	5600xx	Motorola
Stream 4	PNX 1300, 1500 ²	Philips

¹For example, TMS320C62XX a fixed point 200 MHz DSP (Section 2.3.5).

²Media processor, which besides multimedia DSP operations, also does network stream data packet processing.

1.7.4 Embedding an RISC

A RISC microprocessor provides the speedy processing of instructions, each in a single clock-cycle. This facilitates pipelining and superscalar processing. Besides greatly enhanced capabilities mentioned above, there is great enhancement of speed by which an instruction from a set is processed. Thumb[®] instruction set is a new industry standard that also gives a reduced code density in ARM RISC processor. RISCs are used when the system needs to perform intensive computation, for example, in a speech processing system.

1.7.5 Embedding an ASIP

ASIP is a processor with an instruction set designed for specific application areas on a VLSI chip or core. ASIP examples are microcontroller, DSP, IO, media, network or other domain-specific processor.

Using VLSI design tools, an ASIP with instructions sets required in the specific application areas can be designed. The ASIP is programmed using the instructions of the following functions: DSP, control signals processing, discrete cosine transformations, adaptive filtering and communication protocol-implementing functions.

1.7.6 Embedding a Multiprocessor or Dual Core Using GPPs

In an embedded system, several processors or dual core processors may be needed to execute an algorithm fast within a strict deadline. For example, in real-time video processing, the number of MAC operations needed per second may be more than is possible from one DSP unit. An embedded system then incorporates two or more processors running in synchronization. An example of using multiple ASIPs is high-definition television signals processing. [High definition means that the signals are processed for a noise-free, echo-cancelled transmission, and for obtaining a flat high-resolution image (1920×1020 pixels) on the television screen.] A cell phone or digital camera is another application with multiple ASIPs.

In a cell phone, a number of tasks have to be performed: (a) Speech signal-compression and coding. (b) Dialing (c) Modulating and Transmitting (d) Demodulating and Receiving (e) Signal decoding and decompression (f) Keypad interface and display interface handling (g) Short Message Service (SMS) protocol-based messaging (h) SMS message display. For all these tasks, a single processor does not suffice. Suitably synchronized multiple processors are used.

Consider a video conferencing system. In this system, a quarter common intermediate format—Quarter-CIF—is used. The number of image pixels is just 144×176 as against 525×625 pixels in a video picture on TV. Even then, samples of the image have to be taken at a rate of $144 \times 176 \times 30 = 760320$ pixels per second and have to be processed by compression before transmission on a telecommunication or Virtual Private Network (VPN). [Note: The number of frames are 25 or 30 per second (as per the standard adopted) for real-time displays and in motion pictures.] A single DSP-based embedded system does not suffice to get real-time images during video conferencing. Real-time video processing and multimedia applications most often need a multiprocessor unit in the embedded system.

Multiple processors or dual core processors are used when a single microprocessor does not meet the needs of the different tasks that execute concurrently. The operations of all the processors are synchronized to obtain optimum performance.

1.7.7 Embedded Processor/Embedded Microcontroller

An embedded processor is a processor with special features that allow it to embed multiple processes into the system.

Real time image processing and aerodynamics are two areas where fast, precise and intensive calculations and fast context switching (from one program to another) are essential. Embedded processor is the term sometimes used for processor that has been a specially designed such that it has the following capabilities:

1. Fast context switching and thus lower latencies of the tasks in complex real time applications. [Section 4.6] Fast context switching means that the calling program or interrupted service routine CPU registers save and retrieve fast [Section 4.6].
2. 32-bit or 64-bit atomic addition and multiplication, and no shared data problem in the operations with large operands with each operand placed in two or four registers. [Section 7.8.1]
3. 32-bit RISC core for fast, more precise and intensive calculations by the embedded software.

Embedded microcontroller is the term sometimes used for specially designed microcontrollers that have the following capabilities:

1. When a microcontroller has internal RAM, large flash or ROM, timer, interrupt handler, devices and peripherals and there is no external memory or device or peripheral required for the given application.
2. Fast context switching and thus lower latencies of the tasks in complex real time applications. For example, ARM and 68HC1x microcontrollers save all CPU registers fast

An embedded processor is term used for processors with fast processing, fast context-switching and atomic ALU operations. An embedded microcontroller is the term used for a microcontroller that has internal RAM, large flash or ROM, timer, interrupt handler, internal devices and internal peripherals and there is no external memory or device or peripheral required for the given application.

Complex System Embedded Processors Table 1.7 gives different processors that can embed in a complex system.

Table 1.7 Processors in complex embedded systems

Processor	Application	Advantage	Disadvantage
<i>General Purpose Microprocessor</i>	When intensive computations are required, caches are used and pipeline and superscalar operations are needed and large embedded software is to be located in the external memory cores or chips.	No engineering cost for designing the processor.	Additional redundant execution units that are not needed in the given system design
<i>Microcontroller</i>	Used with internal memory, devices and peripherals and when embedded software is to be located in the internal ROM or flash.	No engineering cost for designing the processor with internal memory, devices and peripherals.	Additional manufacturing costs and redundant application units which are not needed in the given system design.
<i>DSP</i>	Used with signal processing-related instructions for filters, image, audio, and video and CODEC operations.	No engineering cost involved for designing the signal processor.	Manufacturing cost may be high.
<i>Single purpose processors and application specific system processor</i>	Control IO and bus operations and peripherals and devices.	They support other processing units in the system and execute specific hardware processes fast.	In-house engineering cost of development, royalty payments for an IP core of processor and time-to-market cost.
<i>Dual core processor</i>	To significantly enhance the performance of the system.	Reduced engineering cost.	Manufacturing cost, as dual core processors are costly.
<i>Accelerator</i>	To accelerate the execution of codes. A floating point coprocessor accelerates mathematical operations and Java accelerator accelerates Java code execution.	Increases performance by co-processing with the main processor.	Engineering cost of development or royalty payments for IP core of processor and time-to-market cost.

A DSP for mobile phones, for example, OMAP of Texas Instruments, uses the effective power dissipation methods of dynamic switching both for power supply voltage and operating frequency of the CPU core.

For a number of applications, the DSPs cores may not suffice. Domain specific ASIPs have specific instruction sets. For IOs, network, media or security applications, smart card, video game, palm top computer, cell phone, mobile-Internet, hand-held embedded systems, Gbps transceivers, Gbps LAN systems, satellite or missile systems, we need special processing units in a VLSI circuit designed to function as a processor with an instruction-set for programmability. These special units are called domain-specific ASIP.

1.7.8 Embedding ARM processor

Examples of Stream 4 GPPs in Table 1.4 are ARM 7 and ARM 9. The core of these processors can be embedded onto a VLSI chip or an SoC. An ARM-processor VLSI-architecture is available either as a CPU chip or for integrating it into VLSI or SoC. ARM, Intel and Texas Instruments and several other companies have developed such processors. ARM provides CISC functionality with RISC architecture at the core. The cores of ARM7, ARM9 and their DSP enhancements are available for embedding in systems. [Refer to <http://www.ti.com/sc/docs/asic/modules/arm7.htm> and [arm9.htm](http://www.ti.com/sc/docs/asic/modules/arm9.htm)].

ARM integrates with other features (for example DSP) in new GPPs, which are available from several sources, for example, Intel and Texas Instruments. Exemplary ARM 9 applications are setup boxes, cable modems, and wireless-devices such as mobile handsets.

ARM9 has a single cycle 16×32 multiple accumulate unit. It operates at 200 MHz. It uses 0.15 μm GS30 CMOSs. It has a five-stage pipeline. It incorporates RISC core with CISC functions. It integrates with a DSP when designed for an ASIC solution. An example is its integration with DSP is TMS320C55x from Texas Instruments. [Refer to <http://www.ti.com/sc/docs/asic/modules/arm7.htm> and [arm9.htm](http://www.ti.com/sc/docs/asic/modules/arm9.htm)]

A lower performance but very popular version of ARM9 is ARM7. It operates at 80 MHz. It uses 0.18 μm based GS20 μm CMOSs. Using ARM7, ARM9 and CORTEX-M3, a large number of embedded systems have recently become available.

Lately, a new class of embedded systems has emerged that additionally incorporates ASSP chips or cores in its design.

1.7.9 Embedding ASSP

Assume that there is an embedded system for real-time video processing. Real-time processing arises for digital television, high definition TV decoders, set-up boxes, DVD (Digital Video Disc) players, web phones, video-conferencing and other systems. An ASSP that is dedicated to these specific tasks alone provides a faster solution. The ASSP is configured and interfaced with the rest of the embedded system.

Assume that there is an embedded system that using a specific protocol interconnects its units through specific bus architecture to another system. Also, assume that suitable encryption and decryption is required. [The output bit stream encryption protects messages or design from passing to an unknown external entity.] For these tasks, besides embedding the software, it may also be necessary to embed some RTOS features [Section 1.4.6]. If the software alone is used for the above tasks, it may take a longer time than a hardwired solution for application-specific processing. An ASSP chip provides such a solution. For example, an ASSP chip [from i2Chip (<http://www.i2Chip.com>)] has a TCP, UDP, IP, ARP and Ethernet 10/100 MAC (Media Access Control) hardwired logic included into it. The chip from i2Chip, W3100A, is a unique hardwired Internet connectivity solution. Much needed TCP/IP stack processing software for networking tasks is thus available as a hardwired solution. This gives output five times faster than a software solution using the system's GPP. It is also an RTOS-less solution. Using the same microcontroller in the embedded system to which this ASSP chip

interfaces, Ethernet connectivity can be added. Another ASSP, which is now available, is the 'Serial-to-Ethernet Converter (IIM7100). It does real-time data processing by a hardware protocol stack. It needs no change in the application software or firmware and provides the most economical and smallest RTOS-solution.

An ASSP is used as an additional processing unit for running application specific tasks in place of processing using embedded software.

1.8 DESIGN PROCESS IN EMBEDDED SYSTEM

The concepts used during a design process are as follows.

1. **Abstraction:** Each problem component is first abstracted. For example, in the design of a robotic system, the problem of abstraction can be in terms of control of arms and motors.
2. **Hardware and Software architecture:** Architectures should be well understood before a design.
3. **Extra functional Properties:** Extra functionalities required in the system being developed should be well understood from the design.
4. **System Related Family of designs:** Families of related systems developed earlier should be taken into consideration during designing.
5. **Modular Design:** Modular design concepts should be used. System designing is fast by decomposition of software into modules that are to be implemented. Modules should be such that they can be composed (coupled or integrated) later. Effective modular design should ensure effective (i) function independence, (ii) cohesion and (iii) coupling.
 - (a) Modules should be clearly understood and should maintain continuity.
 - (b) Also, appropriate protection strategies are necessary for each module. A module is not permitted to change or modify another module functionality. For example, protection from a device driver modifying the configuration of another device.
6. **Mapping:** Mapping into various representations is done from software requirements. For example, data flow in the same path during the program flow can be mapped together as a single entity. Transform and transaction mapping design processes are used in designing. For example, an image is input data to a system; it can have a different number of pixels and colours. The system does not process each pixel and colour individually. Transform mapping of image is done by appropriate compression and storage algorithms. Transaction mapping is done to define the sequence of images.
7. **User Interface Design:** User interface design is an important part of design. User interfaces are designed as per user requirements, analysis of the environment and system functions. For example, in an automatic chocolate vending machine (ACVM) system, the user interface is an LCD multiline graphics display. It can display a welcome message as well as specify the coins needed to be inserted into the machine for each type of chocolate. The same ACVM may be designed with touchscreen User Interface (GUI), or it may be designed with Voice User Interfaces (VUIs). Any of these interface designs has to be validated by the customer. For example, the ACVM customer who installs the machine must validate message language and messages to be displayed before an interface design can proceed to the implementation stage.
8. **Refinements:** Each component and module design needs to be refined iteratively till it becomes the most appropriate for implementation by the software team.

The software design process may require use of Architecture Description Language (ADL). It is used for representing the following: (i) Control Hierarchy (ii) Structural Partitioning (iii) Data Structure and Hierarchy (iv) Software Procedures.

Figure 1.11 shows the activities for software-design cycle during an embedded software-development process and the cycle may be repeated till tests show the verification of specifications.

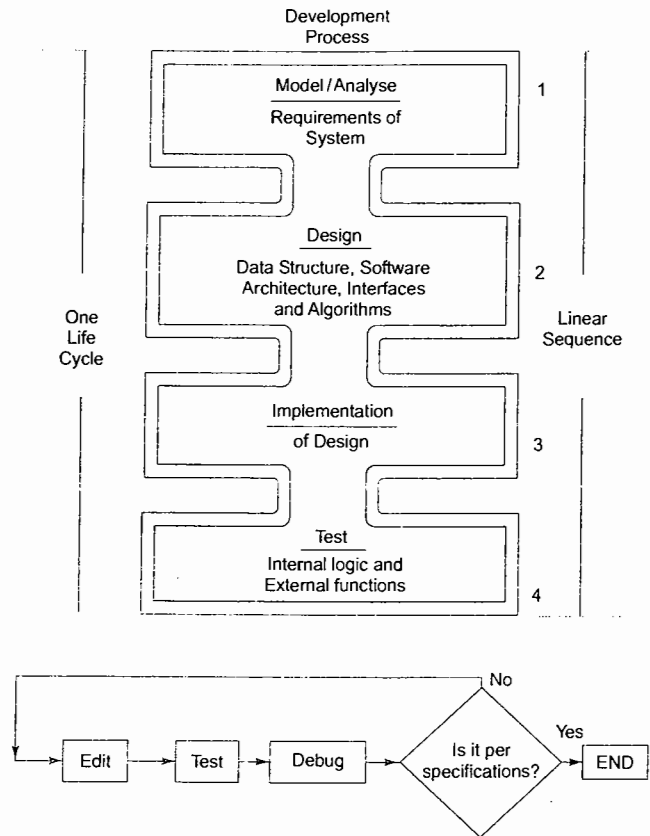


Fig. 1.11 Activities for software design during an embedded software-development process

1.8.1 Design Metrics

A design process takes into account design metrics. There are several design metrics for an embedded system, and these are listed in Table 1.8.

1.8.2 Abstraction of Steps in the Design Process

A design process is called bottom-to-top design if it builds by starting from the components. A design process is called top-to-down design if it first starts with abstraction of the process and then after abstraction the details are created. Top-to-down design approach is the most favoured approach. The following lists the five levels of abstraction from top to bottom in the design process:

- (1) **Requirements:** Definition and analysis of system requirement. It is only by a complete clarity of the required *purpose, inputs, outputs, functioning, design metrics* (Table 1.8) and *validation requirements* for finally developed systems specifications that a well designed system can be created. There has to be consistency in the requirements.

Table 1.8 Design metrics used in the embedded systems

Design Metrics	Description
Power Dissipation	For many systems, particularly battery operated systems, such as mobile phone or digital camera the power consumed by the system is an important feature. The battery needs to be recharged less frequently if power dissipation is small.
Performance	Instructions execution time in the system measures the performance. Smaller execution time means higher performance. For example, a mobile phone, voice signals processed between antenna and speaker in 0.1s shows phone performance. Consider another. For example, a digital camera, shooting a 4M pixel still image in 0.5s shows the camera performance.
Process deadlines	There are number of processes in the system, for example, keypad input processing, graphic display refresh, audio signals processing and video signals processing. These have deadlines within which each of them may be required to finish computations and give results.
User interfaces	These include keypad GUIs and VUIs.
Size	Size of the system is measured in terms of (i) physical space required, (ii) RAM in kB and internal flash memory requirements in MB or GB for running the software and for data storage and (iii) number of million logic gates in the hardware.
Engineering cost	Initial cost of developing, debugging and testing the hardware and software is called engineering cost and is a one-time non-recurring cost.
Manufacturing cost	Cost of manufacturing each unit.
Flexibility	Flexibility in design enables, without any significant engineering cost, development of different versions of a product and advanced versions later on. For example, software enhancement by adding extra functions necessitated by changing environment and software re-engineering.
Prototype development time	Time taken in days or months for developing the prototype and in-house testing for system functionalities. It includes engineering time and making the prototype time.
Time-to-market	Time taken in days or months after prototype development to put a product for users and consumers.
System and user safety	System safety in terms of accidental fall from hand or table, theft (e.g., a phone locking ability and tracing ability) and in terms of user safety when using a product (for example, automobile brake or engine).
Maintenance	Maintenance means changeability and additions to the system; for example, adding or updating software, data and hardware. Example of software maintenance is additional service or functionality software. Example of data maintenance is additional ring-tones, wallpapers, video-clips in mobile phone or extending card expiry date in case of smart card. Example of hardware maintenance is additional memory or changing the memory stick in mobile computer and digital camera.

- (2) **Specifications:** Clear specifications of the required system are must. Specifications need to be precise. Specifications guide customer expectations from the product. They also guide system architecture. The designer needs specifications for (i) hardware, for example, peripherals, devices processor and memory specifications, (ii) data types and processing specifications, (iii) expected system behaviour specifications, (iv) constraints of design, and (v) expected life cycle specifications. Process specifications are analysed by making lists of inputs on events, outputs on events and how the processes activate on each event (interrupt).
- (3) **Architecture:** Data modeling designs of attributes of data structure, data flow graphs (Section 6.2), program models (Section 6.1), software architecture layers and hardware architecture are defined. Software architectural layers are as follows:
1. The first layer is an architectural design. Here, a design for system architecture is developed. The question arises as to how the different elements—data structures, databases, algorithms, control functions, state transition functions, process, data and program flow—are to be organised.
 2. The second layer consists of data-design. Questions at this stage are as follows. What design of data structures and databases would be most appropriate for the given problem? Whether data organised as a tree- like structure will be appropriate? What will be the design of the components in the data? [For example, video information will have two components, image and sound.]
 3. The third layer consists of interface design. Important questions at this stage are as follows. What shall be the interfaces to integrate the components? What is the design for system integration? What shall be design of interfaces used for taking inputs from the data objects, structures and databases and for delivering outputs? What will be the port structure for receiving inputs and transmitting outputs?
- (4) **Components:** The fourth layer is a component level design. The question at this stage is as follows. What shall be the design of each component? There is an additional requirement in the design of embedded systems, that each component should be optimised for memory usage and power dissipation. Components of hardware, processes, interfaces and algorithms. The following lists the common hardware components:
1. Processor, ASIP and single purpose processors in the system
 2. Memory RAM, ROM or internal and external flash or secondary memory in the system
 3. Peripherals and devices internal and external to the system
 4. Ports and buses in the system
 5. Power source or battery in the system

During software development process we can model the components as object-oriented. Table 1.9 lists the stages as components-based object-oriented software development process.

- (5) **System Integration:** Built components are integrated in the system. Components may work fine independently, but when integrated may not fulfill the design metrics. The system is made to function and validated. Appropriate tests are chosen. Debugging tools are used to correct erroneous functioning. Each component and its interface system is integrated after the design stage. Program implementation is in a language and may use an integrated development environment (IDE), and source code engineering tools, which should follow the model, software architecture and design specifications. Program simplicity should be maintained during the implementation process.

The design stages range from abstraction to detailed designing to verification activities. Continuous refinement in design can be made by effective communication between designers and implementers. Software design can be assumed to consist of four layers: architecture design, data design, interfaces design and component level design.

Table 1.9 Components-based object-oriented software development process

Effort	Activities	Model Deficiency
Stage 1	Components that could be used in software development identified	
Stage 2	Selection of available classes (single logically bonded groups) from a software components resource library	Need for robust interfaces and slow development in case the reusable components are not available in required numbers
Stage 3	Sort components, which are available and reusable by re-engineering and which are unavailable	
Stage 4	Re-engineer components and create unavailable components	
Stage 5	Construct software from the components and test them	
Stage 6	Iteratively construct till final validation of software	

Actions at each step Research by software engineering experts have shown that on an average, a designer needs to spend about 50% of the time for planning, analysis and design, 40% for testing, validation and debugging and 10–15% on coding. Action required to be taken at each step in the design process is listed in Table 1.10.

Table 1.10 Action to be taken at each step of design process

Design Metrics	Description
Analysis	Design is analyzed
Steps for improvement	The result of analysis is used to improve design to meet specifications and metrics
Verification	System design must be verified to ensure that it meets the design metrics given in Table 1.8

1.8.3 Challenges in Embedded System Design: Optimizing Design Metrics

Following are the challenges that arise during the design process.

Amount and type of hardware needed: Optimizing the requirement of microprocessors, ASIPs and single purpose processors in the system on the basis of performance, power dissipation, cost and other design metrics are the challenges in a system design. A designer also chooses the appropriate hardware (memory RAM, ROM or internal and external flash or secondary memory, peripherals and devices internal and external ports and buses and power source or battery) taking into account the design metrics given in Table 1.8; for example, power dissipation, physical size, number of gates and the engineering, prototype development and manufacturing costs.

Optimizing Power Dissipation and Consumption: Power, consumption during the operational and idle state of system should be optimal. The following methods are used to meet the design challenges.

Clock Rate Reduction Power dissipation typically reduces 2.5 μ W per 100 kHz of reduced clock rate. So reduction from 8000 kHz to 100 kHz reduces power dissipation by about 200 μ W, which is nearly similar to when the clock is nonfunctional. [Remember, total power dissipated (energy required) may not reduce. This is because on reducing the clock rate, the computations will take a longer time and total energy required equals the power dissipation per second multiplied by computation time].

The power 25 μW is typically the residual dissipation needed to operate the timers and few other units. By operating the clock at a lower frequency or during the power-down mode of the processor, the advantages are as follows: (i) Power loss due to heat generation reduces. (ii) Radio frequency interference also reduces due to the reduced power dissipation within the gates. [Radiated RF (Radio Frequency) power depends on the RF current inside a gate, which reduces due to increase in 'ON' state resistance between drain and channel of each MOSFET transistor and that reduces heat generation.]

Voltage Reduction In portable or hand-held devices such as a cellular phone, compared to 5 V operation, a CMOS circuit power dissipation reduces by one sixth, $\sim(2\text{V}/5\text{V})^2$, in 2.0 V operation. Thus the time intervals needed for recharging the battery increase by a factor of six.

Wait, Stop and Cache Disable Instructions An embedded system may need to be run continuously, without being switched off; the system design, therefore, is constrained by the need to limit power dissipation while it is ON but is in idle state. Total power consumption by the system while in running, waiting and idle states should be limited. A microcontroller must provide for executing *Wait* and *Stop* instructions for the power-down mode. One way to reduce power dissipation is to cleverly incorporate into software the *Wait* and *Stop* instructions. Another is to operate the system at the lowest voltage levels in the idle state and selecting power-down mode in that state. Yet another method is to disable use of certain structural units of the processor—for example, caches—when not necessary and to keep in disconnected state those structure units that are not needed during a particular software execution, for example timers or IO units.

Operations can be performed at low voltage or reduced clock rate in order to control power dissipation. For embedded system software, performance analysis during its design phase must also include the analysis of power dissipation during program execution and during standby. An embedded system has to perform tasks continuously from power-up to power-off and may even be kept 'ON' continuously. Clever real-time programming by using 'Wait' and 'Stop' instructions and disabling certain units when not needed is one method of saving power during program execution.

Process Deadlines Meeting the deadline of all processes in the system while keeping the memory, power dissipation, processor clock rate and cost at minimum is a challenge.

Flexibility and Upgrade ability Flexibility and upgrade ability in design while keeping the cost minimum and without any significant engineering cost is a challenge. Flexibility and upgrade ability allow different and advanced versions of a product to be introduced in the market later on.

Reliability Designing a reliable product by appropriate design, testing and thorough verification, is a challenge. The goal of testing is to find errors and to validate that the implemented software is as per the specifications and requirements. Verification refers to an activity to ensure that specific functions are correctly implemented. Validation refers to an activity to ensure that the system that has been created is as per the requirements agreed upon at the analysis phase, and to ensure its quality.

1.9 FORMALIZATION OF SYSTEM DESIGN

Formalization of system design is done using a top-down approach by abstraction (Section 1.8.2) and by

- Detailing requirements and specifications of hardware and software

- Defining architectures of hardware and software
- Coding and implementation as per architecture
- Testing, validation and verification of system

Since a diagrammatic model clears the design concepts better than abstraction, a modeling language, for formalization can be used. The Universal Modeling Language (UML) is used. In UML, a designer describes the following:

1. 'User Diagram', 'Object Diagram', 'Sequence Diagram', 'State Diagram', 'Class Diagram' and 'Activity Diagram'
2. Classes and Objects, which describe *identity, attributes, components* and *behaviour*
3. Inheritances of the classes and objects
4. Interfaces of the objects and their implementation at the objects
5. Structural description of the design components
6. Behavioral description in terms of states, state machine and signals (Section 6.3)
7. Events description

Section 6.5 will describe UML in detail. Chapters 11 and 12 will describe the model design examples in detail.

1.10 DESIGN PROCESS AND DESIGN EXAMPLES

1.10.1 System Design Process Examples

Chapters 11 and 12 will describe design examples in detail.

1.10.2 Automatic Chocolate Vending Machine (ACVM)

Let us consider an automatic chocolate vending machine. This interesting example given here helps a reader to understand several concepts of programming an embedded system as a multitasking system.

Figure 1.12 shows the diagrammatic representation of ACVM. Assume that ACVM has following components:

1. It has keypad on the top of the machine. That enables a child to interact with it when buying a chocolate. The owner can also command and interact with the machine.
2. It has an LCD display unit on the top of the machine. It displays menus, text entered into the ACVM and pictograms, welcome, thank you and other messages. It enables the child as well as the ACVM owner to graphically interact with the machine. It also displays time and date. (For GUIs, the keypad and LCD display units or touch screen are basic units.)
3. It has a coin insertion slot and a mechanical coin sorter so that child can insert coins to buy a chocolate.
4. It has a delivery slot so that child can collect the chocolate and coins, if refunded.
5. It has an Internet connection port using a USB based wireless modem so that owner can know status of the ACVM sales from a remote location.

ACVM Functions Assume that ACVM functions are as follows:

1. The ACVM displays the GUIs and if the child wishes to enter contact information, birthday information or get answer to FAQs, it displays the appropriate menu.
2. It displays a welcome message when in idle state. It also continuously displays time and date at the right bottom corner of display screen. It can also intermittently display news, weather data or advertisements or important information of interest during idle state.

- When first coin is inserted, a timer also starts. The child is expected to insert all required coins in 2 minutes.
- After 2 minutes the ACVM will display a query to the child if the child does not insert sufficient coins. If the query is not answered the coins are refunded.
- Within 2 minutes if sufficient coins are collected, it displays the message, 'Thanks, wait for few moments please!', delivers the chocolate through the delivery slot and displays message, 'Collect the chocolate and visit again, please!'

Hardware units ACVM embeds the following hardware units.

- Microcontroller or ASIP (Application Specific Instruction Set Processor)
- RAM for storing temporary variables and stack
- ROM for application codes and RTOS codes for scheduling the tasks
- Flash memory for storing user preferences, contact data, user address, user date of birth, user identification code, answers of FAQs
- Timer and interrupt controller
- A TCP/IP port (Internet broadband connection) to the ACVM for remote control and for the owner to get ACVM status reports
- ACVM specific hardware to sort coins of different denominations. Each denomination coin generates a set of status and input bits and port-interrupts. Using an ISR for that port, the ACVM processor reads the port status and input bits. The bits give the information about which coin has been inserted. After each read operation, the status bits are reset by the routine
- Power supply

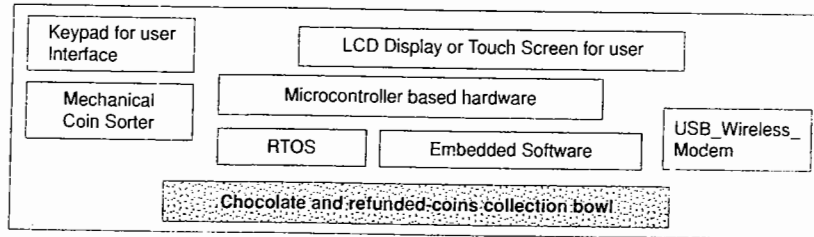


Fig. 1.12 Diagrammatic representation of the ACVM

Software components ACVM embeds the following software components:

- Keypad input read task
- Display task
- Read coins task for finding coins sorted
- Deliver chocolate task
- TCP/IP stack processing task
- TCP/IP stack communication task

1.10.3 Smart Card

Smart card is one of the most used embedded system today. It is used for credit-debit bankcard, ATM card, e-purse or e-Wallet card, identification card, medical card (for history and diagnosis details) and card for a

number of new innovative applications. [Reader may refer to a frequently updated website. <http://www.sguthery@tiac.net> for the answers of frequently asked questions about cards.] The security aspect is of paramount importance for smart card use, when used for financial and banking-related transactions. [Reader may refer to <http://www.home.hkstar.com/~alanchan/papers/smartCardSecurity/> and http://www.research.ibm.com/secure_systems/scard.htm for details of the card-security requirements.]

The smart card is a plastic card ISO standard dimensions, $85.60 \times 53.98 \times 0.80$ mm. It is an embedded system on a card: SoC (System-On-Chip). ISO recommended standards are ISO7816 (1 to 4) for host-machine contact-based cards and ISO14443 (Part A or B) for the contactless cards. The silicon chip is just a few millimeters in size and is concealed in-between the layers. Its very small size protects the card from bending. Figure 1.13 shows embedded-system hardware components for a contactless smart card.

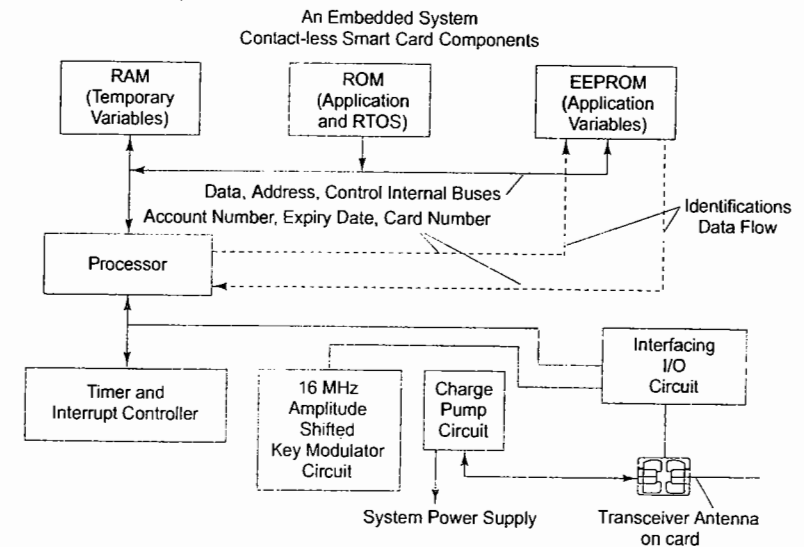


Fig. 1.13 Embedded hardware components in a contact less smart card

Embedded Hardware The embedded hardware components are as follows:

- Microcontroller or ASIP
- RAM for temporary variables and stack
- One time programmable ROM for application codes and RTOS codes for scheduling the tasks
- Flash for storing user data, user address, user identification codes, card number and expiry date
- Timer and interrupt controller
- A carrier frequency ~16 MHz generating circuit and Amplitude Shifted Key (ASK) modulator
- Interfacing circuit for the IOs
- Charge pump for delivering power to the antenna for transmission and for system circuits. The charge pump stores charge from received RF (radio frequency) at the card antenna in its vicinity. [The charge pump is a simple circuit that consists of the diode and high value ferroelectrics material-based capacitor.] The details of the basic hardware units are as follows:

1. The **Microcontroller** used can be MC68HC11D0 or PIC16C84 or a smart card processor Philips Smart XA or a similar ASIP Processor. MC68HC11D0 has 8 kB internal RAM and 32 kB EPROM and 2/3 wire protected memory. Most cards use 8-bit CPUs. The recent introduction in the cards is of a 32-bit RISC CPU. A smart card CPU should have special features, for example, a security lock. The lock is for a certain sections of the memory. A protection bit at the microcontroller may protect 1 kB or more data from modification and access by any external source or instructions outside that memory. Once the protection bit is placed at the maskable ROM in the microcontroller, the instructions or data within that part of the memory are accessible from instructions in that part only (internally) and not accessible from the external instructions or instructions outside that part. The CPU may disable access by blocking the write cycle placement of the data bits on the buses for instructions and data protection at the physical memory after certain phases of card initialization and before issuing the card to the user. Another way of protecting is as follows: The CPU may access by using the physical addresses, which are different from the logical address used in the program.
2. **ROM** is used in the card. The usual size is 8 or 64 kB for usual or advanced cryptographic features in the card, respectively. Full or part of ROM bus activates only after a security check. The processor protects a part of the memory from access. The ROM stores the following.
 - i. Fabrication key, which is a unique secret key for each card. It is inserted during fabrication.
 - ii. Personalization key, which is inserted after the chip is tested on a printed circuit board. Physical addresses are used in the testing phase. The key preserves the fabrication key and this key insertion preserves the card personalization. After insertion of this key, RTOS and applications use only logical addresses.
 - iii. RTOS codes
 - iv. Application codes
 - v. A utilization lock to prevent modification of two PINs and to prevent access to the OS and application instructions. It stores after the card enters the utilization phase.
3. **EEPROM or Flash** is scalable. These means that only that part of the memory required for a particular operation will unlock for use. The authorizer will use the required part; the application will use the other part. It is protected by the access conditions stored therein. It stores the following:
 - i. PIN (Personal Identification Number), the allotment and writing of which is by the authorizer (for example, a bank) and its use is possible by the latter only by using the personalization and fabrication keys. It is for identifying the card user in future transactions. Card user is given this key. Alternatively, a modifiable password is given to the user and password opens the PIN key.
 - ii. An unblocking PIN for use by the authorizer (say the bank). Through this key, the card circuit identifies the authorizer before unblocking. Data of the user unblocks for the authorizer and storing of information on the card is possible by the authorizer through the host.
 - iii. Access conditions for various hierarchically arranged data files.
 - iv. Card user data, for example, name, bank and branch identification number and account number or health insurance details.
 - v. Data post issue that the *application* generates. For example, in case of e-purse, the details of previous transactions and current balance. Medical history and diagnosis details and/or previous insurance claims and pending insurance claims record in case of a medical card.
 - vi. It also stores the application's non-volatile data.
 - vii. Invalidation lock sent by the host after the expiry period or card misuse and user account closing request. It locks the data files of the master or elementary individual file or both.
4. **RAM** stores the temporary variables and stack during card operations by running the OS and the *application*.

5. **Chip power supply** voltage extracts by a charge pump circuit. The pump extracts the charge from the signals from the host analogous to what a mouse does in a computer and delivers the regulated voltage to the card chip, memory and IO system. Signals can be from antenna or from clock pin. In a typical card operation using 0.18 μm technology, 1.6 to 5.5 V is the threshold limit and for a 0.35 μm technology, 2.7 to 5.5 V.
6. **IO System** of chip and host interact through asynchronous serial UART (Section 3.2.3) at 9.6 k or 106 k or 115.2 k baud/s. The chip interconnects to a card hosting system (reader and writer) either through the gold contacts or through a centimeter sized antenna on each side. The latter provides *contactless* interconnection between the IO pins, which are meant for *contact-based* interaction, RST (Reset Signal from host) and Clock (from host).
7. **Wireless Communication** for IO interaction is by radiations through the antenna coils for contactless interaction. The card and host interact through a card modem and a host modem. The application protocol data unit (APDU) is a standard for communication between the card and host computer. Modulation is with 10% index amplitude modulating carrier of 13.66–13.56 Mbps ASK (amplitude shifted keying) is used for contactless communication at data rates of ~1 Mbps. One-sixteenth frequency subcarrier modulates through BPSK (Binary Phase Shifted Keying).

Embedded Software Smart card embeds the following software components:

1. Boot-up, initialisation and OS programs
2. Smart card secure file system
3. Connection establishment and termination
4. Communication with host
5. Cryptography algorithm
6. Host authentication
7. Card authentication
8. Saving addition parameters or recent new data sent by the host (for example, present balance left)

The smart card is an exemplary secure embedded system with security software. The card needs cryptographic software. Embedded software in the card needs special features in its operating system over and above the MS DOS or UNIX system features. Special features needed are as follows:

1. *Protected environment*. It means software should be stored in the protected part of the ROM.
2. *Restricted run-time environment*.
3. Its OS, every method, class and run time library should be scalable.
4. Code-size generated should be optimum. The system needs should not exceed 64 kB memory.
5. Limited use of data types; multidimensional arrays, long 64-bit integer and floating points and very limited use of the error handlers, exceptions (Section 4.2.2), signals (Sections 6.5 and 7.10), serialization, debugging and profiling. [Serialization is the process of converting an object into a data stream (for transferring it to network or from one process to another. The de-serialized data are the receiver end)].
6. A three-layered file system for the data. One file for the *master file* to store all file headers. A header means file status, access conditions and the file lock. The second file is a *dedicated file* to hold a file grouping and headers of the immediate successor elementary files of the group. The third file is the *elementary file* to hold the file header and its file data.
7. There is either a fixed length file management or a variable length file management with each file having a predefined offset.
8. Classes for the network, sockets, connections, data grams, character-input output and streams, security management, digital-certification, symmetric and asymmetric keys-based cryptography and digital signatures.

1.10.4 Digital Camera

Digital cameras may have 4 to 6 M pixel still images, clear visual display (ClearVid) CMOS sensor, 7 cm wide LCD photo display screen, enhanced imaging processor, double anti blur solution and high-speed processing engine, 10X optical and 20X digital zooms and can also record high definition video-clips. It therefore has speaker microphone(s) for high quality recorded sound. It has an audio/video out port for connecting to a TV/DVD player or computer.

Let us assume that the camera is still just a camera. Figures 1.14(a) and (b) show hardware and software components in a simple digital camera. Assume that the camera has the following components:

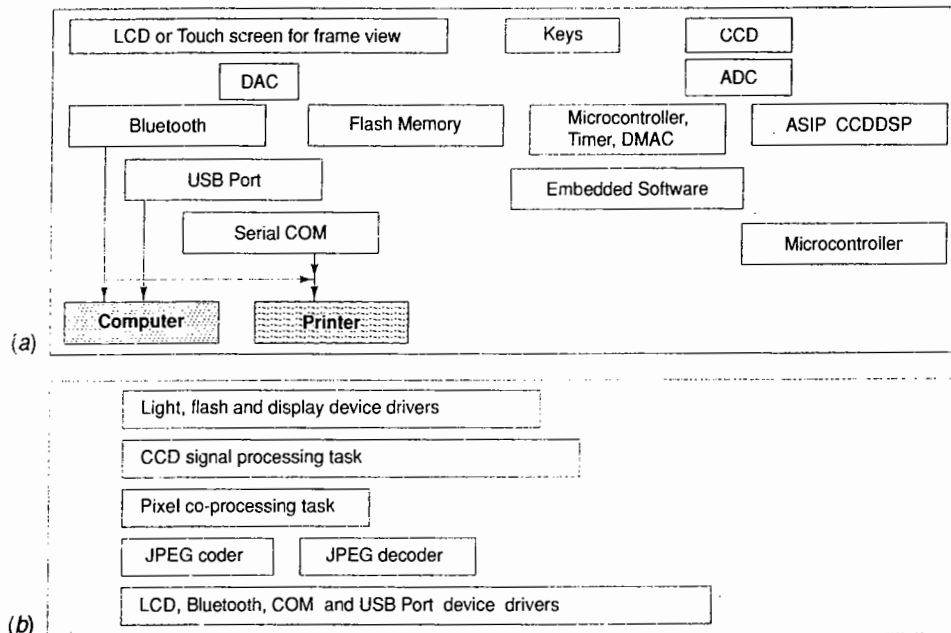


Fig. 1.14 (a) Digital camera hardware components (b) Digital camera software components

1. It has keys on the camera. That enables a user to operate the camera. It has navigation keys to navigate through the images back and forth.
2. Shutter, lens and charge coupled device (CCD) array sensors for images in sizes 2592×1944 pixels = 5038848 pixels, VGA (E-mail) $640 \times 480 = 307200$ pixels, $2592 \times 1728 = 3.2$ M pixels, 2048×1536 pixels = 3 M pixels, or 1280×960 pixels = 1 M pixels.
3. It has a good resolution photo quality LCD display unit on the back of camera to show photographs or recorded video-clips. It displays text such as image-title, shooting data and time and serial number. It displays messages. It displays the GUI menus when the user interacts with the camera.
4. It has a self-timer lamp for flash.
5. Internal memory flash to store OS and embedded software, and limited number of image files.
6. Flash memory stick of 2 GB or more for large storage.

7. It has Universal Serial Bus (USB) port (Section 3.10.3) or Bluetooth interface, which connects it to a computer and printer.

Camera Functions Assume that the camera functions is as follows:

1. It displays the frame view on the LCD screen so that user can adjust the camera inclination before shooting the frame.
2. It displays the saved images on the LCD using navigation keys.
3. When a key for opening the shutter is pressed, the flash lamp glows and the self-timer circuit switches off the lamp automatically.
4. The frame light falls on the CCD array, which transmits the bits for each pixel in each row in the frame through an ADC. Bits from dark area pixels in each row are used for offset corrections in the CCD signal for light intensities in each row.
5. The CCD bits of each pixel in each row and column are offset corrected using a CCD signal processor (CCDSP).
6. The processed signals are compressed using a JPEG CODEC and saved in one jpg file for each frame. A DSP does compression using the discrete cosine transformations (DCTs) and decompression by inverse DCT. Thereafter, it also does Huffman coding for JPEG compression.
7. A DAC sends the inputs for the display unit. The DAC gets the input from the pixel processor, which gets the inputs from the JPEG files for the saved images and gets input directly from the CCDSP through the pixel processor or the frame in the present view.

Digital Hardware units The camera embeds the following hardware units.

1. Microcontroller or ASIP
2. Multiple processors (CCDSP, DSP, pixel processor and others)
3. RAM for storing temporary variables and stack
4. ROM for application codes and RTOS codes for scheduling tasks
5. Timer, flash memory for storing user preferences, contact data, user address, user date of birth, user identification code, ADC, DAC and interrupt controller (Sections 1.3.3, 1.3.5, 1.3.7 and 1.3.11)
6. USB controller (Section 3.10.3)
7. Direct memory access controller (Section 4.8)
8. LCD controller (Section 3.3.4)
9. Battery

Software components The camera embeds the following software components:

1. CCD signal processing for off-set correction
2. JPEG coding
3. JPEG decoding
4. Pixel processing before display
5. Memory and file systems
6. Light, flash and display device drivers
7. COM, USB port and Bluetooth device drivers for port operations for printer and computer communication control

1.10.5 Mobile Phone

The mobile phone today has a large number of features. It has sophisticated hardware and software.

Hardware units A mobile phone embeds an SoC (System-on-Chip) integrating the following hardware units.

1. Microcontroller or ASIP [An ASIP is configured to process encoding and deciphering and another does the voice compression. Third ASIC dials, modulates, demodulates, interfaces the keyboard and touch screen or multiple line LCD graphic displays, and processes the data input and recall of data from memory].
2. DSP core, CCDSP, DSP, video, voice and pixel processors
3. Flash, memory stick, EEPROMs and SRAMs
4. Peripheral circuits, ADC, DAC and interrupt controller
5. Direct memory access controller (Section 4.8)
6. LCD controller (Section 3.3.4)
7. Battery

Software components The mobile phone software development tools are as follows:

1. OS (Windows Mobile, Palm, Symbian) or BREW
2. Java 2 Micro Edition (J2ME) along with KVM as a Java Virtual Machine (Section 5.7.4)
3. Java Wireless toolkit with JDK (Java Development Kit)

The mobile phone embeds the following software components:

1. Memory and file systems
2. Keypad, LCD, serial, USB, 3G or 2G port device drivers for port operations for keypad, printer and computer communication control
3. SMS (Short Messaging Service) message creation and communicator, contact and PIM (personal information manager), task-to-do manager and email
4. Mobile imager for uploading pictures and MMS (multimedia messaging service)
5. Mobile browser for access to the Web
6. Downloader for Java games, ring-tones, games, wall papers
7. Simple camera with (Section 1.10.4)
8. Bluetooth synchronization, IrDA and WAP connections support (Section 3.13)

1.10.6 Mobile Computer

The mobile computer has Windows CE or Windows mobile as OS. It has a touch screen for GUI. The user uses a stylus to enter commands. It has a virtual keypad (the keypad displayed on the screen and entries of text and commands is through the stylus).

In addition to phone, a mobile computer has following software components:

1. OS (Windows CE, Windows Mobile, PocketPC, Palm OS or Symbian OS)
2. Touch screen GUIs, memory and file systems
3. Memory stick
4. Outlook, Internet explorer, Word, Excel, Powerpoint, and handwritten text processor
5. Applications or enterprise software

1.10.7 A Set of Robots

Consider a set of 8 robots. One robot is the master robot (music director) and seven are slave robots (conductors). Assume that the set is used to play an orchestra. Figures 1.14(a) and (b) show hardware and software components in the set of robots. Assume that the robot has the following components.

1. The master robot signals the commands and slave robots play accordingly.

2. Each robot is assumed to have five degrees of freedom. Each robot has a mechanical system of five degrees of freedom. At each degree of freedom, there is a servomotor. A servomotor controls by PWM method (Section 1.3.7). Each motor is controlled in a sequence to let the robot perform the desired action.
3. Each robot has a microcontroller with expansion ports, P0, ..., P8. Actually a single ASIC can perform multiple port functions of a microcontroller. However since the engineering cost of ASIC development is high, a general purpose microcontroller 68HC12 or 8051 is used.
4. The port outputs connect the motors and PWM outputs drive the motors in each robot.
5. Each robot has a serial IO with IrDA protocol. (Section 3.13.1)
6. Internal memory flash to store the OS, embedded software and limited amount of music.
7. There is a music file processor for playing the music. Slave robots have speaker outputs for playing music.

Master Robot Functions Assume that master robot functioning is as follows:

1. It receives commands from a remote controller to start and stop the music and the code for the specific orchestra to be played.
2. It sends PWM signals to the ports for moving the sticks in both hands as per the program.
3. It establishes and binds the sockets (the virtual devices) connection with the slaves. It sends the signals through sockets using IrDA protocols. The byte streams response to the clients are as per the music file to be played by the slave.

Slave Robot Functions Assume that slave robot functioning is as follows:

1. It establishes and binds the sockets (the virtual devices) connection with the master.
2. It receives from a master socket the commands accept () and write (); it also receives commands to start and stop music and the code for the specific orchestra to be played.
3. It receives the signals through sockets using IrDA protocols. The byte streams from the server are as per the music file being played.

Hardware units Robots embed the following hardware units.

1. Microcontroller or ASIP
2. Music file processor
3. RAM for storing temporary variables and stack
4. ROM for application codes and RTOS codes for scheduling robot actions and tasks
5. Timer, flash memory for storing user preferences and music files
6. IrDA controller (Section 3.13.1)
7. Direct memory access controller (Section 4.8)
8. Power supply source or battery

Software components Robots embed the following software components:

1. Socket functions
2. Music coding
3. Music decoding
4. Memory and file systems
5. Light, flash and display device drivers
6. IrDA and socket port device drivers
7. Motor drivers
8. IO ISRs

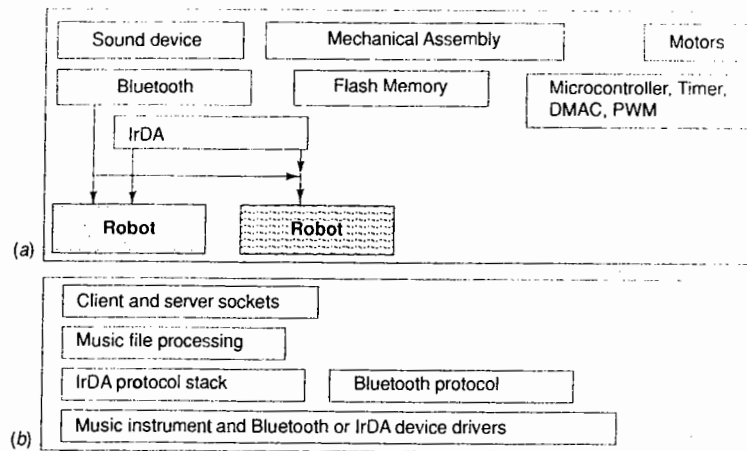


Fig. 1.15 (a) Hardware components in the set of robots (b) software components in the set of robots in which a master robot signals the commands and slave robots play according to the signals from the master

1.11 CLASSIFICATION OF EMBEDDED SYSTEMS

We can classify embedded systems into three types as follows.

1. **Small scale embedded systems:** These systems are designed with a single 8- or 16-bit microcontroller; they have little hardware and software complexities and involve board-level design. They may even be battery operated. When developing embedded software for these, an editor, assembler and cross assembler, an integrated development environment (ISE) tool specific to the microcontroller or processor used, are the main programming tools. Using 'C' language, programs are compiled into the assembly and executable codes are appropriately located in the system memory. The software has to fit within the memory available and keep in view the need to limit power dissipation when the system is running continuously.
2. **Medium scale embedded systems:** These systems are usually designed with a single or a few 16- or 32-bit microcontrollers, DSPs or RISCs. These systems may also employ the readily available single purpose processors and IPs (explained later) for the various functions—for example, bus interfacing. [ASSPs and IPs may also have to be appropriately configured by the system software before being integrated into the system-bus.] Medium scale embedded systems have both hardware and software complexities. For complex software design, the following programming tools are available: C/C++/Visual C++/Java, RTOS, source code engineering tool, simulator, debugger and an integrated development environment. Software tools also provide solutions to hardware complexities.
3. **Sophisticated embedded systems:** Sophisticated embedded systems have enormous hardware and software complexities and may need several IPs, ASIPs, scalable processors or configurable processors and programmable logic arrays. They are used for cutting edge applications that need hardware and software co-design and components that have to be integrated in the final system. They are constrained by the processing speeds available in their hardware units. Certain software functions such as encryption

and deciphering algorithms, discrete cosine transformation and inverse transformation algorithms. TCP/IP protocol stacking and network driver functions are implemented in the hardware to obtain additional speeds. The software implements some of the functions of the hardware resources in the system. Development tools for these systems may not be readily available at a reasonable cost or may not be available at all. In some cases, a compiler or retargetable compiler might have to be developed for these. [A retargetable compiler is one that configures according to the given target configuration in a system.]

1.12 SKILLS REQUIRED FOR AN EMBEDDED SYSTEM DESIGNER

An embedded system designer has to develop a product using the available tools within the given specifications, cost and time frame. [Chapters 6, 13 and 14 will cover the design aspects of embedded systems.]

1. **Skills for Small Scale Embedded System Designer:** Author Tim Wilmshurst in his book states that the following skills are needed in the individual or team that is developing a small-scale system: "Full understanding of microcontrollers with a basic knowledge of computer architecture, digital electronic design, software engineering, data communication, control engineering, motors and actuators, sensors and measurements, analog electronic design and IC design and manufacture". Specific skills will be needed in specific situations. For example, control engineering knowledge will be needed for design of control systems, and analog electronic design knowledge will be needed when designing the system interfaces. The basic aspects of the following topics will be described in this book to prepare the designer who already has a good knowledge of the microprocessor or microcontroller to be used. (i) Computer architecture and organization. (ii) Memories. (iii) Memory allocation. (iv) Interfacing memories. (v) Burning (a term used for porting) the executable machine codes in PROM or ROM. (vi) Use of decoders and demultiplexers. (vii) Direct memory accesses. (viii) Ports. (ix) Device drivers in assembly. (x) Simple and sophisticated buses. (xi) Timers. (xii) Interrupt servicing mechanism. (xiii) C programming elements. (xiv) Memory optimization. (xv) Selection of hardware and microcontroller. (xvi) Use of In-Circuit-Emulators (ICE), cross-assemblers and testing equipment. (xvii) Debugging the software and hardware bugs by using test vectors. Basic knowledge in other areas—software engineering, data communication, control engineering, motors and actuators, sensors and measurements, analog electronic design and IC design and manufacture—can be obtained from the standard text books available. A designer interested in small-scale embedded systems may not need at all concepts of interrupt latencies and deadlines and their handling, the RTOS programming tools described in Chapters 9 and 10 and the program models given in Chapter 6.
2. **Skills for Medium Scale Embedded System Designer:** Knowledge of 'C'/C++/Java programming, RTOS programming and program modeling skills are must to design medium-scale embedded-system. Knowledge of the following are critical. (i) Tasks or threads and their scheduling by RTOS. (ii) Cooperative and preemptive scheduling. (iii) Inter processor communication functions. (iv) Use of shared data, and programming the critical sections and re-entrant functions. (v) Use of semaphores, mailboxes, queues, sockets and pipes. (vi) Handling of interrupt-latencies and meeting task deadlines. (vii) Use of various RTOS functions. (viii) Use of physical and virtual device drivers. [Refer to Sections 4.2.6, 7.10 and 7.11.] Chapters 4 to 10 give detailed descriptions of these seven along with examples, and Chapters 11 and 12 provide an understanding of their use with the help of case studies. A designer must have access to an RTOS programming tool with Application Programming Interfaces (APIs) for the specific microcontroller to be used. Solutions to various functions like memory-allocation, timers, device drivers and interrupt handling mechanism are readily available as the APIs of the RTOS. The designer needs to

know only the hardware organization and use of these APIs. The microcontroller or processor then represents a small system element for the designer and a little knowledge may suffice.

3. **Skills for Sophisticated Embedded System Designer:** A team is needed to co-design and solve the high level complexities of hardware and software design. Embedded system hardware engineers should have skills in hardware units and basic knowledge of 'C'/C++ and Java, RTOS and other programming tools. Software engineers should have basic knowledge in hardware and a thorough knowledge of 'C', RTOS and other programming tools. A final optimum design solution is then obtained by system integration.



Summary

- An embedded system is one that has embedded software in a computer-hardware, which makes it a system dedicated for an application(s) or a specific part of an application or product or a part of a larger system.
- The embedded system processor can be a general-purpose processor chosen from a number of families of microprocessors. Alternatively, an ASIP for example microcontrollers, embedded processors and DSPs may be designed for specific application on a VLSI chip. An application specific instruction set processor (ASSP) may be additionally used for fast hardwired implementation of a certain part of the embedded software. A sophisticated embedded system may also use a multiprocessor or dual core unit.
- Embedded systems locate a software image in ROM. The image often consists of the following: (i) Boot up program. (ii) Initialization data. (iii) Strings for an initial screen display or system state. (iv) Programs for the multiple tasks that the system performs. (v) RTOS kernel.
- The embedded system needs a power source and controlled and optimized power dissipation from the total energy requirement. A charge pump provides a power-supply-less system in certain embedded systems.
- The embedded system needs clock and reset circuits. Use of the clock manager is a recent innovation.
- The embedded system needs interfaces: Input Output (IO) ports, serial UART and other ports to accept inputs and to send outputs by interacting with peripherals, display units, keypad or keyboard.
- The embedded system may need bus controllers for networking its buses with other systems.
- The embedded system needs timers and a watchdog timer for the system clock and for real-time program scheduling and control.
- The embedded system needs an interrupt-controlling unit.
- The embedded system may need an ADC for taking analog input from one or multiple sources. It needs a DAC using PWM for sending analog output to motors, speakers, sound systems, etc.
- The embedded system may need an LED or LCD or touch screen display units, keypad and keyboard, pulse dialer, modem, transmitter, multiplexers and demultiplexers.
- Embedded software is usually made in the high-level languages C, C++, Java or visual C++ with certain features added, enabled or disabled for programming. Use of 'C' and C++ also facilitates the incorporation of assembly language codes.
- The embedded system most often needs a real-time operating system for real-time programming and scheduling, device drivers, file system or device management and multitasking.
- The embedded system needs a debugger.
- A large number of applications and products employ embedded systems. A number of software tools are needed in the development and design phase of an embedded system.
- Five applications are described in detail: An automatic chocolate vending machine, a smart card, digital camera, mobile phone, mobile computer and robots playing an orchestra.
- A VLSI chip can embed ASIP or a GPP core and IPs for the specific application. A system on-chip is the concept in embedded systems; for example, a mobile phone in which analog and digital circuits, processors and software reside on a chip and become a system.



Keywords and their Definitions

- ADC** : A circuit that converts the analog input to digital 8, 10 or 12 bits. The analog input is applied between positive and negative pins and is converted with respect to the reference voltage(s). When input equals reference positive and negative voltage, then all output bits equal 1, and when 0, then all output bits equal 0.
- ASIP (Application Specific Instruction Set)** : A processor with an instruction set designed for specific application on a VLSI chip; for example, microcontroller, DSP, IO, media, network or other domain-specific processor.
- Assembler** : A program that translates assembly language software into the machine codes placed in a file called '.exe' (executable) file.
- ASSP (Application Specific System Processor)** : A processing unit for system specific tasks, for example, image processing, compression and decompression, and that is integrated through the buses with the main processor in embedded system.
- Bus** : A bus consists of a common set of parallel lines to connect multiple devices, hardware units and systems for communication between two of these at any given instance. A communication protocol specifies the ways of communication of signals on bus. Protocol also specifies ways of arbitration when several devices need to communicate through the bus or ways of polling from the device requirements of the bus at an instance.
- Cache** : A fast read and write on-chip memory for the processor execution unit. It stores instructions and data fetched in advance from ROM or RAM for use in the execution unit and for data write back for RAM. It has an advantage in that the processor execution unit does not have to wait for instructions and data from external buses and also does faster write back of data meant for RAM.
- Clock** : Fixed frequency pulses that an oscillator circuit generates and that controls all operations during processing and all timing references of the system. Frequency depends on the needs of the processor circuit. A processor, if it needs a 100 MHz clock then its minimum instruction processing time is a reciprocal of it, which is 10 ns in a single cycle per instruction processing.
- CODEC** : A circuit for encoding the input information in fewer bits and for decoding the encoded information into the complete set of the original. Voice, speech, image and video signals and bits can be encoded and decoded. The CODEC also functions as a compression and decompression unit for voice, speech, image and video signals.
- Coder** : A coder which is a part of CODEC circuit is used to encode the input information.
- Compiler** : A program that, according to the processor specification, generates machine codes from high level language. The codes are called object codes.

- Cycle** : A cycle consists of fetch of an instruction from RAM/ROM, its execution at the processor and writing back the results of the operations.
- DAC** : Digital bits (8 or 10 or 12) converted to analog signal scaled to a reference voltage. When all input bits equal 1, then analog output equals the difference between the positive and negative reference pin voltages; when all input bits equal 0, then the analog output equals negative pin reference voltage.
- Decoder** : A decoder circuit that decodes the input address and activates a selected output among the many outputs. It is used for selecting one among several addressable units. A decoder also decodes the encoded bits and generates the output bits, signal or information.
- Demultiplexer** : A digital circuit that has digital outputs in multiple channels. The channel to which output is sent from the input is the one that is addressed by the channel address bits at the demultiplexer input. A demultiplexer takes the input and transfers it to a select channel output among the multiple output channels.
- Design metrics** : The parameters that define design requirements and must be kept in view during each stage of design process.
- Device Driver** : High level language functions, such as open (configure), connect, bind, listen, read or write or close the device. Each function calls an interrupt service routine (software) that runs after the programming of control register (or word) of a peripheral device (or virtual device) to allow the device inputs or outputs. The routine reads the status register for device status, gets the inputs from the device and writes the output to the device.
- Device Manager** : Software to manage multiple devices and their drivers.
- Device Programmer** : It takes the inputs from a file generated by the locator and burns the fusible link to actually store the data and system codes at the ROM.
- Embedded system** : A system that has embedded software in a computer-hardware that makes it a system dedicated for an application or a specific part of an application or product, or a part of a larger system.
- File** : A data structure (or virtual device) that sends the records (characters or words) to a data sink (for example, a program) and that stores the data from the data source (for example, a program). A file in a computer may also be stored in the hard disk.
- File System** : A file system specifies the ways a file can be created, called, named, used, copied, saved or deleted.
- FPGA** : These are Field Programmable Gate Arrays on a chip. The chip has a large number of arrays with each element having links. Each element of array consists of several XOR, AND, OR, multiplexer, demultiplexer and tristate gates. Complex digital circuit functions are created by appropriate programming of the links on transferring data from memory.
- GPP (General-purpose processor)** : A processor from a number of families of processors, microcontrollers, embedded processors and digital signal processors (DSPs) having a general-purpose instruction set and readily available compilers to enable programming in a high level language.
- Input Output (IO) ports** : The system gets the inputs and outputs from these. Through these, the keypad or LCD units or touchscreen or peripherals and external systems connect to the system.
- Interrupt controller (handler)** : A unit that controls (handles) processor operations arising out of an interrupt from a source.

- Kernel** : An OS program with the following functions: memory allocation and deallocation, task scheduling, interprocess communication, effective management of shared memory access by using signals, exception (error) handling signals, semaphores, queues, mailboxes, pipes and sockets I/O management, file system, interrupts control (handler), device drivers and device management.
- LCD** : Liquid crystal display—a crystal that absorbs or emits light on application of 3 to 4 V 50 or 60 Hz voltage pulses with currents ~ 50 μ A. Multisegment and multiline LCD units are used for a display of digits, characters, charts, pictograms and short messages with very low power dissipation.
- LED** : Light Emitting diode—a diode that emits red, green, yellow or infrared light on forward biasing between 1.6 V to 2 V and currents between 8–15 mA. Multisegment and multiline LED units are used for bright display of digits, characters, charts and short messages.
- Linker** : A program that links the compiled codes with other codes and provides input for a loader or locator.
- Loader** : A program that reallocates the physical memory addresses for loading into the system RAM memory. Reallocation is necessary, as the available memory may not start from 0x0000 at any given instant of processing in a computer. The loader is a part of the OS in a computer.
- Locator** : A program to reallocate the linked files of the program application and the RTOS codes at the actual addresses of the ROM memory. It creates a file in a standard format. The file is called a ROM image.
- Mask and ROM mask** : Created at a foundry for fabrication of a chip. The ROM mask is created from a ROM image file.
- Memory** : This stores all the programs, input data and output data. The processor fetches instructions from it and gives the processed results back to it.
- Memory Stick** : A memory stick (card) is unit of memory for video, images, songs, or speeches and is used as large storage in digital camera and mobile systems. For example, Sony memory stick Micro (M2) has size $15 \times 12.5 \times 1.2 \text{ mm}^3$ and functions as flash memory of 2 GB.
- Multiplexer** : A digital circuit that has digital inputs at any instance in multiple channels. The channel for which the output is sent from the input is the one that is addressed by the channel address bits at the multiplexer input. A multiplexer takes the input among the multiple input channels and transfers a selected channel input to the output channel.
- Microcontroller** : A unit with a processor, memory, timers, a watchdog timer, interrupt controller, ADC or PWM, and so on are provided as required by the application.
- Modem** : A circuit to modulate the outgoing bits into pulses usually used on the telephone line and to demodulate the incoming pulses into bits for incoming messages.
- Multitasking** : Processing codes for the different tasks as directed by the OS.
- Physical Device** : A device like a printer or keypad connected to the system port.
- Pipe** : A data structure (or virtual device), which is sent a byte stream from a data source (for example, a program) and which delivers the byte stream to a data sink (for example, a printer).
- Process** : A program or task or thread that has a distinct memory allocation of its own and has one or more functions or procedures for a specific job. The process may share

- the memory (data) with other tasks. A processor may run multiple processes separately or concurrently as directed by the OS.
- Processor** : A processor executes the instruction cycles and executes one process or many as per the command (instruction) given to it.
- Pulse Width Modulator (PWM)** : A modulator that modulates the pulse width as per the input bits. It provides a pulse of width scaled to the analog output desired. On integrating PWM output the desired DAC operation is achieved.
- Real-time operating system** : Operating system software for real-time programming and scheduling, process and memory manager, device drivers, device management and multitasking.
- RAM** : This is a random access read and write memory that the processor uses to store programs and data that are volatile and which disappear on power down or when switched off.
- Registers** : These are associated with the processor and temporarily store the variable values from the memory and from the execution unit during processing of an instruction.
- Reset** : A processor state in which the processor registers acquire initial values and from which starts an initial program; this program is usually the one that also runs on power up.
- Reset circuit** : A circuit to force a reset state that gets activated for a short period on power up. When reset is activated, the processor generates a reset signal for the other system units needing reset.
- ROM** : A read only memory that locates the following in it—embedded software, initial data and strings and operating system or RTOS.
- System** : A way of working, organizing or doing one or a series of tasks by following a fixed plan, program and set of rules.
- System on Chip** : A system on a VLSI chip that has all the necessary needed analog as well as digital circuits; for example, in a mobile phone.
- Timer** : A unit to provide the time for the system clock and real-time operations and scheduling. It generates interrupts on timeouts as per the preset time or on overflow or on successful comparison of present time with a preset time or on capturing the time on an event.
- Touchscreen** : An input as well as an output device that is used to enter a command, choose a menu or to give user reply as input by physically touching at a screen position either by the finger or by a stylus. A stylus is thin pencil-shaped object. It is held between the fingers and used just as a pen is used to mark a dot. The screen displays the choices or commands, menus, dialog boxes and icons, similar to a computer display.
- UART** : Universal Asynchronous Receiver and Transmitter.
- Virtual Device** : A file or socket or pipe-like device that is programmable for opening, closing, reading and writing similar to a physical device.
- VLSI chip** : A very large-scale integrated circuit made on silicon with $\sim 10^6$ and above transistors.
- Watchdog timer** : A timer that resets the processor in case the program gets stuck for an unexpected length of time.



Review Questions

1. Define a system. Now define an embedded system.
2. What are the essential structural units in the following? (a) a microprocessor (b) an embedded processor (c) a microcontroller (d) a DSP (e) an ASIP. List each of these.
3. How does a DSP differ from a general-purpose processor (GPP)? [Sections 1.2.1, 1.7.3 and 2.3.3].
4. What are the advantages and disadvantages of the following? (a) a processor with only a fixed-point arithmetic unit and (b) a processor with additional floating-point arithmetic processing unit.
5. How does a microcontroller differ from a DSP? [Sections 1.2.3, 1.7.2, 2.1 and 2.3.5].
6. Explain single purpose processors use in convergence technology embedded systems (a) smart mobile phone with mail client, Internet connectivity and image-frame downloads and (b) digital camera.
7. Compare features in a family chip (or core) of each of the following: a microprocessor, microcontroller, RISC processor, DSP and ASSP.
8. Why do later generation systems operate the processor at low voltages (≤ 2 V) and perform IOs at (~ 3.3 V)?
9. What are the techniques of power and energy management in a system?
10. What is the advantage of running a processor at reduced clock speed in certain sections of instructions and at full speed in other sections?
11. What is the advantage of the following? (a) Stop instruction (b) Wait instruction (c) Processor idle mode operation (d) Cache-use disable instruction (e) Cache with multiways and blocks in an embedded system.
12. What do we mean by charge pump? How does a charge pump supply power in an embedded system without using power-supply lines?
13. What do you mean by 'real time' and 'real time clock'?
14. What is the role of processor reset and system reset?
15. Explain the need of watchdog timer and reset after the watched time.
16. What is the role of RAM in an embedded system?
17. Why do we need multiple actions and multiple controlling tasks for devices in an embedded system? Explain, using as an example the embedded system, the remote of colour TV.
18. When do we need multitasking OS?
19. When do we need an RTOS?
20. Why should the embedded system RTOS be scalable?
21. Explain the terms IP core, FPGA, CPLD, PLA and PAL.
22. What do you mean by System-on-Chip (SoC)? How will the definition of an embedded system change with a System-on-Chip?
23. What are the advantages offered by an FPGA for designing an embedded system?
24. What are the advantages offered by an ASIC for designing an embedded system?
25. What are the advantages offered by an ASIP for designing an embedded system?
26. Real time video processing needs sophisticated embedded systems with hard real time constraints. Why? Explain.
27. Why does a processor system always need an 'Interrupts Handler (Interrupt Controller)'?
28. What role does a linker play?
29. Why do we use a loader in a computer system and a locator in an embedded system?
30. Why does a program reside in the ROM in the embedded system?
31. Define ROM image and explain each section of an ROM image in a system.
32. When is the compressed program and data in ROM used? Give five examples of embedded systems having these in their ROM images.
33. When is SRAM used and when DRAM? Explain your answers.
34. What do we mean by the following: physical device, virtual device, plug and play device, bus self-powered device, device management and device-specific processor.

35. Define design metrics in embedded systems. What are the different competing design metrics? What are the constraints of embedded system design?
36. How is power dissipation optimized?
37. What are the challenges faced in designing an embedded system?



Practice Exercises

38. Search for the definitions of embedded system in the books referred to in the 'References' section and tabulate these with the definitions in column 1 and the reference names in column 2.
39. Classify the embedded systems into small scale, medium scale and sophisticated systems. Now, reclassify these embedded systems with and without real-time (response time constrained) systems and give 10 examples of each.
40. An automobile cruise control system is to be designed in a project. What will be skills needed in the team of hardware and software engineers?
41. Take a value, $x = 1.7320508075688$. It is squared once again by a floating-point arithmetic processor unit. Now x is squared by a 16-bit integer fixed point arithmetic processing unit. How does the result differ? [Note: Fixed-point unit will multiply only 17320 with 17320, divide the result by 10000 and then again divide the result by 10000.]
42. Design a four-column table. Write two examples of embedded systems in columns 2 and 3. In Column 1, write the type of processor needed among the following: microprocessor, microcontroller, embedded processor, digital signal processor, ASSP, single purpose and media processor. Give your reasoning in column 4.
43. Why does a CMOS IO circuit power dissipation reduce by compared to 5 V, factor of half, $\sim(3.3/5)^2$, in IO 3.3 V operation?
44. What will be the reduction in power dissipation for a CMOS circuit when voltage reduces from a 5 V to a 1.8 V operation?
45. List the various type of memories and application of each in the following: robot, electronic smart weight display system, ECG LCD display-cum-recorder, router, digital camera, speech processing, smart card, embedded firewall/router, mail client card, and transceiver system with collision control and jabber control. [Collision control means transmission and reception when no other system on the network is using the network. Jabber control means control of continuous streams of random data flowing on a network, which eventually chokes a network.]
46. Tabulate hardware units needed in each of the systems: camera, mobile computer and robot.
47. Give two examples of embedded systems, which need one or more of following units. (a) DAC (Using a PWM) (b) ADC (c) LCD display (d) LED displays (e) Keypad (f) Pulse dialer (g) Modem (h) Transreceiver.
48. An ADC is a 10-bit ADC. It has reference voltages, $V_{ref-} = 0.0$ V and $V_{ref+} = 1.023$ V. What will be the ADC outputs when inputs are (a) -0.512 V (b) $+0.512$ V and (c) $+2.047$ V? What will be the ADC outputs in three situations when (i) $V_{ref-} = 0.512$ V and $V_{ref+} = 1.023$ V (ii) $V_{ref-} = 1.024$ V and $V_{ref+} = 2.047$ V and (iii) $V_{ref-} = -1.024$ V and $V_{ref+} = +2.047$ V.
49. Tabulate the advantages and disadvantages of using coding languages as follows: (a) Machine coding (b) Assembly (c) C (d) C++ and (e) Java.
50. List the software tools needed in designing each of the embedded system—camera, mobile phone and robot.
51. Justify the use of physical and virtual devices drivers in embedded systems.
52. The cost of designing an embedded system may be thousands of times the cost of its processor and hardware units. Explain this statement.
53. An FPGA (Field Programmable Gate Arrays) core integrated with a single or multiple processor unit on chip. How do these help in the design of sophisticated embedded systems for real time video processing?
54. List the memory units and processor needed in a smart card.

8051 and Advanced Processor Architectures, Memory Organization and Real-world Interfacing

2

\mathcal{R}

e

c

a

l

l

The previous chapter dealt with the following:

- Embedded system embeds software and RTOS.
- Embedded system embeds software in hardware consisting of microprocessor or microcontroller or DSP or ASIP and single purpose processors.
- Embedded system has memory (ROM, RAM and caches), ports, timers, devices and interfacing circuits.
- Microcontroller hardware consists of processing unit, RAM, ROM, timing and interrupt handling devices and other application specific units as a single chip or VLSI core.
- Design metrics, processes and challenges.
- Software, device drivers and device-manager.
- Software tools.



L
E
A
R
N
I
N
G
O
B
J
E
C
T
I
V
E
S

In this chapter, we will learn the following

1. 8051 architecture in brief and its processor, memory, ports, counters/timers, serial IO and interrupt handler units
2. Real world interfacing, and internal and external buses that interconnect the processor with the system memories, IO devices and all other system units
3. Interfacing examples with keyboard, displays, ADC and DACs
4. Advanced processors x86, ARM and SHARC architectures
5. Processor and memory organization
6. Instruction-level parallelism and superscalar, processing, pipelining and cache units for improved computational performance of the processor by faster program execution
7. Various types of memory and their uses
8. Devices and memory addresses allocations
9. Performance metrics to measure the performance of a processor
10. Processor selection for embedded system
11. Memory selection for embedded system

2.1 8051 ARCHITECTURE

The following subsections summarize the 8051 architecture in brief. A reader may refer to a standard text for details.

2.1.1 8051 Microcontroller Architecture

Figure 2.1 shows the architecture of the classic 8051 microcontroller. Classic means the original version, based upon which new enhancements and versions are provided. The classic version consists of following hardware:

1. A 12 MHz clock. Processor instruction cycle time is 1 μ s.
2. An 8-bit ALU. The internal bus width is 8-bit.
3. CISC (Complex Instruction Set Computer) architecture. [CISC provides many modes for addressing operands in arithmetic, logical and other instructions. Several complex instructions take more than one cycle time. Complex instructions implement in hardware not by separate hardwired logic circuits for each instruction but by a microprogram control circuit.]
4. Special bit manipulation instructions.
5. A program counter, in which the initial default reset value defined by the processor is 0x0000.
6. A stack pointer, in which the initial default value defined by the processor is 0x07.

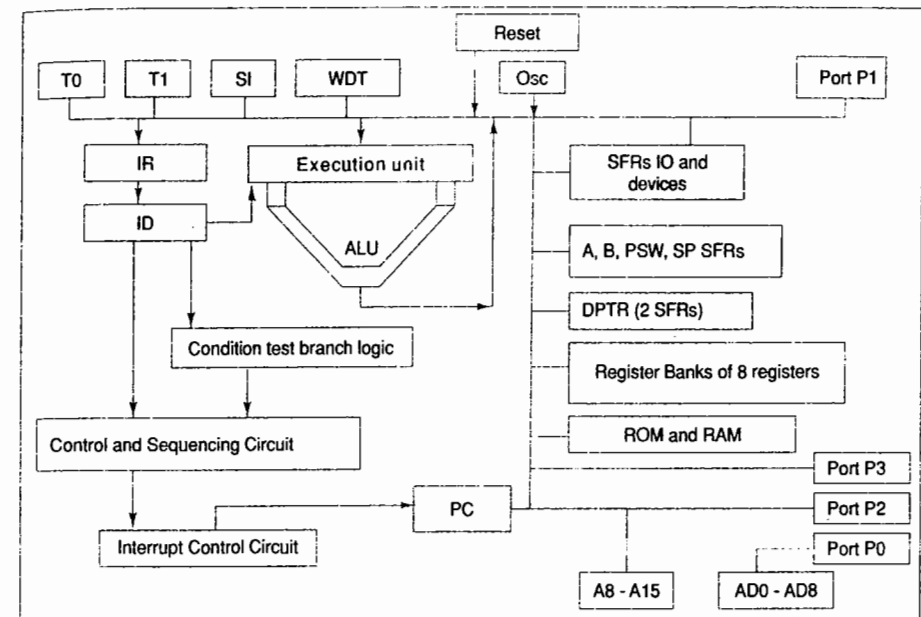


Fig. 2.1 8051 Architecture

7. A simple architecture, with no floating-point processor, no cache, no memory management unit, no atomic operations unit, no pipeline and no instruction level parallelism. (Sections 2.3 and 2.5). There is no DMA controller (Section 4.8) in the classic and most other versions.
8. A Harvard memory architecture (Section 2.4.2). The program memory and data memory have separate address spaces from 0x0000 and separate control signal(s).
9. On-chip RAM of 128 bytes. The 8052 version provides for RAM of 256 bytes; 32 bytes of RAM are also used as four banks (sets) of registers. Each register-set (bank) thus has eight registers. The external data/stack memory can be added upto 64 kB in most versions. In certain 8051 enhancements, this limit has been enhanced to 16 MB.
10. There are special function registers (SFRs). These are PSW (processor status word), A (accumulator), B register, SP (stack pointer) and registers for serial IOs, timers, ports and interrupt handler.
11. 8351 version has on-chip ROM; 8751 version EPROM; 8951 version has on-chip EEPROM or flash memory of 4 kB. Several versions provide for higher capacity ROM. Additional program memory can be added externally upto 64 kB. In extended 8051 and unified address space versions (8051 EX and MX versions), this limit has been extended to 16 MB.
12. Two external interrupt pins, INT0 and INT1.
13. Four ports P0, P1, P2 and P3 of 8 bits each in single chip mode. (Section 2.1.3) There are two timers (Section 2.1.5) and a serial interface (SI). It can be programmed for three full duplex UART modes for a serial IO. [IO with each bit of a word successive transmitted on the data line for a time interval.] The SI can also be programmed for half duplex synchronous IO (Section 2.1.6).

14. Classic version has no pulse width modulator and provides on support to DAC. (Section 1.3.7) It has no modem, no watchdog timer, no ADC. Certain versions support watchdog timer and ADC. Siemens SAB 80535-N supports ADC with programmable reference voltage. Advanced versions support these features and choice of version depends on system requirement. (Section 2.8 and 2.9).

2.1.2 Instruction Set

Figure 2.2 shows instruction types in the 8051 set. There are seven types of instructions.

Full instruction set and instructions in detail can be referred to from a microcontroller text or manual. The important instructions are as follows:

Data Transfer Instructions Data transfer instructions move (copy) one source operand to another destination. $MOV A, R_n$ and $MOV R_n, A$ are for moving (copying) the data into A from R_n and to R_n . R_n is the n^{th} register in a set of 8 registers. PSW bits RS0 and RS1 predefine the set.

Data transfer instructions $MOV A, @R_i$ and $MOV @R_i, A$ are for moving (copying) the data into A from $@R_i$ and to $@R_i$. R_i is the i^{th} register in a set of 8 registers. PSW bits RS0 and RS1 predefine the set. $@R_i$ means data transfer to an 8-bit address pointed by the contents of the i^{th} register in the set.

Four data transfer instructions are $MOV direct, \#data$, $MOV A, \#data$, $MOV R_n, \#data$ and $MOV @R_i, \#data$ for moving 8-bit data into direct or A or R_n or $@R_i$. Direct means data transfer to an 8-bit address of internal 128B RAM or SFR address. $@R_i$ means data transfer to an 8-bit address pointed by the contents of the i^{th} register in the set ($i = 0$ or 1).

Seven data transfer instructions are $MOV direct, direct$, $MOV A, direct$, $MOV direct, A$, $MOV direct, R_n$, $MOV R_n, direct$, $MOV direct, @R_i$, and $MOV @R_i, direct$ are for moving data between the 8-bit direct address to direct or A or R_n or $@R_i$. Direct means data transfer to or from an 8-bit address of internal 128B RAM or SFR address. $@R_i$ means data transfer to an 8-bit address pointed by contents of i^{th} register in the set ($i = 0$ or 1).

There is a 16-bit external memory data pointer, DPTR. The $MOV DPTR, data16$ instruction is used to send 16 bits specified in data16 to DPTR.

MOVX (move external instruction) will transfer the data to or from external data memory. These instructions are $MOVX A, @DPTR$ and $MOVX @DPTR, A$. $@DPTR$ means address as pointed by 16 bits of DPTR. For an 8-bit external memory address, instead of DPTR, $@R_i$ is used ($i = 0$ or 1). MOVC (move code from external program memory instruction) will transfer the data from the external program memory. Instructions are $MOVC A, @A + DPTR$ and $MOVC A, @A + PC$.

For stack operations, there are $PUSH direct$ and $POP direct$ instructions.

Bit Manipulation Instructions Each bit of certain SFRs and an 8-byte internal RAM are assigned bit addresses in 8051 hardware. There are bit manipulation instructions to clear, set, AND, OR, MOV the bit.

Byte Manipulation Instructions There are byte manipulation instructions to rotate right A, rotate left A, rotate right A with carry, rotate left A with carry, complement A, clear A and swap with A lower and upper nibble (set of 4 bits).

Arithmetic Instructions Arithmetic instructions of the source operand is stored in the accumulator and the result of the operation is also stored in the accumulator. For example, $ADD A, R_n$. It adds the byte in A with the byte in the n^{th} register. ($A \leftarrow A + R_n$). Three arithmetic instructions are ADD, ADDC (add including carry bit) and SBBB (subtract including borrow bit). Carry bit is set to 1 when an operation results in carry or

borrow. Borrow is saved in the carry bit. The second operand can be R_n , direct, $@R_i$ or $\#data$ ($i = 0$ or 1). The meaning of these is the same as in case of data transferred instructions (explained earlier). There is also an instruction to adjust hexadecimal addition to decimal addition.

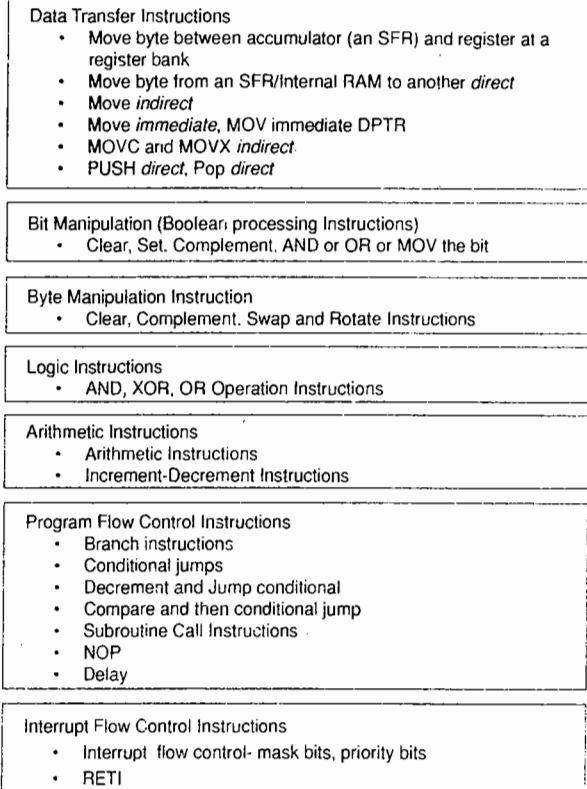


Fig. 2.2 Instruction types in 8051 instruction set

INC and DEC instructions increase (by 1) and decrease (by 1) the bits at the source. The source can be R_n , direct, $@R_i$ or A.

MUL AB and DIV AB are used to find $A - B \leftarrow A \times B$ and $A - B \leftarrow A \div B$. The multiplication results in lower byte in A and higher in B. Division results quotient is stored in A and the remainder in B.

Logic Instructions Logic instructions one of the source operand is accumulator or direct and result of operation is also in the accumulator or direct. For example, $ANL A, R_n$ and $ANL direct, R_n$. It logical ANDs the byte in A or direct with the byte in n^{th} register in the register set. ($A_i \leftarrow A_i \wedge R_n$). Three logic instructions are ANL (AND logic), ORL (OR logic) and XRL (XOR logic). The second source operand is $@R_i$ or R_n or direct when the first source cum destination is A, and is A and $\#data$ when the first source cum destination is *direct*. [Meaning of these are the same as in the case of the data transfer instructions (explained earlier).]

Program Flow Control Instructions Program flow control is done by instructions for short jump, absolute jump or long jump or jump. Short jump instruction is to a relative address within -128 and $+127$. Absolute jump instruction is to direct 11-bit address in program memory. Long jump instruction is to direct 16-bit address in program memory. Jump instruction is pointed by $A + @DPTR$.

Conditional program flow control instructions are also present. Loop control instructions are also present in which jump count value is in R_n or direct, and offset. Both are specified in the loop control instruction.

Program flow control in a subroutine call is by absolute call or long call instruction. Absolute call instruction is to call a specified direct 11-bit address at program memory. Long call instruction is to call a specified direct 16-bit address directly at program memory.

Return from a called routine is by RET instruction and return from interrupt service routine is by RETI.

2.1.3 IO Ports, Circuits and IO Programming

Figure 2.3(a) shows P0, P1, P2 and P3 IO ports in 8051. 8051 in single chip mode has four ports. Single chip means there are no external memory chips or ports or serial port or peripheral attached to the port. Section 2.1.4 will give an expanded mode circuit.

Port driving capabilities depend on the specific version of 8051. P0 is an 8-bit open drain bidirectional IO port and P1 to P3 are quasi bidirectional IO ports. Open drain port means that the output port pins need a pull up circuit or resistance to raise the voltage level to logic 1. A quasi bidirectional IO port can drive for two clock cycles eight logic LSTTL gates in 8051. For higher driving capability, pull up circuits will be required. Port P1 bits are open drain in P83C538 version as two bits P1.6 and P1.7 are used for I²C bus (Section 3.10.1) clock and data signals.

IO Port Circuits Sections 2.2.6 and 3.3 will describe the interfacing circuit of port IO bits to switches, keypad, encoders, motors and LCD controllers. Figures 2.3(b) and (c) show IO port P1 circuits for two stepper motors in a printer and six servomotors in a robot. IO port bytes and bits are programmed and accessed as follows:

- (i) **IO Byte Programming** The internal IO ports P0, P1, P2 and P3 in the 8051 have byte addresses to access and perform read, write or other operations. These addresses are the direct 8-bit addresses of each that are specified in the instructions. Addresses of bytes at P0, P1, P2 and P3 have 0x80, 0x90, 0xA0 and 0xB0. All instructions in the instruction set using *direct* addresses can be used to access and perform read or write operations on the ports.
- (ii) **IO Port Bit Programming** Each port P0, P1, P2 and P3 has 8 bits and each bit has addresses to access and perform read or write operations using bit-manipulation instructions. These addresses are the *bit* address of 8 bits, which are specified in the instructions. Bits P0.0 to P0.7 have addresses 0x80 to 0x87. Bits P1.0 to P1.7 have addresses 0x90 to 0x97, P2.0 to P2.7, 0xA0 to 0xA7 and P3.0 to P3.7 0xB0 to 0xB7. All instructions in the instruction set using *bit* addresses can be used to access and perform complement or read or write operations. The C flag in PSW is the accumulator for logic operations on the bits using bit-addresses.

Example 2.1

1. MOV 0xA0, #0xFF will move bits to port P2 and P2 bits will become = 11111111_b.
2. MOV 0x90, #0x1C will move bits at port P1 = 00011100_b. After this instruction, INC 0x90 will make P1 = 00011100_b + 1 = 00011101_b.

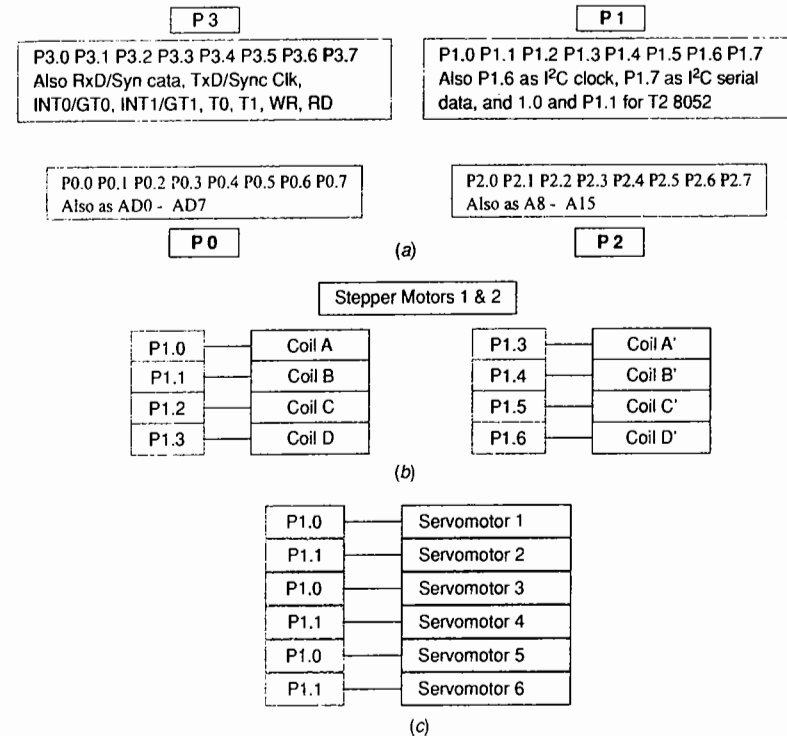


Fig. 2.3 (a) IO ports in 8051 (b) IO port P1 circuit for two stepper motors in a printer (c) IO port circuit for six servo motors in a robot

Example 2.2

1. CPL 0x90 will complement the bit 0 at port P1.
2. CLR 0x80 will make P0.0 as 0. Now after a delay of period = T₁, the SETB 0x80 will make P0.0 as 1. Now after a delay of period = T₂, the CLR 0x80 will make P0.0 as 0. A pulse of time-period T₂ and duty cycle $100 \frac{T_1}{T_1 + T_2}$ creates if the instructions are executed in a loop.
3. SETB C will set carry bit in PSW to 1. After this operation, ANL C, 0x93 will perform logic AND operation between bits C and P1.3 and result will be in C. If P1.3 = 0 then C will become 0 else C will remain 1.
4. CLR C will reset (clear) carry bit in PSW to 0. After the operation, ORL C, 0x92 instruction will perform logic OR operation between bits C and P3.2 and result will be in C. If P3.2 = 0 then C will remain 0 else C will remain 1. After the operation, MOV 0x85, C instruction will move result in C to P0.5.

2.1.4 External Memory Interfacing Circuits

Figure 2.4(a) shows how to connect the external program and data memory circuits in 8051. There are two sets of memory, program memory and data memory. The processor has two control signals \overline{PSEN} and \overline{RD} to control read from program or data memory. The processor has a control signal \overline{ALE} to control use of $\overline{AD0-AD7}$ as address or data at a given instance. Section 2.2.1 will explain \overline{PSEN} , \overline{RD} and \overline{ALE} control signals.

1. Port P0 is used in expanded mode as $\overline{AD0-AD7}$. $\overline{AD0-AD7}$ are the multiplexed signals of the $A0-A7$ lower address bits of the address bus and $D0-D7$ bits of data bus. $A0-A7$ and $D0-D7$ are time division multiplexed. For an interval the processor activates \overline{ALE} (address latch enable) in an instruction cycle and the $\overline{AD0-AD7}$ lines have $A0-A7$; a latch-circuit separates $A0-A7$ signals for the memory.
2. Port P2 has $A8-A15$ address signals. When the processor activates \overline{PSEN} (Program store enable), it reads the byte from the external program memory through the $D0-D7$ data bus. When the processor activates \overline{RD} (read), it reads the byte from the external data memory through the $D0-D7$ data bus.

For addresses outside the internal RAM, SFR and internal program memory, the processor always accesses the external memory. That is irrespective of external enable \overline{EA} active or inactive. Internal RAM and SFR addresses between $0x00$ and $0xFF$ are the same as external memory addresses $0x0000$ and $0xFFFF$. Internal program memory addresses between $0x0000$ to $0xFFFF$ (in case of 4 kB internal ROM) are the same as the external program memory addresses $0x0000$ and $0xFFFF$. When a control signal \overline{EA} activates, the processor accesses the external addresses in the memory instead of these internal memory or register addresses.

The 8051 has a memory mapped IO (Section 2.2.2). Memory and ports are assigned addresses such that each has a distinct range of addresses in the data memory address space. Therefore, interfacing circuit design is identical to that for the memory and connects the external ports and programmable peripheral interface (PPI). Memory and ports are assigned the addresses such that each has a distinct range of addresses. A PPI chip is 8255. Figure 2.4(b) shows the interfacing when using the external PPI ports PA, PB and PC.

2.1.5 Counters and Timers

Figure 2.5 shows the specifications of counters and timers, T0 and T1 in 8051. There are two timing and counting devices T0 and T1 in classic 8051 and three T0, T1 and T2 in 8052. Using the two SFRs, TH1-TL1, the counts at the higher and lower 8 bits of T1 are accessed. The SFRs hold the T1 device 16 bits. Using two SFRs, TH0-TL0, the higher and lower 8 bits of T0 are accessed. The SFRs hold the T0 device 16 bits.

An SFR called TMOD controls the T1 and T0 modes using the upper and lower 4 bits each, which programs the counting/timing of T1 and T0. A bit for each controls whether the external gate input controls or not. Another bit controls whether counter or timer mode is used. Two other bits control the functional mode of timer/counter as mode 0 or 1 or 2.

The counting/timing device records time when inputs are given by the clock. The clock pulses are internally given at the specific time intervals in case of functioning as timer. It also records counts when the inputs are given externally. The counter is given the input to count from external input pins.

1. When timing or counting devices are externally controlled by the gate inputs, when $\overline{GT0}$ or $\overline{GT1}$ is externally activated, the device can function; else, it deactivates in gate input mode. $\overline{GT0}$ or $\overline{GT1}$ signals are given through the P3.2 and P3.3.
2. When T0 is in counter mode, it is given the input to count from the external input pin T0 at P3.4. When T1 is in counter mode, it is given the input to count from the external input pin T1 at P3.5.

The upper four 4 bits of an SFR, called the TCON, programs the counting/timing device in T1 and T0 modes. TCON.7 and TCON.5 show the timer/counter overflow status for T1 and T0, respectively. TCON.6

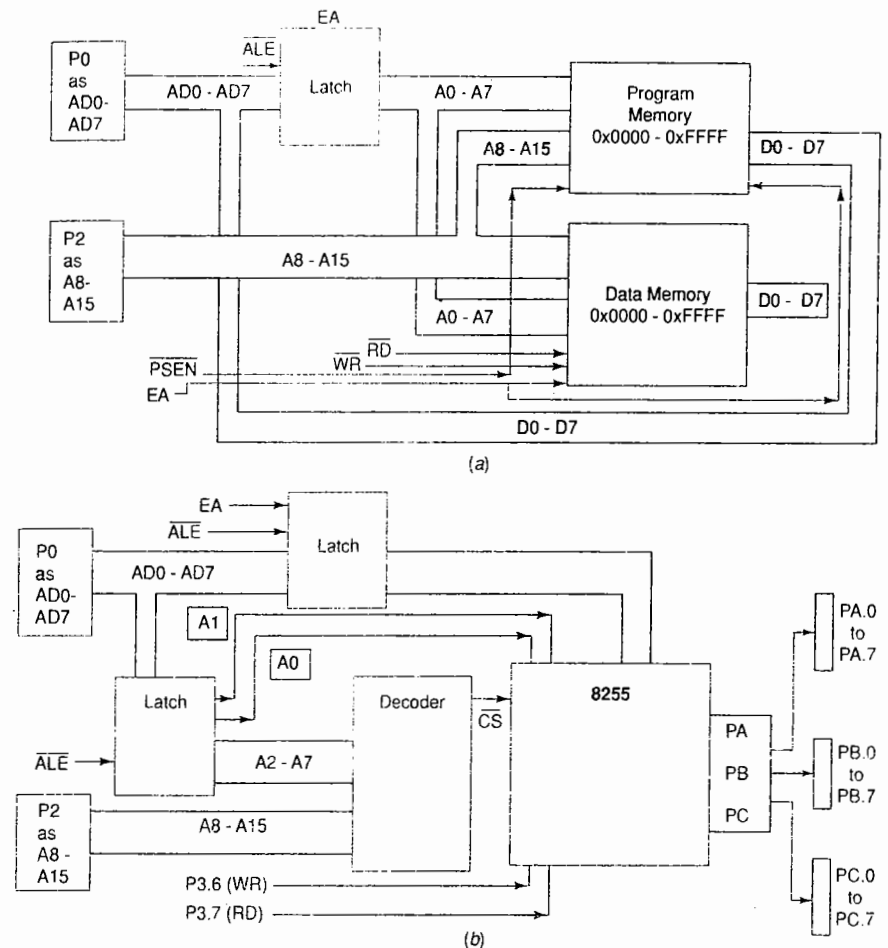


Fig. 2.4 (a) Connection of 8051 to external program and data memory circuits
(b) Interfacing of 8051 to external PPI 8255 ports PA, PB and PC

and TCON.4 control the start and stop of the timer/counter overflow status for T1 and T0, respectively. The lower bits of the TCON are used for interrupt control for INT0 and INT1 (Section 2.1.7).

2.1.6 Serial Data Communication Input/Output

Figure 2.6 shows serial ports and data serial communication using SI (serial interface) in 8051.

There are two SI modes: half duplex synchronous and full-duplex asynchronous. Half duplex means one-way communication and full duplex means both ways at the same instance.

P.3

P3.2, P3.3, P3.4 and P3.5 as GT0 (gate for starting/stopping T1), GT1 (gate for starting/stopping T1), T0 (count input to T0) and T1 (count input to T1) inputs, respectively when TMOD bits 3, 7, 2 and 6 are set to

Timer/Counter T0

- 8-bit SFRs used are **TMOD** (lower 4 bits), **TCON** (bit 5 and 4), **TL0** (count/time bits), **TH0** (count/time bits)
- Counter with inputs at P3.4 when bit 2 at TMOD=1, timer with internal clock timed inputs when bit 2 at TMOD=0
- When mode set = 0, 8-bit timer/Counter mode. TH0 is used as T0 and TL0 is used for prescaling (dividing) count or clock inputs by 32
- When mode set = 1, 16-bit timer/counter mode with TH0-TL0 is used for timing or counting using T0
- When mode set = 2, 8-bit timer/Counter. TH0 is used as T0 and TL0 is used for auto-reloading the TH0 after timeout using a preset value at TL0
- When mode set = 3, two 8-bit timer/Counters mode TH0 and TL0 are independent 8-bit timer/counters and T1 does not function

Timer/Counter T1

- 8-bit SFRs used **TMOD** (upper 4 bits), **TCON** (bit 7 and 6), **TL1** (count/time bits), **TH1** (count/time bits)
- Counter with inputs at P3.5 when bit 6 at TMOD=1, timer with internal clock timed inputs when bit 6 at TMOD=0
- When mode set = 0, 8-bit timer/Counter mode and TH1 is used and TL1 is used for prescaling (dividing) inputs by 32
- When mode set = 1, 16-bit timer/counter mode with TH1-TL1 is used for timing or counting
- When mode set = 2, 8-bit timer/Counter TH1 is used and TL1 is used for auto-reloading the TH1 after timeout using a preset value at TL1
- When mode set = 3, T1 stops as TH0 now functions as independent 8-bit timer in place of T1

Fig. 2.5 Counter-cum-timers T0 and T1 in 8051

P.3

P3.0 and P3.1 as pins for Rx/D and Tx/D UART mode serial input and output, or synchronous serial mode data and clock inputs, or synchronous serial mode data and clock outputs, respectively

Serial Interface SI (programmable for half duplex synchronous serial or full duplex asynchronous UART modes)

- 8-bit SFRs used are **SBUF** (8-serial received bits or transmission bits register depending upon instruction is using SBUF as source or destination), **SCON** (8-serial modes cum control bits register) and SFR **PCON** 7th bit are used
- Synchronous serial mode data and clock inputs, or synchronous serial mode data and clock outputs depending upon instruction is using SBUF as source or destination when SCON bits 7 and 6 are 00 (mode 0)
- 10-bit (start plus 8- serial data plus stop total 10 bits) UART mode serial input and output with programmable baud rate using T1 or T0 timers (T2 in 8052) when SCON bits 7 and 6 are 01 (mode 1)
- 11-bit (start plus 8- serial data plus RB8 or TB8-bit plus stop total 11 bits) UART mode serial input and output with fixed baud rate of $(f/32)+12$ or $(f/64)+12$ Mbaud/s where f = crystal frequency. The rate depends upon PCON 7-bit SMOD = 1 or 0, respectively when SCON bits 7 and 6 are 10 (mode 2)
- 11-bit (start plus 8- serial data plus RB8 or TB8-bit plus stop total 11 bits) UART mode serial input and output with programmable baud rate using T1 or T0 timers (T2 in 8052) when SCON bits 7 and 6 are 11 (mode 2)

Fig. 2.6 Serial ports and data serial communication using SI (serial interface) in 8051

Using the single SFR for transmit and receive byte buffers, the serial output or input is sent. The SFRs hold the SI transmission 8 bits when it is written. 0x99 is the address of SI buffers. For example, MOV 0x99, A instruction writes into transmission buffer from the A register and MOV R1, 0x99 instruction reads the R1 register from the receive buffer.

An SFR called SCON controls the SI interface. Three upper bits program the modes as 0, 1, 2 or 3. Mode 0 is half duplex synchronous. Modes 1 or 2 or 3 are full duplex asynchronous. Mode 2 transmits and receives in 11 T format and mode 1 in 10T format. T is the interval between successive transmitted or received bits and T^{-1} is the baud rate. (Section 3.2.3 gives the details). Modes 1 and 3 are for programmable baud rate and 2 for fixed baud rate.

A bit SCON.4 enables or disables SI receiver functions. Two bits SCON.3 and SCON.2 specify the 8th bit to be transmitted and 8th bit received when the mode is 2 or 3. A bit SCON.1 enables or disables SI transmitter interrupts (TI) on completion of transmission. A bit SCON.0 enables or disables SI receiver interrupts (RI) on completion of transmission.

2.1.7 Interrupts in 8051

Figure 2.7 shows the specifications of system of interrupts in classic 8051. There are multiple interrupts in 8051. When an interrupt is enabled (not masked), then on occurrence of that interrupt event, an ISR is called.

P.3

P3.2 and P3.3 as pins for INT0 and INT1 external interrupt pins when bit 7 at IE (interrupt enable SFR) EA (enable all) bit is 1, and bits 0 and 2 are 1 and 1, respectively.

Interrupt Sources

- 8-bit SFRs used are **IE** (one interrupt enable all EA bit to enable interrupts and remaining enable individual interrupts bits) and **IP** (individual interrupt priorities set high or low bits)
- External INT0 interrupt
- T0 overflow interrupt
- External INT1 interrupt
- T1 overflow interrupt
- SI serial UART or Synchronous mode interrupt
- SI synchronous serial mode interrupt (separate in few families of 8051)
- Timer 2 interrupt in 8052

Vector Address from where either the 8-byte ISR executes or a Jump to the programmed ISR starting address takes place in case EA bit is set as well as specific interrupt bit is set

INT0	0x0003
T0	0x000B
INT1	0x0013
T1	0x001B
Serial	0x0023
T2	0x002B
Syn Serial	0x0053 in few versions

Default Priorities by hardware (Software assigned high priority setting in IP overrides the default)

In-between executing low priority ISR permits the program flow on interrupt to higher priority ISR: YES

Fig. 2.7 Interrupts in 8051 architecture

SI transmission or receiver interrupts and synchronous mode interrupts occur when SI is programmed using SCON. There are timer T1 and T0 overflow interrupts when T1 and T0 are programmed using TMOD and TCON. There are two external pins for interrupts from peripherals or external circuits.

Two external interrupt pins, INT0 and INT1 at P3.2 and P3.2 can interrupt provided these pins are programmed by the TCON lower 4 bits and the IE register bits IE.2 and IE.0.

8051 hardware sets default priorities for service in the case when multiple interrupts occur concurrently. Priorities by default are in the order INT0, T0 overflow, INT1, T1 overflow, SI (UART mode), T2 (in 8052) and SI (synchronous mode). Using the SFR, called IP (Interrupt Priority) register, at address 0xB8 for the byte and at addresses 0x88 to 0x8C, 0x8D, or 0x8E for the individual bits in the register, an instruction can define that a given interrupt is of higher (=1) or lower priority (=0) among the various interrupts in 8051. [Section 4.6.3]

Using the SFR IE (Interrupt Enable) register at address 0xA8 for bytes and at addresses 0xA8 to 0xAF for the individual bits, a program enables or disables the interrupts (Section 4.4.3).

TCON.3 shows the status of the interrupt at INT1 and auto resets to 0 when the ISR for servicing INT1 interrupt starts. TCON.1 shows the status of the interrupt at INT0 and auto resets to 0 when the ISR for servicing INT0 starts. TCON.2 shows the type of interrupt at INT1 and is 1 if it is of the edge-triggered type, else 0. TCON.0 shows the type of interrupt at INT0 and is 1 if it is edge-triggered type, else 0.

8051 has fixed interrupt vector addresses (Section 4.4.1). An 8-byte address space is provided between two vector addresses. An ISR (Section 4.2) is stored either within these addresses or another ISR is called from these addresses if the ISR is long.

2.2 REAL WORLD INTERFACING

2.2.1 System Bus-based and IO Bus-based IOs for Real World Interfacing

Figure 2.8 shows the interconnections for a simple bus structure when interfacing the processor, memory and IO devices. Three sets of signals – classified as address bus, data bus and control bus defines the system bus. The characteristics of the processor's internal bus(es) differ from that of the system's external bus(es). A system-bus interfacing-design is created according to the needs of the processor signal's timing diagram, speed and the word length for instructions and data.

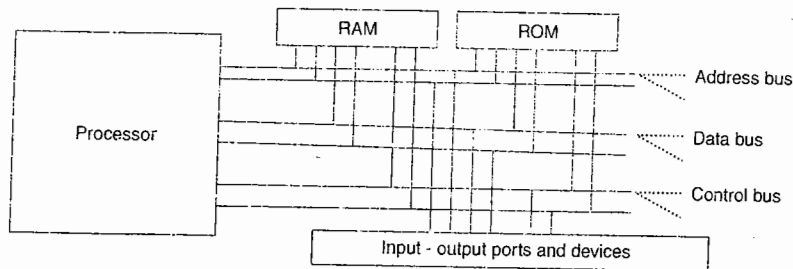


Fig. 2.8 Interconnections for a simple bus structure when interfacing the processor, memory and IO devices using system bus

Address Bus The processor issues the address of the instruction byte or word to memory system through the address bus. The processor execution unit, when required, issues the address of data (byte or word) to memory system through address bus. An address bus of 32 bits fetches instruction or data from an address specified by a 32-bit number.

Example 2.3

1. Let a processor at the start reset the program counter at address 0. Then the processor issues address 0 on the bus and the instruction at address 0 is fetched from memory.
2. Let a processor instruction be such that it needs to load register r1 from the memory address M. The processor issues address M on the address bus and data at address M is fetched.

Data Bus When the processor issues the address of the instruction, it gets back the instruction through the data bus. When it issues the address of the data, it loads the data through the data bus. When it issues the address of the data, it stores the data in the memory through data bus. A data bus of 32 bits fetches, loads, or stores the instruction or data of 32 bits.

Example 2.4

1. When the processor issues address m for an instruction, it fetches the instruction through data bus from address m. [For a 32-bit instruction, word at data bus is from addresses m, m + 1, m + 2 and m + 3.]
2. When an instruction is given to store register r1 to the memory address M, the processor issues address M on the bus and sends the data at address M through the data bus. [For 32-bit data, word at data bus is to the memory addresses M, M + 1, M + 2, and M + 3.]

Control Bus A control bus issues signals to control the timing of various actions during interconnection. These signals synchronize the subsystems. There may be the following:

address latch enable [(ALE) Address Strobe (AS) or address valid (ADV)], memory read (RD) or write (WR) or IO [read (IORD) or write, (IOWR) or data valid (DAV)], interrupt acknowledge (INTA) on a request for drawing the processor's attention to an event, or hold acknowledge (HLDA) on an external hold request for permitting use of the system buses, and other control signals as per processor design. Input control signals may be INTR (Interrupt) when external device interrupts the system and HOLD when external device sends a hold request for direct memory access (DMA).

Example 2.5

1. When the processor issues the address, it also issues a memory-read control signal and waits for the data or instruction. A memory unit must place the instruction or data during the interval in which memory-read signal is active and not inactivated by the processor.
2. When the processor issues the address on the address bus, and (after allowing sufficient time for the all address bits setup) it places the data on the data bus; it also then issues a memory-write control signal (after allowing sufficient time for the all data bits setup) to store the bits at memory. The memory unit must write (store) the data during the interval in which memory-write signal is active and not inactivated by the processor.

The buses may have a time division multiplexed (TDM) address and data bits for memories. The interfacing circuit that demultiplexing the buses uses a control signal. (TDM means that in different time slots, there are different sets (channel) of signals.) The system has address signals in one time slot and data bus signals in another. The control signal is called Address Latch Enable (ALE) in 8051. The control signal is Address Strobe (AS) in 68HC11. It is address valid, (ADV) in 80196. An ALE, AS or ADV demultiplexes the address and data buses to the devices.

The buses for program and data memory may be multiplexed. The interfacing circuit for the demultiplexing of the buses uses a control signal. The control signal is PSEN in 8051 for demultiplexing common address bus for program and data memory. When the PSEN activates, it signals to read the program memory. When another control signal RD activates, it signals to read the data memory.

Each chip of the memory or port that connects the processor has a separate chip select input from a decoder. The decoder is a circuit that has appropriate bits of the address bus at the input and generates corresponding CS (chip select) control signals for each device (memory and ports) which are at the distinct set of addresses in the system. Demultiplexer and decoder circuits use higher bits of address bus, PSEN and ALE in 8051.

A circuit called glue-circuit, which includes the decoder for interfacing the system buses between the processor, memory and IO devices. Interconnections are through data and address and control bus signals. Understanding timing diagrams of bus signals is essential for appropriate design of the interfacing circuit and fusing (burning) it in a PLD (programmable logic device), GAL or FPGA. Figures 2.4, and 2.9(a) and (b) show the circuits interfacing the memory and ports in 8051 and 68HC11, respectively. The 8051 microcontroller uses an additional signal, PSEN (Program Store Enable), for program codes read from program memory. [This is because of the use of Harvard architecture (Section 2.4.2) for system memories.]

An interfacing circuit consists of decoders and demultiplexers and is designed according to the available control signals and timing diagrams of bus signals. This circuit connects all the units, processor, memory and the IO device through the system buses. It is a part of the glue circuit used in the system and is in GAL (generic array logic) or FPGA.

Figure 2.8 shows a simple diagram of a typical computer system in which buses provide an interconnecting network between the processor, memory, and IO systems. In real world interconnections, the network is formed by buses in the main subsystems.

The system bus interconnects the subsystems, which interconnects the processor with the memory systems and also connects another set of signals called the IO bus. Figure 2.10 shows the system and IO buses. It is a two-level bus architecture. Using an IO bus allows a computer to interface with a wide range of IO devices, without having to implement a specific interface for each IO device. An IO bus can also support a variable number of devices, allowing users to add devices to a system after it has been hardwired. Devices can be designed to interface with the bus, allowing them to be compatible with any system that uses the same type of IO bus. The IO bus creates an interface abstraction that follows the processor to interface with a wide range of IO devices using a very limited set of interface hardware.

Detailed descriptions of popular IO buses and wireless communication are given in Sections 3.10 to 3.13. PCI and USB bus (Section 3.12.2) interfaces to devices are designed to meet the PCI standard and USB (Section 3.10.3) standard.

All that is required is a device driver (Section 4.2.4) in each operating system—a program that allows the operating system to control the IO device (Section 8.6.1).

The downside of using an IO bus to interface to IO devices is that all the IO devices in a computer must share the IO bus, and IO buses are slower than dedicated connections between the processor and an IO device because the IO buses are designed for maximum compatibility and flexibility.

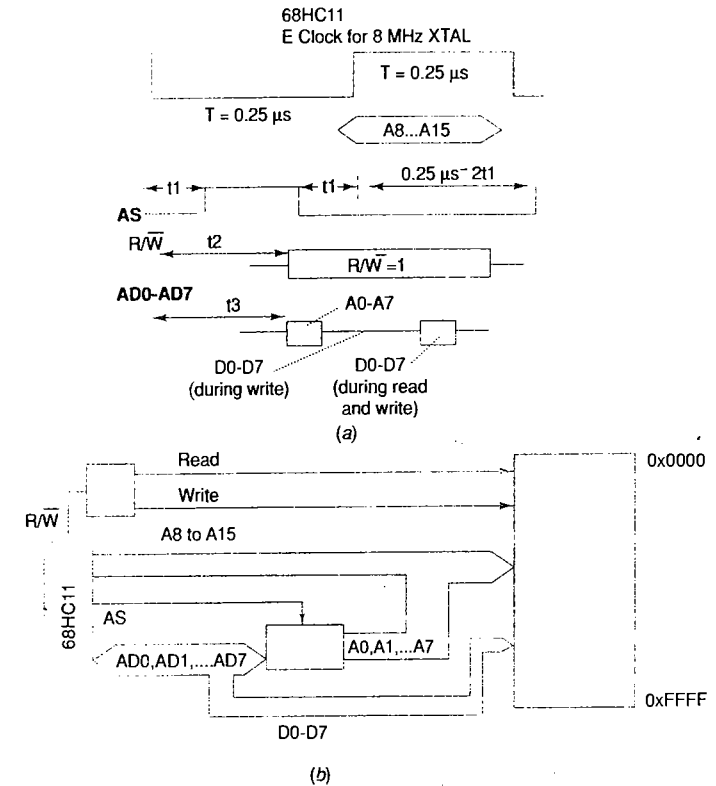


Fig. 2.9 (a) Timing of signals from processor when interfacing memory and ports in 68HC11
(b) Circuits for the interfacing memory and ports in 68HC11

An interfacing circuit consists of decoders and demultiplexers as well as an IO bus bridge controller. The interfacing circuit is designed as per available control signals and timing diagrams of bus signals. This circuit connects all the units, processor, memory, IO bus bridge controller and the IO devices through the system as well as through the IO buses. IO bus bridge controller may be a part of the glue interfacing circuit used in the system and is in PLD (programmable logic device), GAL (generic array logic) or FPGA.

Multilevel Buses Figure 2.10 shows a two-level bus architecture. Figure 2.11 shows a three-level bus architecture.

2.2 IO Addresses of Ports and Devices in Real World Interfacing

Memory Address-Mapped IO Operations Many processors and memory organization require memory-mapped IOs. IO device and port addresses are interfaced such that these are distinct from the memory

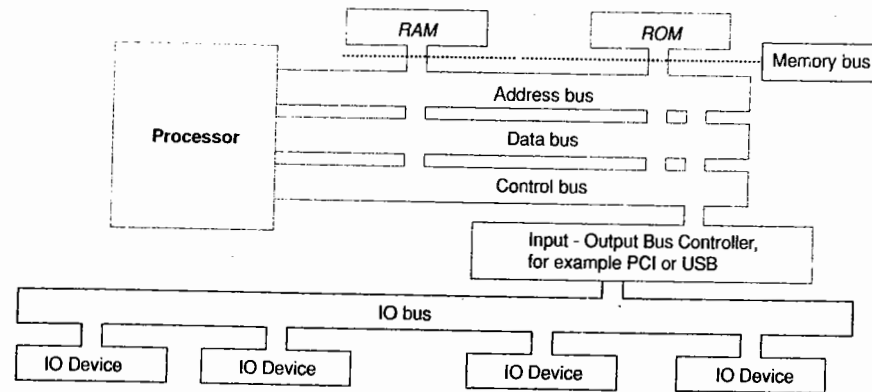


Fig. 2.10 Memory, system bus and IO bus interfacing in a two-level bus structure

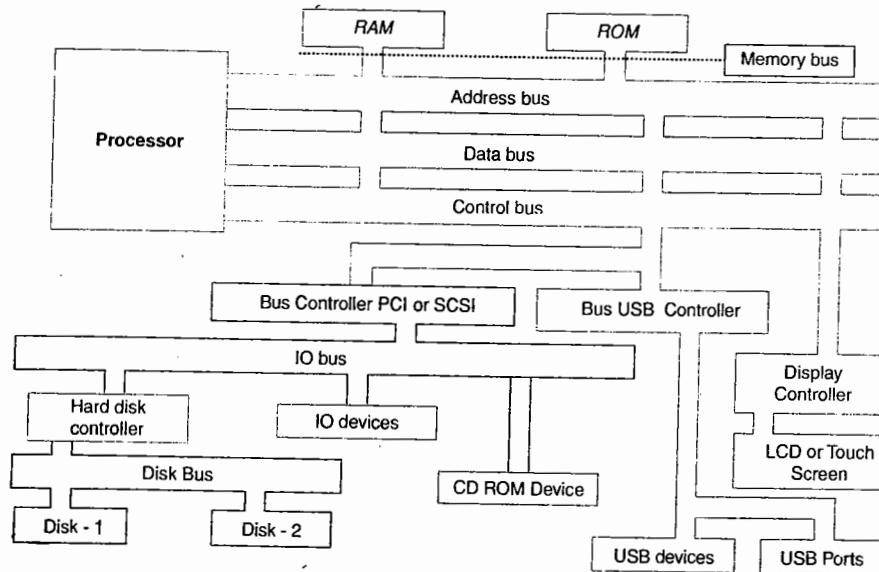


Fig. 2.11 Separate memory and I/O buses to communicate with the memory system, and the I/O system using a bus controller and a separate disk IO bus

addresses. Memory addresses are for data and software, and IO addresses are for the IOs. The following are features of memory-mapped IOs:

- The processor has no separate IO address space for ports and devices.
- The instructions as well as control signals for the operations on bytes at the memory, IO port and device addresses are the same.

- The processor has no separate input-output and memory load-store instructions.
- The arithmetic, logical and bit manipulation instructions that are available for data in memory are also available for IO operations. The processor can directly manipulate the data taken from or stored at the IO port or device. The manipulation of all instructions in the memory can be done using an accumulator, any register or any other memory address where the IO port byte is transferred after, during or before the arithmetic or logical operation.

Almost all microcontrollers, therefore, have no separate instructions for IO processing. The 8051 microcontroller (Section 2.1) is an example of a memory-mapped IO-based processor and memory organization. The 8051, 80196 and 80196 microcontrollers have preassigned device IO addresses for their internal devices and these addresses are not configurable.

Figure 2.12(a) shows that device addresses are within the RAM and are distinct from memory addresses. Motorola processors have no separate instructions for IO processing. Consider another system with a 68HC11 microcontroller.

A configuration is shown in figure. Port A, IO control register PIOC, Port C, B and port control (CTL) registers have addresses between 0x0000 to 0x0004. On-chip RAM is configured between 0x003F to 0x0040. [The port addresses and on-chip RAM are configurable by the bits of the configuration register in 68HC11. For example, the above device addresses can also be re-configured and assigned between 0x0100 and 0x1040.]

IO Addresses Mapped IO Operations Some processor and memory-organization requires IO address-mapped IOs operations. Consider a system with an 80x86 processor. Figure 2.12(b) shows the memory addresses on the left side. It shows the port addresses allocated in IBM PC for timer, keyboard, real time clock and serial port (called COM2) on the right side. This figure shows that device addresses need not be distinct, they can be the same as the memory addresses as a control signal will distinguish between them. The following are features of IO address-mapped IOs:

1. The processor has a separate IO address space for ports and devices.
2. The instructions and control signals for operations on bytes at the memory and IO ports and devices are distinct, making the design simple. IO devices and port addresses are interfaced independently of memory, without considering the memory addresses that are assigned for software and data.
3. The processor has separate input-output (for read and write) instructions and memory load-store (for read and write) instructions.
4. All the arithmetic, logical and bit instructions that are available in memory are first operated using the accumulator and then from there bytes are transferred after an arithmetic or logical operation.

The IO subsystem has input units and output units, also called IO devices. All IO ports and devices have addresses. These are assigned to devices according to the system processor and internal hardware configuration. Direct ALU operations on port byte(s) is not provided.

The addresses of device depend on the system hardware configuration. Most processors follow memory-mapped IOs and process the memory and other devices data with the same instructions. Some processors use IO-mapped IOs; for example, 80x86 processors process these with a different set of instructions (input-output instructions) and control signals.

2.2.3 Device Addresses in Real World Interfacing

During processor instruction, a device when addressed, it gets selected and communicates with system bus or IO bus using a set of addresses. These addresses are selected either as per decoder circuit design or as per the device-driver program for a controller for IO bus. The device addresses access during a read or write operation.

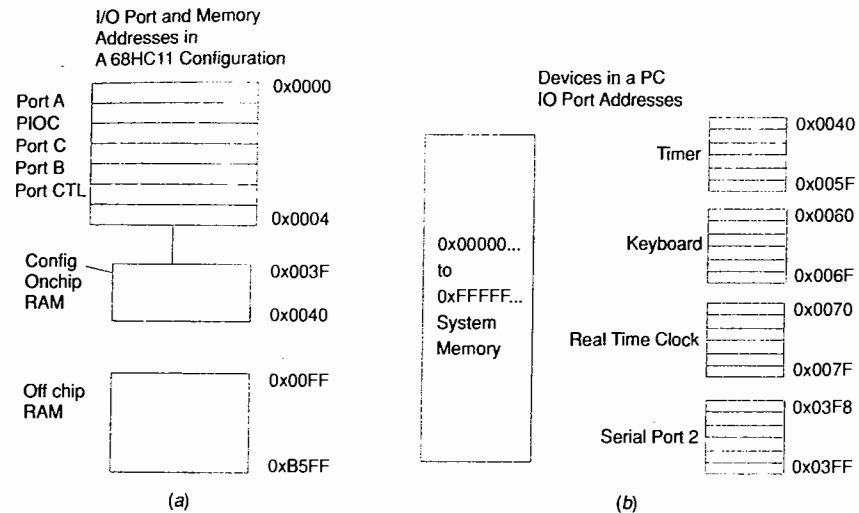


Fig. 2.12 (a) Processor and memory organization with I/O devices memory assignments in 68HC11 (with memory mapped IO architecture) (b) Device Addresses in the 80x86-based IBM PC

1. Device Data Register(s) or RAM buffer(s).
2. Device Control Register(s) to save control and configuration bits.
3. Device Status Register(s) for flag bits as per the device status. A flag may indicate the need for servicing or show an occurrence of device-interrupt.

Each device, and thus each device register, must be allocated addresses at the memory map.

A very important point to remember is that in most cases, each set of IO device addresses is often fixed by the system hardware. A locator or loader cannot reallocate these to any other set of addresses. Also, depending on the device, at a device address there can be one or several device-registers.

A physical or virtual device can be configured to attach or detach from receiving input and sending output. A device address can also be just like a file, record address and can be read only or write only or read and write both.

Example 2.6 gives the details of addresses of the registers of an IO device, called *serial line* or UART device.

Example 2.6

A *serial line device* has the addresses assigned for the device registers. The addresses are fixed by hardware configuration of UART port interface circuit in a system employing an 80x86 processor. They are from 0x2F8 to 0x2FE at COM1 in IBM PC.

1. (A) Two I/O data buffer registers (one for receiving and other for transmitting) are at a common address, 0x2F8, provided a control bit at address 0x2FBH is 0, (i) during read from the address, the processor accesses from the RBR (Receiver Data Buffer Register) and (ii) during write to the address, it accesses from the TRH (Transmitter Holding Register) at 0x2F8H. (B) Provided a control bit at address 0x2FB is 1, the data of two bytes of *Divisor Latch* are at the addresses, 0x2F8 (LSB) and 0x2F9 (MSB). The divisor latch holds a 16-bit value for

dividing the system clock. This then selects the rate of serial transmission of bits at the line. [A bit in another register (control register) changes 0x2F8 assignment from IO register to lower byte at divisor latch register and 0x2F9 to higher byte.]

2. Three control registers of device are assigned three addresses 0x2FA, 0x2FB and 0x2FC for IER, LCR and MCR. (i) IER (Interrupt Enabling Register) enables device interrupts. (ii) LCR (Line Control Register) defines how and how many bits will be on the line. (iii) MCR (Modem Control Register) defines how the modem does a handshake for communication.
3. Three status registers of the device are also at three addresses 0x2FA, 0x2FD and 0x2FE as follows: (i) IIR (Interrupt Identification Register) at 0x2FA. It has flags that set on a device-interrupt and reset at the system reset and at the servicing of the corresponding device-interrupt. (ii) LCR (Line Control Register) at 0x2FD. It defines how and how many bits will be on the line. (iii) MCR (Modem Control Register) at 0x2FE. It defines how a modem does a handshake and communicates.

Each IO device is at a distinct address or set of addresses. Each device has three sets of registers; data buffer register(s), control register(s) and status register(s). There can also be one or more device registers at a device address.

2.2.4 Interrupts and IOs

An IO device functioning is slow compared to that of processor. Therefore, an interrupt-driven IO is used. *Interrupts* are the mechanism used by most processors to handle *asynchronous* events.

Essentially, the interrupts allow devices to request that the processor stops what it is currently doing and execute software (called interrupt service routine) to process the device's request, much like a procedure call. ISR initiates by an event at external device rather than by a program instruction.

Interrupts are also used when a processor needs to perform an operation on some IO device and also needs to do other work while waiting for the operation to complete.

Example 2.7

1. Consider a keyboard example. It takes about 10 ms to send the code for a pressed key and thus a maximum of 10 keys can be pressed in 1 s. When does a key input event occurs is not fixed. Intervals, between two events of successive key inputs are also not fixed. In IO mode called interrupt driven mode, when a key is pressed, an interrupt signal RxRDY (receiver data ready) to a processing unit causes the execution of a service routine and the service routine program reads the byte for that code. Figure 2.13(a) shows the method of input from and to a port using RxRDY interrupts.
2. Consider a printer. Assume that a maximum of 300 characters can be printed in 1 s and it thus takes about 0.3 ms to print the code sent at the output by a port. When a print operation completes for a character that is not fixed. Intervals between two events of successive print of the characters are also not fixed. In interrupt-driven mode, when a print action completes, an interrupt signal TxDE (transmission data empty) to the printer processing-unit (print controller) will cause the execution of a service routine and the service routine will then send another byte as output. Figure 2.13(b) shows the method of output from and to a port using the TxDE interrupts.

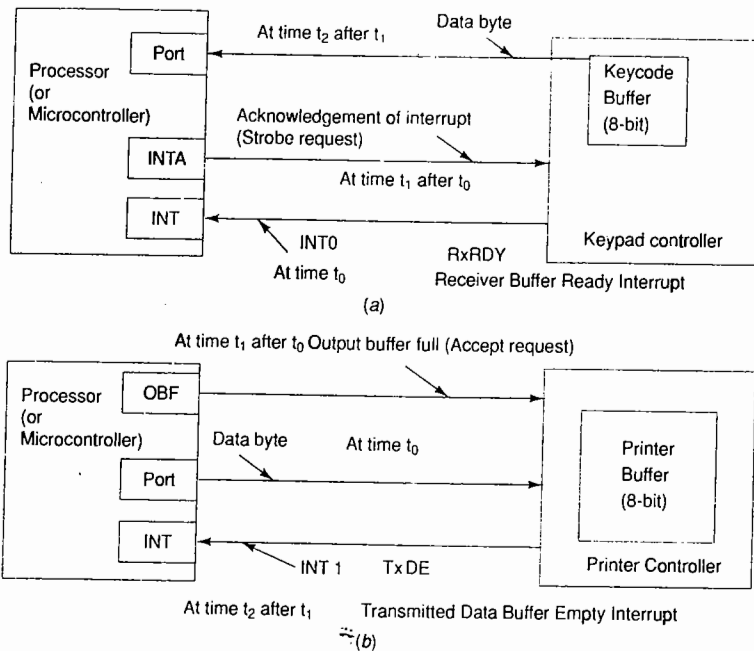


Fig. 2.13 (a) Method of Input from and to a port using the RxRDY interrupts (b) Method of Output from and to a port using the TxDE interrupts

2.2.5 Bus Arbitration

There can be several processors as well as several single purpose processors (Section 1.2.1), which share a bus. A single purpose processor can also be a controller. The controller can be part of a device or peripheral.

Figure 2.14(a) shows how the system buses are shared between the controllers, IO processor and multiple controllers for access. Only one of them is granted the bus master status at an instance. In general, there can be a number of DMA or other controllers or processors trying to get access to a bus at the same time, but access can be given to only one of these. Therefore only one processor or controller can be the bus master. A controller is called *bus master* when it has access to a bus at an instance. Any controller or processor can be the bus master at the different instances.

Bus arbitration process refers to a process by which the current bus master accesses and then leaves the control of the bus and passes it to another bus-requesting processor unit. There are three methods, one of which is used in the bus arbitration process. They are *Daisy Chain*, *Independent Bus Requests and Grant*, and *Polling* methods.

Daisy Chain Method Figure 2.14(b) shows a bus arbitration method called daisy chaining method. It is a centralized bus arbitration process. Bus control passes from one bus master to the next, then to the next and so on. Bus control passes from controller units U0 to U1, then to U2, then U3, and so on. Signals in the arbitration process are as follows: A bus-grant signal (BG) functions like a token, which is first sent to U0. If U0 does not need the signal the bus, U0 transfer it to U1. A controller needing the bus raises a bus-request

(BR) signal. A bus-busy (BUSY) is sent when that controller becomes bus master. When bus master no longer needs the bus, it deactivates BR and BUSY. Another BG is issued and passed from U0 to the controllers one by one lined up according to their priorities.

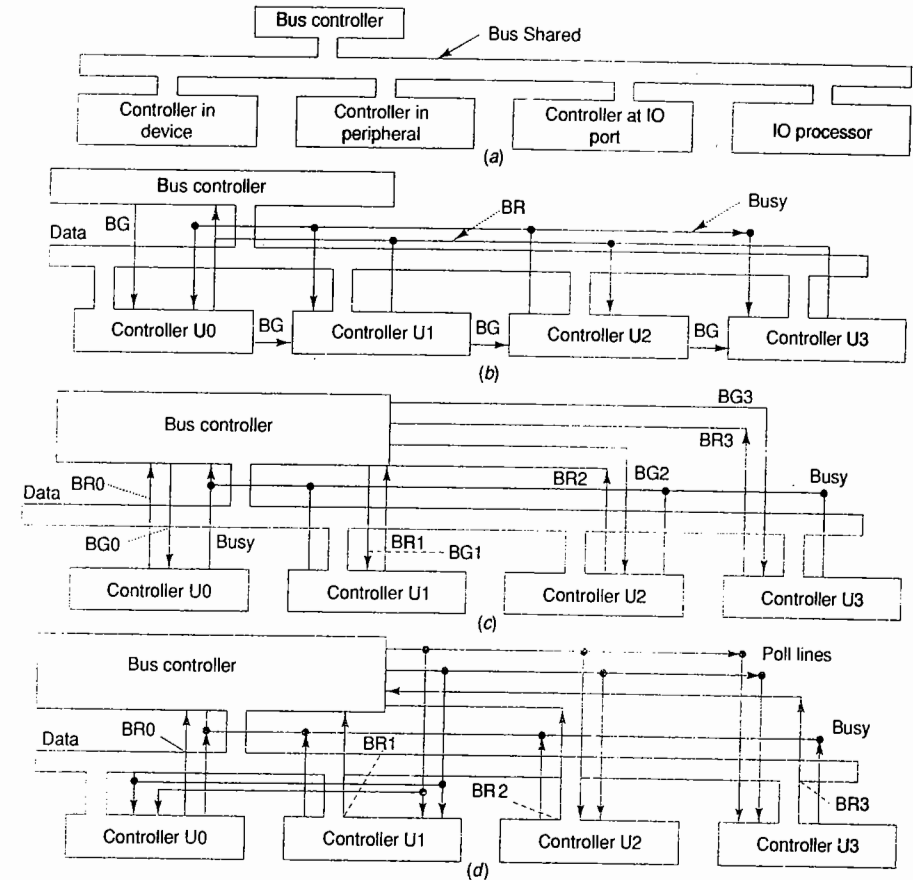


Fig. 2.14 (a) System buses shared between the controllers, when IO processor and multiple controllers access the bus, and only one of them granted bus master status at any one instance (b) Bus Arbitration by daisy chaining method (c) Bus Arbitration by independent bus request method (d) Bus Arbitration by polling bus method

The advantage in this is that at each instance of bus access, the i^{th} controller gets highest priority compared to $(i + 1)^{th}$. The controllers and processor priorities for granting bus access (bus master status) are fixed.

Independent Bus Request Method Figure 2.14(c) shows the bus arbitration called independent bus request method, in which each controller has separate BR signals, BR0, BR1, ..., BRn. Also, there are separate

BG signals, BG0, BG1, ..., BGn for each controller. An i^{th} controller sends BRi (i^{th} bus request signal) and when it receives BGi (i^{th} bus grant signal), it uses the bus and activates BUSY signal. Any controller, which finds an active BUSY, does not send a BR from it. The advantage here is that the i^{th} controller can be programmed to give the highest priority to the bus and the priority of a controller can be programmed dynamically.

Bus Polling Method Figure 2.14(d) shows the bus arbitration called bus polling method with two poll lines for four controllers. A poll count value is sent to the controllers and incremented to provide bus access to the next. Assume there are 8 controllers. Three poll count signals p2, p1, p0, successively change from 000, 001, ..., 110, 111, 000, If on count = i , a BR signal is received, then counts increment stops, and BUSY activates when that controller becomes the bus master. When BR deactivates then BUSY also deactivate and count increment starts. The advantage is that the controller next to the current bus master gets the highest priority to access the bus after the current bus master finishes its operations.

2.2.6 Interfacing Examples with Keyboard, Displays, D/A and A/D Conversions

Keyboard Figure 2.15(a) shows an interface to a keyboard. Two signals from a keyboard controller are KBINT and TxD. KBINT is interrupt due to RxRDY signal from keyboard controller. TxD is the serial UART data output of a controller connected to RxD at SI in 8051, or UART Intel 8250, or UART 16550, which includes a 16-byte buffer.

Bounces create on pressing a key. This is due to a natural spring-like action. Each bounce results in a false pulse. The keyboard controller has a hardware debouncer to neutralize the false pulse. The keyboard controller has a counter, which continuously increments at a certain rate and scans each key whether it is in pressed or released state. It has an encoder to encode the keyboard output for a ROM. The ROM then generates an ASCII code output for the pressed key. The code also takes into account the meaning of multiple keys when they are simultaneously pressed. For example, if shift key is also pressed then the code for an upper case character is generated. The code bits are serially transferred to TxD output, which is received at RxD input of SI.

Display Section 1.3.8 described the LCD, LED and touchscreen displays. Figure 2.15(b) shows an interface circuit to an LCD display controller. Section 3.3.4 gives details. There are 8 output data and 3 bits for E, RS and R/W. One 8-bit port is used for output data. Another port is used for 3 bits.

Digital-to-Analog Converter Section 1.3.7 described the DAC (also called D/A). A D/A needs a PWM circuit, which is an internal device in microcontroller. A pulse width register (PWR) is programmed according to a required analog output. A counter/timer device generates two internal interrupts: one on timer overflow and another after an interval proportionally equal to PWR. On the first interrupt the output becomes 1 and on the second it becomes 0. An external integrator generates the analog output as per the period of output 1 (period between the first and second interrupts) compared to the total period of output pulses (period between successive first interrupts). Figure 2.15(c) shows an interface to an external D/A. The external D/A is used as an alternative when PWM is not used.

Analog-to-Digital Converter Section 1.3.7 described an ADC (also called A/D). An n -bit A/D needs (i) a start pulse for converting using a short duration single pulse generator circuit, (ii) a sample hold amplifier circuit to hold the signal constant during the conversion period and (iii) positive and negative voltage references for providing the reference potential difference for conversion of analog input into n -bits. A four or eight channel A/D is inbuilt in microcontrollers. An external (ADC), for example, ADC0808, can also be used with interfacing similar to that of the ports. Figure 2.15(d) shows an interface to an external A/D when internal A/D not used.

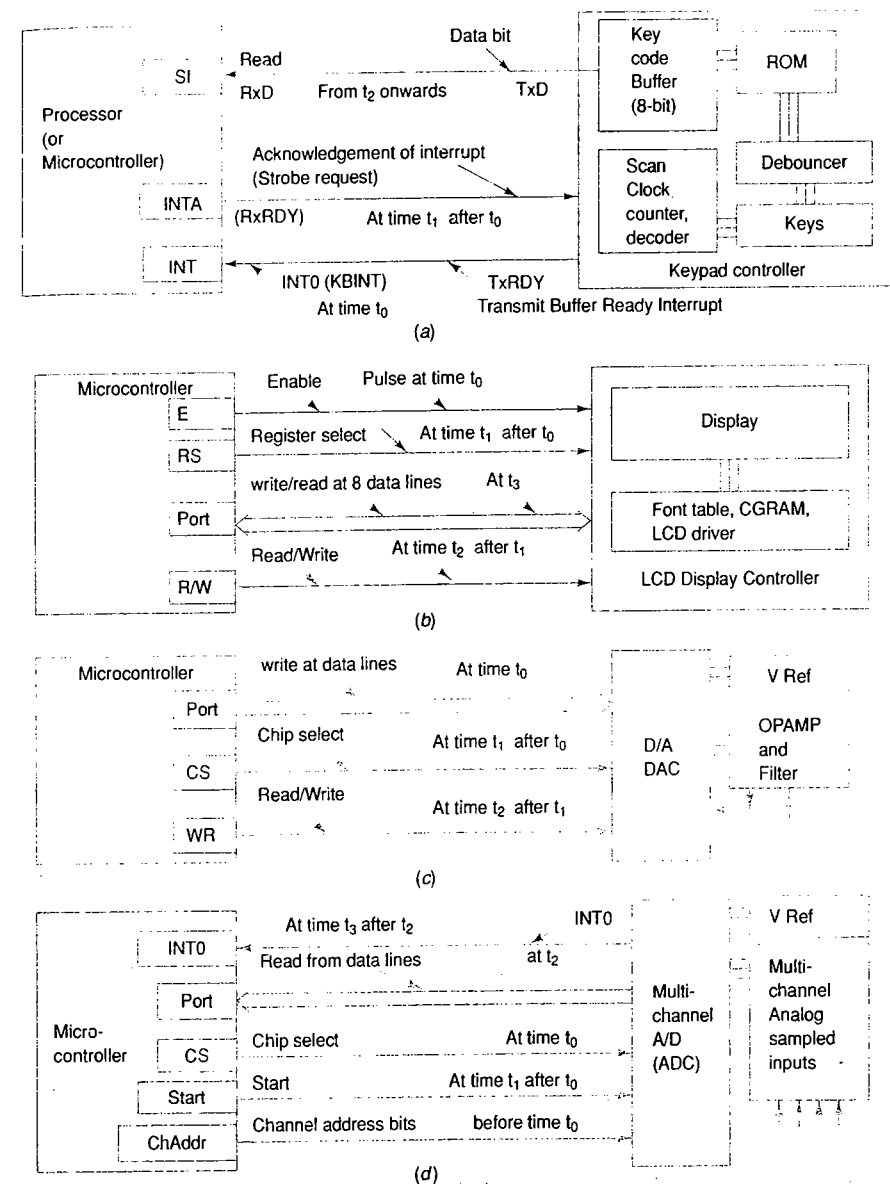


Fig. 2.15 (a) Interfacing to a keyboard using keyboard controller (b) Interfacing to a LCD display controller (c) Interfacing to D/A (DAC) when internal PWM not used (d) Interfacing to external A/D (ADC) when internal ADC not used

2.3 INTRODUCTION TO ADVANCED ARCHITECTURES

Figure 2.16 shows an organization of various processor units. The units, which are shown with the dashed boundary are present in high performance processors. External address, data and control buses interface with the processor and connect to external memory units, ports and devices.

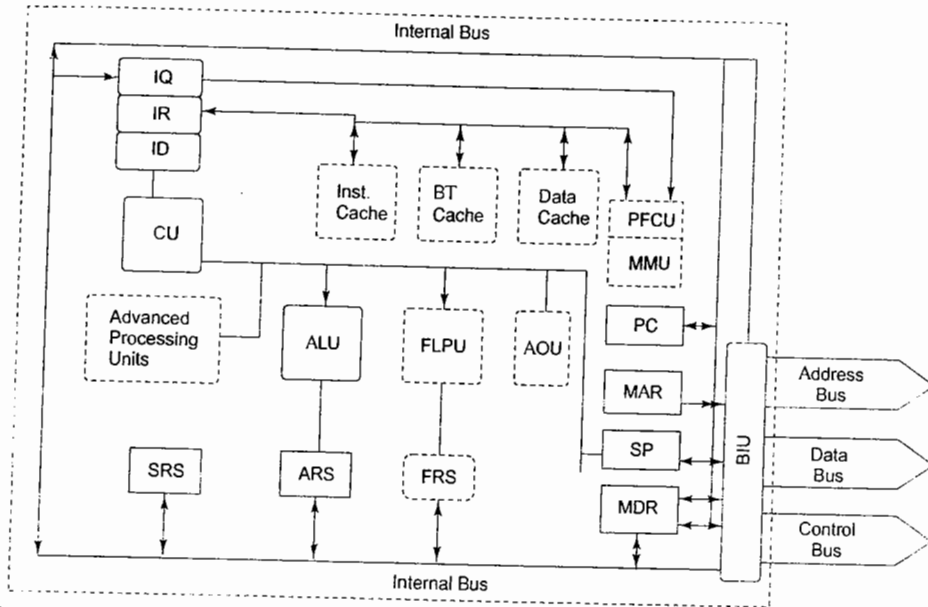


Fig. 2.16 Organization of various processor units. The units, which are shown with the dashed boundary are present in processors having high-performance advanced architecture

The following are the general features present in most processors.

1. **Fixed Instruction Cycle Time:** This is the time taken by a processor to execute a simple instruction, which is $\sim 1 \mu\text{s}$ for the 8051 processor operating at $\sim 12 \text{ MHz}$, and $0.9 \mu\text{s}$ per MHz clock rate for the ARM9 processor. A system designer uses the instruction cycle time as an indicator to select processor clock speed as per the application. For example in applications that need fast processing, the ARM9 processor at 100 MHz would be considered suitable; for other applications for which slower processing will suffice the 8051, 68HC11 or 80196 can be chosen.
2. **Internal bus width:** An ALU gets inputs through the internal buses. Bits in a single operand to ALU (during a single arithmetic or logical operation) are equal to the bus width. A 32-bit bus will facilitate the availability of arithmetic operations on 32-bit operands in a single cycle. The 32-bit bus becomes a necessity for signal processing and control system instructions. When the bus-width is 32 bits, it reads or writes an integer of 32 bits and will process about four times faster than when the width is 8. An internal bus of 128 bits is present in SHARC and 64 bits in Pentium.
3. **Program-counter (PC) bits and reset value:** The number of PC bits decides the maximum possible size of the physical memory that can be accessed by the processor. The reset value tells the designer

the initial program address from where the program runs on a system reset or power up. The processor will start execution from that address. [The initial instruction pointer and code segment register bits decide the initial program's memory address in 80x86 processors.]

4. **Stack-pointer bits and initial reset value:** Stack pointer values must point to addresses of the words stored at the *stack*. These addresses must be within the ones allocated for stack in the memory. The software designer defines an initial reset value and sets the beginning stack pointer accordingly.

Table 2.1 lists the structural units in a general-purpose processor. It lists the functions of each.

Table 2.1 General structural units in a processor architecture

Structural	Unit	Functions
MAR	Memory address register	It holds the address of the byte or word to be fetched from external memories. Processor issues the address of instruction or data to MAR before it initiates fetch cycle.
MDR	Memory data register	It holds a byte or word fetched (or to be sent) from (to) an external memory or IO address.
System buses	Internal Bus	It internally connects all the structural units inside the processor. Its width can be 8, 18, 32, 48 or 64 bits.
	Address bus	An external bus that carries the address from MAR to memory as well as to IO devices and other units of system.
	Data bus	An external bus that carries, during a read or write operation, the bytes for instruction or data from or to an address. The address is determined by MAR.
	Control bus	An external set of signals to carry control signals to processor or memory or device.
BIU	bus interface unit	An interface unit between processor's internal units and external buses.
IR	Instruction register	It sequentially takes instruction codes (opcode) to execution unit of processor.
ID	Instruction decoder	It decodes the instruction received at the IR and passes it to processor CU.
CU	Control unit	It controls all the bus activities and unit functions needed for processing.
ARS	Application register set	(a) A set of on-chip registers used during processing of instructions of an <i>application</i> program or (b) a register window, (c) a subset of registers with each subset storing static variables of a software-routine or (d) a register file associated to a unit such as ALU or FLPU.
ALU	Arithmetic logical unit	A unit to execute arithmetic or logical instructions according to the current instruction present at IR.
PC	Program counter	It generates an instruction cycle by sending the address defined by it to memory through MAR. It auto-increments as the instructions are fetched regularly and sequentially. It is called instruction pointer in 80x86 processors.
SP	Stack pointer	A pointer for an address, which corresponds to a stack-top in memory.

2.3.1 Architecture of the Advanced Processors

Figure 2.16 shows the additional units in boxes with dashed boundary and these units are present in advanced processor architectures (high performance processors). Table 2.2 lists the advanced architecture structural units in a processor organization of general-purpose processor. It lists functions of each unit.

Table 2.2 Structural units in an advanced processor architecture

Structural	Unit	Functions
Instruction level parallelism units	ILP	For instruction level parallelism (Section 2.5), the multistage pipeline processing, multiline superscalar processing, and dual, quad or multicore processing speeds up the performance from one instruction per clock cycle ¹ .
IQ	Instruction queue	A queue of instructions so that the IR does not have to wait for the next instruction after one has been processed.
PFCU	Prefetch control unit	A unit that controls the <i>fetching</i> of data into the I- and D-caches in advance from the memory units. The instructions and data are delivered when needed by the processor's execution unit(s). The processor does not have to fetch data just before executing the instruction. Pre-fetching unit improves performance by fetching instructions and data in advance for processing. Caches along with a MMU improve performance by giving the instructions and data fast to the processor execution unit.
I-Cache	Instruction cache	It sequentially stores, like an instruction queue, the instructions in FIFO mode. It lets the processor execute instructions at great speed using PFCU compare to external system-memories, which are accessed at relatively much slower speeds.
BT Cache	Branch target cache	It facilitates ready availability of the next instruction-set when a branch instruction like <i>jump</i> , <i>loop</i> , or <i>call</i> is encountered. Its fetch unit foresees a branching instruction at the I-cache.
D-Cache	Data cache	It stores the prefetched data from external memory. A data cache generally holds both the key (address) and value (word) together at a location. It also stores write-through data when so configured. Write-through data means data from the execution unit that transfer through the cache to external memory addresses.
MMU	Memory-management Unit	It manages the memories ² such that the instructions and data are readily available for processing.
SRS	System register set	It is a set of registers used while processing the instructions of the supervisory system program.
FLPU	Floating point processing unit	A unit separate from ALU for floating point processing, which is essential in processing mathematical functions fast in a microprocessor or DSP.
FRS	Floating point register set	A register set dedicated for storing floating point numbers in a standard format and used by FLPFU for its data and stack.
MAC	Multiply and accumulate unit	There is also a MAC ³ units for multiplying coefficients of a series and accumulating these during computations.

(Contd)

Structural	Unit	Functions
AOU	Atomic operation unit	It lets a user (compiler) instruction, when broken into a number of processor instructions called atomic operations, finish before an interrupt of a process occurs. This prevents problems from arising out of shared data between various routines and tasks.

1. Instruction cycle time becomes several times less than the processor clock cycle time.
2. The MMU manages the pages in the RAM memory as well as the copies in internal and external caches. Managing has to be done in such a way that when the instructions execute, there are minimum number of page and cache faults (misses).
3. MAC units are invariably needed in DSPs. [Section 2.3.5]

Advanced processor circuits consist of RISC architecture. It improves performance by executing most instructions in a single clock cycle (by hardwired implementation of instructions), by using multiple registers, windows and files and by greatly reducing dependency on the external memory accesses for data due to the reduced number of addressing modes for arithmetic and logic instructions. An RISC has only a few addressing modes for arithmetic and logic instructions. It does not have the following addressing modes: indirect (index), auto-index, and index-relative for ALU instructions. It does not have a second operand fetched by the immediate addressing mode for arithmetical and logical instructions.

Advanced processor circuits consist of a floating-point unit; FRSs process mathematical functions faster and with greater precision than when employing an integer-processing ALU.

Advanced processing units include the instruction pipelining unit, which improves performance by processing instructions in multiple stages. Pipelining allows a processor to overlap the execution of several instructions so that more instructions can be executed in the same period of time. Section 2.5.1 will describe multiple stages of instruction execution and will describe how instruction level parallelism (ILP) further improves processor performance.

Figure 2.17 shows how instructions flow through the pipeline.

In cycle 1, the first instruction I_1 enters the instruction fetch (IF) stage of the pipeline and stops at the pipeline latch (buffer) between the instruction fetch and instruction decode (ID) stage. In cycle 2, the second instruction I_2 enters the instruction fetch stage, and I_1 proceeds to the instruction decode stage. In the cycle 3, I_1 enters the register (inputs) read (RR) stage, instruction I_2 is in the instruction decode stage, and instruction I_3 enters the instruction fetch stage. In fourth cycle, I_1 moves to execute stage and in fifth cycle to result write back stage.

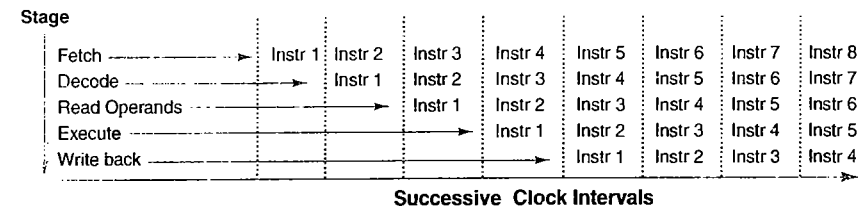


Fig. 2.17 Instruction flow in a pipeline of an advance architecture processor

Instructions proceed through the pipeline at one stage per cycle until they reach the register (result) write-back (WB) stage, at which point execution of the instruction I_1 (instr1 in figure) is complete. Thus, in cycle 6

in the example, instructions I_2 through I_6 are in the pipeline, while instruction I_1 has completed and is no longer in the pipeline. A 5-stage pipelined processor is still executing instructions at a rate (throughput) of one instruction per cycle, but the latency of each instruction is now 5 cycles instead of 1. The faster execution takes place as cycle time now can be one-fifth or less than unpipelined case.

2.3.2 80x86 Architecture

The first four generations of 80x86 are 8086, 80286, 80386 and 80486. The first processor in the 80x86 family of processors is the 16-bit 8086 (1981). The 80x86 has a 32-bit architecture since 80386. Pentium is the fifth generation architecture (1994) based on the 32-bit 80386. Pentium 4 is of seventh generation and Xeon and Core2 are eighth generation architectures. Core2 means dual core architecture. The 80x86 architecture processors have become popular since their application in the IBM PC (personal computer). Itanium is based on 64-bit architecture, which simulates the 80x86 architecture.

The features of 80x86 architecture are as follows:

1. The original 8086 architecture consists of general purpose registers AX, BX, CX and DX. Each can be considered as two 8-bit registers. For example AX as AL (A lower byte) and AH (A upper byte). A 32-bit extension has EAX, EBX, ECX and EDX. EAX registers. Each can be considered as two 16-bit registers. AX then has a lower 16-bit EAX. Figure 2.18 shows the 80x86 architecture registers.
2. The 8086 architecture provides for code, data, stack and stack segmentations. The original 8086 architecture consists of four segment registers CS, DS, SS and ES to enable access to memory assigned to different segments.
3. IP is an instruction pointer, of a 16-bit address, and CS contains a 16-bit program code segment address for 16 upper bits of address.
4. SI contains index of source operand and DI contains 16-bit index of destination. BP is memory offset pointer of 16 bits address and DS contains a 16-bit data memory segment address upper bits.
5. 16-bit or 32-bit or 64-bit words store as little endian. Data need not be aligned at the addresses in multiples of 2 or 4 and can start from any address.

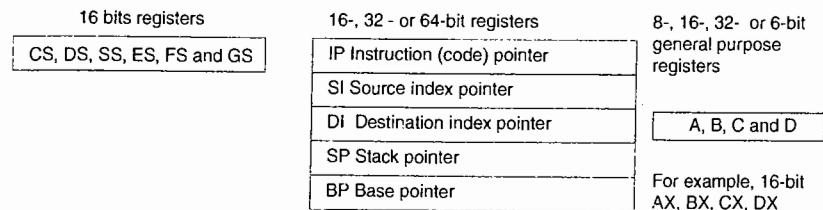


Fig. 2.18 80x86 architecture registers

6. The 80x86 mainly uses two address arithmetic and logic instructions. This means that the accumulator is not the only register to accumulate ALU result, which in turn means that a register operand (AX or BX or CX or DX) can be a destination as well as the first source operand.
7. A memory address can be the first or second operand, a characteristic of CISC addressing modes for ALU instructions.
8. The present generation 80x86 architecture decodes a CISC instruction and creates microoperations that implement on a microarchitecture of RISC.
9. The small number of general registers (also inherited from 8085) has made register-relative addressing (using small immediate offsets) an important method of accessing operands, especially on the stack.

10. The 80x86 has IO mapped IO. An IO address is of 16 bits for an IO byte. Processors of the Intel 8086 family process and access IO units and IO devices by the separate IN and OUT instructions. The IO mapped IO processors have a separate set of addresses for accessing inputs and outputs. It simplifies the IO units interfacing circuit that connects to the processor.
11. The 8086 supports 256 interrupt levels for the hardware as well as software and supports nested interrupts. This means that an ISR can be interrupted and a higher priority ISR can execute in between.
12. The new generation 8086 architecture supports a mode called real mode. Real mode supports direct access without segmentation to peripheral devices and basic input output subroutines (BIOS). Real mode supports 20-bit segmentation instead of 16-bit. The segment register has only the upper 16 bits. The lower bits are 0s.
13. A mode called 32-bit protected mode is also provided and supports pages in memory.
14. 8086 supports many OSs, including Windows and multitasking operating systems.
15. The latest 80x86 architectures support thread handling, integer SIMD and SIMD extension instruction sets.

Program routines and processes can have different segments. For example, a program code can be segmented and each segment stored at a different memory block. A pointer address points to the start of the memory block storing a segment and an offset value is used to retrieve a memory address within that segment. The data can also be segmented with each segment at different blocks. Similarly, strings can be segmented.

The 80x86 architecture is a widely used architecture. The data are not aligned and save as little endian. It has general purpose pointers and segment registers and supports memory segmentation and paging. There can be different segments at the memory for different functions and processes (tasks). These can comprise different segments for data and different segments for the stacks.

2.3.3 ARM

Detailed information on ARM is at <http://www.arm.com>. A brief description of ARM architecture and features that makes it important for embedded systems, such as digital and video cameras and mobile phones, is give here.

Figure 2.19 shows ARM7 registers and a three stage pipeline architecture. ARM has registers R0 to R15. R15 also functions as a program counter. R14 functions as a link register. It has CPSR (current program status register) and SPSR (saved program status register).

The main features of ARM are as follows:

1. It has 32-bit architecture but also supports 16-bit or 8-bit data types. It supports 16-bit instructions also in Thumb® mode. It supports Jazelle Java execution accelerator.
2. ARM is programmable as little endian or big endian data.
3. ARM provides the advantage of using a CISC in terms of functionality, along with the advantage of an RISC in terms of faster program implementation as well as reduced code lengths. It implements faster because the register word instantly availability to execution-unit. Code lengths are reduced because most instructions use registers as operands. Few bits in the instruction specify a register as operand. 8, 16 or 24 bits specify a memory address as operand and the displacement bits in the instruction.
4. ARM7 and ARM9 microprocessors have a combination of RISC and CISC features. ARM supports a complex addressing modes-based instruction set. ARM processor has an RISC core for processing. There is an in-built compilation unit. It first compiles the CISC instructions into RISC formats, which are then implemented by the RISC core of the processor. Internally, the implementation for many instructions is like in a RISC (without the micro-programmed unit).
5. ARM7 has Princeton memory architecture; ARM9 has Harvard architecture. [Section 2.4.2]

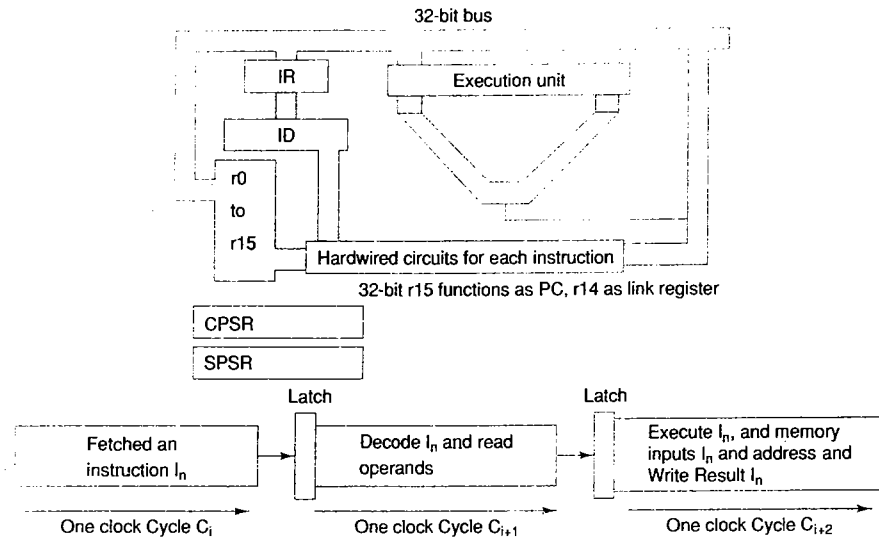


Fig. 2.19 ARM7 registers and three stage pipeline architecture

6. ARM debug and trace tools quickly debug real-time software, and trace instruction execution and associated program data at full core speed.
7. A wide choice of development tools and of simulation models for leading EDA (Electronic Design Automation) environments and excellent debug support for SoC design are available.
8. ARM codes are forward compatible with higher versions. For example, ARM7 codes are forward compatible with ARM9, ARM9E and ARM10 processors as well as with Intel XScale micro-architecture. ARM9E and ARM 10 families use a Vector Floating Point (VFP) ARM coprocessor, which adds full floating point operands. VFP also provides fast development in SoC design when using tools like MatLab®. Applications are in image processing (scaling), 2D and 3D transformations, font generation and digital filters.
9. ARM permits programming by an additional instruction set designed for 16-bit operations. Thumb is an industry standard instruction set, which enables 32-bit performance at the 8/16-bit system cost in terms of memory needs. This provides typical memory savings of up to 35%, over the equivalent 32-bit code, while retaining all the benefits of a 32-bit system (such as access to a full 32-bit address space). There are no overheads (in terms of time and memory) in moving between Thumb and the normal ARM state of the codes. The two states are compatible on a normal basis. This gives the code designer complete control over performance and code-size optimization.
10. ARM uses an Intelligent Energy Manager (IEM) technology. It implements advanced algorithms to optimally balance processor workload and energy consumption. It maximizes system responsiveness. IEM works with the operating system and mobile OS. An application running on a mobile phone dynamically adjusts the required CPU performance level.
11. ARM processors use the AHB (AMBA Advanced High Performance Bus) interface. AMBA is an established open source specification for on-chip interconnects. [Section 3.12.3] AMBA serves as a framework for SoC designs and development of IP cores. It provides a high-performance and fully

synchronous back plane. (Back plane has additional set of controllers, which can access each other through another common bus, which is distinct from system bus. The multilayer AHB in version ARM926EJ-S and all members of the ARM10 family represent a significant advancement. They reduce access latencies and increase the access bandwidth in a multimaster (multiple controllers accessing the bus as master) system.

Instruction Set – ARM7 Processors have the following type of instruction sets. The ARM7 in version with suffix T has instruction set called Thumb® instruction set support.

1. **Data Transfer Instructions** Given below are the instructions for transfer between register-memories. The memory address is as per a register used in index or index-relative or post auto-index addressing mode.
 - (a) load in register a word (LDR)
 - (b) store from register a word (STR)
 - (c) set a memory address in a register (ADR). Address is of 12 bits. [Alternative for 16 bits address setting in a register is using any register or r15 in an arithmetic operation.]
 - (d) load in register a byte (LDRB)
 - (e) store from register a byte (STRB)
 - (f) store from register a half word (STRH) [A word in ARM is of 32 bits.]
 - (g) load in register a half word as such or signed half word (LDRH or LDRSH).
 The following are the instructions for a word transfer between registers:
 - (a) Move (MOV)
 - (b) Move reverse (MVR)

A load or move or store instruction can be conditionally implemented. For example, MOVLT r3, #10. The immediate operand 10 will transfer to r3 provided a previous instruction for comparison showed the first source as less than the second. Conditions are LT (signed number less than), GT (signed number greater than), LE (signed number less or equal), EQ (equal), NE (not equal), VS (overflow), VC (no overflow), GE (signed number greater than or equal), HI (unsigned number higher), LS (unsigned number lower), PL (plus, nor Negative), MI (minus), CC (carry bit reset), and CS (carry bit set).

2. Bit Transfer or Manipulation Instructions

- (a) Register-bits Logical Left Shift (LSL)
 - (b) Register-bits Logical Left Arithmetic Shift (ASL)
 - (c) Register-bits Logical Right Shift (LSR)
 - (d) Register-bits Logical Right arithmetic Shift (ASR)
 - (e) Register-bits Rotate Right (ROR)
 - (f) Register-bits Rotate Right with carry also extended for rotating (RRX).
3. **Arithmetical and Logical Instructions** The following are the instructions for arithmetical operations. Each uses three operands from the registers. One source may, however, be immediate operand addressing in addition and subtraction.
 - (a) Add without carry two words and put result at the third operand (ADD)
 - (b) Add with carry two words and put result at the third operand (ADC)
 - (c) Subtract without carry two words and put result at the third operand (SUB) [Carry bit used as borrow.]
 - (d) Subtract with carry two words and put the result is at the third operand (SBC)
 - (e) Subtract reverse (second source with the first) without carry two words and put result is at the third operand (RSB) [Carry bit used as borrow.]
 - (f) Subtract reverse with carry two words and the result is in the third operand (RSC)
 - (g) Multiply two different registers and put result is at the destined register (MUL)

(h) Multiply two source registers and add the result with the third source register and accumulate the new result in a destined register (MLA) [There are four operand registers.]

The following are the instructions for logical operations:

- Bit wise OR two words and put result at the third operand (ORR)
- Bit wise AND two words and put result at the third operand (AND)
- Bit wise Exclusive OR two words and put result at the third operand (EOR)
- Clear a Bit (BIC). [There is one source for the bits; a second source for the mask and the result is put at the third operand.]

An arithmetical or logical instruction can be conditionally implemented. For example, SUBGE r1, r3, r5. The operand from r3 is subtracted from r5 if the GE condition resulted in earlier operation for test or comparison.

The following are the instructions for *compare and test operations*. The result destined to CPSR, which stores four condition bits, N, V, C, and Z.

- Bit-wise Test two words (TST)
- Bit-wise negated test between two words (TEQ)
- Compare two words and put result at the CPSR condition bits (CMP)
- Compare two negative words and put result at the CPSR condition bits (CMN)

4. **Program-Flow Control Instructions** The following are the instructions for branching operations. A branching instruction can be conditionally implemented. Branch to an address relative to PC word in r15 (B). 'B #1A8' means add 0x1A8 in PC and change the program flow. 'BGE #100' means that if a GE condition resulted on a previous compare or test, then add 1A8 in the PC. There are similar instructions for different conditions of the processor status flags (at CPSR). [PC is r15.]

Example 2.8

This example gives an assembly language program example for the ARM.

Consider the problem of adding three numbers, x, y and z (= 127, 29 and 40) and storing the result at a memory address, M for a [a = x + y + z.] Using the instructions of the above instruction-set, the assembly language codes will be as follows.

- BEGIN: MOV r2, #0x007F ; Transfer 127 into processor register r2.
- MOV r3, #0x001D ; Transfer 29 into processor register r3.
- MOV r4, #0x0028 ; Transfer 40 into processor register r4.
- MOV r1, #0x000 ; Transfer 0 into processor register r1.
- ADD r1, r1, r4 ; Add the register r4 word into the r1.
- ADC r1, r1, r3 ; Add the register r3 word along with the carry (if any) from previous addition into the r1.
- ADC r1, r1, r2 ; Add the register r2 word along with the carry (if any) from previous addition into the r1.
- ADR r5, 0x800 ; Set the address into r5. Memory address M set 0x800.
- STR [r5], r1 ; Store the r1 at the address pointed by r5.

Table 2.3 gives features and comparison of the exemplary high performance ARM family of processors.

- ARM9™ Thumb® family supports Windows CE, Palm OS, Symbian OS, Linux and other OS/RTOS. There is Palm OS support in ARM920T and ARM922T processors. ARM 940T has a memory Protection Unit (MPU) and a support to a range of Real-Time Operating Systems including VxWorks.

- ARM7 and ARM9 integrates instruction and data caches.
- ARM architecture refers specifically to the architectural instruction sets and programmers models, such as ARMv5TE, ARMv5TEJ and ARMv6 architecture in ARM11.
- ARMv4T (version 4 Thumb) microarchitecture is common to ARM7, ARM9, ARM10 and ARM11 families. The term ARM microarchitecture refers specifically to the implementation of architectures such as the ARM9™ family of cores and the ARM10 family of cores. For example, ARM926EJ-S™ core and the ARM1020E™ core are CPU products based on those earlier microarchitectures. An enhancement of v4T architecture is ARMv5TE architecture (introduced in 1999). It has ARM DSP instruction set extensions that improves the speed of instruction set by up to 70% for audio DSP applications. [Certain applications need microcontroller data processing features as well as DSP features in a single processor in place of the multiprocessor system.]

Table 2.3 Comparative features of ARM versions

Feature	ARM7™ Thumb® Family	ARM9™ Thumb® Family	ARM11
Family members Example	(a) ARM7TDMI® (Integer Core) (b) ARM7TDMI-S™ (Synthesizable version of ARM7TDMI) (c) ARM7EJ-S™ (Synthesizable core with DSP and Jazelle technology) and ARM720T™ (cached processor macrocell, 8K Cached Core with Memory Management Unit (MMU) supporting operating systems (OSes) Windows CE, Palm OS, Symbian OS and Linux)	(a) ARM920T (Dual 16 k caches with MMU support, OSes). (b) ARM922T (Dual 8 k caches for applications, support for multiple OSes). (c) ARM940T™ (Dual 4 k caches for embedded control applications running an RTOS)	Families with ARMv6 instruction set architecture that include the Thumb® extensions for code density, Jazelle™ technology for Java™ acceleration, ARM DSP extensions and SIMD media processing extensions, MMU support, OSes and Palm OS
Core with ARM® and Thumb® instruction sets	32-bit RISC core	32-bit RISC processor core super-scaling 5-stage integer pipeline, 8-entry write buffers. It avoids blocking the processor on external memory writes	32-bit RISC processor core with 8-stage integer pipeline, static and dynamic branch prediction, and separate load-store and arithmetic pipelines to maximize the instruction throughput
Application domain	Cost and power-sensitive consumer applications for example, Personal audio MP3, WMA, AAC players, entry level mobile phone, two way pagers, still digital camera, PDAs	Set-top boxes, home gateways, games consoles, MP3 audio, MPEG4 video videophones, portable communicators, PDAs, next-generation hand-held products, digital consumer products, imaging products, desktop printers, still picture cameras, digital video cameras, automotive telemetric and infotainment systems	Battery-powered and high-density embedded applications. Embedded SoCs of latest generation of wireless and consumer applications. Addresses the requirements of embedded application processors, advanced OSes and multimedia, such as audio and video CODECs. Consumer devices include 2.5G and 3G mobile phone handsets, PDAs and multimedia wireless

(Contd)

Feature	ARM7™ Thumb® Family	ARM9™ Thumb® Family	ARM11
			devices, home consumer applications such as imaging and digital camera applications, home gateway and network infrastructure equipment including voice over IP and broadband modem
Performance	130 MIPS using Dhrystone 2.1 benchmark in typical 0.13 μm process	Achieves 1.1 MIPS/MHz, 300 MIPS (Dhrystone 2.1) in a typical 0.13 μm process	Targets a performance range of Dhrystone 400 to 1200 MIPS
Code Density	High code density (comparable to a 16-bit microcontroller)	High code density	High code density
Die size on silicon	Small die size portable to 0.25 μm, 0.18 μm and 0.13 μm versions	Die Size 4.2 mm ² in ARM940T. Portable to latest 0.18 μm, 0.15 μm, 0.13 μm silicon processes. Frequency 185 MHz at 0.18 μm in ARM 940T.	0.13 μm foundry processes deliver 350 to 500+ MHz in worst case and over 1 GHz on next-generation 0.1μm processes
Memory Coupling (Section 6.3)	No tight coupling	No tight coupling	
Power Performance	Very low power consumption	Very low power consumption. 940 T power 0.8 μW/MHz on 0.18 μm silicon foundry generic process. Worst case: 1.62 V, 125°C, and slow silicon. Typical: 1.8 V, 25°C, nominal silicon	Optimum power efficiency, single-issue operation with out-of-order completion to minimize gate count, consuming less than 0.4 μW/MHz on 0.13 μm foundry processes
Bus Interface	AHB	Single 32-bit AMBA bus interface	None

- An enhancement of v5TE architecture is ARMv5TEJ architecture (introduced in 2000). It incorporates Jazelle Java execution accelerator technology for Java. This provides significantly higher Java codes execution by 8x performance than a software-based Java-Virtual-Machine (JVM). There is an 80% reduction in power consumption compared to non Java-accelerated core. This functionality gives platform developers a feature that the Java codes as well as OS applications can run on a single processor in an SoC or embedded system.
- An enhancement of v5TEJ architecture is with ARMv6 architecture (first implementation 2002), used in ARM11 microarchitecture. It has SIMD (Single instruction multiple data) extensions, optimized for applications including video and audio CODECs. SIMD execution performance is enhanced by 4x.

Exemplary Other High Performance Processors Intel XScale and StrongARM SA-1110, TI OMAP MIPS R5000 are other examples of high performance 32 and 32/64-bit processors. These have also been used in many applications in embedded systems.

Some processors are specially dedicated to a particular performance. For example, X10 family network processor delivers 10 Gbps port performance for IPv6 (broadband Internet). DSPs with high performances are SHARC, Tiger SHARC and TMS 64x described in following subsections.

2.3.4 SHARC

SHARC is processor architecture from Analog Devices. SHARC stands for super Harvard architecture single chip computer. Figure 2.19 shows the buses, ALUs, registers and memory in SHARC architecture.

SHARC is used in a large number of DSP applications. It has controlled power dissipation in floating point ALU. Different SHARCs can be linked by serial communication between them.

SHARC has following features:

- SHARC has 32-bit address space for accessing 16 GB or 20 GB or 24 GB as per the word size configuration in the memory. For 32-bit word size external memory configuration, addressable space is 16 GB.
- SHARC provides for two word size configurations—32-bit and 48-bit.
- SHARC has two full sets of 16 general-purpose registers. Therefore, context switching is fast. It thus enables multitasking OS and multithreading in programs easily.
- Registers are called R0 to R15 or F0 to F15 depending upon whether there are used for integer operation configuration or floating point configuration.
- The main registers are of 32-bit. A few registers are of 48 bits so that they may also be accessed as a pair of 16-bit and 32-bit registers.
- SHARC provides for a large ON chip memory of 1 MB. It has program memory and data memory Harvard architecture in ON chip memory.
- SHARC also provides for external OFF chip memory.
- OFF chip as well as ON-chip memory can be configured for 32-bit or 48-bit words.
- SHARC architecture allows program memory configurable for program memory and data memory sections.

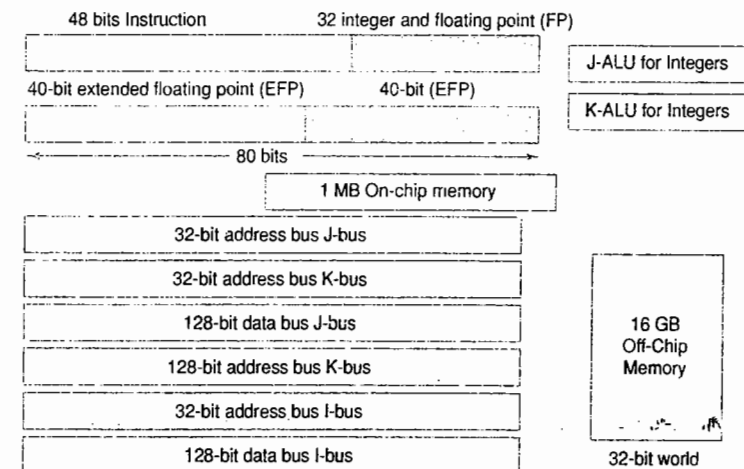


Fig. 2.20 Buses, ALUs, registers and memory in SHARC architecture

10. SHARC has instruction word of 48 bits and 32-bit data word for integer and floating point operations and 40-bit extended floating point. SHARC functions as a VLIW (very large instruction word) processor. The word size is 48-bit for instructions, 32-bit for integers and standard floating-point (FP), and 40-bit for extended floating-point (EFP). Smaller 16 or 8-bit must also store as full 32-bit data. Therefore, the big endian or little endian data alignment is not considered during processing
11. SHARC also provides instructions for saturation integer operations. For example, the integer after operation should limit to a maximum value. These instructions are required in graphic processing.
12. SHARC permits parallel operations. It supports processing instruction level parallelism as well as memory access parallelism. Therefore, there can be multiple data accesses in a single instruction.

TigerSHARC TigerSHARC is a highest performance density family of processors from Analog Devices. The architecture provides precision high-performance integrated circuits used in analog and digital signal processing applications. A version of TigerSHARC is TigerSHARC ADSP-TS201.

TigerSHARC is designed for multiprocessing applications and for peak performance greater than BFLOPS (billion floating-point operations per second). Multiple TigerSHARCs can connect by serial communication at 1 Gbps. ADSP-TS203SABP-050 processor processes using 250 MHz clock and on chip memory of 6 M bits and operates at 1.2 V/3.3 V. Low voltage design helps in processing with little power dissipation. Analog Devices TigerSHARCs have the highest performance per watt. A TigerSHARC version has 24 M bits ON-chip memory. TigerSHARC is available as the IP core also so that new applications with the core can be developed.

TigerSHARC finds applications in software baseband processing, 3G WCDMA baseband communication, cellular base stations and 14 Mbps HSDPA (High Speed Data Packets Access) networks for packet-based multimedia contents.

2.3.5 DSP

Advanced signal processor circuits consisting of MAC (Multiply and Accumulate) unit at a DSP provides fast multiplication of two operands and accumulates results at a single address. It computes fast an expression such as the following, $y_n = \sum (a_i x_{n-i})$, where the sum is made for $i = 0, 1, 2, \dots, N-i$. Here i, n and N are the integers, a is a coefficient, x is independent variable or an input element and y is the dependent variable or an output element.

DSP processors invariably have Harvard architecture. Caches are also organized in Harvard architecture separate I-cache and D-Cache).

Architecture of Digital Signal Processor The architecture of a DSP can be understood by considering an exemplary DSP of TMS320C64x™ DSP generation.

The main structural units in a TMS320C64x™ DSP generation and their functions are given in Table 2.4. Figure 2.21 shows the interconnections between twenty-five structural units by a block diagram for processor structure. Table 2.5 gives the additional structural units and the functions in the processors, TMS320C64x64™ VelociTI™, which is a VLIW architecture Extension.

2.4 PROCESSOR AND MEMORY ORGANIZATION

2.4.1 Processor Organization

Figure 2.22 shows a simple representation of organization of processor and memory in a system. The memory and IO devices interface the processor using buses. Figure 2.16 showed a detailed block diagram for internal

Table 2.4 Structural units and functions of processor in a DSP core

Structural Units in Core	Functions
<i>Basic units</i>	MDR, internal bus, data bus, address bus, control bus, bus interface unit, instruction fetch register, instruction decoder, control unit, instruction cache, data cache, multistage pipeline processing, multiliné superscalar processing for processing speed higher than one instruction per clock cycle, program counter similar to Table 2.2.
<i>Instruction dispatch</i>	For dispatch of instructions to the appropriate units.
<i>Control register</i>	Control registers associated with the control unit of the processor.
<i>Registers emulation unit</i>	Emulation
<i>Register File A</i>	Set of on-chip registers used during processing instructions in data path 1. These are named A0... A 15 and A16 ...A31. A register file is a file that associates with a unit such as ALU or FLPU.
<i>Register File B</i>	Set of on-chip registers used during processing instructions in data path 2. These are named A0... A 15 and A16 ...A31.
<i>Prefetch unit</i>	For fetching eight 32-bit instructions at each cycle.
<i>Processing unit</i>	Two multipliers and six arithmetical units, highly orthogonal, compiler and assembly optimizer, execution resources.
<i>Arithmetic logical subunit</i>	Subunit to execute arithmetical or logical instruction according to current instruction fetched at IR.
<i>Auxiliary Logic subunit</i>	A subunit used during subtraction. [Finds 2's complement before addition and then adds in order to subtract]
<i>Multiplier subunit</i>	Multiply
<i>Floating Point processing (FLP) subunit</i>	Subunit in C67x™ distinct from the ALU and performs FLP operations.
<i>Assembly Optimizer</i>	Optimizer for assembled codes
<i>C compiler</i>	Highly efficient compilation

Table 2.5 Additional structural units and functions of processors in TMS320C64x64™ VelociTI™ VLIW architecture extension

Structural Unit in Core	Functions
<i>Packed data processing</i>	8-bit or 16-bit data packed and processed as 32-bit data
<i>Parallel execution MAC units</i>	Quad 16-bit MAC/Octal 8-bit MAC [Table 2.2]
<i>Special instructions</i>	Broadband and image processing using VLIWs
<i>Level 2 cache</i>	Enhances performance of each fetch cycle
<i>Instruction packing unit</i>	Instructions packed as VLIW, which executes in parallel without in between halts

units of processor and showed the buses. A processor has an ALU. A processor circuit does sequential operations and a clock guides these. A processor has the program counter and stack pointer, which point to the instruction to be fetched and top of the data pushed into the stack, respectively. Certain processors have on-chip memory

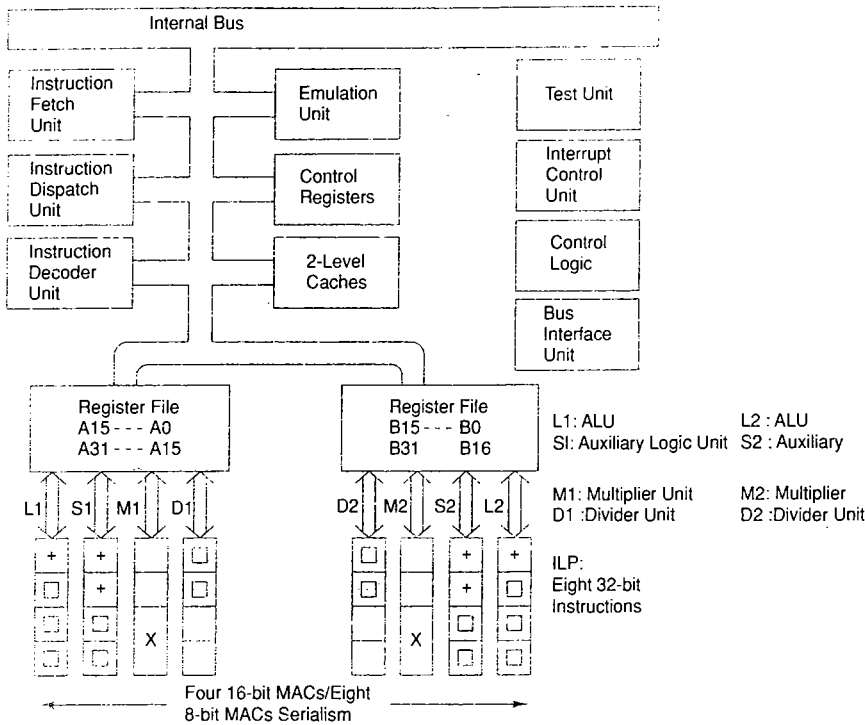


Fig. 2.21 Core and special structure units in DSP, TMS320C64x DSP
Note: Floating Point Units present in C67x

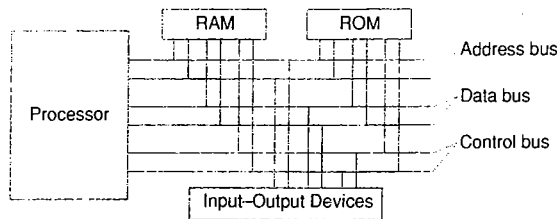


Fig. 2.22 A simple view of organization of processor, buses and memory in a system

management unit (MMU). A processor generally has general-purpose registers. Registers organize onto a common internal bus of the processor. A register is of 32-, 16- or 8 bits depending on whether the ALU performs at an instance 32- or 16- or 8-bit operation.

A processor may have CISC (Complex Instruction Set Computer) or RISC (Reduced Instruction Set Computer) architecture. A CISC has the ability to process complex arithmetic and logic as well as other

instructions and processes complex data sets using fewer registers, as it provides for a large number of addressing modes. An RISC executes simpler instructions and in a single cycle per instruction. New RISC processors, such as ARM 7 and ARM9, also provide for a few most useful CISC instructions also. CISC converges to an RISC implementation because most instructions are hardwired and implement in a single clock cycle.

A processor provides for the inputs for external interrupts so that the external circuits can send the interrupt signals (Section 2.2.4). The processor may possess an internal interrupt controller (handler) to program service routine priorities and to allocate vector addresses. The internal interrupt controller is of great help in most applications.

A processor may provide for bit manipulation instructions. These instructions help in easy manipulation of bits at the ports and memory addresses. Certain processors possess FLPU and FRS units that perform floating-point operations fast. These permit higher computational capabilities in the processor; they are essential for signal processing and sophisticated control applications.

Certain processors provide for direct memory access (DMA) controller with multiple channels on chip. When there are a number of I/O devices and an I/O device needs to access a multibyte data set fast, the system memory on-chip DMA controller is of great help. Section 4.8 will describe the DMA in detail.

Table 2.6 lists the nineteen features for the CISC family of microcontrollers and microprocessors.

Table 2.6 Features in four CISC microcontroller and processor families

Capability	Intel 8051 and Intel 8751	Motorola M68HC11E2 E2	Intel 80196KC	Intel Pentium
Processor instruction cycle in microseconds (typical)	1	0.5	0.5	0.001 ¹
Internal bus width in bits	8	8	16	64
CISC or RISC architecture	CISC	CISC	CISC	CISC with RISC feature ²
Program counter bits with reset value	16 (0x0000)	16 [(0xFFFE)]	16 (0x2080)	32 ³ (0xFFFF FFFF)
Stack pointer bits with initial reset value in case a processor defines these	8 (0x07)	16	16	32 ³
Atomic operations unit	No	No	No	No
Pipeline and super-scalar architecture	No	No	No	Yes
On-chip RAM and/or register file bytes ⁴	128 and 128 RAM	512 RAM	256 and 232	No
Instruction cache	No	No	No	8 kB ⁵
Data cache	No	No	No	8 kB ⁵
Program memory EPROM/EEPROM	4 k	8 k	8 k	No
Program memory capacity in bytes	64 k ⁶	64 k	64 k	4 GB
Data/stack memory capacity in bytes	64 k ⁶	64 k	64 k	4 GB
Main memory, Harvard or Princeton architecture (Section 2.4.2)	Harvard ⁶	Princeton	Princeton	Princeton
External interrupts	2	2	2	1 ⁷
Bit manipulation instructions	Yes	Yes	Yes	Yes
Floating point processor	No	No	No	Yes

(Contd)

Capability	Intel 8051 and Intel 8751	Motorola M68HC11E2 E2	Intel 80196KC	Intel Pentium
Internal Interrupt controller	Yes	Yes	Yes	No
DMA controller channels	No	No	1 (PTS) ⁸	4
On-Chip MMU	No	No	No	Yes

¹ It is maximum time in a typical Pentium 1 GHz version.

² Single clock-cycle hardwired implementation for most instructions implement like a RISC

³ Stack Pointer ESP 32 bits together with the Stack Segment ES 16 bits point to physical stack address at the memory. It equals $ES \times 0x10000 + ESP$.

⁴ This is in standard version. In other versions, it may be different.

⁵ This is in a typical version

⁶ Program and data memory spaces are separate in Intel 8051 family members. It is common in others.

⁷ Using the INTR pin and external programmable interrupt controller, up to 256 external interrupts can be handled

⁸ PTS means there is a Peripheral Transactions Server providing a DMA-like feature.

Table 2.6 shows the memory addresses in hexadecimal. Thus 0x10000 means a hexadecimal memory address 10000; 0x100FF means hexadecimal memory address 100FF. The same is the convention in C. It helps in distinguishing a decimal number from hexadecimal number.

2.4.2 Memory Organization

The memory system (consisting of various units) acts as a storage receptacle for data and programs. Most systems have two types of memory—*read-only memory* (ROM) and *random-access memory* (RAM). A flash memory functions as the ROM. Examples of uses of flash are mobile phone, mobile-computer and digital camera.

Read Only Memory As its name suggests, contents of the ROM does not modify during running of computer or on power off but may be read. In general, the ROM is used to hold a program that is executed automatically by the system every time it is turned on or reset. This program is called bootstrap, or boot loader, which instructs the system to load its operating system from its hard disk or other I/O storage device. The name of this program comes from the idea that the system is “pulling itself up by its own bootstraps” by executing a program that tells it how to load its operating system. An example of ROM is as follows: A system has ROM unit(s) for the bootstrap program(s), basic input-output system (BIOS) program(s) and vector addresses of the interrupts (Section 4.4.1).

Random Access Memory Random-access memory, on the other hand, can be both read and written, and is used to hold the programs, operating system and data required by the system. For example, a mobile phone has 128 kB or 256 kB of RAM to hold the stack and temporary variables of the programs operating system and data. RAM is generally volatile, meaning that it does not retain the data stored in it when the system's power is turned off. Any data that needs to be stored while the system power is off must be written to a permanent storage device, such as flash memory or hard disk.

Addresses Memory (both RAM and ROM) is divided into a set of storage locations, each of which can hold 1-byte (8 bits) of data. The storage locations are numbered, and an assigned number is called *address*. It defines in a memory of system which location the processor wants to reference at a given instance. One of the important characteristics of a computer system is the width of the address lines (bus) it uses, which limits the amount of memory that the processor can address. Most current computers use either 32-bit or 64-bit addresses,

allowing them to access either 2^{32} or 2^{64} bytes of memory. Assume that an IBM PC has 1 MB memory (1024×1024 bytes). Its bootstrap program and BIOS ROM addresses are between 15×2^{16} ($\equiv 0xF0000$) and $2^{20} - 1$ ($\equiv 0xFFFFF$). RAM addresses are between 1×2^{16} ($\equiv 0x10000$) and $15 \times 2^{16} - 1$ ($\equiv 0xEFFFF$).

Random Access Model of Memory A simple model for RAM and ROM both is the random-access model of memory when all memory operations take the same amount of time independent of the address of byte or word in memory. Assume that the memory system will support two operations: load (read operation into processor from memory) and store (read operation from processor into memory). The random access model states as follows: From the memory, a data byte, a word, a double word, or a quad word may be accessed from or at any addressable location, and a similar process is used to access from all locations. There is equal access time for a read or write that is independent of a memory address location. This mode differs from another model, called serial access model.

Store and Load (Write and Read) Instructions Most high performance organizations allow more than 1-byte of memory (generally four bytes) to be loaded or stored at one time. Generally, a load or store operation operates on a quantity of data equal to the system's bus width, and the address sent to the memory system specifies the location of the lowest-addressed byte of data word(s) to be loaded or stored. Each instruction mostly has the opcode followed by operands. Store operations need two operands, a value to be stored and the address in which that the value should be stored. They place the specified value in the memory location specified by the address.

Load operations need an operand that specifies the address containing the value to be loaded and return (fetch) the contents of that memory location into their destination (register), which is specified by another operand.

Using this model, the memory can be thought of as functioning similar to a large sheet of lined paper, where each line on the page represents a 1-byte storage location. To write (store) a value into the memory, we count down from the top of the page until we reach the line specified by the address, erase the value written on the line and write in the new value. To read (load) a value, we count down from the top of the page until we reach the line specified by the address, and read the value written on that line.

Alignment of Multibyte Store and Load in a Memory Organization Some memory organization requires loads and stores to be “aligned”. Assume that a 4-byte word has been aligned at address 0x000C or 0x1000, which is a multiple of 4. This simplifies the organization of the memory system as follows:

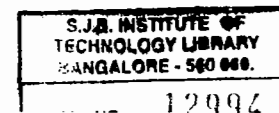
When a memory organization require loads and stores to be “aligned,” it means that the address of a memory reference must be a multiple of the size of the data being loaded or stored, so a 4-byte load must have an address that is a multiple of 4, an 8-byte store must have an address that is a multiple of 8, and so on. Other systems allow unaligned loads and stores, but take significantly longer to complete such operations than aligned loads.

ARM processor memory addresses are aligned either in multiples of four or two or one byte addresses. ARM permits three data types: four bytes word or two byte half word or 1-byte word, which stores at addresses in multiple of 4 or 2 or 1, respectively.

Example 2.9

- (a) Assume that a given memory organization require loads and stores to be “aligned”. Then a 32-bit system loads or stores 32 bits (4 bytes) of data with each operation into the 4 bytes that start with the operation's address, so a load from location 0x424 would return a 32-bit word containing the bytes in locations 0x0424, 0x0425, 0x0426 and 0x0427.

004.16 KAM



- (b) Assume that a given organization require loads and stores to be not aligned. A 32-bit system loads or stores 32 bits (4 bytes) of data with each operation into the 4 bytes that start with the operation's address, so a load from location 0x423 would return a 32-bit word containing the bytes in location 0x0423 0x0424 0x0425 and 0x0426, as in such organizations the store or load address can be any number, not necessarily a multiple of 2 or 4.

Little Endian and Big Endian in a Memory Organization Some processor and memory organizations require little endian and other big endian aligned multiple bytes when there is store into the memory or load into the processor from memory. The ARM processor permits programming at the start and enables a programmer to define one of two possible word-alignments, little endian or big endian, at the beginning. It is important to know how organization orders the bytes written at the memory.

- (a) In a little-endian system, the least-significant (smallest value) byte (8-bit) of a word (of 16 or 32-bit) is written into the lowest-addressed byte, and the other bytes are written in increasing order of significance.
- (b) In a big-endian system, the byte order is reversed, with the most significant byte being written into the byte with the lowest address. The other bytes are written in decreasing order of significance.

Example 2.10

1. Two different ordering schemes are used in modem computers: *little endian* and *big endian*. Assume that a word of 32 bits is 0x90ABCDEF, and the address where the word stores when written is 0x1000. The following shows an example of how a little-endian system and a big-endian system would write a 32-bit (4-byte) data word to address 0x1000.

Little-endian system and a big-endian system

Address	0x1000	0x1001	0x1002	0x1003
Little Endian	EF	CD	AB	90
Big Endian	90	AB	CD	EF

In general, programmers do not need to know the endianness of the system they are working on, except when the same memory location is accessed using loads and stores of different lengths. For example, if a 1-byte store of 0 into location 0x1000 was performed on the 32-bit systems in Example 2.10, a subsequent 32-bit load from 0x1000 would return 0x90ABCD00 on the little-endian system and 0x00ABCDEF on the big-endian system. Endianness is often an issue when transmitting data between different computer systems, as big-endian and little-endian computer systems will interpret the same sequence of bytes as different words of data. To get around this problem, the data must be processed to convert it to the endianness of the computer that will read it.

Figures 2.10 and 11 described the memory, processor and IO units organized on the buses. It can be safely concluded that the memory organization has a tremendous impact on computer system performance and is often the limiting factor on how quickly an application executes. Both bandwidth (how much data can be loaded or stored in a given amount of time) and latency (how long a particular memory operation takes to complete) are critical to application performance.

Other important issues in memory system design include protection (preventing different programs from accessing each other's data) and how the memory system interacts with the IO system.

There may be on-chip memories as RAM and/or register files, windows, caches and ROM in a micro-processor.

The caches are the integral parts of the memory-organization within a system. The software designer should enable the use of caches by an appropriate instruction, to obtain greater performance during the run of a section of a program, while simultaneously disabling the remaining sections in order to reduce the power dissipation and minimize energy requirements. Hardware designers should select a processor with multiway cache units so that only that part of a cache unit gets activated that has the data necessary to execute a subset of instructions. This also reduces power dissipation.

Processor Memory Organization: Princeton Architecture Figure 2.23(a) shows processor and memory organization in Princeton architecture. 80x86 processors and ARM7 have Princeton architecture for main memory. Vectors, pointers, variables, program segments and memory blocks for data and stacks have different addresses in the program in Princeton memory architecture.

Processor Memory Organization: Harvard Architecture Figure 2.23(b) shows processor and memory organization in Harvard architecture. A processor having Harvard main-memory architecture has distinct address spaces, control signal(s), processor instructions, and data paths for the bytes for data and for program. (The 8051-family microcontrollers have Harvard architecture.)

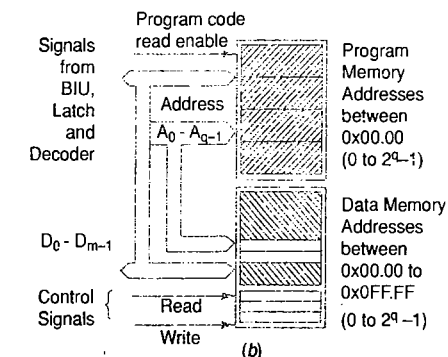
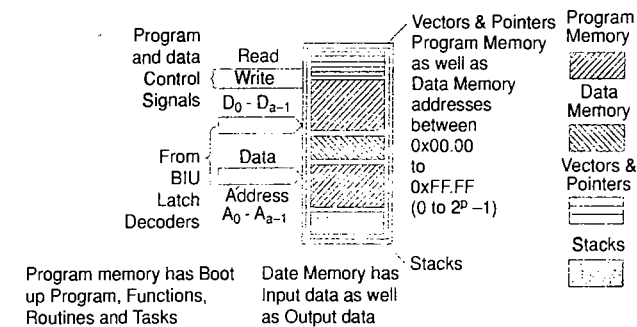


Fig. 2.23 (a) Processor and memory organization in Princeton architecture
(b) Processor and memory organization in Harvard architecture

Harvard architecture helps in handling streams of data that are required to be accessed in cases of single instruction multiple data type instructions and DSP instructions. Separate data buses ensure simultaneous accesses for instructions and data. Program segments and memory blocks for data and stacks have separate sets of addresses. Control signals and read-write instructions are also separate for accessing the program memory and data memory.

It must be remembered when coding in assembly and when organizing the main memories of certain processors, that their memory organization may be Harvard architecture. Program memory and data memory have separate set of addresses and have separate instructions for area accesses. A processor having Harvard architecture is needed for access to streams of data. Examples are (i) single instruction multiple data type instructions and (ii) DSP instructions.

For example, consider a DSP computation of the following expression in a 'Finite Impulse Response (FIR) filter'. An n -th filtered output sequence, $y_n = \sum(a_i x_{n-i})$, where the sum is made for $i = 0, 1, 2, \dots, N-1$. Here i, n and N are the integers. If $N = 10$, then for each value of y , first one of the 10 coefficients, a_i , and one of the 10 input sequences, x , are multiplied and then the summation is done. The total computations for all 10 values of n will need 100 multiplications and 100 summations. Storing and accessing the coefficients from a separate set of memory-addresses in a separate memory will allow fast access by using a separate set of buses.

2.5 INSTRUCTION-LEVEL PARALLELISM

Several instructions can execute in parallel. Two or more instructions can execute in parallel as well as in sequence in pipelines. In the instruction level parallelism (ILP), two parallel pipelines in a processor and two instructions I_n and I_{n+1} execute in parallel at separate execution units. Figure 2.24 shows instruction-level parallelism in pentium processor.

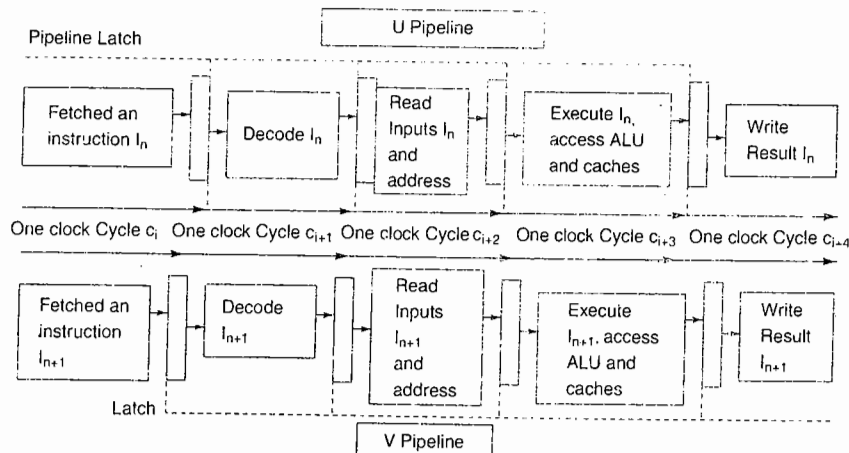


Fig. 2.24 Instruction level parallelism in a processor

2.5.1 Pipelined and Superscalar Units

High processor performance is required in many cases. For example, real-time signal processing. Pipelining and superscalar operations have now become essential. The hardware designer selects the processor as per the

required MIPS or MFLOPS performance. [A multi-processor system (Section 6.4.1) will be needed for very high performance requirements in mobile phones, digital camera, speech processing and video systems.]

Advanced processing units include instruction pipelining unit, which improves performance by processing instructions in multiple stages and the parallel units for superscalar execution, which improves performance on execution of two or more instructions in parallel execution units.

How do pipeline and superscalar units give such higher performance? Let us look at Example 2.11.

Example 2.11

Pipeline and Superscalar Execution

Step 1: Let us assume that the processor instruction cycle time is $0.02 \mu\text{s}$ (at 50 MHz operation) and that the processor executes an instruction in one clock cycle. The processor performance expected without advanced processing units will be 50 MIPS.

Step 2: Assume there is a three-stage pipeline as in ARM7. Let us, for the moment, ignore the effect of branching (called *branch penalty*). Three instructions will process in three clock cycles, but each clock cycle period can be made = 1/3 of the earlier period, as the division of processing units in stages divides the circuit also. The maximum expected performance of the processor without superscalar but with pipeline will be = 150 MIPS.

Step 3: Assume there is a two-line superscalar. Let us ignore the effects of unaligned data (*data dependency penalty*). Six instructions can process in single clock cycle with the three-stage pipeline and two superscalar units. The maximum performance will now be six times the processor cycle time, 300 MIPS.

We now explain the two terms used: branch penalty and data dependency penalty.

Branch penalty: If a branching instruction is encountered at a multistage pipeline, then the instructions executed in part at the preceding stages become redundant. These instructions have to be executed in full again later on after completion of the loop or return from a routine. The time required for re-processing these is called branch penalty.

Data dependency penalty: Assume that there are two instructions in two execution lines during a superscalar operation. Further, that one instruction depends on the data output of another. This is known as improper alignment. Thus, the two instructions are not aligned before putting them in separate lines. One instruction will now have to wait and cannot proceed further till the other instruction is executed. The waiting time is the data dependency penalty.

Superscalar processors possess hardware to extract instruction-level parallelism from sequential programs and possess hardware to efficiently take care of the collisions in execution unit and of data and control hazards.

During each cycle, the instruction issue logic of a superscalar processor examines the instructions in the sequential program to determine which instructions may be issued on that cycle. If enough instruction-level parallelism exists within a program, a superscalar processor can execute one instruction per execution unit per cycle, even if the program was originally compiled for execution on a processor that could only execute one instruction per cycle.

SHARC supports instruction level parallelism. The SHARC processor has instructions that are combined as single instruction word. [SHARC also supports memory parallelism; its processor has a modified Harvard structure called super Harvard structure (Figure 2.20). Multiple data can be fetched in a single instruction.]

ILP capability is one of the greatest advantages of superscalar processors and is the reason why virtually all high performance CPUs are superscalar processors. Superscalar processors can run programs that were originally compiled for purely sequential processors, and they can achieve better performance on these programs than processors that are incapable of exploiting the ILP.

Thus, users who work on new systems containing superscalar CPUs can install their old programs on those systems and see better performance on those programs than was possible on their old systems.

The use of high performance processor ICs and cores in embedded systems providing billion operations per second has become feasible due to the great advances in VLSI and in ILP and multi core processor design technology.

2.6 PERFORMANCE METRICS

Sophisticated embedded systems for high computing performance applications needs optimized use of resources, power, caches and memory. The following are the processor performance metrics:

1. (a) High MIPS, (b) high MFLOPS and (c) high *Dhrystone benchmark* program-based MIPS
2. Optimized compiler unit performance in the processor.

The above metrics are provided by the latest innovatively designed processors. A high-performance processor combines capabilities with optimized use of resources, power, caches, and memory.

A benchmarking program is called Dhrystone, developed in 1984 by Reinhold P. Weicker. It measures the performance of a processor for processing integers and strings (characters) both. It uses a benchmark program available in C, Pascal or Java. It benchmarks a CPU and not the performance of I/O or OS calls. Dhrystones per second is the metric used to measure the number of times the program can run in a second. 1 MIPS = 1757 Dhrystone/s. [Why? VAX11/780, which executed 1 MIPS, ran the Dhrystone benchmark program 1757 times (refer to <http://www.webopedia.com/TERM/D/Dhrystone.html>.)]

There is EDN Embedded Benchmark Consortium (EEMBC) [EDN is a group that publishes the International magazine EDN, which is dedicated to Embedded System information. Refer to <http://www.e-insite.net/edmag/>]. EEMBC proposed five-benchmark program suites for 5 different areas of applications of embedded systems: (a) Telecommunications, (b) Consumer Electronics, (c) Automotive and Industrial Electronics, (d) Consumer Electronics, and (e) Office Automation. These program suites are also used for measuring and comparing embedded system processor performances.

Different systems require different processor performance in terms of processing speed. A hardware designer takes these into view and selects an optimum performance-giving processor.

2.7 MEMORY-TYPES, MEMORY-MAPS AND ADDRESSES

2.7.1 Memory in a System

Section 1.3.5 introduced the memory in a system. A simple credit-debit transaction card may require just 2 kB of memory. On the other hand, a smart card for secure transactions when embedding a Java program for cryptographic functions may require 32 kB (typical value) memory. A complex embedded system may need huge memory.

The following subsections explain and look at certain important aspects of the memory. The various memory are described from the point of view of an embedded systems hardware or software designer.

ROM: Its Uses, Forms and Variants ROM non-volatility is a most important asset and it is extremely useful to embed codes and data in a system. ROM is a loosely used term. For a hardware designer, it may mean masked ROM, PROM, OTP-ROM, EPROM and EEPROM. In a strict sense, ROM means a masked ROM

made at a foundry from the programmer's ROM image file (Section 1.4.1). ROM that embeds the software or an application logic circuit is in one of the three forms: masked ROM, PROM and EPROM. When ROM is to be programmed during runtime and is to hold the processed result, either an EEPROM or flash memory is used.

(i) Masked ROM A masked ROM is built from a circuit that has r inputs (A_0 to A_{r-1}) and 8 outputs (D_0 to D_7). [Byte storing at an address is most common.] The circuit for masked ROM is one of a set of 2^r combinational circuits. Appropriate masking gives the desired set of outputs at each combinational circuit. Certain links fuse and others that are masked do not fuse. [A combination circuit is a circuit made up of logic gates with a distinct set of output logic states during distinct input logic states. It has a distinct truth table for r inputs \times 8 outputs. As soon as the inputs change (or withdraw), the output also changes in this circuit.]

The embedded software designer (after thorough testing and debugging) provides to a manufacturing foundry a file having a table of desired output bits for the various combinations of the input address bits. A program called *locator* creates this table. The manufacturer prepares the programming masks and then programs the ROM at a foundry. This ROM is returned to the system manufacturer.

Normally, one time masking charge could be very high. Generally, therefore, a system manufacturer will place the order, and the manufacturing foundry will accept the order for a minimum of 1000 pieces. The ROM is a cost effective solution to a bulk user of ROMs for the manufacture of embedded systems. An embedded system manufacturer using a masked ROM does not have to use a device programmer (ROM burner) each time a system ROM is made using EPROM or PROM or flash.

(ii) EPROM, E²PROM and OTP ROM Special versions of ROM can be programmed at the designer's or manufacturer's site for an embedded system with the help of a device programmer. One version is EPROM. It is an ultraviolet ray erasable and device programmer *Programmable Read Only Memory*. Erasing the device means restoring 1 at each bit in the cell arrays at each ROM address. Another version is E²PROM (EEPROM). It is an *Electrically Erasable, and Programmable Read Only Memory*. Examples of EPROM and EEPROM are 2732, a 4 kB EPROM, 28F256, a 32 kB EEPROM and 28F001 is 512 K \times 16-bit EEPROM.

EEPROM erasing during an application-program run is done by sending all eight data bus bits as 1s for the write in the presence of inputs of V_{pp} , called programming voltage and short duration write pulse. Sending 1s and 0s in the byte by a write instruction results in EEPROM programming during a program run. Erasing of a byte must precede the write. The processor with the system program can do erasing and writing, as it is similar to the writing in a RAM. What then, is the difference between the EEPROM and RAM? The difference is that in RAM, the read and write timing cycles are identical. Here, the write cycle has to be longer than in case of RAM, and it must succeed the erase of the byte by writing 0xFF. Further, an addition voltage V_{pp} signal is needed when erase and write occurs to the EEPROM. The number of times an EEPROM can be written is one million times plus. There is no limit for RAM, and a practically infinite number of writes is possible without first writing 1s in RAM.

Flash memory is a form of EEPROM in which a sector of bytes can be erased in a flash (very short duration corresponding to a single clock cycle). [Lately flashes of even ~ 3 V form and of capacity 2, 4 and 8 GB have become available even in card or stick form, which enables insertion in camera or pocket computer.] A sector can be from 256 B to 16 kB. The advantage over EEPROM is that the erasing of many bytes simultaneously saves time in each erase cycle that precedes the write cycles. The disadvantage is that once a sector is erased, each byte writes into it again one by one, and that takes too long a time. A new version of flash is **boot back flash**. A sector is reserved to store once only at the time of first boot. Later on it is protected from any further erase. In other words, it has an OTP sector also that can be used to store ROM images like in a ROM. Nowadays, flash has replaced EPROM in the systems.

PROM (an OTP ROM, a one time device programmer) is another form. A PROM once written is not erasable.

(iii) *Uses of ROM or EEPROM or Flash* Figure 1.5 showed what a ROM embeds—program codes for various tasks, interrupt service routines, operating system kernel, initialization (bootstrap program and data) and the standard data or table or constant strings.

An EEPROM is usable by erasing over one million times. It can be erased during runtime itself. Flash memory is usable about 10,000 times for repeated erasing followed by programming during the runtime. The PROM is written only once by a device programmer or the first system run.

Three examples of EEPROM memory applications are as follows. (i) Storing current date and time in a machine. (ii) Storing port statuses. (iii) Storing driving, malfunctions and failure history in an automobile for use by mechanics later on.

Three examples of flash memory applications are as follows. (i) Storing pictures in a digital camera. (ii) Storing voice compressed form in a voice recorder. [Recall of prerecorded message in a phone.] (iii) Storing messages and contacts in a mobile phone.

Examples of use of an OTP ROM are as follows. (i) Smart card identity number and user's personal information. (ii) Storing boot programs and initial data like a pictogram displaying a seal or monogram. (iii) ATM card or credit card or identity card. Once the various details are written at the bank and handed over to the account holder, there is no modification possible in the embedded PROM at the card. Just as a paper holds information permanently once written or printed, so also does a PROM.

A flash or PROM or ROM is not only used for program and data storage, but also for obtaining the preprogrammed logic outputs and output sequences for the given sets and sequences of inputs. [Inputs are given analogous to an address signal by the processor and outputs are obtained analogous to those obtained during a processor read cycle.] Assume that there are 8 inputs ($r = 8$). The truth table for it will have 256 combinations. 8×8 ROM can be programmed to generate 256 sets of 8-bit outputs for each combination. Examples of applications of preprogrammed logic outputs are as follows.

1. Used to hold language-specific bits for the fonts corresponding to each character in a printer.
2. Used to hold the image bits for a display. A pictogram generates from these bits. A ROM is used in a display circuit. It stores the bytes for the full bit-image corresponding to the pixels for a pictogram. Sequential changes at the inputs repeatedly generate the full pictogram.
3. In a CISC a control ROM at a micro-programmed unit is used. It stores sets of microinstructions for each processor instruction. Each set of microinstructions is stored in a sequence such that it specifies a set of signals for the various fetch and executing unit during execution of an instruction fetched from the memory.

RAM A system designer considers RAM devices of eight forms. These forms are 'SRAM', 'DRAM', 'NVRAM', 'EDO RAM', 'SDRAM', 'RDRAM', Parameterized Distributed RAM and Parameterized Block RAM.

(i) *Uses of RAM* RAM stores the variables during a program run and stores the stack (Section 1.3.5). It stores input and output buffers, for example, of speech or image. It can also store the application program and data when the ROM image is stored in a compressed format in an embedded system and decompression is done before the actual run of the system.

1. SRAM is used most commonly for designing caches and in embedded systems and microcontrollers.
2. DRAM is used mostly in high performance computers or high memory density systems.
3. EDO RAM is used in systems with buses to the devices when operating with clock rates up to 100 MHz; a zero-wait state is needed between two fetches, and there is single-cycle read or write.

4. SDRAM synchronizes the read operations and keeps the next word ready while the previous one is being fetched. This device is used when buses can fetch or send to the processor up to speed of 1 GHz.
5. RDRAM accesses in bursts the four successive words in a single fetch and thus gives above 1 GHz performance of the system.
6. Parameterized distributed RAM is the RAM distributes in various system subunits. IO buffers and transceiver subunits can have a slice of RAM each and the system stack can be at another slice. Distribution provides buffering of memory at the subunits before they are fetched and processed by the processor. It facilitates faster inputs from the IO devices than the processor system buses access the IOs using system memory.
7. Parameterised block RAM is used when a specific block of the RAM is dedicated for use by a subunit only, for example, MAC unit. A parameterized block RAM is used when an access by the system or IO or internal bus is slow compared to the processing speed of a subunit.

Different types of memory in varying capacities are available for use as per requirement. (1) Masked ROM or EPROM or flash stores the embedded software (ROM image). Masked ROM is for bulk manufacturing. (2) EPROM or EEPROM is used for testing and design stages. (3) EEPROM is used to store the results during the system program runtime. It is erased byte-by-byte and written during the system-run. It is useful to store modifiable bytes, for example, the runtime system status, time and date and telephone number. (4) Flash stores the results byte by byte during a system run after a full sector erase. (5) Flash is thus very useful when a processed image or voice is to be stored or a data set or system configuration data is to be stored, which can be upgraded as and when required. For example, a new image (after compressing and processing) can be stored and the old one erased from a sector in a single instruction cycle. (6) Boot block flash also has an OPT sector(s) to store the boot program and initial data or permanent system configuration data. It serves by storing the ROM image or its part in the OPT sector(s) and, at the same time, serves by storing as a flash in other sectors. (7) RAM is mostly used in SRAM form in a system. (8) Sophisticated systems use RAM in the form of a DRAM. EDO RAM, SDRAM or RDRAM. (9) Parameterized distributed RAM is used when the IO devices and subunits require a memory buffer and a fast write by another system. (10) Subunits like MAC when operating at fast speed use separate blocks of RAM.

2.7.2 Address Allocations in Memory

Figure 2.23 (a) showed memory addresses needed in the case of Princeton architecture in the system. Figure 2.23 (b) showed memory addresses needed in the case of Harvard architecture.

A *system memory allocation-map* is not only a reflection of addresses available to the memory blocks, and the program segments and addresses available to IO devices, but also reflects a description of the memory and IO devices in the system hardware. It maps guides to the actual presence of the memory at the various units. EPROM, PROM, ROM, EEPROM, flash Memory, SRAM (static RAM), DRAM (dynamic RAM) and IO devices. It reflects memory allocation for the programs, and data and IO operations by a *locator* program. It shows the memory blocks and ports (devices) at these addresses.

System IO devices map may be designed separately. This not only reflects the actual presence of the IO devices, but also guides the available addresses of the various device registers and port-data. [An example of a device is a timer. IO devices are the peripheral units of the system.]

The following are examples that describe memory allocation maps using the *locator*.

Example 2.12

Consider a memory map for an exemplary embedded system—a smart card needing a 2 kB memory, a 256 B RAM mainly for the stacks, EEPROM 512 B for storing the balance amount under credit or debit and the previous transaction records on the card. The memory locator or linker script program for this system to define a memory allocation map [Figure 2.25(a)] is as follows.

1. Memory
2. { ram : ORIGIN = 0x10000, LENGTH = 256
3. eeprom : ORIGIN = 0x20000, LENGTH = 512
4. rom : ORIGIN = 0x00000, LENGTH = 2K
5. }

Example 2.13

Consider a Java embedded card with software for encrypting and deciphering transactions. Assume that the system needs 32 kB ROM, RAM of 4 kB, and EEPROM 512B for storing not only the balance amount under credit or debit but also the cryptographic keys and previous transaction records on the card. So the memory locator or linker script program for this system defines memory map [Figure 2.25(b)] as follows.

Memory

1. { ram : ORIGIN = 0x10000, LENGTH = 4K
2. eeprom : ORIGIN = 0x20000, LENGTH = 512
3. rom : ORIGIN = 0x00000, LENGTH = 32K
4. }

One can also make the following important observation from Examples 2.12 and 2.13. There are memory address gaps between the origin of ROM, RAM and EEPROM in spite of the very small lengths of available memory. This gap is due to a design feature: the designer provides for expansion of these memories in future so no change will be needed in the interfacing decoder circuit between the memory and processor. Further, its software program has to make minimal changes. The changes will only be in length. This is because when there is no gap the origin will also change. This feature ensures that any future changes in the program code sizes and data sizes will not need change in the locator codes. One feature of a locator is also that it does not relocate the addresses of the special purpose ports that are dedicated to a particular IO task or dedicated to the device driver read and write operations.

The final step of the design process in an embedded system is that the bytes locate at the ROM from the image for the bootstrap (reset) program and data, the initialization data as well as the following standard data or table or constant strings device driver data and programs, the program codes for various tasks, interrupt service routines and operating system kernel. [The bootstrap program consists of the instructions that are executed on system reset. The bootstrap data example is for stack pointer initialization. The initialization data may be for defining initial state and system parameters. The constant strings may be for initial screen display.] There is a shadow segment in the ROM. The shadow segment has the initialization data, constant string, and the start-up codes that are copied into the RAM by a shadow segment copy program at system boot up. When a start-up code (booting) program is executed, a copy of the shadow segment from the ROM is generated in the RAM. The RAM also holds the data (intermediate and output data) and stack. A compressed program format locates at the ROM in case of large ROM image is required for the system program. This is because decompression program plus compressed image will need less memory than the large ROM image. The start-

up code task is to generate the decompressed program codes and store them into the RAM before system starts other programs. The processor executes all other programs subsequently by fetches from the RAM.

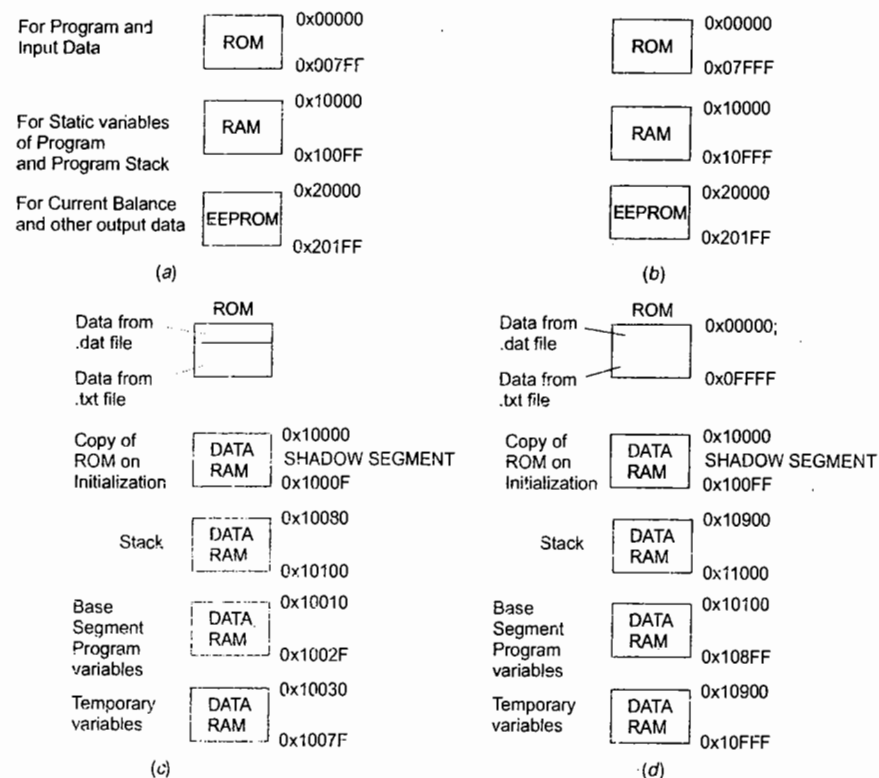


Fig. 2.25 Examples of Memory maps in embedded systems—(a) smart card needing 2 kB memory (b) Java embedded card with software for encrypting and deciphering transactions (c) memory map sections in a smart card (d) memory map sections in another smart card

A processor may have predefined memory locations for the initialization boot record. For example, in 80960, boot record consists of 12 words. The record is stored at ROM addresses, 0xFFFFF00 to 0xFFFFF2C.

Example 2.14

Consider memory map for an exemplary card in which there are sections at the memory allocation map. Consider its description in a locator program. The sections for an exemplary embedded system, smart card memory map [Figure 2.25(c)] may be defined as follows.

1. SECTIONS

```

2.  { /* Stack Top Location for 128 B RAM*/
3.  _TopOfStack = 0x10100;
4.  /* Bottom of Heap */
5.  _BottomOfHeap = 0x10080;
6.  text rom :
7.  { /* Debit-credit card program instructions are at the text file
   named here*/
8.  * (----.txt)
9.  }
10. data ram :
11. {
12. /* Shadow Segment for 16 byte of Initialised Data at RAM for a copy
   from ROM from */
13. _DataStart = 0x10000;
14. /* Debit-credit card shadow segment data at the data file named
   here*/
15. * (----.data)
16. _DataEnd = 0x1000F;
17. }
18. /* Command for copy into the RAM */
19. > rom
20. bss :
21. {
22. /* Base Segment for 32 byte of Program Variables Data at
   RAM */
23. _bssStart = 0x10010;
24. /* Smart card base segment data at the base segment data
   file named here*/
25. * (----.bss)
26. _bssEnd = 0x1002F;
27. }
28. }

```

Example 2.15

Consider another memory map [Figure 2.25(d)] for another exemplary card. The locator specifies different map sections as follows:

SECTIONS

```

1.  { /* Stack Top Location for 4 kB RAM*/
2.  _TopOfStack = 0x11000;
3.  /* Bottom of Heap */

```

```

4.  _BottomOfHeap = 0x10900;
5.  text rom :
6.  { /* Encrypting Java Card program instructions are at
   the text file named here*/
7.  a. (----.txt)
8.  data ram :
9.  {
10. /* Shadow Segment for 256 bytes of Initialized Data at RAM for a copy
   from ROM from */
11. _DataStart = 0x10000;
12. /* The card shadow segment data at the data file named here*/
   a. (----.data)
13. _DataEnd = 0x100FF;
14. }

   /* Command for copy into the RAM */
15. a. rom
16. bss :
17. {
18. /* Base Segment for 2 kB Program Variables Data at RAM */
19. _bssStart = 0x10100;
20. /* Java card base segment data at the base segment data file
   named here*/

21. a. (----.bss)
22. _bssEnd = 0x108FF;
23. }
24. }

```

The memory map that includes the device IO addresses is designed after appropriate address allocations of the pointers, vectors, data sets and data structures. If the main memory is of Harvard architecture, the program memory map will be separate. For example, 8051 reads from the program memory by a separate set of instructions (input-output instructions).

2.8 PROCESSOR SELECTION

A hardware designer must take into account following processor-specific features:

1. A processor, which can operate at higher clock speed, processes more instructions per second.
2. A processor gives high computing performance when there exist (a) Pipeline(s) and superscalar architectures, (b) pre-fetch cache unit, caches, and register-files and MMU and (c) RISC architecture.
3. A processor with register-windows provides fast context switching in a multitasking system.
4. A power-efficient embedded system requires a processor that has programmable auto-shut down feature for its units and programmability for disabling use of caches when the processing need for a function

or instruction set is not constrained by limit on execution deadline. Processor uses Stop, Sleep and Wait instructions, and special cache design.

5. A processor that has a burst mode accesses external memories fast, reads fast and writes fast.
6. A processor with an atomic operation unit provides hardware solution to shared data problems when designing embedded software, else special programming skill and efforts are to be made when program uses shared variables and data buffers among multiple tasks.
7. When coding in assembly language or designing compiler or locator, data may store in big-endian mode in a system and the lower order bytes store at higher address: for example, in Motorola processors. Data may also store in little-endian mode in a system. Lower order bytes store at lower addresses and vice versa: for example, in Intel processors. A processor may also be *configure* at the initial program stage big-endian or little-endian storage of words: for example, the ARM processors.

The StrongArm family processors from *Intel* and TigerSHARC from *Analog Devices* have high power efficiency features.

The processor selection processes can be understood by considering four representative cases. Firstly a design-table similar to Table 2.7 is built. Then a processor having the required structural units and capable of giving the desired processor performance in system is chosen.

1. *Case 1*: Systems in which processor instruction cycle time $\sim 1 \mu\text{s}$ and on-chip devices and memory can suffice. Examples are automatic chocolate vending machine, 56 kbps modem, robots, data acquisition systems like an ECG recorder or weather recorder or multipoint temperature and pressure recorder and real-time robotic controller.
2. *Case 2*: Systems in which processor instruction cycle time ~ 10 to 40 ns required, on-chip devices and memory do not suffice and medium processor performance is required. Examples are 2 Mbps router, image processing, voicdata acquisition, voice compression, video decompression, adaptive cruise control system with string stability and network gateway.
3. *Case 3*: Systems in which instruction cycle times of 5 to 10 ns required and high MIPS or MFLOPS performance is needed. Examples are multiport 100 Mbps network transceiver, fast 100 Mbps switches, routers, multichannel fast encryptions and decryptions systems.
4. *Case 4*: Systems in which instruction cycle time of even 1-ns does not suffice and multi-processor system is required along with use of the floating point and MAC units. Examples are voice processing, video processing, realtime audio or video processing and mobile phone systems.

Different systems require different processor features. A hardware designer takes these into view and selects an optimum performance-giving processor.

2.8.1 Microcontroller Selection

There are numerous versions of 8051. Additional devices and units are provided in these versions. A version and microcontroller is selected for embedded system design as per the application as well as its cost.

1. Embedded system in an automobile, for example, requires a CAN bus (Section 3.10.2). Then a version with CAN bus controller is selected.
2. An 8051 enhancement 8052 has an additional timer.
3. Philips P83C528 has I²C serial bus (Section 3.10.1).
4. 8051 family member 83C152JA (and its sister JB, JC and JD microcontrollers) has two direct memory access (DMA) channels on-chip. (Section 4.8) The 80196KC has a PTS (Peripheral Transactions Server) that supports DMA functions. [Only single and bulk transfer modes are supported, not the burst transfer mode.] When a system requires direct transfer to memory from external systems, the DMA controller, improves the system performance by providing for a separate processing unit for the data transfers from and to the peripherals.

Table 2.7 Essential processor capabilities in four exemplary set of systems

Processor capability Required	Case 1: <i>Automatic Chocolate Vending Machine, Data Acquisition System, Real time Robotic Control</i>	Case 2: <i>Voice data acquisition, Voice-data Compression, Video Compression, Adaptive Cruise Control System with String Stability, Network Gateway</i>	Case 3: <i>Multi-port Network Transceiver, Fast Switches, Routers, Multi channel Fast Encryptions and decryptions</i>	Case 4: <i>Voice Processor, Video processing and Mobile Phone Systems</i>
Required processor	Microcontroller	Microprocessor	Multiprocessor System	Microprocessor +DSP based Multiprocessor System
Processor instruction cycle in μs (typical)	~ 0.5 to 1	0.01 – 0.04	0.0005 – 0.001	0.001 – 0.0005
Processor performance	Low suffices	Medium to high	High	Very High
Internal bus width in bits	8	32	32	64
CISC or RISC architecture	Any	RISC	RISC	RISC
Program counter and stack pointers	16	32	32	32
Stack at external or internal memory	External	External or internal	Internal	Internal
On-chip atomic operations unit	–	–	Yes ¹	–
Pipelined and super-scalar and pipelined architecture	No	Yes	Yes	Yes
Off-chip RAM in view of excessive RAM needs	No, on-chip suffices	Yes	Yes	Yes
On-chip register windows and files due to fast context switching needs	No	Yes	Yes	Yes
Interrupts handler internal in micro-controller or external to processor	Internal microcontroller	External	External	External

(Contd)

Processor capability Required	Case 1: Automatic Chocolate Vending Machine, Data Acquisition System, Real time Robotic Control	Case 2: Voice data acquisition, Voice-data Compression, Video Compression, Adaptive Cruise Control System with String Stability, Network Gateway	Case 3: Multi-port Network Transceiver, Fast Switches, Routers, Multi channel Fast Encryptions and decryptions	Case 4: Voice Processor, Video processing and Mobile Phone Systems
Instruction and data caches and MMU	No	Yes	Yes	Yes
On-chip memory flash or EPROM	Yes, on-chip suffices	No, on-chip does not suffice	No, on-chip does not suffice	No, on-chip does not suffice
External interrupts	1 to 16	1-2	128-256	16-32
Bit manipulation instructions	Used	Heavily used	Heavily used	Heavily used
Floating point processor	No	Yes	No	Yes
Streams of data requiring Harvard main memory architecture	No	Mostly Not necessary	May be Yes	Invariably Yes
DMA controller channels	No	No	Yes	May be Yes
Exemplary processor family	8051, 68HC11 or 12 or 16, 80196, PIC16F84	80x86, 80860, 80960	ARM7, Sunspare	ARM9, TMS family DSPs, PowerPC

¹Needed when multiple ports and multichannel operations need data sharing.

Example 2.16 Case Study of Voice Data Compression System

1. A robotic system motor needs signalling at the rate above 50 to 100 ms. Hence there is enough time available for signalling and real-time control of multiple motors at the robot when we use a processor with instruction cycle time $\sim 1 \mu\text{s}$.
2. The processor speed need not be very high and performance needed is much below 1 MIPS. So no caches and advanced processing units like pipeline and superscalar processing are required.
3. A four-coil stepper motor needs only a 4-bit input and a DC motor needs a 1-bit pulse width modulated output. Therefore an 8-bit processor suffices.
4. Frequent accesses and bit manipulations at IO ports are needed. CISC architecture therefore suffices.
5. The program can fit in 4 kB or 8 kB of internal ROM on-chip. Stack sizes needed in the program are small so that can be stacked in an on-chip 256 or 512-byte RAM. A microcontroller is thus needed. No floating-point unit is needed.

Microcontrollers appropriate for the above case are 8051, 68HC11, 68HC12, 68HC16 or 80196. Microcontrollers 68HC12 and 68HC16 can be used due to availability of large number of ports. The 68HC12's instruction cycle and clock cycle time equals $0.125 \mu\text{s}$. Number of ports equals 12 in 68HC12. Therefore, 6 or more degree of freedom robot with 6 or more motors can be driven directly through these ports. STOP and WAIT instructions in the processor save power when the robot is at rest!

Example 2.17 Case Study of Voice Data Compression System

1. Voice signals are pulse-code modulated. The rate at which bits are generated is 64 kbps. A suitable algorithm can process the data compression of these bits with an instruction cycle time of ~ 0.01 to $0.04 \mu\text{s}$ (100 to 25 MHz) when the processor uses advanced processing units and caches.
2. Let us assume that the processor instruction cycle time is $0.02 \mu\text{s}$ (50 MHz). With a three-stage pipeline and two-line superscalar architecture, the highest performance will be 300 MIPS. [Refer to Example 2.11 for an understanding of the computations of MIPS]. It suffices for not only for voice but also for video compression.
3. Frequent accesses and complex instructions may not be needed.
4. The program cannot fit in 4 kB or 8 kB of internal ROM on-chip, and stack sizes needed in the program are big. Instead large ROM and RAM as well as caches are needed.
5. No floating-point is needed as mostly the bit manipulation instructions are processed during compression.

Exemplary processors that are appropriate for the above case are 80x86 and ARM family processors.

Example 2.18 Case Study of Real-time Video Processing

1. Transfer rates of 100 MHz plus are needed in fast switches on a network. Assuming 10 instructions per switching and transceiver action, instruction cycle time is ~ 0.001 plus. A multiprocessor system is needed for GHz transfer rates.
2. Let us assume that the processor instruction cycle time is $0.01 \mu\text{s}$ (100 MHz). With a five-stage pipeline and two-line superscalar architecture, the highest performance will be 1000 MIPS. [Example 2.11]. Multiprocessor system is thus needed for 1000 MHz plus switches.
3. The processor should have RISC architecture for single cycle instruction processing at each stage and line.
4. ROM and RAM as well as caches are required.
5. No floating-point is needed as mostly the bits are processed for IOs.

Exemplary processors that are appropriate for the above case are ARM7, ARM9 and Pentium.

Example 2.19 Real-time Video Processing

1. Real-time video processing requires fast compression of an image needing use of DSP. Many real-time tasks have to be processed: for instance, scaling and rotation of images, corrections for shadow, colour and hue, image sharpening and filter functions. In such cases, a multiprocessor system with DSP(s) and that has the best processing performance is required.

Exemplary processors that are appropriate for multiprocessor system are ARM9 integrated with TMS family DSP(s) or ARM11 or TigerSHARC.

2.9 MEMORY SELECTION

Once the software designer's coding is over and the ROM image file is ready, the hardware designer is faced with the questions of what type of memory and what size of each should be used. First a design-table, as in Table 2.8, is built. The memory having the required features and address space is chosen. Following are the case studies. The actual memory requirement is known only after coding as per the design functions and specifications. ROM and RAM allocations for various segments, data sets and structures will be available from the software design. However, a prior estimate of the memory type and size requirements can be made. [Remember, the memory are available as: 1 kB, 4 kB, 16 kB, 32 kB, 64 kB, 128 kB, 256 kB, 512 kB and 1 MB. Therefore, when 92 kB of memory is needed, then a device of 128 kB is selected.]

Example 2.20

(a) Case Study of an Automatic Washing machine

Consider an automatic washing machine system. Assume that machine is not saving the pictures and graphics. (a) An EEPROM's first byte is required to store the state (wash, rinse cycle 1, rinse cycle 2 and drying) that has been completed. The second byte is required to store the time in minutes already spent at the current stage. The third byte is needed to store the status of the user set buttons. Thus a 128 B EEPROM at best should suffice in microcontroller. (b) Embedded software can be within 4 kB ROM at the microcontroller. (c) RAM is needed only for a few variables and stacks. An internal RAM of 128 B should suffice. (d) Therefore, no external memory is required with the system when using a microcontroller.

(b) Case Study of a Robotic

Consider a robotic system. (a) EEPROM bytes are required to store the rest status of each degree of freedom. Thus 512 B EEPROM in the microcontroller at best should suffice. (b) Embedded software can be within 32 kB ROM in the microcontroller. (c) RAM need is only for the variables and only one stack is needed for the return address of the subroutine calls. Internal RAM of 512 B should suffice. Therefore, no external memory is required with the system when using a microcontroller.

Example 2.21

(a) Case Study of the Data Acquisition Systems for the sixteen-parameter channels and voice/image processing during acquisition

Consider a data acquisition system. Assume that there are sixteen channels and at each channel 4 B of data store every minute. (a) Bytes are to be stored in flash memory. Assume that the results are stored in flash memory for a day before it is printed or transferred to a computer. Thus 92 kB is the data acquired per day. A 128 kB flash memory will thus suffice. (b) Embedded software can be within 8 kB ROM in the microcontroller. (c) RAM is needed only for the variables and only one stack is needed for the return address of the subroutine calls. An internal RAM of 512 B will suffice. (d) Intermediate calculations are needed for storing ADC results in the proper format. Unit conversion functions need to be calculated, which may necessitate a RAM of about 4 kB to 8 kB. (e) Therefore, a microcontroller with 8 kB EPROM and 512 B RAM is required, and an external flash (or 5 V EEPROM) of 128 kB and external RAM of 4 to 64 kB are required with the system. For acquiring image or voice data on-line, RAM buffer requirement can be 512 MB.

(b) Case Study of the Data Acquisition Systems for the ECG waveforms

Consider another data acquisition system, which is used for recording the ECG waveforms. Let each waveform be recorded at 256 points. A 64 kB flash will be required for 256 patient records.

Table 2.8 Required memory in four example of systems

Memory Required	Case 1: Automatic Chocolate Vending Machine or Real time Robotic Control system	Case 2: Data Acquisition System	Case 3: Multi-port Network Transceiver, Fast Switches, Routers, or Multi channel Fast or Encryption and decryption system	Case 4: Voice Processor or Processor or Video processing or Mobile Phone or Pocket PC System	Case 5: Digital Camera or Voice Recorder System
Processor used	Micro- controller	Micro- controller	Multiprocessor system	Microprocessor + DSP-based Multiprocessor system	Micro- processor
Internal ROM or EPROM	4 to 32 kB	8 kB	—	—	—
Internal EEPROM	256 to 512 B	256 to 512 B	—	—	—
Internal RAM	256 to 512 B	256 to 512 B	—	—	—
ROM or EPROM device	No	No	64 kB	64 MB	64 kB
EEPROM or Flash device ¹	No	64 to 128 kB	512 B	32 kB to 256 kB 2 to 8 GB memory stick	Flash 16 MB to 8 GB memory stick
RAM device	No	64 kB to 512 MB	64 kB to 512 MB	8 MB	1 MB
Parameterised distributed RAM	No	No	Yes for IO buffers, 4 kB per channel	—	—
Parameterised Block RAM	No	No	—	Yes for MAC unit, Dialing IO unit	—

Note: ¹Flash with a boot block can be used to store the protected part of the boot program in its OTP sector(s).

Example 2.22 Case Study of a multichannel Fast Encryption cum decryption Transceiver Systems

1. Consider a system with multiple channels. There are encrypted inputs at each channel. These are decrypted for retransmission to other systems.
2. EEPROM is required for configuring ports and storing their statuses. Assume 16 channels. 512 kB will suffice for a 16 B need per channel.
3. Encryption and decryption algorithms can be in 64 kB ROM.

4. Multichannel data buffers are required before the caches process the algorithms. Therefore, 1 MB to 512 MB RAM may be required.
5. IO buffer storage of 4 kB per channel is needed. If a parameterised distributed RAM is employed at each channel, the system performance will be increased.
6. The system will thus need the following memory: 64 kB ROM, 512 B EEPROM, 1 MB RAM and 4 kB per channel distributed parameterised RAM.

Example 2.23 Case Study of a Mobile Phone system

1. As voice compression–decompression and encryption–decryption algorithms and DSP processing algorithms are required, the ROM image will be large. Assume it can be taken as 64 MB. Now if the ROM image is stored in a compressed format, a boot-up program first runs a decompression program. The decompressed program and data first load at RAM and then the application program runs from there. The RAM is obviously of bigger size in these systems. ROM can be reduced as per compression factor.
2. A large RAM is also needed. It can be taken as 8 MB for storing the decompressed program and data and for the data buffers.
3. The phone memory for entering important telephone numbers can be in a 16 kB flash or EEPROM. A flash of 64 kB can be taken for recording messages, MMS pictures. A memory stick using SDIO port of 2 GB or 8 GB is required for recording songs and videos.
4. Parameterised block RAM at MAC subunit and other subunits will improve system performance.
5. The system will thus require memory of 1 MB ROM, 16 kB EEPROM, 16 kB Flash, 1 MB RAM and block RAM at the subunits.

Example 2.24 Low-resolution uncoloured digital camera system

1. Assume a low-resolution uncoloured digital camera system. Images are to be recorded GIF (graphic image format) compressed format. (a) Assume that an image has a Quarter-CIF (Common Intermediate Format) of 144 x 176 pixels. Then, 25,344 pixels are to be stored per image. Assume that compression reduces the image by a factor of 8 then 3 kB per image will be needed. Flash required will be 0.2 M for 64 camera images. Therefore, a 256 kB flash will be required. (b) A 64 B digital uncoloured images camera system will thus be estimated to need memory of 64 kB ROM, 256 kB flash and 1 MB RAM. High resolution 6 M pixel digital camera need 16 MB 256 MB flash and 2 to 8 GB memory stick.
2. Assume a voice recording system. 64 kbps are required, assuming an 8-bit pulse code modulation of the voice signals. [Average frequency is taken as 8 kHz.] Assume voice data compression factor of 8, 1 kB flash is required per second. A 1 MB flash 4 required for each hour of recording.
3. Since voice compression–decompression algorithms have to be processed, the ROM image will be large. It can be taken as 1 MB. Using compression techniques, a 64 kB ROM can store the ROM image.
4. The RAM needed is large for storing the decompressed program. It can therefore be estimated as 1 MB.
5. A one-hour voice recorder system is estimated to require memory or 64 kB ROM, 4 MB flash and 1 MB RAM.

§ Simple systems like automatic chocolate vending machines or robots needs no external memory. The designer selects a microcontroller that has an on-chip memory required by the system. The data acquisition system needs EEPROM or flash. A mobile phone or pocket PC or digital camera system needs 1 MB plus RAM device and 64 kB to 256 kB internal flash device plus memory stick. Image or voice or video recording systems require a large flash memory in the form of memory stick of 2 GB to 8 GB.



Summary

- 8051 microcontroller has Harvard architecture for memory, has special function registers, internal RAM and ROM or flash. It has two timers, and SI interface for the half duplex synchronous and full duplex UART communication.
- Bus signals interface the processor, memory and devices. The interface circuit takes into account the timing diagram with reference to processor clock output. The circuit uses the processor control, address and data bus signals and takes into account the timing diagram for the bus signals. A PAL-, GAL- or FPGA-based circuit, called glue circuit, provides a single-core or chip solution for the latches, decoders, multiplexers, demultiplexers and other necessary interfacing circuits.
- The buses interface to stepper motor, LCD controller, A/D, D/A circuits using ports and appropriate interface circuit.
- An embedded system hardware designer must select an appropriate processor, appropriate set of memories for the system and design an appropriate interfacing circuit between the processor, memories and IO devices. This is done after taking into account the various available processors, structural units and architecture, memory types, sizes and speeds, bus signals and timing diagrams.
- The structural units of a processor that interconnect through a bus are memory address and data registers, system and arithmetic unit registers, control unit, instruction decoder, instruction register and arithmetic and logical unit. *Registers in processor* also called register set(s) window(s) or file(s) are important and are meant for various functions such as context switch to process another task or ISR.
- Advanced processors have following additional structural units—prefetch control unit, instruction queuing unit, caches for instruction, data and branch transfer, floating point registers and floating point arithmetic unit. Pipelining and superscalar features and caches in the processors are used in high performance systems. MIPS or MFLOPS or Dhrystone per second define the computing performance. The goal is to provide optimal computing performance at the least cost and power dissipation.
- ARM, SHARC, TigerSHARC and DSPs processors are used in high performance computations.
- A system needs ROM and RAM memory of various types and address spaces. Various forms of ROM are masked ROM, PROM, EPROM, EEPROM, flash, boot-back flash and memory stick or card. Basic details of the memories are the addresses available, speed for read and write operations, and modes of memory access.
- The Processors and Memory selection can be done using appropriate design tables.
- Each IO device has a distinct set of addresses. Each IO device also has a distinct set of device registers – data registers, control registers and status registers. At a device address, there may be more than one register. The device addresses depend on the system hardware. Based on the memory map with IO device addresses, a locator program is designed to locate the linked object code file and generate a ROM image.



Keywords and their Definitions

- Absolute addressing mode** : Define all the address bits in an instruction.
- Accelerator** : ASIC, IP core or FPGA, which accelerates the code execution and which may also include the bus interface unit, DMA, read and write units and registers with their cores.
- Accumulator** : A register that provides input to an ALU and that accumulates a resulting operand from the ALU.
- AHB** : A high-performance version of the AMBA used in ARM processors.
- ALU** : A unit to perform arithmetic and logic operations as per the instructions.
- AMBA** : An established open source specification for on-chip interconnects that serves as a framework for SoC designs and IP library development.
- Arithmetic unit registers** : Registers that hold the input and output operands and flags with the ALU.
- ARM** : A family of high performance reduced code density ARM7, ARM9, ARM10 and ARM 11 processors, which are used in embedded systems as a chip, or as a core in an ASIC.
- ARM7 and ARM9** : Two families of RISC processor for an SoC from ARM and Texas Instruments, also available in single-chip CPU versions and in file versions for embedding at a VLSI chip. ARM 7 has Princeton architecture for the main memory and ARM9 has Harvard Architecture.
- Asynchronous serial communication** : Data bytes or frames not maintain uniform phase differences in serial communication.
- Auto index** : When after executing an instruction, the index register contents change automatically.
- Base addressing** : Addressing an address from where a first element of data structure starts.
- Baud Rate** : Rate at which serial bits are received at the line during a UART communication.
- Boot back flash** : A flash with a few sectors similar to an OTP device, to enable storage of bootup program and data.
- Branch transfer cache** : Cache to hold in advance the next set of instructions to be executed on the program branching to this set.
- Burning-in** : A process in which bits are modified from all 1s in erase form to the 1s and 0s in a device as per input 1s and 0s.
- Bus interface unit** : A unit to interconnect the internal buses with the external buses for control, address and data bits.
- CISC** : A complicated Instruction Set Computer that has one feature that provides a big instruction set for permitting multiple addressing modes for the source and destination operands in an instruction. The hardware executes the instructions in a different number of cycles, as per the addressing mode used in an instruction.
- Code Compatibility** : Usability of codes by various generations of a family.

- Code composer studio** : An IDE for TI DSP-specific code composing which provides an environment similar to MS Visual C++. It consists of the following: multilevel (C as well as DSP assembly) debugger, compiler, assembly optimiser, RISC-like assembly codes and RISC-like scheduling for optimum performance and efficiency probe points, file IO functions, comprehensive data visualization displays and GEL scripting language based on C.
- CODEC** : A unit for digital coding after ADC and other operations and decoding to get analog signals using DAC and other operations. It is used in processing audio or CCD device pixels and video signals.
- Control unit** : To control and sequence all the processing actions during an instruction execution.
- DAA** : Direct access arrangement; for example, a typical DAA serial in and out port directly transferring the analog input and output using up to 1 master and 7 slave CODECs.
- Data cache** : Cache to hold the data in content-addressable memory format.
- DCT** : Discrete Cosine Transformation function used in a number of DSP functions, for example, the MPEG2/MPEG4 compression.
- Device address** : A device address used by processor to access its set of registers. At each address there may be one or more device registers.
- Device programmer** : A system or unit for programming a device by burning-in ROM image.
- Device programming** : Programming of bits by burning-in a memory of microcontroller or in a PLA, PAL, CPLD or any other device.
- Device register** : A register in a device for byte, word of data, flags or control bits. Several device registers may have a common address.
- Device** : A physical or virtual unit that has three sets of registers: data registers, control registers and status registers, and which the processor addresses it like a memory.
- Dhrystone** : A benchmarking program that measures the performance of a processor for processing integers and strings (characters). It uses a benchmark program available in C, Pascal or Java. It benchmarks a CPU, not the performance of the IO or OS calls. 1 MIPS = 1757 Dhrystone/s.
- Digital Filtering** : A filter for the signals that use DSP functions.
- Direct address** : A directly usable address in an instruction. It is usually the address on a page in the memory.
- DMA** : A direct memory access by a controller internal or external. DMA operations facilitate the peripherals and devices of the system to obtain access to the system memories directly, without the processor controlling the transfer of the bytes in a memory block.
- DRAM** : Dynamic RAM, which refreshes continuously by a device called DRAM refresh controller. Once programmed, it auto reads and writes the same set of bits repeatedly by scanning the DRAM memory cells.
- Echo cancellation** : A process of eliminating echoes.
- Echo** : A signal received after a delay and which superimposes over the original signal. For example, in a hall or at the hills we hear the original sound as well as the echoed sound. Similarly, there may be echo in electronic signal.

EDA	: A powerful tool for Electronic Design Automation.
EEMBC	: EDN Embedded Benchmark Consortium.
EEPROM	: A type of memory each byte of which is erasable many times and then programmable by the instructions of a program as well as by a device programmer.
EPROM	: A type of memory that is erasable many times by UV light exposure and programmable by a device programmer.
Erase time	: Time taken for device erasing.
Fixed point arithmetic	: Arithmetic using signed or unsigned integers employing processor registers or memory.
Flash	: A memory in which a set of sectors erase simultaneously.
Flash	: A type of memory in a sector of bytes that is erasable many times (maximum, ~10000) in a flash at the same instance in a single cycle. Each erased byte is then programmable by the write instruction of a program as well as by a device programmer.
Floating point arithmetic	: Arithmetic using processor registers or memory, where the decimal numbers and fractional numbers are stored in a standard floating-point representation.
High-level language support	: A supporting unit for given processor structure that facilitates program coding in C or other high level languages and enables their running like machine codes by an internal compilation.
Index register	: A register holding a memory address of a variable in an array, queue, table or list.
Instruction cache	: A cache to sequentially hold the instructions that have been prefetched for pipeline base parallel execution.
Instruction decoder	: The circuit to decode the opcode of the instruction and direct the control unit accordingly.
Instruction format	: Format of expressing an instruction.
Instruction queuing unit	: A unit to hold a queue of instructions and place these into the cache.
Instruction register	: A register to hold the current instruction for execution.
Instruction set	: A definite set of executable instructions in a processor.
Instruction set	: A unique processor-specific set of instructions.
Interface circuit	: A circuit consisting of the latches, decoders, multiplexers and demultiplexers.
Internal bus	: A set of paths that carry in parallel the signals between various internal structural units of a processor. Its size is 64-bit in a 64-bit processor.
Java Accelerator	: An accelerator that helps in the execution of Java codes faster than a JVM.
JVM	: Machine codes that use the compiled byte codes of a Java program and run the program on a given system.
MAC unit	: A unit used in DSP operations for fast calculation of $\Sigma[ax_n + (b_i y_j)]$ or similar expressions.
Master	: A processor, device or system which synchronously or asynchronously controls the output to several different processors, devices and systems called slaves.

	: The master can choose to send output to an addressed slave if a slave has a distinct address. The master can choose to receive an input from any slave selected by it at an instant.
McBSP	: A high-speed communicating multichannel Buffered Serial Port.
Memory address register	: A register that holds the address for a memory unit for placing it on the bus using bus interface unit.
Memory data register	: A register that holds the data for or from a memory unit.
Memory management unit	: A unit to manage the prefetch, paging and segmentation of memories.
Memory map	: A memory addresses allocation table such that the map reflects the available memory addresses for various uses of the processor. A memory map defines the addresses of the ROMs and RAMs of the systems.
Microarchitecture	: When a processor architecture refers specifically to the architectural instruction sets and programmers' models, the term microarchitecture refers specifically to the implementation of those architectures. A processor may have CISC architecture with an RISC microarchitecture implementation.
Noise elimination	: A process that eliminates unrelated randomly introduced signal components.
OMAP 5910 processor	: A TI processor of unique architecture in DSP chips of high performance with low power consumption.
On-chip parallel port	: The port on a chip which receives or sends 8 or 16 bits at an instance.
On-chip serial port	: The port on a chip which receives or sends a bit serially at an instance with a definite rate in kbps [baud rate in UART].
Opcode	: First byte of an instruction for the instruction decoder of the processor. It defines the operation or process to be performed on the operand(s).
Performance benchmarking	: Metrics for evaluating the performance of a system.
Pipelining	: There is pipelining also in the superscalar processor. It means that its ALU circuit divides into n subunits. If in its last part, the processing of a p-th instruction is taking place at an instant, then at the first part processing of (p + n)th instruction is taking place. There may be multiple pipelines in a processor to process in parallel.
Prefetch control unit	: A unit to fetch instructions in advance and data in advance from the memory units.
Program counter	: A processor register to hold the current instruction address to be executed after a fetch cycle on the buses.
Program flow instruction	: An instruction in which the program counter or instruction pointer changes in a way different from its normal changes during program execution.
PROM or OTP	: A type of memory which is programmable only once by a device programmer. OTP is a one-time programmable memory.
Pulse accumulator counter	: A counter that counts the input pulses during a select interval. When used as a timer with a repeatedly loadable value, it functions as a pulse width modulator.
Real time video processing	: Processing of video signals such that all or most incoming frames are processed in a time frame such that each processed frame maintains constant phase differences in the intervals between them.

- RISC with CISC functionality** : A processor with RISC implementation but user programs it similar to a CISC.
- RISC** : A Reduced Instruction Set Computer that has one feature that provides a small instruction set and permits limited addressing modes for the source and destination operands in an arithmetic or logic instruction. The hardware executes each instruction in a single cycle.
- Segment register** : A register to point to the start of a segment for a program code or data set or string or stack.
- Slave** : A processor or device or system, which receives the input from the master processor or device or system. This slave is the one having a distinct address and is chosen by the master.
- Special function register** : A register in 8051 for special functions of accumulator, data pointer, timer control, timer mode, serial buffer, serial control, power down control, ports, etc.
- Stack pointer** : A register that hold an address to define the available memory address to where the processor can push the registers and variables on a stack operation and from where they can be popped.
- Stack** : A block to memory that holds the pushed values for last-in first-out data transfer on popping back the values.
- Superscalar processor** : A processor with the capacity to fetch, decode and execute more than one instruction in parallel at an instant.
- Synchronous communication** : Data bytes or frames maintain uniform phase differences in serial communication.
- System register** : Processor register.
- Thumb[®] instruction set** : An instruction set in which each instruction is of 16 bits on a 32-bit processor. It gives reduced code density. It is a 16-bit instruction set which enables 32-bit performance at 8/16-bit system cost. They are used by ARM processors.
- Timing diagram** : A diagram that reflects the relative time intervals of the signals on the external buses with respect to the processor clock pulses.
- Video accelerator** : An accelerator for the video output.
- Watchdog Timer** : A timer which is set in advance and program codes are made such that its overflow indicates that a process is stuck somewhere and therefore the processor resets and restarts.



Review Questions

1. Explain 8051 architectural features. What are the devices internally present in the classic 8051. How do you interface a programmable peripheral interface in 8051?
2. Describe serial interface, timer/counters and interrupts in 8051.
3. Describe real-word interfacing. Explain interfacing to keyboard.
4. Compare memory-mapped IO and IO-mapped IOs.
5. What are the common structure units in most processors?
6. Compare Harvard and Princeton memory organizations.
7. What are the special structural units in processors for digital camera systems, real-time video processing systems, speech compression systems, voice compression systems and video games?

8. How do having separate caches for instruction, data and branch-transfer help?
9. What is the advantage of having multiway cache units so that only that a part of the cache unit is activated, which has the necessary data to execute a subset of instructions? List four exemplary processors with multiway caches.
10. When do you need MAC units at a processor in the system?
11. Explain three-stage pipeline, superscalar processing and branch- and data-dependency penalties.
12. What are the advantages in Harvard architecture? Why is the ease of accessing stack and data-table at program memory less in Harvard memory architecture compared to Princeton memory architecture?
13. Explain three performance metrics of a processor: MIPS, MFLOPS and Dhrystone per second.
14. Why should a program be divided into functions (routines or modules) and each placed in different memory blocks or segments?
15. How do the ARM7, ARM 9, ARM 11 and StrongArm differ? When will you prefer ARM7, when ARM 9 and when ARM11?
16. How does a memory map help in designing a locator program?
17. What do you mean by the terms: Quarter-CIF, EDORAM, RDRAM, peripheral transactions server, shadow segment, on-chip DMAC and time-division multiplexing.
18. How does a decoder help in memory and IO device interfacing? Draw four exemplary circuits.



Practice Exercises

19. Draw the memory organization in 8051.
20. How will you interface an 8051 to four servomotors in a robot using timer/counters and ports of 8051?
21. A two by three matrix multiplies by another three by two matrix. If data transfer from a register to another takes 2 ns, addition takes 20 ns and multiplication takes 50 ns, what will be the execution time? How will a MAC unit help. Assume that these times are same in a DSP with a MAC unit?
22. An array has 10 integers, each of 32 bits. Let an integer be equal to its index in the array multiplied by 1024. Let the base address in memory be 0x4800. How will the bits be stored for the 0th, 4th and 9th element in (a) big-endian mode (b) little-endian mode?
23. We can assume that the memory of an embedded system is also a device. List the reasons for it. [Hint: Use of pointers like access control registers and the concept of virtual file and RAM disk devices.]
24. Nowadays high-performance embedded systems use either an RISC processor or a processor with an RISC core with a code-optimized CISC instruction set. Why?
25. A circular queue has 100 characters at the memory addresses, each of 32 bits. What will be the total memory space required, including the space for both the queue pointers?
26. Estimate the memory requirement for a 500-image digital camera when the resolution is (a) 1024 x 768 pixels, (b) 640 x 480, (c) 320 x 240 and (d) 160 x 120 pixels and each image stores in compressed jpg format.
27. What are the special structural units in processors for digital camera, real-time video processing, speech compression and video game systems?

Devices and Communication Buses for Devices Network

3

R The following facts have been presented in the previous chapters:

- Embedded systems' hardware consist of processors, microcontrollers or DSPs and basic hardware units such as power supply, clock circuits, reset circuits, memory devices (ROM, flash and RAM) of different sizes and speed of access, and IO peripheral devices and ports for the UART, modem, transceiver, timer-counter, keypad, touch screen or LCD or LED display, DAC, ADC and pulse dialer
- Embedded system microprocessors interface with real world IO devices, such as DMA and bus controllers, peripherals, IO ports and keypad.
- The controllers, peripherals and ports have addresses using which the processor accesses bytes and words. An embedded system connects to devices such as the keypad, touch screen, multiline display unit, printer or modem or motors through ports.
- During a read or write operation, the processor accesses that address in a memory-mapped IO, as if it were accessing a memory address. A decoder takes the system memory or IO address bus signals as the input and generates a port or device select signal, CS and selects the port or device.

L
E
A
R
N
I
N
G
O
B
J
E
C
T
I
V
E
S

We can't think of a computer without IO devices for the video output, mouse, keyboard input, CD and magnetic storage. We can't think of a mobile smart phone without the devices for LCD or touchscreen, IO port interfaces, keypad, timers, dialer, speaker, radio interface and flash memory storage. Similarly, we can't think of an embedded system without IO devices, timers and other devices. In fact, the devices play the most significant role in any embedded system. A device connects and accesses to and from the system-processor and memory either internally or through an internal controller or through a port, with each port having an assigned port addresses similar to a memory addresses.

Distributed devices are networked using sophisticated IO buses. For example, take the case of an automobile: all embedded devices in automobile have a microcontroller, and network through IO bus. The devices in an automobile are distributed at different locations. These are networked using a bus called Control Area Network (CAN) bus. Similarly, a camera interfaces to a computer and printer through a USB bus or Bluetooth device.

Advanced networking devices such as transceivers and encrypting and decrypting devices operate at high speeds.

A hardware engineer designing an embedded system must, therefore, clearly understand the features of interface circuits and their speed of operations and the buses that network the devices.

We will learn the following topics in this chapter:

1. Serial and parallel input, output and IO ports
2. Synchronous serial-communication devices and examples of High-Level Data Link Control
3. Asynchronous serial-communication devices and their examples, RS232C and UART
4. Parallel ports and parallel communication devices
5. Wireless devices
6. Sophisticated interfacing features in the systems for fast IOs, fast transceivers, and real time voice and video IOs
7. Timing and counting devices, and the concept of real time clock, software timers and watchdog timers
8. Inter Integrated Circuit (I^2C) communication bus between multiple distributed ICs and the CAN bus as the control network between the distributed devices in the automobiles
9. Universal Serial Bus (USB) for fast serial transmission and reception between the host embedded system and distributed serial devices like the keyboard, printer, scanner and ISDN system
10. IBM Standard Architecture (ISA) and Peripheral Component Interconnect (PCI)/PCI-X (PCI Extended) buses between a host computer or system and PC-based devices, systems or cards; for example, PCI bus between the PC and Network Interface Card (NIC)

- 11. Internet-enabled embedded devices and their network protocols
- 12. Wireless protocols for mobile and wireless networks

3.1 IO TYPES AND EXAMPLES

A serial port is a port for serial communication. Serial communication means that over a given line or channel one bit can communicate and the bits transmit at periodic intervals generated by a clock. A serial port communication is over short or long distances.

A parallel port is a port for parallel communication. Parallel communication means that multiple bits can communicate over a set of parallel lines at any given instance. A parallel port communicates within the same board, between ICs or wires over very short distances of at most less than a meter.

A serial or parallel port can provide certain special features and sophistication (Section 3.4) by using a processing element.

Ports can interconnect by wireless. Wireless or mobile communication is serial communication but without wires, can be over a short-range personal area network as well as long-range wireless network, and transmission takes place by using carrier frequencies. The carrier modulates the serial bits before transmission in air [Sections 3.5 and 3.13]. A receiver demodulates and retrieve the serial bits back.

Serial and parallel ports of IO devices can be classified into following IO types: (i) Synchronous serial input (ii) Synchronous serial output (iii) Asynchronous serial UART input (iv) Asynchronous serial UART output (v) Parallel port one bit input (vi) Parallel one bit output (vii) Parallel port input (viii) Parallel port output. Some devices function both as input and as output; for example, a modem.

3.1.1 Synchronous Serial Input

The part 1 in Figure 3.1(a) shows a synchronous input serial port. Each bit in each byte and each received byte is in synchronization. Synchronization means separation by a constant interval or phase difference [part 2 in Figure 3.1(a)]. If clock period equals T , then each byte at the port is received at input in period $8T$. The bytes are received at constant rates. Each byte at the input port separates by $8T$ and data transfer rate for the serial line bits is $1/T$ bps [1 bps = 1-bit per second]. The sender, along with the serial bits, also sends the clock pulses SCLK (serial clock) to the receiver port pin. The port synchronizes the serial data-input bits with clock bits.

The serial data input and clock pulse-input are on same input line when the clock pulses either encode or modulate serial data input bits suitably. The receiver detects clock pulses and receives data bits after decoding or demodulating.

When a separate SCLK input is sent, the receiver detects at the middle, positive or negative edge of the clock pulses that indicate whether data-input is 1 or 0 and saves the bits in 8-bit shift register. The processing element at the port (peripheral) saves the byte at a port register from where the microprocessor reads the byte.

Synchronous serial input is also called master output slave input (MOSI) when the SCLK is sent from the sender to the receiver and slave is forced to synchronize sent inputs from the master as per the master clock inputs. Synchronous serial input is also called master input slave output (MISO) when the SCLK is sent to the sender (slave) from the receiver (master) and the slave is forced to synchronize sending the inputs to master as per the master clock's outputs.

Synchronous serial input is used for interprocessor transfers, audio inputs and streaming data inputs.

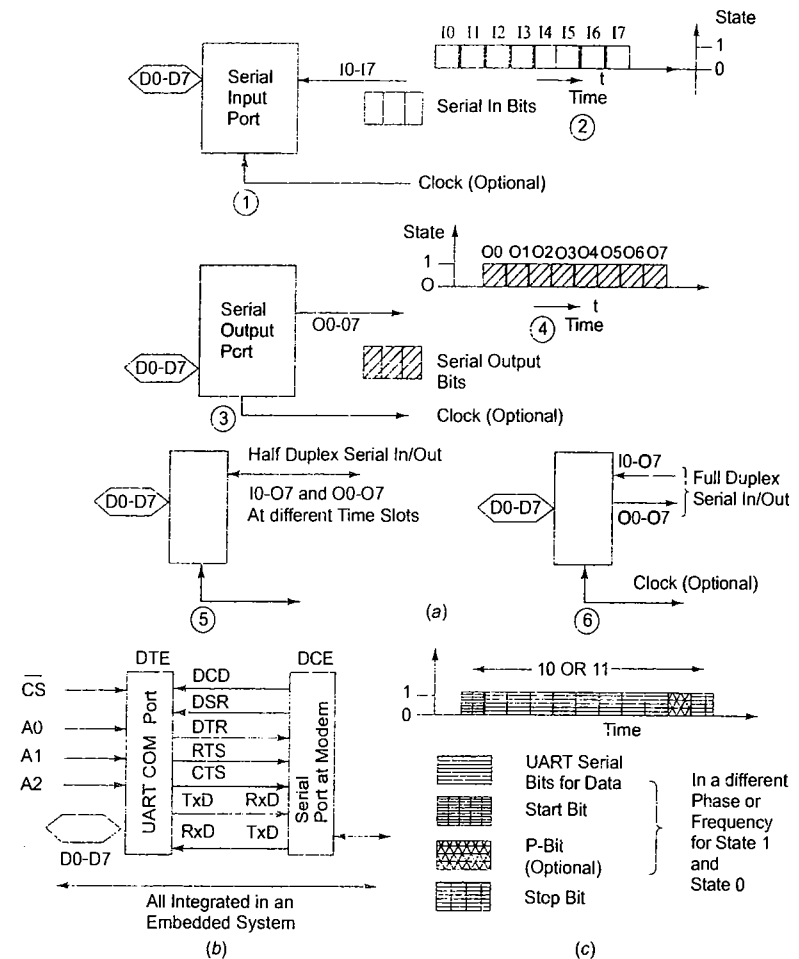


Fig. 3.1 (a) Input serial port, Output Serial port, Bi-directional half-duplex serial port, and Bi-directional full-duplex serial port (b) Handshaking signals at COM port in computer and (c) a UART serial port bits

3.1.2 Synchronous Serial Output

The part 3 in Figure 3.1(a) shows a synchronous output serial port. Each bit in each byte is in synchronization with a clock. The bytes are sent at constant rates [part 4 in Figure 3.1(a)]. If the clock period equals T , then the data transfer rate is $1/T$ bps. The sender sends either the clock pulses at SCLK pin or the serial data output and clock pulse-input through same output line when the clock pulses either suitably modulate or encode the serial output bits.

The processing element at the port (peripheral) sends the byte through a shift register at the port to which the microprocessor writes the byte.

Synchronous serial output is used for interprocessor transfers, audio outputs and streaming data outputs.

3.1.3 Synchronous Serial Input–Output

The part 5 in Figure 3.1(a) shows a synchronous serial input–output port. Each bit in each byte synchronizes with the clock input and output. The bytes are sent or received at constant rates as shown in parts (2) and (4) in Figure 3.1(a)]. The IOs are on same IO line when the clock pulses suitably modulate or encode the serial input and output, respectively. If the clock period equals T , then the data transfer rate is $1/T$ bps. The processing element at the port (peripheral) sends and receives the byte at a port register to or from which the microprocessor writes or reads the byte.

Synchronous serial input/outputs are also called master input slave output (MISO) and master output slave input (MOSI), respectively.

They are used for interprocessor transfers and streaming data. The bits are read from or written on magnetic media such as a hard disk or on optical media such as a CD by using devices with serial synchronous IO ports.

The part 6 in Figure 3.1(a) shows the IO synchronous port when input and output lines are separate.

3.1.4 Asynchronous Serial input

Figure 3.1(b) shows the asynchronous input serial port line, denoted by RxD (receive data). Each RxD bit is received in each byte at fixed intervals but each received byte is not in synchronization. The bytes can separate by variable intervals or phase differences. Figure 3.1(c), on the right side, shows the starting point of receiving the bits for each byte, indicated by a line transition from 1 to 0 for a period T . When a sender shifts after every clock period T , then a byte at the port is received at input in period $10T$ or $11T$. The time of $2T$ is due to use of additional bits at the start and end of each byte. An addition time of $1T$ is taken when a P-bit is sent before the stop bit.

The bit transfer rate (for the serial line bits) is $(1/T)$ baud per second but different bytes may be received at varying intervals. The word 'Baud' is taken from a German word for raindrop. Bytes pour from the sender like raindrops at irregular intervals. The sender does not send the clock pulses along with the bits.

The receiver detects n bits at the intervals of T from the middle of the first indicating bit. $n = 0, 1, \dots, 10$ or 11 , finds out whether the data-input is 1 or 0 and saves the bits in an 8-bit shift register. The processing element at the port (peripheral) saves the byte at a port register, from where the microprocessor reads the byte.

Asynchronous serial input is also called UART input if the serial input is according to the UART protocol (Section 3.2.3). Asynchronous serial input is used for keypad and modem inputs.

3.1.5 Asynchronous Serial Output

Figure 3.1(b) shows the asynchronous output serial port line, denoted by TxD (transmit data). Each bit in each byte is sent at fixed intervals but each output byte is not in synchronization (it is separated by a variable interval or phase difference). The Figure 3.1(c) shows the starting point of sending the bits for each byte, which is indicated by a line transition from 1 to 0 for a period T . The sender port of TxD does not send clock pulses along with the bits.

The sender transmits bytes at the minimum intervals of nT . Bits start from the middle of the start indicating bit, where $n = 0, 1, \dots, 10$ or 11 and sends the bits through a 10- or 11-bit shift register [Figure 3.1(c)]. The processing element at the port (peripheral) sends the byte at a port register to where the microprocessor writes the byte.

Asynchronous serial output is also called UART output if the serial output is according to a UART protocol (Section 3.2.3). Asynchronous serial output is used for modem and printer inputs.

3.1.6 Parallel Port

A parallel port can have one or multibit input or output and can be bi-directional IO.

- (i) One bit input, output and IO
- (ii) Eight or more bit input, output and IO

Section 3.3 will describe parallel device ports in detail.

3.1.7 Half Duplex and Full Duplex

The part 5 in Figure 3.1(a) on the left side shows the IO serial port (bi-directional half-duplex serial port). Half duplex means that at any point communication can only be one way (input or output) on a bi-directional line. An example of half-duplex mode is telephone communication. On one telephone line, we can talk only in the half-duplex mode. The part 6 in Figure 3.1(a) shows the separate input and output serial port lines. Full duplex means that the communication can be both ways simultaneously. An example of the full duplex asynchronous mode of communication is communication between the modem and computer through the TxD and RxD lines [Figure 3.1(b)].

There are two types of communication ports for IOs: serial and parallel. Serial line port communication is synchronous when a clock of the master device controls the synchronization of the bits on the line. Serial line port communication is asynchronous when clocks of the sender and receiver are independent and bytes are received, not necessarily at constant phase differences. Serial communication can be full duplex, which means simultaneously communication both ways, or half duplex, which means one way communication.

3.1.8 Examples of Serial and Parallel Port IOs

Table 3.1 gives a classification of IO devices into various types. It also gives examples of each type.

Table 3.1 Examples of various types of IO devices

<i>IO Device Type</i>	<i>Examples</i>
<i>Serial synchronous input</i>	Inter-processor data transfer, reading from CD or hard disk, audio input, video input, dial tone, network input, transceiver input, scanner input, remote controller input, serial IO bus input, reading from flash memory using SDIO (Secure Data Association IO) card
<i>Serial synchronous output</i>	Inter-processor data transfer, multiprocessor communication, writing to CD or hard disk, audio output, video output, dialer output, network device output, remote TV Control, transceiver output, and serial IO bus output, writing to flash memory using SDIO card
<i>Serial asynchronous input</i>	Keypad controller serial data-in, mice, keyboard controller data in, modem input, character inputs on serial line [also called UART (universal receiver and transmitter) input when according to UART mode]

(Contd)

IO Device Type	Examples
Serial asynchronous output	Output from modem, output for printer, the output on a serial line [also called UART output when according to UART mode]
Parallel port single bit input	(i) Completion of a revolution of a wheel. (ii) achieving preset pressure in a boiler, (iii) exceeding the upper limit of the permitted weight over the pan of an electronic balance, (iv) presence of a magnetic piece in the vicinity of or within reach of a robot arm to its end point and (v) filling a liquid up to a fixed level
Parallel port single bit output	(i) PWM output for a DAC, which controls liquid level, temperature, pressure, speed or angular position of a rotating shaft or a linear displacement of an object or a d.c. motor control (ii) pulses to an external circuit
Parallel port input	(i) ADC input from liquid level measuring sensor or temperature sensor or pressure sensor or speed sensor or d.c. motor rpm sensor (ii) Encoder inputs for bits for angular position of a rotating shaft or a linear displacement of an object
Parallel port output	(i) LCD controller for multiline LCD display matrix unit in a cellular phone to display on screen the phone number, time, messages, character outputs or pictogram bit-images or e-mail or web page (ii) print controller (iii) stepper motor coil driving output bits

3.2 SERIAL COMMUNICATION DEVICES

3.2.1 Synchronous, Iso-synchronous and Asynchronous Communications from Serial Devices

Synchronous Communication When a byte (character) or frame (a collection of bytes) of data is received or transmitted at constant time intervals with uniform phase differences, the communication is called *synchronous*. Bits of a data frame are sent in a fixed maximum time interval. *Iso-synchronous* is a special case when the maximum time interval can be varied.

An example of synchronous serial communication is frames sent over a LAN. Frames of data communicate, with the time interval between each frame remaining constant. Another example is the inter-processor communication in a multiprocessor system. Table 3.2 gives a synchronous device port bits.

Figure 3.1(a) part 2 showed the serial IO bit format and serial line states as a function of time. Two characteristics of synchronous communication are as follows:

1. Bytes (or frames) maintain a constant phase difference. It means they are synchronous, that is, in synchronization. There is no permission for sending either the bytes or the frames at random time intervals; this mode therefore does not provide for handshaking *during* the communication interval. [Handshaking means that the source and destination first exchange the signals between them before they communicate the data bits.] The master is the one whose clock pulses guide the transmission and slave is the one which synchronizes the bits as per the master clock.
2. A clock ticking at a certain rate must always be there to serially transmit the bits of all the bytes (or frames). The clock is not always implicit to the synchronous data receiver. The transmitter generally transmits the clock rate information in the synchronous communication of the data.

Table 3.2 Synchronous device port bits

S.No.	Bits at Port	Compulsory or Optional	Explanation
1.	Sync code bits or bi-sync code bits or frame start and end signaling bits	Optional	A few bits (each separated by interval ΔT) as Sync code for frame synchronization or signaling precedes the data bits ¹ . There may be inversion of code bits after each frame. Flag bits at start and end are also used in certain protocols
2.	Data bits	Compulsory	m frame bits or 8 bits transmit such that each bit is at the line for time ΔT or each frame is at the line for time $m.\Delta T$ ²
3.	Clock bits	Mostly not optional	Either on a separate clock line or on a single line such that the clock information is also embedded with the data bits by an appropriate encoding or modulation

¹Reciprocal of ΔT is the transfer rate in bit per second (bps).

² m may be a large number. It depends on the protocol.

Figure 3.2 gives ten methods by which synchronous signals, with the clocking information, are sent. (i) There are two separate lines for the data bits and clock. The parallel-in serial-out (PISO) and serial-in parallel-out (SIPO) are used for transmitting and receiving the signals for data, respectively. (ii) There is a common line and the clock information is encoded by modulating the clock with the stream of bits. (iii) There are preceding and succeeding additional synchronizing and signaling bits. There are five common methods of encoding the clock information into a serial stream of the bits: (a) Frequency Modulation (FM) (b) Mid Frequency Modulation (MFM) (c) Manchester coding (d) Quadrature amplitude modulation (QAM) (e) Bi-phase coding. The synchronous receiver separates serial bits of the message as well as synchronizing clock.

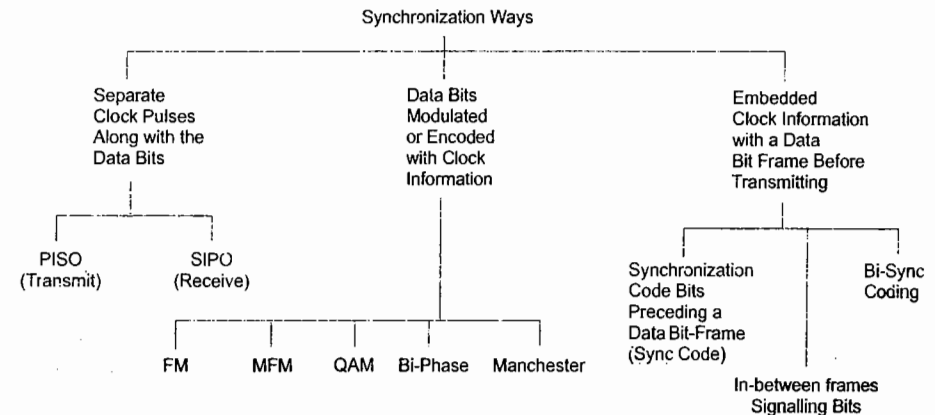


Fig. 3.2 Ten ways by which the synchronous signals with the clocking information transmit from a master device to slave device

Asynchronous Communication When a byte (characters) or frame (a collection of bytes) of data is received or transmitted at variable time intervals, communication is called *asynchronous*. Voice data on the line is sent in asynchronous mode. Over a telephone line the communication is asynchronous. Another example is keypad communication.

An example of mode of asynchronous communication is RS232C communication between the UART devices (Section 3.2.2).

UART communication (Section 3.2.3) for asynchronous data is used for the transfer of information between the keypad or keyboard and computer.

Two characteristics of asynchronous communication are as follows:

1. Bytes (or frames) need not maintain a constant phase difference and are asynchronous, that is, not in synchronization. Bytes or frames can be sent at variable time intervals. This mode therefore facilitates in-between handshaking between the serial transmitter port and serial receiver port.
2. Though the clock must tick at a certain rate to transmit bits of a single byte (or frame) serially, it is *always implicit* to the asynchronous data receiver. The transmitter *does not* transmit (neither separately nor by encoding using modulation) along with serial stream of bits any clock rate information in asynchronous communication. The receiver clock is thus not able to maintain identical frequency and constant phase difference with the transmitter clock.

When a device sends data using a serial communication frame, it may not be as simple as shown in Figures 3.1(a) and (b) or as given in Table 3.2. It can be complex and has to be as per the protocol, which is followed by transmitting and receiving devices during communication between them.

Example 3.1

An IBM personal computer has two COM ports (communication ports), COM1 and COM2. These have 8 bytes at IO addresses 0x3F8 and 0x2F8.

Figure 3.1(b) showed COM port handshaking signals besides TxD and RxD. When a modem connects, it detects a carrier signal on the telephone line. A modem sends *data carrier detect* DCD signal at time t_0 . A modem then communicates *data set ready* (DSR) signal at time t_1 when it receives the bytes on the line. The receiving end responds at time t_2 by data terminal ready (DTR) signal. After DTR, *request to send* (RTS) signal is sent at time t_3 and the receiving end responds by *clear to send* (CTS) signal at time t_4 . After the response CTS, the data bits are transmitted by modem from t_5 to the receiver terminal at successive intervals [Figure 3.1(c)]. Between two sets of bytes sent in asynchronous mode, the handshaking signals RTS and CTS can again be exchanged. This explains why the bytes do not remain synchronized during asynchronous transmission.

A communication system may use the following protocols for synchronous or asynchronous transmission from a device port: RS232C, UART, HDLC, X.25, Frame Relay, ATM, DSL and ADSL. These are protocols for networking the physical devices in telecommunication and computer networks. Ethernet and token ring are protocols used in LAN networks. There are a number of protocols for serial communication. RS232C, UART and HDLC are described in Sections 3.2.2 to 3.2.4.

The protocols in embedded network devices such as bridges, routers, embedded Internet appliances use bridging, routing, application and web protocols. Internet enabled embedded systems use *application* protocols — HTTP (hyper text transfer protocol), HTTPS (hyper text transfer protocol Secure Socket Layer), SMTP (Simple Mail Transfer Protocol), POP3 (Post office Protocol version 3), ESMTP (Extended SMTP), TELNET (Tele network), FTP (file transfer protocol), DNS (domain network server), IMAP 4 (Internet Message Exchange Application Protocol) and Bootp (Bootstrap protocol) and others (Section 3.11).

Embedded wireless appliances use wireless protocols— IrDA, Bluetooth, 802.11 and others (Section 3.13).

Synchronous, iso-synchronous and asynchronous are three ways of communication. Clock information is transmitted explicitly or implicitly in synchronous communication. The receiver clock continuously maintains constant phase difference with the transmitter clock. HDLC is a data link protocol for computer networks and telecommunication devices. RS232C and UART are asynchronous mode communication standard.

3.2.2 RS232C/RS485 Communication

(i) RS232C RS232C communication is between DTE (computer) COM (communication) port and DCE (modem) port. DTE stands for 'Data Terminal Equipment'. DCE stands for 'Data Communication Equipment'. RS232C is an interfacing signal standard between DCE and DTE.

Figure 3.1(b) showed the interfacing (handshaking) signals on a RS232C port. The receive data and transmit data signals from RS232C port are RxD and TxD, respectively. RS232C port serial RxD and TxD bits are asynchronous and follow UART protocol (Section 3.2.3). Receiver end voltage level from -3 to -25 V denotes logic 1 and voltage level from $+3$ to $+25$ V denotes logic 0. Transmitter end voltage level from -5 to -15 V denotes logic 1 and voltage level from $+5$ to $+15$ V denotes logic 0.

Example 3.2

The IBM personal computer two COM ports (communication ports) called COM2 and COM1, have IO addresses 0x2F8-0x2FF and 0x3F8-0x3FF, respectively. The COM port is RS232C port. It has TxD and RxD serial output and input. It has handshaking signals \overline{DCD} , \overline{DSR} , \overline{DTR} , \overline{RTS} and \overline{CTS} .

RS232C port in a computer is used upto 9600 baud/s asynchronous serial transmission rate with UART mode communication. Generally baud rates are set at 300, 600, 1200, 4800 and 9600. When transmitting upto 0.25 m or 1 m on cable (untwisted) the maximum baud rate can be 115.2 k or 38.4 k baud/s, respectively.

RS232C port is used for keyboard serial communication at 1200 baud/s asynchronous serial transmission rate with UART mode communication at IBM PC COM port. The signals used are \overline{RTS} , \overline{CTS} , TxD and RxD for keypad communication. A mice port also is RS232C COM port in the computer. (New mice nowadays use USB in place of COM port).

Example 3.3

A mobile smart phone has a Bluetooth device for personal area wireless network. A Bluetooth device is capable of emulating DCE serial port, which can now communicate in UART mode.

A computer on the other hand has a serial port called COM port (Example 3.1). The mobile device is placed on a cradle. The mobile device port data-pins connect the cradle pins. The cradle connects to the computer or laptop COM port. The mobile and computer serial ports then communicate. The data (for example, pictures or address book data) between them synchronize between COM and emulated serial at Bluetooth device.

RS232C standard provides for UART serial port asynchronous mode communication. A different set of voltage levels are prescribed for the 0s and 1s in RS232C standard.

(ii) **RS485** RS485, now called EIA-485 is a protocol for physical layer in case of two wire full or half duplex serial connection between multiple points. Transmission is at 35 Mbps upto 10 meter and 100 Kbps up to 1.2 km. Electrical signals are between + 12 V and -7 V. Logic 1 is +ve and 0 is reverse polarity. Difference in potential defines logic 1 and 0. A converter is used to convert RS232C bits to RS485 and another for vice versa.

3.2.3 UART

Figure 3.1(b) showed handshaking signals of RS232C port and UART serial bits in the output to a serial line device. The UART mode is as follows:

1. A line is in non-return to zero (NRZ) state. It means that in the idle state the logic state is 1 at serial line.
2. The start of serial bits is signaled by 1 → 0 transition (negative edge) on the line for a period equal to reciprocal of baud rate. The baud rate is preset at both receiver and transmitter. The receiver detects the start bit at middle of the interval, logic 0 state of the transmitter start bit.
3. UART bits, when sending a byte, consist of start bit, 8 data bits (for example, for an ASCII character or for a command word), option programmable bit (P-bit) and stop bit, each during the interval δT . When sending or receiving a byte, the logic states during interval $10 \delta T$ or $11 \delta T$ are as shown in Figure 3.1(c) as a function of time t . A bit period, δT is equal to the reciprocal of baud rate, the rate at which the bits from UART transmitter are sent. One extra bit before the stop bit is programmable bit P and is called TB8 at the transmitter and RB8 at the receiver.
4. The data bits in certain specific cases can be 5 or 6 or 7 instead of 8.
5. The stop bit can be for a minimum interval of $1.5 \delta T$ or $2 \delta T$ instead of δT in certain specific cases.
6. Optional programmable bit (P-bit) can be used for parity detection or can be used to specify the purpose of the serial data bits that are before the P-bit. For example, P can specify bits as the bits of a control or command word when $P = 1$ and data bits when $P = 0$. Bit P can specify the address of receiver when $P = 1$ and data when $P = 0$ so that only the addressed receiver wakes up and receives the data in the subsequent data transfers. When P is used as address/data specification, it provides a means to interface a number of UART devices through a common set of Tx/D and Rx/D lines and form a UART bus.

UART 16550 includes a 16-byte FIFO buffer and is nowadays used more commonly as compared to the original IBM PC COM port, which had an 8-bit register at UART port and was based on 8250 and did not include the FIFO buffer.

UART serial port communication is usually either in 10 bits or in 11 bits format: one start bit, 8 data bits, one optional bit and one stop bit. UART communication can be full duplex, which is simultaneously both ways, or half duplex, which is one way. It is an important communication mode.

3.2.4 HDLC Protocol

When data are communicated using the physical devices on a network, synchronous serial communication may be used. HDLC (High Level Data Link Control) is an International Standard protocol for a data link network. It is used for linking data from point to point and between multiple points. It is used in telecommunication and computer networks. It is a bit-oriented protocol. The total number of bits is not necessarily an integer multiple of a byte or a 32-bit integer. Communication is full duplex.

Table 3.3 gives the synchronous network device port bits in an HDLC protocol. The reader may refer to a standard textbook, for example, "Data Communications, Computer Networks and Open Systems" by Fred Halsall from Pearson Education (1996) for details of HDLC and its field bits.

Table 3.3 Format of bits in synchronous HDLC protocol-based network device

S.No.	Bits ¹ at Port	Present Compulsorily or Optionally	Explanation
1	Frame start and end signaling flag bits	Compulsory	Flag bits at start as well as at end are (01111110)
2	Address bits for destination	Compulsory	8-bits in standard format and 16-bits in extended format
3a	Control bits Case 1: information frame	Compulsory as per case 1 or 2 or 3	First bit 0, next 3 bits N(S), next bit P/F ² and last 3 bits N(R) in standard format N(R) ³ and N(S) = 7 bits each in extended format
3b	Control bits case 2: supervisory frame	—	First two bits (10), next 2 bits RR ³ or RNR or REJ or SREJ, next bit P/F and last 3 bits N(R) in standard format. N(R) ⁴ and N(S) ⁴ = 7 bits each in extended format
3c	Control bits Case 3: un-numbered frame	—	First two bits (11), next 2 bits M ⁵ , next bit P/F and last 3-bit remaining bits for M. [8 bits are immaterial after M bits in extended format]
4	Data bits	Compulsory	m frame bits transmit such that each bit is at the line for time ΔT or each frame is at the line for time $m \cdot \Delta T$ and also there is bit stuffing. ⁶
5	FCS (Frame check sequence) bits	Compulsory	16-bits in standard format and 32 in extended format
6	Frame end flag bits	Compulsory	Flag bits at end are also (01111110)

¹ Bits are given in order of their transmission or reception.

² P/F = 1 and P means when a primary station (Command device) is polling the secondary station (receiving device). P/F = 1 and F means when receiving device has no data to transmit. Usually it is done in last frame.

³ RR, RNR, REJ and SREJ are messages to convey 'Receiver ready,' 'Receiver not ready,' 'Reject,' and 'Selective reject'. REJ or SREJ is a negative acknowledgement (NACK). NACK is sent only when the frame is rejected. [A child cries only when milk is not given on need, else it remains silent.] 'Reject' means that the receiver received a frame out-of-sequence; it is rejected and a repeat transmission of all the frames from the point of frame rejection is requested using REJ. 'Selective reject' means that a frame is received out-of-sequence; it is to be rejected and a selective repeat transmission is requested for this frame using SREJ.

⁴ N(R) and N(S) means received (earlier) and sending (now) frame sequence numbers. These are modulo 8 or 128 in standard or extended format frame, respectively.

⁵ M five bits are for a command (or response) from a transmitter. Examples of a command are *reset*, *disconnect* or *set a defined mode type*; examples of a response are a message from the receiver for a disconnect mode accepted, frame rejected, command rejected, and for an unnumbered acknowledgement.

⁶ When five 1s transmit for the data, one 0 is stuffed additionally. This prevents misinterpretation by receiver the data bits as flag bits (01111110).

3.2.5 Serial Data Communication using the SPI, SCI and SI Ports

Microcontrollers have internal devices for SPI or SCI or SI as explained below. Each device has separate registers for control, status, serially received data bits and transmitting serial bits. Each device is programmable as described below. The device can be used in programmed IO modes or in interrupt driven reception and transmission.

Synchronous Peripheral Interface (SPI) Port Figure 3.3(a) shows an SPI port signals. Figure 3.3(b) shows SPI port in 68HC11 and 68HC12 microcontroller. It has full-duplex feature for synchronous communication. There are signals SCLK for serial clock, MOSI and MISO output from and input to master.

Section 3.1]. Figure 3.3(b) shows programmable features and DDR feature of Port D. An SPI feature is programmable rates for clock bits, and therefore for the serial out of the data bits down to the interval of 1.5 μ s for an 8 MHz crystal at 68HC11.

SPI is also programmable for defining the occurrence of negative and positive edges within an interval of bits at serial data *out* or *in*. It is also programmable in the open-drain or totem pole output from a master to a slave and for device selection as master or slave. This can be done by a signal to hardware input SS (slave select when 0) pin. In the hardware the slave select pin connects to '1' at the *master* SPI device and to '0' at the *slave*. Defining SPI as slave or master can also be done by software. Programming a bit at the device control register does this.

68HC12 provides SPI communication device operations at 4 Mbps. SPI device operates up to 2 Mbps in 68HC11.

Serial Connect Interface (SCI) Port Figure 3.3(c) shows an SCI port programmable features and DDR port bits in 68HC11/12. SCI is a UART asynchronous mode port. Communication is in full-duplex mode for the SCI transmission and receiver. SCI baud rates are fixed as prescaling bits. Rate not programmable separately for individual serial *in* and *out* lines. A baud rate can be selected among 32 possible ones by the three-rate bits and two prescaling bits. The SCI receiver has a *wake up* feature and is programmable by RWU (Receiver wakeup (available) bit). It is enabled if RWU (1st bit of SCC2, Serial Communication Control Register 2) is set, and is disabled if RWU is reset. If RWU is set, then the receiver of a slave is not interrupted by the succeeding bytes. SCI has two control register bits, TB8 and RB8. RWU feature helps in inter-processor communication, and SCI is defined for transmission and for reception using the SCC2 bits. UART communication, when programmed by control bits, is in 11-bit format. A number of processors can communicate on the bus in UART mode by RWU, when RB8 and TB8 bits are set.

There are separate hardware devices at 68HC11 for synchronous and asynchronous communications. These are the SPI and SCI, respectively. 68HC12 provides two SCI communication devices that can operate at two different clock rates. Standard baud rates can be set up to 38.4 kbps. There is only one SCI and standard baud rates in 68HC11 can be set up to 9.6 kbps only.

Serial Interface (SI) Port Figure 3.3(d) shows an SI port. SI is a UART mode asynchronous port interface. It also functions as USRT (universal synchronous receiver and transmitter). SI is therefore a synchronous–asynchronous serial communication port called USART (universal synchronous–asynchronous receiver and transmitter) port. It is an internal serial IO device in 8051. There is an on-chip common hardware device called SI in Intel 80196. Its features are as follows: programmable-rates register after loading the 14 bits at BAUD_RATE register twice. SI operates in one of the following ways:

- (i) Half-duplex synchronous mode of operation, called mode 0. When a 12 MHz crystal is at 8051 and is attached to the processor, the clock bits are at the intervals of 1 μ s.
- (ii) Full-duplex asynchronous serial communication, called mode 1 or 2 or 3. Using a timer, the baud rate varies according to the programmed timer bits in modes 1 and 3. Using SMOD bit at SFR called PCON, when mode 2 is used, the baud rate is programmable at two rates only. It is 1/64 or 1/32 of oscillator frequency at 8051. TB8 and RB8, when using 11-bit format, provide the 10th bit for error-detection or for indicating whether the sent data byte is a command or data for the receiving SI device.

Most microcontrollers have internal serial communication SPI and SCI or SI-like devices for serial communication. The IBM personal computer has two UART chips for the two COM ports. Table 3.4 gives the features of internal serial ports in select microcontrollers.

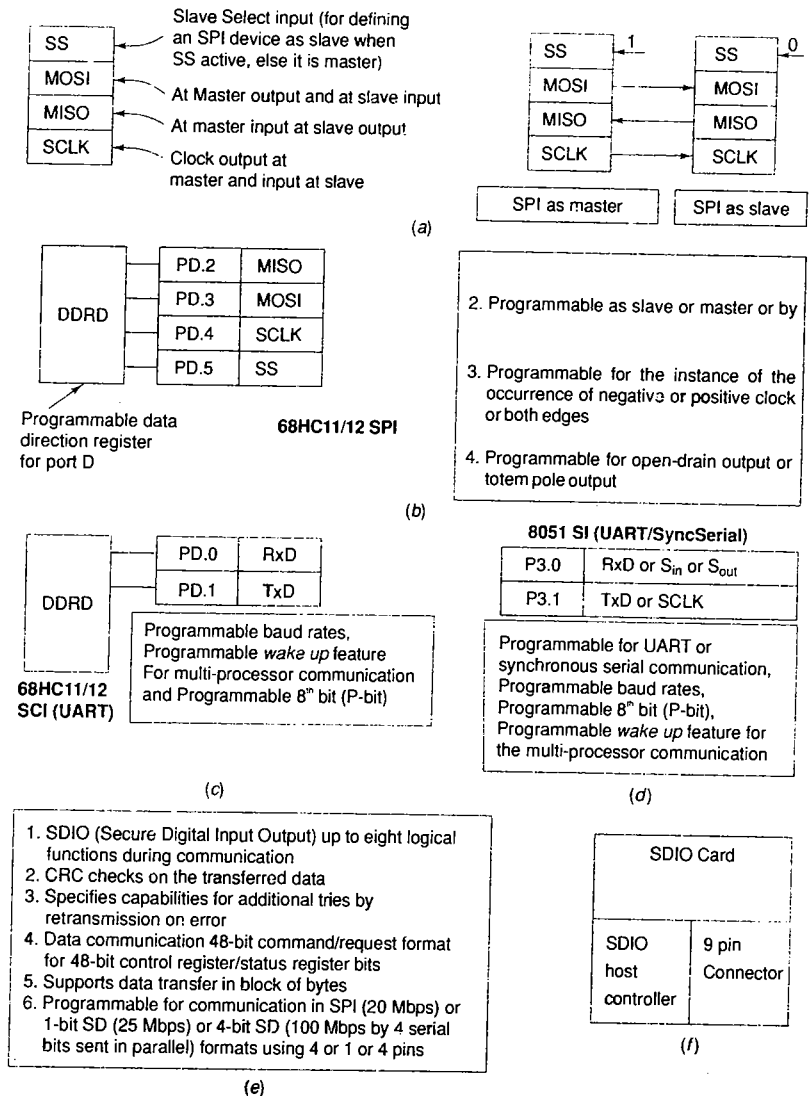


Fig. 3.3 (a) SPI port signals (b) SPI port programmable features and DDR at port D in 68HC11/12 (c) SCI port in 68HC11/12 (d) SI port in 8051 (e) SDIO communication features (f) SDIO card structure

Table 3.4 Processor with internal serial ports in microcontrollers

Features	Intel 8051 and Intel 8751	Motorola M68MC11E2	Intel 80196
Synchronous serial port (half or full duplex)	Half	Full	Half
Asynchronous UART port (half or full duplex)	Full	Full	Full
Programmability for 10 as well as 11 bits per byte from UART	Yes	Yes	Yes
Separate un-multiplexed port pins for synchronous and UART serial ports	No	Yes (Separate 4 Pins)	No
Synchronous serial port as a master or slave definition by software or hardware	Software	Hardware and software	Software
UART serial port programmability as a transmitter or receiver and for additional bit for parity or RWU or control or other purpose.	Yes	Yes	Yes
Synchronous serial port registers	SCON, SBUF and TL-TH 0-1	SPCR, SPSR and SPDR	SPCON, SPSTAT, BAUD_RATE and SBUF
UART serial port registers	SCON, SBUF and TL-TH 0-1 (Time 2 in 8052)	BAUD, SCC1, SCC2, SCSR, SCIRDR and SCITDR	SPCON, SPSTAT, BAUD_RATE and SBUF
Uses internal timer or uses separate programmable BAUD rate generator.	Timer	Separate	Separate as well as the Timer

Microcontrollers have internal devices of three types SPI, SCI and SI. SPI is synchronous master slave mode serial full duplex communication. SCI is UART asynchronous transmitter-receiver mode serial full duplex communication. SI is synchronous half duplex and asynchronous full duplex UART serial communication.

3.2.6 Secure Digital Input Output (SDIO)

Secure Digital (SD) Association created a new flash memory card format, called SD format. It is an association of over 700 companies started from 3 companies in 1999. This SDIO card for SD format IOs [Figure 3.3(e)] has become a popular feature in handheld mobile devices, PDAs, digital cameras and handheld embedded systems. SD card size is just $0.14 \times 2.4 \times 3.2$ cm. SD card [Figure 3.3(f)] is also allowed to stick out of the handheld device open slot, which can be at the top in order to facilitate insertion of the SD card.

SDIO is an SD card with programmable IO functionalities such that it (a) can be used upto eight logical functions, (b) can provide additional memory storage in SD format, and (c) can provide IOs using protocols in systems such as IrDA adapter, UART 16550, Ethernet adapter, GPS, WiFi, Bluetooth, WLAN, digital camera, barcode or RFID code reader.

Figure 3.3(e) shows an SDIO communication device port features. It supports SPI (Section 3.2.5), 4-bit and 1-bit SD formats. Both SPI and SD formats specify that there should be interrupt handling of the IOs and also the CRC checks on transferred data, and specifies capabilities for more tries on error. SDIO card [Figure 3.3(f)] has 9 pins. Six pins are for communication using SPI or SD. A processing element function is used as SDIO host controller to process the IOs. The controller may include SPI controller to support SPI mode for the IOs and supports the needed protocol functionality internally. Maximum clock rate supported for SPI is 20 MHz for a maximum of 20 Mbps data transfers. There is an optional 4-bit SD mode, which uses 4 data lines. Maximum clock rate supported is 25 MHz for maximum 100 Mbps SD bit data transfer in 4-bit SD mode. Four serial bits simultaneously transmits at four times clock rate on 4 SD lines in this mode. Four-bit SD mode is compromise between serial and parallel bits communication to enhance serial transfer rate four times. In 1-bit SD mode, with 25 MHz clock the maximum data transfer rate is 25 Mbps and one serial bit transmits at 1 line only.

SDIO card has a control section called function 0. It necessarily uses function 1 and optionally uses functions between 2 and 7 (depending on the application in devices used such as Bluetooth, PHS, GPS or digital camera). Each function has PCMLA (Personal Computer Manufacturer Interface Adapter) defined card information structure and registers, for example, ID number, function enable bit, supported bus width (1 or 4), voltage, power needs, clock rates and interrupt enable bit. Each function's specifications for the register bits and protocols have been defined in SD standard. A standard device driver can therefore be written. A new function can also be defined.

Data communication is in 48-bit command/request format for 48-bit control register/ status register bits and supports data transfer of blocks of bytes. For single byte transactions, SDIO card may also include a UART 16550 mode communication over the SD bus.

SDIO is an SPI based 9 pin connector card, which supports SPI as well as 1-bit SD or 4-bit SD communication. SDIO supports 8 logic functions. SDIO functions include IOs with several protocols, for example, IrDA adapter, UART 16550, Ethernet adapter, GPS, WiFi, Bluetooth, WLAN, digital camera, barcode or RFID code reader.

3.3 PARALLEL DEVICE PORTS

The parallel port of devices transfers number of bits over the wires in parallel. Parallel wires capacitive effect reduces the length up to which parallel communication can be done. High capacitance results in delay for the bits at the other end undergoing transition from 0 to 1 or from 1 to 0. High capacitance can also result in noise and cross talk (induced signals) between the wires. Therefore, parallel port carries the bits upto short distances, generally within a circuit board or IC.

Figure 3.4(a) shows the parallel input, output, and bi-directional device ports. Figure also shows a device-interfacing circuit with the processor and system buses. Parallel port inputs I0 to I7 may be to a keypad controller. Parallel port outputs O0 to O7 may be output bits to LCD display output controller. BR_i and BR_o are the input and output data buffers at bi-directional IO port.

A device port connects to the address bus signals, A_i and A_j through a port address decoder. \overline{IORD} and \overline{IOWR} are additional control signals for a-port device read and write, respectively, in case of an 80x86 processor, which has IO mapped IOs. The memory read and write signals, \overline{RD} and \overline{WR} are used in the processor with memory mapped IOs [Section 2.2.2].

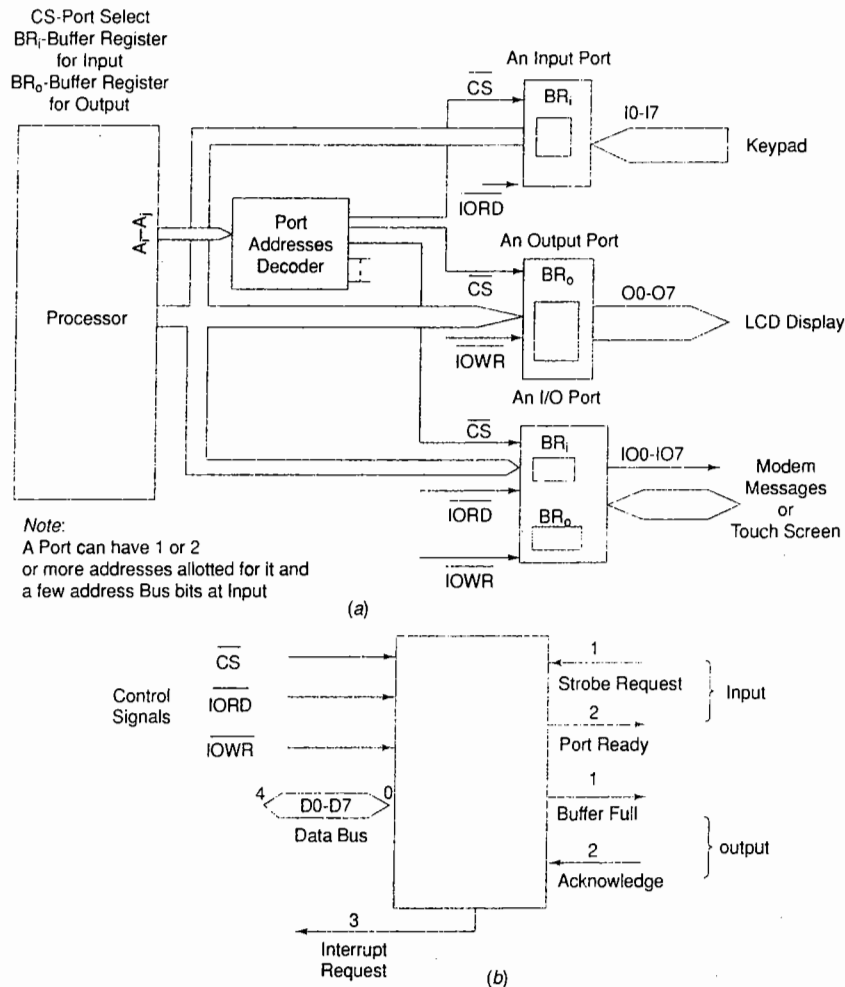


Fig. 3.4 (a) Parallel input port, output port, and a bi-directional port for connecting the device
(b) The handshaking signals when used by the IO ports

Example 3.4

IBM personal computer has a parallel port with a 25 pin connector. There are 8 IO pins, 5 input pins for status signals (four active high S_3 to S_6 , one active low $\overline{S_7}$) from external device port (for example, printing device port) and 4 output pins for control signals (one active high C_2 and three active low C_0 , C_1 and C_3). The 8 pins are ground pins (Pins at 0 V). The status pins and control pins are provided for handshaking between peripheral and computer.

Figure 3.4(b) shows the handshaking signals. An external input device to the device port makes a strobe request, *STROBE*, after it is ready to send the byte and the system IO device sends the acknowledgement, *PORT READY* when BR_i (receiving buffer) is empty.

An external output receiving device sends the message *ACKNOWLEDGE* when the IO device port ends the *BUFFER FULL* signal. The processor is sent the *INTERRUPT REQUEST* message when BR_o transmitting buffer is empty not full (available for next write) or when the receiving buffer is full (available for next read). This enables the processor to interrupt and retransmit next byte(s) in next cycle or receive the byte(s) from input using the appropriate service routines for output or input from the port, respectively.

Example 3.5

Intel 8255 is a programmable peripheral interface (PPI) chip. A PPI device has four addresses, three for the ports and one for the control word. There are three 8-bit ports: port A, B and C. Port C can also be programmed to function in bit set-reset mode. Each bit of this port can be set to 1 or reset to 0 by an appropriate control word. Alternatively, the ports can be grouped as Group A (Port A and Port C upper four bits) or Group B (Port B and Port C lower four bits).

1. In mode 0 programming for a group, each port group does not use handshaking signals.
2. Mode 2 programming is used for port A as input as well as output. In mode 2 programming for the group A, port A uses handshaking signals, *STROBE*, *PORT READY*, *BUFFER FULL*, *ACK* and *INTERRUPT* and port A functions as a bi-directional IO port.
3. Mode 1 programming is either for port as input or as output. In mode 1 programming for the group A or B, port A or B uses only one of the two handshaking signal pairs, either (*STROBE*, *PORT READY*) or (*BUFFER FULL*, *ACK*) plus one *INTERRUPT* signal.

The following characteristics are taken into consideration when interfacing a device port.

1. A device port may have multi-byte data input buffers and data output buffers. Suppose there is an eight-byte buffer. Assuming that a device (as in the 80196 microcontroller) can generate three interrupts, one on receiving a byte, one on receiving the fourth byte and one when the buffer is full, then the deadline for servicing these interrupts increases up to eight times compared to the case when there is a single byte register instead of buffer.
2. A port may have a DDR (Data Direction Register) (as in the 68HC11 microcontroller). This is an advantage since each bit of the port is now programmable. It can be set as input or output. DDR programs the port bits.
3. Port LSTTL-driving capability and port-loading capability are important characteristics. A port may be an OD (open drain) port. It has zero driving capability unless the drain connects the positive supply voltage. If the given port has OD gates, an appropriate pull-up resistance or transistor is connected to each port pin to provide the driving capability. The drain or collector connects to the supply voltage to provide the pull-up.
4. If a given port is *quasi bi-directional* (as in 80196), then the port pins have limited driving capability, which suffices for a period of one or a few clock cycles and drives a LSTTL gate for that period. When this device port connects to more than one LSTTL, then an appropriate pull-up circuit will be required for the port pins.
5. There may be multiple or alternate functionality in the port pins; for example, 80196 input port pins. Each pin of P2 has an alternative use as multi-channel analog input facility for 8 analog inputs. Another example is 8051 two ports P0 and P2. These port bits also have an alternate function in that they bring out when needed the internal multiplexed buses for the external program and memories whenever the

internal memory is insufficient. Each pin of P3 in 8051 has multiple uses. These are used during serial communication, timer/counter signals, interrupt-signals, and \overline{RD} and \overline{WR} control signals for external memories. 68HC11 ports B and C are of 8 bits each and have alternative uses for the port pins in it. One of the alternate functions is to bring out the internal address and data buses, respectively.

6. A port may have provision for multiplexed output to connect to multiple systems or units.
7. A port may have provision for demultiplexed inputs from multiple systems or units.

A parallel device port can have parallel inputs, parallel outputs, bi-directional and quasi-bi-directional IOs. A parallel device port can have handshaking pins. A parallel device port can also have control pins for control-signal outputs to external circuit and status pins for inputs of status signals to external circuits.

3.3.1 Parallel Port Interfacing with Switches and Keypad

A 16 keys keypad has many applications. A mobile smart phone device has 16 keys and four menu: select up, down, left, right keys. Assume that an IO device has two ports, A and C. The device has a processing element which functions as a keypad-controlling device (controller).

Figure 3.5(a) shows how a set of switches or a keypad of 16 keys and four menu-select keys can interface to the device. Four bits of an 8-bit input port A (A_4 - A_7) can be used for the four menu select keys. Assume that the idle state logic state equals 1. The 16 keys can be considered as arranged in four rows and four columns. The other four bits of A (A_0 - A_3) are inputs from sense lines from four rows. Assume that the idle state logic state is equal to 1. The four bits of output port C (C_0 - C_3) are output to sense lines in four columns.

The processing element in device activates for polling the output port C ten times each second and sends C_0 - $C_3 = 0000$; after a wait it reads D_0 - D_7 and A_4 - A_7 . The processing element computes the code of the pressed key and generates a status signal when a key is found pressed. From the bit pattern found at A_0 - A_3 , the processing element computes 7-bit ASCII code of the pressed key at that instance and can output that code at D_0 - D_6 . It also outputs $D_7 = 1$ when a specific key is found pressed, else $D_7 = 0$. The processing element also processes the bounces when a key is pressed. This takes care of bouncing effects. The processing element is thus functioning as a keypad controller, as it is keypad specific.

Example 3.6

A mobile phone keypad is smart and is called T9 keypad. Nine keys are used to enter not only the numbers but also text of messages. The processing element is programmed as a state machine to compute the ASCII code to be sent. A state machine generates the states. For example, a key marked as number 5 is in state (0, 5) in reset state, which is also its idle state. The key-state undergoes transition to state (1, 5) when it is pressed first time. When it is pressed second time within 1 s, the key state becomes (1, j). This state corresponds to character j. If it is pressed third time within 1 s, the key-state becomes (1, k). The state of the key changes in a cyclic fashion. (1, 5) \rightarrow (1, j) \rightarrow (1, k) \rightarrow (1, l) \rightarrow (1, 5) \rightarrow (1, j), The transition of a key state occurs only if it is found pressed within 1 s of the previous transition, and the appropriate action takes place as per the state. The processing element computes the ASCII code from the read value of A_0 - A_3 and key state at an instance. After processing is over or after 1s, the key-state resets to (0, 5).

Two key states simultaneously or separately undergoing transitions can define a transition to another state. For example, when there is transition to (1, j) state after another key state is (1, #), then (1, j) undergoes another transition to (1, j), and when that key state is (0, #) it remains at (1, j).

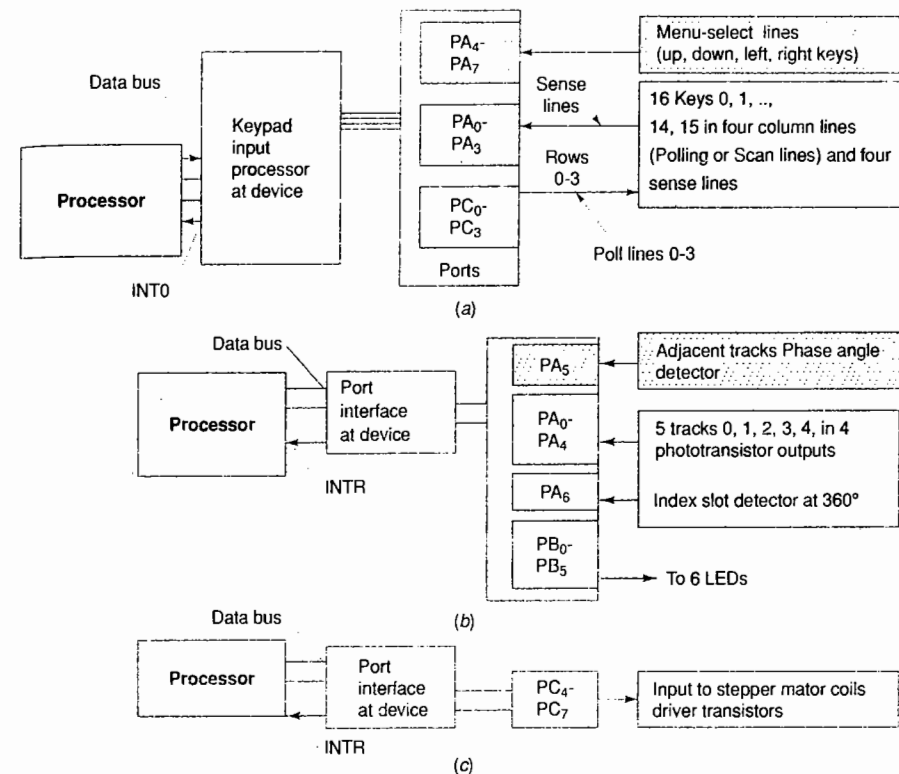


Fig. 3.5 (a) Parallel input port A and a four bit output port C used for interfacing a set of 16 keys in keypad and four menu select keys (b) Parallel input port A connected to an encoder circuit which senses the rotated or linear position of a moving shaft and port B connected to 6 LEDs (c) Four bit parallel output port C connected to a stepper motor

A parallel device having a number of input and output bits can be used to find the code of the pressed key in a matrix of keys. A keypad controller has a processing element to compute the code of the pressed key and to generate a status signal when any key is found pressed. A mobile phone keypad controller processes the states of the keys to enable application of same keypad for dialing as well as editing SMS messages.

3.3.2 Parallel Port Interfacing with Encoders

Encoder is a device that measures angular or linear position of a rotating or moving shaft. It has application in robots and industrial plants. A rotatory-angle encoder has multiple tracks on a rotating disk. Each track has half of the segments transparent and half opaque. A linear encoder has a multi-slotted plate. A set of n infrared (IR) LED and phototransistor pairs generate n-bit inputs for a port. The encoder connects to parallel port, as shown in Figure 3.5(b).

3.3.3 Parallel Port Interfacing with Stepper Motor

A stepper-motor rotates by one step angle when its four coils are given currents in a specific sequence and that sequence is altered. For example, assume that currents at an instance equal $+i, 0, 0, 0$ in four coils X, X', Y, Y' . The motor rotates by one step when the currents change to $0, +i, 0, 0$. The sequences at intervals of T are changed as follows: $1000, 0100, 0010, 0001, 1000, 0100, \dots$ [The bits in the nibble (set of 4 bits) rotate by right shift.] Here 1 corresponds to $+i$. The motor thus rotates n step angles in interval of $(n.T)$. The sequences are changed to rotate the motor in the opposite direction, as follows: $0001, 0010, 0100, 1000, 0001, 0010, \dots$ [The bits in the nibble (set of 4 bits) rotate by left shift.] Alternately, the coils are given the currents in the sequence of $1100, 0110, 0011, 1001, 1100, 0110, \dots$, or $0011, 0110, 1100, 1001, 0011, 0110, \dots$. The motor rotates $(n/2)$ steps in interval equals to $(n.T/2)$. T is the period of clock pulses that drives the motor by change of coil currents to the next sequence.

The coils connect to parallel port 4 output pins, as shown in Figure 3.5(c). Alternatively, a processing element called stepper-motor driver can be used. The driver is given two outputs from the port: clock pulses and a rotating direction bit r . For example, if $r = 1$, motor rotates clockwise and if $r = 0$ then motor rotates anti-clockwise. The motor rotates as long as clock pulses are given at the output PC_4 - PC_7 .

3.3.4 Parallel Port Interfacing with LCD Controller

An LCD controller has a processing element that needs three control signals as inputs and 8 input/output bits for parallel set of 8 IO bits. Eight-bit *parallel output port B* pins PB_0 - PB_7 connect LCD controller, as shown in Figure 3.6(a). LCD controller also connects to one bit PC_0 at an output port for RS (register select) signal.

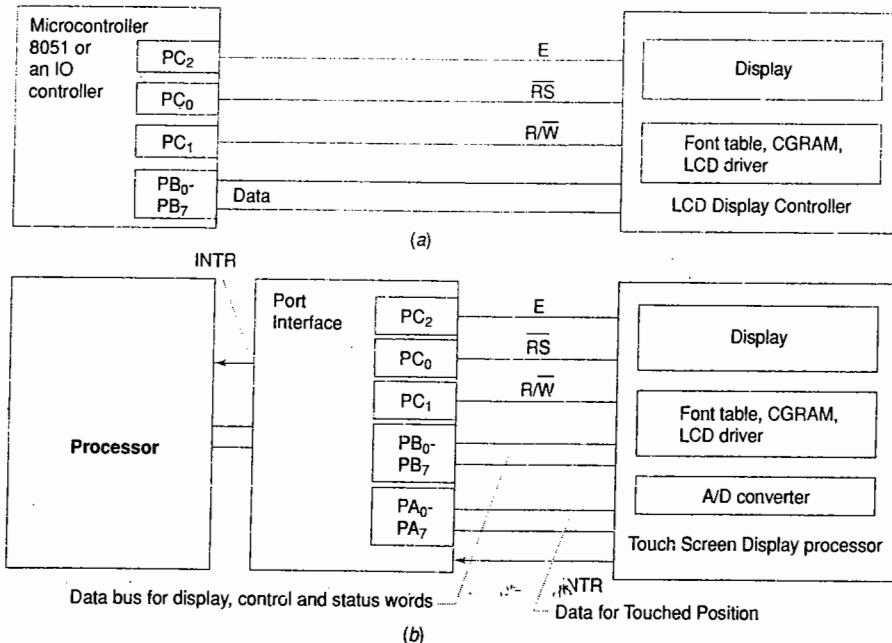


Fig. 3.6 (a) Eight bit parallel output port B connected to an LCD controller (b) 8-bit parallel output port B and 8-bit parallel input port A connected to a touch screen control circuit

When RS is reset as 0, PB_0 - PB_7 communicates a control word to control register of the LCD controller. When RS is set as 1, PB_0 - PB_7 communicates data to the LCD controller.

The LCD controller also connects to a one bit PC_1 at output port for R/\overline{W} (read/write). PC_1 is set to 1 when status register of LCD controller is read from PB_0 - PB_7 . PC_1 is reset to 0 when writing into LCD controller the PB_0 - PB_7 bits. The processing element generates all signals required for LCD displays.

The LCD controller is sent control words and data words for initialization and programming PB_0 - PB_7 bits, PC_0 and PC_1 outputs for each word to LCD controller. The controller then has to be enabled by sending 1 at E pin. It connects to one bit PC_2 at output port for E (enable). There is an interval in which the controller may be in disabled state. During this interval, it cannot accept instructions or data through the output of control word or data port pins. For example, a control instruction is to clear display. The internal processing element has to clear the bytes at all the N addresses in N characters LCD display. Assume that in a typical LCD, it is $150 \mu s$. When the first 1 is written at E , then 0 is written and a $150 \mu s$ delay program is called in-between; the E output creates a negative going pulse at LCD controller. It disables sending of any control word or data for a period of $150 \mu s$.

LCD controller has M displayed character ROM addresses. $M = 128$ for 128 ASCII codes. For each distinct ASCII character, there is a 64-bit graphic. The LCD controller has an internal CGRAM (character graphic RAM). For each ASCII character, 8 bytes are sent from font table ROM to CGRAM address. CGRAM has N addresses. $N = 64$ when 64 characters are displayed at the LCD. An address changes by incrementing or decrementing the cursor position to the previous or next address on screen. By sending appropriate control words followed by data, the LCD controller is programmed to display up to 64 characters on the screen.

A parallel device having 8 output data and 3 bits for E , \overline{RS} and R/\overline{W} can be used to connect to an LCD controller.

3.3.5 Parallel Port Interfacing with Touchscreen

Touchscreen is an input device cum LCD display device. It is also interfaced through IO port B functioning as data bus for display, control and status words to an LCD display device controller. The interface uses an additional input port A for a byte, which corresponds to the address of the position touched on the screen.

A touchscreen is either resistive or capacitive. On touching at a position on the screen, there is change in resistance or capacitance, which depends on the touched position. A touch can be finger or stylus. The stylus is about one-fifth thinner than a pencil and about half of the length of the pencil. The resistance or capacitance is a part of a bridge circuit that generates an analog voltage. An 8-bit ADC is given an input from a bridge circuit and the 8-bit ADC output connects to 8-bit input port A.

Eight-bit *parallel IO port B* pins PB_0 - PB_7 , E , \overline{RS} and R/\overline{W} and eight-bit *parallel input port A* connect to touch screen circuit ports as shown in Figure 3.6(b). An interrupt signal $INTR$ is issued whenever the screen is touched.

Example 3.7

A PocketPC has a touchscreen. The touchscreen device facilitates GUIs. It can display menus as well as a virtual keypad. Using the keypad on screen and stylus, a set of characters can be entered for creating or editing SMS messages, e-mails, or other files. The stylus is held like a pencil and is used to touch the virtual keypad and then the device selects the menu and commands on the screen.

A parallel device having 8 input data bits from an ADC and 8 IO data bus and 4 bits for INTR, E, \overline{RS} and R/\overline{W} can be used to connect a port interface with a processing element to a touchscreen. The ADC generates input bits for the port from the analog signal, which is as per the touched position on the screen.

3.4 SOPHISTICATED INTERFACING FEATURES IN DEVICE PORTS

A device port may not be as simple as the one for a stepper motor port or for a serial line UART. Nowadays, a complex embedded system has highly sophisticated IO devices, for example, SDIO card (Section 3.2.6), IO devices with fast serialization and de-serialization of data, fast transceiver, and real time video processing system. The following are the few sophisticated interfacing device and port features.

1. Let the operation voltage level expected for logic state 1 = 5 V (TTL or CMOS). The Schmitt trigger circuit has a property that when a transition from 0 to 1 occurs, only if the voltage level exceeds 2/3 of the 5 V level is there a transition to 1. Similarly, when a transition from 1 to 0 occurs, only if the voltage level lowers below 1/3 of the 5 V level is there a transition to 0. Hence, the Schmitt trigger circuit eliminates noise as large as 2/3 of 5 V, or 3.3 V, when it is superimposed at an input line to the device. One great advantage of the in-built Schmitt trigger circuit at the port is conditioning of the signal by noise elimination. Otherwise, a device port input will need an external chip for Schmitt trigger-based noise elimination. Such a device is used in transceivers for repeating systems, which are used in long distance communication.
2. When a device port is waiting for instructions, power management can be done at the gates of the device. Lately, a new technology called DataGate (from Xilinx) has been developed for use at ports. DataGate is a programmable ON/OFF switch for power management; DataGate makes it possible to reduce power consumption by reducing unnecessary toggling of inputs when these are not in use. The great advantage of an inbuilt DataGate-like circuit at a device port is reduced power dissipation when the device port is operated at fast speeds. Such a device is extremely useful in systems connected to a common bus and there is a need to control unnecessary input toggling. For example, in a bus interface unit, the input signals should activate only when the input has to be passed to the circuit. As the number of bus interfaces in the system grows, the demand to prevent needless switching of input signals increases.
3. Earlier, port interfaces used to be either open drain CMOSs or TTLs or RS232Cs. (i) Nowadays, a system may be required to operate at a voltage lower than 5 V. [Recall Section 1.3.1.] Low Voltage TTL (LVTTTL) and Low Voltage CMOS (LVCMOS) gates may be used at the device ports for 1.5 V IO. (ii) Nowadays, a system may be required to operate using advanced IO standard interfaces. Examples are High Speed Transceiver Logic (HSTL) and Stub-series Terminated Logic (SSTL) standards. HSTL is used for high-speed operations; SSTL is used when the buses are to be isolated from relatively large stubs.
4. A device connects to a system bus and also to IO bus when it is networked with other devices. Device and bus-impedances during an IO should match. Else, line reflections occur. Recent developments make it feasible to match these dynamically. For example, a new technology, called iCITE (Xilinx Controlled Impedance Technology) can be used. The great advantage of an inbuilt device for dynamically matched impedances is that when resistors are replaced with digitally, dynamically controlled and matched impedances in the devices, there are no line reflections and therefore no missing bits or bus faults.

5. An IO device may consist of multiple gigabit (622 Mbps to 3.125 Gbps) transceivers (MGTs). Special support circuitry is needed for this rate. Rocker IO™ serial transreceivers are examples of circuits that provide support circuitry at this rate.
6. A device for an IO may integrate a SerDes (serialization and de-serialization) subunit. SerDes is a standard subunit in a device where the bytes placed at 'transmit holding buffer' serialize on transmission, and once the bits are received these de-serialize and are placed at the 'receiver buffer'. Once the device SerDes subunit is configured, serialization and de-serialization is done automatically without the use of the processor instructions. The great advantage of the SerDes unit is that these operations are fast when compared to operations without SerDes. [A device for IO may integrate a DAA (direct access arrangement using analog IOs along with one master and seven slave CODECs) or McBSP (multi channel buffered serial port with high speed communication) subunit when serializing.]
7. Recently, multiple IO standards have been developed for IO devices. A support to the multiple IO standards may be needed in certain embedded systems. A technology, Flexible Select IO™ -Ultra technology, supports over 20 single-ended and differential IO signaling standards. Advantages of multiple standard device ports are obvious.
8. An IO device may integrate a digital Physical Coding Sublayer (PCS). Analog audio and video signals can then be pulse code modulated (PCM) at the sublayer. The PCS sublayer directly provides codes from analog inputs within the device itself. The codes are then saved in the device data buffers. The advantage of an inbuilt PCS at device port is that there is then no need of external PCM coding. Besides, these operations are performed in the background as well as fast. It improves the system's performance when there are multimedia inputs at the device.
9. A device for IO may integrate an analog unit Physical Media Attachment (PMA) for connecting direct inputs and outputs of voice, music, video and images. The great advantage of inbuilt PMA is that the device directly connects to physical media. PMA is needed for real-time processing of video and audio inputs at the device.

Nowadays, IO devices have sophisticated features. Schmitt trigger inputs are used for noise elimination. Devices with low voltage gates and devices using power management by preventing unnecessary toggling at the inputs are used for sophisticated applications. Dynamically controlled impedance matching is a new technology and it eliminates line reflections when interfacing the devices. The SerDes subunit serializes and deserializes outputs and inputs in the devices. A port may have DAA, McBSP, PCS and PMA subunits for analog IOs for video and audio devices.

3.5 WIRELESS DEVICES

Wireless devices have become very common in recent years for serial transmission of bits.

Wireless devices use infrared (IR) or radio frequencies after suitable modulation of data bits. IrDA (Section 3.13.1), Bluetooth (Section 3.13.2), WiFi, 802.11 WLAN (Section 3.13.3) and ZigBee (Section 3.13.4) have become popular protocols for wireless communication of data bits from a source to the receiver.

An IR source communicates over a line of sight and the receiver phototransistor is used for detecting infrared rays. Example of applications of IR communication includes handheld TV remote controllers and robotic systems. IR devices use IrDA protocol.

Radio frequencies communicate over short and long distances. The transmitter and receiver use antennae to transmit and receive signals and modulator and demodulators to carry the data bits using RF frequencies. Mobile GSM wireless devices use 890–915 MHz, 1710–1785 MHz, or 1850–1910 MHz bands. Mobile CDMA wireless devices use 2 GHz carrier frequencies. Bluetooth and ZigBee wireless devices (Sections 3.13.2 and 3.13.4) use 2.4 GHz or 900 MHz frequencies.

The number of frequency bands is limited, while a large number of devices may need to communicate. Therefore, time and frequency division multiplexing are used. An innovative method is radio frequency hopping over a wider spectrum, as in Bluetooth devices. The transmitted carrier frequencies hop among different channels at a given hopping rate. The transmitter modulates the data bits as per protocol specifications. The receiver tunes to these hopped carrier frequencies at a given hopping rate and in the same hopping sequence as the ones used by the transmitter. The receiver demodulates and detects the data bits as per physical-layer protocol used for transmitting.

Several wireless devices network use FHSS or DSSS transmitters and receivers. Popular protocols are IrDA, Bluetooth, 802.11 and ZigBee.

3.6 TIMER AND COUNTING DEVICES

Most embedded systems need a timing device.

3.6.1 Timing Device

A timer device is a device that counts the regular interval (δT) clock pulses at its input. The counts are stored and incremented on each pulse. It has output bits (in a count register or at the output pins) for the period of counts. The counts multiplied by interval δT gives the time. The $(\text{counts} - \text{initial counts}) \times \delta T$ interval gives the time interval between two instances when the present count bits are read and the initial counts are read. It has an input pin (or a control bit in a control register) for resetting to make all count bits = 0. It has an output pin (or a status bit in status register) for output when all count bits equal 0 after reaching the maximum value, which also means timeout on the overflow.

3.6.2 Counting Device

A counting device is a device that counts the input for events that may occur at irregular or regular intervals. The counts gives the number of input events or pulses since it was last read.

Blind Counting Synchronization A counting device may be a free running (blind counting) device with a prescaler for the clock input pulses and for comparing the counts with the ones preloaded in a compare register. The prescaler can be programmed as $p = 1, 2, 4, 8, 16, 32, \dots$, by programming a prescaler register. It divides the input pulses as per the programmed value of p . It has an output pin (or a status bit in the status register) for output when all count bits equal 0 after reaching the maximum value, which also means after timeout or on overflow. The counter overflows after $p \times 2^n \times \delta T$ interval. It can have an input pin (or a control bit in control register) for enabling an output when all count bits equal count preloaded in the compare register. At that instance, a status bit or output pin also sets in and an interrupt can occur for event of comparison equality. This device is useful for the alarm or processor interrupts at preset instances or after preset intervals with respect to another event from another source.

The counting device may be the free running (blind counting) device with a prescaler for the clock input pulses, for comparing the counts with the ones preloaded in a compare register as well as for capturing counts on an input event. This device functions are similar to the above, but there is an addition input pin for sensing an event and for saving the counts at the instance of that event. At this instance, a status bit can also set in and a processor interrupt can occur for the capture event.

The above device is useful for alarm generation and processor interrupts at the preset times as well as for noting the instances of occurrences of the events and processor interrupts for requesting the processor to use the captured counts on the events. Alarm generation can be synchronized with the input capture events. Writing counts into the compare register does this. Counts in the register are set equal to capture register counts plus additional counts, which define the interval after which an alarm is to be generated.

A blind counting free running counter with prescaling, compare and capture registers has a number of applications. It is useful for action or initiating a chain of actions, and processor interrupts at the preset instances as well as for noting the instances of occurrences of the events and processor interrupts for requesting the processor to use the captured counts on the events for future actions.

3.6.3 Timer cum Counting Device

A timer cum counting device is a counting device that has two functions. (1) It counts the input due to the events at irregular instances and (2) It counts the clock input pulses at regular intervals. An input or a status bit in the timing device register controls the mode as timer or counter. The counts gives the number of input events or pulses since it was last read. It has an output pin (or a status bit in status register) for output when all count bits equal 0 after reaching the maximum value, which also means timeout or overflow interrupts to the processor.

Table 3.5 lists twelve uses of a timer device. It also explains the meaning of each use.

Table 3.5 Uses of Timer Device

S.No.	Applications and Explanation
1.	Real Time Clock Ticks (functioning as system heart beats). [Real time clock is a clock that once the system starts it, does not stop and can't be reset. Its <i>count value</i> can't be reloaded. <i>Real time endlessly flows and never returns!</i>] Real Time Clock is set for ticks using prescaling bus and rate-set bits in appropriate control registers. Section 3.8 gives the details.
2.	Initiating an event after a preset delay time. Delay is as per <i>count-value</i> loaded.
3.	Initiating an event (or a pair of events or a chain of events) after a comparison between the preset time with counted value. Preset time is loaded in a Compare Register. [It is similar to presetting an alarm.]
4.	Capturing the <i>count-value</i> at the timer on an event. The information of <i>time</i> (instance of the event) is thus stored at the <i>capture register</i> .
5.	Finding the time interval between two events. <i>Counts</i> are captured at each event in the capture register and read. The intervals are thus found out. A service routine does the counts read on interrupt.
6.	Wait for a message from a queue or mailbox or semaphore for a preset time when using an RTOS. There is a predefined waiting period before RTOS lets a task run without waiting for the message. (Section 7.4)

(Contd)

S.No.	Applications and Explanation
7.	Watchdog timer. It resets the system after a defined time. Section 3.7 gives details.
8.	Baud or Bit Rate Control for serial communication on a line or network. Timer timeout interrupts define the time of each baud.
9.	Input pulse counting when using a timer, which is ticked by giving non-periodic inputs instead of the clock inputs. The timer acts as a counter if, in place of clock inputs, the inputs are given to the timer for each instance to be counted.
10.	Scheduling of various tasks. A chain of software-timer interrupts and RTOS uses these interrupts to schedule the tasks.
11.	Time slicing of various tasks. A multitasking or multiprogrammed operating system presents the illusion that multiple tasks or programs are running simultaneously by switching between programs very rapidly, for example, after every 16.6 ms. This process is known as <i>context switch</i> . RTOS switches after preset time-slice from one running task to the next. Each task can therefore run in predefined slots of time.
12.	Time division multiplexing (TDM). Timer device is used for multiplexing the input from a number of channels. Each channel input is allotted a distinct and fixed-time slot to get a TDM output. [For example, multiple telephone calls are the inputs and TDM device generates the TDM output for launching it into the optical fibre.]

A timing device has number of states and Table 3.6 gives the states.

Table 3.6 States in a timer

S.No.	States
1.	Reset State (initial count equals 0)
2.	Initial Load State (initial count loaded)
3.	Present State (counting or idle or before start or after overflow or overrun)
4.	Overflow State (count received to make count equal 0 after reaching the maximum count)
5.	Overrun State (several counts received after reaching the overflow state)
6.	Running (Active) or Stop (Blocked) state
7.	Finished (Done) state (stopped after a preset time interval or timeout)
8.	Reset enabled/disabled State (enabled resetting of count equal 0 by an input)
9.	Load enabled/disabled State (reset count equals initial count after the timeout)
10.	Auto Re-Load enabled/disabled State (enabled count equals initial count after the timeout)
11.	Service Routine Execution enable/disable State (enabled after timeout or overflow)

At least one hardware timer device is a must in a system. It is used as a system clock. Let number of system clock ticks needed before a system interrupt occurs equals *numTicks*. The hardware timer gets the input from a clock-out signal from the processor and activates the system clock tick as per the *numTicks* preset at the hardware timer. On each system clock tick, the user-mode task interrupts and the system takes control. The system enables the privileged mode actions and the CPU context switches as per the preset state of the system. The system control actions are performed by operating system (software).

Figure 3.7 shows hardware timer control bits (and signals) and status flags. *Control bits* are as per the hardware signals and corresponding bits at control register. Control bits (or signals) can be of nine types. These are: (i) Timer enable (to activate a timer). (ii) Timer start (to start counting at each clock input). (iii) Timer stop (to stop counting) from the next clock input. (iv) Prescaling bits (to divide the clock-out frequency signal from the processor). (v) Up count Enable (to enable counting up by incrementing the count value on each clock input) (vi) Down count Enable (to decrement on a clock input). (vii) Load enable (to enable loading of a value at a register into the timer). (viii) Timer-interrupt enable (to enable interrupt servicing when the timer overflows (overflows) and reaches *count value* equals 0) (ix) Time out enable [to enable a signal when the timer overflows (reaches count equals 0)] to another device.

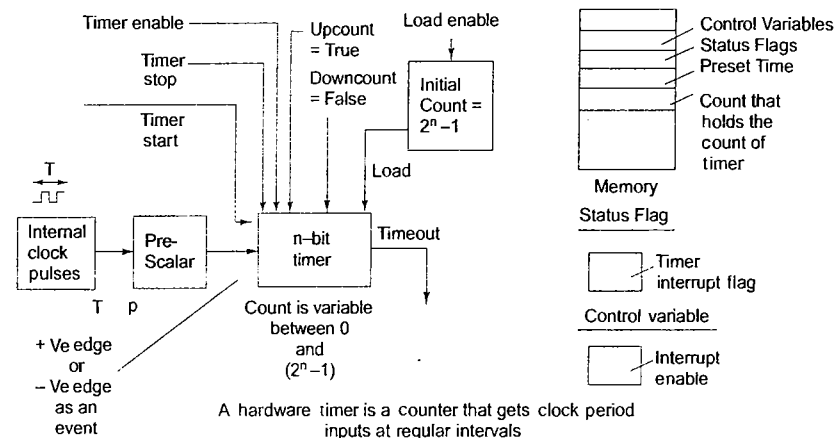


Fig. 3.7 Signals, clock-inputs, control bits and status flags at registers or memory in a hardware timer device

Status flag is as per the corresponding hardware signal time-out from the hardware timer. This flag and signal set when the timer all bits (*count value*) reach to 0.

Table 3.7 lists ten forms of the timers for the uses listed in Table 3.5. Software timer (SWT) is an innovative concept.

The system clock or any other hardware-timing device ticks and generates one interrupt or a chain of interrupts at periodic intervals. This interval is as per the *count-value* set. Now, the interrupt becomes a clock input to an SWT. This input is common to all the SWTs that are in the list of activated SWTs. Any number of SWTs can be made active put in a list of active SWTs. Each SWT will set a status flag on its timeout (*count-value* reaching 0). Figure 3.7 shows the control bits and status bits in an SWT. SWT *control bits* are set as per the *application*. There is no hardware input or output in an SWT. A flag sets when the SWT *count-value* reaches 0 after reading the maximum. Table 3.8 lists all the variables of SWT. It includes the control-bits and status flags. SWT thus has similar control variables and flags as in the hardware timer or counter.

SWT actions are analogous to that of a hardware timer. While there is physical limit (1, 2 or 3 or 4) for the number of hardware timers in a system, SWTs can be limited by the number of interrupt vectors provided by the user. Processors (microcontrollers) also define the interrupt vector addresses of two or four SWTs.

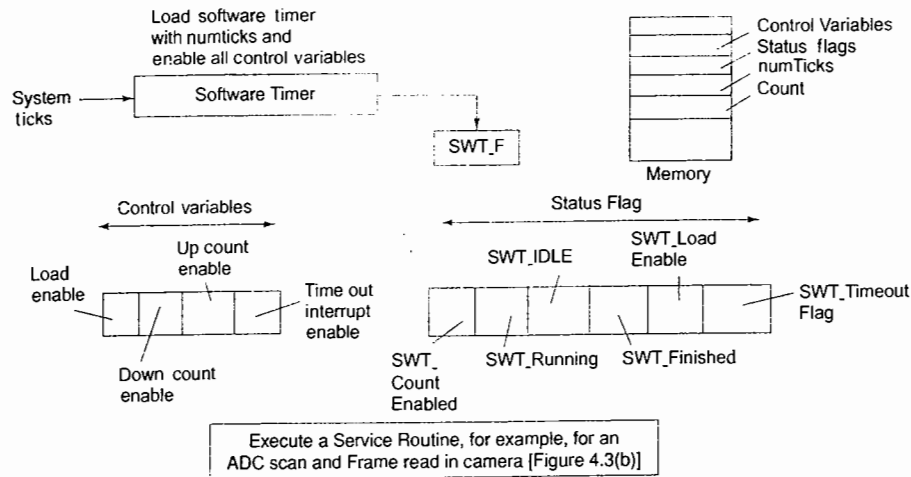


Fig. 3.8 Control bits, status flags and variables of a software timer

Table 3.7 Ten forms of a timer

S.No.	Types
1.	Hardware internal timer
2.	Software timer (SWT)
3.	User software-controlled hardware timer.
4.	RTOS-controlled hardware timer. An RTOS can define the clock ticks per second of a hardware timer at a system. [Refer to function OS_Ticks (N) in Section 9.2.1.]
5.	Timer with periodic time-out events (auto-reloading after overflow state). A timer may be programmable for auto-reload after each time-out.
6.	One Shot Timer. (No reload after the overflow and finished state.) It triggers on event-input for activating it to running state from its idle state. It is also used for enforcing time delays between two states or events. On the event or reaching a state one shot timer starts and after the time out another state or event occurs.
7.	Up count action Timer. It is a timer that increments on each count-input from a clock.
8.	Down count action timer. It is a timer that decrements on each count-input.
9.	Timer with its overflow status bit (flag), which auto-resets as soon as interrupt service routine starts running.
10.	Timer with overflow-flag, which does not auto reset.

Timing devices are needed for a number of uses in a system. (i) There can only be a limited number of hardware timers present in the system. A system has at least one hardware timer. The system clock is configured from this. A microcontroller may have 2, 3 or 4 hardware timers. One of the hardware timer ticks forms the inputs from the internal clock of the processor and generates the system clock. Using the systems clock or internal clock, the number of software timers can be driven. These timers are programmable by the device driver programs. (ii) A software timer is software that executes and increases or decreases a count-variable (count value) on an interrupt on a timer output or on a real-time clock interrupt. The software timer also generates interrupt on overflow of count-value or on finishing value. Software timers are used as virtual timing devices. There are a number of control bits and a time-out status flag in each timer device.

Table 3.8 Variables for control bits and status in a software timer

S.No.	32 or 16 or 8 or 1-bit variables
1.	Reset Value 32/16/8
2.	Initial Load Value (numTicks) 32/16/8
3.	Count-value (Preset value) 32/16/8
4.	Maximum Value 32/16/8
5.	Minimum Value 32/16/8
6.	Timer run enable bit
7.	Timer interrupt enable bit
8.	Timer reset enable bit
9.	Timer load enable bit
10.	Timer reload (after finished state) enable bit
11.	Overflow-flag

3.7 WATCHDOG TIMER

Watchdog timer is a timing device that can be set for a preset time interval, and an event must occur during that interval else the device will generate the timeout signal. For example, we anticipate that a set of tasks must finish within 100 ms. The watchdog timer disables and stops in case the tasks finish within 100 ms. The watchdog timer generates interrupts after 100 ms and executes a routine that runs because the tasks failed to finish in the anticipated interval. A software task can also be programmed as a watchdog timer (Section 9.3.3). A microcontroller may also provide for the watchdog timer.

The watchdog timer has a number of applications. One application in a mobile phone is that the display is turned off in case no GUI interaction takes place within a specified time. The interval is usually set at 15, 20, 25, or 30 s in a mobile phone. This saves power.

Another application in a mobile phone is that if a given menu is not selected by a click within a preset time interval, another menu can be presented or a beep can be generated to invite user's attention.

An application in a temperature controller is that if a controller takes no action to switch off the current within the preset time, the current is switched off and a warning signal raised, indicating controller failure. Failure to switch off current may cause a boiler in which water is heated to burst.

Example 3.8

68HC11 microcontroller has a watchdog timer in the hardware. There are two registers, CONFIG (system configuration control register) and COPRST (computer operating properly and processor reset on failure). They are for programming the interrupts of watchdog timer. CONFIG has a bit, NOCOP. It configures when the processor writes the configuration word at address 0x003F. NOCOP is the 2nd bit of CONFIG. If this bit is reset to 0, the COP facility is enabled. [COP means computer (68HC11) operating properly watchdog timer. The COP watchdog timer provides for keeping a watch on execution time of the user program.]

When user program takes a longer time in a routine than planned or expected the user provides for storing at desired intervals; first, the 0x55 and then the 0xAA at the computer-reset control register COPRST. By keeping a watch means that as soon as the watchdog timer overflows (time outs), the program counter is reset according to 16 bits at the lower and higher bytes that are preloaded at addresses 0xFFFF and

0xFFFFB, respectively. If these 16 bits are same as the bits in 0xFFFFE and 0xFFFFF, then the microcontroller executes instructions, which are same as when it resets on power up or else it executes the routine at the 16-bit address fetched from 0xFFFFE and 0xFFFFF whenever there is failure within the watched time interval.

The 0th and 1st bit of the option register, OPTION, at the address 0x0039 are the CR₁ and CR₀ bits. If NOCOP resets (0) and CR₁-CR₀ = 0-0, the watchdog timer time out occurs after every 2¹⁶ pulses. As T = 0.5 μs for the processor when the E clock output is 2 MHz, the WDT time-out occurs at every 16.384 ms (2¹⁶ × 0.5 μs) unless the user software stores at desired intervals before a time out, first the 0x55 and then the 0xAA at the computer reset control register COPRST. This means user program resets the watchlog timer by itself after finishing the watched section of the program. [After 2¹⁵ pulses if CR₁-CR₀ = 0-1, 2¹⁴ pulses for 1-0, 2¹³ pulses for 1-1].

A watchdog timer has a number of applications and is a timing device such that it is set for a preset time interval and an event must occur during that interval else the device will generate a timeout signal and interrupt for the failure to get that event in the watched time interval.

3.8 REAL TIME CLOCK

Real time clock (RTC) is a clock that causes occurrences of regular interval interrupts on its each tick (timeout). An interrupt service routine executes on each timeout (overflow) of this clock. This timing device once started never resets or is never reloaded with another value. Once it is set, it is not modified later. The RTC is used in a system to save the current time and date. The RTC is also used in a system to initiate return of control to the system (OS) after the preset system clock periods.

Example 3.9

(i) Assume that a hardware timer of an RTC for calendar is programmed to interrupt after every 5.15 ms. Assume that at each tick (interrupt) a service routine runs and updates a memory location. Within one day (86400 s) there will be 2²⁴ ticks, the memory location will reach 0x000000 after reaching the maximum value 0xFFFFF. Within 256 days there will be 2³² ticks, the memory location will reach 0x00000000 after reaching the maximum value 0xFFFFFFFF. Note that battery must be used to protect the memory for that long period.

(ii) Assume that an RTC has to implement using a software timer. Assume that a hardware 16-bit timer ticks from processor clock after 0.5 μs. It will overflow and execute an overflow interrupt service routine after 2¹⁵ μs = 32.768 ms. The interrupt service routine can generate a port bit output after every time it runs and can also call a software routine or sends a message for a task. If n = 30, the RTC initiated software will run every 30 × 32.768 ms, which is close to 1 s.

(iii) A real time clock timer for interrupts at regular intervals is present in a microcontroller. 68HC11 has a register called the Pulse Accumulator Control Register, PACTL and two lowest significance bits, RT₁-RT₀ (1st and 0th). PACTL is write only. If the RT₁-RT₀ pair is 00, an interrupt can occur after 2¹³ pulses of the E clock. If the E clock pulses are of 2 MHz and thus T is 0.5 μs, the interrupts from a real time clock occur after each 4.096 ms. If the RT₁-RT₀ pair is 01, an interrupt can occur after 2¹⁴ pulses of the E clock, that is, after each 8.192 ms. If the RT₁-RT₀ pair is 10, the interrupt can occur after 2¹⁵ pulses of the

E clock, that is after each 16.384 ms. If the RT₁-RT₀ pair is 11, an interrupt can occur after each 2¹⁶ pulses of the E clock, that is, after each 32.768 ms. The real time clock is based on a free running counter in 68HC11. RT₁-RT₀ bits control its rate of ticking.

The interrupts from a real time clock are disabled or enabled by I bit in clock control (CC) register. The interrupts from real time clocks are also locally masked by the 6th bit, RTI in timer interrupt mask register2, TMASK2. This bit is set to unmask and reset to mask the real time clock interrupts. If RTI and I bits permit the interrupt request for real time clock timeout then the microcontroller fetches the lower and higher bytes of the interrupt servicing routine address from the addresses 0xFFFF0 for higher byte) and 0xFFFF1 (for lower byte). This is the vector address for real time clock interrupts in 68HC11. The interrupt service routine must clear (0) the RTIF, which is interrupt flag for the real time clock interrupts. The RTIF is a bit in timer interrupt flag register2, TFLG2. The TFLAG2 is at address 0x0025. It is set by each interrupt from the real time clock interrupt and therefore it must be cleared in order to enable next interrupt before returning from the corresponding service routine and before the next real-time clock-interrupt occurs.

A real time clock (RTC) provides system clock and it has a number of applications. It is a clock that generates system interrupts at preset intervals. An interrupt service routine executes on each tick (timeout or overflow) of this clock. This timing device once started is generally never reset or never reloaded to another value.

3.9 NETWORKED EMBEDDED SYSTEMS

Each specific IO device may be connected to others using specific interfaces; for example, an IO device connects and is interfaced to an LCD controller, keyboard controller or print controller using specific interface. Bus communication simplifies the number of connections and provides a common protocol for interconnecting different or same type of IO devices.

Any device that is compatible with a system's IO bus can be added to the system (assuming an appropriate device driver program is available), and a device that is compatible with a particular IO bus can be integrated into any system that uses that type of bus. This makes systems that use IO buses very flexible, as opposed to direct interconnections between the processor and each IO device, and it allows system support to many different IO devices (depending on the needs of its users), and it also allows users to change the IO devices that are attached to system as their needs change.

The main disadvantage of an IO bus (and buses in general) is that each bus has a fixed bandwidth that must be shared by all the devices, which connect to the bus. Even worse, electrical constraints (wire length and transmission line effects) cause buses to have less bandwidth than using the same number of wires to connect just two devices. Essentially, there is a trade-off between interface simplicity and bandwidth sharing. Assume that bandwidth of a bus is 200 Mbps. If the bus communicates two devices simultaneously then it does so by 100 Mbps communication by each.

IO devices communicate with the processor through an IO bus, which is separate from the memory bus that the processor uses to communicate with the memory system. Embedded systems connected internally on the same IC or systems at very short, short and long distances, and can be networked using the followings types of IO buses, each functioning according to specific protocols.

1. Using a serial IO bus allows a computer or controller or embedded system to interface network with a wide range of IO devices without having to implement a specific interface for each IO device. When

the IO devices in the distributed embedded systems are networked at long distances of 25 cm and above, all can communicate through a common serial bus. A serial bus has very few lines. Sections 3.10.1 to 3.10.5 describe the serial bus communication protocols.

- Using a parallel IO bus allows a computer or controller or embedded system to interface with a number of internal systems at very short distances without having to implement a specific interface for each IO device. Section 3.11 describes the parallel bus communication protocols.
- Using the Internet or intranet, a computer, controller or embedded system's IO device can interface globally and can network with other systems or computers and a wide range of devices in the distributed systems. Section 3.12 describes these systems.
- Using a wireless protocol allows a handheld computer, controller or embedded system IO device to interface and network with a number of handheld system IO devices at short distances up to 100 m using a wireless personal area network (WPAN) protocol, without having to implement a specific wireless interface for each IO device. Section 3.13 describes wireless bus communication protocols.

Embedded systems are distributed and networked using a serial or parallel bus or wireless protocol software and appropriate hardware.

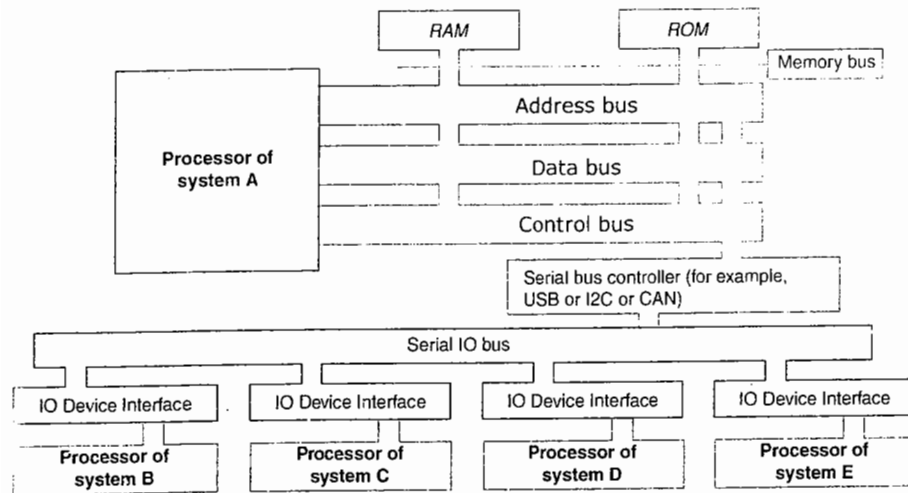


Fig. 3.9 A processor of embedded system connected to system memory bus and networked to other systems through a serial bus

3.10 SERIAL BUS COMMUNICATION PROTOCOLS

Figure 3.9 shows a processor of embedded system connected to system memory bus and networked to other systems through a serial bus. Sections 3.10.1 to 3.10.5 describe popular serial buses.

3.10.1 I²C Bus

Assume that there are number of device circuits in a number of processes in a plant, one IC each for measuring temperatures and pressures. These ICs mutually network through a common synchronous serial bus. I²C (Inter IC connect) bus is a popular bus for these circuits. There are three I²C bus standards: Industrial 100 kbps I²C, 100 kbps SM I²C, and 400 kbps I²C. The I²C was originally developed at Philips Semiconductors.

The I²C Bus has two lines that carry its signals—one line is for clock and one is for bidirectional data. There is a protocol for I²C bus. Figure 3.10(a) shows the signals during a transfer of a byte when using I²C bus.

Each device has an address using which the data transfers take place. The master can address 127 other slaves at an instance. It has a processing element functioning as a bus controller or a microcontroller with I²C bus interface circuit. Each slave can also optionally have an I²C bus controller and processing element. A number of masters can also connect to the bus. However, at any instance, there can be only one master, which is one that initiates a data transfer on SDA (serial data) line and which transmits the SCL (serial clock) pulses. From the *master* or *slave*, a data frame has fields beginning from start bit as per Table 3.9. Figure 3.10(b) shows the format of the bits at the I²C bus.

Table 3.9

Field and its length	Explanation
First field of 1-bit	It is start bit similar to the one in a UART.
Second field of 7 bits	It is called the address field. It defines the slave address being sent the data frame (of many bytes) by the master.
Third field of 1 control bit	It defines whether a read or write cycle is in progress.
Fourth field of 1 control bit	Next bit defines whether the present data is an acknowledgement (from the slave)
Fifth field of 8 bits	It is used for IC device data bits.
Sixth field of 1-bit	It is a negative acknowledgement bit (NACK) from the master. If active, then acknowledgement after a transfer is not needed from the slave, else acknowledgement is expected from the slave.
Seventh field of 1-bit	It is a stop bit like in a UART.

The disadvantage of this bus is the time taken by algorithm in the master hardware that analyses the bits through I²C in case the slave hardware does not provide for the hardware that supports it. Some ICs support the protocol and some do not. In that case, interface circuits for those ICs are also required. Also, there are open collector drivers at the master. Therefore, a pull-up resistance of 2.2 K or an active circuit for pull up of line to logic 1 for on each line is essential.

I²C is a serial bus for interconnecting ICs. It has a start bit and a stop bit like in a UART. It has seven fields for the start, 7-bit address, defining a read or write, defining a byte as an acknowledging byte, data byte, NACK and end.

3.10.2 CAN Bus

Number of devices and controllers are located and are distributed in a car. An automobile uses number of distributed embedded controllers, including those for the brakes, engine, electric power, lamps, inside temperature control, air-conditioning, gate, front dash board display, meter display panel and cruising control. Embedded controllers must network through a bus. CAN (controller area network) bus is a standard bus in distributed network. It is mainly used in automotive electronics. It is also used medical electronics and industrial plant controllers.

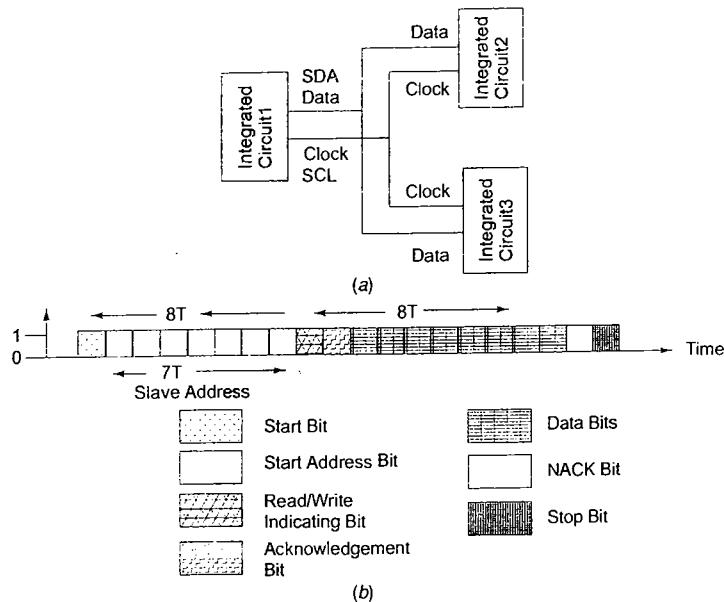


Fig. 3.10 (a) Signals during a transfer of a byte when using the I²C (Inter Integrated Circuit) bus
(b) Format of SDA bits at the I²C bus

The CAN Bus [Figure 3.11(a)] network has a serial line, which is bi-directional. CAN bus has multimaster and multicast features. A CAN device using CAN controller receives or sends a bit at any instance by operating at the maximum rate of 1 Mbps (bit-period = 1 μ s). It employs a twisted pair connection of 120 ohm line impedance at each controller node. The pair can run up to a maximum length of 40 m.

1. CAN serial line is pulled to logic level 1 by a resistor (active or passive) between the line and +4.5 V to +12.3 V. Line is at logic 1 in its idle state, also called the recessive state.
2. Each node has a buffer-gate between an input pin and a CAN serial line. A node gets the input at any instance from the line after sensing that instant when the line is pulled down to 0. The latter is called dominant state.
3. Each node has a current driver circuit between output pin and serial line. The node sends a bit to line by pulling the line 0 by its driver for a bit period. An NPN transistor is used current-driving transistor, the emitter of which also connects to the line ground and collector connects to the line. Using a driver (consisting of a buffer inverter gate connected to base of the NPN transistor), the node can pull the line to 0, which is otherwise at logic 1 in its idle state. This lets other nodes sense the input.
4. A node sends the data bits as a data frame. Data frames always start with 1 and always end with seven 0s. Between two data frames, there are minimum three fields. Table 3.10 gives the details of each field in a CAN frame. Figure 3.11(b) shows the format of the bits in a CAN frame.
5. The CAN-bus line usually interconnects to a CAN controller between the line and host node. [A host node is one that has controller for use as bus master.] The line gives input and gets output during reception and transmission using physical and data link layers at host node. The CAN controller has a BIU (bus interface unit consisting of buffer and driver), protocol controller, status-control registers, receiver-buffer and message objects. These units connect the host node through host interface circuit.

6. There is an arbitration method called CSMA/AMP (Carrier Sense Multiple Access with Arbitration on Message Priority). A node stops transmitting on sensing a dominant bit, which indicates that another node is transmitting.

Table 3.10 Each field in a CAN frame

Field and its length	Function
First field of 12 bits	This is arbitration field, which contains the packet's 11-bit destination address and RTR bit (Packet means a set of bits sent on the bus). RTR stands for 'Remote Transmission Request'. The receiving addressed device is at destination address specified in 11-bit subfield and RTR is defined on the basis of whether the data byte being sent is a data for the device or a request to the device. 11-bit address identifies the device to which data is being sent or the request being made. When an RTR bit is at 1, it means this packet is for the device at destination address. If this bit is at 0 (dominant state) it means this packet is a request for the data from the device.
Second field of 6 bits	It is control field. The first bit is identifier extension. The second bit is always 1. The last 4 bits are code for data length.
Third field of 0 to 64 bits	Its length depends on the data length code in control field.
Fourth field (third if data field has no bit present) is of 16 bits	It is CRC (Cyclic Redundancy Check) field with 15-bit CRC plus 1-bit delimiter bit. The receiver node uses it to detect errors, if any, during the transmission.
Fifth field of 2 bits	First bit is 'ACK slot'. The sender sends it as 1 and the receiver, which would send back 0 in this slot when it detects error in reception. The sender, after sensing 0 in the ACK slot, retransmits the data frame. The second bit is the 'ACK delimiter' bit. It signals the end of ACK field. If the transmitting node does not receive any acknowledgement of data frame within a specified time slot, it should retransmit.
Sixth field of 7 bits	This is the end-of-the-frame specification and has seven 0s.

CAN is a serial bus for interconnecting a central control network. It is widely used in automobiles. It has fields for bus arbitration bits, control bits for address and data length, data bits, CRC check bits, acknowledgement bits and ending bits.

3.10.3 USB Bus

Universal Serial Bus (USB) is a bus between host system and number of interconnected peripheral devices. A maximum 127 devices can connect to a host. It provides a fast (up to 12 Mbps) and as well as a low speed (up to 1.5 Mbps) serial transmission and reception between host and serial devices. A USB host, which includes controller for function as bus master can connect flash memory cards, pen-like memory devices, digital camera, printer, mice, PocketPC and video games. There are three standards: USB 1.1 (a low speed 1.5 Mbps 3 m channel along with a high speed 12 Mbps, 25 m channel); USB 2.0 (high speed 480 Mbps 25 meter channel), and wireless USB (high speed 480 Mbps 3 m).

USB protocol has this feature—a USB device can be hot plugged (attached), configured and used, reset, reconfigured and used; it can share bandwidth with other devices, detached (while others are in operation)

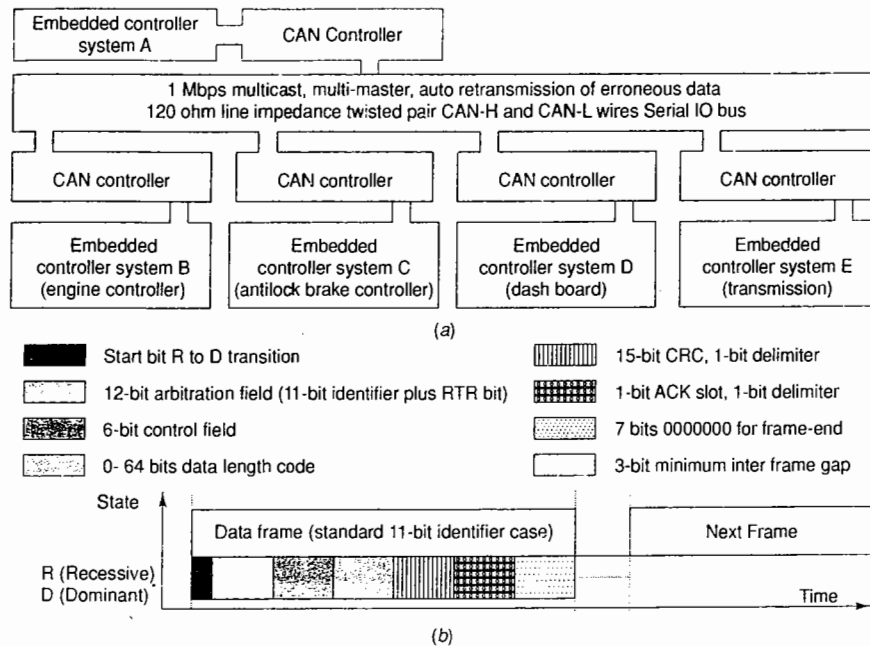


Fig. 3.11 (a) Network during a transfer of data when using the CAN (Controller Area Network) bus (b) Format of the bits at CAN bus

and reattached. Attaching and detaching can be done without rebooting. The host schedules sharing of bandwidth among the attached devices. A USB device can either be bus-powered or self-powered. In addition, there is a power management by software at host for USB ports.

USB host connects to devices or nodes using USB port-driving software and the host controller connected to a root hub. A hub is one that connects to other nodes or hubs. A tree-like topology forms as follows. The root hub connects to the hub and node at level 1. A hub at level 1 connects to the hub and node at level 2 and so on. Only the nodes are present at the last level. The root hub and each hub at a level connect in a star topology with the next level. The USB device descriptor data structure has a hierarchy, which is as follows: It has device descriptor at the root that has number of configuration descriptors and each configuration descriptor has number of interface descriptors and which has number of end point descriptors.

USB bus cable has four wires, one for +5 V, two for twisted pairs and one for ground. There are termination impedances at each end that are as per the device speed. Electromagnetic Interference (EMI)-shielded cable is used for 15 Mbps USB devices.

Serial signals are Non Return to Zero (NRZI) and the clock is encoded by inserting a synchronous code (SYNC) field before each packet. [Refer to Table 3.2]. The receiver synchronizes its bit recovery clock continuously. The data transfer is of four types: (a) Controlled data transfer (b) Bulk data transfer (c) Interrupt driven data transfer (d) Isosynchronous transfer.

USB is a polled bus. The host controller circuit regularly polls the presence of a device as scheduled by the software. It sends a token packet. The token consists of fields for type, direction, USB device address and

device end-point number. The device does the handshaking through a handshake packet, indicating successful or unsuccessful transmission. A CRC field in a data packet enables transmission error detection at the receiver.

USB supports three types of pipes—(a) 'Stream' with no USB-defined protocol. It is used when the connection is already established and the data flow starts. (b) 'Default Control' for providing access. (c) 'Message' for the control functions of the device. The host configures each pipe for the followings: (a) data bandwidth to be used, (b) transfer service type and (c) buffer sizes.

Wireless USB is wireless extension of USB 2.0 and it operates at UWB (ultra wide band) 3.1 to 10.6 GHz frequencies. It is used for short-range personal area network (high speed 480 Mbps, 3 m or 110 Mbps, 10 m channel). FCC has recommended a host wire adapter (HWA) and a device wire adapter (DWA), which provide wireless USB solutions. Wireless USB also supports dual-role devices (DRDs). A device can be a USB device as well as a limited capability host. For example, a wireless USB digital camera uses a USB host when connected to a printer and a USB device when connected to a personal computer. A wireless USB device is used to provide Internet connectivity between laptop or computer and mobile service provider network.

USB is a serial bus that interconnects a system. It attaches and detaches a device from the network. It uses a root hub. Nodes containing the devices can be organized like a tree structure. It is mostly used in networking the IO devices like scanner in a computer system. Wireless USB is used for remote connections without wires.

3.10.4 FireWire — IEEE 1394 Bus Standard

Digital video cameras, digital camcorders, digital video disk (DVD), set-top boxes, and music systems multimedia peripherals, latest hard disk drives, and printers need a high-speed bus standard interface for communicating directly to a personal computer. FireWire (IEEE 1394b) is a standard for 800 Mbps serial isosynchronous data transfers.

A FireWire IEEE 1394 port can operate at up to 400 Mbps and the latest machines include FireWire ports that support IEEE 1394b which operate at up to 800 Mbps. Since FireWire can transfer data at a guaranteed rate, it is also used in real time devices, such as video device data transfers.

A single 1394 port can interface up to 63 external FireWire devices. It supports both plug and play and hot plugging. It also provides self-powered and bus-powered support on the bus.

FireWire is a high speed 800 Mbps serial bus for interconnecting a system with multimedia streaming devices and systems.

3.10.5 Advanced Serial High Speed Buses

Section 3.2.6 described SDIO, which is an advanced high-speed serial bus for handheld devices. An embedded system may need to connect multi gigabits per second (Gbps) transceiver (transmit and receive) serial interfaces. Exemplary products are wireless LAN, Gigabit Ethernet, SONET (OC-48, OC-192, OC-768). The following are examples of the advanced bus protocols.

1. IEEE 802.3-2000 [1 Gbps bandwidth Gigabit Ethernet MAC (Media Access Control)] for 125 MHz performance
2. IEEE P802.3oe draft 4.1 [10 Gbps Ethernet MAC] for 156.25 MHz dual direction performance]
3. IEEE P802.3oe draft 4.1 [12.5 Gbps Ethernet MAC] for four channel 3.125 Gbps per channel transceiver performance]
4. XAUI (10 Gigabit Attachment Unit)
5. XSBI (10 Gigabit Serial Bus Interchange)
6. SONET OC-48
7. SONET OC-192
8. SONET OC-768
9. ATM OC-12/46/192

3.11 PARALLEL BUS DEVICE PROTOCOLS—PARALLEL COMMUNICATION NETWORK USING ISA, PCI, PCI-X AND ADVANCED BUSES

A computer system connects at high speed to other subsystems having a range of IO devices at very short distances (<25 cm) using a parallel bus without having to implement a specific interface for each IO device. When the IO devices in the distributed embedded subsystems are networked, all can communicate through a common parallel bus. A parallel bus has a large number of lines as per the protocol. Figures 3.12(a) and (b) show the processor of an embedded system A connected to system memory bus and networked to other subsystems through a parallel bus PCI using PCI bridge and AMBA-APB bridge, respectively.

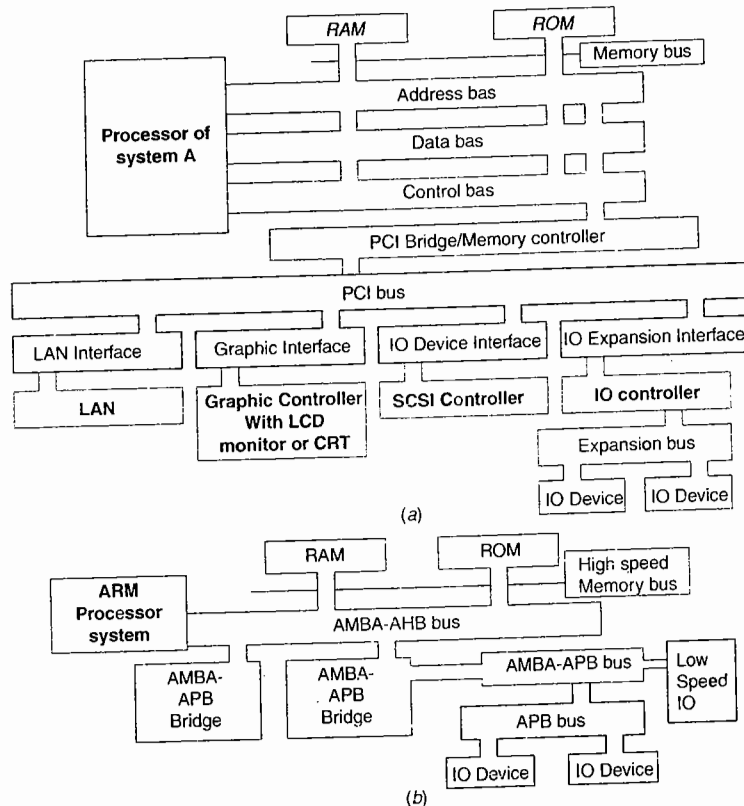


Fig. 3.12 (a) and (b) A processor of embedded system connected to system memory bus and networked to other subsystems through a parallel bus using PCI and AMBA-APB bridges

We need an interconnection bus within PC or embedded system to a number of PC-based IO cards, systems and devices. This bus needs to be separated from system-bus that connects the processor to memories. The

system bus and interconnection bus operate at different levels of speeds. Exemplary devices are display monitor, printer, character devices, network subsystems, video card, modem card, hard disk controller, thin client, digital video capture card, streaming displays, 10/100 Base T card and card using DEC 21040 PCI Ethernet LAN controller. Each of these devices, which performs a specific function, may contain a processor and drives by software. Each device has specific memory address-range, specific interrupt-vectors (pre-assigned or auto configured) and device IO port addresses. A bus of appropriate specifications and protocol interfaces these to host system or computer.

A switch, popularly called PCI bus interface, switches a processor communication with the memory bus to PCI bus. In most systems, the processor has a single data bus that connects to a switch module such as the PCI bridge found in many PC systems, although some processors integrate the switch module onto the same integrated circuit as the processor to reduce the number of chips required to build a system thereby reducing the system cost. The switch communicates with the memory through *memory bus* and dedicated set of wires that transfer data between these systems. A separate *IO bus* connects the switch to IO devices. Separate memory and IO buses are used because the IO system is generally designed for maximum flexibility, to allow as many different IO devices as possible to interface to the computer, while the memory bus is designed to provide the maximum possible bandwidth between the processor and memory system.

Two old interconnection buses for communication between the host and a device are ISA and EISA (Extended ISA). A new interconnection for the bus is either PCI or PCI/X. [A variant of it is Compact PCI (cPCI).] Sections 3.12.1 to 3.12.4 describe three parallel bus communication protocols.

Parallel bus interconnects IO devices and peripherals over very short distances and at high speed. ISA, PCI and ARM buses are examples of parallel buses. A parallel bus interfaces the system memory bus through a bridge or switching circuit.

3.11.1 ISA Bus

ISA bus (used in IBM Standard Architecture) connects only to an embedded device that has an 8086 or 80186 or 80286 processor, and in which the processor addressing and IBM PC architecture addressing limitations and interrupt vector address assignments are taken into account. There is no geographical addressing.

The limitation for memory access by a system using the ISA bus of the original IBM PC were as follows: ISA bus memory accesses can be in two ranges, 640 to 1 MB and 15 to 16 MB. The former range also overlaps with the range used by video boards and BIOS. [Note: Linux OS does not support the second range for accessing directly a device.]

The IO port address limitations for devices are as follows: The 8086 to 80286 processor has IO mapped IOs, not memory mapped IOs. Though the instruction set provides for IO instructions for 64 kB IO addresses, the IBM PC configuration ignores the address lines A_{10} to A_{15} and these are not decoded. Therefore, only 1024 IO port addresses are available. A hexadecimal addressing scheme with three nibble addressing between 000 to 3FF only can be used for a device. The A_{10} to A_{15} bits are thus immaterial. The following are the addresses allocated in IBM Standard Architecture (ISA).

1. Addresses allocated are 0x000–0x00F for DMA chip 8237. The addresses for other devices are as follows.
2. 0x020–0x021 addresses allocated are for programmable interrupt controller 8255. Hex 0x040–0x043 for timer 8253.
3. 0x060–0x063 for parallel port programmable parallel interface.
4. The hexa-decimal addresses 080-083, 0A0-0AF, 0C0-0CF, 0E0-0EF allocated are for components on the motherboard.

5. Reserved addresses from peripherals are hex 220-24F, 278-27F, 2F0-2F7, 3C0-3CF and 3E0 to 3F0.
6. The addresses allocated are hex 2F8-2FF and 3F8-3FF for IBM COM ports.
7. Addresses are hex. 320-32F and 3F0-3F7 for hard disk and floppy diskette, respectively.
8. Only 32 addresses between 0x300 to 0x31F are available for prototype card; for example, ADC card.
9. Addresses allocated are between hex 380-389 and 3A0-3A9 for synchronous communication.
10. Synchronous Data Link Control (SDLC) addresses allocated are between hex. 380-38C.
11. Display monitor ports are within 380-38F (monochrome) and 3D0-3DF for (colour and graphics).

There is a limited availability of interrupt vectors in the IBM PC 80x86 family. Only 256 vectors are available. Interrupt service functions are now shared at software level: for example, SWT interrupts. Original ISA specifications did not allow that.

EISA bus is a 32-bit data and address-lines version of ISA, and devices (system using this bus for IOs) are also supported. An EISA device driver first checks the EISA bus availability on the hosting computer or system. It supports the sharing of interrupt functions, SCI (Serial Communication Interface) controller and Ethernet devices. Unix and Linux support the EISA bus-driven cards and devices.

ISA and EISA buses are compatible with IBM architecture. They are used for connecting devices following IO addresses and interrupt vectors as per IBM PC architecture. EISA is 32-bit extension of ISA. It also supports software interrupt functions and Ethernet devices.

3.11.2 PCI and PCI/X Buses

Recently, the most used synchronous parallel bus in the computer system for interfacing PC-based devices is PCI (Peripheral Component Interconnect). PCI provides a superior throughput than EISA. It is almost platform-independent, unlike the ISA, which depended on the IBM PC platform, interrupt vectors, IO addresses and memory allocations. Its clock rate is nearest to the submultiple of the system clock. PCI provides three types of synchronous parallel interfaces. Its versions are 32/33 MHz, 64/66 MHz, PCI-X 64/100 MHz, PCI Super V2.3 264/528 MBps 3.3 V (on a 64-bit bus), 132/264 (on a 32-bit bus) and PCI-X Super V1.01a for 800 MBps 64-bit bus 3.3 V.

PCI bus has 32-bit data bus extendible to 64 bits. In addition, it has 32-bit addresses extendible to 64 bits. Its protocol specifies the interaction between the different components of a computer. A specification is version 2.1. Its synchronous/asynchronous throughput is up to 132/528 MB/s [33 M x 4/66 M x 8 Byte/s]. It operates on 3.3 V signals. A typical application is an exemplary PCI Card has a 16 MB Flash ROM with a router gateway for a LAN.

A PCI driver can access hardware automatically as well as by addresses assigned by the programmer. The PCI feature of automatically detecting the interfacing systems and assigning new addresses is important for coding a device driver. The PCI bus therefore simplifies the addition and deletion (attachment and detachment) of the system peripherals. A manufacturer registers a global number for PCI device or card, just as, 68HC11 or 80386 are globally registered numbers. A 16-bit register in PCI device identifies this number to let that device auto-detected. Another 16-bit register is for a device ID number. These two numbers allow the device to carry out auto-detection by its host computer. Each device may use FIFO controller with FIFO buffer for maximum throughput.

A device or host identifies its address space by three identification numbers (i) IO port, (ii) memory locations and (iii) configuration registers of total 256 B with a 4-byte unique ID. Each PCI device has address space allocation of 256 bytes to access it by the host computer. The unique feature of PCI bus is its configuration address space. A uniquely assigned interrupt type (a number) handles an interrupt. For example, interrupt type 3 has the interrupt vector address 0x0000C and four bytes at the address specify the interrupt service

routine address. Interrupt type can be between 0x00 and 0xFF. A configuration register number 60 stores the one byte for the interrupt type $n(\text{pci})$. The PCI device or host when interrupted handles the interrupt of type $n(\text{pci})$. Figure 3.13 shows 64-byte standard configuration registers in a PCI device. Following are the abbreviations used in the figure.

VID: Vendor ID. *DID*: Device ID. *RID*: Revision ID. *CR*: Common Register. *CC*: Class Code. *SR*: Status Register. *CL*: Cache Line. *LT*: Latency Timer. *BIST*: Base Input Tick. *HT*: Header Type. *BA*: Base Address. *CBCISB*: Card Base CIS Pointer. *SS*: Sub System. *ExpROM*: Expansion ROM. *MIN_GNT*: Minimum Guaranteed time. *MAX_GNT*: Maximum Guaranteed Time.

VID, *DID*, *RID*, *CR*, *SR*, and *HT* are compulsorily configured. The rest are optional.

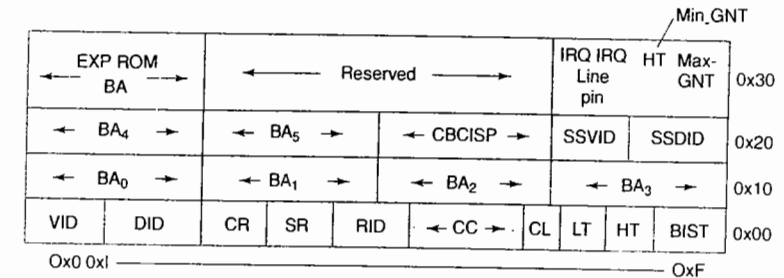


Fig. 3.13 64 bytes at standard device independent configuration registers in a PCI device or host

A PCI controller must access one device at a time. Thus, all the devices within host computer can share IO port addresses and memory locations but cannot share the configuration registers. That means that a device cannot modify other configuration registers but can access other device resources or share the work or assist the other device. If there are reasons for it doing so, a PCI driver can change the default bootup assignments on configuration transactions.

A device can initialize at booting time. This helps in avoiding any address collision. A PCI device on bootup disables its interrupt. Its address space is inaccessible and only the configuration registers space remains accessible. PCI BIOS with the device performs the configuration transactions and then memory and address spaces automatically map to the address space in host computer.

PCI parallel bus is popular in distributed embedded devices. PCI and PCI/X buses are used for parallel bus communication and these are independent from the IBM architecture. PCI/X is an extension of PCI and supports 64/100 MHz transfers. PCI bus new version support 132/528 MB/s data transfer with synchronous/asynchronous throughputs.

3.11.3 ARM Bus

ARM processor interfaces the memory, external DRAM (dynamic RAM controller and on-chip IO devices, which connect to 32-bit data and 32-bit address line at high speed using AMBA (ARM Main Memory Bus Architecture)-AHB (ARM High Performance Bus). Figure 3.12(b) shows AMBA-AHB and AMBA-APB bridges. The bridges interface the memory and external-chip IO devices, which operate at low speed using AMBA-APB. The maximum AHB bps bandwidth is sixteen times the ARM processor clock.

A switch, popularly called the AMBA-APB bridge, switches ARM CPU communication with the AMBA bus to APB bus. The ARM processor-based microcontroller has a single data bus in AMBA-AHB that connects to the bridge, which integrates the bridge onto the same integrated circuit as the processor to reduce the number of chips required to build a system. This reduces the system cost. The bridge communicates with the memory through an AMBA-AHB, a dedicated set of wires that transfer data between these two systems. A separate APB IO bus connects the bridge to the IO devices. Separate AMBA-AHB and APB IO buses are used because the IO system is generally designed for maximum flexibility, to allow as many different IO devices as possible to interface to the computer, while the memory bus is designed to provide the maximum possible bandwidth between the processor and the memory system.

The APB can connect the I²C, touchscreen, SDIO, MMC (multimedia card), USB, CAN and other required interfaces to an ARM microcontroller.

ARM bus is of two types: AMBA-AHB and AMBA-APB. AHB connects to high speed memory. APB connects the external peripherals to the system memory bus through a bridge.

3.11.4 Advanced Parallel High Speed Buses

Many telecommunication, computer and embedded processor-based products need parallel buses for system IOs. Three versions of PCI parallel synchronous/asynchronous buses provide system-synchronous parallel interfaces. These three versions may not have sufficiently high speed, ultra high speed and large bandwidth that are required for system IOs, routers, LANs, switches and gateways, SANs (Storage Area Networks), WANs (Wide Area Networks) and other products. These do not meet the source-synchronous parallel interfacing requirements. Bandwidth needs increase exponentially in the order of audio, graphics, video, interactive video and broadband IPv6 Internet. An embedded system may need to connect IO system using gigabit parallel synchronous interfaces. The following are advanced bus standard and proprietary protocols developed recently.

1. GMII (Gigabit Ethernet MAC Interchange Interface).
2. XGMI (10 Gigabit Ethernet MAC Interchange Interface)
3. CSIX-1. 6.6 Gbps 32-bit HSTL with 200 MHz performance.
4. RapidIO™ Interconnect Specification v1.1 at 8 Gbps with 500 MBps performance or 250 MHz dual direction registering performance using 8-bit LVDS (Low Voltage Data Bus).

3.12 INTERNET ENABLED SYSTEMS—NETWORK PROTOCOLS

Figure 3.14 shows an Internet-enabled embedded system communicating to other systems on the Internet. Internet-enabled embedded systems use html or MIME type files (Section 3.12.1), TCP (Section 3.12.2) or UDP (Section 3.12.3) transport layer protocol, and are addressed by an IP address (Section 3.12.4) and use IP protocol at network layer. An IP address is of 32 bits (four decimal numbers separated by dots in between) or 48 bits in IPv4 or IPv6 respectively. IPv4 means IP protocol version 4 and IPv6 means version 6. A system at one IP address 1 communicates with another system at another IP address 2 or 3 or... using the physical connections on the Internet and the routers. Since the Internet is a global network, the system connects to remotely located as well as short range located system. Network connectivity is through the layers. Each layer has a protocol, which specifies the way in which the data or message from the previous layer transfers to the next layer.

There are five layers in a TCP/IP network. They are the application, transport, network, data-link and physical layers. The TCP/IP application layer protocol also specifies presentation ways. Transport layer protocol specifies session establishment and termination ways also.

Sections 3.12.1 to 3.12.5 describe the TCP/IP suite's five most used protocols.

Embedded systems are Internet enabled by using TCP/IP protocol suite protocols for networking to Internet and assigning the IP addresses to each system.

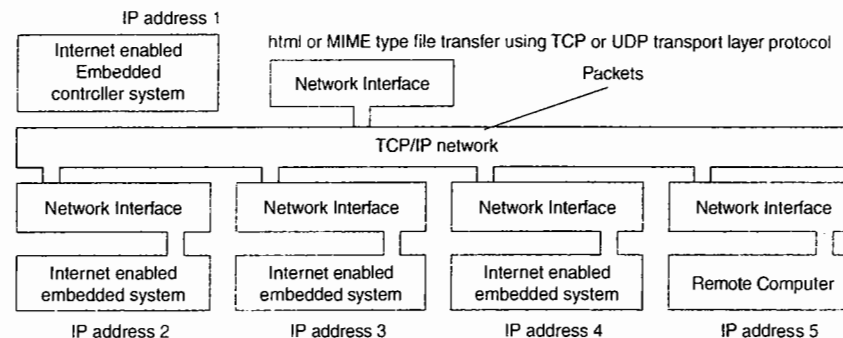


Fig. 3.14 An Internet enabled embedded system communication connected on the Internet

3.12.1 Hyper-Text Transfer Protocol (HTTP)

An application layer protocol is as per the application. This layer accepts the data, for example, in HTML or text format and puts the header words as per the protocol and sends the application layer header plus data to the transport layer. A port number specifies the application in the header. Following are the important application layer protocols that support TCP/IP networking.

1. NTP (Network Time Protocol) synchronizes system clocks on a network.
2. MIME enables attachment of multiple types of files. The examples are:
 - txt (text file),
 - doc (MSOFFICE Word document file),
 - gif (graphic image format file),
 - jpg (jpg format image file), and
 - wav voice or music file.
3. HTTP (Port 80) enables Internet connectivity by Hyper-Text Transfer Protocol (HTTP).
4. FTP (Port 21 for control, 20 for data) enables file transfer connectivity by File Transfer Protocol. TFTP (Port 69) for Trivial FTP. NFS (Network File System) is used for sharing files on a network.
5. TELNET (Port 23) enables remote login to remote terminals by Terminal Access Protocol.
6. SMTP (Port 25) enables e-mail transfer, store and forward by Simple Mail Transfer Protocol.
7. PoP3 (Port 110) enables e-mail retrieval.
8. NNTP (Port 119) (Network News Transfer Protocol) is used for news transfer port.
9. DNS (Port 53) for Domain Name Service.
10. SNMP (Port 161) Simple Network Management Protocol.
11. Bootps and Bootpc (Ports 67 and 68) for Bootstrap Protocol (DHCP) Server and Client, respectively.
12. DHCP (Dynamic Host Configuration Protocol) used for remote booting as well as for configuring a system.

A port-assigned number supports multiple logical connections using a socket. Each socket has IP address and port number. A registered port number can be between 0 and 1023. Registration is done by IANA (Internet

Assigned Number Authority. Port number 0 means host itself. A user unregistered server can have Port number above 5000.

HTTP port 80 is an application layer protocol. The HTTP features are as follows:

1. HTTP is standard protocol for requesting for a URL (universal resource locator) address, for example, <http://www.mcgraw-hill.com>. An URL defines a web page resource, and is used for retrieving or sending web page file. The response from web may be with or without applying a process. An HTTP client requests an HTTP server on the Internet and the server responds by sending a response.
2. HTTP is a stateless protocol. For an HTTP request, the protocol assumes a fresh request. It means there is no session or sequence number field or no field that is retained in the next exchange. This makes a current exchange by an HTTP request independent of the previous exchanges. The later exchanges do not depend on the current one. An e-commerce-like application needs a state management mechanism. A Cookie is a text file created during a particular pair of exchanges of HTTP request and response. The creation is either at CGI or processing program or script or at client (Browser). A prior exchange may then depend on this cookie. By this mechanism, the stateless feature of HTTP is compensated. The cookie provides a HTTP state management mechanism.
3. HTTP is a file transfer-like protocol for HTML (hyper text markup language) files. This makes it easy to explore a web site URL. A request (from a client) is sent and reply (response from a server) is received.
4. The HTTP protocol is very light (a small format) and thus speedy as compared to other existing protocols. HTTP is able to transfer any type of data to a browser (a client) provided it is capable of handling that data.
5. Besides simplicity, another important feature of HTTP is its flexibility. Assume we are surfing the web and our connection breaks (or user does so); then too we can start surfing on the Net from just that point again. Being a stateless protocol, HTTP does not keep track of the state as FTP does. Each time a connection establishes between the web server and the client (browser), both these interpret this connection as a new connection. Simplicity is a must because a web page has its URL resources distributed over a number of servers.
6. HTTP protocol is based on the Object Oriented Programming System (OOPS). Methods are applied to objects identified by URL. It means that as in the normal case of Object Oriented Program, the various methods apply on an object.
7. From HTTP 1.0 and 1.1 version onwards, the following features have been included:
 - (a) Multimedia file access is feasible due to provision of the MIME (Multipurpose Internet Mail Extension) type file definition.
 - (b) From HTTP 1.1 version onwards, there are eight specified methods and extension methods. An extension method is method added for a specific HTTP. There can be none or one or several extension methods. The HTTP specific methods are as follows. 1. GET 2. POST. 3. HEAD 4. CONNECT. 5. PUT 6. DELETE 7. TRACE. 8. OPTIONS. (Last four from 1.1). In earlier versions, GET follows a space and then document name. Server returns the documents and closes the connection. From 1.1, the POST method has permitted form processing, as using it the client transmits the form data or other information to the server. From 1.1, the server does not close the connection after response and thus response can be processed before it is sent.
 - (c) A provision of user authentication exists besides the basic authentication introduced from HTTP 1.1 version onwards, Digest Access Authentication prevents the transmission of username and password as HTML or text.
 - (d) A host header field adds to support those ports and virtual hosts that do not accept or send IP packets. From HTTP 1.1 version onwards, An error report to client when a HTTP request is without a host header field.

- (e) From HTTP 1.1 version onwards, an absolute URL is acceptable to the server. Earlier only proxy server accepted that.
- (f) Status codes in the response.
- (g) Caching of a resource is provided at server (and proxy).
- (h) Byte range specification helps in large response in parts.
- (i) Selection among various characteristics on retrieval by the client is feasible when a server sends response to client request. For example, two characteristics, *language* and *encoding* can be specified in server environment variables while the client sends request header for retrieving a resource. The resource then retrieves in that *language* and with that *encoding*. The contents sent to client do not change, only the way in which these are presented to the client change.
- (j) Length specification helps in presentation in chunks.
8. An HTTP message header during a request from a client or during a response from server consists of two parts (a) A start-line, none or several message-headers (fields) and empty line, and (b) Body of message. HTTP specifies that request message to consist of request message headers. HTTP also specifies that response message to consist of response message headers.
9. HTTP provides for entity headers. These contain information about entity body contained in the message, or in case body is not present then information about the entity, not its body. For example, information of content length in bytes.
10. HTTP interaction scheme is that a client requests server directly or through proxy or a gateway. An HTTP message is therefore either *request* or *response*. The format of the messages called RFC 822, specifies ways of sending text messages on the Internet. The message during request from client or during response from server consists of two parts. (a) Start-line, none or several message-headers (fields) plus empty line, and (b) Body of the message. The start line is either a 'request-line' or 'status-line' for request-or response-message, respectively.

3.12.2 Transport Control Protocol (TCP)

TCP (Transport Control Protocol) is a protocol used in transport layer. This layer accepts messages from the upper layer on transmission by application or session layer. This layer also accepts a data stream from the network layer at receiving end. Before communicating a message to the next network layer, it may add a *header*. The message may communicate in parts or segments or fragments. The header generally has the additional bits for source and destination addresses. Also, there are bits in it for the sequence and acknowledge management, flow and error controls, etc.

There are bits for the offset, window, flags, checksum, urgent pointer, option and for padding also. TCP supports the point to point networking mode.

TCP specifies a format of byte streams at the transport layer of the TCP/IP suite. TCP is used for a full duplex acknowledged flow. Its format has a TCP header of five plus $(n-5)$ words for options and padding and data of maximum l words. Then, $l \geq 2^{14} - n$. Here $n \geq 5$. n equals the number of words in the header and is called data offset, which means the number of words after which data bits start in the stream. If $n > 5$, it means there exists words for options and padding. Padding refers to bits used for filling the remaining part of the available field. For example, the option field may indicate the application to be run by the destined node. An acknowledged flow means that the messages communicate in a point-to-point network mode and that there is an acknowledgment for first establishing a connection. Full duplex means that at a given instance, messages go to and fro from sender to receiver, and that the receiver acknowledges receipt. A request and its response do not form a separate transmission. TCP is virtual-connection oriented. It does not permit multicasting but point-to-point virtual connection.

3.12.3 User Datagram Protocol (UDP)

TCP/IP also supports at the transport layer a simpler protocol than TCP. When a message is connectionless and stateless, then the transport layer protocol in the TCP/IP suite is User Datagram Protocol (UDP). UDP supports the broadcast networking mode. An example is application for communicating header before a data stream. The header specifies the bits for source and destination ports, total length of message including header and check sum (optional). During reception, this message to upper layer flows after deleting the header bits from the received transport layer header. Header bits add at the transmitting time in the application or session layer bytes.

3.12.4 Internet Protocol (IP)

All Internet enabled devices communicate using Internet protocol (IP). The transport layer data transmits on the network, divides into the packets at the network layer. Each packet transmits through a chain of routers on the Internet. A packet is minimum unit of data that transmits on the Internet through routers. Several packets forming a source can reach a destination using different routes and can have different delays. The packet consists of IP header plus data or IP header plus routing protocol along with the routing messages. The packet has a maximum of 2^{16} bytes (2^{14} words, 1 word = 32 bits = 4 bytes).

Network routing is as per standard IPv4 (version 4) or IPv6 (version 6). IPv6 is a broadband protocol. Table 3.11 lists the fields in IPv4 protocol header.

Table 3.11 Various fields at IPv4 header for routing the packets through routers to destination node

Field at the IP header	Explanation
<i>Version</i>	IP version bits are 0100 for IPv4 (presently in wide use) and 0110 for IPv6 (IPng IP next generation for broadband Internet).
<i>Precedance</i>	Precedence type is between 8 th to 10 th bit. Bits 111 specifies highest precedence. For example, for streaming audio or video, 000 specifies common data.
<i>Service</i>	Service type is between 11 th to 15 th bit.
<i>QoS (Quality of Service)</i>	Bits are for QoS (Quality of Service) specification in terms of security, speed, delays and cost desired or must be achieved.
<i>Fragment ID</i>	Each message may have many packets fragmented through the routers. Each fragment must thus provide a unique ID for identification for re-assembly at the receiver end.
<i>Flags</i>	Flags indicate whether present fragment is last one, whether fragments are permitted and whether more fragments will follow. Let q = Number of header words (1 word = 32-bit). Flag bit 1 indicates whether more fragments of the packet will succeed this fragment. Flag bit 2 identifies it as a test fragment or not. Flag bit 3 checks whether the fragmentation is permitted or not.
<i>Checksum</i>	Header checksum checks errors in header transmission.
<i>Time-to-live</i>	Time to live indicates number of retransmission hops permitted in case of failed delivery.
<i>Protocol type</i>	Type indicates whether the packet is transmitting a UDP byte stream or a TCP stream from transport layer. The important routing protocols that encapsulate after the IP header in an IP packet are:

(Contd)

Field at the IP header	Explanation
	IGMP (Internet Group Management Protocol). IGMP is a protocol to manage data transmission between select host groups. Several hosts join a group. Group multicasts use routers that uses the IGMP.
	ICMP (Internet Control Management Protocol). ICMP is a protocol to control routing between networked hosts.
	ICMP data byte stream is inside an IP packet (datagram). Its format is as follows. First 20 bytes minimum are the IP header. Next follows the fields of 1-byte each for Type and Code, successively. Next two bytes are for Checksum. Then follows the ICMP messages the format and length of which is variable.
	Interior routing protocols, for example, the RIP (Routing Information Protocol) and OSPF (Open Shortest Path First) protocol.
	Inter-Domain (exterior routing) protocols, for example, EGP (Exterior Gateway Protocol), BGP (Border Gateway Protocol) and GGP (Gateway to Gateway Protocol).
<i>Header length (data-offset)</i>	The IP header length (data offset) $p \leq 2^{14} - q$. Here $q \geq 5$. q equals the number of words in the header and is called the data offset [Number of words after which data bits start in the stream.]
<i>Source and Destination addresses options</i>	IP Source and destination IP addresses If $q > 5$, there exist words for options and padding. Padding refers to bits that are used for filling the remaining part of the available field. For example, option 4 will mean put time stamp at all the stoppages of the packet during transit to destination through routers. Time stamping enables packet delay measurements to calculate Network Performance Quality.

3.12.5 Ethernet

The inventor of Ethernet LAN is Robert Metcalfe. At present, about one third of the LANs in the world are the Ethernet LANs, and in each frame, there is a header like in a packet. In Ethernet LAN standard is IEEE 802.2 (ISO 8802.2). It is a protocol for local network of computers, workstation and devices. LAN is used for sharing local computers, systems and local resources such as printers, hard disk space, software and data. Table 3.12 gives the features of the Ethernet LAN devices.

Data for transmission fragments into the frames. Each frame has a header. Firstly, the header has eight bytes, which defines a preamble. The preamble indicates the start of a frame and is used for synchronization. Then the header has six bytes of destination address. Six bytes of source address then follows the destination address. Then there are six bytes. These are for the type field. These are meaningful only for the higher network layers and the length definition. The minimum 72 bytes and maximum 1500 bytes of data follow the length definition. Lastly, there are 4 bytes for CRC check for the frame sequence check.

3.13 WIRELESS AND MOBILE SYSTEM PROTOCOLS

Figures 3.15(a), (b) and (c) show a handheld device or computer system connected to other handheld devices or computer through IrDA, Bluetooth and ZigBee wireless protocols, respectively. Sections 3.13.1 to 3.13.4 describe the IrDA, Bluetooth, WLAN (wireless LAN) 802.11 and ZigBee protocols.

Table 3.12 Ethernet LAN features

Feature	Ethernet LAN
Topology and transmission mode	Bus
Speed	10 Mbps, 100 Mbps (unshielded and shielded wires) and 4 Gbps (in twisted pair wiring mode)
Broadcast Medium	Passive. Wired connections-based. Frame format like the IEEE 802.3.
SNMP (Simple Network Management Protocol)	Yes
System	Open (therefore allows equipment of different specifications)
Operation	Each one connected to a common communication channel in the network. It listens and if the channel is idle then transmits. If not idle, waits and tries again. Multi access is like in a packet switched network
Control	Passive, connection-based
Address Resolution Protocol (ARP) for resolving 32-bits Internet protocol addresses with the 48-bit destination host media address.	Yes, There is a media access control (MAC) address for transmitting and forwarding frames on the same LAN. We can also use multicast addressing to send frames to all or few select types of Ethernet devices.
Connectivity to Internet and Intranet	Yes, Outside a LAN the Internet Protocol addresses are sent.

3.13.1 Infrared Data Association (IrDA)

Infrared (IR) is electromagnetic radiation of wavelength greater than visible red light. An infrared source consists of a gallium-arsenic-phosphorus junction-based diode. An infrared receiver consists of a gallium-arsenic-phosphorus junction-based phototransistor, which conducts electric current when the IR beam falls on it and does not conduct when IR does not fall on it. The collector or drain has voltage close to 0 V when it conducts and is close to supply voltage when it does not conduct.

IrDA (Infrared Data Association) recommends a protocol suite as standard. It supports data transfer rates of up to 4 Mbps. It supports bi-directional serial communication over viewing angle between $\pm 15^\circ$ and distance of nearly 1 m. At 5 m, the IR transfer data can be up to data transfer rates of 75 kbps. There should be no obstructions or wall in between the source and receiver.

Figure 3.15(a) shows a handheld device connected to other computer through using IrDA protocol. Protocol-processing hardware device and the protocol software embeds in the system, which support line of sight communication using infrared.

IrDA supports 5 levels of communication. Level 1 is minimum required communication. Level 2 is access-based communication. Level 3 is index-based communication. Level 4 is synchronized communication. Synchronization software, for example, ActiveSync or HotSync is used. Level 5 is SyncML (synchronization markup language)-based communication. A SyncML protocol is used for device management and synchronization with server and client devices, which are connected by IrDA.

IrDA is used in mobile phones, digital cameras, keyboard, mouse, printers to communicate to laptop computer and for data and pictures download and synchronization. IrDA is also used for control TV, air-conditioning, LCD projector, VCD devices from a distance.

IrDA supports several protocols at three layers. Lower layer is physical layer 1.0 or 1.1. It supports data transfer rates of 9.6 kbps to 115.2 kbps and 115.2 kbps to 4 Mbps in IrDA physical layer 1.0 and 1.1, respectively.

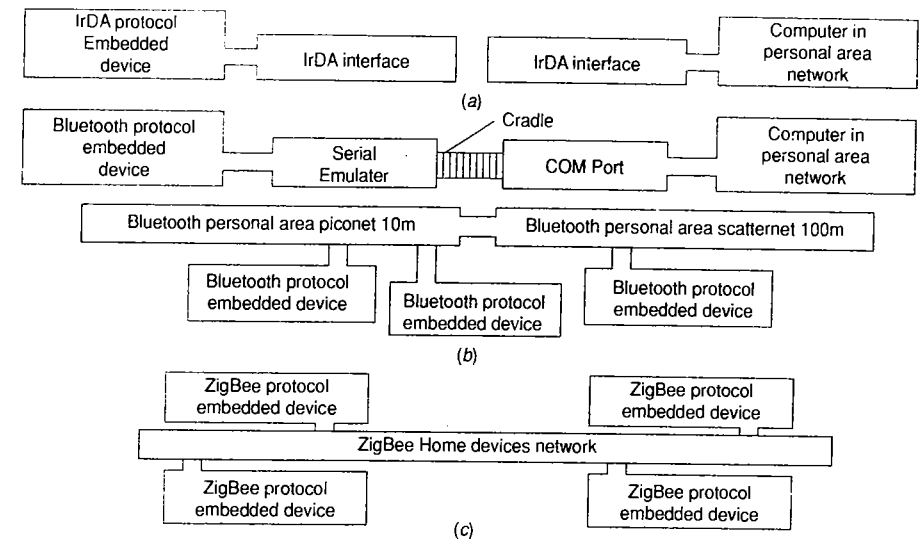


Fig. 3.15 (a), (b) and (c) A handheld device connected to other handheld devices or computer through IrDA, Bluetooth and ZigBee wireless protocols, respectively

Intermediate layer is data-link layer. At data link layer, it specifies IrLMP (IR link management protocol) upper sublayer and IrLAP (IR link access protocol) lower sublayer. An IrLAP is HDLC synchronous communication (Section 3.2.4).

An IrDA upper layer protocol is Tiny TP (transport protocol). Another upper layer protocol is IrLMIAS (IR Link Management Information Access Service Protocol). A transport layer protocol during transmission specifies ways of flow control, segmentation of data and packetization. During reception, it assembles the segments and packets. The upper layer protocol for the session layer is IrLAN, IrBus, IrMC, IrTran, IrOBEX (Object Exchange) and standard serial port emulator protocol IrCOMM (IR communication). IrBus provides serial bus access to game ports, joysticks, mice and keyboard. Application layer protocol is for security and application and as specified by the IrDA. For example, IrDA Alliance Sync protocol is used to synchronize mobile devices personal information manager (PIM) data. It supports Object Push (PIM) or Binary File Transfer.

Windows and other operating systems support IrDA protocol-based communication devices. An infrared monitor in Windows monitors the IR port of the IR device. It detects a nearby IR source. It controls, detects and selects the IR communication activity. An IR device on command sets up connection using IrDA, and starts the IR communication. When IR communication is inactive, the monitor enables plug and play (unless disabled).

IrDA protocol overhead is between 2% to 50% of Bluetooth device overhead. The communication setup latency is just few milliseconds. The requirement of line of sight and unobstructed communication is the limitation.

3.13.2 Bluetooth

Bluetooth hardware is connected to embedded system buses and Bluetooth software embeds in the system to support WPAN using Bluetooth wireless protocol. Figure 3.15(b) shows a handheld device connected to

other computers through wireless protocol using Bluetooth. A large number of CD players and mobile devices are Bluetooth-enabled. Bluetooth is also used for handsfree listening of Bluetooth-enabled iPod or CD music player or mobile phone by using Bluetooth-enabled ear buds.

Bluetooth is an IEEE standard 802.15.1 protocol. The physical layer radio communicates at carrier frequencies in 2.4 GHz band with FHSS (frequency hopping spread spectrum). Hopping interval is 625 μ s and number of hopped frequencies are 79. Data transfer is between two devices or between a device and multiple devices.

It supports range up to 10 m low power and up to 100 m high power. Range depends on radio interface at physical layer. Bluetooth 1.x data transfer rate supported is 1 Mbps. Bluetooth 2.0 has enhanced maximum data rate of 3.0 Mbps over 100 m. Bluetooth protocol supports automatic self-discovery and self-organization of network in number of devices. Bluetooth device self discovers nearby devices (<10 m) and they synchronize and form a WPAN (wireless personal area network). Bluetooth protocol supports power control so that the devices communicate at minimum required power level. This prevents drowning of signals by superimpositions of high power signals with lower level signals.

The physical layer has three sublayers: radio, baseband and link manager or host controller interface. There are two types of links: best effort traffic links and real-time voice traffic links. The real-time traffic uses reserved bandwidth. A packet is of about 350 bytes. The link manager sublayer manages the master and slave link. It specifies data encryption and device authentication handling, and formation of device pairs for Bluetooth communication. It gives specifications for state transmission-mode, supervision, power level monitoring, synchronization, and exchange of capability, packet flow latency, peak data rate, average data rate and maximum burst size parameters from lower and higher layers.

The Host Controller Interface (HCI) interface is a hardware abstraction sublayer. It is used in place of the link manager sublayer. It provides for emulation of serial port, for example, 3-wire UART emulation. A Bluetooth device can thus interface to the COM port of a computer.

Its communication latency is 3s. It has large protocol stack overhead of 250 kB. Provision of encrypted secure communication, self-discovery and self-organization and radio-based communication between tiny antennae are three main features of Bluetooth.

3.13.3 802.11

Wireless LAN uses IEEE standards 802.11a to 802.11g. Data transfer rates are 1 and 2 Mbps. The 802.11b is called wireless fidelity (WiFi). 802.11b support data rates of 5.5 Mbps by mapping 4 bits and 11 Mbps mapping 8 bits simultaneously during modulation.

A given set of the LAN-station access-points network together and the set is called extended service set (ESS). It is a backbone distribution system. A backbone set may network through the Internet. ESS supports fixed infrastructure network.

There are two types of wireless service sets.

1. One service set has one wireless station, which communicates to an access point, also called a hotspot. The service set is called basic service set (BSS). WLAN supports ad-hoc network, which, as and when a nodes come nearby in range, it forms the network. BSS supports ad-hoc network which, when nodes come nearby with in range of the access point, forms the network through ESS. A node can move from one BSS to another.
2. The other service set has several stations. It is called independent basic service set (IBSS). It has no access point. It does not connect to the distribution system. It may have multiple stations, which also cannot communicate among themselves. IBSS supports ad-hoc network.

802.11 provides specifications for physical layer and data link layers.

The data link layer specifies a MAC layer. The MAC layer uses carrier sense multiple access and collision avoidance (CSMA/CA) protocol. A station listening to the presence of the carrier during a time interval is

called distributed inter-frame spacing (DIFS) interval. If the carrier is not sensed (detected) during DIFS, then the station backs off for a random time interval to avoid collision and retries after that interval. A receiver always acknowledges within a short interframe spacing (SIFS). Acknowledgment is made after successful CRC (cyclic redundancy check). If there is no acknowledgement within SIFS, then the transmitter retransmits and upto 7 retransmission attempts are made.

There is a packet called request to send (RTS), which is first sent. If the other end responds by the packet call clear to send (CTS), then the data is transmitted. MAC layer specifies power management, handover and registration of roaming mobile node within the backbone network at a new BSS within the ESS.

There are three communication methods at the physical layer. WLAN can use FHSS or DSSS or Infrared 250 ns pulses. The physical layer has two sublayers. 802.11b has three sublayers: one is Physical Medium Dependent (PMD) protocol which specifies the modulation and coding methods; the second is the Physical Layer Convergence Protocol (PLCP), which specifies the header and payload for transmission. It specifies the sensing of the carrier at receiver and how packet formation takes place at the transmitter and packets assemble at the receiver. It specifies ways to converge MAC (Medium Access Control) to PMD at transmitter and separate MAC (Medium Access Control) from PMD at the receiver. An additional sublayer in 802.11b specifies Complementary Code Keying (CCK).

3.13.4 ZigBee

ZigBee is an IEEE standard 802.15.4 protocol. The physical layer radio operates at 2.4 GHz band carrier frequencies with DSSS (direct sequence spread spectrum). It supports a range up to 70 m. Data transfer rate supported is 250 kbps. It supports sixteen channels. Figure 3.15(c) shows a handheld device connected to other devices through wireless protocol using ZigBee.

The ZigBee network is self-organizing and supports peer-to-peer and mesh networks. Self-organizing means that it detects nearby ZigBee devices and establishes communication and network. Peer-to-peer network means the each node at network functions as a requesting device as well as a responding device. Mesh network means that each network functions as a mesh. A node can connect to another directly or through mutually interconnected intermediate nodes. Data transfer is between two devices in peer-to-peer or between a device and multiple devices in the mesh network.

ZigBee protocol supports a large number of sensors, lighting devices, air conditioning, industrial controller and other devices for home and office automation and their remote control and formation of WPAN (wireless personal area network). ZigBee network has a ZigBee router, end devices and coordinator. ZigBee router transfers packets from a neighboring source to a nearby node in the path to destination. The coordinator connects one ZigBee network with another, or connects to WLAN or cellular network. ZigBee end devices are transceivers of data.

Its communication latency is 30 ms. Protocol stack overhead is 28 kB.



Summary

Important points dealt with in this chapter are as follows.

- IO ports, IO devices and timing devices are essential in any system.
- An embedded system connects to the devices like keypad, touchscreen, multiline display unit, printer or modem or motors through ports. During a read or write operation, the processor accesses that address in a memory-mapped IO, as if it accesses a memory address. A decoder takes the system memory or IO address bus signals as the input and generates a port or device select signal, CS and selects the port or device.

- There are two types of IO ports and devices, serial and parallel. Serial communication is in synchronous (master-slave) mode or asynchronous mode.
- A device connects and accesses from and to the system-processor through either a parallel or serial IO port. A device port may be full duplex or half-duplex.
- A device or port has an assigned port address using which the processor accesses the device port control register or status register or data. A device can use the handshaking signals before storing the bits at the port buffer or before accepting the bits from the port buffer.
- Serial communication bits are received at the receiver according to the clock phases of the transmitter. Synchronous serial communication bits from the master carry the clock information also to slave. Asynchronous serial communication bits from a device do not carry the clock information to receiver. Receiver clock phase is independent of the transmitter clock. However, the receiver clock adjusts its phase according to the received bits, for example, the start bit.
- HDLC is protocol for a synchronous communication data link network between the devices.
- A popular asynchronous serial communication mode is UART. Bits are received at the receiver independent of the clock phases at the UART (asynchronous serial input and output port) transmitter. UART in microcontrollers usually sends and receives a byte in a 10-bit format or 11-bit format.
- Another popular asynchronous serial communication mode is RS232C, which is based on UART and is used to connect the data communication equipment such as modem with a data terminal equipment such as computer.
- UART and RS232C can also use handshaking signal DCD and a pair of handshaking signals, (DSR, DTR) and (RTS, CTS).
- Other popular serial ports in the devices are SPI, SCI, SI and SDIO.
- Parallel communication is without or with handshaking signals. The number of embedded systems parallel port or device interfaces to switches, keypad, encoders, motors, LCD controllers and touchscreen. Special purpose ports exist at microcontroller for their interfacing. On-chip peripheral devices internally interface with the processor in microcontroller.
- A timer is essentially a counter getting the count-inputs (ticks) at regular time intervals. Timing and counting devices have a large number of uses in a system. There has to be at least one hardware timer in a system. Software timer is a virtual timing device. A program can use number of software timers in a system.
- Internal programmable timing devices with a processor (microcontroller) unit can be used for many applications and to generate interrupts for the ticks for software timers.
- Watchdog timer is special timer which timeouts and generates interrupts in case certain specified event does not occur during the preset interval. The watchdog timer is used to take care of a system stuck in a certain section of a task for an unnecessarily long time due to some error or hardware failure.
- Real-time clock generates ticks and interrupts the system at regular intervals.
- The use of buses simplifies the interfacing to multiple devices. Several devices can be placed on a common serial bus. Popular serial buses are I²C, CAN, USB and FireWire. Each device has an assigned device address or set of addresses. Using the device addresses of the receiver or slave, a master-processor accesses the remote devices.
- I²C bus is used between multiple ICs for inter Integrated Circuit communication. A device, which initiates the communication and sends the clock pulses, is the master at an instance. A master can communicate to maximum 127 slaves.
- The CAN bus is popularly used in centrally controlled network in automobile electronics.
- USB (Universal Serial Bus) is standard for serial bus communication between the system and devices like scanner, keyboard, printer and mouse. There is a root-hub and all nodes have a tree-like structure.
- Several devices can be placed on a common parallel bus. Popular parallel buses are ISA, PCI and ARM buses.
- Very short distance devices interconnect to a PC or embedded system main bus through the ISA or PCI or ARM bus can be used. These buses connect to main memory bus through a bridge (switch).
- Internet-enabled embedded systems network through protocols in a TCP/IP protocol suite. Popularly used protocols are HTTP, TCP, UDP, IP and Ethernet.
- Wireless communication is used for networking handheld devices over wireless personal area network.
- Embedded systems can interconnect and network without wires using IrDA, Bluetooth, 802.11 or ZigBee protocol compatible hardware and software support.



Keywords and their Definitions

- Asynchronous Communication* : A communication in which a constant phase difference between the transmitter clock and bit recovery clock at the receiver need not be maintained and the clocks that guide the transmitter and receiver are not synchronized. Time interval between which a set or frame of bytes transmits is not pre-fixed and is indeterminate. Asynchronous communication also provides for exchange of handshaking signals before and during the communication.
- Bluetooth* : A self-discovery and self-organizing network protocol for the wireless personal area network and popularly used in mobile handheld devices.
- CAN bus* : A standard bus used at the control area network generally in automotive and industrial electronics.
- COM port* : A port at the computer where a mice, modem or serial printer or mobile smart phone cradle connects for serial IOs in UART mode and there are handshaking signals for exchange of signals before UART mode communication.
- Control register* : A register for bits, which controls or programs the actions of a device. It is used for a write operation only.
- Control cum Status register* : A register at a port address that saves control and status bits and functions as a control register during write of commands and status register address during read of the status.
- Counter* : Unit for getting the count-inputs on the occurrence of events that may be at irregular intervals. It functions as timer when given count-input at regular intervals.
- Debouncing* : When a key is pressed, due to spring action, the key vibrates and thus makes and breaks the contacts. This causes multiple 0s and 1s before the switch pressed state is accounted for. Debouncing by hardware or software removes the signals due to bounces.
- Delay* : An action or communication or execution of codes or occurrence of an event after blocking for a certain pre-defined period.
- Demultiplexing* : A way to separate a multiplexed input and direct the messages to one of the multiple channels at an instance.
- Device* : A unit that has a processing element and that connects to the processor of embedded system internally or through the port or bus. It has fixed pre-assigned port addresses (device addresses) according to its interfacing or bus controller circuit. A device may not provide for bus controller in it for enabling it to function in bus master mode and thus can function in slave mode only.
- Device Decoder* : A circuit to take the system address bus signals as the input and generate a device select signal, CS, for the port address selection during the device read or write instructions of the system processor.
- Event* : A change of present condition, which gives an electric signal at input or output pin or which changes a status bit or which interrupts the processor to enable some action by switching the context and running can ISR.
- Event flag* : A Boolean variable to indicate the event occurrence when it is true; it can be a status register bit. An event register may store the event flag. A flag auto-resets on response to the event in most systems.

- Free Running Counter** : A counter that starts on power-up, which is driven by an internal clock (system clock) directly or through a prescaler or rate control bits and which can neither be stopped nor be reset.
- FSK modulation** : Frequency Shifted Keying. The 0 and 1 logic states are at different frequency levels. For example, 0 at 1050 Hz and 1 at 1250 Hz on a telephone line. It permits use of a channel or a line such as telephone line for serial bit transmission and reception.
- Full duplex** : A serial port having two distinct IO lines or communication channels. For example, a modem connection to the computer COM port. There are two lines TxD and RxD at 9 pins or 25 pins connector. Message flows both ways at an instance.
- Half duplex** : A serial port having one common IO line or channel. For example, a telephone line. Message flows one way at an instance.
- Handshaking signals** : The signals before storing the bits at the port buffer or before accepting the bits from the port buffer or the signals to setup or end the communication between two source and destination.
- Hardware timer** : A timer present in the system as hardware which gets the inputs from the internal clock with the processor or which enables the system clock ticks (interrupts). A device driver program programs it like any other physical device.
- HDLC** : High Level Data Link Control Protocol. It is for synchronous communication between primary (master) and secondary (slave) as per standard defined. It is a bit-oriented protocol.
- Host** : A controller node using a protocol and a circuit for enabling the system to connect the number of devices or peripherals and for providing bus master mode functioning as well as receiving signals and bits from other hosts or devices.
- IO Port** : A port for input or output operation at an instant. Handshake input and handshake output ports are also known as IO ports. For example, a keypad is said to connect to an IO port.
- I²C bus** : A standard bus that follows a communication protocol and is used between multiple ICs. It permits a system to get data and send data to multiple compatible ICs connected on this bus.
- Input Buffer** : A buffer where an input device puts a byte(s) and the processor read that later.
- IrDA** : Infrared Data Association recommended protocol for IR remote control and communication over short distance to device or system in line of sight.
- ISA bus** : A standard bus based on 'IBM Standard Architecture' Bus.
- Isosynchronous Communication** : Communication in which a constant phase difference is not maintained between the frames but maintained within a frame. Clocks that guide the transmitter and receiver are not separate. Only the maximum time interval is not prefixed between which a frame of bytes transmits that is, it can be variable. Between the frames, there is handshaking between the two ends or there may be a pause. Uses are for transmission on a LAN or between two processors.
- Keypad and Keyboard Controllers** : The controllers for interfacing with keypads and keyboard such that they do debouncing of keys, buffer the input characters and interrupt the processor on each input or at end of the line character and send ASCII code(s) as input(s) to the processor for further processing and interpretation as data or command.

- LCD controller** : A controller for LCD displays with internal CGRAM (character graphic RAM) and ROM for fonts of the characters and which gets the commands and data for display from the system port.
- Master slave communication** : A communication between two processors or devices when one processor guides the other using a clock the transmission of the bits to a slave after or before receiving acknowledgement or reply from the addressed slave. A slave can also function as master or vice versa by an appropriate program bit or hardware control.
- Multiplexing** : A way to direct the messages to output channel from the multiple source channels.
- On-chip ports and devices** : The ports and devices along with the processor unit, for example, in microcontrollers, which communicate using system internal buses.
- Open drain output** : A gate with an internally missing connection between its drain and supply. The advantage is that it pulls up the required circuit voltage and current levels when interfacing. An external pull up circuit is needed when using the output.
- Output buffer** : A register buffer from where an output device receives the byte(s) after a processor-write operation.
- Parallel port** : A port for read and write operations on multiple bits simultaneously at an instance.
- PCI bus** : A standard bus used as a 'Peripheral Component Interconnect' bus.
- PCI/X bus** : A standard bus used as a 'PCI Extended 'bus'.
- PISO** : A shift register for a Parallel Input and Serial Output. It is used for serial bit reception in synchronous mode.
- Plug and play** : A device on a bus can be automatically detected when it is attached to the bus and the device can be used directly without resetting or restarting the system.
- Protocol** : A way of transmitting messages on a network by using software that adds additional bits such as the starting bits, headers, addresses of source and destination, error control and ending bits. A protocol suite may have multiple layers and each layer or sublayer uses its protocol before a message transmits on a network.
- PSK modulation** : Phase Shifted Keying modulation. The 0 and 1 logic have different phases in a high frequency signal. PSK modulation permits use of a channel or line such as telephone line for serial bit transmission and reception.
- QPSK** : Quadrature Phase Shifted Keying. An example is the pair of bits 00, 01, 10 and 11, which are sent at different quadrant phase differences of a voice frequency signal. It permits use of the telephone line for serial bit transmission and reception at double the rate. It permits the 56 kbps modem to show a performance equivalent to 112 kbps. QPSK and its enhancements are also used in wireless communication extensively.
- Quasi bi-directional port** : A port with the dual advantage of using a pull-up circuit as per the voltage and current levels required when interfacing it and using no pull-up circuit for a short period sufficient to drive an LSTTL circuit.
- Real Time** : A time that always increments at constant intervals without stopping or resetting and that is used as a reference by the system at all times.
- Real Time Clock (RTC)** : A clock that continuously generates interrupts at regular intervals endlessly. An RTC interrupt ticks the other timers of the system, for example, software timers (SWTs).

<i>RS232C port</i>	: A standard for UART transmission and reception in which TxD and RxD are at different voltage levels (+12 V for '0' and -12 V for '1') and handshaking signals, CTS, RTS, DTR, DCD and RI are at the TTL levels. The RS232C standard is used at the COM ports.
<i>RxD</i>	: A line used for reception of UART serial bits. The 0 and 1 signals are at TTL or RS232C levels and are similar to that for a TxD line.
<i>Serial Port</i>	: A port for read and write operations with one bit at an instance and where each bit of the message is separated by constant time intervals.
<i>SIPO</i>	: A shift register for Serial Parallel Input and Parallel Output. It is used for serial bits transmission in synchronous mode.
<i>Software timer</i>	: Software (a service routine) that executes and increases or decreases a count-variable on an interrupt from a timer output or from a real-time clock interrupt. A software timer also generates interrupt on overflow of count-value or on finishing value of the count-variable (reaching the predefined value) or generating a message for a task. The interrupts can generate by using software interrupt instruction such as SWI.
<i>Status register</i>	: A register for bits, which reflects the status at the port buffer. It is for a <i>read</i> operation only. The status register bit or bits may or may not auto-reset on device servicing after the read.
<i>Synchronous Communication</i>	: Communication in which a constant phase difference is maintained between the clocks that guide the transmitter and receiver. A maximum time interval is pre-fixed between which a frame of bytes transmits.
<i>System Clock</i>	: A clock scaled to the processor clock and which always increments without stopping or resetting and generates interrupts at preset time intervals.
<i>Time division multiplexing</i>	: A way by which messages from different channels can be sent in different time slots.
<i>Timer Finish</i>	: A state after the timer acquired the preset count-value and stopped. An interrupt generates on finishing.
<i>Timer Overflow or Time-Out</i>	: A state in which the number of count-inputs exceeded the last acquirable value and on reaching overflow state, an interrupt can be generated.
<i>Timer Reset</i>	: A state in which the timer shows all bits as 0s or 1s. A reset can also be after overflow in case a timer is programmed for continuous running.
<i>Timer Reload</i>	: State in which timer shows all bits as 0s or 1s. A reload can also take place after finishing in case a timer is programmed for auto reload and start again.
<i>Touch Screen</i>	: A GUI device for displaying icons, pictograms, menus and virtual keyboard on an LCD screen and giving input commands or selecting menus or keying in the data using finger or stylus for touching at appropriate screen-position.
<i>TxD</i>	: A line used for transmission of UART serial bits. The 0 and 1 signals are at RS232C voltage levels when RS232C COM port is used, or at the TTL levels in microcontrollers.
<i>UART</i>	: A standard Asynchronous Serial Input and output port for serial bits. UART (in microcontrollers) usually sends a byte in 10-bit format or 11-bit format. The

<i>10-bit format</i>	is used when a start bit precedes the 8-bit message (character) and a stop bit succeeds the message. An 11-bit format is used when a special bit also precedes the stop bit.
<i>USB bus</i>	: A standard plug and play bus for fast serial transmission and reception.
<i>Watchdog timer</i>	: A timing device in a system that resets or executes a watchdog timer service routine (WDT routine) after fetching the interrupt vector address at the system after a predefined timeout in case a watched event does not happen. When the watched event occurs, it is restarted so that it does not timeout and does not execute WDT routine.
<i>WLAN</i>	: A wireless LAN for networking mobile and wireless devices with a fixed infrastructure and which enables access of devices through access points. The device functions according to IEEE 802.11 standards specified protocols.
<i>ZigBee</i>	: A new wireless network protocol for short-range communication among number of sensors and devices and has self-discovery and self-organizing features.



Review Questions

- (a) What is the advantage of a processor that maps the addresses of IO ports and devices like a memory-device? (b) Give a diagram to interface the port devices with the system buses.
- Compare the advantages and disadvantages of data transfers using serial and parallel ports/devices.
- (a) Explain three modes of serial communication. 'asynchronous' 'isochronous' and 'synchronous' using serial devices with one example each. (b) Describe and compare UART, RS232C and SDIO devices.
- How do the following indicate the start and end of a byte or data frames? (a) UART (b) HDLC (c) CAN
- What are the internal serial-communication devices in (a) 8051 and (b) 68HC11? Compare the modes of working of each of these.
- A device port may have multibyte data input buffer(s) and data output buffer(s). What are the advantages of these?
- Explain the advantages of Internet-enabled systems. How is the Internet-enabled device incorporated in the embedded system?
- Explain the advantages of wireless devices. How do wireless devices network using different protocols?
- What do you mean by buses for networking of serial devices? What do you mean by buses for networking of parallel devices?
- Explain use of each control bit of I²C bus protocol.
- What do you mean by plug and play devices? What are bus protocols of buses UART, RS232C, USB, Bluetooth, CAN and PCI that support plug and play devices?
- What do you mean by hot attachment and detachment? What are bus protocols of buses Bluetooth, UART, CAN, PCI, and USB that support hot attachment and detachment?
- What is a timer? How does a counter perform (a) timer functions (b) prefixed time initiated events generation and (c) time capture functions?
- Why do you need at least one timer device in an embedded system?



Practice Exercises

15. How do the following device features help in embedded systems? (a) Schmitt trigger input (b) low voltage 3.3 V I/Os (c) Dynamically controlled impedance matching (c) PCS subunit (d) PMA subunit and (e) SerDes. Give one exemplary application of each.
16. PPP protocol for point to point networking has 8 starting flag bits, 8 address bits, 8 protocol specification bits, variable number of data bits, 16-bit CRC and 8 ending flag bits. The maximum number of bits per PPP frame can be 12064. How many maximum number of bytes can be transferred per PPP frame? What is the minimum percentage of overhead in the payload (frame)?
17. List the applications of the free running counter, periodically interrupting timer and pulse accumulator counter (PACT). How do you get PWM output from a PACT? How do you get DAC output from a PWM device?
18. A 16-bit counter is getting inputs from an internal clock of 12 MHz. There is a prescaling circuit, which prescales by a factor of 16. What are the time intervals at which overflow interrupts occur from this timer? What will be period before which these interrupts must be serviced?
19. What do you mean by a software timer (SWT)? How do the SWTs help in scheduling multiple tasks in real time? Suppose three SWTs are programmed to timeout after 1024, 2048 and 4096 times from the overflow interrupts from the timer. What will be rate of timeout interrupts from each SWT?
20. What are the advantages and disadvantages of negative acknowledgement bit?
21. A new generation automobile has about 100 embedded systems. How do the bus arbitration bits, control bits for address and data length, data bits, CRC check bits, acknowledgement bits and ending bits in CAN bus help the networking of devices distributed in an automobile system.
22. How does the USB protocol provide for the device attachment, configuration, reset, reconfiguration, bandwidth sharing with other devices, device detachment (while others are in operation) and reattachment?
23. Design a table that compares the maximum operational speeds and bus lengths and give two example of the uses of each of the following serial devices: (a) UART (b) 1-wire CAN (c) Industrial I²C (d) SM I²C Bus (e) SPI of 68 Series Motorola Microcontrollers (f) Fault tolerant CAN (g) Standard Serial Port (h) FireWire (i) I²C (j) High Speed CAN (k) IEEE 1284 (l) High Speed I²C (m) USB 1.1 Low Speed Channel and High Speed Channel (n) SCSI parallel (o) Fast SCSI (p) Ultra SCSI-3 (q) FireWire/IEEE 1394 (r) High Speed USB 2.0.
24. Use web search. Design a table that compares the maximum operational speeds and bus lengths and give two example of the uses of each of the following parallel devices: (a) ISA (b) EISA (c) PCI (d) PCI-X (e) COMPACT PCI (f) GMII (Gigabit Ethernet MAC Interchange Interface) (g) XGMI (10 Gigabit Ethernet MAC Interchange Interface) (h) CSIX-1. 6.6 Gbps 32-bit HSTL with 200 MHz performance (i) RapidIO™ Interconnect Specification v1.1 at 8 Gbps with 500 MBps performance or 250 MHz dual direction registering performance using 8-bit LVDS (Low Voltage Data Bus).
25. Use web search and design a table that gives the features of the following latest generation serial buses. (a) IEEE 802.3-2000 [1 Gbps bandwidth Gigabit Ethernet MAC (Media Access Control)] for 125 MHz performance (b) IEEE P802.3oe draft 4.1 [10 Gbps Ethernet MAC] for 156.25 MHz dual direction performance (c) IEEE P802.3oe draft 4.1 [12.5 Gbps Ethernet MAC] for four channel 3.125 Gbps per channel transceiver performance (d) XAUI (10 Gigabit Attachment Unit) (e) XSB1 (10 Gigabit Serial Bus Interchange) (f) SONET OC-48, OC-192 and OC-768 (g) ATM OC-12/46/192.
26. Take a mobile smart phone with a T9 keypad. Write a table for the states of each key. Write another table for the new states generated by a combination of two keys.
27. Compare the parallel ports interfaces for the keypad, printer, LCD-controller and touchscreen.
28. Show the use of USB devices in the digital camera, printer and computer for downloading a picture from camera to computer, printing the pictures in camera and saving in flash memory. What is the difference between USB host and USB device in a system?
29. Compare different serial buses.
30. Compare different wireless protocols.

Device Drivers and Interrupts Service Mechanism

4

R

We have learnt the following in previous chapter:

- Embedded system hardware has devices which communicate through serial and parallel ports and buses. There may also be ports for real-time voice and video I/Os.
- A microcontroller has serial communication and timing devices. It may have keypad, stepper motor, LCD and touch screen controllers.
- Serial or parallel buses interconnect the distributed ports and devices.
- I²C bus is used for inter-IC communication. It interconnects multiple distributed ICs.
- CAN bus is used at control network of the distributed devices. It is used in automobiles and industrial systems.
- USB is used for the fast serial transmission and reception between the embedded-system and serial devices such as the keyboard, printer and scanner.
- FireWire (IEEE 1394) is bus used for the high-speed interfacing of 800 Mbps multimedia devices.

e

c

a

f

f

R

- Parallel buses, ISA and PCI/PCI-X are used for bus communication of devices between the host computer or system and PC-based devices or systems or cards, for example, NIC (Network Interface Card).
- Wireless protocols are used for the communication and synchronization of distributed devices in wireless personal area network.
- Internet-enabled embedded systems can network to the Internet using the TCP/IP suite of protocols.

e

We also learnt

- A device-access is required for opening, connecting, binding, reading, writing, disconnecting or closing it. Processor accesses a device using the addresses of device registers and buffers. A processor accesses the internal devices, devices at the I/O ports, peripheral devices and other off-chip devices using the addresses.
- A simple device such as SPI port (Section 3.2.4) has addresses for three sets of its registers: data register(s) (or buffers), control register(s) and status register(s).
- A device can also have number of registers (Table 3.4). For example, PCI bus-driven device (Section 3.12.2) has 64 bytes standard device-independent configuration registers.

C

a

[

[

L

E

A

R

N

I

N

G

O

B

J

E

C

T

I

V

E

S

In this chapter, we will learn how the concept of interrupt service routines is used to address and service the device I/Os, requests and interrupts. We will learn the following:

1. Programmed I/O busy and wait method and problems with this I/O method.
2. Interrupts and working of interrupts service mechanism in the system and simple examples of hardware and software interrupts.
3. Interrupt service routine (ISRs) are called by the system when device-hardware interrupts take place.
4. Software functions for the signals and exceptions also call ISRs. An ISR is also called on a trap or execution of a software instruction for interrupt.
5. Use of interrupt vectors, vector table and masking.
6. Interrupt latency and deadline for an interrupt service.
7. Context and context switching on an interrupt.
8. New methods for the fast context switching adopted in the processors.
9. Classification of processors for an interrupt service that 'Save' or 'Don't Save' the context other than the program counter.
10. Use of the DMA channel for facilitating the small interrupt latency period for the multiple data transfers in quick succession.
11. Assignment of software and hardware priorities among the multiple sources of interrupts.
12. Methods of service in case of simultaneous service demand from multiple interrupting sources.
13. Device drivers for a device or port initialization and accesses.
14. Use of device drivers, for example, Linux Internals.
15. Examples of device initialization and device driver coding for the parallel ports and serial-line UART.

4.1 PROGRAMMED-I/O BUSY-WAIT APPROACH WITHOUT INTERRUPT SERVICE MECHANISM

Example 4.1 shows an example of programming a device service with programmed I/O busy-wait approach without using a device interrupt and the corresponding ISR. This example will make clear the problems in this approach and advantages of using an interrupt-based service mechanism.

Example 4.1

Assume a 64 kbps network. Using a UART that transmits in the format of 11-bit per character, the network transmits at most $64 \text{ kbps} \div 11 = 5818$ characters per second, which means that for every $171.9 \mu\text{s}$ a character is expected. Before $171.9 \mu\text{s}$, the receiver port must be checked to find and read another character assuming that all the received characters are in succession without any time gap.

Let port A be at an Ethernet interface card in a PC, and port B be its modem input which puts the characters on the telephone line. Let *In_A_Out_B* be a routine that receives an input character from port A and re-transmits an output character to port B. Assume that there is no interrupt generation and interrupt service (handling) mechanism. Let *In_A_Out_B* routine has to call the following steps *a* to *e* and executes the cycles of functions *i* to *v*, thus ensuring that the modem port A does not miss reading the character.

In_A_Out_B routine:

1. Call function *i*
2. Call function *ii*
3. Call function *iii*
4. Call function *iv*
5. Call function *v*
6. Loop back to step 1

In_A_Out_B routine calls the following steps.

Step *a*: Function *i*: Check for a character at port A. If not available, then wait.

Step *b*: Function *ii*: Read port A bytes (characters for message) and return to step *a* instruction, which will call function *iii*.

Step *c*: Function *iii*: Decrypt the message and return to step *a* instruction, which will call function *iv*.

Step *d*: Function *iv*: Encode the message and return to step *a* instruction, which will call function *v*.

Step *e*: Function *v*: Transmit the encoded message to port B and return to step *a* last instruction, which will start step *a* from the beginning.

Step *a* is also called polling. Polling a port means to find the status of the port, ready with a character (byte) at input. Polling must start before 171.9 μ s because characters are expected at 64 kbps. If the program instructions in the steps *b*, *c*, *d* and *e* and functions *ii* to *v* take a total running time of less than 171.9 μ s then this approach works.

Problems with the busy-wait programming approach is as follows.

1. The program must switch to execute the *In_A_Out_B* cycle of steps *a* to *e* within a period less than 171.9 μ s. Programmer must ensure that steps of *In_A_Out_B* and any other device program steps never exceed this time.
2. When the characters are not received at Port A in regular succession, the waiting period during step *a* for polling the port can be very significantly. Wastage of processor time for the waiting periods is the most significant disadvantage of the busy-wait approach.
3. When other ports and devices are also present in the system, the programming problem is to poll each port and device and ensure that the program switches to execute the *In_A_Out_B* step *a* as well as switches to poll each port or device on time and then execute each service routines related to the functions of other ports and devices within a specific time interval and ensure that each one is polled on time.
4. The program and functions are processor- and device-specific in the previous busy-wait approach and all system functions must execute in synchronization and the timings are completely dependent on periods taken for software execution.

Instead of continuously checking for characters at the port A by executing function (*i*), when a modem receives an input character and sets a status bit in its status register, an interrupt from port A should be generated. In response to the interrupt an interrupt service routine *ISR_PortA_Character* should then be executed (Example 4.2 in Section 4.2). This will be the efficient solution instead of wait at step *a*.

Device service without using an ISR is by the routine (function) call similar to *In_A_Out_B*.

Each routine (function) call has the following features.

1. A function call after executing any instruction in any program is a planned (user-programmed) diversion from the current sequence of instructions to another sequence of instructions and this sequence of instructions executes till the return from that.
2. On a function call, the instructions are executed as a function in the 'C' or a method in Java.
3. Function calls are nested. Nesting can be explained as follows: when a function 1 calls another function 2 which in turn calls another function 3, then on return from 3, the return is to function 2 and then to function 1.

Figure 4.1 shows the *In_A_Out_B* routine steps *a* to *e* for the five functions *i* to *v* called by *In_A_Out_B* and how each called function processes on a *call* and on a *return* from that. Numberings on the arrows show the sequences during the program run (flow).

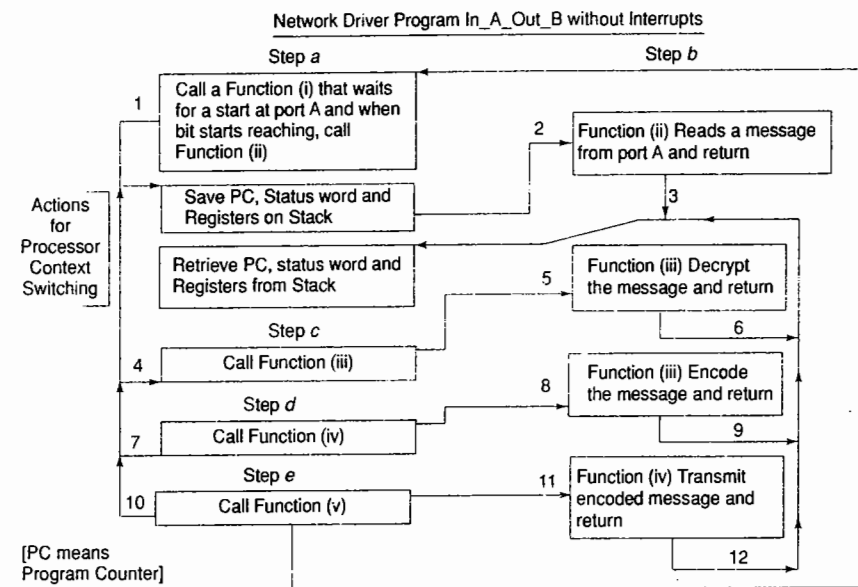


Fig. 4.1 Steps *a* to *e* for five function calls in an exemplary network drive program. *IN_A_OUT_B*, also shown is how each called function processes on a *call* and on a *return*. Numberings on the arrows show the program running sequences

One approach is 'programmed IO' transfer, also called 'busy and wait' transfer for service (accessing the device addresses for input or output or any other action). System functions in synchronization and the timings are completely software-dependent. When waiting periods are a significant fraction of the total program's execution period, wastage of the processor's time in waiting is the most significant disadvantage of this approach. Programmed IO approach can be used in single-purpose processors (controllers).

4.2 ISR CONCEPT

Interrupt means event, which invites the attention of processor for some action on the hardware or software event.

1. When a device or port is ready, a device or port generates an interrupt or when it completes the assigned action, it generates an interrupt. This interrupt is called hardware interrupt.
2. When software run-time exception condition is detected, either processor hardware or a software instruction generates an interrupt. This interrupt is called software interrupt or *trap* or *exception*.
3. Software can execute the software instruction for interrupt to *signal* the execution of ISR. The interrupt due to signal is also a software interrupt [The *signal* differs from the function in the sense that the execution of the signal handler function (ISR) can be masked and till the mask is reset, the handler will not execute. Function on the other hand always executes on the call after a call-instruction.]

In response to the interrupt, the routine or program, which is running at present gets interrupted and an ISR is executed. ISR is also called device driver ISR in the case of devices and is called *exception* or *signal* or *trap handler* in the case of software interrupts. Device driver ISRs execute on software interrupts from device *open()*, *close()*, *read()*, *write()* or other device functions.

Examples in Sections 4.2.1 and 4.2.2 show the importance of interrupts and accessing of the devices using the ISRs and the importance of using the ISRs which generate on *traps* or *exceptions* or *signals*.

4.2.1 Examples of Port or Device Interrupts and ISRs

Following are the examples of interrupt events and accessing of devices using the ISRs.

Example 4.2

Recapitulate Example 4.1. Assume that a character input to the modem generates a port A interrupt and sets a status bit in the status register. On interrupt, a service routine `ISR_PortA_Character` runs so that it ensures that the modem port A does not miss reading the character. `ISR_PortA_Character` executes step *f* in place of the step *a* function *i* and step *b* function *ii* of `In_A_Out_B` routine in Example 4.1. It places the read character in a memory buffer. Steps *c*, *d* and *e* are independent and are now parts of a function-call `Out_B`. `ISR_PortA_Character` executes as follows:

1. Step *f* function *vi*: *Read* Port A character. *Reset* the status bit so that the modem is ready for the next character input (resetting of the status bit is generally automatic without the need for specific instruction). *Put* it in a memory buffer. Memory buffer is a set of memory addresses where the bytes (characters) are queued for processing later.
2. Return from the ISR.

`Out_B` routine is as follows:

1. Step *g*: Call function *vi* to decrypt the message characters at the memory buffer and return for the next instruction step *h*.
2. Step *h*: Call function *vii* to encode the message character and return for the next instruction step *k*.
3. Step *k*: Call function *viii* to transmit the encoded character to port B.
4. Return from the function.

Figure 4.2 shows the step *f* executing on `ISR_PortA_Character` on port A interrupt and steps *g* to *k* in `Out_B` routine. Numberings on the arrows show the program running sequences.

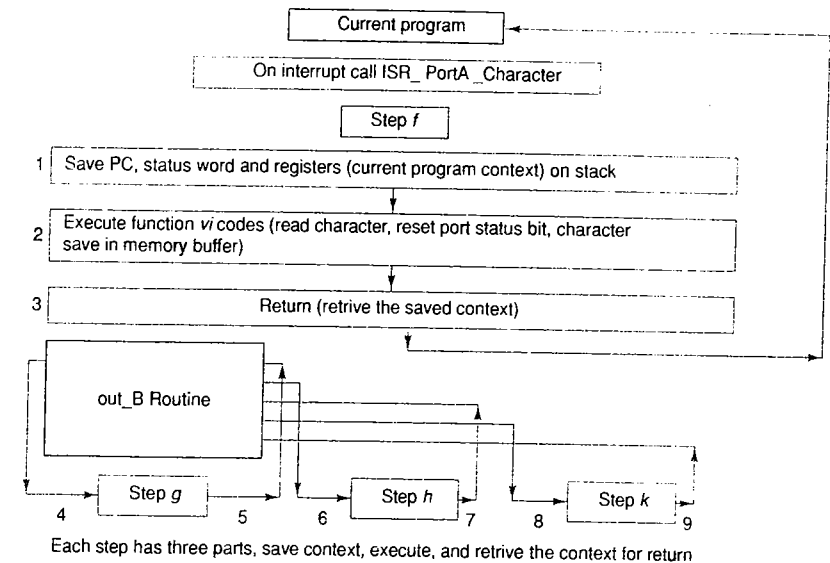
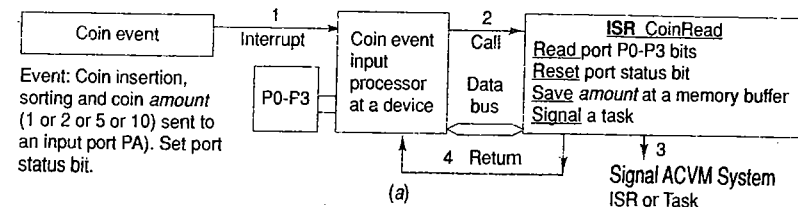


Fig. 4.2 Step *f* executing on `ISR_PortA_Character` on port A interrupt and steps *g* to *k* in `Out_B` routine. Numberings on the arrows show the sequences of running the program-steps *f*, *g*, *h* and *k*

Example 4.3

Assume a device for coin amount input in an automatic chocolate-vending machine (Section 1.10.2). Without interrupt mechanism, one way is the busy wait transfer by which the device waits for the coin continuously, activates on sensing the coin and runs the service routine.

In the event-driven method the device should awaken and activate on each *interrupt* after sensing each coin-inserting event. The device is at an input port. It collects a coin inserted by a child. The system awakens and activates on interrupt through a hardware interrupt. The system on port hardware *interrupt* collects the coin by running a service routine. This routine is called interrupt handler routine or ISR or *device driver function* for the coin-port read. Figure 4.3(a) shows the ISR in the ACVM example.



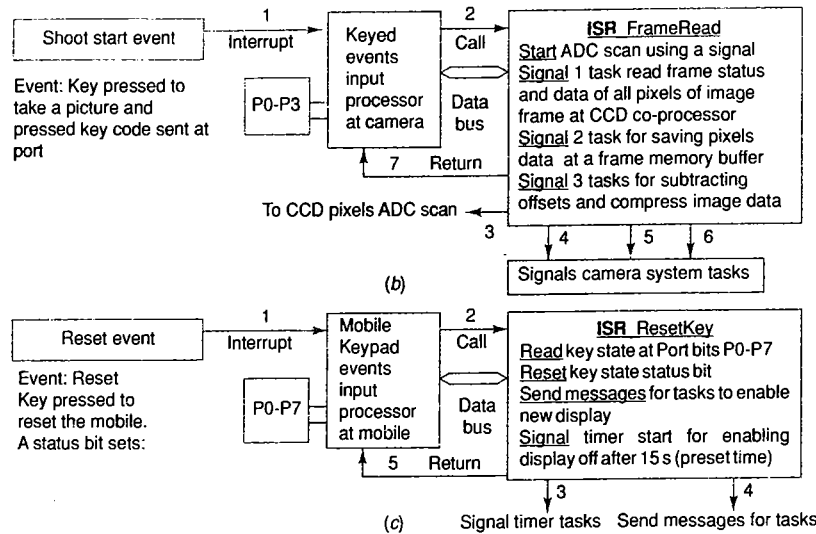


Fig. 4.3 (a) Use of ISR in the automatic chocolate vending machine (b) Use of ISR and three signals 1, 2 and 3 for three tasks in the digital camera example (c) Use of ISR in the mobile phone reset-key interrupt example

Example 4.4

Assume a digital camera system (Section 1.10.4). It has an image input device. When the system activates the device should grab image-frame data. The system awakens and activates on a switch *interrupt*. The interrupt is through a hardware signal from the device switch. On the *interrupt*, an ISR (can also be considered as camera's imaging device-driver function) for the *read* starts execution, it passes a message (signal) 1 to a function or program thread or task, which senses the image and then the function reads the CCD device frame buffer; then the routine passes signal 2 to another function or program thread or task to process and then signal 3. Subtracts offsets using a task and compresses image-data using a task. This task also saves the image frame data-compressed file in a flash memory. The camera system again awakens and activates on *interrupt* through a hardware signal from a device switch and prints the file picture image after file decompression. The system on *interrupt* then runs another ISR. The ISR routine is the device-driver write-function for the outputs to printer through the USB bus connected to the printer. Figure 4.3(b) shows the use of the ISR for frame read in the digital camera example.

ISR accesses a device for service (configuring, initializing, activating, opening, attaching, reading, writing, resetting, deactivating or closing). ISRs thus function as the device drivers.

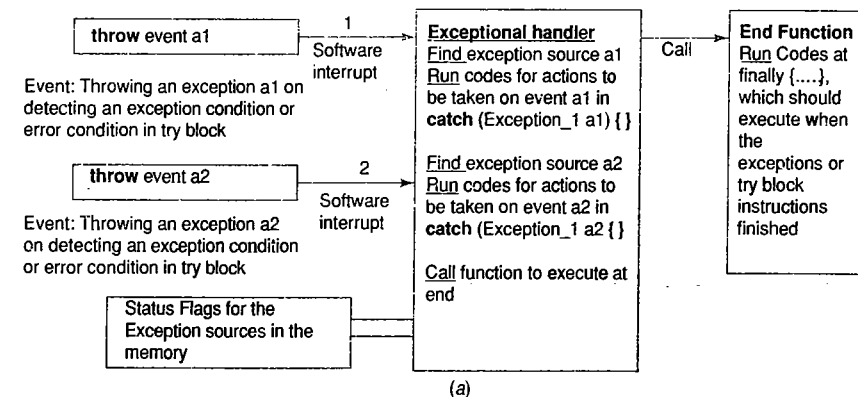
Example 4.5

Assume a mobile phone system (Section 1.10.5). It has a system reset key, which when pressed resets the system to an initial state. When reset key is pressed the system awakens and activates a *reset interrupt* through a hardware signal from reset key circuit. On the *interrupt*, an ISR (can also be considered as reset-key device-driver function) suspends all activity of the system, sends messages to the display LCD screen, and also signals to activate LCD display-off timer device for 15s timeout (for example). After the timeout the system again awakens and activates on *interrupt* through the internal hardware signal from timer device and runs another ISR to send a control bit to the LCD device. The ISR routine is the device-driver LCD off-function for the LCD device. The devices switch off by reset of control bit. Figure 4.3(c) shows the use of the ISR in the mobile-phone reset-key interrupt.

4.2.2 Examples of Software Interrupts and ISRs

Examples 4.2 to 4.5 clearly show that interrupts and ISRs (device-drivers) play the major role in using the system hardware and devices. Think of any system hardware and it will have devices and thus needs device drivers. The embedded software or the operating system for application software must consist of the codes for the device (i) configuring (initializing), (ii) activating (also called opening or attaching), (iii) driving function for read, (iv) driving function for write and (iv) resetting (also called deactivating or closing or detaching). Each device task is completed by first using an ISR—a device-driver function calls the ISR by using a software interrupt instruction (SWI).

A program must detect error condition or run-time exceptional condition encountered during the running. In a program either the hardware detects this condition (called trap) or an instruction SWI is used that executes on detecting the exceptional run-time condition during computations or communication. For example, detecting that the square root of a negative number is being calculated or detecting the illegal argument in the function or detecting that the *connection* to network is not found. Detection of exceptional run-time condition is called *throwing* an exception by the program. An interrupt service routine (exceptional handler routine) executes, which is called *catch* function as it executes on catching the exception thrown by executing an SWI. Figure 4.4(a) shows use of SWI instruction for calling an ISR in the function for throwing and catching the exceptional run-time conditions encountered during computations. Figure 4.4(b) shows the use of signal generated by SWI, and signal handling after that.



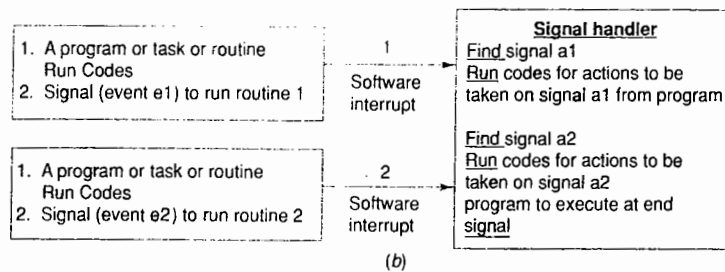


Fig. 4.4 (a) Use of software interrupt (SWI) instruction for calling an interrupt service routines (ISR) in the software on throwing and catching the exceptional run-time conditions encountered during computations (b) Use of SWIs to signal another routines or program tasks or program threads to start

Example 4.6 is given here to clearly show that SWI and execution of the ISRs (also called *exception handlers* or just *exceptions*) on SWIs. The SWIs also play a major role in embedded system software by the use of ISRs for device driver functions—create (), open (), read () or other.

Example 4.6

Consider the following codes.

```
try {
/* Codes for execution in which a run-time exception or number of run-
time exception conditions may encounter, for example square root
of a negative number or a percentage value exceeding 100% or decreasing
below 0%. The condition is trapped and on trapping throws an
exception*/
.
If ((A - B) < 0.1) throw a1; x = y + sqrt (A - B); /* Findif A - B is
-ve number. If yes, throw an exception a1 and call a catch
function). */
.
y = .../ * Calculate y */
If ((y > 100 || y < 0) throw a2; /* Find if y > 100. If yes, throw
exception a2 and call a catch function*/
.
}
```

```
catch (Exception_1 a1) {
/* Code for action on throwing (trapping) A - B < 0.0 exception */
```

```
.
}
.
catch {Exception_1 a2
/* Code for action on throwing (trapping) y < 0 or y > 100
exception*/
.
}
.
finally {
/* Final codes, which should execute on exception or after
try block instructions over */
.
}
```

High-level Java or C++ codes when compiled, during compilation the SWI instructions will be inserted for trapping (A - B) as a negative number and for trapping $y > 100$ or less than 0 as follows:

1. Software instruction SWI a1 will cause processor interrupt. In response, the software ISR function 'catch (Exception_1 a1) { }' executes on throwing of the exception a1 at try block execution. SWI a1 is used after catching the exception a1 whenever it is thrown.
2. Software instruction SWI a2 will cause processor interrupt. In response, the software ISR function 'catch (Exception_2 a2) { }' executes on throwing of the exception a2 during try block execution. SWI a2 is used after catching the exception a2 whenever it is thrown.
3. Software instruction SWI a3 will cause processor interrupt and in response will *signal* software ISR function 'finally { }' to execute either at the end of the try or at the end of the catch function codes. SWI a3 is used after the try and catch functions finish, then *finally* function will perform final task, for example, exit from the program or call another function.

A user program under execution currently by the processor does not know when its try function will throw the exceptions a1 or a2 or when the signal handler throws a3.

An ISR call has the following features.

1. An ISR call due to interrupt executes an event. Event can occur at any moment and event occurrences are asynchronous.
2. ISR call is event-based diversion from the current sequence of instructions (routine or program) to another sequence of instructions (routine or program). This sequence of instructions executes till the return instruction.
3. Event can be a device or port event or software computational exceptional condition detected by hardware or detected by a program, which throws the exception. An event can be signalled by software interrupt instruction SWI used in device driving functions create (), open (), etc.
4. An interrupt service mechanism exists in a system to call the ISRs from multiple sources (Section 4.4).

5. Diversion to ISR may or may not take place on finishing the execution of any instruction in the presently running routine. The execution of the ISRs can be masked by an instruction to set a mask bit and can be unmasked by another instruction to reset the mask bit. [Except a few interrupt sources called non-maskable source (Section 4.4.3).] An instruction in a function or program thread or task can disable or enable an ISR call or all ISR calls (Section 4.4.3).
 6. On an interrupt call, the instructions do not execute continuously exactly like a C function or a Java method. These execute as per the interrupt mechanism of the system. For example, 'return' from an ISR differs in certain important aspects. An interrupt mechanism may be such that an ISR on beginning the execution may disable automatically other device(s) interrupt services. These are automatically re-enabled if they were enabled before a service call. Another interrupt mechanism may be such that an ISR on beginning the execution does not disable automatically other device(s) interrupt services and there can be in-between diversion in the case of the unmasked higher priority interrupts (Section 4.5.1).
 7. There can be multiple interrupt calls during running of an ISR for diversion to other ISRs. The ISR calls need not be the nesting of the ISRs unlike the case of the function calls and there is diversion to pending higher priority interrupt either at the end or in-between the interrupted ISR.
- Section 7.6 will explain the distinction between functions, ISRs and tasks by their characteristics.

Interrupt is an event from a device or hardware action or software instruction. In response to the interrupt, a presently running program is interrupted and a service routine executes. The routine is called ISR. It is also called *device driver* in case of interrupts from the devices. It is also called *exception handler* in case of interrupts from the software. ISR-based approach facilitates an efficient synchronization of the function-calls and ISR-calls. The timings when an ISR executes are hardware or software interrupt event dependent. There is therefore no waiting period due to no need of device polling.

4.2.3 Interrupt Service Threads as Second-Level Interrupt Handlers

An ISR can be executed in two parts.

1. One part is the short execution time taking service routine and can be called as first-level ISR (FLISR). It runs the critical part of the ISR and execute a *signal* function to enable the OS to schedule for running the remaining part later. It can also send a message using a function to enable the OS to initiate a task later on after return from the ISR. The task waits during execution of interrupts routines and signal functions. The FLISR does the device-dependent handling only. For example, it does not perform decryption of data received from the network. It simply does the transfer of data to the memory buffer for the device data.
2. The second part is the long service routine called interrupt service thread (IST) or second-level ISR (SLISR), which executes on the *signal* of the first part. The OS schedules the IST as per its priority. IST does the device-independent handling. IST is also the software interrupt thread as it is triggered by an SWI (software interrupt instruction) for the *signal* in FLISR.

Figure 4.3(b) showed used of *signal* in ISR-FrameRead in digital camera system. Figure 4.5 shows how ADC scan is initiated by an SLISR call from FLISR. Figure 4.5 shows the FLISR and second-level IST approach to handle the device hardware interrupts followed by software interrupts in upper part and the use of this approach in a camera in lower part.

Interrupt service can be done in two parts: a hardware device-dependent code in the FLISR, which has a short execution time and a software interrupt initiated SLISR, which is also called IST. A task can also be sent message by FLISR. The task runs after the IST.

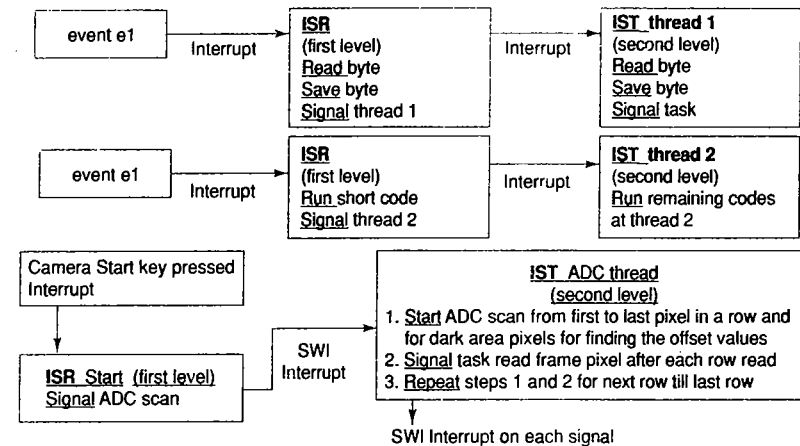


Fig. 4.5 First-level interrupt service routine and second-level interrupt service thread approach to handle the device hardware interrupt followed by the software interrupt to call SLISR—an IST and the use of this approach in a camera

4.2.4 Device Driver

Each device in a system needs device driver routines. An ISR relates to a device driver function. A device driver is a function used by a high-level language programmer and does the interaction with the device hardware and communicates data to the device, sends control commands to the device and runs the codes for reading the device data. A programmer uses generic commands for the device driver for using a device. The OS provides these generic commands.

The examples of generic functions used for the commands to the device are device *create ()*, *open ()*, *connect ()*, *bind ()*, *read ()*, *write ()*, *ioctl ()* [for IO control], *delete ()* and *close ()*. Device driver code is different in different OS. Same device may have different codes for the driver when the system is using different OS.

A device driver function uses SWI, which initiates the interrupt service. The device uses the system and IO buses required for the device service. Device driver can be considered as a function in software layer of an application program and the device.

For example, the application program sends the commands to write on display screen of a mobile the *contact names* from the contact database. LCD display device driver calls an SWI, and an ISR does that without the application programmer knowing how does LCD device interface in the system, what are the addresses which are used, what and where and how are the control (command) and status registers used.

For a programmer, using the device driver's generic functions for reading or writing from and to the device is analogous to reading or writing any other device or data file except that the device and file have different device identity numbers.

The driver routine controls a device without requiring understanding of the device configuration, control, status, data and other registers, when using the generic functions. Device driver runs the ISRs of the device. Each ISR is the low-level part of the device driver generic function, which executes on software interrupt instruction.

The driver translates a program generic function for using the device and sends the necessary commands to the device configuration and control registers. The driver uses the device control, status and data registers.

The driver does the opening, configuring, initializing, attaching, reading, writing, closing and detaching the device by initiating the corresponding ISRs.

Drivers of many devices, such as printers, touch screen, LCD display, keypad, keyboard, are part of the OS. Section 4.9 will describe device drivers in detail.

Each device high-level language program in a system uses device driver functions. A programmer uses generic commands, `create()`, `open()`, `connect()`, `bind()`, `read()`, `write()`, `ioctl()`, `delete()` and `close()` and uses for each device a device identity number. Device driver executes the SWIs, which call the ISRs for using the device hardware and memory allotted to that. SWIs are dedicated for the device service and perform all the necessary actions.

4.3 INTERRUPT SOURCES

Hardware sources can be from internal devices or external peripherals, which interrupt the ongoing routine and thereby cause diversion to corresponding ISR. Software sources for interrupt are related to (i) processor detecting (trapping) computational error for an illegal op-code during execution or (ii) execution of an SWI instruction to cause processor-interrupt of ongoing routine.

Each of the interrupt sources (when not masked) (or groups of interrupt sources) demands a temporary transfer of control from the presently executed routine to the ISR corresponding to the source.

The internal sources and devices differ in different processors or microcontrollers or devices and their versions and families. Table 4.1 gives a classification as hardware and software interrupts from several sources. Not all the given types of sources in the table may be present or enabled in a given system. Further, there may be some other special types of sources provided in the system.

Hardware Interrupts Related to Internal Devices There are number of hardware interrupt sources which can interrupt an ongoing program. These are processor or microcontroller or internal device hardware-specific. An example of a hardware-related interrupt is timer overflow interrupt generated by the microcontroller hardware. Row 1 of Table 4.1 lists common internal devices interrupt sources.

Hardware Interrupts Related to External Devices – 1 There can be external hardware interrupt source for interrupting an ongoing program that also provides the ISR address or vector address (Section 4.4.1) or interrupt-type information through the data bus. Row 2 of Table 4.1 lists these interrupt sources. External hardware interrupts with ISR addresses information sent by the devices themselves (Section 4.4.1) and are device hardware-specific.

Example 4.7

An example of external hardware-related interrupt with device sending the interrupt on INTR pin is the 80x86 processor. When INTR pin activates on an interrupt from the external device, the processor issues two cycles of acknowledgements in two clock cycles through the INTA (interrupt acknowledgement) pin. During the second cycle of acknowledgement, the external device sends the type of interrupt information on data bus. Information is for one byte n . 80x86 internally signals instruction INT n , which means that it executes interrupt of type n , where n can be between 0 and 255. INT n causes the processor vectoring to address $0x00004 \times n$. [SWI in 80x86 is denoted by INT.]

Hardware Interrupts Related to External Devices – 2 External hardware interrupts with their ISR vector addresses (Section 4.4.1) are processor or microcontroller-specific interrupts of an ongoing program. External interrupting source does not send interrupt-type or ISR address-related information. An example of external hardware-related interrupt in which the interrupt-type information internally generates is an interrupt on NMI (non-maskable interrupt) pin in the 80x86 processor. Row 3 of Table 4.1 lists these interrupt sources.

Table 4.1 Classification and Sources of Interrupts¹

Sources	Examples
<i>Internal hardware device sources</i>	1. Parallel port; 2. UART serial receiver port – [Noise, Overrun, Frame-Error, IDLE, RDRF in 68HC11]; 3. Synchronous receiver byte completion; 4. UART serial transmit port-transmission complete [e.g. TDRE (transmitter data register Empty)]; 5. Synchronous transmission of byte completed; 6. ADC start of conversion; 7. ADC end of conversion; 8. Pulse-accumulator overflow; 9. Real-time clock time-outs (Section 3.8); 10. Watchdog timer resets (Section 3.7); 11. Timers overflows on time-out (Section 3.6); 12. Timer comparison with output compare register; 13. Timer capture on input (Section 3.6)
<i>External hardware devices providing the ISR address or vector address or type externally²</i>	INTR in 8086 and 80x86
<i>External hardware devices with internal vector address generation</i>	1. Non-maskable pin [NMI in 8086 and 80x86]; 2. Within first few clock cycles unmaskable declarable pin (interrupt request pin) but otherwise maskable XIRQ [in 68HC11]; 3. Maskable pin (interrupt request pin) [INT0 and INT 1 in 8051, IRQ in 68HC11]
<i>Software error-related sources (exceptions³ or SW-traps)</i>	1. Division by zero detection (or trap) by hardware; 2. Over-flow by hardware; 3. Under-flow by hardware; 4. Illegal opcode by hardware
<i>Software instruction-related sources (exceptions⁴ or SW-traps SW-signal)</i>	Programmer-defined exceptions ³ or traps for handling exceptional run-time conditions or programmer-defined signal for executing ISR to handle further actions or signals from device driver functions

¹ Processor-specific examples are in bracket.

² Example 4.7 explains this.

³ The processor internally generates a trap or exception. An example is *division by zero* in 80x86. Example 4.8 explains this.

⁴ The second type of exception is the user program-defined exception. Example 4.6 explained this. *Signal* is a term sometimes used in high level program for software interrupt instruction in assembly language. For example, in VxWorks RTOS. [Refer Section 9.3.] *Signal* or *exception* is an interrupt on the setting of certain conditions or on obtaining certain results or output during a program run or a signal for some action. The condition examples are square root of a negative number or percentage computation resulting in values greater than 100% or an IO connection not found.

Software Error-Related Hardware interrupts There can be the software-error related interrupts generated by processor hardware. Each processor has a specific instruction set. It is designed for that set only. An illegal code (instruction in the software) is an instruction, which does not correspond

to any instruction in this set. Whenever the processor fetches illegal code, an interrupt occurs in certain processors. The error-related interrupts are also called hardware-generated *software traps* (or *software exceptions*). A software error called *trap* or *exception* may generate in the processor hardware for an illegal or not-implemented opcode found during execution. The examples are as follows: (i) There is an *illegal opcode trap* in 68HC11. This error causes an interrupt to a vector address (Section 4.4.1). (ii) Non-implementable opcode error causes an interrupt to a vector address in 80196.

Software error *exception* or *trap*-related sources cause the interrupt of an ongoing program computations in certain processors. Examples are the division by zero (also known as type 0 interrupt as it is also generated by a software interrupt instruction *INT 0* in 80x86) and overflow (also known as type 2 interrupt as it is also generated by *Int 2* instruction) in 80x86. These two interrupts, types 0 and 2 are generated by the hardware within the ALU of the processor. Row 4 of Table 4.1 lists these interrupt sources. Example 4.8 explains a software-related *trap* or *exception*, which is an interrupt generated by the processor hardware on division by 0.

Example 4.8

Assume that a division by zero occurs during execution of a certain instruction of a program. An ISR is needed which must execute whenever the division by zero occurs. This ISR could be to display 'A division by zero error at ' on the screen and then terminate or pause the ongoing program.

A user program under execution currently by the processor does not know when its ALU will issue this internal error flag (a hardware signal). The service routine executes by using an interrupt mechanism which is meant for service on a zero-division error-signal. On setting of the signal, an interrupt of the ongoing program happens just after completing the current instruction that is being executed, and then the ISR executes for postzero division tasks after resetting the flag.

Executing software error-related processor interrupts are needed to respond to errors such as division by zero or illegal opcode, which is detected by the processor hardware. These are called traps and some time also called exceptions. These are essential for handling run-time errors detected by the system hardware.

Software Instruction-Related Interrupts Sources A program can also *handle* specific computational errors or run-time conditions or signalling some condition. For instance, Example 4.6 showed the handling of negative number square root SWI, which is handled by SWI instruction in the instruction set of a processor. Processors provide for software instruction(s) related to the traps, signals or exceptions.

1. There are certain software instructions for interrupting and then diverting to the ISR also called the signal handler. These are used for signalling (or switching) to another routine from an ongoing routine or task or thread (Section 7.10). Figure 4.4(b) showed the signal generated by SWI and signal handling.
2. Software instructions are also used for trapping some run-time error conditions (called throwing exceptions) and executing exceptional handlers on catching the exceptions (Example 4.6).

An example of a software interrupt is the interrupt generated by a software instruction *INT n* in the 80x86 processor or SWI in ARM7. Row 5 of Table 4.1 lists these interrupt sources. SWI instruction differs from a function *call* instruction as follows.

1. Software interrupt in 68HC11 is caused by instruction, SWI.
2. There is a single-byte instruction *INT0* in 80x86. It generates type 0 interrupt, which means that the interrupt should be generated with the corresponding vector address 0x00000. Instead of the type 0 interrupt that 8086 and 80x86 hardware may also generate on a division by zero, the instruction *INT0* does exactly that.

3. There is another single byte 8086 and 80x86 instruction *TYPE3* (corresponding vector address $0 \times 00C0H$). This generates an interrupt of type 3, called break point interrupt. This instruction is like a *PAUSE* instruction. *PAUSE* is a temporary stoppage of a running program. It enables a program to do some housekeeping, and return to the instruction after the break point by pressing any key.
4. There are another 80x86 two-byte instruction *INT n*, where *n* represents the type and is the second byte. This means 'generate type *n* interrupt' and processor hardware get the ISR address by computing by the vector address $0x00004 \times n$. When $n = 1$, it represents single-step trap in 8086 and 80x86.
5. There is another 80x86 instruction, which uses a flag called trap and is denoted by *TF*. This flag is at the *FLAG* register and *EFLAG* register of 8086 and 80x86, respectively. This means when *TF* sets (written '1'), automatically after every instruction, the processor action causes an interrupt of type 1 repeatedly. The processor fetches each time the ISR address from the vector address 0x00004 (same as type 1 interrupt address). *INT 1* software instruction will also cause type 1 interrupt once but the *TF* flag set instruction action is identical to the action caused at the end of each instruction after type 1 interrupt.
6. There is instruction in 80196 called *Trap*. It enables debugging of instructions. Till the next instruction after the *Trap* is executed, no interrupt source can interrupt the process and cause diversion to ISR.

SWI-related details in the instruction set help in programming the program diversion to ISR on exception. The exceptional condition if occurs (sets) during execution, causes a diversion to the ISR called *exception handler* or *signal handler* using the software instruction for interrupt in the set.

A programmer can program for the exception on a *queue* (a memory buffer similar to a print buffer) getting full. This is an exceptional run-time condition. It should cause the diversion to routine called *exception handler* function that initiates the appropriate action. *Exceptions* are important routines for handling the run-time errors.

Software instruction-related or software-defined condition-related software interrupts are used in the embedded system. They are essential to design ISRs like error-handling ISRs, software timer-driving ISRs and signalling another routines to run. These interrupts are also called *traps* or *exceptions* or *signals*.

4.4 INTERRUPT SERVICING (HANDLING) MECHANISM

Each system has an interrupt servicing (handling) mechanism. The OS also provides for mechanism for interrupt-handling (Section 8.7).

4.4.1 Interrupt Vector

Interrupt vector is a memory address to which the processor vectors. The processor transfers the program counter to the interrupt vector new address on an interrupt. Using this address, the processor services that interrupt by executing corresponding ISR. The memory addresses for vectoring by the processor are processor- or microcontroller-specific. Vectoring is as per the provisions in interrupt-handling mechanism. The various mechanisms are as follows:

Processor Vectoring to the ISR_ VECTADDR On an interrupt, a processor vectors to a new address, *ISR_ VECTADDR*. It means that the *PC* (program counter), which has the instruction address of next instruction, saves that address on stack or in some CPU register, called link register and the processor loads the *ISR_ VECTADDR* into the *PC*. The stack pointer register of CPU provides the saved address to enable return from the ISR using the stack. When the *PC* saves at the link register it is part of the CPU register set. Section 4.6 will explain the mechanism for saving the CPU registers in detail. The ISR last instruction is *RETI* (return from interrupt) instruction.

A processor provides for one of the following ways of using the ISR_VECTADDR-based addressing mechanism.

Processor Vector Address

1. A system has internal devices like the on-chip timer and A/D converter. In a given microcontroller, each internal device interrupt source or source-group has a separate ISR_VECTADDR address. Each external interrupt pin has separate ISR_VECTADDR. An example is 8051. Figure 4.6(a) shows the ISR_VECTADDRs for the hardware interrupt sources. A very commonly used method is that the internal device (interrupt source or interrupt source group) in the microcontroller autogenerates the corresponding interrupt vector address, ISR_VECTADDR. Thus vector addresses are specific for specific microcontroller or processor with that internal device. An internal hardware signal from the device is sent for the interrupt source or source group.
2. In 80x86 processor architecture, a software instruction, for example, *INT n* explicitly also defines the type of interrupt and the type defines the ISR_VECTADDR. Figure 4.6(b) shows the ISR_VECTADDRs with different vector addresses for different interrupt types. This mechanism results in the handling of *n* number of exception handling routines or ISRs for *n* interrupt types.

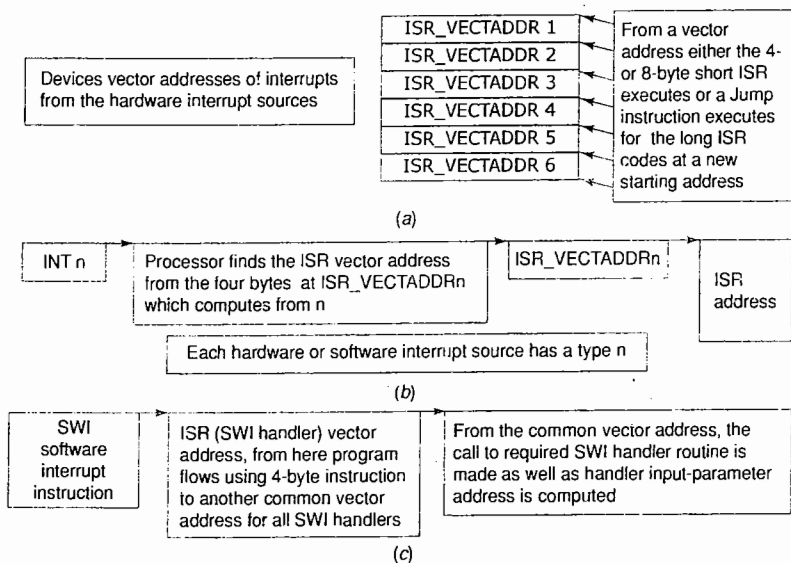


Fig. 4.6 (a) ISR_VECTADDRs for hardware interrupt sources (b) ISR_VECTADDRs with different vector address for different interrupt types using *INT n* instruction (c) The ISR_VECTADDR with common vector addresses for different exceptions, traps and signals using software interrupt instruction SWI

3. In ARM processor architecture, the software instruction SWI does not explicitly define the type of interrupt for generating different vector address and instead there is a common ISR_VECTADDR for each *exception* or *signal* or *trap* generated using SWI instruction. ISR that executes after vectoring has to

find out which exception caused the processor to interrupt and divert the program. Such a mechanism in the processor architecture results in provisioning for the unlimited number of exception handling routines in the system all having the common interrupt vector address. Figure 4.6(c) shows the ISR_VECTADDR with common vector address for all *exceptions*, *traps* and *signals* resulting from SWI.

A group of Interrupt Sources having Common Vector Address A source group in the hardware may have the same ISR_VECTADDR.

Example 4.9

Consider 8051, T1 (transmitter interrupt) and R1 (receiver interrupt) are the sources in the same group having identical ISR_VECTADDR. T1 is an interrupt that is generated when the serial buffer register for transmission completes serial transmission, and R1 is when the buffer receives a byte from the serial receiver. ISR at the ISR_ADDR to which the program jumps or which is called from bytes at the ISR_VECTADDR must first identify the interrupt source (whether T1 or R1) in case of the identical vector address or ISR address for a group of sources. Identification is from a flag in the status register. Setting of a specific status flag in the device flag register enables identification of the interrupt source in the group by the ISR that runs after vectoring.

There are two types of handling mechanisms in processor hardware. The processor-handling mechanism provides for fetching into the PC either (i) the ISR instruction at the ISR_VECTADDR or (ii) the ISR address from the bytes at the ISR_VECTADDR.

1. There are some processors, which use ISR_VECTADDR directly as ISR address and the processor fetches the ISR instruction from there, for example, ARM or 8051. The ARM permits the use of 4-byte instruction for the jump to the ISR (routine for the interrupt servicing). Figure 4.7(a) shows the use of ISR_VECTADDR in ARM for the jump to the routine for the interrupt servicing. The 8051 microcontroller permits the use of short ISR of maximum 8 bytes for the internal devices. The short ISR codes can also use a call instruction to call a detailed routine. Figure 4.7(b) and (c) shows the use of ISR_VECTADDR in 8051 in case of short-code and long-code ISR, respectively.
2. There are some processors, which use ISR_VECTADDR indirectly as ISR address and the processor fetches the ISR address from the bytes saved at the ISR_VECTADDR, for example, 80x86. Figure 4.7(d) shows the use of ISR_VECTADDR address in 8086. Processor of interrupt of type *n* vectors to address $0x00004 \times n$ and fetches 16 bits for sending into IP (instruction pointer register) and another 16 bits for sending into CS (code segment register) The ISR for interrupt will execute from address $0x100000 \times CS + IP$.

Interrupt Vector Table System software designer must provide for specifying the bytes at each ISR_VECTADDR address. The bytes are for either ISR short code [Figure 4.7(b)] or jump instruction to the ISR first instruction [Figure 4.7(a)] or ISR short code with call to the full code of the ISR [Figure 4.7(c)] or for fetching the bytes for finding the ISR address [Figure 4.7(d)].

A table facilitates the service of the multiple interrupting sources for each internal device. Each row of table has an ISR_VECTADDR and the bytes are saved at each ISR_VECTADDR. Vector table location in the memory depends on the processor. It is located at the higher memory addresses, 0xFFC0 to 0xFFFFB in 68HC11. It is at the lowest memory addresses 0x00000 to 0x003FF in 80x86 processor. It starts from the lowest memory addresses 0x00000000 in ARM7. Figure 4.8 shows a vector table in the memory in case of multiple interrupt sources or source groups.

An external device may also send to the processor the ISR_VECTADDR through the data bus (row 2, Table 4.1).

An interrupt vector is an important part of interrupts service mechanism, which associates a processor. The processor first saves the PC and/or other registers of CPU on interrupt and then loads a vector address into the PC. Vector address provides the ISR or ISR address to the processor for an interrupt source or a group of sources or for the given interrupt type. The interrupt vector table is an important part of interrupts service mechanism, which associates the system provisioning for the multiple interrupt sources and source groups.

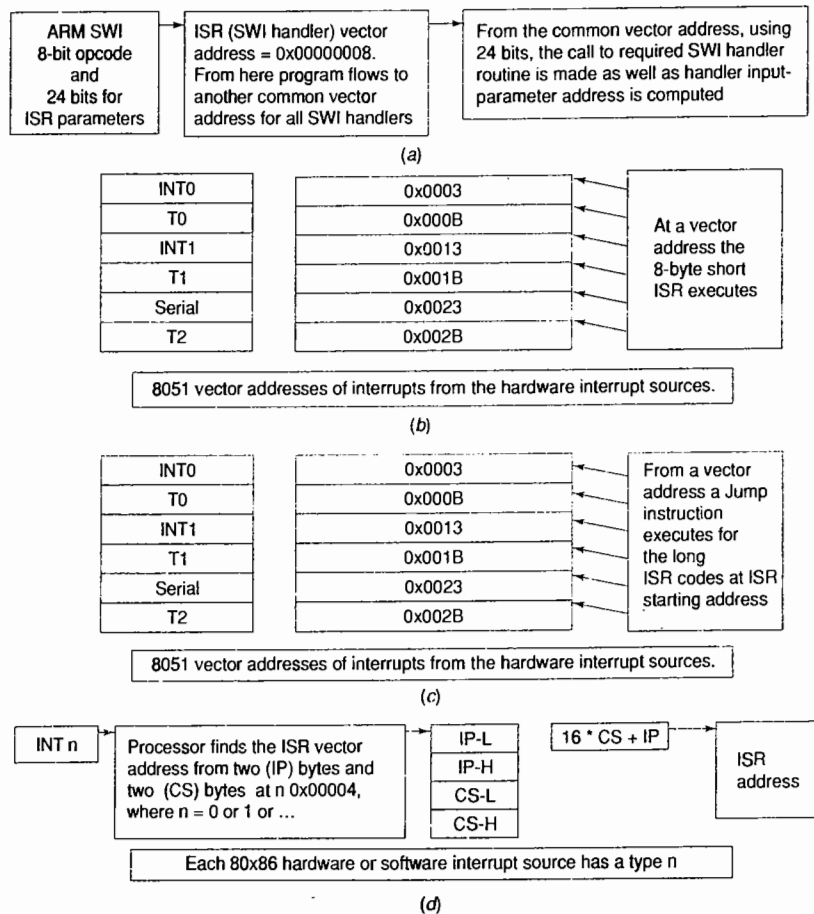


Fig. 4.7 (a) Use of ISR_VECTADDR in ARM for the jump to the routine for the interrupt servicing (b) Use of ISR_VECTADDR in 8051 in case of short-code interrupt service routine (ISR) (c) Use of ISR_VECTADDR in 8051 in case of long-code ISR (d) Use of ISR_VECTADDR address in 80x86 processors

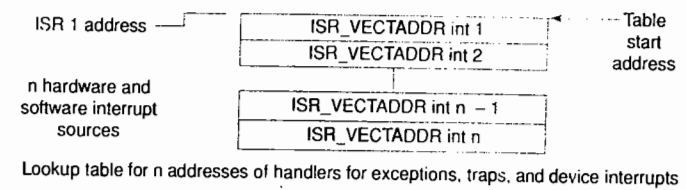


Fig. 4.8 Vector table in memory in case of multiple interrupt sources or source groups

4.4.2 Classification of All Interrupts as Non-Maskable and Maskable Interrupts

Maskable sources of interrupts provide for masking (no diversion) and unmasking the interrupt services (diversion to the ISRs). Execution of ISR for each device interrupt source or source group can be masked or unmasked. An external interrupt request can also be masked. Execution of a software interrupt (trap or exception or signal) can also be masked. Most interrupt sources are maskable. A few specific interrupts cannot be masked. A few specific interrupts can be declared non-maskable within few clock cycles of the processor reset, else that is maskable. There are three types of interrupt sources in a system.

1. *Non-maskable*: Examples are RAM parity error in a PC and error interrupts like division by zero. These must be serviced.
2. *Maskable*: Maskable interrupts are those for which the service may be temporarily disabled to let higher priority ISRs be executed first uninterruptedly.
3. *Non-maskable only when defined so within few clock cycles after reset*: Certain processors like 68HC11 has this provision. For example, an external interrupt pin, XIRQ interrupt, in 68HC11. XIRQ interrupt is non-maskable only when defined so within few clock cycles after 68HC11 is reset.

4.4.3 Enabling (Unmasking) and Disabling (Masking) in Case of Maskable Interrupt Sources

There can be interrupt control bits in devices. There may be one bit EA (enable all), also called the primary-level bit for enabling or disabling the complete interrupt system. When a routine or ISR is executed by the codes in a critical section, an instruction DI (disable interrupts) is executed at the beginning of the critical section and another instruction EI (enable interrupts) is executed at the end of the critical section. DI instruction resets the EA (enable all) bit and EI instruction sets primary level bit denoted by EA (enable all). An example of a critical section code is as follows. Assume that an ISR transfers data to the printer buffer, which is common to the multiple ISRs and functions. No other ISR of the function should transfer the data to the print buffer, else the bytes at the buffer will be from multiple sources. Data shared by several ISRs and routines need to be generated or used by protecting its modification by another ISR or routine.

There may be multiple bits denoted by E_0, \dots, E_{n-1} for n source group of interrupts in case of multiple devices. These bits are called mask bits and are also called secondary-level bits for enabling or disabling specific sources or source-groups in the system. By appropriate instructions in the user software, a write to the primary enable bit and secondary level enable bits (or the opposite of it, mask bits) either all or a part of the total maskable interrupt sources are disabled.

Example 4.10

Consider a system in which there are two timers and each timer has an interrupt control bit. Timer interrupt control bits are ET0 and ET1. Consider a system in which there is an SI device and an interrupt control bit ES, common to serial transmission and serial reception. There is an EA bit to interrupt control for disabling all interrupts.

When EA = 0, no interrupt is recognized and timers as well as SI interrupts service is disabled.

When EA = 1, ET0 = 0, ET1 = 1 and ES = 1, interrupts from timer 1 and SI are enabled and timer 0 interrupt is disabled (masked).

4.4.4 Status Register or Interrupt Pending Register

An identification of a previously occurred interrupt from a source is performed by one of the following:

1. A local-level flag (bit) in a status register, which can hold one or more status flags for the one or several of the interrupt sources or groups of sources.
2. A processor-interrupt service pending flag (boolean variable) in an interrupt-pending register (IPR), that sets by the source (setting by hardware) and auto-resets immediately by the internal hardware when at a later instant, the corresponding source service starts diversion to the corresponding ISR.

Example 4.11

Consider a system in which there are two timers and each timer has a status bit TF0 and TF1. Consider a system in which there is SI device there are the status bits TxEMPTY and RxReady for serial transmission completed and receiver data ready.

1. The ISR_T1 corresponding to timer 1 device reads the status bit TF1 = 1 in the status register to find that timer 1 has overflowed; as soon as the bit is read the TF1 resets to 0.
2. The ISR_T0 corresponding to timer 0 device reads the status bit TF1 = 0 in the status register to find that timer 0 has overflowed; as soon as the bit is read the TF0 resets to 0.
3. The ISR corresponding to the SI device is common for the transmitter and the receiver. The ISR reads the status bits TxEMPTY and RxReady in the status register to find whether a new byte is to be sent to the transmit buffer or whether the byte is to be read from the receiver buffer. As soon as the byte is read the RxReady resets and as soon as the byte is written into the SI for transmission, TxEMPTY resets.

Some processor hardware provide for use of status register bits and some IPR bits. The IPR and status registers differ as follows. The status register is read only. (i) A status register bit (an identification flag) is read only, and is cleared (auto-reset) during the read. An IPR bit either clears (auto-resets) on the service of the corresponding ISR or clears only by a write instruction for resetting the corresponding bit. (ii) An IPR bit can be set by a write instruction as well as by an interrupt occurrence that waits for the service. A status register bit is set by the interrupting source hardware only. (iii) An IPR bit can correspond to a pending interrupt from a group of interrupt sources, but identification flags (bits) are separate for each source among the multiple interrupts.

Properties of the interrupt flags are as follows. A separate flag for every identification of an occurrence from each of the interrupt sources must exist. The flag sets on occurrence of interrupt: (i) It is present either in the internal hardware circuit of processor or in the IPR or in the status register. (ii) It is used for a read by

processor or instruction after a write by the interrupting source hardware. (iii) It resets (becomes inactive) as soon as it is read. This is auto-reset characteristic provided in most hardware designs in order to enable this flag to indicate the next occurrence from same interrupt source. (iv) If set at once, it does not necessarily mean that it will be recognized and serviced later using an ISR. When a mask bit corresponding to that interrupt is set, even if the flag sets, the processor may ignore it unless the mask (or enable) bit modifies later. This makes it possible to prevent an unwanted interrupt from being serviced.

Example 4.12

Consider a touch screen.

It generates an interrupt when a screen position is touched. A status bit b_i is also set. It activates a interrupt request (IRQ). From the status bit, which is set, the interrupting source is recognized among the sources group (multiple sources of interrupts from same device or devices). The ISR_VECTOR_{IRQ} and ISR_{IRQ} are common for all the interrupts at IRQ.

IRQ results in processor vectoring to an ISR_VECTOR_{IRQ} . Using ISR_VECTOR_{IRQ} when the ISR_{IRQ} starts, ISR_{IRQ} instruction reads the status register and discovers that bit b_i as set. It calls for service function (get_touch_position), which reads register R_{pos} for touched screen position information. This action of reading b_i also resets the b_i if the touch screen controller-processing element provides for auto-resetting of b_i . This enables next IRQ interrupt and thus reading next-position on next touch.

4.5 MULTIPLE INTERRUPTS

4.5.1 Multiple Interrupt Calls

When there are multiple interrupt sources, each occurrence of interrupt from a source (or source group) is identifiable from a bit in the status register and/or in the IPR (Section 4.4.4). There can be interrupt service calls in succession case higher priority interrupt sources activate in succession. Then return from high priority ISR is to lower priority pending ISR.

Let us understand two processor interrupt service mechanisms for the case of multiple interrupts.

1. Certain processors do not provide for in-between routine diversion to higher priority interrupts and presume that all interrupts or interrupts of priority greater than the presently running routine are masked till the end of the routine. Figure 4.9(a) shows diversion to higher priority interrupts at the end of the present interrupt service routine only.
2. Certain processors permit in-between routine diversion to higher priority interrupts. Figure 4.9(b) shows the actions in such processors. These processors provide, in order to prevent diversion in-between, a mechanism as follows: There is provisioning for masking of all interrupts by a primary-level bit. These processors also provision selective diversion by provisioning for masking the interrupt service selectively by secondary-level bits (Section 4.4.3).

4.5.2 Hardware Assigned Priorities

There is assigned priority order by hardware. ARM7 provides for two types of external interrupt sources (requests), IRQs and FIQs (fast interrupt requests). 8051 provides for priority order in order of interrupt vector addresses. Lower address has highest and higher has the lower priority. Interrupts in 80x86 are assigned priority order according to interrupt-types. Interrupt of type 0 has highest priority and 255 has lowest assigned priority.

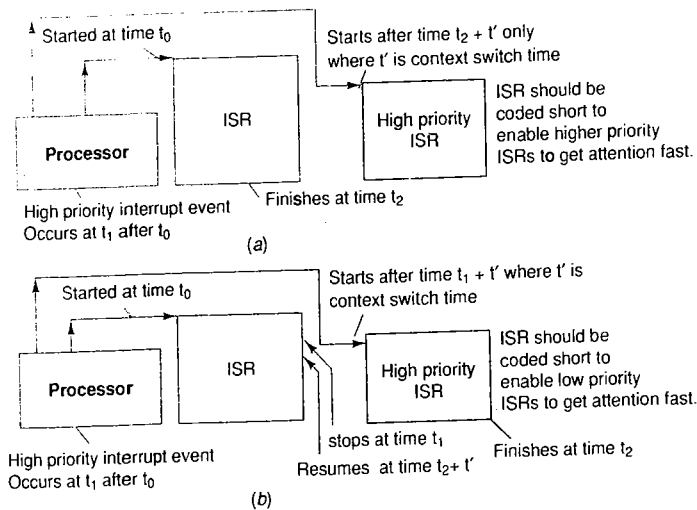


Fig. 4.9 (a) Diversion to higher priority interrupts at the end of the present interrupt service routine only (b) In-between routine diversion to higher priority interrupts unless all interrupts or interrupts of priority greater than the presently running routine are masked

When there are multiple sources of interrupts from the multiple devices, the processor hardware assigns to each source (including traps or exceptions) or source group a presumed priority (or level or type). Let us assume a number, p_{hw} , that represents the hardware-presumed priority for the source (or group). Let the number be among 0, 1, 2, ..., k , ..., $m - 1$. Let $p_{hw} = 0$ mean the highest; $p_{hw} = 1$ next to highest, ..., $p_{hw} = m - 1$ assigned the lowest. Why does the hardware assign the presumed priority? Several interrupts occur at the same time during the execution of a set of instructions, and either all or a few are enabled for service. The service using the source corresponding to the ISRs can only be done in a certain order of priority. (There is only one processor.) Assume that there are seven devices or interrupt source groups. The processor's hardware can assign $p_{hw} = 0, 1, 2, \dots, 6$. The hardware service priorities will be in the order $p_{hw} = 0, 1, 2, \dots, 6$.

Software assigned priorities override these priorities, for example in 8051. Section 4.6.3 will explain this point. Consider the example of the 80x86 family processor. Consider its six interrupt sources: division by zero, single step, NMI (non-maskable interrupt from RAM parity error and so on), break point, overflow and print screen. These interrupts are presumed to be of $p_{hw} = 0, 1, 2, 3, 4$ and 5, respectively. The hardware processor assigns the highest priority for a *division by zero*. This is so because it is an exceptional condition found in user software. The processor assigns the *single stepping* as the next priority as the user enables this source of interrupt because of the need to have a break point at the end of each instruction whenever a debugging software is run. NMI is the next priority because external memory *read* error needs urgent attention. Print screen has the lowest priority.

Which is the Interrupt to be Serviced First among those Pending? Some Way of Polling Resolves this Question. The 8086 has a 'Vectored Priority Polling Method' A processor interrupt mechanism may internally provide for the number of vectors, ISR_VECTADDRs. The *vectored priority' method* means that the interrupt mechanism assigns the ISR_VECTADDR as well as p_{hw} . There is a

call at the end of each instruction cycle (or at the return from an ISR) for a highest priority source among those enabled and pending. Vectored priorities in 80x86 are as per the n_{type} : $n_{type} = 0$ highest priority and $n_{type} = 0xFF (=255)$ lowest priority.

When there are multiple device drivers, traps, exceptions and signals as a result of hardware and software interrupts the assignment of priorities for each source or source group is required so that the ISRs of smaller deadline execute earlier by assigning them higher priorities. Hardware-defined priorities are used as default. Software assigned priorities override these priorities, for example, in 8051.

4.6 CONTEXT AND THE PERIODS FOR CONTEXT SWITCHING, INTERRUPT LATENCY AND DEADLINE

Getting an address (pointer) from where the new function begins, loading that address into the PC and then executing the called function's instructions will change a running function at the CPU to another. Before executing new instructions of the new function the processor or the OS also saves the current program's status word, registers and program contexts. If not done automatically by the processor or the OS, then the new functions, instruction should do that. This is because these (status word and registers) may be needed by the newly called function. *CPU registers including processor status word, registers, stack pointer and program current address in the PC define a function's context*. Figure 4.10(a) shows a current program context. What should exactly constitute the context? It depends on the processor or the operating system supervising the program.

The context must save if a function program or routine left earlier has to run again from the state which was left. When there is a *call* to a function (called *routine* in assembly language, *function* in C and C++, *method* in Java also called task or process or thread when it runs under supervision of the OS), the function or ISR or *exception-handling* function executes by three main steps.

1. Saving all the CPU registers including processor status word, registers and function's current address for next instruction in the PC. Saving the address of the PC onto a stack is required if there is no link register to point to the PC of left instruction of earlier function. Saving facilitates the return from the new function to the previous state.
2. Load new context to switch to a new function.
3. Readjust the contents of stack pointer and execute the new function.

These three actions are known as *context switching*. Figure 4.10(b) shows a current program's context switching to the new context.

The last instruction (action) of any routine or function is always a *return*. The following steps occur during return from the called function.

1. Before return, retrieve the previously saved status word, registers and other context parameters.
2. Retrieve into the PC the saved PC (address) from the stack or link register and load other part of saved context from stack and readjust the contents of stack pointer.
3. Execute the remaining part of the function, which called the new function.

These three actions are also known as *context switching*.

We can say that on interrupt or function call and return the context switches and a new program is executed whenever the *new context* loads into the processor CPU registers. Figure 4.10(c) and (d) shows context switching for new routine and another context switch on return or on in-between call to another routine. Nesting means one function calling the second which in turn calls the third and so on and the return to the calling functions will be in the reverse order. In case of function calls there is nesting and in the case of multiple ISRs because of the presence of multiple interrupts there may or may not no nesting.

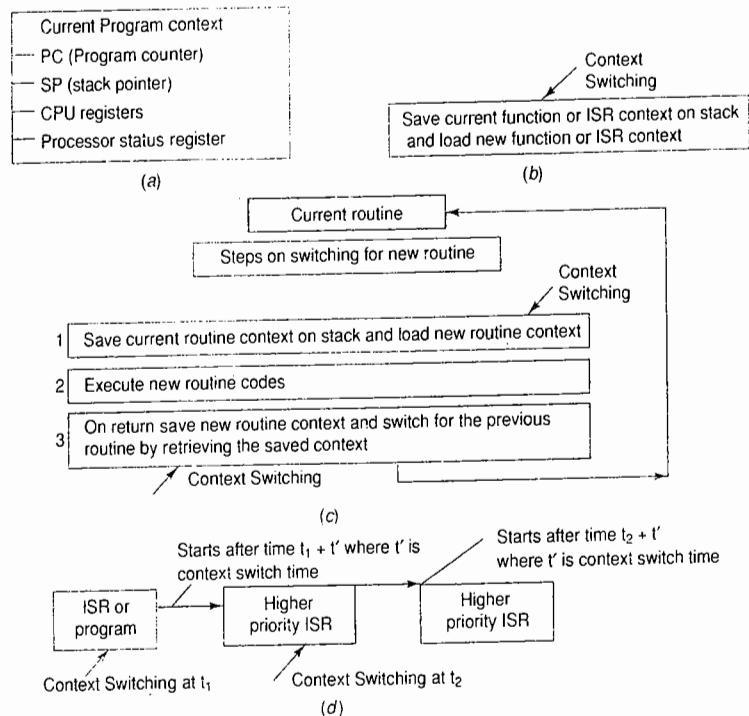


Fig. 4.10 (a) Current program context (b) New program executes with the new context of the called function or routine (c) Context switching for new routine and another switch on the return from routine (d) Context switching for new routine and another switch on return or in-between the call to another routine

Context switching means saving the context of the interrupted routine (or function) and then retrieving or loading the new context of the called routine. Example 4.13 shows how the context switching takes place in the ARM processor.

Example 4.13

Context switching is as follows in the ARM7 processor on *ISR call*. (i) The interrupt mask (disable) flags are set. (Disable low priority interrupts.) (ii) Next instruction PC is saved at link register. (iii) Current program status register (CPSR) copies into the saved program status register (SPSR) and CPSR stores the new status during new instructions. (iv) PC gets the new value as per the interrupt source from the vector table. An *ISR return* context switching back to the previous context is as follows. (i) PC is retrieved from link register. (ii) The corresponding SPSR copies back into the CPSR. (iii) The interrupt mask (disable) flags are reset. (Enable again the earlier disabled low priority interrupts.)

The time taken in context switching, T_{switch} has to be included in a period called *interrupt latency* period, T_{lat} . Example 4.14 shows how to calculate the context switching time period, which is to be accounted in calculating the interrupt latency (Section 4.6.1).

Example 4.14

1. ARM7 processor context switching's minimum period equals two clock cycles plus 0–20 clock cycles for finishing an ongoing instruction plus 0–3 cycles for aborting the data. The 0 cycle when an interrupt occurs just before the end and 3–20 when during an instruction. Longest time taken for an ARM instruction is 20 cycles.
2. During context switching for new routine call or for return, CPSR copies into SPSR on switching from a routine. CPSR means current program status register and SPSR means saved program status register. 3 cycles are taken in switching the CPSR.
3. Two clock cycles are needed for the start of the execution stage of switched routine's first instruction.

Aborting the processor data means CPSR not coping into an SPSR. Then step 2 three cycles are not taken up.

1. Minimum period is thus four (2 + 2) for data abort interrupt. [Steps 1 and 2 above]
2. Maximum is 27 clock cycles (2 + 20 + 3 + 2) for other than data abort interrupt. Maximum is when the interrupt occurs just at the start of execution of the longest time taking instruction in the processor. [Steps 1, 2 and 3 above]

Thus for any latency period calculation, 27 clock cycle periods as context switching time are taken into account when estimating latency in an ARM-based system.

Each running program has a context at an instant. Context reflects a CPU state (PC, stack pointer(s), registers and program state (variables that should not be modified by another routine). Context saving on the call of another ISR or task or routine is essential before switching to another context.

4.6.1 Interrupt Latency

When an interrupt occurs the service of the interrupt by executing the ISR may not start immediately by context switching. The interval between the occurrence of an interrupt and start of execution of the ISR is called *interrupt latency*.

1. When the interrupt service starts immediately on context switching the interrupt latency T_{switch} equals the context switching period. When instructions in a processor take variable clock cycles, maximum clock cycles for an instruction are taken into account for calculating the latency. Figure 4.11(a) shows latency in case the interrupt service starts immediately.
2. When the interrupt service does not start immediately but context switching starts after all the ISRs corresponding to the higher priority interrupts complete the execution. If the sum of time intervals for completing the higher priority ISRs equals ΣT_{exec} , then interrupt latency equals $T_{\text{switch}} + \Sigma T_{\text{exec}}$. Figure 4.11(b) shows latency in case the interrupt service starts after present ISR of higher priority interrupt completes the execution.
3. We disable the interrupt system when a routine enters a critical section and enable the interrupts when the routine exits the critical section codes. A routine of function or ISR may consist of codes for critical region instructions and before the critical section codes all the interrupts are disabled and enabled by the

end of the critical section. T_{disable} may or may not be included depending on the programmer's approach. Let T_{disable} be the period for which a routine is disabled in its critical section. The interrupt service latency from the routine with the interrupt-disabling instruction (because of the presence of the routine with critical section) for an interrupt source will be $T_{\text{switch}} + \sum T_{\text{exec}} + T_{\text{disable}}$. Figure 4.11(c) shows interrupt latency as sum of the periods for T_{switch} , $\sum T_{\text{exec}}$ and T_{disable} when the presently running routine to be interrupted is executing critical section codes.

Worst case latency is sum of the periods T_{switch} , $\sum T_{\text{exec}}$ and T_{disable} where the sum is for the interrupts of higher priorities only. Minimum latency is the sum of the periods T_{switch} and T_{disable} when the interrupt is of the highest priority. For latency computations, worst case is taken into account.

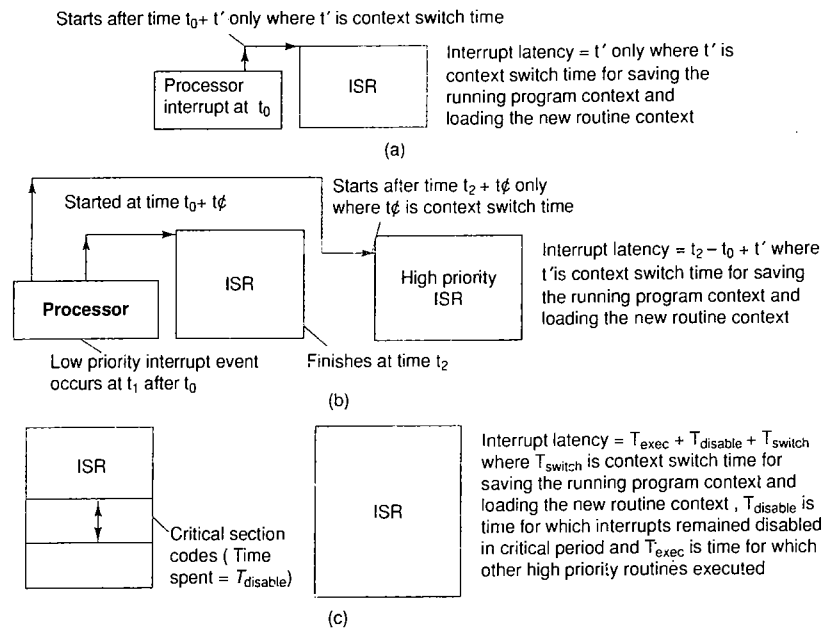


Fig. 4.11 (a) Latency in case the interrupt service starts immediately (b) Latency in case the interrupt service starts only after the interrupt service routine presently running completes execution (c) Interrupt latency as sum of the periods for T_{switch} , $\sum T_{\text{exec}}$ and T_{disable} when the presently running low priority routine to be interrupted is having critical section codes

Example 4.15

80196 microcontroller has an SI device which has a FIFO (first in first out) buffer and the SI reads the bytes and puts it in the buffer. SI generates three interrupts: RI on one byte reception, FIFO_4thEntry interrupt when FIFO is half full and FIFO_Full interrupt when the FIFO is full. Assume that a microcontroller has two devices: SI similar to 80196 and timer T. SI has a serial input buffer of 8 bytes (a FIFO of 8 bytes

0 – 7). Assume that a serial input at the SI reads a byte from a network and generates three types of interrupts. T generates timer-overflow and timer-capture interrupts, called TF and TCAPTURE interrupts. Worst-case interrupt latencies are as follows.

1. When the seventh byte is received, the controller generates interrupt FIFO_FULL and a FIFO_FULL flag sets in S0. Assume that it is the top priority serial interrupt and the ISR execution time is T_{exec} (FIFO_FULL). For the FIFO_FULL interrupt, the interrupt latency is $T_{\text{switch}} + T_{\text{disable}}$ because it is a top priority ISR.
2. When the zeroth byte is received, the SI generates an interrupt RI and an RI flag sets in the status register S0. Assume RI has the lowest priority serial interrupt and RI ISR execution time is T_{exec} (RI). For the RI interrupt, the interrupt latency is $T_{\text{switch}} + T_{\text{exec}}$ (TCAPTURE) + T_{exec} (TF) + T_{disable} because it has the lowest priority than the timer interrupts.
3. When the third byte is received, the SI generates an interrupt FIFO_4thEntry and a FIFO_Half flag sets in S0. Assume that it is the middle priority serial interrupt and has priorities lower than the TCAPTURE interrupt but higher than the timer overflow. Assume that the ISR execution time is T_{exec} (FIFO_Half). For FIFO_4thEntry interrupt, the interrupt latency is $T_{\text{switch}} + T_{\text{exec}}$ (TCAPTURE) + T_{disable} because it has higher priority than timer overflow but it has lower priority than TCAPTURE. The T_{exec} (RI) is not taken into account because if RI is not responded then only FIFO_Half interrupt occurs. Both interrupts RI and FIFO_Half belong to the same SI device.

Each running program when interrupts, the interrupting source service routine takes some time before starting the servicing codes. That time interval is called interrupt latency. It is the sum of the execution time of higher priority interrupts and the context switching period. If an interrupted routine is having a critical section (interrupts disabled), the interrupt latency increases by period equal to the interrupts disabled period.

4.6.2 Interrupt Service Deadline

For every source, the service of its ISR instructions can be kept pending up to a maximum period. This period defines the deadline during which the service must be completed. It should not be less than the worst-case interrupt latency. Figure 4.12(a) shows interrupt latency period and deadline for an interrupt.

A 16-bit timer device on overflow raises TF interrupt on transition of counts from 0xFFFF to 0x0000. It has to be responded by executing an ISR for TF before the next overflow of the timer occurs, else the counting period between 0x0000 after overflow and 0x0000 after the next-to-next overflow will not be accounted. The timer counts increment every 1 μ s; the interrupt service deadline is 65536 μ s.

Video frames in video conferencing reach after every 1 ÷ 15s. The device on getting the frame interrupts the system and the interrupt service deadline is 1 ÷ 15s, else the next frame will be missed.

Example 4.15 FIFO_Full interrupt must be executed fast as it has shorter deadline compared with RI and the fourth entry interrupt. If ISR for FIFO_Full interrupt does not execute before the next character at the SI device, the character will be missed. If ISR for FIFO_4th entry interrupt does not execute fast, it does not matter, because eventually there is a cushion of SI raising the FIFO_4th entry interrupt as well as FIFO_Full interrupt. FIFO_Full interrupt is said to have a service interrupt service deadline. If SI device is receiving characters at 64 kbps and in 11-bit UART format, the FIFO_Full interrupt service deadline is 171.9 μ s. FIFO_RI interrupt service deadline is 171.9 μ s if SI device does not have the buffer and provisions for FIFO_Half and FIFO_full interrupts.

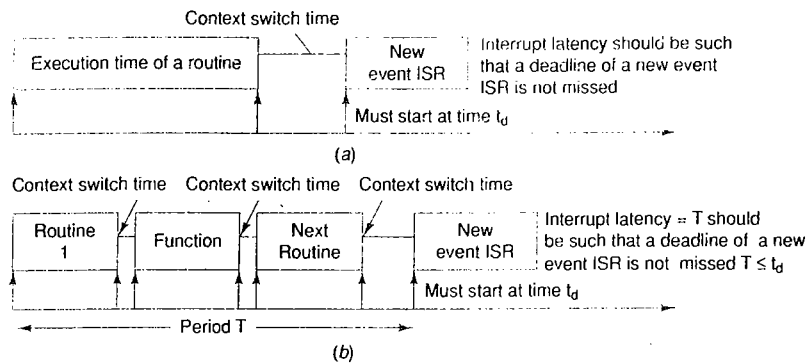


Fig. 4.12 (a) Interrupt latency period and deadline for an interrupt (b) Short interrupt service routines (ISR) and functions, which run at later instances so that the other ISR deadlines are not missed

A good software design principle for multiple interrupt sources is to keep the ISR as short as possible. Why? This is to service the in-between pending interrupts and leave the functions that can be executed afterwards for a later time. When this principle is not adhered to, a specific interrupting source may not be serviced within the deadline (maximum permissible pending time). Section 4.2.3 described use of interrupt service threads, which are the second-level interrupt handlers. Figure 4.12(b) shows a short ISR and functions, which run at later instances so that the other ISR deadlines are not missed.

The system therefore has to meet the deadlines set for service of each system device. This can be understood by the following examples. Consider the example of a video system. When the system is running, two device-driver ISRs also run. One driver is for the voice device and the other for the image device. The ISRs and the other system software design for these two device drivers have to maintain synchronization else the next set of images and the next set of voice signals will be missed.

Therefore, the system software designer designs the appropriate ISRs for multiple device interrupts so that all device interrupt calls are serviced within the stipulated deadlines of each interrupt. The design should provide optimum latencies and set appropriate deadlines for each service routine and functions.

Each ISR may have an interrupt service deadline when interrupts. An ISR with a deadline must have interrupt latency less than the deadline.

4.6.3 Software Over-riding of Hardware Priorities to Meet Service Deadlines

Which source or source group has higher priority with respect to the others that is first decided among the ISRs that have been assigned higher priority in the user software. If user-assigned priorities are equal then the highest priority is that which is preassigned at the processor internal hardware. The 8051 internal interrupt mechanism is as follows. There is the interrupt priority (IP) register at 8051 in which there are five priority bits for the five interrupt sources in 8051. Also there are the five interrupt-enable bits in the IE register. These are secondary-level enable bits of the processor's service of ISRs. When a

priority bit at IP is set, the corresponding interrupt source gets a high priority, and if reset, it gets a lower priority. The 8051 first selects by polling among the high priority according to the bits at IP register.

There is a need for over-riding the priority order by assigning priorities. The need of reassigning priorities over hardware pre-assigned priorities can be understood from the following example.

Example 4.16

Assume that there are two sources of interrupts: serial port input and A/D conversion. A/D conversion time is 200 μ s and SI device with no data buffer receiving inputs at 64 kbps with minimum separation between the characters equals 171.9 μ s. The A/D conversion should therefore have the lower priority than RI interrupts of SI. When the system hardware has the internal devices, it assigns lower priority to the A/D end of the conversion interrupt. Suppose that the SI device is used to receive input at 16 kbps and assume that the UART mode has 11 bits per character. When the A/D conversion is needed continuously (e.g., when ECG signals are input), the software should assign the higher priority to A/D, because SI receives character every 11/16 ms = 687 μ s and A/D every 200 μ s, at a rate faster than the SI.

Software-assigned priorities can be used to over-ride the hardware priorities. OS provides the functions, which assign the software priorities to each ISR, IST and task of the real-time system.

4.7 CLASSIFICATION OF PROCESSORS INTERRUPT SERVICE MECHANISM FROM CONTEXT-SAVING ANGLE

1. The 8051 interrupt service mechanism is such that on occurrence of an interrupt service, the processor pushes the processor registers PCH (program counter higher byte) and PCL (program counter lower byte) onto the memory stack. The 8051 family processors do not save the context of the program (other than the absolutely essential PC) and a context can save only by using the specific set of instructions in the called routine. For example, using push instructions. It speeds up the start of ISR and return from ISR but at a cost. The onus of context saving is on the programmer in case the context (SP and CPU registers other than PCL and PCH) is to be modified on service or on function calls during execution of the remaining ISR instructions.
2. The 68HC11 interrupt mechanism is such that processor registers save onto the stack whenever an interrupt service occurs. These are in the order of PCL, PCH, IYH, IYH, IXL, IXH, ACCA, ACCB and CCR. The 68HC11 thus does automatically save the processor context of the program without being so instructed in the user program. As context saving takes processor time, it slows a little the start of ISR and return from the ISR but at the great advantage that the onus of context saving is not on the programmer and there is no risk in case the context modifies on service or function calls.
3. Certain processor provides for fast context switching two stack frames with each stack frame consisting of the same number of registers, for example, 16 or 32 registers. The PC, stack-pointer and link-register define one stack frame. When context switches from one routine to another, only the pointer to the stack frame changes. The ISR stack frame that is called has the current program context and the interrupted program context becomes the saved program context. ARM7 provides such a mechanism. Certain processors provide for more than two stack frames with each stacking a context.

The OS program also provides for memory blocks, which are used as multiple stack frames for the tasks (processes or threads). This enables multi-threading and multi-tasking.

Certain processors provide for saving only the PC. Certain processors provide for saving only the PC and other CPU registers before calling the ISR and context switching. Certain processors provide for fast context switching by providing internal register frames for the stack or providing sets of local (internal) stack for the contexts. Fast context switching reduces the interrupt latencies and enables the meeting of each function or routine deadline for service. The operating system provides for multiple stack frames to enable multitasking and context switching using the multiple stack frames.

4.8 DIRECT MEMORY ACCESS

Assume that the data transfer is to occur between hard disk system memory. The DMA is used in that case. When the I/Os are needed for large amount of data from a peripheral device to the memory addresses in the system or large amount of data is to be transferred by the I/Os, the interrupt-based mechanism is not suitable.

A DMA facilitates a multi-byte data set or a burst of data or a block of data transfer between the external device and system or between two systems. A device that facilitates DMA transfer has a processing element (single purpose processor). The device is called DMAC (DMA Controller). Data transfer occurs efficiently between the I/O devices and system memory with the least processor intervention when using DMAC. The system address and data buses become unavailable to processor and available to the IO device that interconnects using DMAC during the data transfer. Figure 4.13 shows the interconnections using the DMAC. It also shows the buses and control signals between the processor, memory, DMAC and data-transferring I/O device.

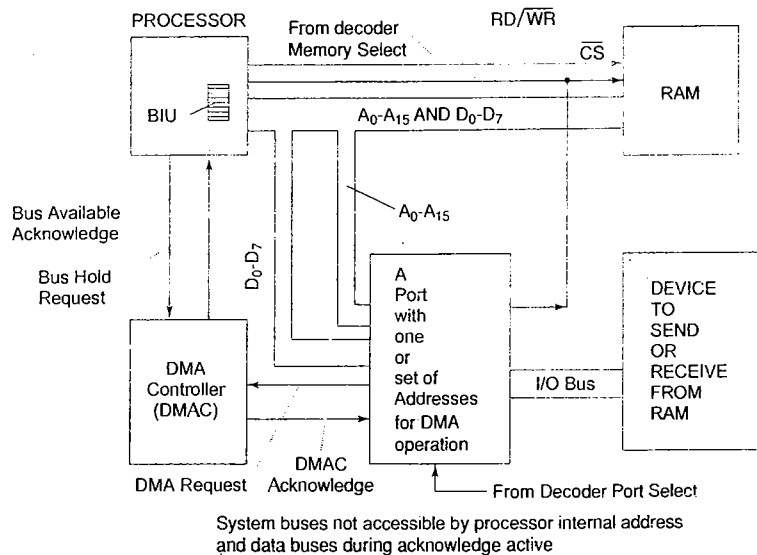


Fig. 4.13 The buses and control signals between the processor, memory, DMA controller and data-transferring I/O device

The DMAC sends a hold request to the CPU and the CPU acknowledges that if the system memory buses are free to use. Three modes are usually supported in DMA operations. (i) Single transfer at a time and then

release IO bus hold on the system bus after each transfer. (ii) Burst transfer at a time and then release of the IO bus hold on the system bus. A burst may be of a few kilobytes. (iii) Bulk transfer and then release of the IO bus hold on the system bus only after the transfer is completed.

4.8.1 Use of DMAC

Whenever a DMA request is made to the DMAC for the I/Os, the DMAC is first initialized. It is programmed for (i) read or write, (ii) mode (bytes, burst or bulk) of DMA transfer, (iii) total number of bytes to be transferred and (iv) starting memory address. Consider a read operation (external device to memory transfer). DMA proceeds without the intervention of the CPU, except (i) at the start of DMAC programming and initializing and (ii) at the end. Whenever a DMA request by the external device is made to the DMAC, the CPU is requested the DMA transfer by DMAC at the start to initiate the DMA and at the end to notify the end of the DMA by DMAC. Example 4.16 explains the data transfer operation.

Example 4.17

Assume that 2 kb of data needs to be transferred. One method is that device interrupts the processor, when 1 or 4 or 8 bytes of data are ready and generate the interrupt. The ISR reads the 1 or 4 or 8 bytes and put these into the memory addresses. Assume that the device generates the interrupt and transfers the 8 bytes. Number of interrupts required will be $2 \text{ k}/8 = 2048/8 = 256$ and ISR has to be run 256 times. A DMA is the better approach.

An IO program initializes the DMAC for 2 kb burst mode transfer from a memory address for the I/O to an external device starting from memory address M_1 . DMAC loads 2048 in a data count register and loads M_1 in address register on initialization.

On an external device requesting the DMA, the DMAC sends HOLD request signal to the processor. Processor acknowledges by the HLDA (hold acknowledge) signal that when the system buses are not in use.

DMAC transfers the bytes from I/O bus to the memory bus in burst from IO bus to the memory bus D₀-D₇ lines and keeps track of the data counts in the DC (data count) register. Transfer takes place to addresses from M_1 to $M_1 + 2047$. DC = 0 after the transfer completes.

DMAC interrupts the processor so that the processor is notified at the end of DMA transfer and an ISR can re-initialize the DMAC for the next transfer.

A DMAC may also provide memory access to multiple channels. A multi-channel DMAC provides DMA action from system memories and two (or more) IO devices. There is a separate set of registers for programming each channel. There may be the separate or common interrupt signals in the case of multi-channel DMAC.

The 80x86 processors do not have on-chip DMAC units. The 8051 family member 83C152JA (and its sister JB, JC and JD versions) have two DMA channels on-chip. The 80196KC has a PTS (peripheral transactions server) that supports DMA functions. (Only single and bulk transfer modes are supported, not the burst transfer mode.) The MC68340 microcontroller has two on-chip DMA channels. 80960CA has four-channel DMAC on chip, with a mode called demand transfer mode also provided.

On-chip or a separate DMAC facilitates fast direct byte transfers between memory and I/O devices compared with interrupt-driven data transfer as that has in-built processing element and uses the system buses as and when they are made available by the processor. Designers can use DMAC in sophisticated systems so that the system performance improves by separate processing of bulk or burst data transfer from and to the peripherals.

4.8.2 Use of DMA Channel in Case of Multiple Interrupts in Quick Succession from the Same Source

A good feature of DMA-based data transfer service is very small latency periods compared with data transfer using multiple IO interrupt sources and multi-byte bulk or burst data transfers. The interrupt service routine period from start to end can now be very small as the ISR that initiates the DMA to the interrupting source, simply programs the DMA registers for the command, data count, memory block address and I/O bus start address (Section 4.8.1).

The use of DMA channels for the IO services in place of processor interrupt-driven ISRs provides an efficient method when the device has to transfer large amount of data by I/O. This is because a DMA transfer uses the periods when the system buses are free.

4.9 DEVICE DRIVER PROGRAMMING

A system has number of physical devices (Chapter 3). A device may have multiple functions. Each device function requires a driver. Examples of multiple functions in a device are as follows.

1. A timer device performs timing functions as well as counting functions. It also performs the delay function and periodic system calls.
2. A *transceiver* device transmits as well as receives. It may not be just a repeater. It may also do the *jabber control* and *collision control*. (Jabber control means prevention of continuous streams of unnecessary bytes in case of system fault. Collision control means that it must first sense the network bus availability then only transmit.)
3. Voice-data-fax modem device has transmitting as well as receiving functions for voice, fax as well as data.

A common driver or separate drivers for each device function are required. Device drivers and their corresponding ISRs are the important routines in most systems. The driver has following features.

1. *The driver provides a software layer (Interface) between the application and actual device:* When running an application, the devices are used. A driver provides a routine that facilitates the use of a device function in the application. For example, an application for mailing generates a stream of bytes. These are to be sent through a network driver card after packing the stream messages as per the protocol used in the various layers, for example, TCP/IP. The network driver routine will provide the software layer between the application and network for using the network interface card (device).
2. *The driver facilitates the use of a device by executing an ISR:* The driver function is usually written in such a manner that it can be used like a black box by an application developer. Simple commands from a task or function can then drive the device. Once a driver function is available for writing the codes, the application developer does not need to know anything about the mechanism, addresses, registers, bits and flags used by the device. For example, consider a case when the system clock is to be set to tick every 10,000 μs (100 times each second). The user application simply makes a call to an OS function like OS_Ticks (100). It is not necessary for the user of this function to know which timer device will perform it. What are the addresses, which will be used by the driver? Which will be the device register where value 100 registers for the ticks? What are the control bits that will be set or reset? OS_Ticks (100) when run, simply interrupts the system and executes the SWI instruction which calls the signalled routine (driver ISR) for the system ticking device. Then the driver ISR which executes takes 100 as *input* and

configures the real time clock (Section 3.8) to let the system clock tick each 10,000 μs and generate the system clock interrupts continuously every 10,000 μs to get 100 ticks each second.

Generic device driver functions in high level language are used in high level language program. The functions are open, close, read, write, listen, accept etc.

Device driver-ISR programming in assembly needs an understanding of the processor, system and IO buses and the addresses of the device registers in the specific hardware. It needs in-depth understanding of how the software application program will seek the device data or write into the device data and what is the platform. Platform means the operating system and hardware, which interfaces with the system buses.

A common method of using the drivers is as follows: a device (or device function module) is opened (or registered or attached) before using the driver. If means device is first initialized and configured by setting and resetting the control bits of device control register and use of the interrupt service is enabled. Using a user function or an OS function, a device (or device function module) can also be closed or de-registered or detached by another process. After executing that process, the device driver is not accessible till the device is re-opened (re-registered or re-attached).

4.9.1 Writing Physical Device-Driving ISRs in a System

For writing the software for driver in assembly, the following points must be clear.

1. Information about how the device communicates.
2. Information about the three sets of device registers—data registers or buffers, control registers and status registers. A device *initializes* (configures, registers, attaches) by setting the control register bits. A device *closes* (resets, de-registers, detaches) by resetting the control register bits. (Example 4.18)
3. Information of other registers and common addresses to a device register.
4. Control register bits control all actions of the device. A control bit can even control which address corresponds to which data register at an instant. For example, at the instance when the DLAB control bit is set, the 0x2F8 corresponds to the divisor-latch lower byte (Example 4.19).
5. Status register bits reflect the flags for status of the device at an instant and change after performing actions as per the device driver. A status flag at a status register reflects the present status of device. For example, an instance between finishing the transmission of bits from a TRH buffer register and obtaining the new bits for next transmission, a transmitter empty flag (TDRE) reflects it (Example 4.19).
6. Either setting of an enable bit (interrupt control flag) is used by the system to initiate a call for executing an ISR related to the device driver function. ISR executes if: (i) it is enabled (not masked at the system) and (ii) the interrupt system itself is also enabled.

The following information must therefore be available when writing a device control and configuring and driver codes.

1. *Addresses for each register:* Physical device hardware and its interfacing circuit fix the addresses for a physical device and they usually cannot be relocated. The device becomes the *owner* of these addresses. For example, IBM PC hardware is designed such that the device addresses are as following:
 - a. *Timer* addresses between 0x0040-5F;
 - b. *Keyboard* addresses between 0x00600-6FD, real-time clock (system clock) addresses between 0x0070-7F;
 - c. *Serial COM port 2* addresses between 0x02F8-2F and *serial COM port 1* addresses between 0x03F8-3F.
2. There may be input-buffer register as well as output-buffer register at a common address. This is because during device write and read instructions at the control bus the different signals RD and WR

are issued. The physical device can thus select the appropriate register when taking action. For example, there is a register SBUF at 8051. It addresses both the output serial buffer and input serial buffer.

3. There may be multiple registers at the same address. Refer Example 4.19. This example shows the following. RBR (receiver data buffer register) and TRH (transmitter holding register) are at the same address (0x2F8) in PC COM2 serial device. This address is also common for the lower byte of divisor latch, which is used for presetting the device baud rate. A control bit is made 1 to write this byte when setting the device baud rate and later it is made 0 for using the same address as RBR and TRH during the device 'read' and 'write' instructions, respectively.
4. Purpose of each bit of the control register.
5. Purpose of each status flag in the status register. Which status bit when set and reflects a device interrupt, calls to which ISR.
6. Whether control bits and status flags are at the same address. The processor reads the status from this address during the read instructions. The processor writes the control bits at that address during the write instructions.
7. Whether both, control bits and flags coexist in the same register.
8. Whether the status flag, which sets on a device interrupt, auto-resets on executing the ISR or if an ISR instruction should reset.
9. Whether control bits need to be changed, reset or set again before return to the interrupted process.
10. List of actions required by the driver at the data buffers, control registers and status registers.

Section 8.6.1 will describe in detail the device management functions at an OS. The OS usually provides device-related functions so that for the new device also the drivers are written in an identical manner. For example, Unix device driver components are: (i) device ISR, (ii) device initialization codes (codes for configuring device control registers) and (iii) system initialization codes, which run just after the system resets (at bootstrapping). Microsoft OS Windows provides the Windows driver functions (WDF) and user-mode driver framework (UDMF). Linux provides device drivers (Section 4.9.6). Using object-oriented programming approach, *Class drivers* are also written for operation on large number of similar type of devices using identical bus or network protocol, for example, printers or CD drives or class drivers for the USB-based devices.

When a device driver function such as read or write or open is called, the OS first initiates the logical layer part. The logical layer then initiates the physical layer, which implements OS device function using driver ISR functions written in assembly so that the device hardware performs the actions accordingly. Similarly the device sends the response of the commands to the logical layer of the driver through the physical layer.

The device drivers execute according to the device hardware, interrupt service mechanism, OS, system and IO buses. A device driving ISR is designed using the device addresses and three sets of device registers—data register(s) or buffer(s), control register(s) and status register(s). A device is configured and controlled by the control bits. The driver ISR initiates and executes on status flag change. A list of actions required by the driver at the data buffers, control registers and status registers is needed and is prepared before writing the driver codes. The driver codes are sensitive to the processor and memory. This is because: (i) when the device addresses change, the program should also be modified and (ii) when a processor changes, the interrupt service mechanism changes. The OS usually provides device drivers for the system devices.

4.9.2 Virtual Device Drivers

Virtual device drivers emulate the device hardware, for example, hard disk and generate software interrupts similar to physical device drivers. The file and pipe are two examples of virtual devices.

Both virtual devices and physical device drivers have functions for device *open*, *read*, *write* and *close*. Consider the analogies of a file device with a physical device. (i) Just as a *file* needs to be *opened* to enable

read and write operations, a device may need to be sent an *interrupt call* for initializing and configuring it (opening, registering or attaching it. Setting control bits appropriately does this). (ii) Just as a file is sent a *read call*, a device must be sent another *interrupt call* when its input buffer(s) is to be read. (iii) Just as a file is sent a *write call*, a device needs to be sent another *interrupt call* when its output buffer is to be written. (iv) Just as a file is sent a *close-call* a device needs to be sent another *interrupt call* to *disable* (close or deregister or detach) it from the system for further read and write operations.

The concept of virtual (software) device drivers is very important in programming. Examples are as follows.

1. A memory block can have data buffers in analogy to buffers at an IO device and can be accessed from a *char* driver or a *block* driver. The device is called the *char* device or the *block* device when it can access a character or a block of characters, respectively.
2. A physical device transceiver (with input–output block buffer) or repeater is equivalent to a virtual device called *loop back device*. It stores allocated memory blocks using a block device driver and returns the data back from the memory.
3. A bounded buffer device in memory can be like a printer buffer. A data stream is sent by one routine (driver) and read by another routine (driver). Bounded buffer device is a virtual device, usually called *pipe* device.
4. A program can store in a set of memory blocks called *RAM disk* in the analogous way a file system does at the hard disk. RAM disk is a device that consists of multiple internal file devices.

The virtual device is an innovative concept for system software design. Drivers for these are also written like the physical device drivers. Important devices are char device, block device, loop back device, file device, pipe, socket and RAM disk. Device configuring is equivalent to creating a file. Device activation on the interrupt is equivalent to opening a file. Device resetting is equivalent to closing a file. Device detaching is equivalent to freeing the memory space allotted for a file data.

4.9.3 Parallel Port Drivers in a System

Device driver *read ()* function can be implemented by calling an ISR is *Port_ISR_Input*, which handle the port input. Figure 4.14(a) shows control and status bits used in the ISRs in the device drivers and port pins interface with the data bus. Figure 4.14(b) shows step *A* for port initialization, step *B* for calling the driver and steps 0 to 5 for driver *Port_ISR_Input*. The driver reads byte from port and puts it into a queue that builds in memory on successive inputs to the port.

Port_ISR_Input does the following:

1. Step *A* sets the device control bit for read. Step *B* is no action till input event.
2. Steps 0 to 2 are for reading the input buffer(s) by emptying the buffer and storing the byte(s) in memory or using the bytes received as per the system requirement.
3. Step 3 resets the device receive-buffer ready flag (in status register) and thus prepares the device for the next read after step 4. In step 4, interrupt flag resets to enable next byte read on next interrupt.

An example for device driver *write ()* function is a driver ISR for handling the port outputs. The ISR does the following:

1. Sets the device control bit for write.
2. Sends into the device output buffer (s) the byte(s) from the memory.
3. Resets the device-transmit buffer-empty flag (in status register) on completion of transmission of the byte(s) and prepare the device for the next write.

Example 4.18 gives a device driver ISR example using 68HC11 microcontroller port C (68HC11 microcontroller knowledge is presumed here).

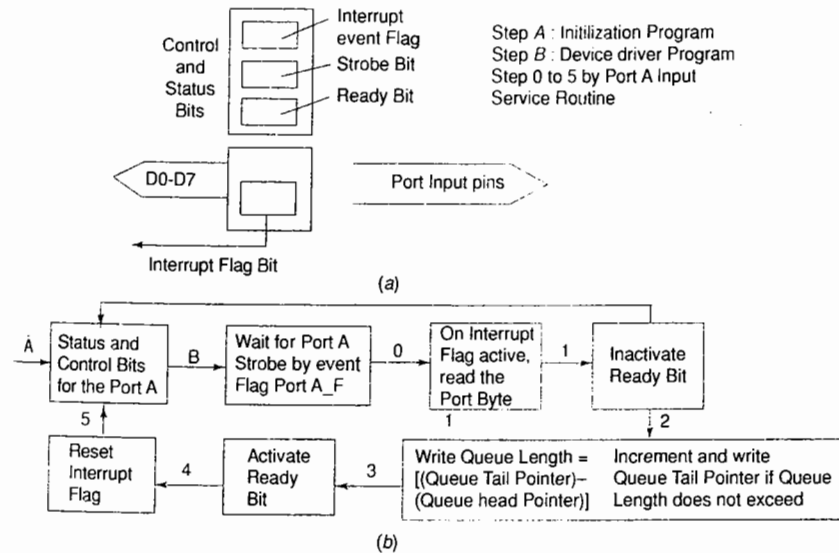


Fig. 4.14 (a) Control and status bits used in the interrupt service routines (ISRs) called by the device drivers and port pins used to interface the data bus (b) Step A for initialization, step B for the interrupt of the driver and steps 0 to 5 for driver Port_ISR_Input execution. The driver reads a byte from a port and puts it into a queue that builds in memory on successive inputs to the port

Example 4.18

Device driver *read* () function calls an ISR PortC_ISR for handling the port C inputs in 68HC11. Port C uses the hand-shaking signals. Figure 4.15(a) shows hand-shaking signal to port C. Figure 4.15(b) shows control and status bits used in the call to the driver. Figure 4.15(c) shows port C as input and its interface with the data bus. Figure 4.15(d) shows port C as output. Figure 4.15(e) shows step A for initialization, step B of the interrupt and step C for executing PortC_ISR. The ISR reads from the port and inserts the byte into a queue. The latter builds in memory on successive inputs to the port. An external peripheral activates STRA pin. The peripheral requests a transfer of its byte to port C through the STRA. When STRA pin activates by '0', the port C gives an acknowledgment in case STAI (STRA interrupt mask bit) at a control register is not set (STAI is not at '1'). STRB pin sends hardware signal for the ready status (or acknowledgement) from the port C to the peripheral. When STAI is programmed to '0', the peripheral puts the byte into the port buffer as soon as STRB pin sends acknowledgement. As soon as the peripheral completes by putting the byte at the port C, the STAF sets (=0). STAF is at status register. STAF is the interrupt flag, which sets when the external device completes putting the byte at the port C.

Port C memory address is 0xp003, when page address configured on 68HC11 is 0xp000 (p is 4-bit maximum-significant nibble). A call to the device driver ISR for port C device *open* (), three actions occur by the device initialization program. (i) Define port C address as follows. # define PortC 0x1003 /* p bits as 0001 */. (ii) Reset all eight bits to 0s at DDRC so that port C becomes an input parallel port. DDRC is data direction

register for port C at memory address 0xp007. (iii) On initialization call, STAI sets to '1' for enabling interrupting by the peripheral, which connects to port C. STAI is the sixth bit of PIOC (port I/O control register). It is at memory address 0xp002.

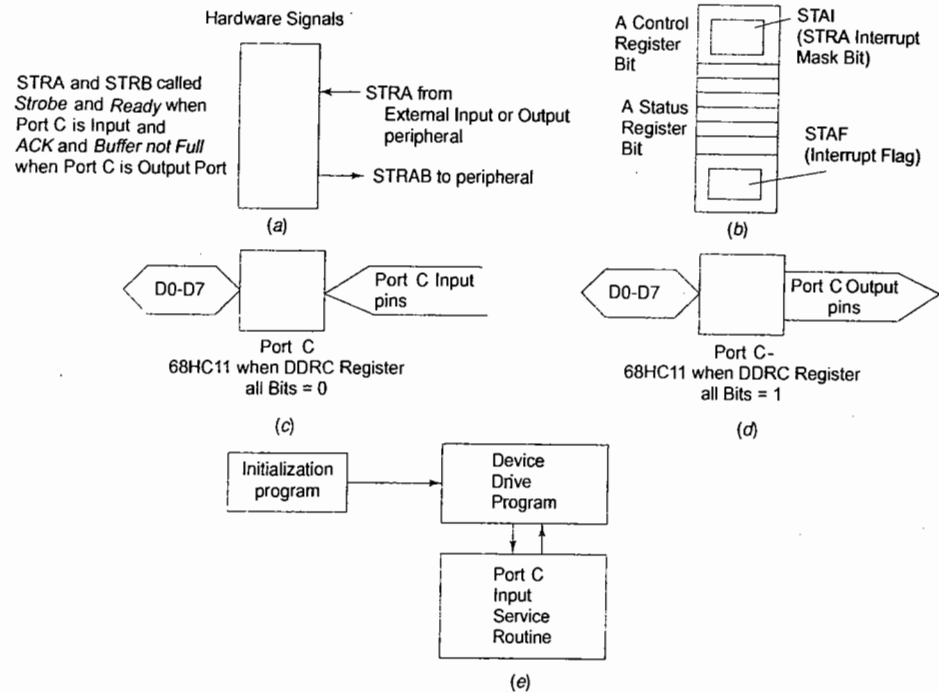


Fig. 4.15 (a) Hand-shaking signal to a parallel port (b) Control and status bits used in the system calls by driver functions (c) Port C as input and its interface with data bus (d) Port C as output (e) Step A for initialization, step B for system call to the driver and step C for driver PortC_ISR

A driver ISR program for Port C read will execute after the following actions.

1. If STAI is set '0' then read STAF. (STAF is the seventh bit at PIOC. PIOC also provides the status bits. It is for control cum status bits.)
2. If STAF is set '0' then interrupt for call to portC_ISR (port C service routine), otherwise wait.
3. There is no need to software reset STAF as there is automatic hardware reset of it by 68HC11 as soon as portC_ISR is called.

Driver routine portC_ISR programming is done as follows. Assume the name of pointers and variables are as following: (i) *portC_Queueback is a pointer that points to a memory address where the byte from port C inserts into to a queue. (ii) portC_QueueLength is present queue length. (iii) portC_Max QueueSize is the maximum queue length defined for the port C received bytes.

1. If *quasi_bidir* bit does not equal to false, write 0xFF to port C.
2. Read port C.
3. If *portC_QueueLength* is less than the *portC_MaxQueueSize* store port bits at the address defined by **portC_QueueTail*.
4. If *portC_QueueLength* is not equal to *portC_MaxQueueSize* then increment **portC_QueueTail* to let it now point to next address. When equal then call an exception (*error* routine) for port C.

4.9.4 Serial Port Drivers in a System

There is IEEE standard called POSIX (portable operating system interface) standard. Portability of the UART drivers in different systems is essential. In a PC with 80x86 processor an UART 8250 or a new generation UART device UART 16550, which includes the 16 byte FIFO input and output buffer is used. Example 4.19 gives all three sets of the registers (data, control and status) for a serial-line UART device in a PC. All PCs have this device.

Example 4.19

A serial-line device 8250 or 16550 in a 80x86-based IBM PC has the addresses of device registers as follows. These addresses are fixed by hardware configuration of the UART port interface circuit in IBM PC system employing the 80x86 processor. They are from 0x2F8 to 0x2FE at COM2 port in a PC and 0x3F8 to 0x3FE at COM1 port. Consider COM 2.

1. Two I/O data buffer registers (RBR for receiving and TRH for transmitting) are at a common address, 0x2F8—
 - (a) Provided a control bit at address 0x2FB is 0, (i) during read from the address, the processor accesses from the RBR or (ii) during write to the address, processor accesses the TRH.
 - (b) Provided a control bit at address 0x2FB is 1, data of two bytes of *divisor latch* are at distinct addresses, 0x2F8 (LSB) and 0x2F9 (MSB). Divisor latch holds a 16-bit value for dividing the system clock. This then selects the rate of serial transmission of bits at the serial line. [While writing a device driver, remember that a bit in another register (control register) changes the 0x2F8 access from access to the IO register to the lower byte register at divisor latch register.
 2. Three control registers are at three distinct addresses 0x2FA, 0x2FB and 0x2FC. These are for writing in registers as follows—
 - (a) IER (interrupt-enabling register). It enables the device interrupts.
 - (b) LCR (line control register). It defines how and how many bits will be on the line.
 - (c) MCR (modem control register). It defines how the modem handshakes and communicates.
 3. Three status registers of the device are also at three addresses 0x2FA, 0x2FD and 0x2FE and are used during read from these. These are as follows—
 - (a) IIR (interrupt identification register) for flags at 0x2FA. A flag sets on a device interrupt and resets at the servicing of corresponding device interrupt.
 - (b) LSR at 0x2FD. It is for reading line status the number of bits that will be present on the line.
 - (c) MSR at 0x2FE. It specifies the modem status bits during handshakes and communication.
- Assume that device has been given identity number 5. It is also a file descriptor for the device that points to description parameters of the device.

- (a) A serial device high-level driver function, *open* (5, *baudrate*) will configure and initialize the device. It sets the reset flag in IIR. The device initializes by unmasking the device interrupts and writing the control bits for clock divisor latch at the specified address. Divisor latch bits will define the baud rate configured for the device.
- (b) A serial device high-level driver function *write* (5, *length1*, *memTxaddress*) will send bytes into TRH one by one and the device transmits total bytes = *length1* from addresses *memTxAddress* to *memTxAddress + length1 - 1*.
- (c) A serial device high-level driver function *read* (5, *len2*, *memRxaddress*) will receive bytes through RBR and the device receives the bytes one by one and *len2* number of bytes are put in the buffer at memory address from *memRxaddress* to *memRxaddress + len2 - 1*.
- (d) A serial device high-level driver function *close* (5) will close the device. It can then be reused only after opening it by *open* (). The device closes by masking the device interrupts.

4.9.5 Device Drivers for Internal Programmable Timing Devices

Generally, there is at least one hardware timer T as an internal device in any systems needing functions related to timers. Using the time-outs (ticks) from T (using overflow interrupts) several needed software timers (SWTs) as can also be driven.

Example 4.20

A given hardware timer time-outs every 2^{16} (16384) counts and let the timer clock give input at every $2 \mu\text{s}$. Assume that an SWT is to be programmed to tick every $31 \times 32,768 \mu\text{s} = 1.015808 \text{ s}$. SWT should interrupt after *swt* counts *swtcnt* becomes equal to *numTicks* (preset number of ticks). The SWT is first initialized to *swtcnt* equals 0 and on every T overflow an ISR increments *swtcnt* and when *swtcnt* equals 31 then *swtcnt* is reset to 0 after generating software interrupt by an SWI instruction. An SWT-ISR then executes to perform required actions on SWT interrupt.

Timer device driver function call an ISR. The ISR programming needs an understanding of the programming of each bit of timer control register(s) and status register(s). An important step is programming of each bit of one or two control registers present and the use of status register. The programmer must also take into account the following. (i) Instead of interrupt enable, a device may have a mask bit. Mask bit means interrupt disables on set and enables on reset. Its actions are opposite to that of the enable bit. The programmer must also remember that a certain interrupt cannot disable (cannot mask, NMI). These enable or mask bits are the secondary bits. There is an overall interrupt system enable bit, which is like a master key (primary-level bit) for all maskable interrupt sources. The driver must set that bit also.

Step I: Write in a register that holds the timer maximum count value, the number of count inputs, *numTicks* for the SWT.

Step II: Write in status register the timer status flag(s) equal to reset [in case the device does not reset flag(s) automatically on a read of the status flag(s)].

Step III: Write each bit present in the control register(s). Write interrupt secondary- and primary-level enable bits equals true in control register, write other bits according to their uses. It is essential to write the device enable bit to let the device work. Definition of each bit in the mode register, if present is also essential.

Assume that a free running counter (FRC) is used as a timing device. Device-driver ISR programming steps require use of 68HC11 RTC described in Section 3.8. Consider following example. (68HC11

microcontroller knowledge is presumed here). The example gives the details of bits that are initialized for using FRC device of 68HC11.

Example 4.21

- Step I:* Define the output compare register(s) to hold count instance(s) of the FRC when OC flag(s) sets and OC interrupt(s) occur(s).
- Step II:* Flag(s) on its read from status register must be reset in case the device does reset automatically. The flags that may be present are the FRC overflow status flag, OC flag(s), ICAP_F flag(s), RTC flag and SWT flag(s). These are to be reset on a read of the status register.
- Step III:* Define control register(s) bits. Here, definition for each bit present is essential. The bits may be as follows. Prescaling bits for count input clock, overflow interrupt enable bit, RTC interrupt enable, OC interrupt enable bit(s), OC enable bit(s), OC output level bit(s), ICAP enable bit(s), ICAP input edge bit, ICAP input bit(s), ICAP interrupt(s) enable bit, SWT enable bits and SWT interrupts enable bit(s).
- Step IV:* Also enable the primary level interrupt enable bit, if already not enabled.

4.9.6 Linux Internals as Device Drivers and Network Functions

Drivers for port, keypad, display, timer and network devices (Sections 3.3, 3.6 and 3.9) are most commonly used in the systems. Drivers for PCS (physical coding sublayer) and PMA (physical media attachment) are required in media devices for most voice and video systems. It becomes impractical for a programmer to write the codes for each function of device. For commonly used devices, a programmer most often relies on drivers that are readily available in the thoroughly tested and debugged operating system (Refer to Chapters 9 and 10 for μ COS II, VxWorks OS, Windows CE, OSEK and Real Time Linux).

The Linux operating system is a tested and debugged operating system and is used throughout. It has a large number of drivers (Table 4.2) that are, moreover, in the public domain. Public domain means non-proprietary and usable by anyone. A programmer may therefore choose Linux drivers when the embedded system being designed has the devices that have the drivers available in Linux (refer <http://www.linuxdoc.org>).

Linux has internal functions called Internals. Internals exist for device drivers and network-management functions. Examples of useful Linux drivers for the embedded system are given in Table 4.2.

Linux internal functions exist for sockets, handling of socket buffers, firewalls, network protocols (e.g., NFS, IP, IPv6 and ethernet) and bridges. These are in the *net* directory. They work separately as drivers and also form a part of the network management function of the OS. (The reader can refer a standard textbook for bit-wise meaning of UDP, PPP and SLIP and for socket functions, firewall and network protocols. For example, refer *Internet and Web Technologies* from Tata McGraw-Hill, 2002 for bit-wise description of PPP, SLIP, TCP, IP, ethernet and other protocols).

Device drivers play a key role in most embedded system as these provide software layers between the application and devices. Drivers control almost all devices except the memory devices and the processor in a system. Linux device drivers are also used popularly because they are tested and debugged and are in the public domain. The Linux OS has internals and a large number of readily available device drivers for the most common physical and virtual devices and has the functions for the network sockets and protocols.

Table 4.2 Useful Linux Device Drivers

Driver type	Explanation
<i>char</i>	Drivers for char devices. A char device is a device for handling a stream of characters (bytes).
<i>block</i>	Drivers for block devices. A block device is a device that handles a block or part of a block of data. For example, 1 kB of data handled at a time. (<i>Note:</i> Unix block driver does not facilitate use of a part of the block during read or write.)
<i>net</i>	Drivers for network devices. A net device is a device that handles network interface device (card or adapter) using a line protocol, for example, tty or PPP or SLIP.
<i>input</i>	Drivers for the standard input devices. An input device is a device that handles inputs from a device, for example, keyboard.
<i>media</i>	Drivers for the voice and video input devices. Examples are video-frame grabber device, teletext device, radio-device (actually a streaming voice, music or speech device).
<i>video</i>	Drivers for the standard video output devices. A video device is a device that handles the frame buffer from the system to other systems as a char device does or a UDP network packet sending device does.
<i>sound</i>	Drivers for the standard audio devices. A sound device is a device that handles audio in a standard format.
<i>system</i>	Platform-specific drivers. Recently, system processor-specific drivers have also become available in this operating system. Examples are drivers for an ARM processor-based system.



Summary

The following is a summary of the important points that were discussed in this chapter.

- Interrupt means event, which invites attention of the processor for the action of hardware. Event can be a hardware or software event. In response to the interrupt, running routine or program interrupts and a service routine executes.
- When a device or port is ready, a device or port generates an interrupt or when it completes the assigned action, it generates an interrupt. These interrupts are called hardware interrupts. When software run-time exception condition is detected, either processor hardware or a software instruction SWI generates an interrupt for *exception*. An SWI instruction is *INT n* in 80x86.
- Device, run-time error and software instruction-related interrupts are studied. Various possible sources of the software and hardware interrupts are listed.
- Device driver functions execute software-interrupt routines for servicing the device, and drive a peripheral or internal device by create, open, read, write, close or other device function. A device is configured and initialized by using the control bits at its control register(s). The device driver executes on hardware or software interrupts as per set flags in status register(s).
- Physical device drivers, virtual device drivers and ISRs for the software instruction, software-defined condition and error condition are used to program the system.
- Every system has an interrupt service mechanism.
- We learnt *device initialization, driver ISR and function coding* for the parallel ports, serial-line UART and internal timing device in 68HC11.
- Virtual devices like char device, block device or file device, RAM disk, socket, pipe, loop back device are used during programming. These are treated in a way analogous to the physical devices.
- There are the device interrupts as well as other interrupt sources, driver functions and ISRs for which must be written by the programmer. A list of the various possible sources of software and hardware interrupts is given. A

software instruction or a condition during running-related or run-time error-related or device driver function interrupts are important in the systems.

- Interrupt system and individual device interrupt enabling and disabling, interrupt vectors, interrupt pending registers and status registers, non-maskable, maskable, non-maskable only when so defined within a few clock cycles after reset.
- Each running program has a *context* at each running code instance. Context means a CPU state (PC, stack pointer(s), registers and program state (variables that should not be modified by another routine). The context must be saved on a *call* to another ISR or *task* or routine. It must be done before processor switching to another context. Processor switches to another context by retrieving called program context. Certain processors like those from the ARM family provide for fast context switching. These have the internal stack frames or sets of local registers for the contexts. Fast context switching reduces the interrupt latencies and enables the meeting of each real-time task deadline. The OS program provides for memory blocks (allocates the blocks) to be used as stack frames in a multitasking system.
- Programming should be such that interrupt latencies are made as short as possible. This helps in meeting the deadlines for each interrupt service. Use of interrupt service threads (slow-level ISRs initiated by SWIs) helps in having the main first-level ISR codes short. The use of a DMA channel facilitates the small interrupt latency periods of an IO interrupt source requiring bulk or burst data transfers.
- There may be simultaneous service demands from multiple sources. Assignment of software priorities among the multiple sources of interrupts keeping in view the available hardware priorities is essential.
- Linux has a large number of device drivers, which are open-source.



Keywords and their Definitions

Context	:	PC, stack pointer as well as the program status word and processor registers for a foreground program or ISR or task. It can also include memory block addresses allotted to the program or routine.
Context switching	:	Saving the foreground program [interrupted routine (or function)] context and retrieving or loading the new context of the called routine. The time taken in context switching is included in the interrupt latency period.
Deadline	:	A period during which service to an interrupt must start.
Device attaching (adding)	:	Configuring a device and enabling the use of its driver.
Device detaching (removing)	:	Disabling the use of a device driver by the system.
Device driver ISR codes	:	Codes for read and write or other operations at the device addresses after reading device status on interrupt.
Device initialization codes	:	Codes for programming the control register of a device.
Device opening	:	Resetting the device control bits and preparing it for the use of its driver.
Device closing	:	Resetting the device control bits and its next time use is then possible only by opening it again.
Exception	:	An interrupt on detection of a run-time event during computations or communication. Setting of a condition that may be defined by the programmer.
Exception handler	:	Programmer defines the exception handler ISR also for handling service for that condition. Error conditions are handled by the exception handlers. Exception handler is called on executing an SWI instruction.
Foreground program	:	Foreground program is one that is executed when no interrupt call is being serviced.
Hardware-assigned Priority	:	Priority assigned by the processor itself to service a source when several interrupts need the interrupt service.

Hardware Interrupt	:	Interrupt of devices or ports at the system.
Hardware timer	:	A timer present in the system as hardware and which gets inputs from the internal clock with the processor. A device-driver program programs it like any other physical device.
Interrupt	:	CPU on interrupt event may initiate a further action by vectoring to a vector address and calling an ISR or else it continues with the current process (task) if the interrupt is disabled or masked.
Interrupt enable bit	:	It enables (unmasks) the interrupts from a source(s).
Interrupt flag	:	A register bit for a Boolean variable that sets to reflect a need for executing an ISR. It resets when corresponding ISR starts executing.
Interrupt latency	:	A period for waiting for service after a service demand is raised (source status flag sets).
Interrupt mask bit	:	When this bit is reset (= false) the request for initiation of interrupt service is responded, otherwise it is not responded.
Interrupt pending register	:	A register to show the interrupt sources or source groups from various devices that are pending for service by executing the corresponding ISRs. It is a 'read and write' register. A bit auto resets in it when the corresponding interrupt service starts. A user instruction can also reset a bit in the register.
Interrupt service mechanism	:	A mechanism for interrupt-driven service of the devices and ports. It saves the processor waiting time, because it lets the processor process the multiple devices and virtual devices. The mechanism also sets the priorities and provides for enabling and disabling the services.
ISR	:	A program that is executed on interrupt after saving the necessary parameters called context onto the stack so that the same can be retrieved on return from the routine last instruction. An ISR is also a device driver ISR when a high level language device driver function executes SWI and it services a device-interrupt. An ISR is also a trap when it services a software error or other condition-related interrupts with error detected by processor hardware. An ISR is also called <i>exception handler</i> on <i>exception</i> , which is <i>thrown</i> when it services software run-time condition detection and the condition is detected in the software routine. It is also called <i>signal handler</i> . When a <i>signal function</i> is called in a program. Each signal or exception throwing function executes an SWI, which initiates an ISR.
Interrupt vector	:	A memory address where there are bytes to provide the corresponding ISR address. The system has the specific vector addresses assigned by the hardware for each interrupting source for each internal device.
Interrupt vector table	:	A table for the interrupt vectors in the memory. The table facilitates the service of the multiple interrupting sources or source-groups for each internal device. In each row for an interrupt vector address, there are bytes to provide the corresponding interrupt service routine address.
Linux	:	An open source OS. It has a large number of device drivers and network management functions.
Linux device drivers	:	Device drivers taken from the Linux source.
Maskable interrupt source	:	A source, service routine call for which can be disabled or service of which masked.
Non-maskable interrupt source	:	A source, which cannot be disabled and which is used for the highest priority interrupt service cases, like RAM parity error.
Polling	:	A method to find the status of a peripheral or device. It is also a method by which at the end of an instruction or at the end of an ISR, the pending interrupts are searched by the processor from the status register or interrupt pending register to service the one with the highest priority.

Primary-level enable bit	:	A bit, which enables or disables any service on interrupt by all the maskable sources. It helps in executing critical section codes and preventing service to any other maskable source during disabling of service.
Secondary-level mask bit	:	It disables service from an individual source or source group.
Signal	:	Signal is function to initiate software-interrupt on an instruction to call a software-initiated ISR. An <i>exception</i> or trap may also be called a <i>signal</i> . Signal is called by SWI instruction in ARM and INT n in 80x86.
Software-assigned priority	:	(Hardware signal is different.) A priority for a source or source group. It is defined at a register called interrupt priority register. When several interrupts occur at the same time, software-assigned priorities over-ride the hardware priorities.
Software Interrupt	:	An interrupt by an error condition trap or illegal opcode or an SWI instruction (or INT n instruction 80x86) in a routine or ISR or software timer or signal.
Stack frame	:	A set of registers or a memory block that stores the context for a program or ISR.
Status register	:	A read only register for a device to set a flag on arising of an interrupt. A user instruction can also reset a bit in it. If a device has a number of sources the status register has a number of flags and a distinct source for each source. When it is read by a processor instruction, the flag resets.
Trap	:	An interrupt on detection by hardware a run-time computational or other event. The processor may also signal an exception on the <i>trap</i> . Example of a trap is division by zero in 80x86.
Virtual device	:	A device, which emulates the physical device and drives by virtual device drivers provided in an operating system. Examples are file, pipe, socket, etc.
Worst-case latency	:	Maximum interrupt latency found in the worst possible case.



Review Questions

1. What are the disadvantages and advantages of *busy and wait transfer* mode for the I/O devices?
2. What are the advantages and disadvantages of interrupt-driven data transfer?
3. What are the advantages of DMA based or peripheral-transaction-server based data transfer over the interrupt-driven data transfer?
4. How is the vector address used for an interrupt source?
5. Interrupt vector addresses are prefixed in the interrupt mechanism for the known internal peripherals in a microcontroller. How are the vector addresses assigned for exceptions and user-defined interrupts?
6. interrupt mechanism in each processor differs from a processor family to another. Explain, why the device drivers are processor-sensitive programs.
7. How do you initialize and configure a device? Take an example of serial-line driver at COM port of PC.
8. How is a *file* at the memory act handled as a device?
9. What are the advantages of RAM disk?
10. Make a list of Linux internal *net* directory functions for sockets, handling of socket buffers, firewalls, network protocols (e.g., NFS, IP, IPv6 and ethernet) and bridges. Why are these device drivers assigned in a separate directory of network management function of Linux OS.
11. Define *context*, *interrupt latency* and *interrupt service deadline*.
12. Why is the context switching in an embedded processor faster than saving the pointers and variables on the stack using a stack pointer? How does the context switching time reduce in processor architectures for embedded systems?

13. How is the context switching handled in ARM7?
14. DMA helps in reducing the processor load by providing direct access for the I/Os. How does it help in faster task execution in a multi tasking system by the reduced interrupt service latencies?
15. What do you mean by throwing an exception? How is the exception condition during execution of a function (routine) handled?
16. How do the device driver functions and ISRs differ? How do the ISR calls differ in 80x86 and 8051?
17. How do you assign service priority to the multiple device drivers of a system? How do you assign priorities to the timer devices and ADC device?
18. What are the uses of hardware-assigned priorities in an interrupt service mechanism?
19. What are the uses of software-assigned priorities in an interrupt service mechanism?
20. How is break point interrupt important for debugging embedded software?
21. What do you mean by POSIX function?



Practice Exercises

22. How do you write device driver? List the steps involved in writing a device driver.
23. Search web and design a table to show features in device driver modules of embedded Linux OS. Explain with examples each a char device, block device and block device configurable as char device. UART is a char device. Why is it a char device?
24. Give software-related interrupt examples. What are the interrupts in 8086, which generate software error?
25. Show the state machine-generated states in a key marked as number 4 in the mobile device. How will you use the SWI instruction to generate an SMS message in a mobile phone having a T9 keypad?

Programming Concepts and Embedded Programming in C, C++ and Java

5

R

Let us recapitulate the following points covered in the previous chapters.

- e * System hardware consists of processor(s), memory (ROM and RAM), I/O port, timing and external devices.
- c * Programming is required for computational tasks.
- a * Programming is required for the ISRs called on software interrupts from traps, exceptions, errors, signals and interrupts from physical and virtual devices.
- a * OS software is required for using any device in a simple or sophisticated application(s).
- l * Certain software instructions are required according to the given processor, memory and device hardware and as per the system interrupt servicing mechanism.

Programming is the most essential part of any embedded system design. Except for certain processor and memory-sensitive instructions where program codes may be written in assembly, most of the codes are written in a high level-language.

L

E

A

R

N

I

N

G

O

B

J

E

C

T

I

V

E

S

For an in-depth learning of the programming language, a reader should refer to the standard textbooks and do required practice exercises. We will learn basics of programming the functions (methods) and concepts of object-oriented programming with reference to building software for the embedded systems. OS concepts, we will learn later.

The following are the topics that will be discussed in this chapter:

1. Programming in the assembly language vs. high-level language and the powerful features of C for embedded systems.
2. Program elements: Preprocessor directives and the header files, include files and source files that are used in a program for an application.
3. Program elements: Macros and functions and their uses in a C program.
4. Program elements: Data types, pointers, data structures, arrays, queues, stacks, lists and trees, modifiers, conditional statements and loops.
5. Program elements: Function calls, multiple functions, function pointers, function queues and service-routine queues.
6. Object-oriented Programming concepts, embedded programming in C/C++, Java and J2ME

Program models for building the software will be dealt with in Chapter 6. Concepts of the processes, tasks, threads and the concepts of interprocess synchronization will be covered in Chapter 7 and of RTOSes in Chapter 8. Popular RTOSes are described in Chapters 9 and 10.

5.1 SOFTWARE PROGRAMMING IN ASSEMBLY LANGUAGE (ALP) AND IN HIGH-LEVEL LANGUAGE 'C'

5.1.1 Assembly Language Programming

Assembly language coding of an application has the following advantages.

1. The assembly codes are sensitive to the processor, memory, ports and devices hardware. It gives a precise control of the processor internal devices and complete use of the processor-specific features in its instruction set and its addressing modes.
2. The machine codes are compact, processor- and memory-sensitive. This is because the codes for declaring the conditions, rules and data type do not exist. The system thus needs a smaller memory. Excess memory needed does not depend on the programmer data type selection and rule declarations. The program is also not compiler specific and library functions dependent.
3. Device driver codes may need only a few assembly instructions. For example, consider a small embedded system, a timer device in a microwave oven or an

automatic washing machine or an automatic chocolate-vending machine. Assembly codes for these can be compact and precise, and are conveniently written.

4. We use *bottom-up-design* approach. It is an approach in which programming is first done for the sub-modules of the specific and distinct sets of actions. An example of the modules for specific sets of actions is a program for a software timer, RTCSWT: run (real-time clock software timer function run). Programs for delay, counting, finding time intervals and many applications can be written. Then the final program is designed. The approach to this way of designing a program is to first code the basic functional modules and then use these to build a bigger module.

5.1.2 High-level Language Programming

High-level language coding of source files in C or C++ or Java has great advantages and therefore most programming is in high-level language. Basic advantages are as follows:

1. *High-level program development cycle is short even for complex systems* because of the followings: use of routines (called functions in C/C++ and methods in Java), standard library functions, use of modular programming approach, top-down design or object-oriented design approach. Application programs are structured to ensure that the software is based on sound software engineering principles and programmed with the given OS, file systems, device and network drivers.
 - (a) A function defines a method of operation, and sets of statements and commands are run when that function is called.
 - (b) Library functions are standard functions, which are readily available to a programmer and the codes for them are not defined by programmer. For example, square root method of operation. The use of the standard library function, square root (), saves the programmer time for coding. New sets of library functions exist in an embedded system-specific C or C++ compiler. Exemplary library functions are delay (), wait () and sleep ().
 - (c) Identical devices such as serial-line device (UART) are used in a number of embedded systems. Directly programming these functions in each system will mean the *repetitive and redundant coding* for each device. It is better to use the device drivers in high-level language, which use the functions specified in the OS. The programmer simply specifies device ID and some of the function arguments passed to the device driver function when needed and use it at another instance of the device use.
 - (d) *Modular programming approach* is an approach in which the building blocks are reusable software components. A module is built by software components. The components are built by a set of functions. Consider an analogy to an IC (integrated circuit). Just as an IC has several circuits integrated into one, similarly a module building block may call several functions and library functions. A module is then well tested for a well-defined goal and for the well-defined data input and outputs. It should have only one calling method. There should be one return point from it. It should not affect any data other than the one it operates: this means that there should be data encapsulation. It must return (report) error conditions encountered during its execution.
 - (e) *Top-down design* is another programming approach in which the *main* program is first designed, then its modules, sub-modules, and finally, the functions.
2. *High-level program facilitates data type declarations*: Data type declarations provide programming ease. For example, there are four types of integers, *int*, *unsigned int*, *short* and *long*. When dealing with positive only values, we declare a variable as *unsigned int*. For example, numTicks (number of ticks to a clock) has to be *unsigned*. We need a signed integer, *int* (32 bits) in arithmetical calculations. An integer can also be declared as data type, *short* (16 bits) or *long* (64 bits). To manipulate the text and strings for a character, another data type is *char*. Each data type is an abstraction for the methods, which are permitted for using, manipulating, representing and for defining a set of permissible operations on that data.

3. *High-level program facilitates 'type checking'* making the program less prone to error. For example type checking does not permit subtraction, multiplication and division on the *char* data types. It permits 'plus' operator to be used for concatenation when using *char* data types and lets the 'plus' operator to be used for arithmetic addition when using *int*, *unsigned int*, *short* and long type of data. (Concatenation operation can be understood as follows: the *micro* plus *controller* concatenates into the *microcontroller* where *micro* is an array of *char* values and *controller* is another array of *char* values.)
4. *High-level program facilitates use of control structures* (e.g., *while*, *do-while*, *break* and *for*) and *conditional statements* (e.g., *if*, *if-else*, *else-if* and *switch-case*) to specify the program flow by simple statements.
5. *High-level program has portability* of non-processor-specific codes. Therefore, when the hardware changes, only the modules for the ISRs of device drivers and device management, initialization and program-locator modules and initial boot-up record data need modifications.

Additional advantages of C as a high-level languages are as follows. It is a *language between low-(assembly) and high-level languages*. Inserting the assembly language codes in-between is called in-line assembly. A direct hardware control is thus also feasible by in-line assembly, and the complex part of the program can be in high-level language.

High-level language programming makes the program development cycle short, enables use of the modular programming approach and allows us to follow sound software engineering principles. It facilitates the program development with top-down design approaches. Embedded system programmers have since long preferred C for the following reasons: (i) The feature of embedding assembly codes using in-line assembly. (ii) Readily available modules in C compilers for the embedded system and library codes that can directly port into the system-programmer codes.

5.2 C PROGRAM ELEMENTS: HEADER AND SOURCE FILES AND PREPROCESSOR DIRECTIVES

A C program has following structural elements.

1. Preprocessor declarations, definitions and statements.
2. Main function.
3. Functions, exceptions and ISRs.

A C program has the following preprocessor structural elements.

1. *Include* directive for the file inclusion.
2. *Definitions for preprocessor global variables* (global means throughout the program module).
3. *Definitions* of constants.
4. *Declarations* for global data type, type declaration and data structures, macros and functions.

The C program elements, header and source files and preprocessor directives are explained in the following subsections.

5.2.1 Include Directive for the Inclusion of Files

Any C program first includes the header and source files that are readily available. A case study of sending a stream of bytes through a network driver card using a TCP/IP protocol is given in Example 11.3. Its C program starts with the codes given in Example 5.1. The purpose of each included file is mentioned in the comments within the */** and **/* symbols as per the practice in C.

Example 5.1

```
# include "VxWorks.h" /* Include VxWorks functions*/
# include "semLib.h" /* Include Semaphore functions Library */
# include "taskLib.h" /* Include multitasking functions Library */
# include "msgQLib.h" /* Include Message Queue functions Library */
# include "fioLib.h" /* Include File-Device Input-Output functions Library */
# include "sysLib.c" /* Include system library for system functions */
# include "netDrvConfig.txt" /* Include a text file that provides the 'Network Driver
Configuration'. It provides the frame format protocol (SLIP or PPP or Ethernet) description, card
description/make, address at the system, IP address (s) of the node (s) that drive the card for
transmitting or receiving from the network. */
# include "prctlHandlers.c" /* Include file for the codes for handling and actions as
per the network layer-protocols used for driving streams to the network. */
```

Include is a preprocessor directive to include the contents (codes or data) of a file. The files that can be included are given next. Inclusion of all files and specific header files has to be as per the requirements.

1. *Including code files:* These are the files for the codes already available. For example, # include 'prctlHandlers.c'.
2. *Including constant data files:* These are the files for the codes and may have the extension '.const'.
3. *Including strings data files:* These are the files for the strings and may have the extension '.strings' or '.str.' or '.txt'. For example, # include 'netDrvConfig.txt'.
4. *Including initial data files:* There are files for the initial or default data for the shadow ROM of the embedded system. The boot-up program is copied later into the RAM and may have the extension '.init'. On the other hand, RAM data files have the extension, '.data'.
5. *Including basic variable files:* These are the files for the local or global static variables that are stored in the RAM because they do not possess the initial (default) values. The static means that there is a common not more than one instance of that variable address and it has a static memory allocation. For example, system has only one real-time clock, and therefore only there is one instance of addresses of the clock variables. The basic variables of the clock are stored in the file with the extension '.bss'.
6. *Including header files:* It is a preprocessor directive, which includes the contents (codes or data) of a set of source files. These are the files of a specific module. A header file has the extension '.h'. Examples are as follows. The string manipulation functions are needed in a program using strings. These become available once a header file called 'string.h' is included. The mathematical functions, *square root, sin, cos, tan, atan* and so on are needed in programs using mathematical expressions. These become available by including a header file, called 'math.h'. The preprocessor directives will be '# include <string.h>' and '# include <math.h>' for including standard library functions for the strings and mathematical operations in the program.

Also included are the header files for the codes in assembly, and for the I/O operations (conio.h), for the OS functions and RTOS functions. '# include VxWorks.h' is directive to the compiler, which includes VxWorks RTOS functions.

Note: Certain compilers provide for *conio.h* in place of *stdio.h*. This is when embedded systems do not need the file functions for opening, closing, read and write operations on the keyboard and video monitor.

So when including *stdio.h*, it will make the code too big.

What is the difference between inclusion of a header file, and text file or data file or constant file? Consider the inclusion of *netDrvConfig.txt.txt* and *math.h*. (i) The header files are well-tested and debugged modules.

(ii) The header files provide access to standard libraries. (iii) The header file can include several text files or C files. (iv) A text file is a description of the text that contain specific information.

5.2.2 Source Files

Source files are program files for the functions of application software. The source files need to be compiled. A source file will also possess the preprocessor directives of the application and have the *first function from where the processing will start*. This function is called *main* function. Its codes start with *void main ()*. The *main* calls other functions. A source file holds the codes.

5.2.3 Configuration Files

Configuration files are the files for configuration of the system. Device configuration codes can be put in a file of basic variables and included when needed. If these codes are in the file "serialLine_cfg.h" then # include 'serialLine_cfg.h' will be the preprocessor directive. Consider another example. '# include 'os_cfg.h'; this will include os_cfg header file.

5.2.4 Preprocessor Directives

Preprocessor constants, variables and inclusion of configuration files, text files, header files and library functions are used in embedded C programs. A preprocessor directive starts with a sharp (hash) sign. These commands are for the following directives to the compiler for processing.

1. *Preprocessor global variables:* For example, in a program the IntrDisable, IntrPortAEnable, IntrPortADisable, STAF and STAI may be the global variables for disabling interrupts, enabling port A, disabling port A, status flag, status flag for interrupt, respectively. Now '# define volatile boolean IntrEnable' is a preprocessor directive. It means it is a directive before processing to consider IntrEnable a global variable of boolean data type and is volatile. (Volatile is a directive to the compiler not to take this variable into account while compacting and optimizing the codes.)
2. *Preprocessor constants:* '# define false 0' is a preprocessor directive in an example. It means it is a directive before processing to assume 'false' as 0. The directive 'define' is for allocating pointer values in the program. Consider # define portA (volatile unsigned char *) 0x1000 and # define PIOC (volatile unsigned char *) 0x1001. 0x1000 and 0x1000 are the addresses fixed for port A (port A register) and PIOC (port input-output control register) are the constants defined for the 68HC11 register addresses.

Strings can also be defined. Strings are the constants, for example, those used for an initial display on the screen in a mobile system. For example, # define welcome 'Welcome To ABC Telecom'.

5.3 PROGRAM ELEMENTS: MACROS AND FUNCTIONS

Table 5.1 lists these elements and gives their uses.

Preprocessor Macros: A macro is a collection of codes that is defined in a C program by a name. It differs from a function in the sense that once a macro is defined by a name, the compiler puts the corresponding codes for it at every place where that macro name appears. For example, consider the macros, 'enable_Maskable_Intr ()' and 'disable_Maskable_Intr ()'. (The pair of brackets in the macro is optional. If

it is present, it improves readability as it distinguishes a macro from a constant.) Whenever the name `enable_Maskable_Intr` appears, the compiler places the codes designed for it.

Table 5.1 Uses of the Various Sets of Instructions as the Program Elements

Program Element	Uses	Saves context on the stack before its start and retrieves them on return	Feasibility of nesting one within another
Macro	Executes a named small collection of codes.	No	None
Function	Executes a named set of codes with values passed by the calling program through its arguments. Also returns a data object when it is not declared as void. It has the context-saving and retrieving overheads.	Yes	Yes, can call another function and can also be interrupted
Main function	Declarations of functions and data types, typedef and either: (i) executes a named set of codes, calls a set of functions and calls on the interrupts the ISRs or (ii) starts OS Kernel.	No	None
Reentrant function	Refer Sections 5.4.6.	Yes	Yes to another reentrant function only
Interrupt service routine (ISR)	Declarations of functions and data types, typedef and executes a named set of codes. Must be short so that other sources of interrupts are also serviced within the deadlines. Must be a re-entrant routine. Not allowed to wait for interprocess messages (semaphore or mailbox or queue messages) but allowed to send the interprocess messages for the ISTs or tasks.	Yes	To ISR of higher- priority unmasked sources
Process or task or thread	Refer Sections 7.1–7.3. Must either be a reentrant routine or must have a solution to the shared data problem.	Yes	None
Recursive function	A function that calls itself. It must be a reentrant function also. Most often its use is avoided in embedded systems due to memory constraints. (Stack grows after each recursive call and it may choke the memory space availability.)	Yes	Yes

Macros, called test macros or test vectors are also designed during programming and are used for debugging. How does a macro differ from a function?

1. The codes for a function are compiled once only. On calling that function, the system has to save the context, and on return restore the context. Further, a function may return nothing (*void* declaration case) or return a Boolean value, or an integer or any primitive or reference type of data. (Primitive means similar to an integer or character. Reference type means similar to an array or structure.) For example, the `enable_PortA_Intr()` and `disable_PortA_Intr()` are the function calls. (The brackets are now not optional.)

2. The codes for macro are compiled in every function wherever that macro name is used, as the compiler, before compilation, puts the codes at the places wherever the macro is used. On using the macro, the processor does not have to save the context, and does not have to restore the context, as there is no return.
3. Macros are used for short codes only. This is because, if a function call is used instead of a macro, the overheads (context saving and new context retrieving, and other actions on function call and return) will take a time, $T_{\text{overheads}}$ that is the same order of magnitude as the time, T_{exec} for execution of short codes within a function. We use a function for codes when the $T_{\text{overheads}} \ll T_{\text{exec}}$, and a macro for codes when $T_{\text{overheads}} \approx$ or $> T_{\text{exec}}$.

Macros and functions are used in C programs. Functions are used when the requirement is that the codes should be compiled once only. However, on calling a function, the processor has to save the context, and on return, restore the context. A function may also return either nothing (*void* declaration case) or return a Boolean value, or an integer or any primitive or reference type of data. Macros are used when short functional codes are to be inserted in a number of places or functions.

5.4 PROGRAM ELEMENTS: DATA TYPES, DATA STRUCTURES, MODIFIERS, STATEMENTS, LOOPS AND POINTERS

5.4.1 Use of Data Types

Whenever a data is named, it will have the address(es) allocated at the memory. The number of addresses allocated depends on the data type. For example, a data type *long* is declared for `numTicks` (number of ticks). Then `numTicks` will need four memory addresses.

C allows the following primitive data types. The *char* (8 bits) for characters, *byte* (8 bits), *unsigned short* (16 bits), *short* (16 bits), *unsigned int* (32 bits), *int* (32 bits), *long double* (64 bits), *float* (32 bits) and *double* (64 bits). (Certain compilers do not take the 'byte' as a data type definition. The 'char' is then used instead of 'byte'. Most C compilers do not take a Boolean variable as data type. The *typedef* is used to create a Boolean type variable in the C program.)

A data type appropriate for the hardware is used. For example, a 16-bit timer can have only the unsigned short data type, and its range can be from 0 to 65535 only.

The *typedef* is also used. It is made clear by the following example. A compiler version may not process the declaration as an unsigned byte. The 'unsigned character' can then be used as a data type. It can then be declared as follows.

```
typedef unsigned character portAdata
#define Pbyte portAdata 0xF1
```

5.4.2 Use of Pointers and Null Pointers

Pointers are powerful tools when used correctly and according to certain basic principles. Pointer is a reference to a starting memory address. A pointer can refer to a variable, data structure or function. Before a pointer, in C language symbol, * is used. For example, `unsigned char *0x1000` means a character of 8 bits at address 0x1000.

A NULL pointer declares as following: `#define NULL (void*) 0x0000`. (We can assign any address instead of 0x0000 that is not in use in a given hardware.)

Exemplary uses are as follows.

Example 5.2

1. Consider 'unsigned short *timer1'. Pointer *timer1* will point to two bytes, and the compiler will reserve two memory addresses for the contents of *timer1*. Consider two statements, 'unsigned short *timer1;' and 'timer1++;'. The second statement adds 0x0002 in the address of *timer1*. Why?

In C, if *x* is a variable, then *x* ++ means increment of the value of *x* by 1. If *p* is a pointer, then *p* ++ means increment of the value of *p* to the next address.

timer1 ++ means point to the next address, and unsigned short declaration allocated two addresses for *timer1*. (*timer1* ++; or *timer* +=1 or *timer* = *timer* +1; will have identical actions.) Therefore, the next address is 0x0002 more than the address of *timer1* that was originally defined. Had the declaration been 'unsigned int *timer1;' and 'timer1++;' (in case of 32-bit timer), the second statement would have incremented the address by 0x0004.

2. Let a byte each be stored at a memory address. Let a port A in a system have a buffer register that stores a byte. Now a program using a pointer declares the byte at port A as follows: 'unsigned byte *portA'. The * means 'the contents at'. This declaration means that there is a pointer and an unsigned byte for port A. The compiler will reserve one memory address for that byte.
3. Consider declarations as follows: void *portAdata; the void is the undefined data type for portAdata. The compiler will allocate address for the *portAdata without any type check.
4. A pointer can be assigned a constant fixed address. Two preprocessor directives: '# define portA (volatile unsigned byte *) 0x1000' and '# define PIOC (volatile unsigned byte *) 0x1001'. Alternatively, the addresses in a function can be assigned as follows: 'volatile unsigned byte * portA = (unsigned byte *) 0x1000' and 'volatile unsigned byte *PIOC = (unsigned byte *) 0x1001'. An instruction, 'portA ++;' will make the portA pointer point to the next address and which is PIOC.
5. Consider, unsigned byte portAdata; unsigned byte *portA = &portAdata. The first statement directs the compiler to allocate one memory address for portAdata because there is a byte each at an address. The & (ampersand sign) means 'at the address of'. This declaration means the positive number of 8 bits (byte) pointed by portA is replaced by the byte at the address of portAdata. The right side of the expression evaluates the contained byte from the address, and the left side puts that byte at the pointed address. As the right-side variable of portAdata is not a declared pointer, the ampersand sign is kept to point to its address so that the right-side pointer gets the contents (bits) from that address. (Note: The equality sign in a program statement means 'is replaced by'.)

5.4.3 Use of Data Structures: Queues, Stacks, Lists and Trees

A data structure is a way of organizing several data elements of same types or different types together at consecutive memory addresses. A data element in a data structure can then be identified and accessed with the help of a few pointers and/or indices and/or functions. Marks (or grades) of a student in the different subjects studied in a semester are put in a proper table. The table in the mark sheet shows them in an organized way. Similarly, when there is a large amount of data, it must be organized properly.

A data structure is an important element of any program. Few important data structures include: *stack*, *one-dimensional array*, *queue*, *circular queue*, *pipe*, a *table* (two-dimensional array), *lookup table*, *hash table* and *list*. Following describes different data structures and how it is put in the memory blocks in an organized way. Any data structure element can be retrieved using the pointers.

Stack A data structure, called *stack* is a special program element. A *stack* means an allotted memory block data from which a data element is read in a LIFO (*last in first out*) way and an element is popped or pushed from an address pointed by a pointer, called SP (stack pointer) or S_{top} and SP changes on each push or pop such that it points to the top of stack.

Various stack structures may be created during processing. For handling each stack, one pointer, which points to the stack top is needed. Figure 5.1 shows the various stack structures that are created during execution of the embedded software.

1. A call can be made for another routine during running of a routine. In order that on completion of the called routine, the processor returns only to the one calling, the instruction address for return must be pushed on the stack. Pushing means saving on the stack top and incrementing stack to point to the next top. Popping means retrieving the saved address from the stack top and decrementing the stack to point to the previous top. There can also be nested calls and returns. Nesting means one routine calls another, which calls another and return from the called routine is always to the calling routine. Therefore, at the memory a block of memory address is allocated to the stack that saves the pushed *return addresses* of the nested calls. It is shown in Figure 5.1(a). Two bytes of address are acquired in the PC from stack on return from a call to a routine (function). Assume 10 nested calls are present in the system or other functions. Assume that PC address is of 4 B. Memory allocation required for a stack structure for pushing the return instruction addresses is 40 B.
2. There may be at the beginning of an input data, for example, received call numbers in a phone, which is saved onto a stack at RAM in order to be retrieved later in the LIFO mode. It is shown in Figure 5.1(b). Consider for example, on each push the following are saved on a stack. (i) Four pointers (addresses each of 4 bytes); (ii) four integers (each of 4 bytes) and (iii) four floating point numbers (each of 4 bytes). Memory allocation required for a stack structure for pushing the function parameters = $4 \times 4 + 4 \times 4 + 4 \times 4 = 48$ B.
3. An application may also create the run-time stack structures. There can be *multiple data stacks* at the different memory blocks, each having a separate pointer address. There can be *multiple stacks* shown as Stack 1, ..., Stack N in Figure 5.1(c).
4. Each task or thread in a multi-tasking or multi-threading software design (Sections 7.1–7.3) should have its own stack where its *context* (Section 4.6) is saved. The context is saved on the processor on switching to another task or thread. The context includes the return address for the PC for retrieval on switching back to the task. There can be *multiple stacks* shown as saved contexts of the threads as the stacks shown in Figure 5.1(d) at the memory for the different task contexts at the different memory blocks, each having a separate pointer address. Threads of application programs and supervisory (OS) programs have separate stacks at separate memory blocks.

Each processor has at least one stack pointer register so that the instruction stack can be pointed and calling of the routines can be facilitated.

Some advanced processors have multiple stack pointers. There are four pointers as follows:

1. RIP (return instruction pointer): RIP is for saving the return address of the PC when a routine calls another routine or ISR. RIP is called link register (LR) in ARM processor.
2. SP (stack pointer): SP is pointer to a memory block dedicated to saving the context on context switch to another ISR or routine. There is a stack pointer in 8051, 68HC11 and 80196.
3. FP (data frame pointer): FP is pointer to a memory block dedicated to saving the data and local variable values of presently running program (routine).
4. PFP (previous program frame pointer): PFP is pointer to a memory block dedicated to saved the program data frame.

Motorola MC68010 processor provides USP (user stack pointer) and SSP (supervisory stack pointer). Program runs in two modes: user mode and supervisory mode. In supervisory mode the OS functions execute.

There is switch from the user mode to the supervisory mode after every tick of the system clock (Section 8.1.2 will give the details). MC68040 provides for USP, SSP, MSP (memory stack frames pointers), ISP and (instruction stack pointer).

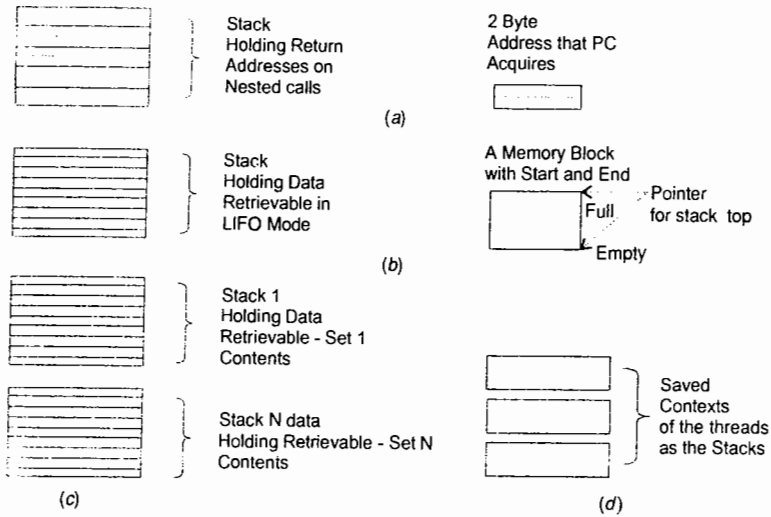


Fig. 5.1 (a) A stack due to nested function calls and pushing of program counters (b) Stack of pointers and parameters pushed onto the stack before the context switch (c) N stacks each having a separate pointer (d) Multiple stacks of contexts for the multiple threads

When a processor has only one SP, the OS allocates the memory addresses that are used as the pointers for the multiple instructions and data stacks.

A processor has at least one SP. Each process should have a separate top of SP and a separate block at its allocated memory for the nested function calls. A stack can also be a special data structure at the memory. It has a pointer address that always points to the top of stack. A value from the stack is retrieved from the memory in LIFO mode, while a row of data or a data in a table or a data in the queue is accessed in a FIFO (first in first out) mode. As there are multiple processes in an embedded system, each having separate context, there are multiple stacks.

Array A data structure, *array* is an important programming element. An array has multiple data elements, each identifiable by an index or by a set of indices and indices are unsigned integers. An n-dimensional array is a special data structure at the memory. It has a pointer address that always points to the first element of the array. For handling an n-dimensional array, one pointer, which points to the first element and n indices are needed.

Assume one-dimension array. From the first element pointer and an index of that element, an address is constructed from which the processor can access one of the array elements. Index is an integer that starts from 0 to (array-length-1) in a given dimension. Data word can be retrieved from any element address in the block that is allocated to the array. A processor register may also be used for storing the index and another register for array base pointer.

Following examples clarify the concept of array.

Example 5.3

1. Consider that unsigned int [] phone_num means an array of phone numbers, phone_num [0] refers to that first phone number, phone_num [1] refers to the second phone number, and so on. The phone_num itself points to the first element.
2. Consider unsigned char [] name which means an array of characters for the name. name [0] refers to the first character, name [1] refers to the second character and so on. The name without index points to the first array element.
3. Consider the results of a test in a class with 30 students with roll numbers 1-30. Let i be an index used instead of a roll number. Let marks in the test of roll number 1 be in the scalar integer variable, M[0]. Let M[0], M[1], ..., M[28] and M[29] be the variables for the marks of roll numbers 1, 2, ..., 29, 30, respectively. There is a pointer that points to the first scalar value M[0]. A register called index register may point to M[0]. The index register could then be incremented from 0 to 29 by an instruction within a loop to point to the marks of students of succeeding roll numbers. Figure 5.2(a) shows an array in the memory block with the pointers for base and index that jointly point to its element marks [i].

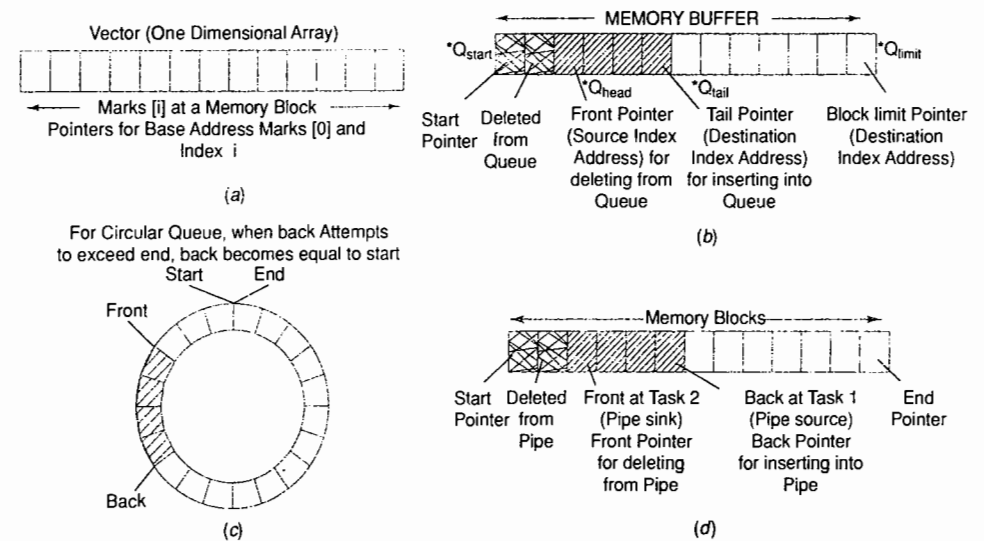


Fig. 5.2 (a) An array at a memory block with one pointer for its base, first element with index = 0. Data word can be retrieved from any element by defining the pointer and index (b) A queue at a memory block between *Q_start and *Q_limit with two pointers *Q_head and *Q_tail to point to its two elements at the queue front and back. A data word retrieves in the FIFO mode from a queue (c) A circular queue at a memory block with two pointers to point to its two elements at the front and back. A pointer on reaching a limit of the block returns to the start of the block (d) Memory blocks at source and sink for a pipe

Example 5.4

Take another example. An expression, $y_k = \Sigma(a_i \cdot x_{k-i})$ has coefficient a_i . These are stored as an array. Input x_i also stores as another array and output y_k as yet another array. Here, i and k are the integers each varying from $-N$ to $N-1$, where N is per the limits. Three arrays $y []$, $a []$ and $x []$ are used for calculating 20 filtered output sequences by the expression, $y_k = \Sigma(a_i \cdot x_{k-i})$. Memory allocations required for the each array structure = $20 \times 8 = 160$ B. [Assume each element as double precision floating pointer number of 8 B (64-bits IEEE 754 format).]

Example 5.5

When the index increments by 1 in case of an array, the pointer to the previous element actually increments by 4, and thus the address will increment by $0x0004$ in case of an array of integers. [An integer stores as 4-byte number.] For array data type, $*$ is never put before the identifier name, but an index is put within a pair of square brackets after the identifier. Consider a declaration, 'unsigned char portAMessageString [80];'. The port A message is a string, which is an array of 80 characters. Now, portAMessageString is itself a pointer to an address without the star sign before it. [Note: Array is therefore known as a reference data type.] However, *portAMessageString will now refer to all the 80 characters in the string. portAMessageString [20] will refer to the twentieth element (character) in the string.

Queue A data structure, called *queue* is another important programming element. In case of array, the reading is with the help of indices and first element address. So any element can be read or written at any instance. In queue, each element is read from an address next to the address from where the queue element was last read. This reading is called *deletion*. In queue, it is written to an address next to the address from where the queue element was last written. This writing is called *insertion*. A *queue* means an allotted memory block from which a data element is retrieved in the FIFO mode. Using the queues, the bytes are sent onto a memory buffer or network or printer.

For handling the queue, two pointers are needed and a memory buffer is allocated between buffer start address pointed by $*Q_{start}$ and buffer-end pointed by $*Q_{limit}$. One pointer $*Q_{tail}$ is for pointing to an address in a memory block where an element can be inserted (added on writing). For a queue of integers, $int *Q_{tail}$, $*Q_{head}$ are declared. $*Q_{tail}$ initially equals $*Q_{start}$ and it should increment on each insertion at queue back or tail pointer. The other $*Q_{head}$ (queue head pointer) initially equals $*Q_{start}$ and is for pointing to an address in a memory block from where an element can be deleted (remove on reading). This pointer should increment on each deletion. Both pointers $*Q_{tail}$ and $*Q_{head}$ point at the beginning to $*Q_{start}$ the starting memory buffer address at the block. Insertions into a queue are usually faster than deletions. For example, in a queue at the printer the system inserts the values faster than the rate at which values are printed. The difference in addresses at the two pointers at an instance is the present queue length.

There is a possibility that the tail pointer may increment beyond a limit set for the queue end address in the memory block. An exception (an error indication) is usually thrown whenever the pointer increments beyond the block end boundary. Else, on further increments an intrusion into other block may occur. Figure 5.2(b) shows a memory block between $*Q_{start}$ and $*Q_{limit}$ with the two pointers $*Q_{head}$ and $*Q_{tail}$ needed for deletions and insertions.

A queue is a data structure with an allotted memory block (buffer) from which a data element is retrieved in the FIFO mode. It has two pointers, one for its head and the other for its tail. Any deletion is made from the head address and any insertion is made at the tail address. An exception (an error indication) must be thrown whenever the pointer increments beyond the block end boundary so that appropriate action can be taken.

Circular Queue A queue is called *circular queue* when a pointer on reaching a limit $*Q_{limit}$, returns to its starting value $*Q_{start}$. (A *circular queue* means a bounded memory block allotted to a queue such that its pointer on incrementing never exceeds the set limit and returns to start on increment beyond the limit.) From a circular queue also, the data element is retrieved in the FIFO mode but no exception is thrown on exceeding the limit of the memory block allocated. Figure 2.4(c) shows a memory block with a circular queue with its two pointers needed for insertions and deletions.

A circular queue is a queue in which tail and head pointers cannot increment beyond the memory block (buffer) and reset to the starting value on insertion beyond the boundary.

Pipe A *pipe* is a device, which uses device driver functions and in which insertions are from the source end and deletions are at sink-end. The deletions are at the destination end, and are like in the queue. The insertion source has an identity distinct from a destination (sink) entity where deletions are made and source and destination are connected by some function pipe_connect (). Figure 5.2(d) shows memory blocks for a *pipe*.

A pipe is a device with insertions and deletions at distinctly defined source and destination.

Table A *table* is a two-dimensional structure (matrix) and is an important data set that is allocated a memory block. There is always a base pointer for a table. It points to its first element at the first column and first row. There are two indices, one for a column and the other for a row. Figure 5.3(a) shows a memory block with the pointers for a table. Like an array, any element can be retrieved from three addresses for the table base, column index and row index. Instead of a column or row pointer, a value is used in an instruction which is called the *displacement*. Displacement can be used for a column or row.

A *lookup table* is a two-dimensional structure (matrix) and is an important data set. It has only rows and each row has a key and on reading the key, the addressed data is traced.

A table is a data set allocated with a memory block. Three pointers, table base, column index and destination index pointers (or two pointers and one displacement) can retrieve any element of the table. Lookup table is a table of keys (pointers) and from reading a key the addressed data is retrieved.

Hash Table A *hash table* is a data set that is a collection of pairs of *key* and corresponding *value*. A hash table has a key or name in one column. The corresponding value or object is at the second column. The keys may be at non-consecutive memory addresses. Figure 5.3(b) shows a memory block with the pointers for a hash.

A hash table is a data set allocated with a memory block for key and value pairs.

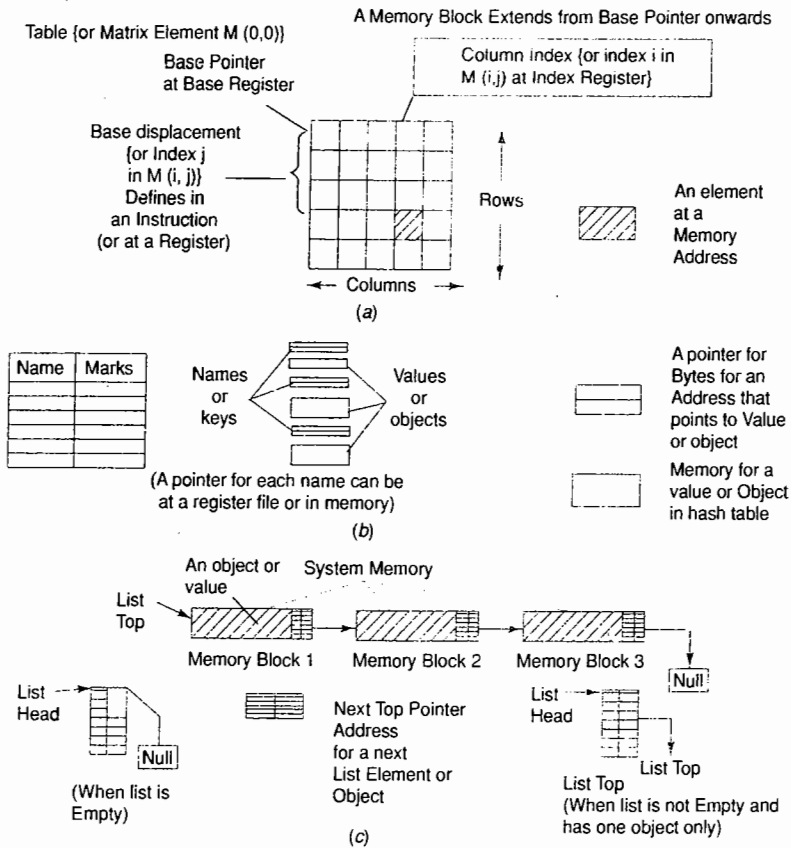


Fig. 5.3 (a) A memory block with the pointers for a table (b) A memory block for hash table with the pairs of key and value in a hash (c) The memory blocks in system memory with the pointers for a list

list A list is a data structure with a number of memory blocks, one for each element. A list has a top (head) pointer for the memory address from where it starts. Each list element at the memory also stores the pointer to the next element. The last element points to null. A list is for non-consecutively located objects at the memory. Figure 5.3(c) shows the memory blocks with the pointers for a list.

Example 5.6

Assume on a real-time clock tick, the ISR increments the counts in a number of RTCSWTs (real-time clock interrupts-triggered software timers). Assume that there is a list of RTCSWT timers that are active at an instant. The top of the list can be pointed as '*RTCSTW_List.top' using the pointer. RTCSTW_List.top is

now the pointer to the top of the contents in a memory for a list of the active RTCSWTs. Consider the statement 'RTCSTW_List.top++;' It increments this pointer in a loop. It will not point to the next top of another object in the list (another RTCSTW) but to some address that depends on the memory addresses allocated to an item in the RTCSTW_List. Let ListNow be a pointer within the memory block of the list top element. A statement '*RTCSTW_List.ListNow = *RTCSTW_List.top;' will do the following. RTCSTW_List pointer is now replaced by RTCSTW list top pointer and now points to the next list element (object). (Note: RTCSTW_List.top++ for pointer to the next list object can only be used when RTCSTW_List elements are placed in an array. This is because an array is analogous to consecutively located elements of the list at the memory).

Consider a statement: 'while (*RTCSTW_List.ListNow->state != NULL) {numRunning++;}'. When a pointer to ListNow in a list of software timers that are running at present is not NULL, then only execute the set of statements in the given pair of opening and closing curly braces. One of the important uses of the NULL pointer is in the last element of the list to point to the end of a list, or to no more contents in a queue or empty stack, queue or list.

A list is a data structure in which each element (object or data structure) also stores a pointer to the next element at the list. It has one memory block allotted to each element. The list top pointer points to first element and the last element points to NULL.

Table 5.2 summarizes the exemplary uses of queues, stacks, arrays, lists and trees.

Table 5.2 Uses of the Various Data Structures in a Program Element

Data Structure	Definition and when used	Example(s) of its use
Queue	It is a structure with a series of elements with a header element waiting for a read operation, called deletion operation. An operation can be done only in the first in first out (FIFO) mode. It is used when an element is not to be accessible by any index and pointer directly, but only through the FIFO. An element can be inserted only at the end in the series of elements waiting for an operation. There are two pointers, one for deleting after the operation and the other for inserting. Both increment after an operation.	(i) Print buffer. Each character is to be printed in the FIFO mode. (ii) Frames on a network (Each frame also has a queue of a stream of bytes). Each byte has to be sent for receiving as a FIFO. (iii) Image frames in a sequence. (these have to be processed as a FIFO).
Stack	It is a structure with a series of elements with its last element waiting for an operation. An operation can be done only in the last in first out (LIFO) mode. It is used when an element is not to be accessible by any index or pointer directly, but only through the LIFO. An element can be pushed (inserted) only at the top in the series of elements still waiting for an operation. There is only one pointer used for pop (deleting) after the operation as well as for push (inserting). Pointers increment or decrement after an operation. It depends on insertion or deletion.	(i) Pushing of variables and context on interrupt or call to another function. (ii) Retrieving popping the pushed data from a stack.

Data Structure	Definition and when used	Example (s) of its use
Array (one-dimensional vector)	It is a structure with a series of elements with each element accessible by identifier name and index. Its element can be used and operated easily. It is used when each element of the structure is to be given a distinct identity by an index for easy operation. Index starts from 0 and is a positive integer.	$ts = 12 * s(1)$; Total salary. ts is 12 times the first month salary. $marks_weight[4] = marks_weight[0]$; weight of marks in the subject with index 4 is assigned the same as in the subject with index 0.
Multi-dimensional array	It is a structure with a series of elements each having another sub-series of elements. Each element is accessible by the identifier name and two or more indices. It is used when every element of the structure is to be given a distinct identity by two or more indices for easy operation. The dimension of an array equals the number of indices that are needed to distinctly identify an array element. Indices start from 0 and are positive integers.	Handling a matrix or tensor. Consider a pixel in an image frame. Consider Quarter-CIF image pixel in 144×176 size image frame (recall Section 1.2.7.) $pixel[108, 88]$ will represent a pixel at the 108-th horizontal row and the 88-th vertical column. ¹ See the following note also.
List	Each element has a pointer to its next element. Only the first element is identifiable and the list top pointer (header) does it. No other element is identifiable and hence is not accessible directly. By going through the first element, and then consecutively through all the succeeding elements, an element can be read or read and deleted or can be added to a neighbouring element or replaced by another element.	A series of tasks which are active. Each task has pointer of the next task. Another example is a menu that points to a sub-menu.
Tree	There is a root element. It has two or more branches each having a daughter element. Each daughter element has two or more daughter elements. The last one does not have daughters. Only the root element is identifiable and it is done by the treetop pointer (header). No other element is identifiable and hence is not accessible directly. By traversing from the root element, then proceeding continuously through all the succeeding daughters, a tree element can be read or read and deleted or can be added to another daughter or replaced by another element. A tree has data elements arranged as branches. The last daughter, called leaf node has no further daughters. A binary tree is a tree with a maximum of two daughters (branches) in each element and none at leaf-element.	An example is a directory. It has number of file folders. Each file folder has a number of other file folders and so on. In the end is a file.

¹ pixel [0,0] represents the pixel at the left corner on the top and pixel [143, 175] represents that at the right bottom. pixel [10,108, 88] is a pixel data element in a three-dimensional array form. It represents pixels at the same position (108 × 88) in the tenth frame.

The reader may refer to a standard textbook for the C and C++ data structure algorithms.

5.4.4 Use of Modifiers

The actions of modifiers are as follows:

Case (i): Modifier *'auto'* or no modifier means that there is ROM allocation for the variable by locator if it is initialized in the program. RAM is allocated by the locator, if it is not initialized in the program.

Case (ii): Modifier *'unsigned'* is a modifier for a short or int or long data type. It is a directive to permit only the positive values of 16, 32 or 64 bits, respectively.

Case (iii): Modifier *'static'* declaration is inside a function block. Static declaration is a directive to the compiler that the variable should be accessible outside the function block also, and there is to be a reserved memory space for it. It then does not save on a local parameter stack. When several tasks are executed in cooperation, the declaration *static* helps. Consider an exemplary declaration. *'private: static void interrupt ISR_RTI ()*. The static declaration here is for the directive to the compiler that the *ISR_RTI ()* function codes limit to the memory block for *ISR_RTI ()* function. The private declaration here means that there are no other instances of that method in any other object. It then does not save on the stack. There is ROM allocation by the locator if it is initialized in the program. There is RAM allocation by the locator if it is not initialized in the program.

Case (iv): Modifier *static* declaration is outside a function block. It is not usable outside the class or module in which it is declared. There is ROM allocation by the locator for the function codes.

Case (v): Modifier *const* declaration is outside a function block. It must be initialized by a program. For example, *#define const Welcome_Message "There is a mail for you"*. There is ROM allocation by the locator.

Case (vi): Modifier *register* declaration is inside a function block. It must be initialized by a program. For example, *'register CX'*. A CPU register is temporarily allocated when needed. There is no ROM or RAM allocation.

Case (vii): Modifier *interrupt*: It directs the compiler to save all processor registers on entry to the function codes and restores them on their return from the function (this modifier is prefixed by an underscore, *'_interrupt'* in certain compilers).

Case (viii): Modifier *extern*: It directs the compiler to look for the data type declaration or the function in a module other than the one currently in use.

Case (ix): Modifier *volatile* outside a function block is a warning to the compiler that an event can change its value or that its change represents an event. An event example is an interrupt event, hardware event or inter-task communication event. For example, consider a declaration: *'volatile Boolean IntrEnable'*. It changes to false at the start of service by a service routine, if true previously. The compiler does not perform optimization for a *volatile* variable. Let a variable be assigned, $c = 0$. Later, it is assigned $c = 1$. The compiler will ignore the statement $c = 0$ during code optimization and will take $c = 1$. But if c is an event variable, it should not be optimized. $IntrEnable = 0$ is at the beginning of service routine in case an interrupt-enabled variable is used for disabling any interrupt during the period of execution of ISR. $IntrEnable = 1$ is executed before return from the ISR. This re-enables the interrupts at the system. Declaration of *IntrEnable* as *volatile* directs the compiler not to optimize two assignment statements in the same function. There is no ROM or RAM allocation by the locator.

Case (x): Modifier *volatile static* declaration is inside a function block. Examples are: (a) *'volatile static boolean RTIEnable = true'*; (b) *'volatile static boolean RTISWTEnable'*; and (c) *'volatile static boolean RTCSWT_F'*. The static declaration is for directive to compiler that the variable should be accessible outside the function block also, and there is to be a reserved memory space for it. The *volatile* is a directive that cannot optimize as an event can modify. It then does not save onto the local parameter stack of the function. When several tasks are executed in cooperation, the declaration *static* helps. The compiler does not optimize the code because of declaration *volatile*. There is no ROM or RAM allocation by the locator.

5.4.5 Use of Loops, Infinite Loops and Conditions

Sometimes a set of statements is repeated in a loop. Generally, in case of array, the index changes and the same set is repeated for another element of the array. Loops are used when executing a set of statements repeatedly. A loop starts from an initial value or condition and executes till the limiting condition is fulfilled. There can be certain parameter, which changes each time from its initial condition up to a limiting condition.

For example, consider the following. `for (i = 0; i <= 100; i++) { /* a set of statements which repeatedly execute */ }`. The initial condition is assigned as `i = 0` and the last condition for the loop to execute till `i` is less or equal to 100. The set of statements in the bracket executes from start to end and before return to start the `i` increments by 1. The `for` statement allows set of statements to repeatedly execute 101 times with values of `i = 0, 1, ..., 99, 100`.

For another example, consider the following. `i = 0; while (i <= 100) { /* a set of statements which repeatedly execute */; i++; }`. The initial condition is assigned as `i = 0` and is set before the `while` loop. The `while` loop executes till `i` remains less or equal to 100. `i++` increments before the return to test the `while` condition. The `while` statement allows the set of statements to repeatedly execute with values of `i = 0, 1, ..., 99, 100`.

If a condition remains true, then `while` loop will execute infinitely. For example, `while (1) { /* a set of statements which execute repeatedly execute */ }`. The loop will execute infinitely because 1 is always true. Infinite loops are never desired in usual programming. Why? The function or task will never end and never exit or proceed further to the codes after the loop. *Infinite loop is a feature in embedded system programming!* The system software in the telephone has to be always in a waiting loop that finds the ring on line. An exit from the loop will make the system hardware redundant.

Example 5.7 gives a C program design in which the program starts executing from the `main ()` function. There are calls to the functions and calls on interrupts in-between. It has to return to the start. The system main program is never in a halt state. Therefore, the `main ()` is in an infinite loop within the start and end.

Example 5.7

```
# define false 0
# define true 1
/*****
void main (void) {
/* The Declarations here and initialization here */
.
.
/* Infinite while loop follows. Since the condition set for the while loop is always true, the statements
within the curly braces continue to execute */
while (true) {
/* Codes that repeatedly execute */
.
.
}
```

Example 5.8 gives an example for use of polling for an event or message in a program.

Assume that the function `main` has a waiting loop and simply passes the control to an RTOS. Each task controlled by the RTOS will also have codes in an infinite loop. Example 5.9 demonstrates the infinite loops within each task.

Example 5.8

```
# define false 0
# define true 1
/*****
```

```
void main (void) {
/* Call RTOS run here */
rtos.run ( );
while (1) {
/* Infinite while loops follows in each task. So never there is return from the RTOS. */
}
}
/*****/
void task1 (....) {
/* Declarations */
.
.
while (true) {
/* Codes that repeatedly execute */
.
.
/* Codes that execute on an event */
if (flag1) {....}; flag1 =0;
/* Codes that execute for sending message to the kernel */
message1 ( );
}
}
/*****/
void task2 (... ) {
/* Declarations */
.
.
while (true) {
/* Codes that repeatedly execute */
.
.
/* Codes that execute on an event */
if (flag2) {.....}; flag2 =0;
/* Codes that execute for sending message to the kernel */
message2 ( );
};
}
/*****/
.
.
/*****/
void taskN (... ) {
/* Declarations */
```

```

while (true) {
/* Codes that repeatedly execute */

/* Codes that execute on an event*/
if (flagN) {...}; flagN =0;
/* Codes that execute for sending message to the kernel */
messageN ();
};
}

```

There can be more than one infinite loops. The code inside infinite-loop waits for an inter-process-communication (IPC) message or event flag or a set of events through the OS.

The code inside the loop of the running task generates a message that transfers to the kernel. The OS kernel, which passes to the waiting task message, detects it and when that task starts the OS pre-empt the previously running task.

Let an event be setting of a flag, and the flag setting is to trigger the running of a task whenever the kernel passes it to the waiting task. The instruction SWI executes to send the message to the another task function for a service.

Conditional statements are used very often. If a defined condition(s) is fulfilled, the statements within the curly braces after the condition (or a statement without the braces) are executed, otherwise the program proceeds to the next statement or to the next set of statements.

A set of statements is called switch-case. A program switches to a case as per the result of switch expression result. For example, Switch (*i*) means switch as per the case for the value of *i*. Example 5.9 shows an application of infinite loop and switch case statement for programming for GUI in mobile phones (Example 1.5.4).

Example 5.9

Consider a smart mobile phone (Example 1.5.4). Assume that the screen state *j* is between 0 and *K*, among 0, 1, 2, ... or *K* - 1 possible states (set of menus). An interrupt is triggered from a touch screen GUI and an ISR posts an event message *m* = 0, 1, 2, ..., or *N* - 1 as per the selected the menu choice 0, 1, 2, ..., *N* - 1 when there are *N* menu choices for a mobile phone user to select from a screen in state *j*. The *m* will depend on the screen position at the touched position. Figure 5.4 shows the use of a programming model here, which facilitates execution of one of the multiple possible function calls; a function executes after polling for screen state *j* and for a message *m* from an ISR as per the user choice

```

#define true 1
#define false 0

/*****/
void main (void) {
/* declarations */
while (true ) { /* Execute infinite loop */
poll_Screen_State (i); /* Call a function to poll screen state. A state means a set of choices of menu
displayed on the touch screen */

```

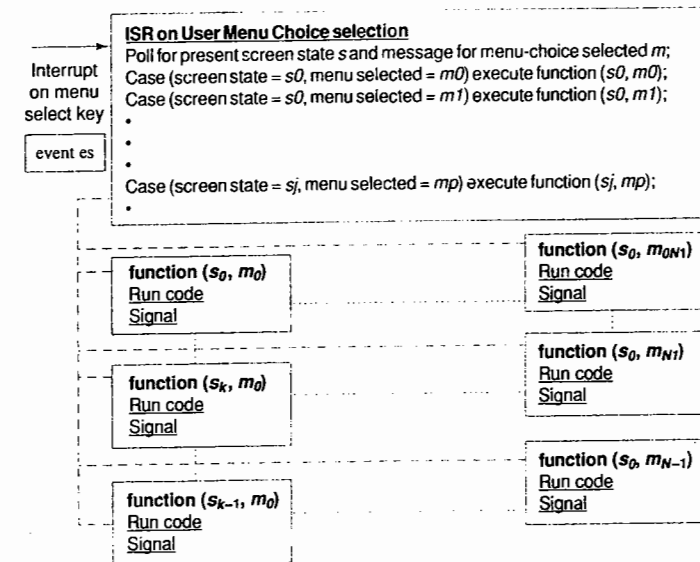


Fig. 5.4 Programming model use here, which facilitates execution of one of the multiple possible function calls and the function executes after polling for screen state *j* and for a message *m* from an interrupt service routine as per the user choice

```

}
/*****/
void poll_Screen_State (i)
/* Let number j identify a screen state */
Switch (i) {
Case 0: poll_menu0 (); exit ()
Case 1: poll_menu1 (); exit ()
.
.
Case j: poll_menuJ (); exit ()
.
.
Case K: poll_menuK (); exit ()
}
}
/*****/
void poll_menu0 { /* Code for polling for choice from menu 0 for screen state 0 */ }
/* An ISR sends message m as per the choice selected by the user from the menu in screen state j */
Switch (m) {
Case 0: { /*Code, which executes when the choice is menu 0 Screen state 0*/; exit ();}

```

```

Case 1: { /*Code, which executes when the choice is menu 1 Screen state 0*/ /*; exit ( );
.
.
Case N – 1: { /*Code, which executes when the choice is menu N – 1 Screen state 0*/ /*; exit ( );
}
}
*****
/* Codes for Screen state 1, 2, ..., j */
.
/* An ISR sends message m as per the choice selected by the user from the menu in screen state j = */
void poll_menuJ { /* Code for polling for choice from menu m for screen state J */
/* An ISR sends message m as per the choice selected by the user from the menu in screen state j */
Switch (m){
Case 0: { /*Code, which executes when the choice is menu 0 Screen state j*/; exit ( );}
Case 1: { /*Code, which executes when the choice is menu 1 Screen state j*/ /*; exit ( );}
.
.
Case N – 1: { /*Code, which executes when the choice is menu N – 1 Screen state j*/ /*; exit ( );}
}
}
*****
/* Codes for Screen state j + 1, 2, ..., K – 1 */
.
void poll_menuK { /* Code for polling for choice from menu m for screen state K – 1 */
}
}
*****

```

4.6 Use of Function Calls

Table 5.1 gave the meanings of the various sets of instructions in the C program. There are special functions for starting the execution of a program, 'void main (void)'. Given next are the steps to be followed when using a function in the program.

1. *Declaring a function:* Just as each variable has to have a declaration, each function must be declared.

Example 5.10

1. Declare a function as follows: 'int run (int indexRTCSWT, unsigned int maxLength, unsigned int numTicks, SWT_Type swtType, SWT_Action swtAction, boolean loadEnable);'. The **run** is the function name. Here *int* specifies the returned data type. There are arguments inside the brackets. Data type of each argument is also declared. A modifier is needed to specify the data type of the returned element (variable or object) from any function. Here, the data type is specified as an integer. (A modifier for specifying the returned element may also be *static*, *volatile*, *interrupt* and *extern*.)
2. Consider a device driver function open (fd, options, device_parameter). The called function name is 'open'. It sets the device configuration. When the function is called by statement, open (4, O_RDWR, 9600). First, second and third arguments that are passed, are 4, O_RDWR and 9600. First argument

is for the device descriptor and passes the value fd = 4. The descriptor is an identity, which is an integer number. The second argument describes the device option setting as read and write device. The third argument describes the device parameter, baud_rate, the rate by which the serial line device is to be configured for UART communication. The same open function is used for the other options and parameters of the device. For example, open (4, O_RDWR, 1200) devices, which means that the device 4 is read only and the device parameter is 1200.

2. *Defining the statements in function:* Just as each variable has to be given the contents or value, each function must have statements. Consider the statements of the function 'run'. These are *within a pair of curly braces* as follows: 'int RTCSWT:: run (int indexRTCSWT, unsigned int maxLength, unsigned int numTicks, SWT_Type swtType, SWT_Action swtAction, boolean loadEnable) {...}'. The last statement in a function is for the *return* and may also be for returning an element or data structure or object.
 3. *Call to a function:* Consider an example: 'if (delay_F == true && SWTDelayIEnable == true) ISR_Delay (100);'. There is a call on fulfilling a condition. The call can occur several times and can be repeatedly made. On each *call*, the values of the arguments given within the pair of brackets pass for use in the function statements. There is only one argument in ISR_Delay.
- Given next are the steps in transfer of values from the arguments of calling function to called function's arguments.
- (1) *Passing the Values (elements):* The values are copied from argument in calling to called function argument. When the function is executed in this way, it does not change a variable's value at the calling function on return from the called function. A function can only use the copied values as its own variables through the arguments.

Example 5.11

Consider a statement, 'run (int indexRTCSWT, unsigned int maxLength, unsigned int numTicks, SWT_Type swtType, SWT_Action swtAction, boolean loadEnable) {...}'. Function 'run' arguments indexRTCSWT, maxLength, numTick, swtType and loadEnable original values in the calling program will remain unchanged during execution of the codes. The advantage is that the same values for use in the remaining instructions are present on return to the calling function. The arguments that are *passed by the values* are saved temporarily on a local parameter stack and retrieved on return from the called function.

- (2) *Reentrant functions call:* Re-entrant function is usable by the several tasks and routines synchronously (at the same time). This is because all its argument values are retrievable from a stack of the local variables, data structures and objects. A function is called re-entrant function when the following three conditions are satisfied.
 - (i) All the arguments pass the values and none of the argument is a pointer (address) whenever a calling function calls it. There is no pointer as an argument in the Example 5.11 of function 'run'.
 - (ii) When an operation is not atomic, the function should not operate on any variable, which is declared outside the function or which an ISR uses or which is a global variable but passed by reference and not passed by value as an argument into the function. (The value of such a variable or variables, which is not local, does not save on the stack when there is a call to another program.)

Let us understand *atomic operation*. The following is an example that clarifies it further. Assume that at a server (software), there is a 32-bit variable *count* to count the number of clients (software) needing service. There is no option except to declare the *count* as a global variable that shares with all clients. Each client on

a connection to a server initiates increment of *count*. Count increment operation should be atomic and till it completes the server disables requests from other clients. Assume that a service routine for real-time clock tick increments the clock ticks in *numTicks*, which is a 64-bit variable. All operations using four or eight bytes of the variable represent one atomic unit. The implementation by the assembly code for increment at that memory location becomes non-atomic in the following situation. Assume that the processor is of 8 bits, and therefore the 32- or 64-bit increment is done in four or eight operations, respectively. Assume that there is no disabling of interrupts or execution of other routines or tasks till all operations for 64-bit increment are complete. Now if an interrupt occurs in-between the incrementing, the wrong values of *count* can be passed.

(iii) That function does not call any other function that is not itself re-entrant.

(3) *Passing the references*: When an argument value to a function passes through a pointer, the called function can change this value. On returning from this function, the new value will be available in the calling program or another function called by this function. This is because there is no saving on stack of a value that either passes through a pointer in the function arguments or operates on the function on a global variable or operates through a variable declared outside the function block.

5.4.7 Multiple Function Calls in Cyclic Order

One of the most common methods is multiple function calls made in a cyclic order in an infinite loop.

Example 5.12

Assume 64 kbps network (Example 4.1). Figure 5.5 shows the model of the multiple function calls.

```
typedef unsigned char int8bit;
#define int8bit boolean
#define false 0
#define true 1

void main (void) {
/* The Declarations of all variables, pointers, functions here and also initializations here */
```

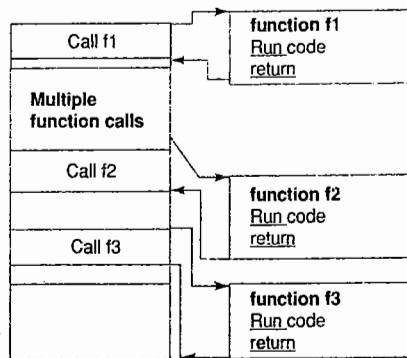


Fig. 5.5 Programming model of multiple function calls

```
unsigned char *portAdata;
boolean charAFlag;
boolean checkPortAChar ( ); /* An interrupt service function to return a Boolean flag, if there is character
received at port A */
/* A declarations for the functions */ void inPortA (unsigned char *);
void decipherPortAData (unsigned char *);
void encryptPortAData (unsigned char *);
void outPortB (unsigned char *);

while (true) {
/* Codes that repeatedly execute */
/* Function for availability check of a character at port A*/
while (charAFlag != true) checkPortAChar ( );
/* Function for reading PortA character*/
inPortA (unsigned char *portAdata);
/* Function for deciphering */
decipherPortAData (unsigned char *portAdata);
/* Function for encoding */
encryptPortAData (unsigned char *portAdata);
/* Function for retransmit output to PortB*/
outPort B (unsigned char *portAdata);
};
}
/*****
/* An interrupt service function to return a Boolean flag, if there is character received at port A */
boolean charAFlag;
boolean checkPortAChar ( );
void inPortA (unsigned char *{...}); /* The ISR, which gets the input character at the address
portAdata */
*****/
```

5.4.8 Function Pointers, Function Queues and ISR Queues

Let the * sign not be put before a function's name and there are arguments within a pair of brackets after the name. The statements for the function execute using the argument values or references for data variables. The statements are present inside a pair of the curly braces. Consider a declaration in Example 5.12, 'boolean checkPortAChar ()';. 'checkPortAChar' is a function, which returns a boolean value. Now, checkPortAChar is itself a pointer to the code's starting address. The address has the codes for statements. The PC will fetch the address of checkPortAChar when a call the function is made, and the CPU sequentially executes the function statements from here.

Now, let the * sign be put before the function's name. '*checkPortAChar' will now refer to all the compiled statements in the memory that are specified within the curly braces.

Consider a declaration in the example, 'void inPortA (unsigned char *);'.

1. inPortA means a pointer to the statements of the function. Inside the bracket, there is an unsigned character pointed by some pointer.

2. *inPortA will refer to all the compiled statements of inPortA.
3. (* inPortA) will refer to calling of statements of inPortA.
4. What will a statement, 'void create (void (*inPortA) (unsigned char *), void *portAStack, unsigned char portAPriority);' mean?
 - a. First modifier 'void' means create a function, which does not return anything.
 - b. 'create' is the function, which can be called after its declaration in a statement.
 - c. Consider first argument of this function 'void (*inPortA) (unsigned char *portAdata)'. (*inPortA) means to call the statements of inPortA the argument of which is 'unsigned char *portAdata'.
 - d. The second argument of create function is a pointer for the portA stack at the memory.
 - e. The third argument of create function is a byte that defines the portA priority.

An important lesson to be remembered from this discussion is that a returning data type specification (e.g., void) followed by '(functionName) (functionArguments)' calls the statements of the functionName using the functionArguments, and on a return it returns the specified data object. We can thus use the function pointer for invoking a call to the functions in a C function.

5.4.9 Queuing of Functions on Interrupts

When there are multiple ISRs, a high priority ISR is executed first and the lowest priority, in the end. (Refer Section 4.5). It is possible that function calls and statements in any of the higher priority interrupts may block the execution of low priority ISR and there may be deadline miss for the low priority ISR. Using the function pointers in the routines, and forming a queue (FIFO) for the function pointers is a solution for the deadline problem for low priority routines. The queued functions are then executed at a later stage. The queued functions are the deferred procedure calls (DPCs) and can be called ISTs when the OS handles them as threads (Section 7.3). Figures 5.6(a), (b) and (c) show the main function, function multiple calls calling the FunctionQueues and ISR, respectively. The figure shows a programming model example in which the multiple function pointers are queued by the ISRs and device-driving ISRs. Each ISR statements have a short set of codes. It executes essential codes within the ISR and rest of the codes in queued functions. Figure 5.6(d) shows queue of function pointers.

Example 5.13

The following is code in which on a return from a service routine, the operation function 'operationFunctionQueues ()' gets the function pointers from the queue and then executes the pointed functions.

```

/* Insert here all preprocessor directives, commands and functions except the main and portA_ISR_Input
() functions. */
void main (void) {
/* The Declarations of all variables, pointers, functions here and also initializations */
.
while (true) {operationFunctionQueues (); /*Call Functions from the Queue in cyclic (Round Robin)
Mode*/
};
}
/*****
void operationFunctionQueues () {
unsigned char *portAdata;

```

```

boolean checkPortAChar ( );
void inPortA (unsigned char *);
void decipherPortAData (unsigned char *);
void encryptPortAData (unsigned char *);
void outPortB (unsigned char *);
void checkPortAChar ( );
/* In_A_Out_B is an array, which inserts the queue elements */
QueueElArray In_A_Out_B = new QueueElArray (QEIType * QelementsArray, 65536);
portAIF = false; portAIEnable = true;
/* Codes that repeatedly execute */
while (portAFlag != true) checkPortAChar (In_A_Out_B, STAF);
In_A_Out_B.QEIReturn ( ); /* Get an In_A_Out_B queue element */
In_A_Out_B.QEIReturn ( ); /* Get next In_A_Out_B queue element */
In_A_Out_B.QEIReturn ( ); /* Get next In_A_Out_B queue element */
In_A_Out_B.QEIReturn ( ); /* Get next In_A_Out_B queue element */
};
/*****Interrupt Service Routine *****/

```

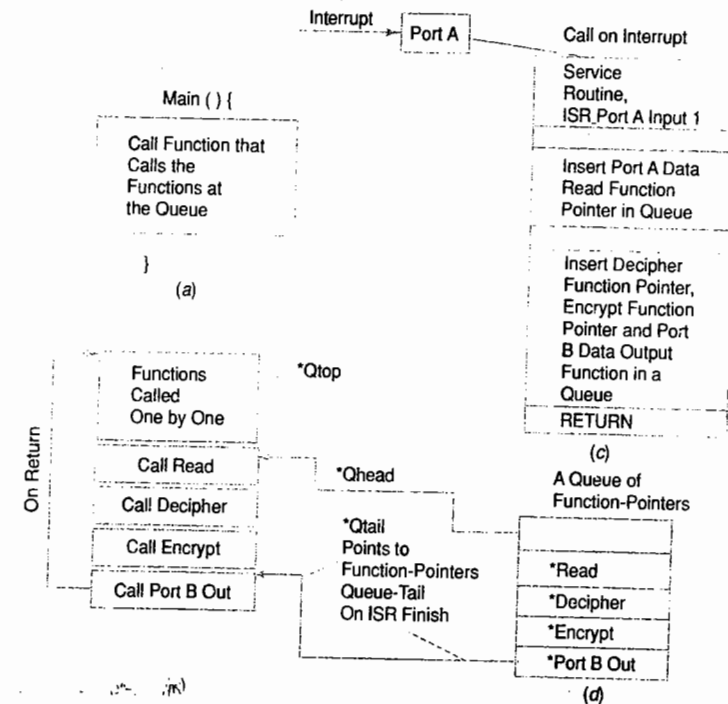


Fig. 5.6 (a) Main () function (b) Function 'operationFunctionQueues ()' (c) Creation of a queue of the function pointers by the interrupt service routine (d) Queue of function pointers

```

void checkPortAChar (QueueElArray In_A_Out_B, volatile boolean portAIF) {
while (portAIF != true) { }; /*Wait till the occurrence of Port A Interrupt */
/* Call ISR_PortAInputf, an Interrupt Service Routine called on the port interrupt */
ISR_PortAInputf (QueueElArray In_A_Out_B);
}
/*****Interrupt Service Routine *****/
void interrupt ISR_PortAInputf (QueueElArray In_A_Out_B) {
disable_PortA_Intr ( ); /* Disable another interrupt from port A*/
void inPortA (unsigned char *portAdata); /* Function for retransmit output to Port B*/
void decipherPortAData (unsigned char *portAdata); /* Function for deciphering */
void encryptPortAData (unsigned char *portAdata); /* Function for encrypting */
void outPortB (unsigned char *portAdata); /* Function for Sending Output to Port B*/
/* Insert the function pointers into the queue */
In_A_Out_B.QElinsert (const inPortA & *portAdata);
In_A_Out_B.QElinsert (const decipherPortAData & *portAdata);
In_A_Out_B.QElinsert (const encryptPortAData & *portAdata);
In_A_Out_B.QElinsert (const outPortB & *portAdata);
enable_PortA_Intr ( ); /* Enable another interrupt from portA*/
}
/*****

```

A programming model concept is that use the function queues and queues of function pointers built by the ISRs. It reduces significantly the ISR latency periods. Each device ISR is therefore able to execute within its stipulated deadline.

5.5 OBJECT-ORIENTED PROGRAMMING

When a large program is made, an object-oriented language offers many advantages. An *object-oriented programming (OOP) language* provides for the following:

1. Defining the object or set of objects, which are common or similar objects within a program and can be used in many programs.
2. Defining the methods that manipulate the objects without modifying their definitions.
3. Creation of multiple instances of the defined object or set of objects or new objects.
4. Inheritance.
5. Data encapsulation.
6. Design of reusable components.

An object can be characterized by the following.

- (a) An *identity* (a reference to a memory block that holds its state and behaviour).
- (b) A *state* (its data, property, fields and attributes).
- (c) A *behaviour* (method or methods that can manipulate the *state* of the object).

In a procedure-based language, like FORTRAN, COBOL, Pascal and C, large programs are split into simpler functional blocks and statements. In an object-oriented language like Smalltalk, C++ or Java, the logical groups (also known as *classes*) are first made. Each group defines the data and the methods of using the data. A set of these groups then gives an application program. Each group has internal user-level fields for the data and the

methods of processing that data at these fields. Each group can then create many objects by copying the group and making it functional. Each object is functional. Each object can interact with other objects to process the user's data. The language provides for formation of classes by the definition of a group of objects having similar attributes and common behaviour. A class creates the objects. An object is an instance of a class.

5.6 EMBEDDED PROGRAMMING IN C++

5.6.1 Advantages of C++

C++ is an OOP language, which in addition, supports the procedure-oriented codes of C. Program coding in C++ codes provides the advantage of OOP as well as the advantage of C and in-line assembly. Programming concepts for embedded programming in C++ are as follows:

1. A class binds all the member functions together for creating objects. The objects will have memory allocation as well as default assignments to its variables that are not declared *static*. Let us assume that each software timer is an object. It gets the count input from a real-time clock. It has a terminal count value after which it generates a software interrupt. It is initialized to a count value. Now consider the codes for a C++ class RTCSWT. A number of software timer objects can be created as the instances of RTCSWT. Each instance of RTCSWT can have different values of present, initial and terminal counts but has identical methods to manipulate the count.
2. A class can derive (inherit) from another class also. Creating a *child* class from RTCSWT as a *parent* class creates a new application of the RTCSWT.
3. Methods (C functions) can have the same name in the inherited class. This is called *method overloading*. Methods can have the same name as well as the same number and type of arguments in the inherited class. This is called *method over-riding*. These are the two significant features that are extremely useful in a large program.
4. Operators in C++ can be overloaded like method overloading. Following statements show how the operators ++ and ! are overloaded to perform a set of operations. (Usually the ++ operator is used for postincrement and preincrement and the ! operator is used for a *not* operation.)

```

const OrderedList & operator ++ ( ) {if (ListNow != NULL) ListNow =
ListNow -> pNext;
return *this;}
boolean int OrderedList & operator ! ( ) const {return (ListNow != NULL)
};

```

(Java does not support operator overloading, except for the 'plus' operator, which is used for summation as well string concatenation.)

There is *struct* that binds all the member functions together in C. But a C++ *class* has object features. It can be extended and child classes can be derived from it. A number of child classes can be derived from a common class. This feature is called polymorphism. A class can be declared as public or private. The data and methods' access are restricted when a class is declared private. *Struct* does not have these features.

5.6.2 Disadvantages of C++

Program codes become lengthy, particularly when following features of the standard C++ are used.

1. Template.

2. Multiple inheritance (deriving a class from many parents).
3. Exceptional handling.
4. Virtual base classes.
5. Classes for IO streams. [Two library functions are *cin* (for character(s) in) and *cout* (for character(s) out.)] The I/O stream class library provides for the input and output streams of characters (bytes). It supports *pipes*, *sockets* and *file management features*.

5.6.3 Optimization of Codes in Embedded C++ Programs to Eliminate the Disadvantages

Embedded system codes can be optimized when using an OOP language by the following

1. Declare private as many classes as possible. It helps in optimizing the generated codes.
2. Use *char*, *int* and *boolean* (scalar data types) in place of the objects (reference data types) as arguments and use local variables as much as feasible.
3. Recover memory already used once by changing the reference to an object to NULL.

A special compiler for an embedded system can facilitate the disabling of specific features provided in C++. Embedded C++ is a version of C++ that provides for a selective disabling of the aforementioned features so that there is a less run-time overhead and less run-time library. The solutions for the library functions are available and ported in C directly. The IO stream library functions in an embedded C++ compiler are also re-entrant. Hence using embedded C++ compilers or the special compilers make the C++ a significantly more powerful coding language than C for embedded systems.

GNU C/C++ compilers (called *gcc*) find extensive use in C++ environment in embedded software development. Embedded C++ is a new programming tool with a compiler that provides a small run-time library. It satisfies small run-time RAM needs by selectively de-configuring features like template, multiple inheritance, virtual base class and so on, when there is a less run-time overhead and when the less run-time library-using solutions are available. Selectively removed (de-configured) features could be template, run-time type identification, multiple inheritance, exceptional handling, virtual base classes, IO streams and foundation classes. [Examples of foundation classes are GUIs (graphic user interfaces). Exemplary GUIs are buttons, checkboxes or radios.]

An embedded system C++ compiler (other than *gcc*) is Diab compiler from Diab Data. It also provides the target (embedded system processor) with specific optimization of the codes. The run-time analysis tools check the expected run-time error and give a profile that is visually interactive.

Embedded system programmers use C++ due to the OOP features of software reusability, extensibility, polymorphism, function over-riding and overloading along with the portability with the C codes and in-line assembly codes. C++ also provides for overloading of operators. Embedded C++ is a C++ version, which makes large program development simpler by providing OOP features of using an object, which binds state and behaviour and which is defined by an instance of a class. We use objects in a way that minimizes memory needs and run-time overheads in the system.

5.7 EMBEDDED PROGRAMMING IN JAVA

5.7.1 Java Programming Basics

Java programming starts from coding for the classes. A class has members. A field is like a variable or struct in C. A method defines the operations on the fields, similar to function in C. Table 5.3 summarizes the basic

uses and the exemplary uses. Class instance fields and instance methods are the members, whose new instances are also created as when the objects are created from the class. Class is a named set of codes that has a number of members – data fields (variables), methods (functions), and so on, so that the object can be created from it. The operations are performed on the objects by passing the messages to objects in OOP. Each class is a logical group with identity, state and behaviour specifications. For an in-depth learning of the programming language, a reader should refer to the standard textbooks and do the required practice exercises.

Table 5.3 Various Elements in a Java Program

Java Program Element	Explanation	Example(s) of its use
Local variable	A variable within a block of codes is defined inside the curly braces and has limited scope.	<pre>{for (int i = 0; int totalOfMarks = 0; i<5; i++) {totalOfMarks += subjectMarks[i];}; return totlaOfMarks;}</pre> Here <i>i</i> is the local variable. The <i>i</i> does not have any scope outside the <i>for</i> loop.
Instance method	Blocks of Java codes, which are given a name, a call (invocation) is made by other Java codes that can also pass (transmit) the needed reference to the values, parameters, and so on.	<pre>findTotalMarks () { };</pre> The method <code>find TotalMarks () { }</code> will also be created in object created from the class.
Instance field	An identifier with a name and using that name a declaration is made in a Java class. It does have a default value and the field is also present in the objects which are instances of the class.	<code>String tele_number;</code> Here, <code>tele_number</code> is an instance field of the class and will also be created in the objects created from that class.
Class	A class is a basic structural unit in a Java program. A class consists of data fields and methods that operate on the fields. A class defines a group of objects with similar attributes and common behaviour and relationships. A class is used to create objects as its instances. It has instance and static fields and methods.	<pre>public class Salaries { public float monthly salary; public float totalSalary; findTotalSalary () { }; }</pre>
Inheritance	Java class inherits members when a Java class is extended from a parent class called super class. The inherited instance fields and methods can be over-riden by redefining them in extended class using same name, arguments and argument-types. Methods can be overloaded by redefining them for different numbers or types of arguments.	<pre>public class AccountDetails extends BankDetails {...};</pre> The class <code>AccountDetails</code> will inherit members of class <code>BankDetails</code> .
Interface	Interface has only the abstract methods and the corresponding static data fields and the methods do not have implementation in the interface. A Java class which is interfaced to an interface implements the abstract methods specified at the interface.	<pre>public class AccountDetails extends BankDetails implements InterestComputations{...}.</pre>

(Contd)

Java Program Element	Explanation	Example(s) of its use
Data types	Java class uses primitive data types: byte (8-bit), short (16-bit), int (32-bit), long (64-bit), float, double, unicode char (16-bit). Java class uses reference data types. A reference can be to the class type in which there are groups of fields and methods to operate on the fields. A reference can be to the array type in which there are groups of objects as array elements.	<pre>byte portData; /* 8-bit port data */ short counts; /* 16-bit count data */ int num Ticks; /* 32-bit number of clock ticks */ String accountNum, eMailId; /* account Number and email ID as String class objects */</pre>
Exception	Java has built-in exception classes. The occurrences of exceptional conditions are handled when exception is thrown. It is also possible to define exception conditions in a program so that exceptions are thrown from try block codes and caught by catch exception method. (Section 4.2.2).	<pre>java.lang.ArrayIndexOutOf BoundExceptions: 0 at addArray (... This throws an exception. java.lang package has an Object java.lang.Throwable. We can also define exceptions in try {..} catch (Exception e){ } Finally { }; (Example 4.6).</pre>

5.7.2 Java Programming Advantages

Java has advantages for embedded programming as follows:

1. Java is completely an OOP language. Java program starts with classes. Application program consists of classes, objects and interfaces.
2. There is a huge class library on the network that makes program development quick.
3. Java has extensibility.
4. Java has in-built support for creating multiple threads. It obviates the need for an OS-based scheduler for handling threads.
5. Java generates byte codes. These execute on an installed JVM (Java virtual machine) on a machine. Virtual machine takes the Java byte codes in the input and runs on the given platform (processor, system and OS). [Virtual machine (VM) in embedded systems is stored at the ROM.] Therefore, Java codes can host on diverse platforms. Platform independence in hosting the compiled codes permit Java for network applications.
6. Platform independence gives *portability* with respect to the processor and OS used. Java is considered as write once and run anywhere.
7. Java is the language for most Web applications and allows machines of different types to communicate on the Web.
8. Java is easier to learn by a C++ programmer.
9. Java does not permit pointer manipulation instructions. So it is robust in the sense that memory leaks and memory-related errors do not occur. A memory leak occurs, for example, when attempting to write after the end of a bounded array.
10. Java does not permit dual way of object manipulation by value and reference. There are no struct, enum, typedef and union. Java does not permit multiple inheritances. Java does not permit operator overloading except for the 'plus' sign used for string concatenation.

5.7.3 Disadvantages of Java

Java has following disadvantages for embedded programming as follows:

1. As Java codes are first interpreted by the JVM, it runs comparatively slowly. This disadvantage can be overcome as follows: Java byte codes can be converted to native machine codes for fast running using just-in-time (JIT) compilation. A Java accelerator (co-processor) can be used in the system for fast code-run.
2. Java byte codes that are generated need a larger memory. An embedded Java system may need a minimum of 512 kB ROM and 512 kB RAM because of the need to first install JVM and run the application.

5.7.4 J2ME

Use of J2ME (Java 2 Micro Edition) or Java Card or Embedded Java helps in reducing the code size to 8 kB for the usual applications like smart card. How? The following are the methods.

1. Use core classes only. Classes for basic run-time environment form the VM internal format and only the programmer's new Java classes are not in internal format.
2. Provide for configuring the run-time environment. Examples of configuring are *deleting the exception handling classes, user-defined class loaders, file classes, AWT classes, synchronized threads, thread groups, multi-dimensional arrays and long and floating data types*. Other configuring examples are adding the specific classes—datagrams, input, output and streams for connections to network when needed.
3. Create one object at a time when running the multiple threads.
4. Reuse the objects instead of using a larger number of objects.
5. Use scalar types only as long as feasible.

JavaCard, EmbeddedJava and J2ME are three versions of Java that generate a reduced code size. J2ME provides the optimized run-time environment. Instead of the use of packages, J2ME provides for the codes for the core classes only. These codes are stored at the ROM of the embedded system. It provides for two alternative configurations, connected device configuration (CDC) and connected limited device configurations (CLDC). CDC inherits a few classes from packages for net, security, io, reflect, security.cert, text, text.resources, util, jar and zip. CLDC does not provide for the applets, awt, beans, math, net, rmi, security and sql and text packages in java.lang. There is a separate javax.microedition.io package in CLDC configuration. A PDA (personal digital assistant) or mobile phone uses CDC or CLDC.

There is scaleable OS feature in J2ME. There is new virtual machine, KVM as an alternative to JVM. When using the KVM, the system needs a 64 kB instead of 512 kB run-time environment. KVM features are as follows:

1. Use of following data types is optional. (a) Multi-dimensional arrays, (b) long 64-bit integer and (c) floating points.
2. Errors are handled by the program classes, which inherit only a few needed error-handling classes from the java I/O package for the exceptions.
3. Use of a separate set of APIs (application program interfaces) instead of JINI. JINI is portable. But in the embedded system, the ROM has the application already ported and the user does not change it.
4. There is no verification of the classes. KVM presumes the classes as already validated.
5. There is no object finalization. The garbage collector does not have to perform time-consuming changes in the object for finalization.
6. The class loader is not available to the user program. The KVM provides the loader.
7. Thread groups are not available.
8. There is no use of java.lang.reflection. Thus, there are no interfaces that do the object serialization, debugging and profiling.

J2ME need not be restricted to configure the JVM to limit the classes. The configuration can be augmented by profiler classes. For example, MIDP (mobile information device profiler) is a profiler class for mobile devices. A profile defines the support of Java to a device family. The profiler is a layer between the application and the configuration. For example, MIDP is between CLDC and application. Between the device and configuration, there is an OS, which is specific to the device needs.

A mobile information device has the following.

1. A touch screen or keypad.
2. A minimum of 96 × 54 pixel colour or monochrome display.
3. Wireless networking.
4. A minimum of 32 kB RAM, 8 kB EEPROM or flash for data and 128 kB ROM.
5. MIDP used as in PDAs, mobile phones and pagers.

MIDP classes describe the displaying text. It describes the network connectivity. For example, HTTP (Internet Hyper Text Transfer Protocol). It provides support for small databases stored in EEPROM or flash memory. It schedules the applications and supports the timers.

An RMI (remote method invocation) profiler is an exemplary profiler for use in distributed environments.

2.7.5 JavaCard and Embedded Java

A smart card (Section 1.10.4) is an electronic circuit with a memory and CPU or a synthesized VLSI circuit. It is packed like an ATM card. For smart cards, there is Java card technology. (Refer to <http://www.java.sun.com/products/javacard>.) Internal formats for the run-time environments are available mainly for the few classes in Java card technology. Only one applet can run and each applet is stateless. Java classes for connections, datagrams, input, output and streams, security and cryptography provide the environment.

There is restricted runtime environment. A smart card simple application uses a JavaCard. The Java advantage of platform independence in byte codes is an asset. The smart card connects to a remote server. The card stores the user account past balance and user details for the remote server information in an encrypted format. It decipheres and communicates to the server the user needs after identifying and certifying the user. The intensive codes for the complex application run at the server.

For EmbeddedJava, refer to <http://www.sun.java.com/embeddedjava>. It provides an embedded run-time environment and a closed exclusive system. Every method, class and run-time library is optional.

Java objects bind the state and behaviour and are instances of a Java class. EmbeddedJava is a Java version, which makes large program development simpler by providing complete OOP features in Java. JVM is configured to minimize memory needs and run-time overheads in the system. Embedded system programmers use Java in a large number of readily available classes for the IO stream, network and security. Java programs possess the ability to run under restricted permissions. JavaCard is a technology for the smart cards and is based on Java.

Summary

- Programming in the assembly language gives the important benefits of precise control of the processors, internal devices and complete use of processor-specific features in its instruction set and addressing modes.

- Program in a high-level language gives the important benefits of short development cycle for a complex system and portability to system hardware modifications. It easily makes larger program development feasible.
- C language support to in-line assembly (fragments of codes in assembly) gives the benefits of both.
- The C program uses various instruction elements, preprocessor directives, macro and constants, including of the source files and header files and functions. Basic C programming elements are the data types, data structures, modifiers, conditional statements and loops, function calls, multiple functions, function queues and service routine queues.
- Infinite looping is a greatly used feature in embedded systems, as it keeps a task or system ready for execution whenever passed a message or signalled to run.
- The C function arguments pass the variable values as well as pass reference to the functions, pointers, NULL pointers and function pointers.
- Queue is an important data structure used in a program. The queue data structure-related functions are 'constructing' a queue, 'inserting' an element into it, deleting an element from it and 'destruction' of the queue. A queue is a FIFO data structure. Queues of bytes play a vital role in a network communication or client server communication also.
- Queuing of pointers to the function on interrupts and later on calling the functions from this queue is a better approach (programming model) as it provides the use of short execution time ISRs.
- Use of 'stack' is very frequent for saving the data in case of interrupts or function calls. Stack-related functions are: 'constructing' a stack, 'pushing' an element into it, popping an element from it and 'destruction' of stack.
- The 'list' and priority-wise 'ordered list'-related functions are 'constructing' a list, 'inserting' an element into it, finding an element from it, deleting an element from it and 'destruction' of the list. One exemplary application is a list of real-time clock interrupts-driven software timers. Another is the list of ready tasks for scheduling the multiple tasks.
- C++ provides all the advantages of C as well as OOP. Its code size can be reduced by optimizing the generated codes as follows: (a) Declaring private as many classes as possible. (b) Using *char*, *int* and *boolean* (scalar data types) in place of objects (reference data types) as arguments and use local variables as much as feasible. (c) Recovering memory once already used by changing reference to an object to NULL. (d) Selectively de-configuring certain C++ features to get less run-time overhead and less run-time library use. (e) Selectively removing the features of template, run-time type identification, multiple inheritances, exceptional handling, virtual base classes, IO streams and foundation classes.
- Java provides the benefits of extensive class libraries availability, modularity, robustness, portability and platform independence. J2ME is Java 2 Micro Edition, which configures and profiles for the small devices. Java Card and Embedded Java is used in smart card and small embedded devices.



Keywords and their Definitions

- Class** : A named set of codes that has a number of members – variables, functions, etc. so that the objects can be created from it. The operations are done on the objects by passing the messages to the objects in OOP. Each class defines a logical group with identity, state and behaviour specifications.
- Class libraries** : Classes for a number of applications like exception, encryption, security, may be provided after thorough debugging and testing for using these in the requirements. Use of class libraries speeds up program development cycle.
- Data structure** : A multi-element structure that can be referenced by a common name (identity).
- Data type** : Type of data for a variable, for example, an integer, and on which only a defined set of operations can be performed.

- Development cycle** : A cycle of coding, testing and debugging. A number of cycles may be needed before finalizing the source codes for porting in the embedded system ROM.
- Exception handling** : A way of calling the functions to handle on development of an exceptional condition. For example, buffer unable to store any further byte. A programmer thinks of the exceptional conditions and provides for the functions and their calling on occurrence (throwing) of the *exception*.
- Foundation classes** : Classes meant for GUIs (e.g., button, checkbox, menu and so on.)
- Function queue** : A queue of pointers for the functions awaiting execution later.
- Header file** : File containing codes (mostly standard functions) for the user. For example, a file 'math.h' containing codes for the mathematical functions.
- High-level language** : Programming language in which it is easier to write codes than in the assembly language, and which also gives the important benefits of short development cycle for a complex system, OOP, data types and many features such as portability to system hardware modifications.
- In-line assembly** : A fragment of codes in assembly language in a high-level language that gives the benefits of processor-specific instructions and addressing modes.
- Include file** : File that is included along with the user source code before the compilation by the compiler.
- Infinite loop** : A loop from the program that cannot exit except on interrupt or on a change in certain parameters used by it or on requiring certain parameter or message or token at certain instruction in the loop.
- IO stream** : A memory buffer created by sending the bytes or characters from a source to a destination so that the destination acts as a sink and accepts them in the sequence that they are sent. An IO stream object does the writing to a file or printer or to a queue or to a network device.
- List** : A data structure into which elements can be sequentially inserted and retrieved, not necessarily in FIFO or LIFO mode. Each element has a pointer also which points to the address of the next element at the list. Last element points to NULL. A top (head) pointer points to its first element.
- Local variable** : A variable defined within a function which no other function can use or modify.
- Memory optimization** : Certain steps changed to reduce the need for memory and having a compact code. It reduces the total size of the memory needed. It may also reduce the total number of CPU cycles, and thus, the total energy requirements.
- Modularity** : A set of codes are said to be modular if they are usable in multiple applications.
- Multiple inheritance** : A daughter (derived class) inheriting the member functions from more than one class.
- NULL** : When a pointer points to NULL, it means that there is no reference to the memory. A memory occupied by an element or object or data structure can be freed by pointing it to the NULL.
- Object-oriented programming** : A programming method in which instead of operations on data types and structures, variables and functions as individuals, the operations are done on the objects. A class creates the objects in C++ and Java.
- Ordered list** : A priority-wise ordered list in which it is easy to delete operations (read and then set the pointer to next). It is done sequentially, starting from top.
- Passing the reference** : From a function, an address of the argument value is passed from the called function when transferring processing to another calling function. The called

- function argument becomes modified when operated and the function may get a different argument value back after return from the calling function. The argument value does not save on the stack when that passes by its reference.
- Passing the value** : From a function, a value is transferred to another function but the same value is reassigned to the original function after return from the called function. Before passing, the argument values save on the stack and retrieves back on return from the function.
- Platform independence** : A code that can port on different machines and OSs.
- Portable** : A code that can be ported in another program by suitable configuration changes.
- Preprocessor directives** : Program statements and directives for the compiler before the main function to include files and define global variable, global macro (section of code), new data type and global constants.
- Private** : A variable belonging to a specific class and not usable outside that class.
- Queue** : A data structure into which elements can be sequentially inserted and deleted in FIFO mode. It needs two pointers, one for the queue tail (back) for insertion and the other for queue head (front) for deletion (read and point to next element).
- Reference data types** : Array and strings are examples of reference data type.
- Robustness** : A program is said to be robust if it can function without errors like stack overflow and out of memory errors. Avoiding pointer manipulation instructions, frequently freeing the memory if not needed later and using exceptions, make a code robust.
- Run-time library** : A library function that links dynamically at the run time. Run-time links increase run-time overheads and out of memory errors can arise.
- Run-time overhead** : Use of RAM for data and stack is called run-time overhead.
- Scalar data types** : The character, integer, unsigned integer, floating point numbers, long and double are called scalar data type. Unlike an array, data consist of one single element.
- Stack** : A data structure in which elements can be pushed for saving in certain memory blocks and can be popped in LIFO mode. It needs one pointer for the stack head (top) for popping (read and point to next element) as well pushing.
- Source code engineering tool** : A power tool to engineer source codes and also to help in debugging and performance analysis of the codes in high-level languages.
- Template** : A set of classes using which new classes are built.
- Virtual base classes** : A special type of class provided in C++.



Review Questions

1. What are the criteria by which an appropriate programming language is chosen for embedded software of a given system?
2. What is the most important feature in C that makes it a popular high-level language for an embedded system?
3. What is the most important feature in Java that makes it a highly useful high-level language for an embedded system in many network-related applications?
4. What is the advantage of polymorphism, when programming using C++?
5. Why do you break a program into header files, configuration files, modules and functions?
6. Design a table to give the features of top-down design and bottom-up design of a program.

7. Explain the importance of the following declarations: static, volatile and interrupt in embedded C.
8. How and when are the following used in a C program? (a) # define (b) typedef (c) null pointer (d) passing the reference (e) recursive function.
9. What are the advantages of using freeware, GNU C/C++ compiler?
10. Why do you need a cross-compiler?
11. Why do you use infinite loop in embedded system software?
12. What are the advantages of re-entrant functions in embedded system software?
13. What are the advantages of using multiple function calls in cyclic order in the main?
14. What are the advantages of building ISR queues?
15. What are the advantages of having short ISRs that build the function queues for processing at a later time?
16. How are the queues used for a network?



Practice Exercises

17. Why do the features in C++ make the code lengthy when using template, multiple inheritance (deriving a class from many parents), exceptional handling, virtual base classes and IO streams? Tabulate the reasons.
18. Write a device driver for a COM serial line port in C including in-line assembly codes.
19. What are the most commonly used preprocessor directives? Give four example of each.
20. How does the use of a macro differ from a function? Explain with exemplary codes.
21. Write program C codes for a loop for summing 10 integers with odd indices only. Each integer is 32 bits. Now unroll the loop and write C codes afresh. Compare the code length in both cases.
22. A set of images in a video frame are to be processed. Which data structure will be best suited for storing the inputs before compressing in an appropriate format?
23. How does combining two functions reduce the memory requirement? Explain with four examples.
24. Consider the format of PPP (point-to-point protocol). Write a C program to transmit PPP data frames encapsulating 4096 data bits. Bits are to be transmitted in a sequence of 32-bit integers stored in memory as in big-endian format.
25. Give two programming examples each in an embedded software, which employs data structures: (a) array (b) queue (c) stack (d) list (e) ordered list (f) binary tree.

Program Modeling Concepts

6

R

1. Two models for programming languages are procedure and object oriented programming (OOP).
2. Procedure-oriented language examples are ALP and C. The C language provides for functions and *main* C function defines the first function that executes and the other functions are called from the *main*. A function can call another function. There can be nesting of function-calls. There can also be multiple function calls within a function. (Example 5.12, Section 5.4.7).

e

c

a

l

l

3. Programming elements in C are preprocessor directives, modifiers, conditional statements and loops, pointers, function calls, multiple functions, function pointers, function queues and ISRs. Program uses data of various types and with various structures: arrays, queues, stacks, lists and trees.
4. OOP language examples are C++ and Java. C++ support object-oriented as well as procedure-oriented codes. Java is purely object-oriented. Object is a reusable software unit and using these units the reusable software components are built. Large complex software can be easily built using the software components.

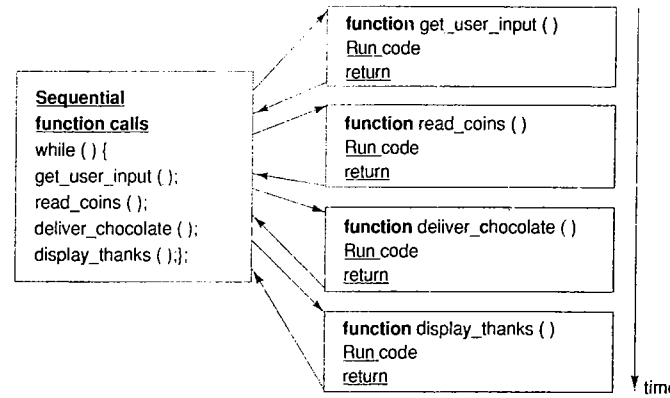


Fig. 6.1 Sequential programming model of an ACVM

L
E
A
R
N
I
N
G
O
B
J
E
C
T
I
V
E
S

A standard design practice adopted by engineers is to use a model when solutions to problems are to be found. The events polling based programming, concurrent processes programming, sequential programming and OOP are the programming models most often used. The objective of this chapter is to learn the important concepts of program modeling. The following concepts of program modeling are explained.

1. Data flow model using data flow graph and control data flow graph.
2. State machine model.

A powerful modeling language is UML based on objects oriented design. UML fulfills the need for a unified language, which can model many types of processes, classes, objects, activities, designs and development process approaches. It should also be understood. An objective of this chapter is to learn the following.

1. UML basic elements.
2. UML diagram.

Example of a use of UML is modeling a software implementation.

Embedded systems may be considered in concurrent processing model as systems with concurrently running processes and the processes may require real-time constraints. Concurrent process model and interprocess communication will be described in Chapter 7.

6.1 PROGRAM MODELS

1. *Polling for events model:* Section 5.4.5 explained this model by Example 5.9 and Figure 5.4. There is polling in cyclic loop for the events, state variables, messages, and signals using the switch-case statements.
2. *Sequential program model:* Example 5.12. Section 5.4.7 gave an example of sequential programming model in which there are sequential multiple function calls within a function. Section 5.4.9 gave another example of sequential program model in which the ISRs provided short period deviations from the sequence for executing short codes and sent function pointers as messages inserted into the queue and then the functions executed in FIFO order.

Example 6.1

Figure 6.1 shows a sequential program model for ACVM (Section 1.5.2). The following functions run in sequence.

1. Run function `get_user_input ()` for obtaining input for the choice of chocolate from the child.
2. Run function `read_coins ()` for reading the coins inserted into the ACVM for the cost of chocolate.
3. Run function `deliver_chocolate ()` for delivering the chocolate.
4. Run function `display_thanks ()` for displaying 'Collect the nice chocolate. Visit again!'

3. *Data flow model:* Data flow graphs, abbreviated as DFGs and control data flow graphs, abbreviated as CDFGs are used for modeling the data paths and program flows of software. A program is modeled as handling the input data streams and creating output data streams. The models based on data flow model concept will be described in Section 6.2.
4. *State machine model:* A programming model is that there are different states and the model considers a system as a machine, which is producing the states. Example 5.9 considered different states, which have different displayed menus and the program action depended on the state. Program sequentially polled for the screen state and menu choice selected by the user. Example 3.6 showed how a key marked 5 can produce on pressing different states (0, 5), (1, 5), (1, j), The transition of a key occurs if it is pressed again within an interval. The state of the key undergoes in a cyclic fashion as: (1, 5) → (1, j) → (1, k) → (1, l) → (1, 5) → (1, j). The models based on state machine concept will be described in Section 6.3.
5. *Concurrent processes and interprocess communication model:* A programming model is that there are several concurrent tasks (or processes or threads) and each task has the sequential codes in infinite loop. A task sends a message or signal for another task. A task, which gets a message or signal, runs and the remaining tasks remain in the blocked state. Example 5.8 gave the exemplary codes. Example 6.2 gives the concurrent process model based program for the sequential program model in Example 6.1. The model of concurrent processes, tasks or threads and interprocess communication between the concurrent processes will be described in detail in Chapter 7.

Example 6.2

Figure 6.2 shows a program model based on concurrent running of the processes in ACVM (Section 1.10.2). Assume that the program consists of following processes, which can run concurrently.

1. Process `get_user_input ()` for obtaining input for the choice of chocolate from the child and signalling to process `read_coins` start.
2. Process `read_coins ()` wait for signal `get_user_input ()` and start reading on signal from for reading the coins inserted in the ACVM for the cost of chocolate. Post a signal to process `deliver_chocolate` to start and also post a signal to process `display_wait ()` to start.

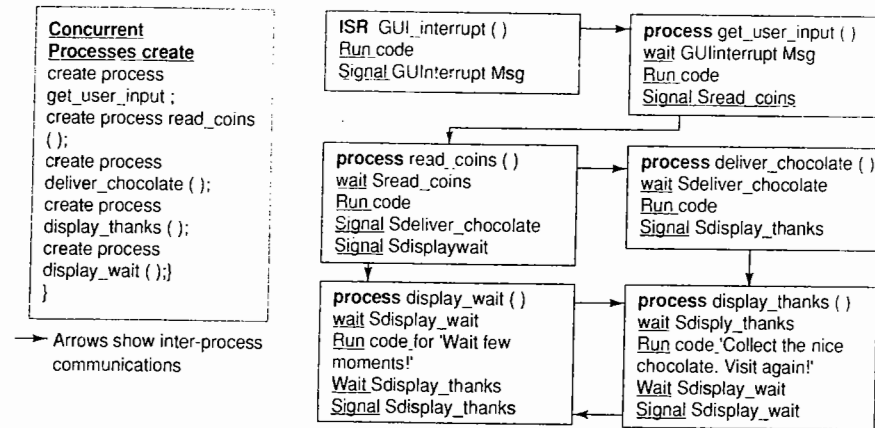


Fig. 6.2 Concurrent processing program model of ACVM

3. Process `deliver_chocolate ()` wait for signal from `read_coins ()` and starts delivering the chocolate and post a signal to `display_thanks ()` to start.
4. Process `display_wait ()` waits for signal from `read_coins ()` and starts displaying 'Wait few moments!' and then wait for signal for `display_thanks ()`.
5. Process `display_thanks ()` waits for signal from `deliver_chocolate ()` and from `display_wait ()` and starts displaying 'Collect the nice chocolate. Visit again!'
6. *OOP model*: Object-oriented language is used for the following features:
 - (a) When there is a need for reusability of the defined object or set of objects that are common within a program or between many applications; when there is a need for abstraction and when, by defining objects by inheritance and polymorphs, new objects can be created. There is data encapsulation within an object.
 - (b) An object is characterized by its identity (a reference to it that holds its state and behaviour), by its state (its data, property, fields and attributes) and by its behaviour (operations, method or methods that can manipulate the state of the object).
 - (c) Defining the logically related group makes a class. Class defines the state and behaviour. It has internal user-level fields for its state and behaviour. It defines the methods of processing the fields.
 - (d) Objects are created from the instances of a class. A class can thus create many objects by copying the group and making it functional. Each object is functional. Each object can interact with other objects to process the states as per the defined behaviour. A set of classes and their objects then gives application-program.

Example 6.3

This example gives an object-based model instead of the ACVM sequential program model and concurrent processes-based model given in Examples 6.1 and 6.2, respectively. Figure 6.3 shows classes and objects,

and inheritance and interface features in a program model based on the ACVM (Section 1.10.2). The following can be the classes and objects.

1. Class `GUI` for graphic-user interaction. It has two methods `display_menu ()` and `get_user_input ()` and for obtaining input for the choice of chocolate from the child. It has method `set_choice ()` to set the choice selected.
2. Class `Read_Coins ()` for reading the coins inserted. It has a method `read ()`, to read one, two and five rupee coins from three ports and a method `sum ()` for summing the total coins.
3. Class `Deliver_chocolate`. It has methods, `get_choice ()` to get the choice and `deliver ()` for delivering the chocolate.
4. Class `MsgDisplay`. It has methods `display_wait ()` and `display_thanks ()` for displaying wait and thank messages.

Class `GUI` is used to create GUI objects as multiple instances of GUI. Class `MsgDisplay` is used to create message display objects as multiple instance of wait and thanks messages. Class `MsgDisplay` can be interfaced to an interface `screen_size ()`, which has an abstract method `screen_size ()`. The abstract method `screen_size ()` is implemented in class `MsgDisplay`

Extending class `MsgDisplay` can specify a new class `MsgTime_Display`. Extended class `MsgTime_Display` inherits all attributes and methods of class `MsgDisplay`. Extended class have another method `display_time_date ()` for displaying time and date also with each message. Extended class can interface to interface `set_display_period`. `MsgTime_Display` will now implement the method `set_display_period ()` to set display period of 1 or 2 minutes for thanks and wait messages.

In the object-oriented approach, there is reusability of defined objects from GUI and a set of objects that are common within a program or between the many applications are created. Also we have abstract methods, `screen_size ()`, and `set_display_period` which are defined in the interfaces but implemented in the interfacing classes. There is inheritance in the new objects, which are created by extending the class `MsgDisplay`. There is encapsulation of methods and attributes in the class and objects.

UML is modeling language based on the object-oriented model. Section 6.5 will describe the UML.

6.2 DFG MODELS

6.2.1 Data Flow Graph

A data flow means that a program flow and all program execution steps are determined specifically only by the data. The software designer predetermines the data inputs and designs the programming steps to generate the data output. For example, a program for finding an *average* of the grades in various subjects will have the data inputs of the grades and data output of the *average*. The program executes a function to generate the appropriate output. The DFG model is appropriate to model the program for the average.

How does data flow in a program? Data that is input after the operations in the program becomes data that is output after a data flow. A diagram called the DFG represents this graphically. A DFG does not have any conditions within it so that the program has one data entry point and one data output point. There is only one independent path for the program flow when the program is executed.

A circle represents each process in DFG. An arrow directed towards the circle represents the data input (or set of inputs) and an arrow originating from the circle represents a data output (or a set of outputs). Data input

along an input edge is considered as token. An input edge has at least one token. The circle represents the node. The node is said to be fired by the tokens from all input edges. The output is considered by the outgoing tokens, which are produced by the node on firing.

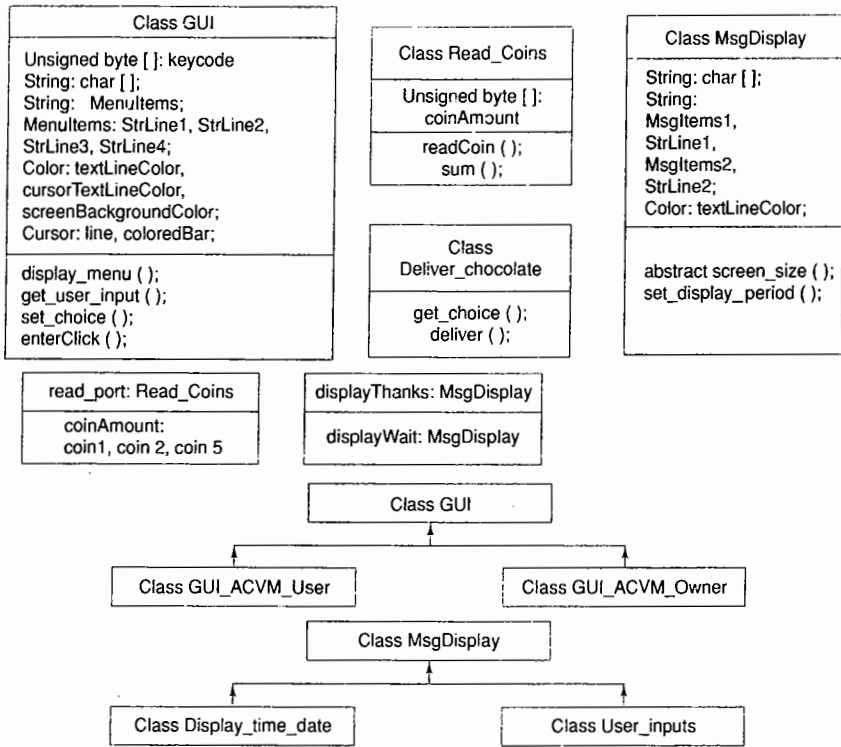


Fig. 6.3 Classes, objects, inheritance and interface features in OOP model based ACVM program

When there is only one set of values of each of the inputs and only one set of values of the outputs for the given input, a DFG is also known as the ADFG, (acyclic data flow graph). All inputs are instantaneously available in APDFG. Examples of non-acyclic data input are as follows: (i) an event; (ii) a status flag setting in a device and (iii) input as per output condition of the previous process.

Example 6.4 gives a DFG during a DSP algorithm.

Example 6.4

Figure 6.4 shows a DFG of the following expression for an output sequence y_6 of a 'finite impulse response (FIR) filter'. An n-th filtered output sequence, $y_n = \sum (a_i \cdot x_{n-i})$ where the sum is made for $i = 0, 1, 2, \dots, N-1$.] Figure 6.4(a) shows the DFG for a process for the sixth FIR sequence and Figure 6.4(b) shows the DFG for a set of processes of the same sequence. Following are the points notable for the process of calculating $y_6 = a_0 \cdot x_6 + a_1 \cdot x_5 + a_2 \cdot x_4 + a_3 \cdot x_3 + a_4 \cdot x_2 + a_5 \cdot x_1 + a_6 \cdot x_0$.

1. There is one input point to the process represented by the circle for calculating y_6 .
 2. There is one output point for y_6 .
 3. There is only one memory address and variable for each coefficient and each filter input. There is only one value of each of the six inputs for x and there is only one value of each of the coefficients, a . (DFG is therefore also the ADFG.)
- The order in which inputs are obtained and the summation is done is also immaterial.

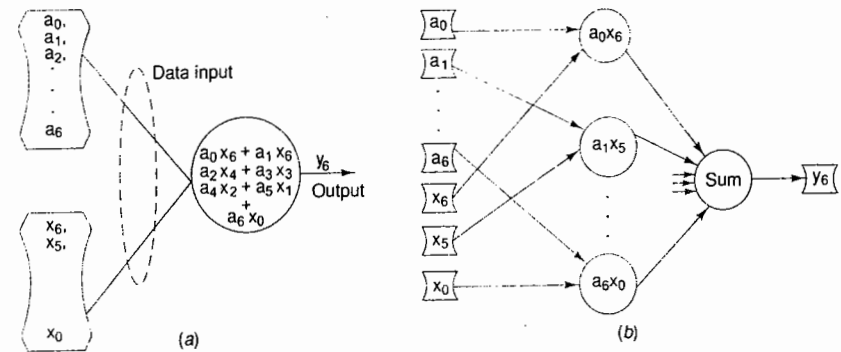


Fig. 6.4 (a) Data flow graph (DFG) for a process for the sixth finite impulse response (FIR) sequence (b) DFG for a set of processes of the same sequence for an FIR filter with 6 inputs and 6 coefficients

It must be noted from Example 6.4 that there is no complexity in the process for y_6 . DFG models help in a simple code design. A simple code design can be defined as that in which the program mostly breaks into DFGs. A DFG models a fundamental program element having an independent path. It gives that unit of a system, which has no control conditions and thus a single path for the program flow. A unit gives the program context and helps in analysing a program in terms of complexity. A more complex program would have a lower number of DFG processes than a simple program.

Figure 6.5 shows a DFG model for the program for saving a picture in a digital camera.

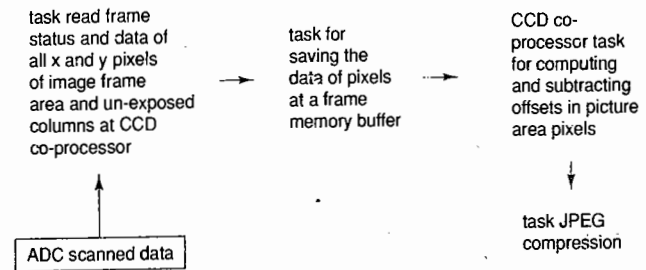


Fig. 6.5 DFG model for program for saving a picture in a digital camera

DFG model program translates and executes as a single-process sequential model program. A program executes as per the input (message or event or set of events) and the input determines the output.

Software implementation becomes greatly simplified when using the DFGs because in the DFG model, there is a single data-in point and a single data-out point, with a process or set of processes that are represented by circle(s). Programming tasks are simplified by representing the code for each process by a circle, using the data input from incoming arrow(s) and generating data output along outgoing arrow(s). When the assignment to an input is fixed in a DFG, it is also called ADFG. Programming complexity is minimized by modeling a program in terms of as many DFGs as possible and the use of as many ADFGs as possible.

6.2.2 Control DFG Model

A control flow means that specifically only the program determines all program execution steps and the flow of a program. The software designer programs and predetermines these steps. How does one design a process that incorporates controls for taking decisions during the data operations and data flow into a program? A process may have the statements that control the inputs or outputs. It may have loops or condition statements in-between (recall Section 5.4.5). Data that is input generate the data output after a control data flow as per the controlling conditions. Output(s) depends on the control statements for various decisions in a process. A CDFG is a diagram, which graphically represents the conditions and the program flow along a condition-dependent path.

The CDFG diagram also represents the effect of events among the processes and shows which processes are activated on each specific event. Here, a variable value changing above a limit or below a limit or falling within a range is also like an event that activates a certain process.

A circle also represents each process (called node) in a CDFG. A directed arrow towards the circle represents the data input (or set of inputs) and a directed arrow from the circle represents a data output (or a set of outputs). A box (square or rectangle with its diagonal axes horizontal and vertical) may represent a condition, for example in Figure 6.6(a). Alternatively, a condition can be marked (or denoted) at the start of the directed arc or arrow. A directed arrow from the box or a marked starting condition determines the action to be taken when the condition is true.

Example 6.5

Figure 6.6(a) shows the controlling input (decision) nodes by the test condition specifying boxes, and the data inputs to a CDFG for an FIR filter with 10 inputs and 10 coefficients [recall Example 6.4 for meanings of various terms in the n -th filtered output sequence, $y_n = \sum(a_i \cdot x_{n-i})$; where the sum is made for $i = 0, 1, 2, \dots, 9$]. Following are the points notable for the process of calculating y_n . There is one input point to the process represented by the circle for calculating y_n .

1. There is one output point for y_n . There is only one memory address and variable for each coefficient and each filter input. These are the variables, i , n , s and e , which take multiple values during the program flow.
2. The order in which inputs are obtained and summation is done does matter.

Figure 6.6(b) shows the controlling input in the In_A_Out_B program of Examples 4.1 and 4.2. Here, instead of boxes, the condition is marked at the start of the arc.

There is increased program complexity in the process for y_n . The CDFG model helps in understanding all conditions and in determining the number of paths a program may take. It also shows us that the software must be tested for each path starting from a decision node, and helps in analyzing the program in terms of complexity.

CDFG model program translates and executes as a concurrent process model program. A controlled decision as per the message or event or set of events determine, which process to execute at an instance.

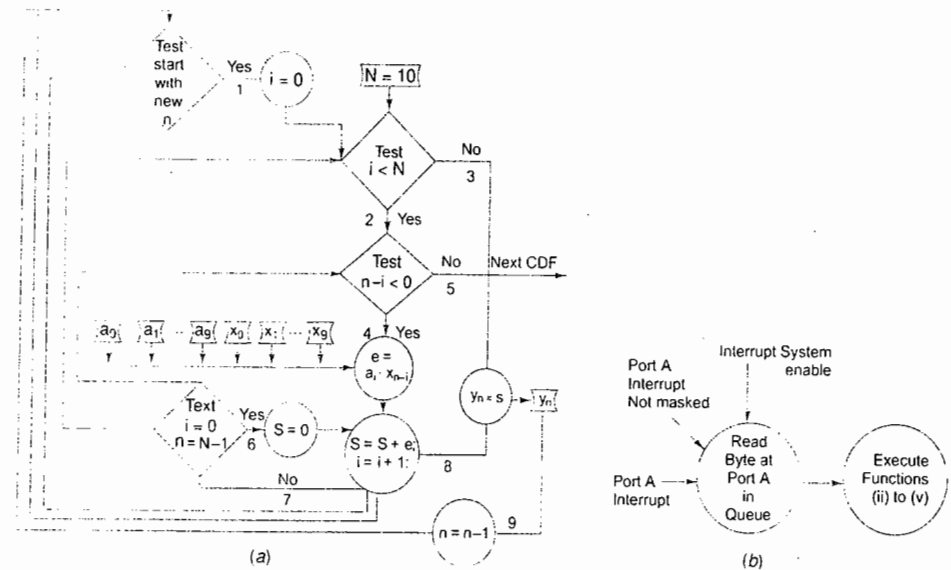


Fig. 6.5 (a) Data inputs and controlling input (decision) nodes shown by test boxes in a Control data flow graph for a finite impulse response filter with 10 inputs and 10 coefficients (b) Controlling input conditions marked in the In_A_Out_B programs in Examples 4.1 and 4.2 (instead of a box, the condition is marked at the start of an arc)

Software implementation becomes simplified when using the specifications of the conditions and decision nodes in the CDFGs that represent the controlled decision at the nodes, and the program paths (DFGs) that are traversed consequently from the nodes after the decisions.

6.2.3 Synchronous Data Flow Graph (SDFG) Model

When there are number of tokens (inputs) required for a computation to generate more tokens (outputs) in a single firing, the data flow is said to be synchronous. The SDFG model is as follows. [Refer E. A. Lee and D. G. Messerschmitt, 'Static scheduling of synchronous data flow', *IEEE Transactions on Computers*, Feb. 1987.] Let an arc represent a buffer in physical memory. The arc can contain one or more initial tokens with the delays. A token does not fire the computations at a vertex till it is received at the vertex. Vertices (circles) in this graph are called the actors. Actors do the computations. An actor also represents a complete DFG within itself. An edge between the vertices (ares with an arrow for the direction) represents a queue of output values from one vertex and a queue of input values to another vertex. Edges carry the values from one actor to another.

Let X and Y be two sets of instructions that once fired (started), do not need any further inputs from any source during the computations. Let X generate the output values (tokens/data) a, b and c. Let Y get the input values (tokens/data), a, c, i and j and let i have a delay. The number of inputs to Y need not equal

the number of outputs from X. Y gets additional inputs and does not need all the outputs from X. These computations and data are now modelled by a directed DFG that exists from X to Y. The number of outputs and inputs are labelled near the arc origin and arc end. Figure 6.7 shows actors (vertices, which does the computations on firing) and arcs in a directed graph between X and Y. The figure shows the outputs a, b and c and inputs a, c, i and j. The i is with a delay (dot). The dot on an arc represents the initial token(s) in an SDFG model. Then an initial token may also represent a *delay* that is shown by a dot on the edges of the SDFG. If there is more than one initial token the number of initial tokens are mentioned on the dot (Figure 6.7). The i and j are initial tokens for the vertex Y in Figure 6.7 which show that i has a *delay*.

A number of vertices may be present in a system. All computations are static scheduled in SDFG execution at each vertex (firing elements for the computations and creating another set of output tokens). SDFG model program translates into a sequential model program.

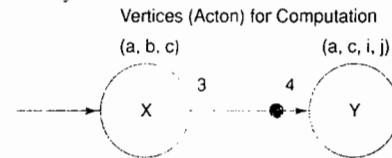


Fig. 6.7 Actors and arcs in a directed graph between X and Y. The outputs a, b and c and inputs a, c, i and j also shown. The i is with a delay (dot)

An SDFG model is like a DFG, but also models the delays as well as the number of inputs and outputs. The edges directed to the circle can be assumed to have a physical memory buffer and till the buffer has the data, the computations do not fire.

6.3 STATE MACHINE PROGRAMMING MODELS FOR EVENT-CONTROLLED PROGRAM FLOW

A state machine is a model in which it is assumed that there are states and state transition functions, which produce the states. A state transition function is a function which changes a state to its next state.

Example 6.6

- Telephone system *idle*, *receiving a ring*, *dialing*, *connected* and *exchanging messages*. There are five finite number of states.
- Consider states of a timer in running state. Figure 6.8 shows the states of timer by circles and state transition by arrows. The count input is the clock input. The changed count value is the output. The output function is the increment in the count value. The state transition function is the time-out on overflow when a predetermined numbers of count inputs are reached. A timer has four finite states: 'idle', 'start', 'running' and 'finished'.
 - 'Idle' state starts state transition on loading an input, *numTicks* (number of ticks after which the timer finishes).
 - 'Running' state: on each clock input for decrement, the count value decrements.
 - 'Finish' state: program flows to the finished state. This is when the count value reaches 0.
- A task has four finite states—*idle*, *ready*, *running* and *finished* [Figure 6.9(a)]. For output from one state, which becomes the input to the next state, tokens (inputs) from the scheduler are *ready flag* and *block flag*. The tokens (outputs) to the scheduler are *running flag*, *blocked flag* and *finish flag*.

- An '*idle state*' to the '*ready state*' transition occurs when the RTOS schedules this task by sending a token (message) to it. Output from this state consists of saving the scheduler context onto the scheduler stack.
- The '*running state*' has instructions being executed and the PC continuously changes as per the program flow.
- Program flows to the '*blocked state*' when the scheduler pre-empts a task. It sends a token (message) to the task. Output from this state consists of saving of the task context at the task stack.
- Program flow to the '*running state*' again occurs on a token from the scheduler and after retrieving the values from the stack.
- The flow to the '*finish state*' happens when the instruction reaches the end stage. The output is a message to the scheduler.
- The flow to the '*ready state*' instead of '*finished state*' occurs when the tasks are in an infinite waiting loop.
- The flow to the '*idle state*' occurs when a message to the scheduler is sent by the task and the task is deleted from the ready list.

When is a system modeled as the states and state machine? Frequently, there are inputs to a program that change the state of the systems to a new state, and generate outputs, which may also be the inputs for the next state. Now it can be assumed that in a model the running of the program and its flow can be considered as running of a machine generating the states. The program flow can be modeled simply by interstate transitions (from one state to another) from next state transition-functions (Moore model) or next output transition-functions (Mealy model).

Following subsections describes finite state machine model.

6.3.1 Finite States Machine (FSM) Model

FSM model states that there is finite number of possible states in a system and a system can only exist in one of these states at an instance. Figure 6.8 showed the *states* modelled as FSM of a timer since there are finite number of timer states. Figure 6.9(a) shows how the states of a *task* can be modelled as a FSM (refer to Section 7.3 for understanding the concept of a task). Figure 6.9(b) shows the FSM states in a program model of an ACVM. There can be transition of the present state to the next state, which depends on the inputs and state transition function. A set of outputs represents a state in the Moore model and a set of outputs represent a state transition in the Mealy model.

Let a circle represent a state and let a directed arc (or, an arrow) represent the program flow from a state to another. When modelling a process as FSM, the software designer specifies the following for each state.

- The state one of the finite number of states.
- Finite set of inputs (tokens or event flags or status flags) with their values for the state.

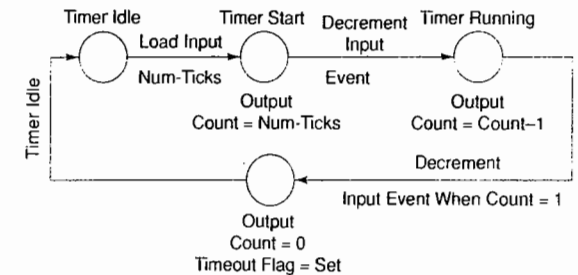


Fig. 6.8 States of a timer using finite states machine model

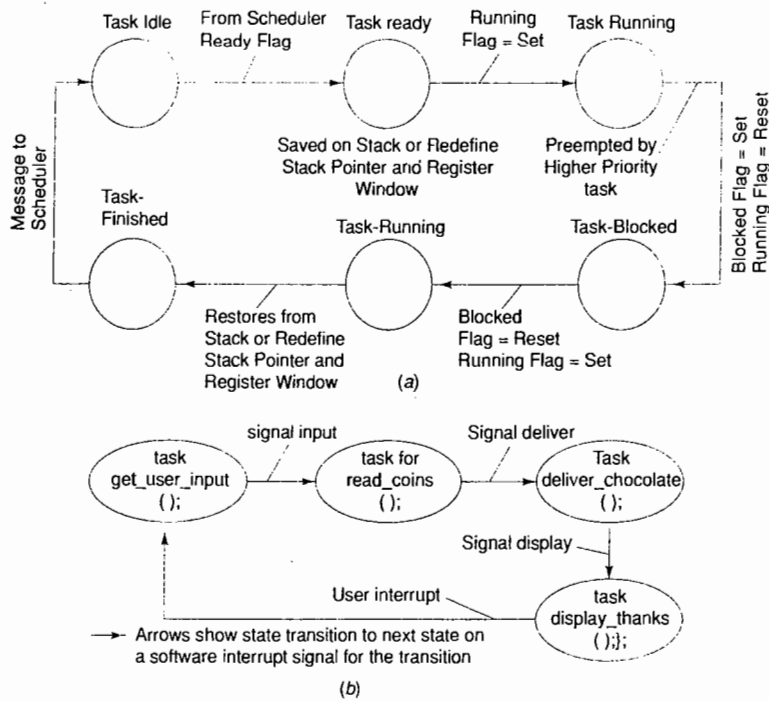


Fig. 5.9 (a) States in the finite states machine (FSM) of a task in a multi-tasking program (b) FSM states in a program model of an ACVM

3. Finite actions (e.g., computations) during the state and finite set of outputs with their possible values (or tokens or event flags or status flags) and an output (action) function for the state that gives the outputs.
4. State transition function for each state to take it to the next state.

The steps that model or represent the states and interstate transitions in FSM data path are as follows.

1. A transition to a new state occurs from the previous state on an event (input). The event may be setting a value of a certain parameter or the result of the execution of certain codes. A transition may be also be interrupt flag-driven (after a flag sets) or semaphore-driven or interrupt source servicing need-drive.
2. A state can receive multiple tokens (inputs, messages, flags interrupts or semaphores) from another state(s). A token (event) is used here as a general term that means either an input or event input. An event input characteristic is that it is asynchronous (one never knows when an event may happen). An event input may happen when there is setting or resetting of a flag. It may occur when there is: (i) a semaphore given or taken or (ii) some indication for a resource or signal or data item generated or (iii) completion of execution of a set of codes. (Refer Sections 7.7, 7.10 and 7.11 for meanings of semaphores and signal)
3. A state can generate multiple tokens (outputs, messages, flag interrupts or semaphores). An output or set of outputs and variables identifies the next state on mapping the inputs, variables and previous states using the output state transition (action) function (Mealy model). A flag indicating the state

condition or a set of codes being executed or a set of values of certain parameters identifies the next state on mapping the inputs, variables and previous states using the next state transition function (Moore model).

When the FSM model is represented graphically with circles and directed arcs, it becomes complex in the case of a complex process with a large number of states. FSM state table can then be helpful.

6.3.2 FSM State Table

To design a software using the FSM model, a state table can be designed for representation of every state in its rows. The following columns are made for each row.

1. Present state name or identification.
2. Action(s) at the state until some event(s).
3. The events (tokens) that cause the execution of the state transition function.
4. Output(s) from the state output function(s).
5. Next State.
6. Expected time interval for finishing the transitions to a new state after the event.

The coding using each row can now be easily done as follows.

```
while (presentState) {action ( ); if (event = .....; token = .... )
    {output = .....; stateTransitionFunction ( ); ;}}
```

or

Switch (State)

```
Case presentState: action ( ); if (event = .....; token = .... )
    {output = .....; stateTransitionFunction ( ); ;};'
```

Here presentState is a boolean variable, which is true as long as the present state continues and turns false on transition to the next. The action () is a function that executes at the state. If certain events occur and tokens are received (e.g., clock input in a timer), a state transition function, stateTransitionFunction, is executed which also makes presentState equals false and transition occurs to the next state by setting nextState (a boolean variable) equals true.

Example 6.7

Figure 6.10 shows the states, state transitions, events, outputs from state output function and finite number of state transitions of a mobile phone key '5' of T9 keypad (Example 3.6). A mobile phone T9 keypad's key marked 5 has five states. It undergoes transitions from initial state (0, 5) as follows: (0, 5) → (1, 5) → (1, j) → (1, k) → (1, l) → (1, 5) → (1, j). First state, variable $s_0 = 0$ represents initial key inactive state. When it is 1, it represents the active state. Second-state variable $s_1 = 5$ or j or k or l represents the final character, which is accepted as output after 1-second delay. A transition occurs when pressed again within a period less than 1 second. The timer has count register, compare register and two flags—TR (timer running) and TF (time compare output) flags. State transition occurs when key input interrupt flag KF is equal to set or TF = 1. The transition on TF resets the key. The timer starts on key interrupt and when timer is in ON state, timer-run flag TR equals 1. When the timer is off or timer time outs, TR = 0. When count = x, timer flag TF equals false before timeout. Timeout occurs when count x equals compare registerth loaded for 1 second. After 1 second, the TF equals 1. There are 12 finite number of states of the key and the timer together. Examples 6.8 and 6.9. give the state table and state machine program.

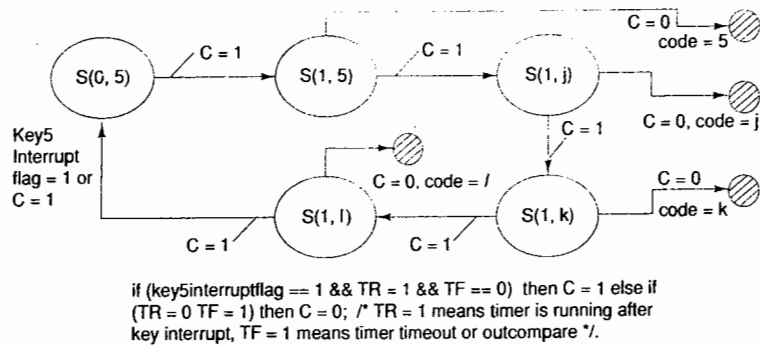


Fig. 6.10 The states, state transitions and finite number of state transitions in a key '5' in mobile phone T9 keypad

Example 6.8

Make state table for FSM in Example 6.7. Table 6.1 gives the state table for the key '5' in T9 keypad.

Table 6.1 State Table for the Key '5' in T9 Keypad

Present State		Action			Events			Next State			Output
Key	TR	TF	KF		KF	Count	Key	TR	TF	KF	
(0, 5)	0	0	0	Wait	1	x'	(1, 5)	1	0	0	Timer start
(1, 5)	1	0	0	Wait	1	x	(1, j)	1	0	0	Timer restart
(1, j)	1	0	0	Wait	1	x	(1, k)	1	0	0	Timer restart
(1, k)	1	0	0	Wait	1	x	(1, l)	1	0	0	Timer restart
(1, l)	1	0	0	Wait	1	x	(1, 5)	1	0	0	Timer restart
(1, 5)	0	1	0	-	0	0	(1, 5)	0	0	0	Timer reset
(1, j)	0	1	0	-	0	0	(1, j)	0	0	0	Timer reset
(1, k)	0	1	0	-	0	0	(1, k)	0	0	0	Timer reset
(1, l)	0	1	0	-	0	0	(1, l)	0	0	0	Timer reset

Note: Count x' initial count register value, = x means counts greater than x' and less than a compare register value set for 1 second time out. TR = 0 means timer stopped. TR = 1 means timer running after loading a value in compare register for capture time out after 1 second. TF = 1 means timer compare time out. KF = 1 key press event. KF = 0 means key reset or inactive.

Example 6.9

The C codes from Table 6.1 can be written as follows.

```
# define true 1
# define false 0
```

```
# define initialState '05000'
# define state1 '15100'
# define state2 '1j100'
# define state3 '1k100'
# define state4 '1l100'
# define state5 '15010'
# define state6 '1j010'
# define state7 '1k010'
# define state8 '1l010'
# define state9 '15000'
# define state10 '1j000'
# define state11 '1k000'
# define state12 '1l000'

void Key5FSM ( ) {
char [ ] state;
initialState = "05000"
while (true) { /* An infinite loop */
/*
/* function display ("x") shows character x on the screen and function cursor_next ( ) moves the
cursor position to next when keyiing in an SMS text message. SWI is software interrupt instruction */
Switch (State) {
/*****
initialState: if ((KF == 1) && Count == 0)) {
SWI timerstart; /* Execute Interrupt routine to start the timer */
display ("5"); State = State1;}
break;
/*****
State1: if ((KF == 1) && Count == x) {
SWI timerRestart; /* Execute Interrupt routine to restart the timer */
display ("j"); State = State2;}
break;
/*****
State2: if ((KF == 1) && Count == x) {
SWI timerRestart; /* Execute Interrupt routine to restart the timer */
display ("k"); State = State3;}
break;
/*****
State3: if ((KF == 1) && Count == x) {
SWI timerRestart; /* Execute Interrupt routine to restart the timer */
display ("l"); State = State4;}
break;
/*****
State4: if ((KF == 0) && Count == x) {
```

```

SWI timerRestart; /* Execute Interrupt routine to restart the timer */
display ("I"); State = State5;}

break;
/*****
State5: if ((KF == 0) && Count == 0) {
    SWI timerReset; /* Execute Interrupt routine to reset and stop the timer */
    display ("5"); cursor_next ( ); State = State9;}

    exit ( );
/*****
State6: if ((KF == 0) && Count == 0) {
    SWI timerReset; /* Execute Interrupt routine to reset and stop the timer */
    display ("j"); cursor_next ( ); State = State10;}

    exit ( );
/*****
State7: if ((KF == 0) && Count == 0) {
    SWI timerReset; /* Execute Interrupt routine to reset and stop the timer */
    display ("k"); cursor_next ( ); State = State11;}

    exit ( );
/*****
State8: if ((KF == 0) && Count == 0) {
    SWI timerReset; /* Execute Interrupt routine to reset and stop the timer */
    display ("l"); cursor_next ( ); State = State12;}

    exit ( );
/*****
}
/* ----- End of Switch-case ----- */
} /* End of While infinite loop */
} /* End of Key5FSM */
    
```

FSM model assumes the finite number of states and reduces the programming tasks to the following: (i) coding for each state transition function and each output function: (ii) knowing the time periods taken by the process at each state transition function and between each state, when programming for real time. The FSM model is appropriate for one process at a time, for the sequential flows from one state to the next state and for the controlled flow of the program. When using the FSM model, a state table representation becomes very handy while coding for state machine.

6.4 MODELING OF MULTIPROCESSOR SYSTEMS

6.4.1 Multiprocessor Systems

A large complex program can be partitioned into the tasks or sets of instructions (or processes or threads) and the ISRs. The tasks and ISRs can run concurrently on different processors and by some mechanism the tasks can communicate with each other.

Example 6.10

1. Assume a large program has four tasks: task 1, task 2, task 3 and task 4. It has four ISRs: ISR_A, ISR_B, ISR_C and ISR_D. Assume a processor PA is statically scheduled to run task 2, task 4, ISR_B and ISR_D. Processor PB is statically scheduled to run task 1, task 3, ISR_A and ISR_C. Figure 6.11(a) shows the scheduling on the processors.
2. Assume a large program has four tasks: task 1, task 2, task 3 and task 4. It has 4 ISRs: ISR_A, ISR_B, ISR_C and ISR_D. Assume a processor has dual core with one core statically scheduled to run the tasks and the other the ISRs. ISRs send the messages to the tasks running on the other core. Figure 6.11(b) shows the scheduling on a dual-core processor.

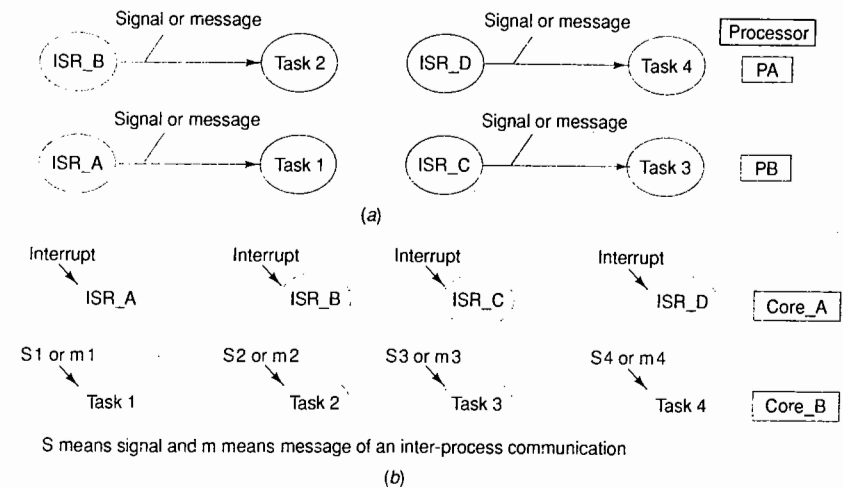


Fig. 6.11 (a) Static scheduling of tasks and interrupt service routines on two processors (b) Static scheduling on two processor cores

The problem is how to partition the program into tasks or sets of instructions between the various processors, and then how to schedule the instructions and data over the available processor times and resources so that there is optimum performance. Should there be static scheduling for running one task on one processor? Then, suppose one processor finishes computations earlier than the other. What is the performance cost? Performance cost is more if there is idle time left from the available. What is the performance cost if one task needs to send a message to another and the other waits (blocks) till the message is received? Following are the problems in modeling the processing of instructions in a multiprocessor system.

1. Partitioning of processes, instruction sets and instruction(s).
2. Concurrent processing of *processes* on each processor.
3. Static scheduling by the compiler, analogous to scheduling in a superscalar processor. (Each superscalar processor has multiple processing units in parallel.)
4. When superscalar units are present in a processor, it means two or more pipelines of instructions are executed in parallel. Pipeline has a number of stages (3 to 9) and different instructions are at different

stages. Problem is then not only of scheduling of concurrent processing instructions on different processors but also scheduling of concurrent processing instructions on each superscalar unit and pipeline in the processor.

5. **Hardware scheduling issue**, for example, whether static scheduling of hardware (processors and memories) is feasible or not (it is simpler and its use depends on the types of instructions when it does not affect the system performance).
6. **Static scheduling issue** (e.g., when the performance is not affected and when the processing actions are predictable and synchronous).
7. **Synchronizing issues**, synchronization means the use of interprocessor or process communications (IPCs) such that there is a definite order (precedence) in which the computations are fired on any processor in a multiprocessor system (IPC is a message or signal to another process or processor so that it can proceed further. Section 7.9 will describe the IPC in detail).
8. **Dynamic scheduling issues** (e.g., the performance is affected when there are interrupts and when the services to the tasks are asynchronous. It is also relevant when there is pre-emptive scheduling as that is also asynchronous).
9. **Scheduling of the instructions**, SIMDs (single instruction multiple data), MIMDs (multiple instructions and multiple data) and VLIWs (very large instruction words within each process) and scheduling them for each processor.

There are several methods of scheduling and synchronizing the execution of instructions, SIMDs, MIMDs and VLIWs in the system. In a multiprocessor system, scheduling is done after analysing the scheduling and synchronizing options for the concurrent processing and scheduling of instructions, SIMDs, MIMDs and VLIWs.

Consider two processors, PA and PB, interfaced with the memory in a system. *Case 1*: Processors share the same address space through a common bus, called tight coupling between processors. *Case 2*: Processors have different autonomous address spaces (like in a network) as well as shared data sets and arrays, called loose coupling. Figure 6.12(a) and (b) show both the cases. *Case 3*: Processors share the memories in alternative bus architecture, for example, three-dimensional mesh, ring, torrid or tree in place of a shared bus between the different tightly coupled processors. Processors process concurrently as follows:

1. One way of concurrent processing is to schedule each task so that it is executed on different processors and synchronize the tasks by some interprocessor communication mechanism.
2. The second way is, when an SMID or MIMD or VLIW instruction has different data (e.g., different coefficients in Example 6.5), each task is processed on different processors (tightly coupled processing) for different data. This is analogous to the execution of a VLIW in TMS320C6, a Texas Instruments DSP series processor. It employs two identical sets of four units and a VLIW instruction word can be within 4 and 32 bytes. It has instruction level parallelism when a compiler schedules such that the processors run the different instruction elements at the different units in parallel.
Note: The compiler does *static scheduling for VLIWs*. Static scheduling is one in which a compiler compiles such that the codes are run on different processors or processing units as per the schedule decided and this schedule remains static during the program run even if a processor waits for others to finish the scheduled processing.
3. An alternate way is that a task instruction is executed on the same processor or different instructions of a task can be done on different processors (loosely coupled). A compiler schedules the various instructions of the tasks among the processors at an instance.

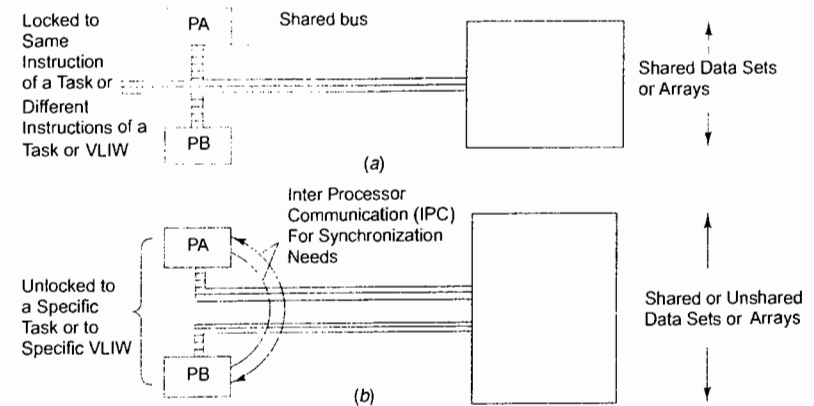


Fig. 6.12 (a) Tightly coupled processors sharing the same address space while processing multiple tasks (b) Loosely coupled processors having separate autonomous address spaces as in a network as well as shared address space for data sets and arrays

6.4.2 Model of Unfolding SDFGs into Homogeneous SDFGs

An SDFG models the delays as well as the number of inputs and outputs (Section 6.2.3). The edges directed to a circles are assumed to have a physical memory buffer and till the buffer has the data, the computations do not fire. When there is only one token at the input, and one at the output, an SDFG is called homogenous SDFG (HSDFG). Figure 6.13(a) shows a modeling of computations by an SDFG. Figure 6.13(b) shows an HSDFG representation after unfolding the SDFG in Figure 6.13(a). The dot and label over the edge show delayed two number input tokens at vertex Y.

For example, suppose that the outputs from vertex X' (a set of computations) is a and input to Y' (another set of computations) is also a. An SDFG can therefore unfold into a HSDFG. An SDF graph can be unfolded into one or more HSDFGs. Two vertices can be connected by two or more edges in the HSDF graph. An HSDF graph will naturally have more vertices and edges than an SDFG because only one token is permitted at a vertex.

When there is an indefinitely long data sequence, SDFG-based modelling and the consequent unfolding into the HSDF graphs helps. For example, HSDFGs applied to the computations of a fast Fourier transform or for coding voice data. An HSDF graph can also effectively model an IPC (interprocessor communication) graph. All computations are static scheduled in HSDFG execution at each vertex (firing elements for the computations and creating another set of output tokens). Let there be a sequence of computations that are fired at the vertices. Let *precedence* in a directed graph define the computations order by which the vertices are placed first, then next, and then next to next. A sequence on one processor among the set of processors can be delayed at the arcs. Input from another processor (initial token) can also be delayed. A SDF model program then translates into a number of parallel concurrent or sequential model programs using HSDFGs.

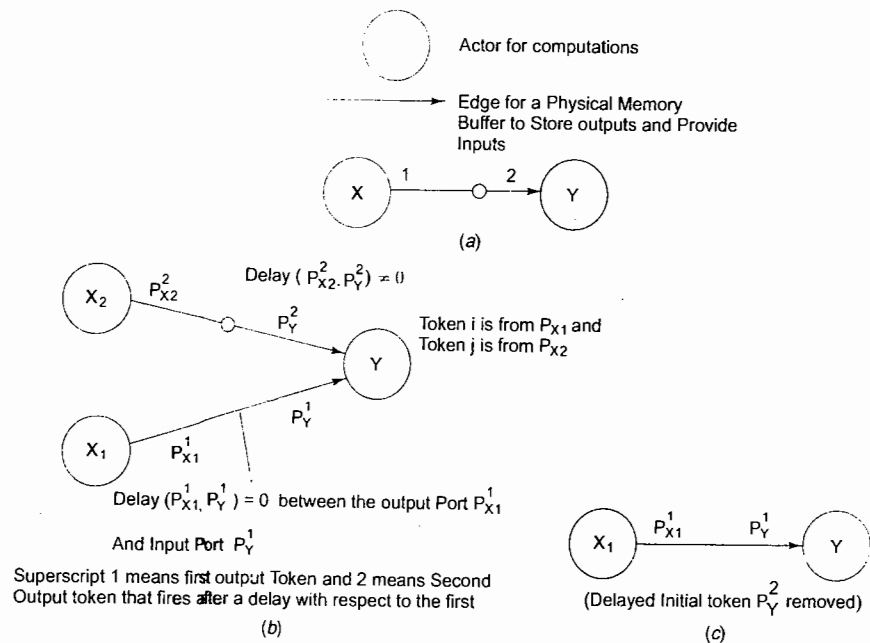


Fig. 5.13 (a) A modeling of computations by an SDFG. The dot and label over the edge show delayed two number input tokens at vertex Y (b) A homogeneous SDFG representation after unfolding the SDFG (c) An APEG representation from an HSDFG after removing the delayed edge

Multiprocessor system computations and their firing instances can be modeled. Modeling simplifies the programming, scheduling and synchronizing of the processes. HSDFG models are like an SDFGs but have the feature that there is only one token that delays along an edge (arc or arrow) because there is only one token at input, and one at output.

5.4.3 Model of Unfolding HSDFGs into APEGs

Acrylic precedence is a precedence of vertices in a directed graph such that there are no delays at the arcs. If initial tokens (delays) are taken off from an HSDFG graph, an acrylic precedence expansion graph (APEG) is obtained.

APEGs are important for scheduling in multiprocessor systems. An APEG not only has along the arc, starting inputs identical to the output from a previous vertex, but also no delaying for the token. Hence, the execution is smooth along the arc with no interprocessor communication time. An APEG-based algorithm becomes the simplest to schedule such that the precedence constraints in the algorithm remain the same as before. Figure 6.13(c) shows a corresponding APEG that is a graph with no delays. It derives from a HSDFG [Figure 6.13(b)] or SDFG [Figure 6.13(a)].

A task-level concurrent process as well as an IPC graph can be modeled using APEGs and HSDFGs. A thread running on one processor modeled as APEG can pass a control to another by blocking itself or by sleeping, but the sequence and process flow along the APEG remain intact. Example 6.11 explains this.

Example 6.11

Let $V_1, V_2,$ and V_3 be the computation vertices assigned to processor PA. Let V''_1, V''_2, V''_3 be the computation vertices assigned to processor PB concurrently processing with PA. An IPC is needed when algorithm (or set of computations) V''_3 cannot proceed till there is a message (token) from V_2 . Let IPC be between V''_3 and V_2 . This synchronizes the processes at PA and PB through the IPC. Figure 6.14 shows one APEG and one HSDFG with an IPC to PB from PA.

APEG models are such that there are no delays during execution at any stage in an APEG or chain of APEGs. Complex problems are therefore first modelled as the SDFGs, then SDFGs are unfolded into HSDFGs and HSDFGs are separated into APEGs. Processing is as per precedence constraints between the APEGs. APEG-based algorithms become the simplest to schedule but precedence constraints in the algorithm among its APEGs remains the same as before.

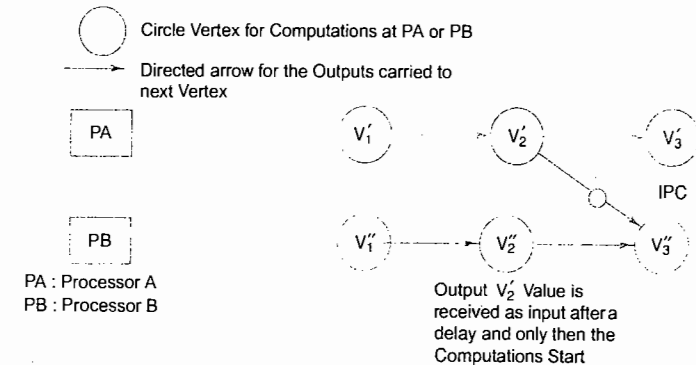


Fig. 6.14 A two-processor system with one acrylic precedence expansion graph and one homogeneous synchronous data flow graph with an IPC to PB from PA

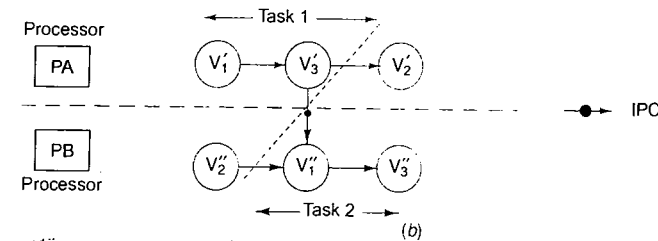
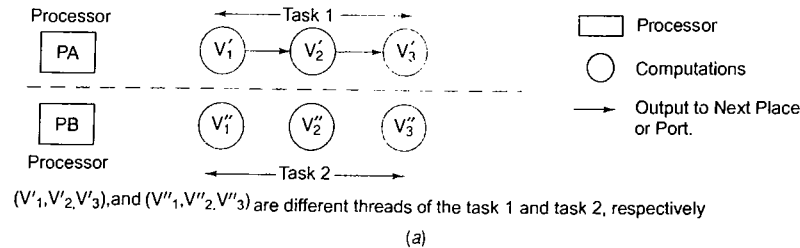
6.4.4 Applications of the Graphs to Multiprocessor Systems: Partitioning and Scheduling

When there are multiple processors in parallel, the partitioning of a program is done as follows.

1. There are minimum number of IPCs so that the total time of IPC delays (waiting periods) minimized.
2. There is load balancing. Each processor has the least waiting time by sharing the processing load.
3. The performance cost minimizes. Performance cost means the execution time required (i) for computations for the tokens and delays at the edge (communication time), (ii) the computation time before firing (computations) by a vertex (transition) and (iii) context switch time.

Consider Figure 6.15 vertices. At each vertex computations occur such that the precedence constraints maintain (remain intact). The graph of a program thus partitions into the functions or tasks or threads. One of the three following strategies can schedule a program for running.

1. Each task or function is executed on an assigned processor. Each task or function is executed on different processors at different periods. Instructions of four different tasks are partitioned on two processors. Instructions of four different tasks are partitioned and scheduled on two processors differently in different periods [Figures 6.15(a) to (d)] show these four partitioning and scheduling strategies.



c'_{ij} is an instruction in V'_i to take i -th column and j -th row element of matrix A and add bits with the Corresponding Element in B.

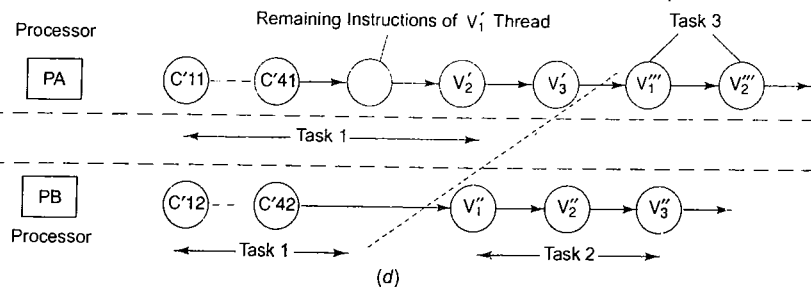
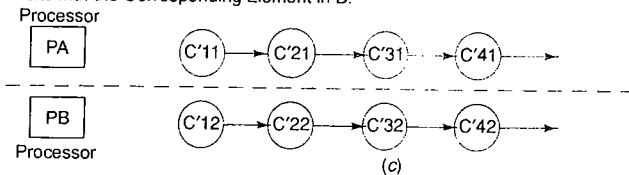


Fig. 6.15 (a) Each task or function is executed on an assigned processor (b) Each task or function is executed on different processors at different periods (c) Instructions of four different tasks are partitioned on two processors (d) Instructions of four different tasks are partitioned and scheduled on two processors differently during different periods

2. Each set of data is partitioned in a VLIW instruction and is executed on the different processors, which execute the same program. Consider a matrix addition process. Each row can be added on a different processor when the data of the rows are partitioned among the processors. Such data partitioning is preferred when processing a DSP-VLIW.

A combined partitioning is done both at the data level as well as the task (or function) level. Different functions themselves may run concurrently on different processors but at the micro or atomic-level data is partitioned and the instructions are run.

Partitioning and scheduling of vertices can be done in a number of ways. (i) Each task or function is executed on an assigned processor. (ii) Each task or function is executed on different processors at different periods. (iii) Instructions of four different tasks are partitioned on two processors. (iv) Instructions of four different tasks are partitioned and scheduled on two processors differently in different periods. (v) Data partitioning in case of SIMDs, MIMDs and VLIWs.

6.5 UML MODELLING

Recapitulate Section 5.5. The concepts used in object-oriented language are also used in software designing. Object-oriented designing is also done as before.

1. Object-oriented design is done when there is a need for reusability of the defined software components as object or set of objects (reusable components). The new component can be abstracted from the existing. New components and object designs are created by the object inheritances and polymorphs. There is information encapsulation within a designed component or object.
2. A designed component object is also characterized by its identity (a reference to it that holds its state and behaviour), by its state (its designs for data, property, fields, attributes and algorithms) and by its behaviour (method or methods that can manipulate the state of the design).
3. New object designs are created from the instances of a designed class. Class defines the state, attributes, operations and behaviour of a design concept. It has internal user-level fields for its state and behaviour. It defines the ways of using the designs.
4. A designed class can then create many component objects (designs) by copying the group and making designs functional. Each design is a functional design. Each object design can interface with other designs to process the states as per the defined behaviour.
5. A set of classes then gives the complete software design for a system.

UML is a unified (common) modeling language for any general system for which object-oriented analysis and design are feasible and which can be abstracted by models. Unification in UML means its common applicability to many designs or processes. We can then model the following by a similar set of diagrams: (i) software visualizing, (ii) data design(s), (iii) algorithms design(s), (iv) software design(s), (v) software specifications, (vi) software development process, (vii) an industrial process.

UML is a language for modeling. Details of the language can be learnt from a standard textbook. [For example, "The Unified Modeling Language User Guide" by Grady Booch, James Rumbaugh and Ivar Jacobson, Addison Wesley, 1999.] UML features and its applications in designing of embedded systems can be understood from the following brief description.

Figure 6.16(a) to (f) shows representations of six basic UML elements: class, package, stereotype, object, anonymous object and state. Table 6.2 gives a list of these and their description.

A conceptual design modeling can follow the UML approach. A conceptual design can use the user, object, sequence, state, class and activity diagrams. Table 6.3 gives UML 'class', 'state', 'sequence', 'collaboration' and 'object' diagrams.

UML allows the SpecCharts and StateCharts: SpecCharts (specification charts) is another language for specifications and charts. It allows state machines to use sequential programs to model the state actions. StateChart is a language for implementing the activity diagram, FSM states and state transitions, concurrency, synchronization, timing and behavioural hierarchy. The message sequence charts are first prepared and from these the StateChart, to show an activity diagram. For example, StateChart can model two concurrent activities of two FSMs. Its models along with its StateCharts-like features provide implementation of the exception-handling (trapping and interrupting) routines easily. UML sequence diagrams may also use the StateChart substrates, or models created by the StateChart language.

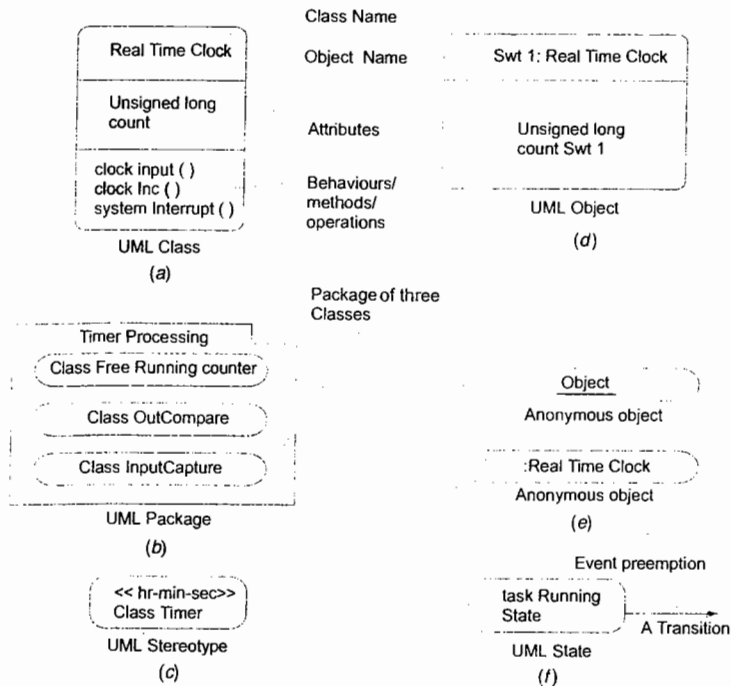


Fig. 6.16 Representation of unified modeling language basic elements (a) class (activeClass and abstract or inactive class) (b) package (c) stereotype (d) object (e) anonymous object and (f) state

UML is a powerful modeling language for: (i) software visualizing, (ii) data design, (iii) algorithms design, (iv) software design, (v) software specifications and (vi) software development process. UML basic elements are class, package, stereotype, object, anonymous objects and state. UML modeling is by class diagrams, state diagrams, object diagrams, sequence diagrams and collaboration diagrams.

Table 6.2 UML Basic Elements

Modeling Diagram	What does it model and show?	Exemplary Diagrammatic Representation
Class	Class defines the states, attributes and behaviour. A class can also be an active or abstract class.	Rectangular box with divisions as shown in Figure 6.16(a) for class names for its identity, attributes and behaviours (operations or methods or routines or functions).
Abstract class	A class in general may be abstract when either one or more states, operations or behaviour not completely defined, being in an abstract stage, or when it is not for creating objects but only a class, which extends, implements the abstract behaviours (methods) and specifies the abstract attributes (fields or properties) that class can create the object.	Rectangular box with divisions for class names for its identity, attributes and operations, but with prefix abstract with each abstract behaviour and attribute.
Object	An instance of a class that is a functional entity formed by copying the states, attributes and behaviour from a class.	Rectangular box with object identity followed by semicolon and class identity as shown in Figure 6.16(d).
Active object	An active class defines an active object instance of an active class. A process or thread is equivalent to the active object in UML, because active object posts the signals like thread and can wait before starting or resuming the operations using the methods.	Rectangular box with object identity followed by semicolon and class identity, but with prefix active with object identity.
Active class	An active class means a thread class that has a defined state, attributes, behaviours and behaviours for the signals. Active class in addition, defines the control by signal behaviours (for a signalling object, which can be posted and for which it may wait before starting or resuming). Thus there is control on the class behaviour.	Rectangular box with thick border lines and inner divisions for the class names for the identity, attributes and behaviours (operations and signals), but with prefix active with class identity.
Signal	An object, which is sent (posted) from one active class (active object) to another active class, which waits for start or resumption. Signal object behaviour defines the behaviour (operation method) of the interprocess communication. [Signal (Section 4.2.2) is software instruction or method (function), which generates interrupt.] Signal object has attributes (parameters). Attribute may be just a flag of 1-bit.	Signal identity within two pairs of starting and closing signs followed by class identity (Similar to stereotype).
Stereotype	An unpacked collection of elements (attributes or behaviours) that is repeatedly used.	Rectangular box with stereotype identity name given within the two pairs of starting and closing signs followed by the class identity as shown in Figure 6.16(c).

Modelling Diagram	What does it model and show?	Exemplary Diagrammatic Representation
Anonymous object	An object without identity.	Rectangular box with no object identity before the semicolon and class identity as shown in Figure 6.16(c).
Package	A packed collection of classes and objects.	A rectangular box with inner boxes for each class with name for class-identity. Package name is given over the top of box as shown in Figure 6.16(b).
State	A state.	Rounded rectangle with state name for its identity and with an arrow from the box. The arrow indicates a transition as shown in Figure 6.16(f).

Table 6.3 UML Diagrams

Modeling Diagram	What does it model and show?	Representation
State diagram	State diagram shows a model of a structure for its start, end, in-between associations through the transitions and shows event labels (or condition) with associated transitions.	A dark circular mark shows the starting point, arrows show the transitions. A label over the arrow shows the condition or event, which fires the transition. A dark rectangular mark within a circle shows the end [Figure 6.17(a)].
Class diagram	Class diagrams show how the classes and objects of a class relate hierarchical associations and object interactions between the classes and objects.	Rectangular boxes show the classes and arrows with unfilled triangles at the end show the class hierarchy. Classes in the hierarchy can be joined using a line. Start and end numbers on a line show the number of objects of a class associates with the number of objects of the other [Figure 6.17(b)].
Object diagram	Object diagram defines the static configuration of the system. It also gives the relationship among the consequent objects.	Refer Figure 6.17(c).
Sequence diagram	Sequence diagram visualizes the interactions between the objects. Sequence diagrams also specify the sequences of states.	Rounded rectangles for states and rectangular boxes with object identity and class connects by arrows. Vertical axis pointing downward shows progressively the time [Figure 6.18(a) and (b)].
Collaboration diagram	Collaboration diagram visualizes the concurrent sequences of states or object interactions.	Horizontal or vertical axis pointing right or downwards shows progressively the time and a parallel set of sequences show concurrency. Conditions or events can be labelled on the arrow [Figure 6.18(c)].

¹ Figure 6.18 shows the UML sequence diagrams. Figure 6.18(a) shows sequence of interaction between the states, (b) shows the sequence diagram (e.g., automatic chocolate-vending machine sequences of states).

² Figure 6.18(c) shows the collaboration diagram (concurrent multiprocessing).

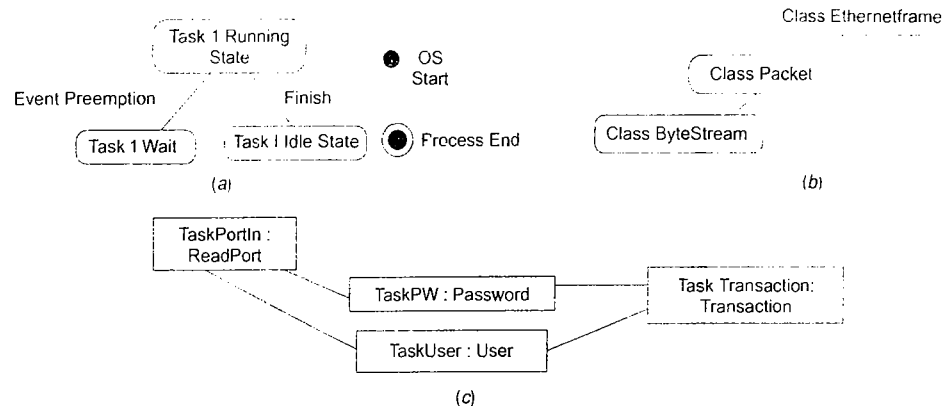


Fig. 6.17 Unified modeling language diagram: (a) state diagram (b) class diagram (c) object diagram

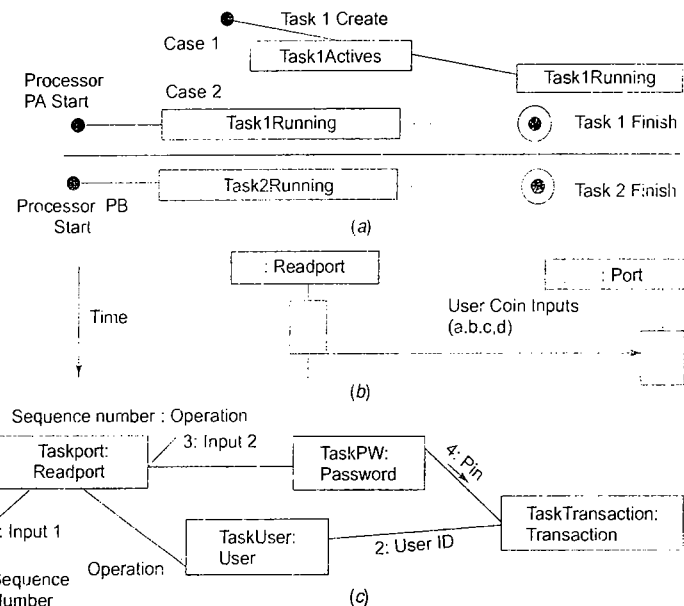


Fig. 6.18 Unified modeling language diagrams: (a) sequence of interaction between the states (b) sequence diagram (e.g., automatic chocolate-vending machine sequences of states) (c) collaboration diagram (e.g., multiprocessing concurrently processing system)



Summary

- Important models for programming are multiple function calls, polling for events, function queuing, sequential functions, data flow graph, CDFG state machine, concurrent processing and OOP.
- A standard design practice is using a model or set of models during the development process for software. Software implementation becomes greatly simplified by the use of DFGs because programming tasks are thereby reduced to the following. Coding for each process represented by a circle using the data input from incoming arrow(s) and generating data output to the outgoing arrow(s).
- In a DFG model, there is a single data-in point and a single data-out point with a process or set of processes that are represented by circle(s). When the assignment to an input is fixed in a DFG, it is also called ADFG. Programming complexity is minimized by modeling a program in terms of as many DFGs as possible and the use of as many ADFGs as possible.
- Another important concept of program modeling is the CDFG for program design and analysis. The CDFGs represent the controlled decision at the nodes and program paths (DFGs) that are traversed consequently from the nodes after the decisions.
- Program modeling can be done by the FSM and state machine models.
- The FSM model is appropriate for one process at a time, for the sequential flows from one state to the next state and for a controlled flow of the program. The FSM is found to easily model the states in processes in the systems.
- The multiprocessor system uses two or more processors for faster execution of the following: (i) program functions, (ii) tasks or (iii) single instruction multiple data instructions or (iv) multiple instructions multiple data instructions or (v) very long instruction words. The VLIWs in the DSP instructions can be completed at high speed. Modelling of multiprocessor system uses SDFG and HSDFG representations in which there is unfolding of the SDFG so that there is only one token which delays along its edge and/or an APEG in which there are no delays.
- Models are used for partitioning, load balancing, scheduling, synchronization and resynchronization during the program flow on the multiple processors. This gives minimum total performance costs (processing delays). UML is used for modeling the object-oriented programs. UML specifies the following basic elements: class, package, stereotype, object, anonymous objects and state. UML specifies the class diagrams, state diagrams, object diagrams, sequence diagrams and collaboration diagrams.

Keywords and their Definitions

ADFG	:	Acrylic DFG model when the assignment to the input is fixed.
APEG	:	HSDFG with no delays and vertices arranged in the precedence order.
Concurrent processing	:	When several processes execute a set of instructions such that each can proceed further by passing or exchanging messages or signals or tokens.
CDFG	:	Modeling by representing the controlled decision at the nodes and program paths (DFGs) that are traversed consequently from the nodes after the decisions.
DFG	:	The code for each process is represented by a circle and the data inputs to the process are by incoming arrow(s) and the generation of data output is through the outgoing arrow(s).
Finite state machine	:	A model in which there are finite states. After a given set of inputs, a state changes according to the state transition function.

HSDFG	:	Representation in which there is unfolding of the SDFG so that there is only one token, which may delay along its edge.
Interprocess communication	:	During concurrent processing, IPC is a message or signal or token to another waiting process or processor to proceed further.
Load balancing	:	Partitioning and scheduling of threads (set of instructions) and instructions such that each processor shares the processing load in a multiprocessor system.
Model	:	It is a representation by which problem, process, design or analysis can be easily understood and the problem becomes simplified after modelling.
Multiprocessor system	:	A system that uses two or more processors or that uses dual cores or multiple cores for faster execution of the (i) program functions, (ii) tasks or (iii) single instruction multiple data instructions or (iv) multiple instructions multiple data instructions or (v) very long instruction words.
Partitioning	:	Partitioning the graphs into parts, with each part scheduled on the processors as per the scheduling strategy adopted.
Performance cost	:	Time taken in waiting for execution at a vertex or at a sub-graph or micro thread.
Resynchronization	:	Repeating synchronization by suitable mathematical analysis, reducing the number of IPCs and thus the delays caused at the processors waiting for the IPCs in a multiprocessor system.
Scheduling	:	Allocation of different vertices or sub-graphs on different processes.
SDFG	:	A DFG representation in which input(s) delays are also shown. Circles (vertices) are the actors where computations take place. Nodes in a DFG edge (arc or arrow) has dots for the delays and labels the number of inputs and outputs.
State transition function	:	A process or state of codes that carry a program state from one to another.
UML	:	A language used for modelling the (i) software visualizing, (ii) data designs, (iii) algorithms design, (iv) software designs, (v) software specifications and (vi) software development process.
Total performance cost	:	Total of all performance costs. This will be the minimum if the load on the processors is balanced and there is appropriate partitioning and scheduling.



Review Questions

1. Why does program complexity increase with a reduced number of DFGs and increasing decision nodes?
2. Explain with one example each, APEG, SDFG and HSDFG.
3. Why do you unfold SDFGs into as many HSDFGs as feasible and then HSDFGs into as many APEGs as possible?
4. How does concurrent processing help in VLIW instruction execution at high speed?
5. How will you schedule an SIMD instruction on two processors?
6. How will you schedule a MIMD instruction on two processors?
7. How will you schedule an VLIW instruction on two processors?
8. What do you mean by completely dynamic scheduling and completely static scheduling in a multiprocessor system?
9. What do you mean by load balancing? How do you achieve it by combined partitioning?
10. How is an anonymous object denoted in UML?
11. What are the features of UML?



Practice Exercises

12. Tabulate various program models and give two application examples of each.
13. How will the DFG for FIR filter modify as CDFGs?
14. Draw a CDFG to incorporate decision nodes at the loop start and loop end to limit the summation terms up to $n = 10$ for Equation as follows: $y_n = \sum a_i x_{(n-i)} + \sum b_i y_{(n-i)}$ used in an IIR filter.
15. Draw an FSM model of an automatic chocolate-vending machine program. The machine permits only one type of coin, Rs 1, one chocolate at a time and one chocolate is cost is Rs 8.
16. Give the program model of master and slave robots in robot orchestra.
17. Give the program model of a digital camera.
18. Draw multiprocessor system for the cases: (i) tightly coupled to the memory; (ii) loosely coupled; (iii) coupled by mesh; (iv) ring-coupled; (v) torrid-coupled and (vi) tree-like coupling.
19. How do you solve the following problems: 'How can a program be partitioned into tasks or sets of instructions between the various processors? How can then the instructions and data be scheduled over the available processor times and resources so that there is an optimum performance'?
20. Assume that four processes are scheduled to run on two processors. A program is partitioned in such a way that with each 10,000 ns each process schedules 10 times on each processor. What will be the minimum number of contexts switching/microsecond?
21. How will you create and display SMS message T9 keypad of a mobile phone? Use the states, FSM model and state tables for all keys 0, 1 to 9 with T9 keypad. Use Examples 6.7 to 6.9 as templates.
22. Draw the 'class diagram', 'state diagram', 'sequence diagram', 'collaboration diagram' and 'object diagram' for AVCM in Section 1.10.2.

Interprocess Communication and Synchronization of Processes, Threads and Tasks

7

\mathcal{R}

e

c

a

l

l

The following important points have been discussed in earlier chapters.

1. Software embedded in a system can be highly complex as an application program has a number of functions, ISRs, threads, multiple physical and virtual device drivers, and several program objects that may be sequentially or concurrently processed on a single processor or multiple processors.
2. The system tasks may have vastly different functionalities, priorities, response-time constraints, latencies and deadlines.

7.1 MULTIPLE PROCESSES IN AN APPLICATION

7.1.1 Process

Application program can be said to consist of a number of processes [Figure 7.1(a)] and each process runs under the control of an OS (Section 1.4.6). Meaning and the basic concept of process can be understood as follows:

1. A process consists of sequentially executable program (codes) and *state-control* by an OS.
2. The *state* during running of a process is represented by the information of process state (created, running, blocked or finished), process structure—its data, objects and resources, and process control block (PCB) [PCB explanation follows later].
3. A process runs on scheduling by OS (kernel), which gives the control of CPU to the process. Process runs instructions and the continuous changes of its state takes place as the PC changes. [PC is program counter or instruction pointer to point to the current instruction of running program.]

Process is that unit of computation, which is controlled by some process at the OS for scheduling that lets it execute on the CPU and by some process at OS for resource management that permits use of system memory and other system resources such as network, file, display or printer.

Process is defined as a computational unit that processes on a CPU and whose state changes under the control of kernel of an OS. It has a state, which at an instance defines by the *process status* (running, blocked or finished), *process structure*—its data, objects and resources and *process control block*.

Example 7.1

Consider a mobile phone device (Section 1.5.5). The device-embedded software is highly complex. It has a number of functions, ISRs, threads, multiple physical and virtual device drivers and several program objects that must be concurrently processed on a single processor. The OS assumes the application-embedded software as consisting of a number of processes. Exemplary processes at the device are as follows: (i) *Voice encoding and convoluting process*: the device captures the spoken words through a speaker, and generates the digital signals after analog to digital conversions, and does the digits encoding and convoluting using a CODEC; (ii) *Modulating process*; (iii) *display process*; (iv) GUIs (graphic user interfaces) and (v) *Key-input process* for provisioning of user keypad interrupts.

Process Control Block PCB is a data structure having the information using which the OS controls the process state. The PCB stores in the protected memory addresses at kernel. The PCB consists of following information about the process state.

1. Process ID, process priority, parent process (if any), child process (if any) and address to the next process PCB, which will run next.
2. Allocated program memory address blocks in physical memory and in secondary (virtual) memory for the process codes.
3. Allocated process-specific data address blocks.
4. Allocated process heap (data generated during the program run) addresses.
5. Allocated process stack addresses for the functions called during running of the process.
6. Allocated addresses of the CPU register-save memory as a *process context* represents by CPU registers, which include the PC and SP [These register contents (process context) load into the CPU registers from the memory when the process starts running, and the addresses of CPU register-save memory saves the registers on context switch to another process].

We will discuss the following with examples.

1. Processes or threads or tasks are controlled by an OS, which enables their running concurrently in a system.
2. The tasks and their states.
3. The tasks and task-control-blocks, thread-stacks and process-control-blocks.
4. The context and context switching in multiprocessing, multitasking and multithreading system.
5. The distinction between functions, ISRs and tasks in order to understand the finer details of processing of each during a program run.

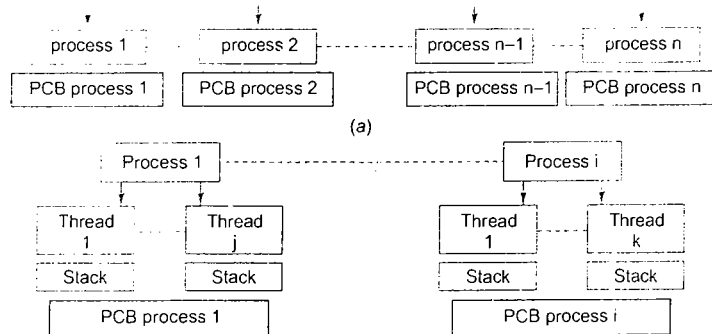
We will learn the uses of the following IPCs (inter-process communication functions) and devices.

1. Semaphore to communicate occurrence of an event at one process to another which waits for the event to proceed further.
2. Semaphore as mutex or counting semaphore and understanding of the P and V semaphores.
3. Problem and solution of the data that have to be shared between multiple tasks.
4. Mutex in solving the shared data problem and application in running the critical section codes.
5. Solution of the priority inversion problem and deadlock situation when using a semaphore.
6. Signal by a process to force a running process to interrupt and start a signal handler function (ISR) or process.
7. Queue in which messages are inserted by a process and communicated to other which are waiting for messages.
8. Mailbox to communicate a message from one process to another which waits for the message to proceed further.
9. Pipe device to communicate the bytes for messages from one process to another which takes the messages.
10. Socket as a bi-direction device to communicate the bytes as a stream or as per the protocol from one socket address in a process to another socket address in another process which can be local or remote.
11. Remote procedure call from a process to call a function or method in another process which is remote as per the protocol from one address in a process to another address in another process which can be local or remote.

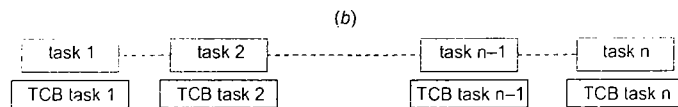
An OS provides mechanism of the IPCs to enable processes to synchronize and transfer the signals and messages. The OS also provides functions for the process, memory, IOs, device, time and event management. The OS also provides interrupt-handling mechanism. The OS also provides scheduling mechanism for the processes or tasks or threads. Chapter 8 will describe these mechanisms.

Chapters 9 and 10 will describe with exemplary RTOSes. The RTOSes also provide for handling the task-priorities and real-time constraints.

Units of computation, execution of codes in which is controlled by the OS scheduler, inter-process communication, resource-manager, system-memory and other system-resources (such as network, file, display or printer) access control mechanisms and are processed concurrently.



A thread is a process or sub-process within a process that has its own program counter, its own stack pointer, and stack, its own priority-parameter for its scheduling by a thread-scheduler, and its own variables that load into the processor registers on context switching and is processed concurrently along with other threads.



Tasks are embedded program computational units that run on a CPU under the state-control using a task control block. The tasks are processed concurrently

Fig. 7.1 (a) Processes (b) Threads (c) Tasks

7. Process-state signal mask [when mask is set to 0 (active) the process is inhibited from running and when reset to 1, the process is allowed to run].
8. Signals (and messages) dispatch table (for the process IPC functions).
9. OS-allocated resources' descriptors [e.g., file descriptors for open files, device descriptors for open (accessible) devices, device-buffer addresses and status, socket-descriptor for open socket].
10. Security restrictions and permissions.

The present CPU registers, which include PC and SP are called context and save on the PCB-pointed process stack and register-save memory addresses. Then the running process stops. Other process CPU registers now load and that process runs. This also means that the context has switched to another process.

7.2 MULTIPLE THREADS IN AN APPLICATION

Application program can be said to consist of a number of threads or a number of processes and threads [Figure 7.1(b)]. Meaning and basic concept of thread can be understood as follows:

1. A thread consists of sequentially executable program (codes) under state-control by an OS.
2. The state information of a thread is represented by *thread-state* (started, running, blocked or finished), *thread-structure*—its data, objects and a subset of the process resources and *thread-stack*.
3. A thread is a lightweight entity.

[Note: A process is considered as a heavyweight process and a kernel-level controlled entity. A process can have codes in the secondary memory from which the pages can be swapped into the physical primary memory during running of the process. The process may therefore have process structure with the virtual memory map, file descriptors, user-ID and so on. A thread can be considered a lightweight process and a process-level controlled entity. [Note: What the structure is, however, depends on the OS.]

A *thread* is a process or subprocess within a process that has its own PC, its own SP and stack, its own priority parameter for its scheduling by thread-scheduler, its variables that load into the processor registers on context switching. It has its own signal mask at the kernel. The signal mask when unmasked allows the thread to activate and run. When masked, the thread is put into a queue of pending threads. A thread stack is a memory address block allocated by the OS. When a function in a thread in OS is called, the calling function state is placed on the stack top. When there is return the calling function takes the state information from the stack top.

A multiprocessing OS runs more than one process. When a process consists of multiple threads, it is called multithreaded process. A thread can be considered as daughter process. A thread defines a minimum unit of a multithreaded process that an OS schedules onto the CPU and allocates the other system resources.

A process structure consists of data for memory mapping, file description and directory. Different threads of a process may share a common process structure. Multiple threads can share the data of the process.

Process can be allocated program memory address blocks in the physical memory as well as in the secondary (virtual) memory for the process codes. Memory mapping means mapping of the running program logic addresses with the physical addresses where the pages of the process codes load. A map called virtual memory map is used for memory mapping. A thread need not possess this data.

How does a task differ from a thread? Thread is a concept used in Java or Unix. A thread can either be a subprocess within a process or a process within an application program. To schedule the multiple processes, there is the concept of forming thread groups and thread libraries. A task is a process and the OS does the multitasking. Task is a kernel-controlled entity while thread is a process-controlled entity. A task is analogous to a thread in most respects. A thread does not call another thread to run. A task also does not directly call another task to run. Both need an appropriate scheduler. Multithreading needs a thread-scheduler. Multitasking needs a task-scheduler. There may or may not be task groups and task libraries in a given OS.

Example 7.2

Consider a mobile phone device (Section 1.5.5). *Display_process* can have multiple threads. A thread *Display_Time_Date* can be for displaying clock time and date. A thread *Display_Battery* can be for displaying battery power. A thread *Display_Signal* can be for displaying signal power for communication with the mobile service provider. A thread *Display_Profile* can be for displaying silent or sound-active mode. A thread *Display_Message* can be for displaying unread message in the inbox. A thread *Display_Call Status* can be for displaying call status; whether dialing or call waiting. *Display_Menu* can be for displaying menu. These threads can share the common memory blocks and resources allocated to the *Display_Process*. A display thread is now the minimum computational unit controlled by the RTOS. Each thread has independent parameters—ID, priority, PC, SP, CPU registers and its present status.

A *thread* is a process or subprocess within a process that has its own PC, its own SP and stack, its own priority parameter for its scheduling by thread-scheduler. Thread is a concept in Java and Unix and it is a lightweight subprocess or process in an application program. The thread can share a process structure. It has a thread stack at the memory. It has a unique ID. It has states in the system as follows: starting, running, blocked and finished.

7.3 TASKS

Task is the term used for the process in the RTOSes for the embedded systems. (For example, VxWorks and μ COS-II are the RTOSes, which use the term task.) A task is similar to a process or thread in an OS. Some OSes use the term task and some use the term process. Figure 7.1(c) shows the application software consisting of a number of tasks.

1. A task consists of a sequentially executable program (codes) under a state-control by an OS.
2. The state information of a task is represented by the task state (running, blocked or finished), task structure—its data, objects and resources and task control block (TCB).

An application program can also be defined as a program consisting of the tasks and task behaviours in the various states. The task states are controlled by some process at the OS for scheduling that allows it to execute on the CPU and by some process at OS for resource-management that allows it to use the system memory and other system resources such as network, file, display or printer.

Embedded software for an application may consist of a number of tasks and each task run needs a control of the state by OS. Assume that there is only one CPU in a system. Each task is independent in that it takes control of the CPU when scheduled by a scheduler at the OS. The scheduler controls and runs the tasks. A task is an independent process. No task can call another task. [It is unlike a C (or C++) function, which can call another function.] The task can send signal(s) or message(s) that can let another task run waiting for that signal or message. The OS can block a running task and let another task gain access of the CPU to run the servicing codes.

Task is defined as embedded program computational unit that runs on a CPU under the state-control of kernel of an OS. It has a state, which at an instance defines by *status* (running, blocked, or finished), *structure*—its data, objects and resources and control block.

Example 7.3

Consider an ACVM (Section 1.10.2). The ACVM-embedded software is highly complex and the OS schedules to run the application-embedded software as consisting of a number of tasks. Exemplary tasks at the ACVM are as follows: (i) *Task User Keypad Input*: the keypad gets the user input. (ii) *Task Read-Amount*: for reading the inserted coins amount. (iii) *Chocolate delivery task*; delivers the chocolate and signals the machine to get ready for the next input of the coins. (iv) *Display Task*. (v) *GUL_Task* (for graphic user interfaces). (vi) *Communication task* for provisioning the AVCM owner access to the machine status and information.

7.4 TASK STATES

Figure 7.2(a) shows a task and its states. Task has state, which includes its status at a given instance in the system. It can be one of the following state: idle (created), ready, running, blocked and deleted (finished). It is

in the ready state again after finish when it has infinite waiting loop—an important feature in embedded system design. Multitasking operations are by context switching between the various tasks.

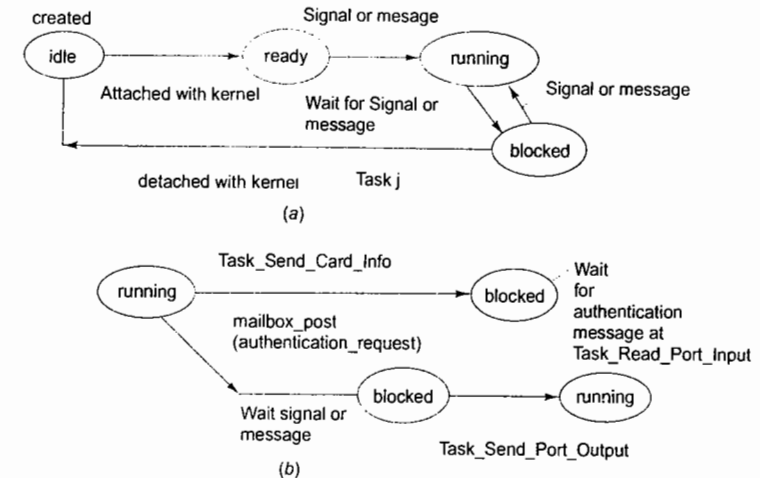


Fig. 7.2 (a) Task and its states (b) States of the task Task_Send_Card_Info in Example 7.4

A task can be considered to be in one of the five states. What the states can be, however, depends on the ROS. Five states are as follows.

1. *Idle (created) state*: The task has been created and memory allotted to its structure. However, it is not ready and is not schedulable by the kernel.
2. *Ready (active) state*: The created task is ready and is schedulable by the kernel but not running at present as another higher priority task is scheduled to run and has the system resources at this instance.
3. *Running state*: Executing the servicing codes and getting the system resources at this instance. It will run till it needs some IPC (input) or starts wait for an event or till it pre-empts by another higher priority task than this task.
4. *Blocked (waiting) state*: Execution of the servicing codes suspends after saving the needed parameters into its context. It needs some IPC (input) or waiting for an event or waiting for higher priority task to block. For example, a task is pending while it waits for an input from the keyboard or file. The scheduler then puts it in the blocked state.
5. *Deleted (finished) state*: The created task has memory de-allotted to its structure. It frees the memory. Task has to be re-created.

A created and activated task will be in one of the three states, ready, running and blocked.

Example 7.4

Consider a smart card (Section 1.10.3). When it is inserted into a card reader host machine, it gets the radiation and charges up.

Step 1: Let the main program first run an OS function *OS_initiate* (). This enables use of the RTOS functions.

- Step 2:** The main program runs an OS function `OS_Task_Create ()` to create a task, `Task_Send_Card_Info`. The task is for sending card information to the host. The task is allocated memory for the stack. The Task has a TCB using which the OS controls the task. The task state is idle state. Let this task be of high priority.
- Step 3:** `OS_Task_Create ()` runs two more times to create two other tasks, `Task_Send_Port_Output` and `Task_Read_Port_Input` and both of them are also in idle state. Let these tasks be of middle and low priorities, respectively.
- Step 4:** The functions for starting `OS_Start ()` and for initiating n system clock interrupts `OS_Ticks_Per_Sec ()` run. The system switches from the user mode to the supervisory mode every $1/60$ seconds if $n = 60$. All three task states will be made in `ready_state` by an OS function.
- Step 5:** The OS runs a function, which makes the `Task_Send_Card_Info` state as running. The `Task_Send_Card_Info` runs an OS function `mailbox_post (authentication_request)`, which sends the server identification request through IO port to the host using the task `Task_Send_Port_Output`.
- Step 6:** The `Task_Send_Card_Info` runs a function `mailbox_wait ()`, which makes the task state as blocked and the OS switches context to another task `Task_Send_Port_Output` and then to `Task_Read_Port_Input` for reading IO port input data.
- Step 7:** When the mailbox gets the authentication message from the host server, the OS switches context to `Task_Send_Card_Info` and the task comes to the running state again.

Figure 7.2(b) shows the task `Task_Send_Card_Info` states in different steps.

7.5 TASK AND DATA

Figure 7.3 shows a task and its data including its context and TCB. A task has the following data specific to a task, which saves at the TCB.

1. Each task has an ID just as each function has a name. The ID is of one byte and is called the index of the task if a typical OS assigns each ID a number between 0 and 255.
2. Each task may have a *priority parameter*. The priority, if between 0 and 255, is represented by a byte (usually, the higher the value, the lower the priority of that task).
3. Each task has its independent (distinct from other tasks) values of the following at an instant: (i) PC (memory address from where it runs if granted access to the CPU) and (ii) SP (memory address from where it gets the saved CPU registers and parameters, which includes registers for the task PC and pointer to task stack-top after the scheduler grants access to the CPU). These two values are the part of its context of a task.

Context Each task has a context (CPU registers and parameters, which includes registers for the task PC and pointer to the called function stack-top). This reflects the CPU state just before the OS blocks one task and initiates another task into the running state. The context thus continuously updates during the running of a task, and the context is saved before switching occurs to another task.

Context Switch Only after saving these registers and pointers does the CPU control switch to any other process or task. The context must retrieve on transfer of program control to the CPU back for running the same task again, on the OS unblocking its state and allowing it to enter the running state again. The context-switching action must happen each time the scheduler blocks one task and runs another task.

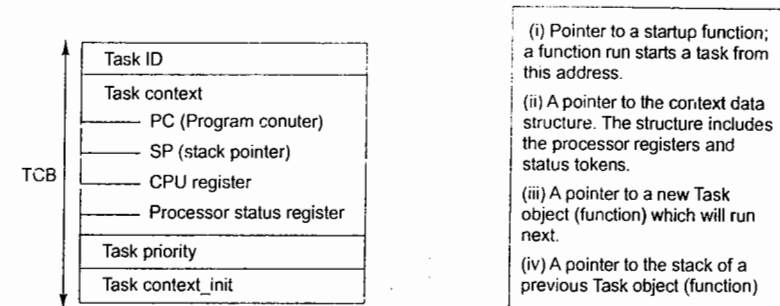


Fig. 7.3 A task and its data including its context and task control block

Each task also has an initial context, `context_init`. The `context_init` has the initial parameters of a task. The parameters of `context_init` are as follows: (i) Pointer to a start-up function: a function run starts a task from this address. (ii) Pointer to the context data structure: the structure includes the processor registers and status tokens. (iii) The task context may also include a pointer to a new task object (function) which will run next. (iv) It may also include a pointer to the stack of a previous task object (function).

Example 7.5

Consider an ACVM (Section 1.5.2). After the *Task Read-Amount* (for reading the inserted coins amount) gets the required cost of the chocolate, it send an IPC (a signal or message) to let the OS context switch and start the *Chocolate delivery task*.

Chocolate delivery task delivers the chocolate, sends an IPC for *Display task* to display 'thank you and visit again' and sends another IPC to the machine ready for the next input of the coins.

There are context switches from *Task Read-Amount* to *Chocolate delivery task*, from *Chocolate delivery task* to *Display task* and from *Display task* to *Task User Keypad Input*.

7.5.1 Task Control Block

Each task has a TCB. TCB is a memory block. Figure 7.3 showed the TCB data for a task. The TCB is a data structure having the information using which the OS controls the task state. The TCB stores in the protected memory area of the kernel. The TCB consists of the following information about the task. It stores the current instant PC information (to indicate the address of the next instruction to be executed for this task), memory map, signal (message) dispatch table, signal mask, task ID, CPU state (registers, task PC and task SP) and kernel stack (for executing system calls and so on). [Note: (i) The TCB is similar to the process control block (PCB) and (ii) TCB data structure can vary from one OS to another.]

7.6 CLEAR-CUT DISTINCTION BETWEEN FUNCTIONS, ISRs AND TASKS BY THEIR CHARACTERISTICS

7.6.1 Task Coding in Endless Event-Waiting Loop

Each task may be coded such that it is in endless event-waiting loop to start with. An event loop is one that keeps on waiting for an event to occur. On the start event, the loop starts from the first instruction of the loop.

Execution of service codes (or setting a token that is an event for another task) then occurs. At the end, the task returns to the start event waiting loop.

Example 7.6

Consider an ACVM *Chocolate delivery task*. It can be coded as follows.

```
/* The codes for the Chocolate_delivery_task */
static void Task_Deliver (void *taskPointer) {
    /* The initial assignments of the variables and pre-infinite loop statements that execute once only*/
    while (1) { /* Start an infinite while-loop. */
        /* Wait for an event indicated by an IPC from Task Read-Amount */

        /* Codes for delivering a chocolate into a bowl. */

        /* Send message through an IPC for displaying "Collect the nice chocolate. Thank you, visit again" to
        the Display Task*/
        /* Resume delayed Task Read-Amount */
        }; /* End of while loop*/
    } /* End of the Task_Deliver function */
```

7.5.2 Distinction between Function, ISR and Task

When there are multiple devices, functions, ISRs and program objects, the embedded software can be modelled as consisting of multiple tasks and each task is scheduled by the kernel schedule and uses IPCs for synchronization. Threads are used in embedded Linux- or Unix-based applications. Threads are used in Java. Functions are subunits of the processes or tasks or ISRs or another function. Functions and ISRs do not have analogue of PCB or TCB. They have only a stack. Function has no associated scheduler-like tasks scheduler or thread scheduler at the kernel. ISR has associated interrupt handler at the kernel. Table 7.1 summarizes the characteristics of functions, ISRs and tasks.

1. **Function** is used in any routine for performing a specific set of actions as per the arguments passed to it and which runs when called by a process or task or thread or from another function. Functions run by nesting. Function runs after the previous context saving and after retrieving the context from a common stack.
2. An **ISR** is a function, which executes on interrupts. An ISR executes on an event and pending ISRs run as per priority-based scheduling. ISR can post the events or signals or messages. ISRs run as per the hardware-based interrupt-handling mechanism. ISRs may or may not run by nesting. ISR runs after the context saving and after retrieving the context from a common stack in case of nesting.
3. A **task** is a function, which executes on scheduling. A task can wait as well as post the events or signals or messages. The tasks run after saving of the previous context at the SP pointed address in task TCB and the context switching to new context at the new task SP pointed address in TCB. The tasks run as per the task scheduling and IPC management mechanism of the OS.

Recall that Section 5.4.6 explained the re-entrant function. Each task must be either reentrant function or must have a way to solve the shared data problem. Section 7.7 will explain the shared data problem and use of semaphores.

Table 7.1 Characteristics of the Functions, Interrupt Service Routines (ISRs) and Tasks

Function	ISR	Task
<p>1. Uses: Function is used in any routine or process or task for running specific set of codes for performing a specific set of actions as per the arguments passed to it.</p>	<p>ISR is used for running a specific set of codes for performing a specific set of actions. ISR has code, which runs once and for servicing the interrupt-call only.</p>	<p>Task is used for running specific set of codes for performing a specific set of actions. Task has codes in an endless waiting loop.</p>
<p>2. Calling source: A call to run a function is from another function or process or thread or task.</p>	<p>An interrupt call for running an ISR can be from hardware or software at any instance. All interrupt source calls for running the ISRs are independent.</p>	<p>A call to run the task is from the system (OS). A OS preemptive scheduler can allow another higher priority task to execute after blocking the present one. It is the RTOS (kernel) only that controls the task scheduling.</p>
<p>3. Context save: Each function code run by changes in program counter instantaneous value. There is a stack. On the top of which the program counter value (for the code left without running) and other values (called functions' context) must save when calling another function or on interrupt and start of ISR. When there is return from a function to the function, which called it, the program counter restores from stack top to that value where the code left earlier [Figure 7.4(a)]. The stack of the functions in a process, thread or task is at the common memory block when the different functions execute.</p>	<p>Each ISR is an event-driven function code. The code run by changes in program counter instantaneous value. ISR has a stack for the program counter instantaneous value and other values that must save before allowing another ISR to execute. The stack need not be at a distinct memory block when different ISRs execute and the ISR stack is at a common memory block when there is nesting. (This is similar to the stack that associates with the functions.) Processor hardware may or may not provision for allowing the ISRs to execute in the nested mode.</p>	<p>Each task code run by change in program counter instantaneous value. Each task has a distinct task stack at the distinct memory block for the context (program counter instantaneous value and other CPU register values in task control block) that must save when blocking from its running state due to an interrupt or pre-emption by another higher priority task. Each task has a distinct process structure (TCB) at the distinct memory block.</p>
<p>4. Response and synchronization: A function calls another function and there is nesting of one another [Figure 7.4(b)]. There is a hardware mechanism for sequential nested mode synchronization between the functions directly without the control of scheduler or OS.</p>	<p>There is a hardware mechanism for responding to an interrupt for the interrupt source calls and there is, according to the given OS kernel feature, a synchronizing mechanism for the ISRs. (Refer to the next chapter; Figures 8.1(a) to (c) and 8.4.</p>	<p>According to the given OS kernel feature, there is a task-responding and synchronizing mechanism. The kernel functions are used for task synchronization because only the kernel calls a task to run at a time. When a task runs and when it blocks it is fully under the control of the OS.</p>

(Contd)

Function	ISR	Task
<p>5. Structure: A function or a number of functions can be the subunits of a process or thread or task or ISR or subunit of another function or main function.</p>	<p>An ISR is independent and can be considered as a function, which runs on an event due to the interrupting source. A pending interrupt is scheduled to run using an interrupt-handling mechanism in the OS. The system, during running of an ISR, can let another higher priority ISR run.²</p>	<p>A task is independent and can be considered as a function, which is called to run by the OS scheduler using a context switching and task-scheduling mechanism of the OS. The system, during running of a task, can let another higher priority task run. The kernel manages the task scheduling.</p>
<p>6. Global variables use: Function can change the global variables. The interrupts must be disabled and after completing the use of the global variable the interrupts are enabled (Section 7.8).</p>	<p>When using a global variable, the interrupts must be disabled and after completing the use of the global variable the interrupts are enabled.</p>	<p>When using a global variable, either the interrupts are disabled and after completing the use of global variable the interrupts are enabled or the semaphores are used as mutex critical sections (Sections 7.7.2, 7.7.5 and 7.8.3).</p>
<p>7. Posting and sending parameters: Function can get the parameters and messages through the arguments passed to it or global variables, reference to which is made. Function returns the results of the operations through the references in the arguments and through return as per reference data type defined for it.</p>	<p>ISR using IPC functions for post can send (post) the signals (Section 7.7.1) and messages (Sections 7.10 to 7.16), for example, OSSemPost (). ISR cannot use the mutex protection of the critical sections. ISR does not for signal or message during running.</p>	<p>Task can send (post) the signals and messages and the task can wait for the signals and messages using the IPC functions, for example, OSSemPost () and OSSemPend () (Section 7.7). Task can use the mutex protection of the code sections. (Sections 7.7.2, 7.7.4 and 7.8.3).</p>

Depending on the scheduling of the real-time operating system (RTOS) by the kernel two common methods are: (i) cooperative scheduling, (ii) pre-emptive scheduling (Refer to Sections 8.10.1 to 3). Either the interrupt handler or RTOS interrupt handler functions control the ISR scheduling. It depends on how the kernel or interrupt-handling hardware manages the ISRs (Section 8.7).

Figure 7.4(a) shows a characteristic feature of the nested function calls in a program. Figure 7.4(b) shows the PC assignments at different times on the nested calls.

7.7 CONCEPT OF SEMAPHORES

7.7.1 Use of a Semaphore as an Event-Signalling or Notifying Variable

Suppose that there are two trains. Assume that they use an identical track. When the first train A is to start on the track, a signal or token for A is set (true/taken) and the signal or token for the other train, B is reset (false/released).

OS provides for the use of a semaphore for signalling or notifying of a certain action and for notifying the acceptance of the notice or signal. Let a binary Boolean variable, *s*, represent the semaphore. The operations on the variable *s* signals or notifies the operations for communicating the occurrence of the event and for communicating taking note of the event. It is like a token. Release of a token is the occurrence of an event and the acceptance of the token is taking note of that event.

Let us assume that the *s* increments from 0 to 1 for signalling or notifying occurrence of an event from a

section of codes in a task or thread. When the event is taken note by a section in another task waiting for that event, the *s* decrements from 1 to 0 and the waiting task codes start at another action.

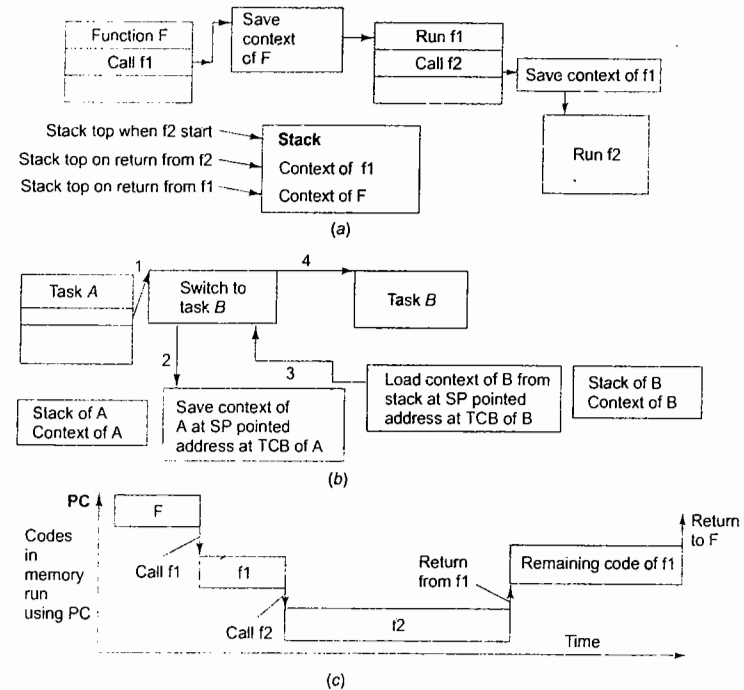


Fig. 7.4 (a) Actions on the function calls in a program (b) Action on pre-emption of task A by B or switch to task B during program run (c) Program counter assignments to the various functions in a process on the nested calls to the functions

A semaphore is called *binary* semaphore when its value is 0, it is assumed that it has been taken (or accepted), and when its value is 1, it is assumed that it has been released (or sent or posted) and no task has taken it yet.

An ISR can release the token. A task can release the token as well accept the token or wait for taking the token (row 7 Table 7.1).

Following is an example how to use a binary semaphore for signalling or notifying occurrences of an event from a task or thread and for signalling or notifying another task waiting for that event.

Example 7.7

Consider an ACVM (Section 1.10.2) *Chocolate delivery task* (Examples 7.5 and 7.6). After the task delivers the chocolate, it has to notify to the display task to run a waiting section of the code to display 'Collect the nice chocolate. Thank you, visit again'. The waiting section for the display of the thank you message takes this notice and then it starts the display of thank you message.

Assume `OSSemPost()` is an OS function for IPC by posting a semaphore and assume `OSSemPend()` is another IPC function for waiting the semaphore. Let `sdispT` is the binary semaphore posted from *Chocolate delivery task* and taken by a *Display task* section for displaying the thank you message. Let `sdispT` initial value = 0. The following will be the codes.

```
static void Task_Deliver (void *taskPointer) {
while (1) {
/* Codes for delivering a chocolate into a bowl. */
OSSemPost (sdispT) /* Post the semaphore sdispT. This means that OS function increments sdispT in
corresponding event control block. sdispT becomes 1 now. */
};
static void Task_Display (void *taskPointer) {
while (1) {
OSSemPend (sdispT) /* Wait for the semaphore sdispT. This means that task waits till sdispT is posted
and becomes 1. When sdispT becomes 1, the wait is over. an OS function runs to decrement sdispT
in corresponding event control block. sdispT becomes 0 now, and Task then runs further the
following code*/
/* Code for display "Collect the nice chocolate. Thank you, visit again" */
};
```

1. Semaphore provides a mechanism to allow section of the task code wait till another notifies an action (finish running of a section of the codes at a task or ISR). It provides a way of signalling an event occurrence. It provides a way of signalling taking of a note of the event. Semaphore can be used as a signalling or notifying variable (token).
2. Semaphore increments when posted (sent or released) by a task or ISR instruction and decrements when accepted or taken by the waiting task section.
3. A waiting task section is notified to start on sending the semaphore. A waiting task section starts on taking the semaphore.

7.7.2 Use of a Semaphore as Resource Key and for Critical Section

OS provides for the use of a single semaphore as a resource key and for running of the codes in critical section. A task A, when getting access to a resource (e.g., printer file or network or section of codes called critical section or printer) notifies to the OS to have taken the semaphore (take notice). [An OS function, e.g., `OSSemPend()` runs to notify. The OS returns the semaphore as taken (accepted) by decrementing the semaphore from 1 to 0.] Now, the task A accesses the resource (e.g., accesses the file, or network or runs the section of codes).

The task A, after completing access to a resource (e.g., memory buffer or file or network, or critical section) it notifies to the OS to have posted that semaphore (post notice). [An OS function, e.g., `OSSemPost()` runs to notify. The OS returns the semaphore as released by incrementing the semaphore from 0 to 1.]

The task B can access the same resource using `OSSemPend()` if it is waiting for that semaphore. The task B posts the semaphore using `OSSemPost()` after completing the access to that resource.

Figure 7.5(a) shows the use of a semaphore between A and B. It shows the five sequential actions at five different times, T0, T1, T2, T3 and T4. Figure 7.5(b) shows the timing diagram of the tasks in the running states as a function of time. It marks the five sequential actions at five different times, T0, T1, T2, T3 and T4.

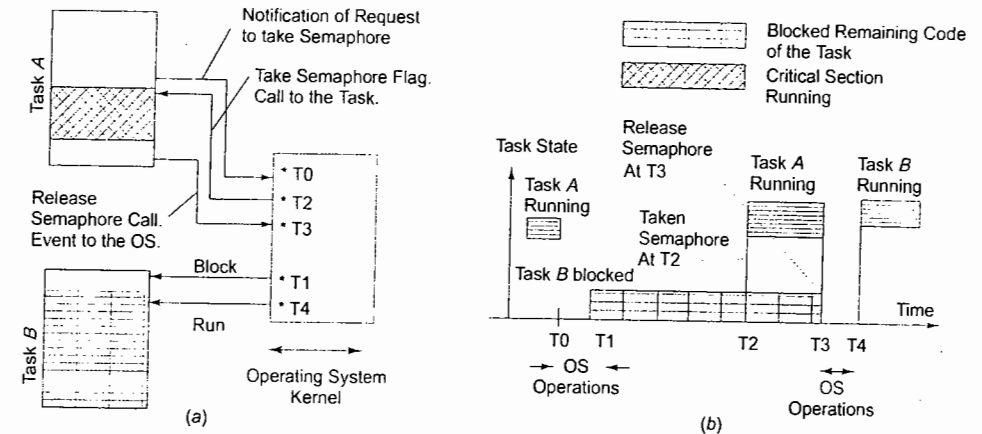


Fig. 7.5 (a) Use of a semaphore between tasks, A and B. It shows the five sequential actions at five different times, T0, T1, T2, T3 and T4 (b) Timing diagram of the tasks in the running states as a function of time. It marks the five sequential actions at five different times, T0, T1, T2, T3 and T4, and shows the use of a semaphore between tasks A and B by the operating system functions

An ISR can't be used to wait for the resource key since an ISR can just release the key. A task on the other hand can release the key as well accept the key or wait for taking the key. (row 7 Table 7.1)

Example 7.8

Consider an (Section 1.5.5) *Update_Time* task. When the task for updating time t on a system clock tick interrupt I_S starts, it has to notify that it is writing the t in a *time device* and that the t is changing. After the *Update_Time* task updates t information at the time device on I_S , it has to notify to the *Read_Time* task to run a waiting section of the code to read t from the *time device*. After the *Read_Time* reads t , it has to notify to *Update_Time* task to make note of it.

Assume `OSSemPost()` is an IPC function at OS for posting a semaphore and assume `OSSemPend()` is another IPC function for waiting the semaphore. Let `supdateT` be the binary semaphore pending and posted at *Update_Time* and pending and posted at *Read_Time* section for reading t . Let `supdateT` initial value = 1. The following will be the codes.

```
static void Task_Update_Time (void *taskPointer) {
```

```
while (1) {
```

```
    OSSemPend (supdateT) /* Wait the semaphore supdateT. This means that when wait over, an OS function
        decrements supdateT in corresponding event control block and supdateT becomes 0 at T2. */
    /* Codes for writing date and time into the time device. */
```

```
    OSSemPost (supdateT) /* Post the semaphore supdateT. This means that OS function increments supdateT
        in corresponding event control block and supdateT becomes 1 at T3. */
```

```
};
```

```
static void Task_Read_Time (void *taskPointer) {
```

```
while (1) {
```

```
    OSSemPend (supdateT) /* Wait for the semaphore supdateT. This means that task waits till supdateT is posted
        and becomes 1. When supdateT becomes 1 and the OS function then decrements supdateT in corresponding
        event control block and supdateT becomes 0 at T4. Task then runs further the following code*/
    /* Code for reading the date and time at time device */
```

```
    OSSemPost (supdateT) /* Post the semaphore supdateT. This means that OS function
        increments supdateT in corresponding event control block. supdateT becomes 1. */
```

```
};
```

Mutex When a binary semaphore is used to at beginning and end of critical sections in two or more tasks such that at any instance only one section code can run, then the semaphore is called *mutex*. (*mutex* word is derived from mutually exclusive). Example 7.8 showed that `Task_Update_Time` and `Task_Read_Time` uses the semaphore `supdateT` as mutex and at any instance either code section in `Task_Update_Time` or code section in `Task_Read_Time` runs and has exclusive access to the time device date and time variables. Mutex helps in getting access to a file or network or printer by multiple tasks at distinct instances because at an instance only codes for one of the tasks will be run to send the data to the file or network or printer. Section 7.8.3 will discuss help in sharing data between the tasks.

In certain OS, a semaphore as a resource key is called mutex when the semaphore takes care of priority inversion problem also. Section 7.8.5 explains the inversion problem. In certain OS, a semaphore as a resource key is called mutex even when the semaphore does not take care of the priority inversion problem. In certain OS, a semaphore as a resource key is called mutex and the option is provided to programmer for using priority inversion safe or without inversion safe mutex.

1. Semaphore provides a mechanism to let a section of the task code or a task wait till another task section finishes another set of codes such that these sections use a common resource, device or file or variable.
2. When a semaphore is waiting (for taking or accepting) by a task code, then that task has the access to the necessary resources when semaphore is 'given' (sent or posted), the resources unlock.
3. Semaphore can be used as a resource key. Resource key is one that permits the use of resources like CPU, memory or other functions or critical section codes.
4. Binary semaphore can be used as a mutex as well as event notifying flag.

7.7.3 Use of Multiple Semaphores for Synchronizing the Tasks

OS semaphore functions are provided for multitasking operations. Figure 7.6 shows an example of the use of two semaphores for synchronizing the tasks *I*, *J* and *M* and the tasks *J* and *L*. Example 7.9 gives another example in which *I*, *J*, *K* and *L* are synchronized to run sequentially.

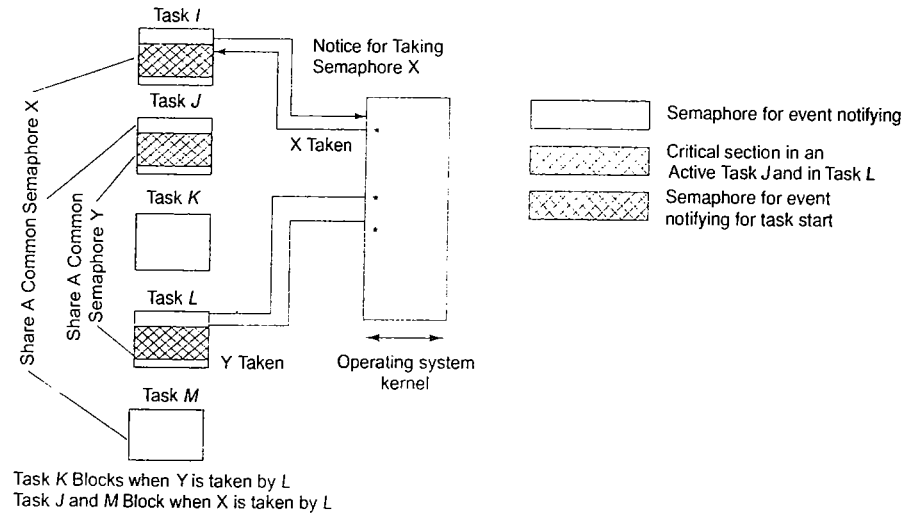


Fig. 7.6 An example of the use of two semaphores —one for synchronizing the tasks *I*, *J* and *M* and other for tasks *J* and *L*

Example 7.9

A semaphore can let one task run among many in the initiated into a list with *state* = 'ready' or 'running' run at an instance while others are waiting. Let the following be the codes.

Assume `OSSemPost ()` is an OS IPC function for posting a semaphore and assume `OSSemPend ()` is another OS IPC function for waiting for the semaphore. Let *sTask* be the binary semaphore pending and posted at each task to let another run. Let *sTask1* initially be 1 and *sTask2*, *sTask3* and *sTask4* initially be 0.

The following will be the codes and the first task *I* will run, then *J*, then *K*, then *L*, then *I* when at that instance *sTask1* = 1 and *sTask2*, *sTask3*, *sTask4* = 0.

```
static void Task_I (void *taskPointer) {
```

```
while (1) {
```

```
    OSSemPend (sTask1) /* wait for semaphore sTask1 and when wait over then an OS function decrements
        sTask1 in corresponding event control block and sTask1 becomes 0 */
```

```
    /* Codes for Task_I */
```

```

OSSemPost (sTask2) /* Post the semaphore sTask2. This means that OS function increments
sTask2 in corresponding event control block. sTask2 becomes 1 */
};
static void Task_ J (void *taskPointer) {
.
while (1) {
OSSemPend (sTask2) /* Wait for the semaphore sTask2. This means that task waits till sTask2is posted
and becomes 1. When sTask2 becomes 1 and the OS function is to decrements sTask2 in corresponding
event control block. sTask2 becomes 0. Task then runs further the following code*/
/* Code for Task J */
.
OSSemPost (sTask3) /* Post the semaphore sTask3. This means that OS function increments sTask3 in
corresponding event control block. sTask3 becomes 1. */
};
static void Task_ K (void *taskPointer) {
while (1) {
OSSemPend (sTask3) /* Wait for the semaphore sTask3. This means that task waits till sTask3 is posted
and becomes 1. When sTask3 becomes 1 and the OS function is to decrements sTask3 in corresponding
event control block. sTask3 becomes 0. Task then runs further the following code*/
/* Code for Task K */
.
OSSemPost (sTask4) /* Post the semaphore sTask4. This means that OS function increments sTask4 in
corresponding event control block. sTask4 becomes 1. */
};
static void Task_ L (void *taskPointer) {
while (1) {
OSSemPend (sTask4) /* Wait for the semaphore sTask4. This means that task waits till sTask4 is posted
and becomes 1. When sTask4 becomes 1 and the OS function is to decrements sTask3 in
corresponding event control block. sTask4 becomes 0. Task then runs further the following code*/
/* Code for Task J */
.
OSSemPost (sTask1) /* Post the semaphore sTask1. This means that OS function increments
sTask1 in corresponding event control block. sTask1 becomes 1. */
};

```

For example, when a task *K* is to start running, it takes the semaphore *sTask3*. The OS blocks the tasks *I*, *J* and *L*. A task *L* waits for the release of the semaphore by *K*.

Number of tasks Waiting for the Same Semaphore An RTOS has the answer to the following: when a number of tasks has the same semaphore waiting then which of them takes the semaphore? In certain OS, a semaphore is given to the task of highest priority among the waiting tasks. In certain OS, a semaphore

is given to the longest waiting task in the FIFO mode. In certain OS, a semaphore is given to select an option and the option is provided for priority or FIFO mode. The task having priority if started, takes a semaphore first in case the priority option is selected. The task pending since a longer period takes a semaphore first in case the FIFO option is selected.

1. Multiple semaphores are used and different set of semaphores can share among different set of tasks.
2. Semaphore provides a mechanism to synchronize the task codes. Multiple semaphores can be used in multitasking system.

7.7.4 Counting Semaphores

An OS may provide for the counting semaphores. A counting semaphore can be an unsigned 8- or 16- or 32-bit integer. A value of counting semaphore controls the blocking or running of the codes of a task. The counting semaphore decrements each time it is taken. It increments when released by a task.

The value of counting semaphore at an instance reflects the initialized value minus the number of times it is taken plus the number of times released. The counting semaphore can be considered as the number of tokens present and the waiting task will not wait and run further if at least one token is present. The use of a semaphore is such that one of the task thus waits to execute the codes or waits for a resource till at least one token is found.

Assume that a task can send the stacks on a network into eight sequentially transmitting buffers. Each time the task runs it takes the semaphore and sends the stack into a buffer, which is next to the earlier one. Assume that a counting semaphore *scnt* is initialized = 8. After sending the data of the stack, the task takes the *scnt* and *scnt* decrements. When a task tries to take the *scnt* when it is 0, then the task blocks and cannot send stack into any buffer.

Example 7.10

Consider an ACVM (Section 1.10.2). Consider *Chocolate delivery task*. It cannot deliver more than the total number of chocolates, *total* loaded into the machine. Assume that a *semCnt* is initialized equal to the *total*. Each time, the new chocolates are loaded in the machine, *semCnt* increments by the number of new chocolates added. The *Chocolate delivery task* can be coded as follows.

```

static void Task_Deliver (void *taskPointer) {
.
while (1) { /* Start an infinite while-loop. */
/* Wait for an event indicated by an IPC from Task Read-Amount */
.
OSSemPend (semCnt) /* If chocolate available is true then the task takes the semaphore if
semCnt is 1 or > 1 (which means is not 0) and decrement the semCnt and continue remaining
operations */
.
};

```

Counting semaphore can be consider as an unsigned integer semaphore that can be 'taken' till its value = 0 and is initialized to a high value. It can also be 'given' a number of times.

7.7.5 P and V SEMAPHORES

An OS may provide for an efficient synchronization mechanism, called P and V semaphores in a standard, called POSIX 1003.1.b, an IEEE standard (POSIX stands for portable OS interfaces in Unix). OS semaphore functions P and V represent the semaphore by integer variables. A semaphore variable, apart from initialization, is accessed only through two standard atomic-operations: P and V. [P (for *wait* operation) is derived from a Dutch word 'Proberen', which means 'to test'. V (for *signal* notifying operation) is derived from the word 'Verhogen' which means 'to increment'.] (Atomic-operation is one, which can not be in parts.)

1. P semaphore function signals that the task requires a resource and if not available waits for it.
2. V semaphore function signals from the task to the OS that the resource is now free for the other users.

Consider P semaphore. It is a function, P (&sem_1) which, when called in a process, does the following operations using semaphore, sem_1.

1. /* Decrease the semaphore variable*/
sem_1 = sem_1 -1;
2. /* If sem_1 is less than 0, send a message to OS by calling a function waitCallToOS. Control of the process transfers to OS, because less than 0 means that some other process has already executed P function on sem_1. Whenever there is return to the OS, it will be to step 1. */
if (sem_1 < 0){waitCallToOS (sem_1);}

Consider V semaphore. It is a function, V(&sem_2) which, when called in a process, does the following operations using semaphore, sem_2.

3. /* Increase the semaphore variable*/
sem_2 = sem_2 + 1;
4. /* If sem_2 is less or equal to 0, send a message to OS by calling a function signalCallToOS. Control of the process transfers to OS, because < or = 0 means that some other process has already executed P function on sem_2. Whenever there is return to the OS, it will be to step 3. */
if (sem_2 <= 0){signalCallToOS (sem_2);}

Use of P and V Semaphore Functions with a Signal or Notification Property P and V functions can represent a signalling or notifying variable, sem_s when used as shown in Example 7.11.

Example 7.11

Let sem_s be a semaphore variable. Let it function as a signal or notifying an event using variable sem_s. P and V semaphore functions are used in two processes, task 1 and task 2 as follows.

<pre>Process 1 (Task 1) while (true) { /* Codes */ . . . V (&sem_s);</pre>	<pre>Process 2 (Task 2) while (true) { /* Codes */ . P (&sem_s); /* The following codes will execute only when sem_s is not less than 0. */</pre>
--	---

```
/* Continue Process 1 if sem_s is not equal to 0
or not less than 0. It means that no process is
executing at present. */
.
.
.
};
```

Use of P and V Semaphore Functions with a Mutex Property P and V functions can represent a mutex semaphore variable, sem_m when used as explained in Example 7.12.

Example 7.12

Let sem_1 and sem_2 be the same variable, sem_m. The latter functions as a mutex, as follows, when P and V semaphore functions are used in two processes, task 1 and task 2.

<pre>Process 1 (Task 1) while (true) { /* Codes before a critical region*/ . . . /* Enter Process 1 Critical region codes*/ P (&sem_m); /* The following codes will execute only when sem_m is not less than 0. */ . . . /* Exit Process 1 critical region codes */ V (&sem_m); /* Continue Process 1 if sem_m is not equal to 0 or not less than 0. It means that no process is waiting and has executed P function using sem_m at present. */ . . . };</pre>	<pre>Process 2 (Task 2) while (true) { /* Codes before a critical region*/ . . . /* Enter Process 2 Critical region codes*/ P (&sem_m); /* The following codes will execute only when sem_m is not less than 0. */ . . . /* Exit Process 2 critical region codes */ V (&sem_m); /* Continue Process 2 if sem_m is not equal to 0 or not less than 0. It means that no process is waiting and executed P function using sem_m at present. */ . . . };</pre>
--	--

The same variable, sem_m, is shared between process 1 and process 2. Its use is in making both processes gain mutually exclusive access to the resource (CPU). Either process 1 runs after executing P or process 2 runs after executing P. Also, either process 1 runs after executing V or process 2 runs after executing V.

Figure 7.7(a) shows the use of P and V semaphores and the task I, task J and the scheduler. When a task takes a semaphore P, if sem_m = 'true' (=1) earlier then it becomes 'false' (=0) and task continues as sem_m is not less than 0. When a task executes V, if sem_m was 'false' earlier then it sets to 'true' and the task continues running, else the task blocks and waits for the execution of another task. Figure 7.7(b) shows the PC assignments to a process or function when using P and V semaphores.

Use of P-V Semaphore Functions with a Counting Semaphore Property Let there be a process (task *c*). The P function decrements the count and the function increments the counts. The P function operates on a counting semaphore, *sem_c1* as shown in Example 7.13.

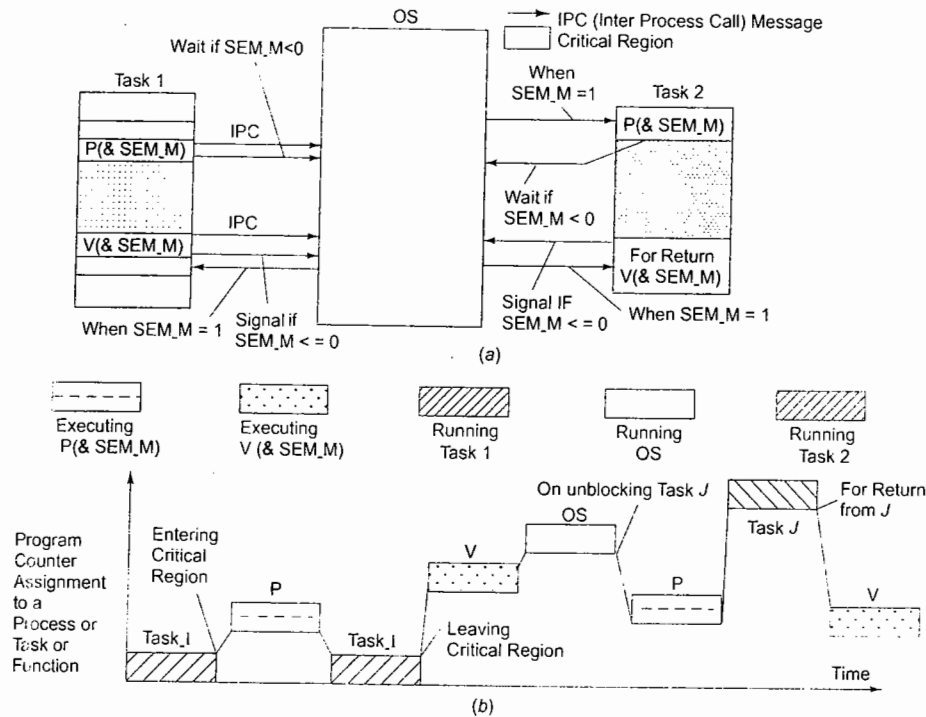


fig. 7.7 (a) Use of P and V semaphores at a task 1, and at another task 2 and at a scheduler (b) The program counter assignments to a process or function when using P and V semaphores

Example 7.13

Assume a processes using P semaphore functions in task, *task_c*. Let *sem_c1* be a counting semaphore variable and represent the number of empty places created by the process *c*. P functions operate on these and reduces the number of empty places as follows:

```
Process c (Task_c)
while (true) {
    /* Codes on entering a producer region*/
```

```
/* After exiting the producer Process 3 region codes */
P (&sem_c1);
/* Continue Process c if sem_c1 is not less than 0. */
};
```

Use of P and V Semaphore Functions with a Counting Semaphore Property for Bounded Buffer Problem Solution

Let a task generate the outputs for use by another task. A task can have a separate counting semaphore.

Consider three examples:

- (i) a task transmits bytes to an I/O stream for filling the available places at the stream;
- (ii) a process 'writes' an I/O stream to a printer buffer and,
- (iii) a task of producing chocolates is being performed.

In example (i) another task reads the I/O stream bytes from the filled places and creates empty places.

In example (ii), from the print buffer an I/O stream prints after a buffer-read and after the printing, more empty places are created.

In example (iii) again, as consumer consumes the chocolates produced more empty places (to stock the produced chocolates) are created.

A task blockage operational problem is commonly called producer-consumer problem. A task cannot transmit to the I/O stream if there are no empty places in the stream. The task cannot write from the memory at the print buffer if there are no empty places at the print buffer. The *producer cannot produce if there are no empty places at the consumer stock.*

A classic program for synchronization is called the *producer-consumer problem program*. It is also called *bounded buffer problem program*. Here, one or more producers (task or thread processes) create data outputs that are then processed by one or more consumers (tasks or processes). The data outputs from the producers are passed for processing by the consumers using some type of IPC (Section 7.8) that uses a shared memory and counting semaphores (or message queues or mailboxes).

Let there be two processes (tasks 3 and 4). The P and V functions operate on two shared counting semaphores, *sem_c1* and *sem_c2*, as in Example 7.14.

Example 7.14

Assume two processes using P and V semaphore functions and two tasks, tasks 3 and 4. Let *sem_c1* and *sem_c2* be two counting semaphore variables and represent the number of filled places created by the process 3 and a number of empty places created by process 4, respectively. P and V functions operate on these as follows.

```
Process 3 (Task 3)
while (true) {
```

```
Process 4 (Task 4)
while (true) {
```

```

/* Codes before a producer region*/
.
.
.
/* Enter Process 3 Producing region codes*/
P (&sem_c2);
/* The following codes will execute only when
sem_c2 (number of empty places)is not less
than 0. */
.
.
.
/* Exit Process 3 region codes */
V (&sem_c1);
/* Continue Process 3 if sem_c1 is not equal to 0
or not less than 0. */
.
.
.
};

/* Codes before a consumer region*/
.
.
.
/* Enter Process 4 Consuming region codes*/
P (&sem_c1);
/* The following codes will execute only when
sem_c1 (number of filled places) is not less
than 0. */
.
.
.
/* Exit Process 4 region codes */
V (&sem_c2);
/* Continue Process 4 if sem_c2 is not equal to 0
or not less than 0. It means that filled places are
still available at present. */
.
.
.
};

```

Two semaphores, `sem_c1` and `sem_c2` are shared between processes 3 and 4. When process 3 executes, it first reduces the number of empty places at process 4. When process 3 completes production, it increases the number of filled places at process 3. When process 4 consumes, it first reduces the number of filled places at process 3. When process 4 completes consumption, it increases the number of empty places at process 4. Either process 3 produces output after executing P, or process 4 consumes (uses) inputs after executing P. Also either process 3 proceeds after executing V or process 4 proceeds after executing V.

P and V semaphore functions are in POSIX 100.3b, an IEEE-accepted standard for the IPCs. They can be used as an event signalling, as a mutex, as a counting semaphore and the semaphores for the bounded buffer problem solution.

7.8 SHARED DATA

7.8.1 Problem of Sharing Data by Multiple Tasks and Routines

The shared data problem can be explained as follows: Assume that several functions (or ISRs or tasks) share a variable. Let us assume that at an instant the value of that variable operates and during its operations, only a part of the operation is completed and a part remains incomplete. At that moment, let us assume that there is an interrupt. Now, assume that there is another function. It also shares the same variable. The value of the variable may differ from the one expected if the earlier operation had been completed. The incomplete operation occurs as follows.

Suppose a variable is of 128 bits and the processor is of 32 bits. The operations on the variable will be using our 32-bit ALU operations in order to use the 32-bit ALU of this processor. Atomic operation is one, which

cannot be subdivided into the suboperations or none of the suboperations can be left incomplete and no other operation can start before all suboperations are complete. Now, assume that the 128-bit operation on the variable is non-atomic. This means that the operation can be interrupted before all the four operations are completed. An interrupt can occur at the end of each 32-bit ALU operation, not necessarily at the end of the 128-bit operation. Therefore, the called ISR or another function can use the incompletely operated variable or can change that variable when it shares with another function. On return, new values of that variable will be found and the incomplete operations will be performed on that new 128-bit variable in place of the old one.

Example 7.15

- Consider x , a 128-bit variable, $b_{127} \dots b_0$. Assume that the operation OP_{sl} is shift left by 2 bits to multiply it by 4 and find $y = 4 \times x$. Let OP_{sl} done non-atomically in four suboperations, OPA_{sl} , OPB_{sl} , OPC_{sl} and OPD_{sl} for $b_{31} \dots b_0$, $b_{63} \dots b_{32}$, $b_{95} \dots b_{64}$ and $b_{127} \dots b_{96}$, respectively. Assuming at an instance that suboperations OPA_{sl} , OPB_{sl} and OPC_{sl} are completed and OPD_{sl} remained incomplete. Now interrupt I occurs at that instance. The I calls some function which uses x if x is the global variable. It modifies $x = b'_{127} \dots b'_0$. On return from interrupt, as OPD_{sl} is not complete, OPD_{sl} operates on $b'_{127} \dots b'_{96}$ not on $b_{127} \dots b_{96}$.
- Consider date d and time t . Let d and t be taken in the program as global variables. Assume that a thread `Update_Time_Date` is updating t and d information on system clock tick interrupt I_s . The thread `Display_Time_Date` in Example 7.2 displays that t and d information.
 - Assume that when `Update_Time_Date` ran $t = 23:59:59$ and date $d = 17$ July 2007.
 - The `Display_Time_Date` gets interrupted and assumes that display d and operation t are non-atomic. Display of d was completed but display of t was incomplete when interrupt I_s occurred.
 - After a while, the t changes to $t = 00:00:00$ and $d = 18$ July 2007 when the thread `Update_Time_Date` runs. The display will show $t = 00:00:00$ and date $d = 17$ July 2007 on the return from interrupt and re-start of blocked thread `Display_Time_Date`.

The use of shared variables d and t in two threads `Update_Time_Date` and `Display_Time_Date` causes the display error.

7.8.2 Shared Data Problem Solutions

One solution is the use of atomic operation for solving the shared data problem. We need atomic operations because of the following.

- An interrupt can occur at the end of an instruction cycle, not at the end of a high-level instruction.
- A DMA operation can occur at the end of a machine cycle itself and a compiler or program may not taking these atomic-level details into account (DMA operation means direct memory access, an IO device loads into the memory using the system address and data buses when CPU is not performing any bus-operation).
- A context switch operation can occur at the end of an instruction for calling a new function, cycle itself and a compiler or program may not be taking into account these atomic-level details.

The following are the steps that, if used together, almost eliminate a likely bug in the program because of the shared data problem:

- Use modifier *volatile* with a declaration for a variable that returns from the interrupt. This declaration warns the compiler that certain variables can modify because the ISR does not consider the fact that the variable is also shared with a calling function.

2. Use *reentrant functions* with atomic instructions in that part of a function that needs its complete execution before it can be interrupted. This part is called the critical section. For example, the suboperations of shifting of x in Example 7.15 are in critical section.
3. Put a *shared variable in a circular queue*. A function that requires the value of this variable always deletes (takes) it from the *queue front*, and another function, which inserts (writes) the value of this variable, always does so at the *queue back*. Now a problem can occur in case there are a large number of functions that post the values into and take the values but the maximum required queue size is not provided.

Example 7.16

Example shows how shared data problem get solved by using queue. Consider the Example 7.15(b). Assume that variables t for time and d for date are shared variables and there is a queue Q_{TD} into which a thread inserts the shared variables and another thread deletes these variables. [Note: Insertion into a queue means writing a value at the queue tail and then changing the pointer to the queue tail for the next insertion. Deletion from a queue means reading the value from the queue head and then changing the pointer to the queue head for the next read.]

1. Assume that when *Update_Time_Date* runs the $t = 23:59:59$ and date $d = 17$ July 2007 and inserts these in a queue Q_{TD} .
2. The *Display_Time_Date* reads t and d from Q_{TD} and displays. When the thread gets interrupted after reading d , display of d , the Q_{TD} is still holding t when interrupt I_5 occurs. Display of t will complete on return from the interrupt.
3. After a while, the t changes to $t = 00:00:00$ and date $d = 18$ July 2007 and the thread *Update_Time_Date* runs and inserts the new values of t and d at the back of the earlier values in Q_{TD} . The display will show $d = 17$ July 2007 and $t = 23:59:59$ and on return from the interrupt and in the next cycle run the thread will show $d = 18$ July 2007 and $t = 00:00:00$.

The use of queue for the shared variables d and t in two threads when *Update_Time_Date* inserts these into the queue and *Display_Time_Date* deletes these from the Q_{TD} causes no display error.

4. *Disable the interrupts before a critical section starts executing and enable the interrupts on its completion.* It is a powerful but drastic option. An interrupt, even if of higher priority than the present critical function, gets disabled. Advantage of it is that the semaphore functions have greater computational overhead than disabling of the interrupt. The difficulty with this option is that it increases in the interrupt latency period for all the tasks. The latency increases by the time taken in executing the codes of the section. A deadline may be missed for an interrupt service by that task which does not share the critical section or the resource.

As an alternative to disabling interrupts, Section 7.7.2 described using of semaphores for the shared data problem. A software designer must not use the drastic option of disabling interrupts in all the critical sections. [Note: In the OS for automobile applications, the disabling of interrupts is used before entering any critical section to avert any unintended action because of improper use of semaphores.] Another alternative is use of lock or spin-lock functions in a scheduler. (Section 7.11)

The use of disabling the switching of task from one to another and other steps and use of semaphores (Section 7.7) must eliminate the shared data problem completely from a multitasking, multi-ISR's and multiple shared variable cases. Each of the step has its own inherent benefits in solving the problem. A programmer must utilize the various steps optimally suited to solve the problem.

Shared data problem can arise in a system when another higher priority task finishes an operation and modifies the data or a variable. Using *reentrant functions*, disabling interrupt mechanism, using semaphores and IPCs such as mailbox and queue are the solutions which are used for taking care of shared data problem.

7.8.3 Applications of Semaphores and Shared Data Problem

Use of mutex facilitates mutually exclusive access by two or more processes to the resource (CPU). The same variable, sem_m , is shared between the various processes. Let process 1 and process 2 share sem_m and its initial value = 1.

1. Process 1 proceeds after sem_m decreases and equals 0 and gets the exclusive access to the CPU.
2. Process 1 ends after sem_m increases and equals 1; process 2 now gets exclusive access to the CPU.
3. Process 2 proceeds after sem_m decreases and equals 0 and gets exclusive access to CPU.
4. Process 2 ends after sem_m increases and equals 1; process 1 now gets the exclusive access to the CPU.

The sem_m is like a resource key and shared data within the processes 1 and 2 is the resource. Whosoever first decreases it to 0 at the start gets the access to and prevents the other to run with whom this key shares.

Mutex is a semaphore that provides at an instance two tasks mutually exclusive access to resources and is used in solving shared data problem.

7.8.4 Elimination of Shared Data Problem

The use of semaphores does not eliminate the shared data problem completely. Software designers may not take the drastic option of disabling interrupts in all the critical sections by using semaphores. When using semaphores, the OS does not disable the interrupts. Alternatively, task-switching flags can be used (Section 8.10.3). The following problems that can arise when using semaphores.

1. Sharing of two semaphores creates a deadlock problem (Refer to Section 7.8.5).
2. Suppose the semaphore taken is never released? There should therefore be some time-out mechanism after which the error message is generated or an appropriate action taken. There is some degree of similarity with the watchdog timer action on a time-out. A watchdog timer on timeout resets the processor. Here, after the time out, the OS reports an error and runs an error-handling function. Without a time out, an ISR worst-case latency may exceed the deadline.
3. A semaphore not taken, and another task uses a shared variable.
4. What happens when a train takes a signal for a wrong track? When using the multiple semaphores, if an unintended task takes the semaphore, it creates a problem.
5. There may be priority inversion problem (Refer to Section 7.8.5).

7.8.5 Priority Inversion Problem and Deadlock Situations

Let the priorities of tasks be in an order such that task I is of the highest priority, task J is of a lower and task K of the lowest. Assume that only tasks I and K share the data and J does not share data with K . Also let tasks I and K alone share a semaphore and not J . Why do only a few tasks share a semaphore? Can't all share a semaphore? The reason is that the worst-case latency becomes too high and may exceed the deadline if all tasks are blocked when one task takes a semaphore. The worst-case latency will be small only if the time taken by the tasks that share the resources is relevant. Now consider the following situation.

At an instant t_0 , suppose task K takes a semaphore, the OS does not block task J and blocks task I . This happens because only tasks I and K share the data and J does not. Consider the problem that now arises on selective sharing between K and I . At the next instant t_1 , let task K become ready to run first on an interrupt. Now, assume that at the next instant t_2 , task I becomes ready on an interrupt. At this instant, K is in the critical section. Therefore, task I cannot start at this instant due to K being in the critical region. Now, if at next instant t_3 , some action (event) causes the unblocked higher than the K priority task J to run. After instant t_3 , running task J does not allow the highest priority task I to run. This is because even though K is not running and thus

unable to release the semaphore that it shares with *I*. Further, the code of task *J* may be such that even when the semaphore is released by task *K*, it may not let *I* run (*J* runs the codes as if it is in critical section all the time). The *J* action is now as if *J* has higher priority than *I*. This is because *K*, after entering the critical section and taking the semaphore when the OS is letting *J* run, but did not share the priority information about *I*—that task *I* is of higher priority than *J*. The priority information of another higher-priority task *I* should have also been inherited by *K* temporarily, if *K* waits for *I* but *J* does not and *J* runs when *K* has still not finished the critical section codes. This did not happen because the given OS design was such that it did not provide for temporary priority inheritance in such situations.

This situation is also called *priority inversion problem*. An OS must provide for a solution for the priority inversion problem. Some OSes provide for priority inheritance in these situations and thus priority inheritance problem does not occur when using them. Refer to Section 7.7.2 for use of a mutex for resources sharing. A mutex should be a mutually exclusive Boolean function, by using which the critical section is protected from interruption in such a way that the problem of priority inversion does not arise. Mutex is automatically provided in certain RTOS so that the priority inversion problem does not arise. Mutex use may also be just analogous to a semaphore defined in Section 7.7.2 in another RTOS and which does not solve the priority inversion problem.

Consider another problem. Assume the following situation.

1. Let the priorities of tasks be such that task *H* is of highest priority. Then task *I* has a lower priority and task *J* has the lowest.
2. There are two semaphores, *SemTok1* and *SemTok2*. This is because the tasks *I* and *H* have a shared resource through *SemTok1* only. Tasks *I* and *J* have two shared resources through two semaphores, *SemTok1* and *SemTok2*.
3. Let *J* interrupt at an instant t_0 and first take both the semaphores *SemTok1* and *SemTok2* and run.

Assume that at a next instant t_1 , being now of a higher priority, the task *H* interrupts the tasks *I* and *J* after it takes the semaphore *SemTok1*, and thus blocks both *I* and *J*. In-between the time interval t_0 and t_1 , the *SemTok1* was released but *SemTok2* was not released during the run of task *J*. But the latter did not matter as the tasks *I* and *J* do not share *SemTok2*. At an instant t_2 , if *H* now releases the *SemTok1*, allows the task *I* to take it. Even then it cannot run because it is also waiting for task *J* to release the *SemTok2*. The task *J* is waiting at a next instant t_3 , for either *H* or *I* to release the *SemTok1* because it needs this to again enter a critical section. Neither task *I* can run after instant t_3 nor task *J*. There is a circular dependency established between *I* and *J*.

This situation is also called a *deadlock* situation. On the interrupt by *H*, the task *J*, before exiting from the running state, should have been put in queue-front so that later on, it should first take *SemTok1*, and the task *I* put in queue next for the same token, the deadlock would not have occurred (refer to Section 7.12 for queuing of messages).

The use of mutex solves the deadlock problem in certain OSes. Its use may be just analogous to a semaphore defined in Section 7.2.1. In other OSes, the mutex use may not solve the deadlock situation.

Priority becomes inverted and deadlock (circular dependency) develops in certain situations when using semaphores. Certain OSes provide the solution to this problem of semaphore use by ensuring that these situations do not arise during the concurrent processing of multitasking operations.

7.9 INTERPROCESS COMMUNICATION

Is it possible to send through the kernel an output data (a message of a known size with or without a header) or processing by another task? One way is the use of global variables. Use of these now creates two problems.

One is the shared data problem (Section 7.8). The other problem is that the global variables do not prevent (encapsulate) a message from being accessed by other tasks.

IPCs in a multiprocessor system are used to generate information about certain sets of computations finishing on one processor and to let the other processors waiting for finishing those computations take note of the information. IPC means that a process (scheduler, task or ISR) generates some information by signal or value or generates an output so that it allows another process to take note or use it through the kernel functions for the IPCs. IPCs in a multitasking system are used to set or reset a signal or token or flag or generate message from the certain sets of computations finishing on one task and to let the other tasks take note of the signal or get the message.

OSes provide the software programmer the following IPC functions, which can be used.

1. Signals
2. Semaphores as *token* or *mutex* or counting semaphores for the intertask communication between tasks sharing a common buffer or operations
3. Queues and mailboxes
4. Pipes and sockets
5. Remote procedure calls (RPCs) for distributed processes.

Section 7.7 described two IPC functions, *OSSemPend ()* and *OSSemPost ()* and use of semaphores for the IPCs. The following shows an application for the printing from buffer by a task.

Example 7.17

Consider using a mutex semaphore in the tasks, which needs to use the *print* for the buffer data from one task at an instance. A task runs a *print* function, which reads from a print buffer and prints. The kernel should let the other tasks share this task. The *print* task can be shared among the multiple tasks, which use the mutex semaphore IPCs in their critical sections.

When the printer buffer becomes available for new data, an IPC from the *print* task is generated and the kernel takes note of it. Other tasks then take note of it. A task takes note of it by the *OSSemPend ()* function of the kernel used at the beginning of the critical section and the task gets mutually exclusive access to the section to send messages into the print buffer by using the *OSSemPost ()* function of the kernel at the end of the section (Sections 7.7.2, 7.8.3 and 7.11.1).

Consider Example 7.18, which shows use of the functions for semaphore and mailbox IPCs (semaphore and mailbox IPC functions will be described in detail in Sections 7.11 and 7.13).

Example 7.18

Consider a mobile phone device (Section 1.10.5) *Update_Time* task (Example 7.8). Assume that there is a task, *Task_Display* for a multiline display of outputs which displays current time on the last line. When the multiline display task finishes the display of the last but one line, an IPC semaphore *supdateTD* from the *display* task is generated and the kernel takes note of it. The task—continuously updating time—then can take the *supdateTD* and generate an IPC as a mailbox output for the current time and date.

When the task for updating time *t* on a signal posted on system clock-tick interrupt I_3 starts, on taking the *supdateTD* posted by the *Task_Display*, it can write the *td* into a mailbox using an IPC for posting mailbox message, *timeDate*.

Assume *OSSemPost ()* is an OS IPC function for posting a semaphore and assume *OSSemPend ()* is another OS IPC function for waiting for the semaphore. Let *supdateTD* be the binary semaphore posted at *Task_Display* and pending at *Task_Display* section for displaying *t* and *d*. Let *supdateTD* initial value = 1.

Let the IPC function be `OSMboxPost ()` for posting the mailbox IPC message from `Update_Time` task and `OSMboxPend ()` for waiting for the mailbox IPC at `Task_Display` section. Let `timeDate` initial value = null.

The following will be the codes:

```
static void Task_Display (void *taskPointer) {
while (1) {
/* IPC for waiting time and date in the mailbox */
TimeDateMsg = OSMboxPend (timeDate) /* Wait for the mailbox message timeDate. The timeDate becomes
null after the mailbox is posted time and date by Task_Update_Time and TimeDateMsg equals the
updated time and date */
/* Code for display TimeDateMsg Time: hr:mm Date: month:date */
/* IPC for requesting TimeDate */
OSSemPost (supdateTD) /* Post for the semaphore supdateTD. supdateTD becomes 1
*/
static void Task_Update_Time (void *taskPointer) {
while (1) { /* wait for system clock inter interrupt signal or semaphore notification from ISR of I_s */
OSSemPend (supdateTD) /* Wait the semaphore supdateTD. This means that OS function decrements
supdateTD in corresponding event control block. supdateT becomes 0 */
/* Codes for updating time and date as per the number of clock interrupts received so far */
/* Codes for writing into the mailbox */
OSMboxPost (timeDate) /* Post for the mailbox message and timeDate, which equaled null
now equals newupdated time and date */
};
};
```

The need for IPC and thus intertask communications also arises in a client-server network.

IPC means that a process (scheduler or task or ISR) generates some information by setting or resetting a token or value, or generates an output so that it lets another process take note or use it under the control of OS.

7.10 SIGNAL FUNCTION

One way for messaging is to use an OS function `signal ()`. It is provided in Unix, Linux and several RTOSes. Unix and Linux OSes use `signals` profusely and have 31 different types of `signals` for the various events. Section 9.3 will describe `signal` in VxWorks RTOS. Just a hardware mechanism sends the interrupt to the OS, task (or process) or the OS itself sends `signal`. The task or process sending the signal uses a function `signal ()` having an integer number `n` in the argument. A signal is function, which executes a software interrupt instruction `INT n` or `SWI n`.

A 'signal' provides the shortest communication. The `signal ()` sends an output `n` for a process, which enables the OS to unmask a signal mask of a process or task as per the `n`. The task is called signal handler and has coding similar to the ones in an ISR. The handler runs in a way similar to a highest priority ISR. An ISR runs on an hardware interrupt provided that the interrupt is not masked. The handler runs on the signal provided that the signal is not masked.

The `signal ()` forces the OS to first run a signalled process or task called signal handler. When there is return from the signalled or forced task or process, the process, which sent the signal, runs the codes as happens on a return from an ISR. A signal mask is the software equivalent of the flag at a register that sets on masking a hardware interrupt. Unless masked by a `signal` mask, the `signal` allows the execution of the signal-handling function and allows the handler to run just as a hardware interrupt allows the execution of an ISR.

An integer number (for example `n`) represents each signal and that number associates a function (or process or task) signal handler, an equivalent of the ISR. The signal handler has a function called whenever a process communicates that number.

A signal handler is not called directly by a code. When the signal is sent from a process, OS interrupts the process execution and calls the function for signal handling. On return from the signal handler, the process continues as before.

For example, `signal (5)`. The signal mask of signal handler 5 is reset. The signal handler and connect function associate the number 5. The function represented by number 5 is forced run by the signal handler.

An advantage of using it is that unlike semaphores it takes the shortest possible CPU time to force a handler to run. The `signals` are the interrupts that can be used as the IPC functions of synchronizing.

A `signal` is unlike the semaphore. The semaphore has use as a token or resource key to let another task process block, or which locks a resource to a particular task process for a given section of the codes. A signal is just an interrupt that is shared and used by another interrupt-servicing process. A signal raised by one process forces another process (signal handler) to interrupt and catch that signal in case the signal is not masked (use of that signal handler is not disabled). Signals are to be handled only for forcing the run of very high priority processes as it may disrupt the usual schedule and priority inheritance mechanism. It may also cause reentrancy problems.

An important application of the `signals` is to handle exceptions. (An exception is a process that is executed on a specific reported run-time condition.) A `signal` reports an error (called 'exception') during the running of a task and then lets the scheduler initiate an error-handling process or function or task. An error-handling signal handler handles the different error login of other task. The device driver functions also use the signals to call the handlers (ISRs).

The following are the signal related IPC functions, which are generally not provided in the RTOS such as μ COS-II and provided in RTOS such as VxWorks or OS such as Unix and Linux.

1. `SigHandler ()` to create a signal handler corresponding to a signal identified by the signal number and define a pointer to the signal context. The signal context saves the registers on signal.
2. `Connect` an interrupt vector to a signal number, with signalled handler function and signal-handler arguments. The interrupt vector provides the PC value for the signal-handler function address.
3. A function `signal ()` to send a signal identified by a number in the argument to a task.
4. `Mask` the signal.
5. `Unmask` the signal.
6. `Ignore` the signal.

1. The simplest IPC for messaging from processes that forces a handler function to run (provided unmasked) is the use of 'signal'.
2. A 'signal' provides the shortest communication. Signals are used for initiating exceptions and error-handling processes.
3. IPC function for a signal is `signal (n)` for signalling signal-handler associated with `n` to run if not masked.

7.11 SEMAPHORE FUNCTIONS

The OS provides for semaphore as notice or token for an event occurrence. Semaphore facilitates IPC for notifying (through a scheduler) a waiting task section change to the running state upon event at presently running code section at an ISR or task. A semaphore as binary semaphore is a token or resource key (Sections 7.7.1 and 7.7.2). The OS also provides for mutex access to a set of codes in a task (or thread or process) (Sections 7.7.2 and 7.8.3). The use of mutex is such that the priority inversion problem is not solved in some OSes while it is solved in other OSes. The OS also provides for counting semaphores. The OS may provide for POSIX standard P and V semaphores which can be used for notifying event occurrence or as mutex or for counting. The timeout can be defined in the argument with wait function for the semaphore IPC. The error pointer can also be defined in the arguments for semaphore IPC functions.

The following are the functions, which are generally provided in an OS, for example, μ COS-II for the semaphores.

1. *OSSemCreate*, a semaphore function to create the semaphore in an event control block (ECB). Initialize it with an initial value.
2. *OSSemPost*, a function which sends the semaphore notification to ECB and its value increments on event occurrence (used in ISRs as well as tasks).
3. *OSSemPend*, a function, which waits the semaphore from an event, and its value decrements on taking note of that event occurrence (used in tasks not in ISRs).
4. *OSSemAccept*, a function, which reads and returns the present semaphore value and if it shows occurrence of an event (by non-zero value) then it takes note of that and decrements that value (no wait; used in ISRs as well as tasks).
5. *OSSemQuery*, a function, which queries the semaphore for an event occurrence or non-occurrence by reading its value and returns the present semaphore value, and returns pointer to the data structure *OSSemData*. The semaphore value does not decrease. The *OSSemData* points to the present value and table of the tasks waiting for the semaphore (used in tasks).

An OS provides the IPC functions create, post, pend, accept and query for using semaphores. The time-out can be provided with 'pend' function arguments. A pointer to error-handling function can also be specified in the arguments.

7.11.1 Mutex, Lock and Spin Lock

An OS, using a mutex blocks a critical section in a task on taking the mutex by another task's critical section. Other task unlocks on releasing the mutex. The mutex wait by blocked task can be for a specified timeout.

There is a function in kernel called lock (). It locks a process to the resources till that process executes unlock (). A wait loop creates and when wait is over the other processes waiting for the lock starts. Use of lock () and unlock () involves little overhead compared to uses of *OSSemPend* () and *OSSemPost* () when using a mutex. Overhead means number of operations needed for blocking one process and starting another. However, a resource of high priority should not lock the other processes by blocking an already running task in the following situation. Suppose a task is running and a little time is left for its completion. The running time left for it is less compared with the time that would be taken in blocking it and context switching. There is an innovative concept of spin locking in certain OS schedulers. A spin lock is a powerful tool in the situation described before. [Refer to 'Multithreaded Programming with Java' by

Bil Lewis and Daniel J. Berg, Sun Microsystems Inc., 2000.] The scheduler locking process for a task *I* waits in a loop to cause the blocking of the running task first for a time interval *t*, then for $(t - \delta t)$, then $(t - 2\delta t)$ and so on. When this time interval spin-downs to 0, the task that requested the lock of the processor now unlocks the running task *I* and blocks it from further running. The request is now granted to task *J* to unlock and start running provided that task is of higher priority. A spin lock does not let a running task to be blocked instantly, but first successively tries with or without decreasing the trial periods before finally blocking a task. A spin-lock obviates need of context-switching by pre-emption and use of mutex function-calls to OS.

An OS provides the IPC functions for creating and accessing the resource using mutex for a process and to prevent the resource for the other processes. An OS may also provides for lock or spin-locks at the scheduler.

7.12 MESSAGE QUEUE FUNCTIONS

Some OSes do not distinguish, or make little distinction, between the use of queues, pipes and mailboxes during the message communication among processes, while other OSes regard the use of queues as different. A message queue is an IPC with the following features.

1. An OS provides for inserting and deleting the message pointers or messages.
 2. Each queue for the message or message-pointers needs initialization (creation) before using functions in kernel for the queue.
 3. Each created queue has an ID.
 4. Each queue has a user-definable size (upper limits for number of bytes).
 5. When an OS call is to insert a message into the queue, the bytes are as per the pointed number of bytes. For example, for an integer or float variable as a pointer, there will be 4 bytes inserted per call. If the pointer is for an array of eight integers, then 32 bytes will be inserted into the queue. When a message-pointer is inserted into queue, the 4 bytes inserts, assuming 32-bit addresses.
 6. When a queue becomes full, there is error handling function to handle that.
- Figure 7.8(a) shows functions for the queues in the OS. Figure 7.8(b) shows a queue-message block with the messages or message-pointers. Two pointers, *QHEAD and *QTAIL are for queue head and tail memory locations. The OS functions for a queue, for example, in μ COS-II, can be as follows:
1. *OSQCreate*, a function that creates a queue and initializes the queue.
 2. *OSQPost*, a function that sends a message into the queue as per the queue tail pointer, it can be used by tasks as well as ISRs.
 3. *OSQPend* waits for a queue message at the queue and reads and deletes that when received (wait, used by tasks only, not used by ISRs).
 4. *OSQAccept* deletes the present message at queue head after checking its presence yes or no and after the deletion the queue head pointer increments. (no wait; used by ISRs as well as tasks)
 5. *OSQFlush* deletes the messages from queue head to tail. After the flush the queue head and tail points to QTop, which is the pointer at start of the queuing. (used by ISRs and tasks)
 6. *OSQQuery* queries the queue message block but the message is not deleted. The function returns pointer to the message queue *Qhead if there are the messages in the queue or else returns NULL. It returns a pointer to the structure of the queue data structure for *QHEAD, number of queued messages, size and table of tasks waiting for the messages from the queue. (query is used in tasks)
 7. *OSQPostFront* sends a message to front pointer, *QHEAD. Use of this function is made in the following situations. A message is urgent or is of higher priority than all the previously posted message into the queue (used in ISRs and tasks).

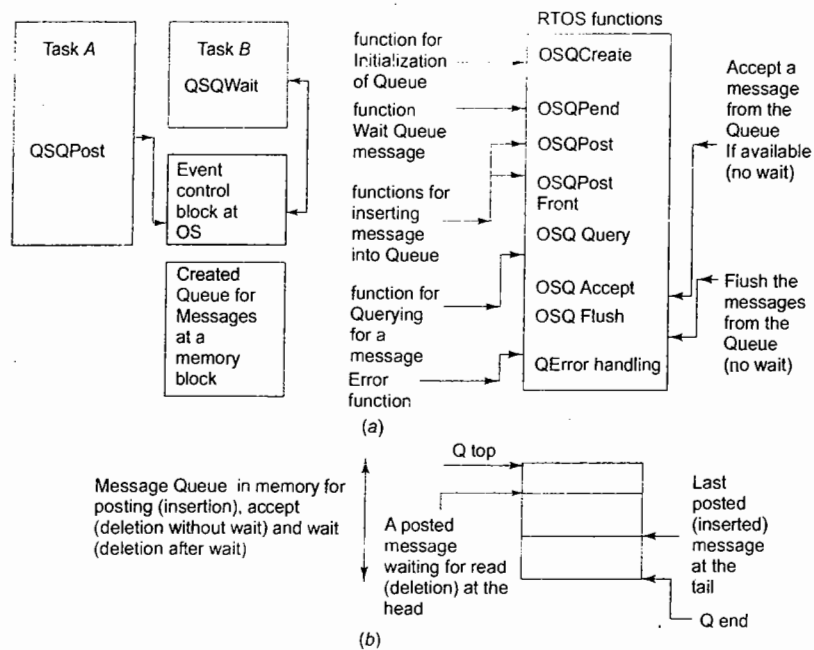


Fig. 7.8 (a) OS functions for queue and use of post and wait functions by Tasks A and B
(b) Queue message block in memory

Example 7.19

Consider Orchestra Playing Robots (Example 1.10.7). `Task_Director_Output` puts the musical notes into the queue at conducting and directing robot. `OSQEntries` equals the number of queue entries and `OSQSize` equals the maximum number of notes that can be put into the queue.

```
static void Task_Director_Output (void *taskPointer) {
while (1) {
/* Codes for inserting musical notes into the queue */
for (OSQEntries = 0; OSQEntries < OSQSize; OSQEntries ++ )
{ OSQPost (QDirector, note) } /* Post for the Queue QDirector messages upto the OSQSize */
};
static void Task_Player_Input (void *taskPointer) {
while (1) {
```

```
/* Codes for deleting notes from the queue */
for (OSQEntries = OSQSize; OSQEntries >0 ; OSQEntries -- )
note (i) = OSQPend (QDirector, 0, err) /* wait for the message */
};
```

In certain RTOS, a queue is given select option and the option is provided for priority or FIFO. The task having priority if started deletes a queue message first in case the priority option is selected. The task pending since longer period deletes a queue message first in case the FIFO option is selected.

An OS provides the IPC functions create, post, postfront, pend, accept, flush and query for using message queues. The timeout can be provided with 'pend' function argument. The error-pointer can also be provided in the argument.

7.13 MAILBOX FUNCTIONS

A message-mailbox is for an IPC message that can be used only by a single destined task. The mailbox message is a message-pointer or can be a message. (μ COS-II provides for sending message-pointer into the box). The source (mail sender) is the task that sends the message pointer to a created (initialized) mailbox (the box initially has the NULL pointer before the message posts into the box). The destination is the place where the `OSMBoxPend` function waits for the mailbox message and reads it when received.

A mobile phone LCD display task is an example that uses the message mailboxes as an IPC. In the mailbox, when the time and date message from a clock-process arrives, the time is displayed at side corner on top line. When the message from another task to display a phone number, it is displayed the number at middle at a line. When the message from another task to display the signal strength at antenna, it is displayed at the vertical bar on the left.

Another example of using a mailbox is the mailbox for an error-handling task, which handles the different error logins from other tasks.

Figure 7.9(a) shows three mailbox-types at the different RTOSes. Figure 7.9(b) shows the initialization and other functions for a mailbox at an OS. The following may be the provisions at an OS for IPC functions when using the mailbox.

1. A task may put into the mailbox only a pointer to the message-block or number of message bytes as per the OS provisioning.
2. There are three types of the mailbox provisions.

A queue (Section 7.12) may be assumed a special case of a mailbox with provision for multiple messages or message pointers. An OS can provide for *queue* from which a read (deletion) can be on a FIFO basis or alternatively an OS can provide for the multiple mailbox messages with each message having a priority parameter. The read (deletion) can then only be on priority basis in case mailbox message has multiple messages with priority assigned to high priority ones. Even if the messages are inserted in a different priority, the deletion is as per the assigned priority parameter.

An OS may provision for *mailbox* and *queue* separately. A mailbox will permit one message pointer per box and the queue will permit multiple messages or message pointers. μ COS-II RTOS is an example of such an OS.

The RTOS functions for mailbox for the use by the tasks can be the following:

1. `OSMBoxCreate` creates a box and initializes the mailbox contents with a NULL pointer.
2. `OSMBoxPost` sends (writes) a message to the box.

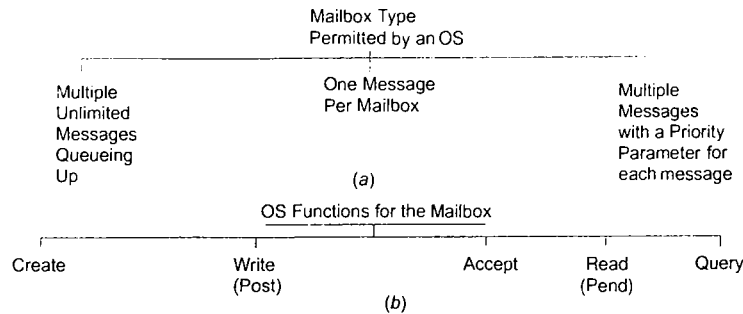


Fig. 7.9 (a) Mailbox types at the different operating systems (OSes) (b) Initialization and other functions for a mailbox at an OS

- OSMBoxWait (pend) waits for a mailbox message, which is read when received.
 - OSMBoxAccept reads the current message pointer after checking the presence yes or no (no wait). Deletes the mailbox when read.
 - OSMBoxQuery queries the mailbox when read and not needed later.
- An ISR can post (but not wait) into the mailbox of a task.

Example 7.20

(a) Consider an AVCM (Section 1.10.2). Assume that a message pointer IPC posts into the mailbox the amount collected by *Task Read_Amount* and the *Chocolate_delivery_task* section waits for taking message into the mailbox to make the amount equal to NULL after delivering the chocolate. Assume *mboxAmt* is a pointer to mailbox and *fullAmount* is a string *full amount* which should be made NULL after delivering the chocolate. OSMboxPost (*mboxAmt*, *full Amount*) is an OS IPC function for posting a message pointer into the mailbox and assume OSSemPend () is another OS IPC function for waiting for the message pointer, *fullAmount*.

(b) Also assume that the OSMboxPost () is also used by keypad for posting the mailbox IPC message for *userInput* into mailbox *mboxUser* from *Task User Keypad Input* and OSMboxPend () for waiting for *userInput* for display. A mailbox IPC 'pend' is for *mboxUser* message in *Task_Display*.

The following will be the codes for (a) and (b).

```
(a) static void Task_Read_Amount (void *taskPointer) {
.
while (1) {
.
/* Codes for reading the coins inserted into the machine */
/* Codes for writing into the mailbox full amount message if cost of chocolate is received*/
OSMboxPost (mboxAmt, fullAmount) /* Post for the mailbox message and fullAmount, which equaled
NULL now equals fullAmount */
.
};
static void Chocolate_delivery_task (void *taskPointer) {
while (1) {
```

```
/* IPC for requesting full amount message */
fullAmountMsg = OSMboxPend (mboxAmt, 20, *err) /* Wait for the mailbox mboxAmt message for
20 clock-ticks and error if message not found. mboxAmt becomes NULL after message is read.
.
};
(b) static void Task_User_Keypad_Input (void *taskPointer) {
.
while (1) {
.
/* Codes for reading keys pressed by the user before the enter key */
/* Codes for writing into the mailbox */
OSMboxPost (mboxUser, userInput) /* Post for the mailbox message and userInput, which equaled NULL
now equals userInput */
.
};
static void Task_Display (void *taskPointer) {
.
while (1) {
.
/* IPC for waiting for User input message */
UserInputMsg = OSMboxPend (mboxUser, 20, *err) /* Wait for the mailbox mboxUser
message for 20 clock ticks and error if message not found. mboxUser becomes null after
message is read.
.
/* Code for display of user Input */
TimeDateMsg = OSMboxPend (timeDate, 20, err) /* Wait for the mailbox message timeDate.
/* Code for display TimeDateMsg Time: hr:mm Date: month:date */
.
};
```

An OS provides the IPC functions *create*, *post*, *pend*, *accept* and *query* for using the mailbox. The time-out and error-pointer can be provided with the 'pend' function arguments.

7.14 PIPE FUNCTIONS

The OS pipe functions are unlike message queue functions. The difference is that pipe functions are similar to the ones used for devices such as file.

A message-pipe is a device for inserting (writing) and deleting (reading) from that between two given interconnected tasks or two sets of tasks. Writing and reading from a pipe is like using a C command *fwrite with a file name* to write into a named file, and *fread with a file name* to read from a named file. Pipes are also like Java *PipeInputOutputStreams*.

- One task using the function *fwrite* in a set of tasks, can insert (write) to a pipe at the back pointer address, *pBACK.
- Another task using the function *fread* in a set of tasks can delete (read) from a pipe at the front pointer address, *pFRONT.

3. In a pipe there may be no fixed number of bytes per message but there is end-pointer. A pipe can therefore be inserted limited number of bytes and have a variable number of bytes per message between the initial and final pointers.

4. Pipe is unidirectional. One thread or task inserts into it and the other one deletes from it.

An example of the need for messaging and thus for IPC using a pipe is a network stack.

The OS functions for pipe are the following:

1. *pipeDevCreate* for creating a device, which functions as pipe.
2. *open* () for opening the device to enable its use from beginning of its allocated buffer. its use is with options and restrictions (or permissions) defined at the time of opening.
3. *connect* () for connecting a thread or task inserting bytes to the thread or task deleting bytes from the pipe.
4. *write* () function for inserting (writing) from the bottom of the empty memory space in the buffer allotted to it.
5. *read* () function for deleting (reading) from the pipe from the bottom of the unread memory spaces in the buffer filled after writing into the pipe.
6. *close* () for closing the device to enable its use from beginning of its allocated buffer only after opening it again.

Figure 7.10(a) shows functions at an OS. A function is for initialization and creating a pipe. It defines pipe ID, length, maximum length (not defined in some OSEs) and initial values of two pointers. These are *pFRONT and *pBACK for pipe message destination (head) and pipe message source (tail) memory locations, respectively. A function is for pipe connecting and thus defining source ID and destination ID. A function is for the error handling. Figure 7.10(b) shows pipe messages in a message buffer.

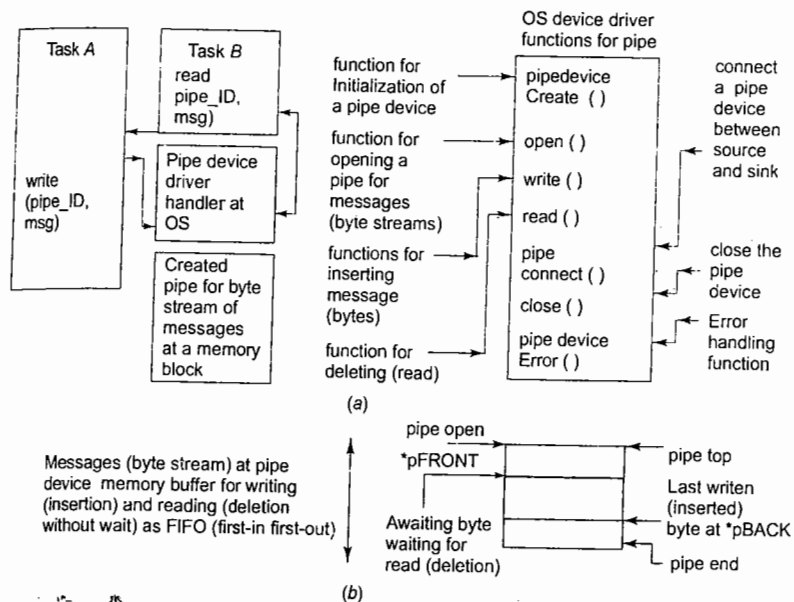


Fig. 7.10 (a) Functions at operating system (initialization, connect, read, write and error-handling functions) and the use of write and read functions by tasks A and B (b) Pipe messages in a message buffer

Example 7.21

Consider a smart card. [Section 1.10.3 and Example 7.4] When it is inserted into a card reader host machine, it gets the radiation and charges up.

1. Assume that the main program runs an OS function *pipeDevCreate* () to create a task.
2. Assume that a pipe is used by the task, *Task_Send_Card_Info* for writing card information to the host using the pipe.

The codes at the card can be as follows.

```
pipeDevCreate ("/pipe/pipeCardInfo", 4, 32) /* Create a pipe pipeCardInfo, which can save four messages
each of 32 bytes maximum */
fd = open ("/pipe/pipeCardInfo", O_WRONLY) /* Open a write only device. First argument is pipe ID /pipe/
pipeCardInfo, second argument is option O_WRONLY for specifying write only and third argument is 0 for
unrestricted permission.*/
```

```
static void Task_Send_Card_Info (void *taskPointer) {
```

```
while (1) {
```

```
cardTransactionNum = 0; /* At start of the transactions with the machine*/
write (fd, cardTransactionNum, 1) /* Write 1 byte for transaction number after card insertion */
write (fd, cardFabricationkey, 16) /* Write 16 bytes for fabrication key */
write (fd, cardPersonalisationkey, 16) /* Write 16 bytes for personalisation key */
write (fd, cardPIN, 16) /* Write 16 bytes for PIN, personal identification number
granted by the authorising bank */
```

```
};
```

An OS provides the IPC functions *pipeDevcreate*, *open*, *connect*, *write*, *read* and *close*. The pipe ID, limit of the total number of messages and the maximum size per message are also provided when creating a pipe.

7.15 SOCKET FUNCTIONS

Example 7.21 showed that a pipe could be used for inserting the byte stream by a process and deleting the bytes from the stream by another process. However, the use of pipe between a process at the card and a process at the host will have the following problems.

1. We need the card information to be transferred from a process A as bytes stream to the host machine process B and the B sends messages as bytes stream to A. There is need for bi-directional communication between A and B.
2. We need the A and B's ID or port as well as address information when communicating. These must be specified either for the destination alone or for both source and destination (It is similar to sending the messages in a letter along with address specification).

A protocol provides for communication along with the byte stream information of the address or port of the destination alone or addresses or ports of both source and destination. A protocol may provide for the addresses as well as ports of both source and destination in case of the remote processes (for example, in IP protocol). Also, there are two types of the protocols.

1. There may be the need of using a *connectionless protocol* when sending and receiving message streams. An example of such a protocol is UDP (user datagram protocol). UDP protocol requires a UDP header, which contains source port (optional) and destination port numbers, length of the datagram and checksum for the header-bytes. Port means a process or task for specific application. The number specifies the process. Connectionless means there is no connection establishment between source and destination before actual transfer of data stream can take place. Datagram means a set of data, which is independent and need not be in sequence with the previously sent data. Checksum is sum of the bytes to enable the checking of the erroneous data transfer. For remote communication, the address, for example, IP address is also required in the header.
2. There may be the need of using a *connection-oriented protocol*, for example, TCP. Connection-oriented protocol means a protocol, which provides that there must first a connection establishment between the source and destination, and then the actual transfer of data stream can take place. At end, there must be connection termination.

Socket provides a device-like mechanism for bi-direction communication. It provides for using a protocol between the source and destination processes for transferring the bytes; it provides for establishing and closing a connection between the source and destination processes using a protocol for transferring the bytes; it may provide for listening from multiple sources or multicasting to multiple destinations. Two tasks at two distinct places are locally interconnect through the sockets. Multiple tasks at multiple distinct places interconnect through the sockets to a socket at a server process. The client and server sockets can run on the same CPU or at distant CPUs on the Internet.

Sockets can be using a different domain. For example, a socket domain can be TCP, another socket domain may be UDP, the card and host example socket domains are different.

The use of a socket for IPC is analogous to the use of sockets for an Internet connection between the browser and webserver. A socket provides a bi-directional transfer of messages and may also send protocol reader. The transfer is between two or between the multiple clients and server-process. Each socket may have the task source address (similar to a network or IP address) and a port (similar to a process or thread) number. The source and destination sets of tasks (addresses) may be on the same computer or on a network. Figure 7.11 shows the initialized sockets between the client set of tasks and a server set of tasks at an OS.

Example 7.22

Assume that using the OS socket functions, a socket interconnects a byte stream between the source set of processes I and destination targeted set of processes J . Let there be the four process threads: a, b, c and d in process set I . There are two threads, x and y in a set of processes, J . Let the socket be used to send a byte stream from (process set I , process thread c) to a (process set J , process thread x). Now, the socket at the source (process I socket) is specified as the socket at (I, c) and socket at the process 2 as the socket at (J, x) . When the bytes are sent or received at the socket at (I, c) from or to the socket at (J, x) , the protocol specifies (I, c) and (J, x) . The protocol may also specify the length of the bytes being communicated. The protocol may also specify the checksum of the bytes being communicated so that if any bit is lost in communication to remote then retransmission request can be sent.

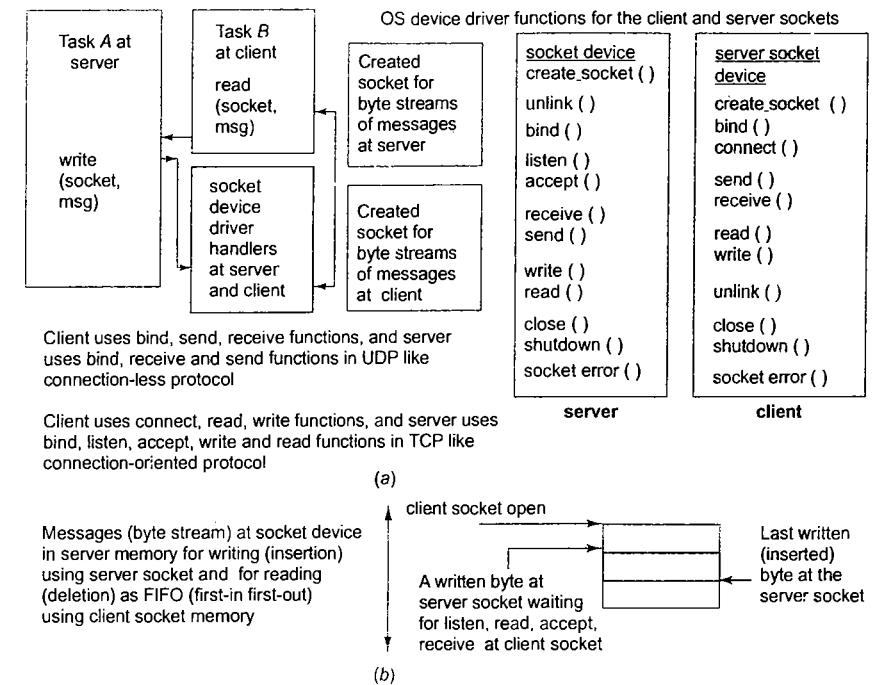


Fig. 7.11 (a) Initialized virtual sockets between the client set of tasks and a server set of tasks and the operating system provisions for the socket-functions (b) Byte stream between client and server

Example 7.23

Consider orchestra-playing robots (Example 1.5.7).

1. A process, Task_Master in the director robot creates a socket using a statement as follows:
`sfd1 = socket ("/socket/serversocket1", playStream, 0).`
 The `sfd1` is an unsigned integer for a socket descriptor. `"/socket/serversocket1"` is the path and file from which the stream, `playStream` will be sent or received from play robots. `0` represents unrestricted permission to use the file.
2. Task_Master socket binds the `sfd1` and data structure at the socket address, `sockAddr` by using the function as follows:
`bind (sfd1, (struct sockaddr *)&local, lBytes).`
 The `lBytes` is length of the bytes in the play stream;
3. Task_Master socket listens to eight orchestra-playing robots by using function as follows:
`listen (sfd1, 8)`
4. Task_Master socket accepts bytes (from a playing robot process socket) from the socket with descriptor `sfd2` using function as follows: `sfd2 = accept (sfd1, &playRobotSockAddress, &playRlBytes)`

The `&playRIBytes` refers to address for the maximum length of bytes from the playing robot. The `playRobotSockAddr` is the address of the data structure for the client (playing robot) socket.

5. **Task_Master socket** sends the bytes by using the function as follows:
`send(sfd2, &playBuffer, playstreamlen, 0); /* Send total playstreamlen bytes from playBuffer using the socket sfd2.`
6. **Task_Master socket** receives the bytes from the playing robots (e.g., for acknowledgement) by using the function as follows:
`while (streamlength > 0 && streamlength <= playRIBytes) { streamlength = recv (sfd2, &ackBuffer, 200, 0); /* The recv () returns -1 if no more bytes are to be received. Bytes are received in ackBuffer */`
7. **Task_Master socket** closes the socket by using a function as follows:
`close ();`
8. A process, **Task_Client** in the playing robot creates a client 1 socket using a statement as follows:
`sfd2 = socket ("socket/clientsocket1", sockStream, 0).`
 The `sfd2` is an unsigned integer for a socket descriptor. 'socket/serversocket1' is the path and file from which `stream, socStream` will be sent or received from play robots. 0 represents unrestricted permission to use the file.
9. **Task_Client socket** connects the `sfd1` and data structure at the socket address, `sockClientAddr` by using the function at the server process as follows:
`connect (sfd2, (struct sockClientaddr *)&remote, CIIBytes).`
 The `CIIBytes` is maximum length of the bytes in the play stream.
10. **Task_Client socket** sends the acknowledgement bytes by using the function as follows:
`send (sfd2, &ackBuffer, playRIBytes, 0); /* Send total ackstreamlen bytes from ackBuffer using the socket sfd.`
11. **Task_Client socket** receives the bytes from the playing robots (e.g., for acknowledgement) by using the function as follows:
`while (streamlength > 0 && streamlength <= playstreamlen) { streamlength = recv (sfd2, &playBuffer, 200, 0); /* The recv () returns -1 if no more bytes are to be received. Bytes are received in playBuffer */`

Application of sockets are as follows:

1. An *application* of sockets is to connect the tasks in the distributed environment of embedded systems. For example, a network interconnection or a card process connects a host-machine process.
2. A TCP/IP socket is another common *application* for the Internet. Another exemplary *application* of socket is a task receiving a byte stream of TCP/IP protocol at a mobile internet connection.
3. An exemplary *application* is when a task writes into a file at a computer or at a network using NFS protocol (network file system protocol).
4. Another *application* of the socket is the interconnection of a task or a section in a source set of tasks in an embedded system with another task at a destined set of tasks. The kernel has the socket-connecting functions with the codes specifying the source and destination sets and tasks.

The OS functions for socket in Unix are as following.

1. The `socket ()` [in place of `open ()` in case of pipe] gives a socket descriptor `sfd`. The `socket ()` enables its use from beginning of its allocated buffer at the socket address, its use with option and restrictions or permissions defined at the time of opening. A socket can be a stream, `SOCK_STREAM` or UDP datagram `SOCK_DGRAM`.
2. The `unlink ()` before the `bind ()`.

3. The `bind ()` for binding a thread or task inserting bytes into the socket to the thread or task and deleting bytes from the socket. `bind ()` the socket descriptor to an address in the Unix domain. `bind (sfd, (struct sockaddr *)&local, len);` where `len` is string length. `sockaddr` is a data structure with a record of 16-bit unsigned `num` and a path for the file and a data structure `struct sockaddr_un {unsigned short num; char path[108]; }`
 4. The `listen (sfd, 16)` function for listening 16 queued connections from the client socket.
 5. The `accept ()` accepts the client connection and gives a second socket descriptor.
 6. The `recv ()` function for deleting (reading) and receiving from the socket from the bottom of unread memory spaces in buffer. The buffer has messages after writing into the socket.
 7. The `send ()` function for inserting (writing) and sending from the socket from the bottom of the memory spaces in the buffer filled after writing into the socket.
 8. The `close ()` for closing the device to enable its use from beginning of its allocated buffer only after opening it again.
1. A Socket is an IPC for sending a byte stream or datagram from one or multiple task sockets to another task or server process socket as a bi-direction FIFO-like device using a protocol for transferring the bytes. Datagram provide protocol-header bytes along with the byte stream.
 2. The sockets can be a client-server set of sockets (multiple processes and single server process) or peer-to-peer sockets IPC. A socket has a number of applications. An Internet connection socket is for virtual connection between two ports: one port at an IP address to another port at another IP address.
 3. An OS provides the IPC functions for creating socket, unlinking, binding, listening, accepting, receiving, sending and closing.

7.16 RPC FUNCTIONS

RPC is a method used for connecting two remotely placed methods by first using a protocol for connecting the processes. It is used in the cases of distributed tasks.

The OS can provide for the use of RPCs. These permit distributed environment for the embedded systems. The RPC provides the IPC when a task is at system 1 and another task is at system 2. Both systems work in the peer-to-peer communication mode. Each system in peer-to-peer can make RPCs. The OS IPC function allows a function or method to run at another address space of shared network or an other remote computer. The process 1 makes the call to the function that is local or remote and the process 2 response is either remote or local in the process 1 to process 2 method call.



Summary

- A process is a computational unit that processes on a CPU under the state-control of a kernel in OS.
- A process may consist of multiple threads that define thread as a minimum unit for a scheduler to schedule it to run the CPU and provide other system resources. Unix provides for processes and their threads as light-weight processes. Light weight mean functions not dependent on functions like memory-management functions. Java use the threads.
- A single CPU system runs one process (or one thread of a process) at a time. A scheduler is a must to schedule a multitasking or multithreading system.

- A task is a computational unit or set of codes, actions or functions that processes on a CPU under the state-control of a kernel in OS.
- A task is similar to a process or thread. Each task is an independent process that takes control of the CPU when scheduled by a scheduler at an OS. No task can call another task. Each task and its state is recognized by its TCB (memory block) that holds information of the PC, memory map, the signal (message) dispatch table, signal mask, task ID, CPU state (registers and so on) and a kernel stack (for executing system calls and so on). A task is in one of the four states: idle, ready, blocked and running, that are controlled by the scheduler.
- Often the same data is used in two different tasks (or processes) and if another task interrupts before without completing the operation on that data, then the shared data problem arises. Disabling of interrupts till the completion of the operation by the first task, and then re-enabling interrupts, is one solution. Use of *semaphores* (as *tokens*, *mutex* or *counting semaphores*) is another efficient way for solving the shared data problem and running critical section codes. Use of lock functions and spin-locks are also provided in the OSes.
- A buffer is a memory block for a queue or stream of bytes between an output source and input sink. [For example, between the tasks, files, computer and printer, physical devices and network.] It has to be bounded between two limits. It cannot be unlimited or infinite. For example, a print buffer cannot accept unlimited output from a computer. The bounded buffer problem is of synchronizing the source and sink. A *producer cannot keep on producing beyond a limit if consumers do not consume. The consumers cannot keep consuming unless the producer keeps producing.* Counting semaphores provide solution to this problem.
- There has been POSIX IEEE standardization of the OS and IPC functions, for example, the P and V semaphore POSIX functions. These function are event notifying, resource key, mutex and counting semaphores.
- *The priority inversion* problem and deadlock situation can arise in certain situations when using a semaphore. An OS should be such that it can take care of it by having the appropriate provisions to avoid these situations. Certain OSes provide mutex semaphores such that priority inversion problem does not arise.
- The OS functions handle IPCs between the multiple tasks.
- The OS provides for the following IPCs: signals, semaphores, queues, mailboxes, pipes, sockets and RPCs.
- A mailbox may either provide for only one message or multiple messages in an RTOS.
- A pipe is a queue or stream of messages that connects the two tasks and that uses the functions as used in a device.
- The sockets are used in networks or client-server-like communication between the tasks using functions as used for the devices. The RPCs are used for the case of distributed tasks.



Keywords and their Definitions

- Buffer** : A memory block for a queue or network stack or pipe or stream of bytes between an output source and input sink, for example, between the tasks, files, computer and printer, physical devices and network.
- Counting semaphore** : A semaphore in which the value of which can be initialized to an 8- or 16- or 32-bit integer and that is decremented and incremented. A task does not block if its value is found to be >0 and the task blocks if its value is found to be 0.
- Critical section** : A section in a task the execution of which should block execution of another such section in another task, for example, when a buffer in printer is shared between two or more tasks.

- Deadlock situation** : A task waiting for the release of a semaphore from a task and another a different task waiting for another semaphore release to run. None of these is able to proceed further due to circular dependency. An OS can take care of this by appropriate provisions.
- Interprocess communication** : A mechanism from one task (or process) sending signal or messages or event notification from one task to the system and which the OS communicates to another task. Using IPC mechanism and functions a task uses signals, exceptions, semaphores, queues, mailboxes, pipes, sockets and RPCs.
- Mailbox** : A message(s) from a task that is addressed for another task.
- Message queue** : A task sending the multiple messages into a queue for use by another task(s) using queue messages as an input.
- Mutex** : The special variable and mechanism used to take note of certain actions to prevent any task or process from proceeding further and at the same time let another task exclusively proceed further. Mutex helps in mutual exclusion of one task with respect to another by a scheduler in the multitasking operations.
- P and V semaphores** : The semaphore functions defined in a POSIX IEEE standard to be used as event notification or mutex or counting semaphore, or to solve the classical producer-consumer problem when using a bounded buffer.
- Pipe** : A device for use by the task for sending the messages and another task using the device receives the messages as stream. A pipe is a unidirectional device.
- Priority inversion** : A problem in which a low priority task inadvertently does not release the process for a higher priority task. An operating system can take care of this by appropriate provisions.
- Process** : A code that has its independent PC values and an independent stack. A single CPU system runs one process (or one thread of a process) at a time. A *process* is a concept (abstraction). It defines a *sequentially executing (running) program and its state.* A *state*, during the running of a process, is represented by its status (running, blocked or finished), its control block, called process control block (PCB) or *process structure*, its data, objects and resources.
- Remote procedure call** : A method used for connecting two remotely placed methods by first using a protocol for connecting the processes. It is used in the cases of distributed tasks.
- Semaphore** : A special variable operated by the OS functions which are used to take note of certain actions to prevent another task or process from proceeding further.
- Shared data problem** : If a variable is used in two different processes (tasks) and if another task interrupts before the operation on that data is completed, then the shared data problem arises.
- Signal** : A function to call a signal handler by interrupting the processes. It uses INT n SWI n instruction, where n defines the handler, which should run.
- Socket** : It provides the logical link using a protocol between the tasks in a client-server or peer-to-peer environment. It enables a bi-directional stream or datagram or network stack.
- Synchronization** : To let each section of codes, tasks and ISRs run and gain access to the CPU one after the other sequentially or concurrently, following a scheduling strategy, so that there is a predictable operation at any instance.

- Task** : A task is for the service of specific actions and may also correspond to the codes, which execute for an interrupt. A task is an independent process that takes control of the CPU when scheduled by a scheduler at an OS. Every task has a TCB.
- Task control block** : A memory block that holds information of the PC, memory map, the signal (message) dispatch table, signal mask, task ID, CPU state (registers and so on), and a kernel stack (for executing system calls and so on).
- Task state** : A state of a task that changes on scheduler directions. A task at an instance can be in one of the four states, *idle*, *ready*, *blocked* and *running* that are controlled by the scheduler.
- Thread** : A minimum unit for a scheduler to schedule the CPU and other system resources. A process may consist of multiple threads. A thread has an independent process control block like a TCB and a thread executes codes under the control of a scheduler. It is a light weight process.



Review Questions

- How does a data output generated by a process transfer to another using an IPC?
- What are the parameters at a TCB of a task? Why should each task have distinct TCB?
- What are the states of a task? Which is the entity controlling (scheduling) the transitions from one state to another in a task?
- Define critical section of a task. What are the ways by which the critical section run by blocking other processes?
- How is data (shared variables) shielded in a critical section of a process before being operated and changed by another higher priority process that starts execution before the process finishes?
- How does use of a counting semaphore differ from a mutex? How is a counting semaphore used?
- Give an example of a deadlock situation during multiprocessing (multitasking) execution.
- What is the advantage and disadvantage of disabling interrupts during the running of a critical section of a process?
- Explain the term multitasking OS and multitasking scheduler.
- Each process or task has an endless (infinite) loop in a pre-emptive scheduler. How does the control of resources transfer from one task to another?
- What is an *exception* and how is an error-handling task executed on throwing the exception?
- How do functions differ from ISRs, tasks, threads and processes? Why is an ISR not permitted to use the IPC pend wait functions.
- List the features of P and V semaphores and how these are used as a resource key, as a counting semaphore and as a mutex.
- What are the situations, which lead to priority inversion problems? How does an OS solve this problem by a priority inheritance mechanism?
- What is meant by a pipe? How does a pipe may differ from a queue?
- What is meant by a spinning lock? Explain the situation in which the use of the spin lock mechanism would be highly useful to lock the transfer of control to an higher priority task?
- What is a mailbox? How does a mailbox pass a message during an IPC?
- When are the sockets used for IPCs? List four examples. When are RPCs used? List two examples.
- What are the analogies between process, task and thread? Also list the differences between the process, task and thread.



Practice Exercises

- Design a table to clearly distinguish the cases when there is concurrent processing of processes, when tasks and when threads by using a scheduler.
- Make a table similar to Table 7.1 to clearly distinguish ISRs, ISTs and tasks.
- What is the advantage of using a *signal* as an IPC? List the situations which warrant use of signals.
- List five exemplary applications of solutions to the bounded buffer problem using P and V mutex semaphores.
- Every tenth second a burst of 64 kB arrives at 512 kbps in an interval of 100 seconds. Is an input buffer required? If yes, then how much? If yes, then write a program to use the buffer using P and V semaphores.
- Use Web search to understand an IEEE-accepted standard POSIX 1000.3b in detail.
- Can different IPCs be used? Given the choice, how will you select an IPC from signal, semaphore, queue or mailbox?
- List the tasks in the automatic chocolate-vending machine (Example 1.10.2). List the IPC functions required and their uses in the ACVM.
- List the processes used in smart card (Example 1.10.3). How does the card communicate with the host using the sockets? List the IPC functions required and their uses in the smart card.
- List the tasks in the digital camera (Example 1.10.4). List the IPC functions required and their uses in the camera.
- List the processes in the smart mobile phone (Example 1.10.5). The display process has multiple threads in the phone. List the threads. List the IPC functions required and their uses in the phone.
- List the processes in the PDA (Example 1.10.6). Assume that PDA services the events by the ISRs and signal handlers using a queue of the events. How can this be done? Show it by a diagram.
- List the processes in the director of OPRs (Example 1.10.7). List the processes in eight playing robots in OPRs.

Real-Time Operating Systems

8

R

Following points have been discussed in the previous chapter.

e

- Application program has functions, ISRs, threads, processes (tasks) multiple physical as well as virtual device drivers and several program objects, which concurrently process on single or multiple processors.

c

- OS functions provide a mechanism to create multiple tasks (processes and threads), control the task-states and allocate system-resources to the tasks.

a

- OS functions control the context-switching in multiprocessing (multitasking and multi-threading) program.

[

- The OS functions provides the IPC functions to enable communication of the signals, semaphores and messages from the ISRs and tasks to other waiting service-thread or tasks.

[

- The OS functions also provides the IPC functions for the pipes, sockets and RPCs.

- The OS also provides for mutex, lock and spin-lock functions and for disabling of interrupts to let a critical section of code run without preemption by other process(es).

L

We will learn the following in this chapter:

E

1. OS services.

A

2. Program structure, in-between layers of OS and interfaces between the top application software and down system-hardware layers.

R

3. OS functions for the process (task), timer, event, memory, device, file-system and I/O-subsystem management.

N

4. RTOS providing in addition to the basic OS services, a control on the context switching between the tasks such that the system satisfies the real-time requirements, time constraints and deadlines of tasks.

I

5. ISR-handling in an RTOS environment.

N

6. Basic design principles when using an RTOS.

G

7. Soft and hard real-time scheduling considerations.

8. RTOS task-scheduling service models and basic strategies for scheduling the multiple tasks—cooperative, cyclic, time slicing round robin and preemptive scheduling RTOS, and critical section handling in priority scheduling cases.

9. OS security issues.

Chapters 9 to 12 will describe with exemplary RTOSes and case studies.

O

8.1 OS SERVICES

B

8.1.1 Goal

J

The OS goals are *perfection and correctness* to achieve the following:

E

1. *Facilitating easy sharing of resources as per schedule and allocations.* Resources mean processor(s), memory, I/Os, devices, virtual devices (e.g., pipes, sockets), system timer, keyboard, displays, printer and other such resources, which processes (tasks or threads) request from the OS. No processing task or thread uses any resource until it has been allocated by the OS at a given instance.

C

2. *Facilitating easy implementation* of the application-program with the given system-hardware. An application programmer for a system can use the OS functions that are provided in given OS without having to write the codes for the services (functions) that follow.

T

I

3. *Optimally scheduling* the processes on one (or more CPUs if available) and providing an appropriate context-switching mechanism.

V

E

4. *Maximizing the system performance* to let different processes (tasks or threads) share the resources most efficiently with protection and without any security breach. Examples of security breach are tasks obtaining illegal access to other task-data directly without system calls, overflow of the stacks into memory and overlaying of PCBs (Section 7.1) at the memory.

5. *Providing management functions* for the processes (tasks or threads), memory, devices and I/Os and other functions.
6. *Providing management and organization functions for the devices, files and virtual devices and I/Os.*
7. *Providing easy interfacing* and management functions for the network protocols and networking.
8. *Providing portability* of application on different hardware configurations.
9. *Providing interoperability* of application on different networks.
10. *Providing a common set of interfaces* that integrates various devices and applications through the standard and open systems.

The OS goals are perfection, correctness, portability, interoperability and providing a common set of interfaces for the system, and orderly access and control when managing the processes.

8.1.2 User and Supervisory Mode Structure

When using an OS, the processor in the system runs in two modes. There is clock, called system clock. At every tick of the clock, there is an interrupt. On interrupt, the system time updates, the system context switches to the supervisory mode from the user mode. After completing the supervisory (kernel-space) functions, the system context switches back to the user mode.

1. *User mode:* The user process is permitted to run and use only a subset of functions and instructions in OS. This is done in the user mode either by sending a message to a waiting process associated with the OS kernel or by initiating a system call (call by an OS function). The use of hardware resources including memory is not permitted without making the call to the OS functions. The OS calls the resources by system call. User function call is distinct from a system call, and is not permitted to read and write into the protected memory allotted to the OS functions, data, stack and heap. This protected memory space is also called kernel space. Hence execution of user functions calls is slower than the execution of the OS functions (which run on system call). This is because of the protected access to memory by the functions running in user-space.
2. *Supervisory mode:* The OS runs the privileged functions and instructions in the protected mode and the OS (more specifically, the kernel) only accesses the hardware resources and the protected area memory. [The term kernel means nucleus.] In the supervisory mode the kernel codes run in protected mode. Only a system-call is permitted to read and write into the protected memory allotted to the OS functions, data, stack and heap. The kernel space functions execute faster than the user-space functions.

Example 8.1

RTOS Windows CE and several RTOSes enables running of all the application-program threads in the supervisory mode (kernel mode). Therefore, the threads execute fast. This improves the system performance. If the threads are to execute in the user mode, as in Unix or in non-real-time OS, then the execution slows down due to checks on the code access to the protected kernel space.

8.1.3 Structure

A system can be assumed to have a structure as per Table 8.1.

Table 8.1 Layered Model of the System

Layer from Top	Top-down Structure Layers	Actions
1	Application software	Executes as per the applications, run on the given system hardware using the interfaces and the system software.
2	Application program interface (API)	Provides the interface (for inputs and outputs) between the application software and system software so that it is able to run on the processor using the given system software.
3	System software other than the one provided at the OS (operating system)	The OS may not have the functions, for example, for the specific network and certain device drivers, such as a multimedia device. This layer gives the system software services other than those provided by the OS service functions.
4	OS interface	Interface (for inputs and outputs) between the above and OS
5	OS	Kernel supervisory mode services (Table 8.2), file management and other functions such as the user-mode processing services.
6	Hardware-OS interface	Interfaces to let the functions be executed on the given hardware (processor, memory, ports and devices) of the system.
7	Hardware	Processor(s), memories, buses, interfacing circuits, ports, physical devices, timers and buses for devices networking.

The OS structure consists of kernel and other service functions. The OS enables an *application* run on the system-hardware.

8.1.4 Kernel

The OS is the middle layer between the application software and system hardware. An OS includes some or all of the following structural units.

1. Kernel with file management and device management as part of the kernel in the given OS.
2. Kernel without file management and device management as part of the kernel in the given OS and any other needed functions not provided for at the kernel.

The kernel is the basic structural unit of any OS in which the memory space of the functions, data and stack are protected from access by any call other than the system-call. It can be defined as a secured unit of an OS that operates in the supervisory mode while the remaining part and the application software operates in the user mode. Table 8.2 gives the functions (services) in the kernel, they are as per the OS design.

The kernel has management functions for processes, resources, ISRs, ISTs, files, device drivers and IO subsystems and network subsystems. The memory or device and file management functions may be outside the kernel in a given OS, especially in an embedded system.

Table 8.2 Kernel Services in an operating system (OS)

Function	Actions
Process ¹ management: creation to deletion	Enables process creation, activation, running, blocking, resumption, deactivation and deletion and maintains process structure at a PCB (process control block) (Section 7.1.1)
Process management: process structure maintenance	Enables process structure maintenance and its information at PCB
Process management: processing resource requests	Processing resource requests by processes made either by making calls that are known as system calls or by sending message(s)
Processes management: scheduling	Processes <i>scheduling</i> . For example, in the cyclic scheduling or priority scheduling mode (Section 8.10)
Process management: interprocess communication (IPC) (communication between tasks, ISRs, OS functions)	Processes synchronizing by sending data as messages from one task to another. The OS effectively manages shared memory access by using the IPC signals, exception (error) handling signals, semaphores, queues, mailboxes, pipes and sockets (Section 7.9)
Services memory management allocation and de-allocation ²	Memory allocation, de-allocation and management. It also restricts the memory access region for a task (Section 8.5)
File management ³	File management provides management of the creation, deletion, read (), write () to the files on the secondary memory disk (Section 8.6). A file in the embedded system (disk-free system) can be in RAM, where the operations are done in RAM memory in a way identical to the file on disk
Device management ⁴	A physical device management is such that it is accessible to one task or process only at an instant. Device manager components are: (i) device drivers and device ISRs (device interrupt handlers); (ii) resource managers for the devices (Section 8.6). Besides physical devices, the management of <i>virtual device</i> like pipe or socket is also provided (Sections 7.14 and 7.15). Virtual devices emulate a hardware device and the virtual device driver send signals (Section 7.10) similar to the ISR calls by the physical device
Device drivers	Facilitate the use of number of physical devices like keyboard, display systems, disk, parallel port, network interface cards, network devices and virtual devices (Section 8.6)
I/O management	Character or block I/Os management. For example, to ensure actions such that a parallel port or serial port is accessible to only one task at a time (Section 8.6)
Interrupts control mechanism (for handling ISRs)	Facilitate running of the ISRs and ISTs (Section 8.7)

¹ When considering the processes controlled by an OS, a process also means task in multitasking OS, and thread in multithreading OS (Refer to Sections 7.1 to 7.3).

^{2,3,4} Memory, file and device management functions form the part of kernel in a given OS. However, these functions may be outside the kernel in a given OS, especially in an embedded system using only the microkernel of an OS. This exclusion makes the kernel code small.

8.2 PROCESS MANAGEMENT

8.2.1 Process Creation

At reset of processor in a computer system, an OS is initialized first, and then a process, which can be called initial process, is created (initialization of OS means enabling the use of OS functions, which includes the function to create the processes). Then the OS is started and that runs an initial process (starting the OS means calling the OS functions, which includes the call to all the created processes before the OS start but after the OS initialization). Processes can be created hierarchically. The initial process creates subsequent processes. OS schedules the processes and provides for context switching between the processes and threads (Sections 7.1 to 7.3).

Example 8.2

(a) Recall Example 7.4. It showed an RTOS function to create a process `Task_Send_Card_Info` by using `OS_Task_Create ()` function in the `main`. `Task_Send_Card_Info` task creates two other tasks, `Task_Send_Port_Output` and `Task_Read_Port_Input`. The OS then controls the context switching between the tasks.

(b) Consider Example 7.2 for a mobile phone device (Section 1.10.5).

An OS function first creates the `Display_process`. The display process then creates the following threads:

1. `Display_Time_DateThread`
2. `Display_BatteryThread`
3. `Display_SignalThread`
4. `Display_ProfileThread`
5. `Display_MessageThread`
6. `Display_Call_StatusThread`
7. `Display_MenuThread`.

Creation of a process means specifying the resources for the process and address spaces (memory blocks) for the created process, stack, data and heap, and placing the process initial information at a PCB. The process manager allocates a PCB (or TCB in case task represents a process) when it creates the process and later manages it. The other OS units can send the manager the queries for the process when required. PCB is a *process descriptor* used by the process manager.

A PCB or TCB (Sections 7.1 and 7.3) describes the following.

1. *Context*: Processor status word, PC, SP and other CPU registers at the instant of the last instruction run executed when the process was left and the processor switched to another process.
2. *Process stack pointer*.
3. *Current state*: Is it created, activated or spawned? Is it running? Is it blocked? (spawn means create and activate)
4. *Addresses* that are allocated and that are presently in use.
5. *Pointer for the parent process* in case there exists a hierarchy of the processes.
6. *Pointer to a list of daughter processes* (processes lower in the hierarchy).
7. *Pointer to a list of resources, which are usable (consumed) only once*. For example, input data, memory buffer or pipe, mailbox message, semaphore (there may be producers and consumers of these resources).
8. *Pointer to a list of resource type usable more than once*: A resource type example is a memory block. Another example is an IO port. Each resource type will have a count of these types. For example, the number of memory blocks or number of IO ports.

9. *Pointer to queue of messages.* It is considered as a special case of resources that are usable once. It is because messages from the OS also queue up to be controlled by a process.
10. *Pointer to access permissions descriptor* for sharing a set of resources globally, and with another process.
11. *ID* by which identification is made by the process manager.

8.2.2 Management of the Created Processes

Recall process, thread and task definitions in Sections 7.1 to 7.3. A process (or thread or task) is considered a unit in which sequential running is feasible only under the control of an OS, with each process having an independent control block (descriptor of the process at an instant). (Recall the PCB and TCB described in Sections 7.1 and 7.3.)

Process manager is a unit of the OS that is the entity responsible for controlling a process execution. Process management enables process *creation, activation, running, blocking, resumption, deactivation* and *deletion*. A process manager facilitates the following. Each process of a multiple process (or multitasking or multithreading) system is executed such that a process state can switch from one to another. A process does the following sequential execution of the states: 'created', 'ready or activate', 'spawn' (means create and activate), 'running', 'blocked' or 'suspended', 'resumed' and 'finished' and 'ready' after 'finish' (when there is an infinite loop in a process) and finally 'deleted'. Blocking and resuming can take place several times in a long process. The different OSes make the provisions for possible states between creation and deletion differently.

The process manager executes a process request for a resource or OS service and then grants that request to let the processes share the resources. For example, an LCD display is a shared resource. The LCD display can be used only by one task or thread at an instance. A running process requests by two methods, which are listed in Table 8.3.

The process manager (i) makes it feasible for a process to sequentially (or concurrently) execute or block when needing a resource and to resume when it becomes available; (ii) implements the logical link to the resource manager for resources management (including scheduling of processes on the CPU); (iii) allows specific resources sharing between specified processes only; (iv) allocates the resources as per the resource allocation mechanism of the system and (v) manages the processes and resources of the given system.

A process manager creates the processes, allocates to each a PCB, manages access to resources and facilitates switching from one process state to another. The PCB defines the process structure for a process-state.

8.3 TIMER FUNCTIONS

A real-time clock (hardware timer timeout) in the system interrupts the system with each tick, which occurs a number of times in 1 second. An interrupt on a tick can be called SysClkIntr (system real-time clock timer interrupt). An OS provides a number of OS timer functions. *These functions use SysClkIntr interrupts on the clock ticks.*

The periodic SysClkIntr interrupts on this tick is used by the system to switch to the supervisory mode from the user mode on every tick. The following are the steps.

1. Before servicing of SysClkIntr, the context of presently running task or thread saves on the TCB data structure.
2. SysClkIntr service routine calls the OS.
3. The OS finds the new messages or IPCs (Sections 7.9 to 7.15), which are received from the system call by the OS event control blocks for IPC functions.
4. The OS either selects the same task or selects a new task or thread [by preemption (Section 8.10.3) in case of preemptive scheduling] and switches the context to the new one.
5. After return from the interrupt, the new task runs from the code, which was blocked from running earlier.

Table 8.3 Request for a Resource or Operating System (OS) Service by a Running Process

Request Method	Explanation
Message(s)	A process running on user mode generates and puts (sends) a message so that the OS lets the requested resource (e.g., input from a device or from a queue) use or run an OS service function (e.g., to define a delay period after which the process needs to be run again). A message can be sent for the OS to let the LCD display be used by a task or thread for sending the output. An ISR sends a message to a waiting thread to start on return from the ISR (Section 8.7).
System call	A call to a function defined at the OS. For example, OSTaskCreate () is a system call to create a task. First an SWI instruction is issued to trap the processor and switch it to supervisory mode. The OS then executes a function like a library function. On finishing the instructions of the called function, the processor switches back from the supervisory mode to the user mode and lets the calling process run further.

Each OS has a function for defining the OS ticks per second, which defines generation of the SysClkIntr interrupts and which in turn provides the timer functions of the OS. The function thus defines the SysClkIntr interrupt intervals after initiating the ticks. The functions thus also define the period after which the system calls the ISR on the SysClkIntr interrupts and switches to the OS supervisory mode functions listed before.

Example 8.3

- (a) # define OS_TICK_PER_SEC 100 /*µCOS-II function to define the number of ticks per second = 100 before the beginning of main () and initiating of OS by OSInit () function*/.
- (b) OSTickInit () /*µCOS-II function to initiate the defined number of ticks per second after the beginning of the first task and creation of all the tasks to which the context will be switched by the OS on the tick. It initiates SysClkIntr interrupts every 10 ms as OS ticks/s = 100 */.

The number of ticks if made large, then the frequent running of the OS codes to be run on SysClkIntr interrupt takes place because the context switching to the supervisory mode takes place too frequently. The number of ticks, if made small, the total time spent on SysClkIntr interrupts per second reduces, but then system time accuracy for the OS timer functions becomes small.

Example 8.4

Let OS ticks period be set to 10 ms. Assume that time spent in servicing SysClkIntr interrupt is 10 µs. $10 \mu\text{s} \times 100 \div 10 \text{ ms} = 0.1\%$ of time is spent in SysClkIntr interrupt-initiated ISRs. Let the OS ticks period be set to 100 µs. Assume that time spent in servicing SysClkIntr interrupt is 10 µs. $10 \mu\text{s} \times 100 \div 100 \mu\text{s} = 10\%$ of time is spent in SysClkIntr interrupt-initiated ISRs though time resolution of OS time functions improve to 100 µs instead of 10 ms earlier.

Table 8.4 gives the example of RTOS timer functions and the actions on calling these functions. There are RTOS timer functions for the delay, delay resume, time set, time get and for waiting-time setting for the IPC events (e.g., semaphore, mailbox, message queue message).

Table 8.4 Exemplary Timer Functions

Function	Action on Calling the Function
OS_TICK_PER_SEC	Defines the number of system clock ticks per second, which is also the number of SysClkIntr interrupts per second.
OSTickInit ()	Initiate the clock ticks in the system and SysClkIntr interrupts
OSTimeDelay ()	Delay the task executing this function
OSTimeDelayResume ()	Resume the delayed task
OSTimeSet ()	Set the system clock tick count value. OSTimeSet (1000) sets the <i>count</i> = 1000. OSTimeSet (0) sets the <i>count</i> = 0. After each SysClkIntr and clock tick the <i>count</i> increments by 1
OSTimeGet ()	Get the system clock-tick's <i>count</i> value.
OSSemPend (semVal, twait, *semErr)	Wait for the semaphore release and semVal becoming 1 for the period as per twait. If semaphore released in this period, then take it and task proceeds further after decrementing the semVal to 0, else after twait defined period, the task code proceeds with no further wait. If twait = 100, then wait for 100 system clock ticks. The semErr points to the error.
OSMboxPend (semVal, twait, *mboxErr)	Wait for the mailbox message for the period as per twait, if mailbox is posted the message, then take it and the task proceeds further, else after twait defined period, the task code proceeds further. If twait = 100, then wait for 100 system clock-ticks. The mboxErr points to the error.

Example 8.5

(a) OSTimeDelay (n) by period equal to period of n clock ticks.
 (b) Assume that using the timer functions, OS_TICKS_PER_SEC 100 and OSTickInit (); the system clock ticks every 10 ms. Assume that ACVM (Section 1.10.2), the Task_ReadCoins for reading the coin is running. Consider that the following codes in the while loop of an AVCM task, Task_ReadAmount (Example 7.3) waits for 1 minute for the coin amount:

```

/*****
count =0; OSTimeSet (count);
while (count <= 6000) /* While loop waits for for the coins amount upto 60000 ms = 1 minute */
{count = OSTimeGet ( ); OSTimeDelay (100);
/* Code for finding the coins amount after every 100 clock ticks, which means every 1 s*/
; }
*****/

```

(c) Calling OSTimeGet (), finding the count, *count1*, running a section of codes and then calling OSTimeGet (), finding the new count, *count2* defines the interval, T spent by the system in-between the two function calls of OSTimeGet (). $T = \{(count2 - count1) \times interval$ between two clock ticks} is the interval between two calls to OSTimeGet.

8.4 EVENT FUNCTIONS

In case of IPC (Section 7.9), there is wait for only one semaphore post event or mailbox message-posting event (Sections 7.10 to 7.15). Provisioning of event functions in an OS offers an advantage that there can be wait for more than one event and the events can also be from the different tasks or ISRs.

The queue messages can be from the different tasks or ISRs. Queues can offer the same advantage that there can be wait for more than one messages. However, the OS functions for queue execute in more time than the event-functions. Some OS supports and some do not support event functions. The event-functions enable OS actions after a group of events. The OS event functions can be understood as follows:

There is an event register. It has 8 or 16 or 32 event-flags, which form the groups. Each bit of the group in the register corresponds to one event flag in a set of flags.

An event register creates using an OS function, OSEventCreate (). An event register can be divided into groups, each group assigned to different tasks. For example, a 16-bit register can be divided into four groups. Group 0 is from bit 0 to bit 3, group 1 from bit 4 to bit 7, group 2 is from bit 8 to bit 11 and group 3 is from bit 12 to bit 15. An OS function, OSEventQuery () queries an event register to find the event register existence and its contents. An event register deletes using an OS function, OSEventDelete ().

Each event sets one of the bits at the event register using the SET (eventFlag) function. Event flag in the register can be set by an ISR or task. CLEAR (eventFlag) clears the flag in the event register. An event flag can be cleared by an ISR or task.

A task can use the WAIT_ALL function for the occurrences of setting all the event flags in a group. [Wait till AND operation between all flags in the group equals to true.] The task can use WAIT_ANY function for an occurrence of setting of any of the event flags in the group. [Wait till OR operation between all flags in the group equals to true.]

8.5 MEMORY MANAGEMENT**8.5.1 Memory Allocation**

When a process is created, the memory manager allocates the memory addresses (blocks) to it by mapping the process address space (Section 7.1). Threads of a process share the memory space of the process (Section 7.2).

8.5.2 Memory Management after Initial Allocation

The memory manager of the OS has to be secure, robust and well protected. There must be control such that there are no memory leaks and stack overflows. Memory leaks mean attempts to write in the memory block not allocated to a process or data structure. Stack overflow means that the stack exceeds the allocated memory block(s) when there is no provision for additional stack space. Table 8.5 gives memory-management strategy.

Example 8.6

An OS provides for dynamic memory allocation and de-allocation functions. Dynamic memory allocation is used for creating memory address space for a buffer or a messages queue or some other purpose during execution of a task. Dynamic memory de-allocation is used for freeing the memory taken up for the buffer during execution of the task.

Consider fragmented physical memory allocations. Fragmented means that memory addresses in two variable-size blocks of a process are not continuous. When a block of memory address is allocated, the time is spent in first locating the next free memory address before allocating that to the process. A standard memory allocation scheme scans a linked list of indeterminate length to find a suitable free memory block. When one allotted block of memory is de-allocated, the time is spent in first locating the next allocated memory block before de-allocating that to the process. The time for allocation and de-allocation of the memory and blocks are variable (not deterministic) when the block sizes are variable and when the memory is fragmented. In RTOS, this leads to unpredictable task-performance (run-time).

RTOS μ COS-II provides for memory partitioning. A task must create a memory partition or several memory partitions by using function `OSMemCreate ()`. Then the task is permitted to use the partition or partitions. A partition has several memory blocks. A task gets a memory block or blocks from the partition by using function `OSMemGet ()`. A task releases a memory block or blocks to the partition by using function `OSMemPut ()`. Therefore, the task consists of several fixed size memory blocks. The fixed size memory blocks allocation and de-allocation time takes fixed time (deterministic). Therefore, it leads to a predictable task-performance.

Table 8.5 Memory Managing Strategy for a System

Managing Strategy	Explanation
Fixed blocks allocation	Memory address space is divided into blocks with processes having small address spaces getting a lesser number of blocks and processes with big address spaces getting a larger number of blocks.
Dynamic blocks allocation ¹	Memory address space is divided into fixed blocks as above and then later the memory manager later allocates variable size blocks (in units of say 64 or 256 bytes) dynamically allocated from a free (unused) list of memory blocks description table at the different computation phases of a process.
Dynamic page allocation ¹	Memory has fixed sized blocks called pages and the memory manager MMU (memory management unit) allocates the pages dynamically with a page descriptor table.
Dynamic data memory allocation	The manager allocates memory dynamically to different data structures like the nodes of a list, queues and stacks.
Dynamic address relocation ¹	The manager dynamically allocates the addresses initially bound to the relative addresses. It adds the relative address to address with relocation register. The memory manager now dynamically changes only the contents of relocation register. It also takes into account a limit-defining register so that the relocated addresses are within the limit of available addresses. This is also called run-time dynamic address binding.
Multiprocessor memory allocation	Refer to Section 6.3. The memory adopts an allocation strategy either shared with tight coupling between two or more processors or shared with loose coupling or multisegmented allocations.
Memory protection to OS functions ²	Memory protection to the OS functions means that the system call (call to an OS function) and function call in user space are distinct. The OS function code, data and stack are in the protected memory area. It means that when a user function call attempts to write or read in the exclusive memory space allocated to the OS functions, it is blocked and the system generates an error. The memory of kernel functions is distinct and can be addressed only by the systems calls. The memory space is called kernel space.
Memory protection among the tasks ²	Memory protection to the tasks means that a task function call cannot attempt to write or read in the exclusive area of memory space allocated to another task. The protection increases the memory requirement for each task and also the execution time of the code of the task.

¹ RTOS may disable the support to the dynamic block allocation, MMU support to the dynamic page allocation and dynamic binding as this increases the latency of servicing the tasks and ISRs.

² RTOS may not support memory protection of the OS functions from user function calls, as this increases the latency of servicing the tasks and ISRs. The user functions are then runnable in kernel space and run like kernel functions.

³ RTOS may provide for disabling of the support to memory protection among the tasks as this increases the memory requirement for each task.

The memory manager manages the following: (i) use of memory address space by a process, (ii) specific mechanisms to share the memory space, (iii) specific mechanisms to restrict sharing of a given memory space and (iv) optimization of the access periods of a memory by using an hierarchy of memories (caches, primary and external secondary magnetic and optical memories). Remember that the access periods are in the following increasing order: caches, primary and external secondary magnetic and then optical.

The memory manager allocates memory to the processes and manages it with appropriate protection. There may be static and dynamic allocations of memory. The manager optimizes the memory needs and memory utilization. An RTOS may disable the support to the dynamic block allocation, MMU support to the dynamic page allocation and dynamic binding as this increases the latency of servicing the tasks and ISRs. An RTOS may or may not support memory protection in order to reduce the latency and memory needs of the processes.

8.6 DEVICE, FILE AND IO SUBSYSTEMS MANAGEMENT

8.6.1 Device Management

Recall Section 4.2. There are number of device driver ISRs for each device in a system, each driver-function of a device (e.g. open, close, read) calls a separate ISR. Device manager (inside or outside the kernel space) is the software that manages these for all. When device driver functions are a part of the OS (inside or outside the kernel space), the device manager effectively operates and adopts appropriate strategy for obtaining optimal performance for the devices. The manager coordinates between application process, driver- and device-controller. A process sends a request to the driver functions by an interrupt using SWI; and the driver provides the actions on calling and executing the ISR. The device manager polls the requests at the devices and the actions occur as per their priorities. The device manager manages IO interrupt (requests) queues. The device manager creates an appropriate kernel interface and API, and that activates the control register-specific actions of the device. The device controller is activated through the API and kernel interface (recall Section 1.4.6). An OS device manager provides and executes the modules for managing the devices and their driver ISRs.

1. It manages the physical as well as virtual devices like the pipes and sockets through a common strategy.
2. Device management has three standard approaches to three types of device drivers: (i) programmed I/Os by polling the service need from each device; (ii) interrupt(s) from the device driver ISR and (iii) DMA operation used by the devices to access the memory. Most common is the use of device driver ISRs.
3. A device manager has the functions given in Table 8.6.

Table 8.6 Functions of a Device Manager

Function	Action(s)
Device detection and addition	Provides the codes for detecting the presence of various devices, then adding (initializing, configuring and testing) them for the use of OS device driver functions. A manager can provide for tracking the hardware inventory (list of devices present in the system and connected to the system).
Device deletion	Provides the codes for denying the device resources.
Device allocation and registration	Allocates and registers the port (it may be a register or memory) addresses for the various devices at distinctly different addresses and also includes codes for detecting any collision between them.

(Contd)

Function	Action(s)
Detaching and deregistration	Detaches and deregisters the port (it may be a register or memory) addresses for the various devices at distinctly different addresses and also includes codes for detecting any collision between existing addresses in case of addresses reallocation to the remaining attached (registered) devices.
Restricting device to a specific process	Restricts a device access to one process (task) only, at an instant.
Device sharing	Permits sharing of access of a device to the set of processes, but to one process (task) at an instant.
Device control	A manager can also provide for remote control of the devices from the remote server at the service provider. (For example, mobile devices with server at the service-provider)
Device access management	(i) sequential access, (ii) random access, (iii) semi-random access, (iv) serial communication may be by UART or USB, and (v) 4 (or more) serial bits in parallel during IOs (for example, SDIO) (Chapter 3). The device manager provides the necessary interface.
Device buffer management	Device hardware may merely have a single byte buffer, or double buffer or 8-byte buffer. A device buffer manager uses a memory manager to buffer the I/O data streams from the device that sends the data and manages computations without wait while the buffer receives the data at a slow rate. ¹ Also used are the multiple buffers and producer-consumer-type bounded buffers (Section 7.7.6).
Device queue, circular-queue or blocks of queue management	Device IO data streams from the device can be organized as the queues, circular queues and blocks of queues (Section 5.4.3).
Device driver	A manager manages the device drivers. A device driver or a software driver is the software for interface with the device hardware through the buses on the one hand and for interface with the OS and application on other hand. The software commands for read and write enables the read and write functions through the ISRs called by using the SWIs. ² The interface software to OS enables the creation, connection, binding, opening and closing of the device ³ (Section 4.1).
Device drivers updating and uploading of new device functions	A manager can also provide for updating the driver software from the Internet and uploading the new device functions, which become available at a later date.
Backup and restoration	A manager can also provide for the backup and restoration for drivers.

¹ For example, when the computations for deciphering the input data is slower than the receiving data in the buffer, the buffer(s) will soon choke. When the computations for deciphering the input data is faster than the receiving data in the buffer, the computations will wait for data in buffer.

² For hardware devices, a device ISR can also be called a system ISR or a system interrupt handler.

³ For example, Unix device driver components are: (i) device ISR, (ii) device initialization codes (codes for configuring device control registers) and (iii) system initialization codes, which run just after the system resets (at bootstrapping).

Table 8.7 gives the set of OS command functions for a device.

Table 8.7 Set of Command Functions for Device Management

Commands	Action(s)
create and open	<i>create</i> is for creating and <i>open</i> is for creating (if not created earlier) and configuring and initializing the device. ¹
write	Write into the device buffer or send output from the device and advance the pointer (cursor).
read	Read from the device buffer or read input from the device and advance the pointer (cursor).
ioctl ²	Specified device configured for specific functions and given specific parameters.
close and delete	<i>close</i> is for de-registering the device from the system and <i>delete</i> is for close (if not closed earlier) and detaching the device.

¹ There are two types of devices: char devices and block devices. (Refer to Table 4.2 for definitions.)

² In a system the *ioctl ()* is used for the following: (i) accessing specific partition information; (ii) defining commands and control functions of device registers; (iii) IO channel control.

The *ioctl ()* has three arguments for the device-specific parameters.

1. *First argument*: Defines the chosen device and its functions by passing as argument, the device-descriptor (a number), for example, *fd* or *sfd* in Sections 7.14 and 7.15 for a device control. Example is *fd = 1* for read device, *fd = 2* for write device.
2. *Second argument*: Defines the control option or use option for the IO device. Network devices control by defining baud rate or other parameters. Its use is as per function defined, as a second argument. Controlled device will be according to the first argument.
3. *Third argument*: Values needed by the defined function are at the third argument.

Example 8.7

Status = *ioctl (fd, FIOBAUDRATE, 19,200)* is an instruction in RTOS VxWorks. The *fd* is the device descriptor (an integer returned when the device is opened) and *FIOBAUDRATE* is a pointer for IO baud rate function that takes value of baud rate = 19,200 from third argument. This configures the device for operation at the 19,200-baud rate.

A device driver ISR uses several OS functions. Examples are as follows: *intlock ()* to disable interrupts system, *intUnlock ()* to enable interrupts, *intConnect ()* to connect a C function to an interrupt vector (the interrupt vector address for a device ISR points to a specific C function). Function *intContext ()* finds whether interrupt is called when an ISR was in execution.

UNIX OS makes it feasible for devices and files to have an analogous implementation as far as possible. A device has *open ()*, *close ()*, *read ()*, *write ()* functions analogous to a file *open*, *close*, *read* and *write* functions (Section 7.14). APIs and kernel interfaces in BSD (Berkeley sockets for devices) UNIX are *open*, *close*, *read* and *write*. The following are the in-kernel commands: (i) *select*, which is to first check whether a read or write will succeed, (ii) *ioctl* to transfer driver-specific information to the device driver. [For example, baud rate in Example 8.7.] (iii) *stop* to cancel the output activity from the device, (iv) *strategy* to permit a block *read* or *write* or character *read* or *write*.

The device manager initializes, controls and drives the physical and virtual devices of the system. The main classes of devices are char devices and block devices. Device driver functions may be similar to file functions, open, read, lseek, write and close.

8.6.2 File System Organization and Implementation

A file is a named entity on a magnetic disk, optical disk or system memory. A file contains the data, characters and texts. It may also have a mix of these. *Each OS may have differing abstractions of a file.* (i) A file may be a named entity that is a structured record as on a disk having random access in the system. (ii) A file may be a structured record on a RAM analogous to a disk and may also be either separately called *RAM disk* or simply, a 'file' (virtual device). (iii) A file may be an unstructured record of bits or bytes. (iv) A file device may be a pipe-like device.

It is necessary to organize the files in a systematic way and to have a set of command functions. Table 8.8 gives these functions for POSIX file system.

Table 8.8 Set of Command Functions in the Portable Operating System Interface (POSIX) File System

Command in POSIX	Action(s)
open	Function for creating the file
write	Writing the file
read	Reading the file
lseek (list seek) or set the file pointer	Setting the pointer for the appropriate place in the file for the next read or write
close	Closing the file

- Copy: 1. File devices are block devices in Unix. Linux permits the use of a block device as a char device also. This is because between *block device* and *char device*, Linux has an additional interface. In other words, the kernel interface is identical for the char and block devices in Linux but not in Unix.
2. The file on the RAM that is hierarchically organized is known as RAM disk. RAM memory storage is analogous to that on the disk and accessing is also analogous to a disk. For example, path for accessing a file is directory, then subdirectory, then folder and then subfolder. There is hierarchical tree like the filing organization.
3. Unix has a structured file system with an unstructured hardware interface. Linux supports different standard file systems for the system.

Should a file having integers differ from a file having bytes? Should a file having bytes differ from a file having characters? Due to the differing approaches to device and file management interfaces, the development of a set of standard interfaces becomes must. Only then can systems be portable. A standard set of interfaces is called POSIX, from IEEE. POSIX stands for portable operating system interface standard for coding programs when using the multiple threads. The X after I is because of the interfaces being similar to the ones in Unix. It is according to the definitions at the AT & T UNIX System V Interface. POSIX defines the functions: open, close, read, write, lseek and fcntl. Function lseek is to move the pointer position in the byte stream. Function fcntl is for file control. The POSIX standard for file operations are as the operations on a linear sequence of bytes.

Window NT assumes a file as the named entity for a record of bytes placed sequentially and the OS has the command functions, createFile, ReadFile, WriteFile and SetFilePointer, and CloseHandle for creating a file, reading a file, writing a file and setting the file pointer from the present to a new location.

A file in Unix has *open ()*, *close ()*, *read ()*, *write ()* functions analogous to a device, *open*, *close*, *read* and *write* functions. The BSD Unix interface differs slightly from Unix.

There are two types of file systems.

1. *Block file system*. Its application generates records to be saved into the memory. These are first structured into a suitable format and then translated into block streams. A file pointer (record) points to a block from the start to the end of the file.
2. *Byte stream file system*. Its application generates record streams. These streams are to be saved into the memory. These are first structured into a suitable format and then translated into byte streams. A file pointer (byte index) points to a byte from the start index = 0 to N-1 in a file of N bytes.

Just as each process has a processor descriptor (PCB); a file system has a data structure, called file descriptor (Table 8.9). The structure differs from one file manager to another. File descriptor, fd, for a file is an integer, which returns on opening a file. fd points to the data structure of the file. fd is usable till the closing of the file.

Table 8.9 Data Structure of File Descriptor in a Typical File System

File Descriptor	Meaning(s)
Identity	Name by which a file is identified in the application
Creator or owner	Process or program by which it was created
State	A state can be 'closed', 'archived' (saved), 'open executing file' or 'open file for additions'
Locks and protection fields	O_RDWR file opens with read and write permissions. O_RDONLY file opens with read only permissions. O_WRONLY file opens with write only permissions
File info	Current length, when created, when last modified, when last accessed
Sharing permission	Can be shared for execution, reading or writing
Count	Number of directories referring to it
Storing media details	Blocks transferable per access

A file manager creates, opens, reads, seeks a record, writes and closes a file. A file has a file descriptor.

8.6.3 I/O Subsystems

I/O ports are the subsystems of OS device management systems. Drivers communicate with the many devices that use them. I/O instructions depend on the hardware platform. I/O systems differ in different OSes. Subsystems of a typical IO system are as given in Table 8.10.

There are two types of IO operations—synchronous and asynchronous. There may be separate functions for synchronous and asynchronous operations in an RTOS. In case of traditional OS, only synchronous IOs may be supported.

Synchronous IO operations are at certain fixed data transfer rates. Therefore, a task (process) blocks till completion of the IO. For example, a write function, write () for 1 kB data transfer to a buffer. Synchronous IO operation means once synchronous IO initiates, the data transfer will block the task till 1 kB data gets transferred to the buffer. Similarly, read () once initiated blocks the task till 1 kB is read.

Table 8.10 Input/Output (I/O) Subsystem in a Typical I/O System in an Operating System (OS)

Subsystems Hierarchy	Action(s) and Layers between the Subsystems
Application	An application having an I/O system. There may also be a sublayer between the application and I/O basic functions
IO basic functions	These are device-independent OS functions, for example, file system functions for read and write, buffered IO or file (block) read and write functions. There may also be a sublayer between the basic I/O functions and I/O device-driver functions
IO device driver functions	These are device-dependent OS functions. A driver may interface with a set of library functions, for example, for serial communication
Device hardware or port or IO interface card	Serial device or network

Asynchronous IO operations are at the variable data transfer rates. It provisions for a process of high priority not blocked during the IOs.

Example 8.8

POSIX has the following asynchronous functions: *aio_read()* and *aio_write* for the asynchronous read and write in an I/O system. Therefore, an *aio_read()* and *aio_write()* do not block the task till completion of the IO. *aio_list()* is to initiate a list of certain maximum asynchronous I/O port requests. *aio_error()*, *aio_cancel*, *aio_suspend* are functions for asynchronous IO error status retrieval and for cancelling and suspending I/O operations, respectively. Suspension is till the next port device interruption or till a timed out *aio_return* returns the status of completed operations.

I/O subsystems are an important part of OS services. Examples are the UART access and the parallel port access. There are synchronous and asynchronous IOs. A task gets blocked during the synchronous IOs, for example, *fread()* or *write()* (Section 7.14). RTOSes support asynchronous IOs, for example, *aio_read()* and *aio_write* also in order to not to block a task during the IOs.

8.7 INTERRUPT ROUTINES IN RTOS ENVIRONMENT AND HANDLING OF INTERRUPT SOURCE CALLS

In a system, the ISRs should function as following.

1. ISRs have higher priorities over the OS functions and the application tasks. An ISR does not wait for a semaphore, mailbox message or queue message (Sections 7.11 to 7.13).
2. An ISR does not also wait for mutex (Sections 7.7.3 and 7.8.3) else it has to wait for other critical section code to finish before the critical codes in the ISR can run. Only the accept function for these events can be used (row 7 Table 7.1 and Section 7.11).

There are three alternative systems for the OSes to respond to the hardware source calls from the interrupts. Figure 8.1(a-c) show the three systems. The following sections explain the three alternative systems in three OSes for responding to a hardware source call on interrupts.

8.7.1 Direct Call to an ISR by an Interrupting Source and ISR Sending an ISR Enter Message

Figure 8.1(a) shows the steps. On an interrupt, the process running at the CPU is interrupted and the ISR corresponding to that source starts executing (step 1). A hardware source calls an ISR directly. The ISR just sends an ISR enter message to the OS (step 2).

OS is simply sent an ISR enter message (ISM) from the ISR in step 2. Later the ISR code can send into a mailbox or message queue (step 3) but the task waiting for the mailbox or message queue does not start before the return from the ISR (step 4). The ISR enter message in step 2 is to inform the OS that an ISR has taken control of the CPU. The ISR continues execution of the codes needed for the interrupt service till the ISR exit message is sent just before the return (step 4).

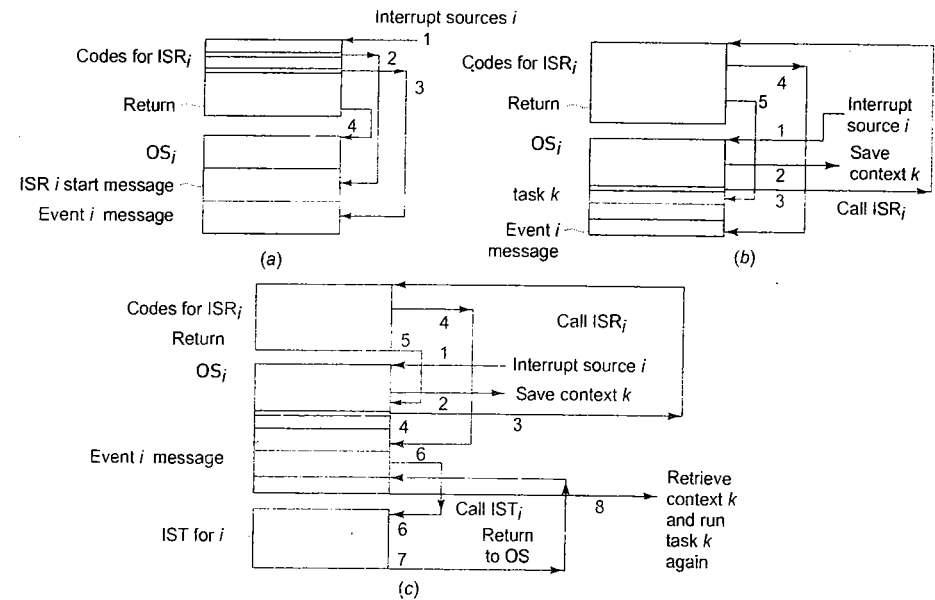


Fig. 8.1 (a) – (c) Three alternative systems in three real-time operating systems for responding to a hardware source call on interrupts

There are two functions, ISR and OS functions, in two memory blocks. An *i*-th interrupt source causes *i*-th ISR, *ISR_i*, to execute. The routine sends an ISR enter message to the OS. The message is stored at the memory allotted for OS messages. When the ISR finishes, it sends *ISR exit* to the OS and there is return and either there is the execution of interrupted process (task) or rescheduling of the processes (tasks). OS action depends on the event messages, whether the task waiting for the event is a task of higher priority than the interrupted task at the interrupt.

On certain OS, there may be a function *OSISRSemPost()*. The ISR semaphore is a special semaphore, which *OSISRSemPost()* posts and on return from the OS to be taken by the calling ISR itself. OS ensures that *OSISRSemPost* executing ISR is returned after any system call from the ISR.

Example 8.9

Consider the RTOS μ COS-II. Assume that a microcontroller has a hardware timer, which is programmed to interrupt every 10 ms. The microcontroller on timer interrupt calls and PC changes to an ISR vector address, `ISR_Timer_Addr`. At `ISR_Timer_Addr`, there is a routine `ISR_Timer` for servicing the timer interrupt. `ISR_Timer` first executes `OSIntEnter()` just after the start of `ISR_Timer` is called. `ISR_Timer` then executes `OSIntExit()` before the return code.

The `OSIntEnter()` sends the message to the RTOS that there should be context-switch and return from the ISR only after any system call is made by the ISR or until the `OSIntExit()` executes in the ISR code. Any task waiting for the post of semaphore or mailbox message or queue message should not start on execution of the post function within the ISR or in any other task or ISR. RTOS schedules that later on return from ISR.

The multiple ISRs may be nested and each ISR of low priority sends high priority ISR interrupt message (ISM) to the OS to facilitate return to it on the completion and return from the higher priority interrupt. Nesting means when an interrupt source call of higher priority, for example, system real-time clock interrupt (`SysClkIntr`) occurs, then the control is passed to higher priority `SysClkIntr` and on return from the higher priority the lower priority ISRs or tasks starts executing. The number of ISRs can be nested with execution order in sequence to their priorities. Each ISR on letting a higher priority interrupt call sends the ISM (step 4) to the RTOS.

There is common stack for the ISR nested calls, similar to the nested function calls (Table 7.1).

8.7.2 RTOS First Interrupting on an Interrupt, then OS Calling the Corresponding ISR

Figure 8.1(b) shows the steps. On interrupt of a task, say, k -th task, the OS first gets the hardware source call (step 1) and initiates the corresponding ISR after saving the present process status (or context) (step 2). The called ISR (step 3) during execution then can post one or more outputs (step 4) for the events and messages into the mailboxes or queues.

Assume that there are the routine (i -th ISR) and two processes (OS and j -th task) in three memory blocks other than the interrupted k -th task. An i -th interrupt source causes the OS to get the notice of that, then after step 1 it finishes the critical code till the pre-emption point and calls the i -th ISR. `ISRi` executes (step 3) after saving the context (step 2) onto a stack. The preemption point is the last instruction of the critical part of the presently running OS function, after which the ISR being of highest priority is called. The ISR in step 4 can post the event or mailbox message(s) to the OS for initiating the j -th task or k -th task after the return (step 5) from the ISR and after retrieving the j -th or k -th task context.

The events or mailbox messages are stored at the memory allotted for OS messages. The OS initiates the j -th task (if it is of higher priority than the interrupted task k) or runs the interrupted task k .

The ISR must be short and it must simply post the messages for another task. This task runs the remaining codes whenever it is scheduled (according to priorities). OS schedules only the tasks (processes) and switches the contexts between the tasks only. ISR executes only during a temporary suspension of a task.

OS may provide for nesting or an OS may provide for the ISRs such that the OS initiates running of the ISR calls from a priority ordered FIFO (Table 7.1).

The system priorities are ISRs and then tasks (or ISTs). IST is just a task initiated on signal or message from an ISR (for example, task j in above example).

Example 8.10

Each device event has the codes for an ISR, which executes only on scheduling by the RTOS and provided an interrupt is pending for its service. Consider mobile PDA device example (Section 1.10.6). The 5 steps for the interrupt servicing by first interrupting the RTOS process are as follows:

Assume that using RTOS, touch screen ISR, `ISR_TouchScreen` has been created using a function `OS_ISR_Create()`. The ISR can share the memory heap with other ISRs. A function, `IntConnect` connects the touch screen event with the event identifier in an interrupt handler, `ISR_handler`.

Let a touch screen event occur, which means the user of the mobile device taps the screen at a select icon or menu (step 1). After saving context of current process (step 2) the OS sends the signal on behalf of the `ISR_handler` to the initiate `ISR_TouchScreen` (step 3). An interrupt service thread or a `Task_TouchScreen` Input `IST_TouchScreen` waits using a function `OS_eventPend()` for message (an object, such as semaphore, mailbox or queue message) (step 4) from the `ISR_TouchScreen`. The `IST` executes as per its priority `Task` or `IST_TouchScreenPriority` among the other pending `ISTs` or tasks before it starts executing.

Before return from the `ISR_TouchScreen`, it sends a message to the kernel using a `OS_ISR_Exit()` just before the end of the codes in the `ISR_TouchScreen` (step 5). The `ISR_TouchScreen` can be restarted on the next interrupt event and gets ready for the next hardware event of tap on the screen.

8.7.3 RTOS First Interrupting on an Interrupt, then RTOS Initiating the ISR and then an ISR

An RTOS can provide for two levels of ISRs, a fast-level ISR, FLISR and a slow-level ISR (SLISR). The FLISR can also be called hardware interrupt ISR and the SLISR as software interrupt ISR. FLISR is called just the ISR in RTOS Windows CE. The SLISR is called interrupt service thread (IST) in Windows CE. The use of FLISR reduces the interrupt latency (waiting period) for an interrupt service and jitter (worst-case and best-case latencies difference) for an interrupt service.

An IST functions as a deferred procedure call (DPC) of the ISR. An i -th IST is a thread to service an i -th interrupt source call.

Figure 8.1(c) shows seven steps on the interrupt. On interrupt, the RTOS first gets the hardware source call (step 1) and initiates the corresponding ISR after finishing the critical section and reaching the pre-emption point and then saving the processor status (or context) (step 2). The ISR executes the device- and platform-dependent code (step 3). The ISR at the start can mask (disable) further pre-emption from the same or other hardware sources. The ISR during execution then can send one or more outputs for the events and messages into the mailboxes or queues for the `ISTs` (step 4). The `IST` executes the device- and platform-independent code. The ISR just before the end, unmask (enable) further pre-emption from the same or other hardware sources (step 5).

There are the ISRs and number of `ISTs`, RTOS and tasks in the memory blocks other than the interrupted task. Any interrupt source causes the RTOS to get the notice of that, then completes the critical code till the pre-emption point and calls the ISR. ISR executes after saving the context onto a stack. The ISR can post message(s) into the FIFO for the `IST(s)` after recognizing the interrupt source and its priority. The `ISTs` in the FIFO that have received the messages from the `ISR(s)` executes (step 6) as per their priorities on return (step 5) from the ISR. The ISR has the highest priority and pre-empts all pending `ISTs` and tasks.

When no ISR or `IST` is pending execution in the FIFO, the interrupted task runs on return (step 7).

The ISRs must be short, run critical and necessary codes only, and then they must simply send the initiate all or messages to ISTs into the FIFO. It is the IST, which runs the remaining codes as per the priority-based schedule. The system priorities are in order of ISRs, ISTs and tasks. The ISTs are SLISRs running device-independent codes as per the device priorities on signals (SWIs) from the ISRs.

The ISTs run in the kernel space. The ISTs do not lead to priority inversion and have the priority inheritance mechanism.

RTOS schedules the ISTs and tasks (processes) and switches the contexts between the ISTs and tasks.

Example 8.11

Consider Mac OS X. The Mac OS X is RTOS for the mobile device, for example, iPod. An interrupt handler first receives the primary interrupt and then it generates a software interrupt known as a secondary interrupt. The secondary software interrupt is sent to initiate an IST.

The OS does not receive the actual interrupt but the low level process intercepts the interrupt. It calls a low-level (hardware level) ISR, LISR. It resets the pending interrupt bit in the device interrupt controller and calls a device-specific ISR, say, DISR_i. The DISR_i posts a message to an IST_i specific to the device. The message notifies the IST_i that an interrupt has occurred, and then the DISR_i returns to LISR. LISR resets another pending interrupt bit in the device interrupt controller and calls the another device-specific ISR, say, DISR_j.

When no further interrupts are pending, the OS control returns to the currently executing thread, which was interrupted and when the OS passed control to the LISR.

The IST_i are scheduled by the OS, the IST_i finds that the SWI has occurred, it starts and run the codes. ISTs run as if a thread is running.

An RTOS uses one of the three strategies on interrupt source calls: (i) an ISR servicing directly after merely informing the RTOS at the start of ISR; (ii) kernel intercepting the call and calling the corresponding SRs and tasks. RTOS kernel schedules only the tasks (processes) and ISR executes only during a temporary suspension of the task by the RTOS; (iii) kernel intercepting the call and calling the ISR, which initiates and queues the ISR calls into a priority FIFO. The ISR signals the SWIs for the ISTs. The RTOS kernel schedules the ISTs as priority queue and then tasks processes as per the priority queue.

8.8 REAL-TIME OPERATING SYSTEMS

RTOS is multitasking OS for the applications needing meeting of time deadlines and functioning in real-time constraints. Real-time constraint means constraint on time interval between occurrence of an event and item-expected response to the event.

RTOS-software has the OS services listed in Table 8.11. These enable design of software for a large number of embedded systems.

An RTOS is an OS for response-time-controlled and event-controlled processes. The processes have predictable latencies. An RTOS is an OS for the systems having the real timing constraints and deadlines for the tasks, ISTs and ISRs.

Table 8.11 Real-Time Operating System (RTOS) Services

Function	Activities
Basic OS functions	Process management, resources management, device management, I/O devices subsystems and network devices and subsystems management.
Process priorities management: priority allocation	User-level priorities allocation, called static priority allocation or real-time priority allocation is permitted. The real-time priorities are higher than the dynamically allocated priorities to the OS functions and the idle priority allotted to low priority threads. The idle priority thread or task is one which runs when no other high priority ones are running.
Process management: preemption	The RTOS kernel preempts a lower priority process when a message or event for which it is waiting to run a higher priority process takes place. The RTOS kernel has the preemption points at the end of the critical code and therefore the RTOS can be preempted at those points by a real-time high priority task. Only small sections in the RTOS functions are non-preemptive.
Process priorities management: priority inheritance	Priority inheritance enables a shared resource in low priority task, for example, LCD display, be used by high priority task first. An intermediate priority task will not pre-empt the low priority task when it is locked to run the critical shared resource or code for the high priority task (Section 7.8). Priority sealing in place of priority inheritance option can also be used for a specific system.
Process predictability	A predictable timing behaviour of the system and a predictable task synchronization with minimum jitter (difference between best-case and worst-case latencies).
Memory management: protection	In RTOS threads of application program can run in kernel space. The real-time performance becomes high. However, then a thread can access the kernel codes, stack and data memory space, and this could lead to unprotected kernel code.
Memory management: MMU	Memory management is by either disabling the use of MMU and virtual memory or by using memory locks. Memory locking stops the page swapping between the physical memory and disk when MMU is disabled. This makes RTOS task latencies predictable and reduces jitter (time between worst-case and best-case latencies for a task or thread).
Memory allocation	In RTOS, the memory allocation is fast when there are fixed length memory block allocations. First, speed of allocation is important (Example 8.6).
RTOS scheduling and interrupt latency control functions	Real-time task scheduling and interrupt latency control (Section 4.6) and use of timers (Sections 3.6, 3.7 and 3.8) and system clocks.
Timer functions and time management	Provides for timer functions. There is time allocation and de-allocation to attain efficiency in given timing constraints.
Asynchronous IO functions	Permits asynchronous IOs, which means IOs without blocking a task.
IPC synchronization functions	Synchronization of tasks with IPCs (semaphores, mailboxes, message queues, pipes, sockets and RPCs).
Spin locks	Spin locks for critical section handling (Section 8.10.4).
Time slicing	Time slicing of the execution of processes which have equal priority.
Hard and soft real-time operability	Hard real-time and soft real-time operations (Section 8.9.3)

8.9 BASIC DESIGN USING AN RTOS

An embedded system with a single CPU can run only one process at an instance. The process at any instance may either be an ISR, kernel function or task (Sections 8.1.2 and 8.7). An RTOS use in embedded system facilitates the following.

1. An RTOS provides running the user threads in kernel space so that they execute fast (Example 8.1).
2. An RTOS provides effective handling of the ISRs, device drivers, ISTs, tasks or threads (Section 8.7) and the disabling and enabling of interrupts in the user mode critical section codes. A critical section means a section of codes or a resource or codes that must run without blocking. One critical situation is when there is a shared data or resource with the other routines or tasks. RTOS provides for effective handling of such a situation.
3. An RTOS provides memory allocation and de-allocation functions in fixed time and blocks of memory (Example 8.6) and restricting the memory accesses only for the stack and other critical memory blocks (Section 8.5).
4. An RTOS provides for effectively scheduling and running and blocking of the tasks in cases of many tasks (Section 8.10).
5. I/O management with devices, files, mailboxes, pipes and sockets becomes simple using an RTOS. (Sections 7.9 and 8.6) RTOS provides for the use of message queues and mailbox, pipes, sockets and other IPC functions (Sections 7.9 to 7.15). RTOS provides for the use of semaphore(s) by tasks or for the shared resources (critical sections) in a task or OS functions (Section 7.7.2).
6. Effective management of the multiple states of the CPU and internal and external physical or virtual devices. Assume that the following actions are concurrently needed in an application. (i) Physical devices timer, UART and keyboard have issued the interrupts and the service routines are to be executed. (ii) A file is taken as a virtual device. The file also must be opened with its pointer to its first record. (iii) A physical timer is to configure its control register. (iv) Another timer gets a count input from the system clock. (v) A virtual device, a file, gets the inputs for writing onto it. (vi) A timer states changes on timeout and generates a need for its service. (vii) A file states changes on transfer of all needed records to it. (viii) A timer executes a service routine on timeout. (ix) A file needs execution of a function, close (). By effectively using a common method to handle these needs, the RTOS solves all the problems.

Basic design principles in RTOS environment are as follows.

8.9.1 Principles

Following are the design principles when using an RTOS to design an embedded system.

Design with the ISRs and Tasks The embedded system hardware source call generates interrupts. On interrupt, if the interrupt is not masked (disabled) the interrupt saves the current process (a task or thread or OS function) context on a stack and executes the ISR corresponding to that interrupt. The handling of interrupt source calls is as described in Section 8.7. It is done by an RTOS by one of the three methods by a given RTOS environment. Interrupts are masked by disable interrupt command and unmasked by enable interrupt commands.

The ISR can only post (send) the messages for the RTOS and parameters for the tasks. No ISR instruction should block any task. Therefore, the ISR should not use mutex locks and should not use OS pending functions for the IPCs. Only an RTOS initiates the actions according to the ISR-posted signals, semaphores, queues, mailboxes and pipes (Section 7.9) and the RTOS control states of the tasks and interactions with the tasks. The

variables and task-switching flags must always be under the RTOS control. No ISR instruction should wait for taking the messages. The ISR should execute the codes that should not wait for actions by the RTOS and tasks.

RTOS provides for nesting of ISRs. This means that a running ISR can be interrupted by a higher priority interrupt and the higher priority ISR starts executing, blocking the running of low priority ISR. When the high priority interrupt service completes and there is return to the low priority interrupt after retrieving the saved context from the stack for the low priority interrupt.

A task can wait and take the messages (IPCs) and post (send) the messages using the system calls.

A task or ISR should not call another task or ISR. Each ISR or task has to be under the control of the RTOS. Such an attempt should generate an error.

Each ISR Design Consisting of Shorter Code As ISRs have higher priorities over the tasks, the ISR code should be made short so that the tasks do not wait longer to execute. A design principle is that the ISR code should be optimally short and the detailed computations be given to an IST or task by posting the message or parameters for that. The frequent posting of the messages by the IPC functions from the ISRs should be avoided.

When there are frequent interrupts from the same source, then the messages can be first put in the buffer on each interrupt and when the buffer is sufficiently filled, the IPC message can be posted for ready buffer. This is because if the buffer is not used and the IPC messages are posted frequently by making system calls, OSMsgQPost () function, there will be frequent context switches and hence wastage of time. Example 8.12 shows how the time is saved from frequent context switches.

Example 8.12

Consider ACVM (Section 1.5.2). Consider the task, *Task User Keypad Input* (Example 7.3). When a user presses a key on the AVCM, it generates a hardware device interrupt. Assume that the pressed key is read by taking the ASCII code for the pressed key by the ISR_KeyInputDevice. One option is sending the ASCII code as the queue message for *Task User Keypad Input*. Every time the key is pressed the ISR_KeyInputDevice posts the ASCII code into the message queue. *Task User Keypad Input* is the task, which has to interpret the user key entries. When there is return from the ISR_KeyInputDevice the RTOS ends the wait of the message into the queue and *Task User Keypad Input* gets the ASCII code. However, till such time the user entry is complete, there is no further action and *Task User Keypad Input* again enters the wait for taking the message from the queue.

Second option is that the ISR_KeyInputDevice sends the ASCII code into a key buffer, *KBuffer*. On each interrupt, this is done as long as the user key entry is for the *enter* key. On the *enter* key, the ISR_KeyInputDevice posts a semaphore, *semKB* to the *Task User Keypad Input*. The *Task User Keypad Input* takes the semaphore and reads the *KBuffer* and takes appropriate action as per the user entries of the keys. Second option saves the time spent in the first option in frequent context switches.

Design with Using Interrupt Service Threads or Interrupt Service tasks In certain RTOSes, for servicing the interrupts, there are two levels, fast-level ISRs and slow-level ISTs, the priorities are first for the ISRs, then for the ISTs and then the task (Section 8.7.3 and Example 8.11). The ISRs post the messages for the ISTs and do the detailed computations. If RTOS is providing for only one level, then use the tasks as ISTs.

Design Each Task with an Infinite Loop from Start (Idle State) up to Finish (Last State) Each task has a while loop which never terminates. A task waits for an IPC or signal to start. The task, which gets the signal runs or takes the IPC for which it is waiting, runs from the point where it was blocked or preempted. In pre-emptive scheduler, the high priority task can be delayed for some period to let the low priority task execute.

Example 8.13

Consider ACVM (Section 1.5.2). Example 7.3 showed that ACVM (Section 1.10.2) system can be divided into: (i) *Task User Keypad Input*, (ii) *Task Read-Amount*, (iii) *Chocolate delivery task* (iv) *Display Task*, (v) *GUI_Task* and (vi) *Communication task*. Let us use the five semaphores, which are posted from one task and taken up by another task. Five semaphores are semKB, SemRead, SemChocolate, SemDisplayThanks and SemDisplayCollect. A code design is as follows.

```

/*****
Task User Keypad Input {
.
while (1) { /* Code for sending into key buffer ASCII codes on each run of ISR_Key Input Device */
OSSemPend (semKB); /* Example 8.12 showed use of SemKB */
/* Code for action as per key entries*/
.
OSSemPost (SemRead) /* Post a semaphore to let the Task read amount start */
.
}
}
Task Read_Amount {
int cost; /* The cost of the chocolate selected by the user from user input key*/
while (1) {
OSSemPend (SemRead); /* Wait for message from Task User Keypad Input */
/* Code for action as per reading the Coins, get the value of coins in variable amount */

If (amount >= cost) OSSemPost (SemChocolate) /* Post a semaphore to let the Task Chocolate delivery
start if amount is equal or more than the cost*/
OSSemPost (SemDisplayThanks) /* Post a semaphore to let the Task_Display show the message. Wait
for the nice chocolate*/

}
}
Task Chocolate_Delivery {
.
while (1) {
OSSemPend (SemChocolate); /* Wait for message from Task Read_Amount */
/* Code for action for chocolate delivery */

OSSemPost (SemDisplayCollect) /* Post a semaphore to let the Task_Display show
the message, Thanks you, Collect the nice chocolate. Visit Again. */

}
}
Task Display {
.
while (1) {

```

```

OSSemPend (SemDisplayThanks); /* Wait for message from Task Read_Amount */
/* Code for displaying the message. Thanks you. Collect the nice chocolate. Visit Again */
OSSemPend (SemDisplayCollect); /* Wait for message from Task Read_Amount */
/* Code for displaying the message. the message. Wait for the nice chocolate */
.
.
}
}
/*****

```

First the ISR_KeyInputDevice posts semKB, then Task User Keypad Input starts, which posts SemRead, then Task_Read_Amount starts. Then it posts semaphore semChocolate after the user inserts coins for the appropriate amount. The task design is such that each code is in the waiting loop and waits for the message through the RTOS to start and run the task codes. ISR_KeyInputDevice initiates the chain of actions on the ACVM.

Assume that the priority assigned to the tasks are in the following order: *Task User Keypad Input*, then *Task Read-Amount*, then *Chocolate delivery task* then *Display Task*, then *GUI_Task* and then *Communication task*. As each a semaphore is being posted from one task in higher priority and taken only by another task of lower priority, we could have used the same semaphore, say, *sem0* for semKB, B SemRead and SemChocolate to synchronize the tasks. However, to encapsulate the semaphores between the tasks, we use five semaphores, semKB, SemRead, SemChocolate and SemDisplayThanks.

Design in the Form of Tasks for the Better and Predictable Response Time Control The RTOS provides the control over the response time of different tasks. The different tasks are assigned different priorities and those tasks which system needs to execute with faster response are separated out. For example, in a mobile phone device (Example 1.10.5) there is need for faster response to the phone call-receiving task then the user key input. In digital camera (Example 1.10.4), the task for recording the image needs faster response than the task for downloading the image on a computer through USB port.

Design in the Form of Tasks for Modular Design System of multiple tasks makes the design modular. The tasks provide modular design. For example, in a mobile phone device (Example 1.5.5) we consider the user key input and display as separate tasks. When the display size changes and new display hardware is introduced, only the codes for the display task and resource or data-sharing tasks and ISRs need to be modified. When a new functionality is introduced in the system, the user key input task and new functionality-associated tasks need to be modified.

Design in the Form of Tasks for Data Encapsulation System of multiple tasks encapsulates the code and data of one task from the other. In Example 8.13, the cost is encapsulated in TaskRead_Amount from other tasks and the messages such as 'Thank you, Collect the nice chocolate, Visit Again' encapsulate in the display task.

Design with Taking Care of the Time Spent in the System Calls The expected time in general depends on the specific target processor of the embedded system and the memory access times. However, in order to provide the relative magnitude of the time taken for basic actions at a preemptive scheduler, a new parameter is defined. It defines the time taken for an action by an RTOS scheduler in terms of an assumed scaling parameter, S . S emphasizes the relative magnitudes of execution times for various actions in a typical RTOS.

Let time taken for the simplest instruction be t_{min} . The minimum time is when the semaphores P and V are assigned certain initial values, true or false. Let S be defined in units of T_s . The T_s and t_{min} depend on a specific

target processor of the embedded system and the memory access times. For example, a typical value for a processor, t_{\min} is $0.6 \mu\text{s}$ and let $T_s = (4 \mu\text{s} + 0.6 \mu\text{s})$. Let $t_{\text{exec}} = (t_{\min} + S \cdot T_s)$. This equation defines S as the time over and above t_{\min} in units of a basic time unit, T_s .

If the same RTOS runs on a different processor, the S will therefore remain the same. S is taken as the nearest positive integer for the relative magnitude of execution times for the scheduler actions.

1. $S = 1$ will mean t_{exec} between $2.8 \mu\text{s}$ and $5.2 \mu\text{s}$
2. $S = 2$ will mean $7.4 \mu\text{s}$ to $8.8 \mu\text{s}$
3. $S = 4$ will mean $14.2 \mu\text{s}$ to $18.0 \mu\text{s}$
4. $S = 5$ will mean $20.6 \mu\text{s} \pm 3.0 \mu\text{s}$
5. $S = 10$ will mean $40.6 \mu\text{s} \pm 6.0 \mu\text{s}$
6. $S = 15$ will mean $60.6 \mu\text{s} \pm 8.0 \mu\text{s}$

Table 8.12 gives a list of basic actions in a preemptive RTOS and execution time in terms of a scaling parameter S .

Table 8.12 A List of Basic Actions in a Preemptive RTOS and Execution Time in Terms of a Scaling Parameter S

Action	S	Action	S	Action	S	Action	S
Context Switch	2	Task suspend	1	Sem post/take	1	Message Q delete	10
Task initiate	12	Task resume	1	Take semaphore, mutex, counting semaphore when sem available	1	Q receive Message available	2
Task create and activate	28	Task Lock when no lock or Unlock when lock exists	t_{\min}	Semaphore, mutex, counting semaphore create or delete	6	Q receive Message not available	1
Task delete	10	Mem allocate	2	Release semaphore, mutex, counting semaphore when in Q	3	Message Q Send task pending	5
Task switch flag for running	1	Mem free	4	Mutex flush	1	Message Q send task not pending	2
Task create or delete	18	Network byte send	t_{\min}	Release semaphore, mutex, counting semaphore when no task in Q	1	Message Q Send queue full	1
		Semaphore flag or counting semaphore flush	4	Message Q create	105		

Abbreviations are as in brackets: Semaphore (Sem), Queue (Q) and Memory (Mem).

The RTOS create () function to create a task takes longer CPU time than writing into a queue and then reading from the queue, and using a semaphore takes the least. Therefore, a design should create all the tasks at the beginning, even before the start of the tick of the system clock.

As queue takes longer time than the semaphore, use semaphores if it suffices. For example, consider the codes in Example 8.13. OSSemPost (SemDisplayCollect) is used in place of the queue for posting the message 'Thank you. Collect the nice chocolate. Visit Again.' for the display task. Getting a semaphore takes the least CPU time.

As signals take the least time among the IPCs, use signals for the most urgent IPCs (e.g., error reporting by throwing the exceptions).

Limit the Number of tasks and select the appropriate number of tasks to increase the response time to the tasks, better control over shared resource and reduced memory requirement for stacks

We limit the number the number of tasks appropriately. The tasks, which share the data with number of tasks, can be designed as one single task. The examples of such tasks are: (i) display task for the messages from a number of tasks, (ii) printer task for the messages from a number of tasks and (iii) flash memory writing task for writing into the flash memory by the number of tasks.

Example 8.14

Consider a mobile phone device (Example 1.5.5). It needs to do the following tasks: *display time and date* as per the message posted by the other task, *display battery power* as per the message posted by the other task, *display signal strength* as per the message posted by the other task, *display device profile* (general or silent) as per the message posted by the user input task, *display messages* posted by the other task and *display call status* and *display menus*.

A design can be that which assume each display action as a separate thread. Another design can be a display task, which accepts messages from different tasks and displays. It will lead to creation of only one task for display and one TCB and stack for the display task. A single task for display provides better control of when, where and what to display.

Use Appropriate Precedence Assignment Strategy and Use Preemption in Place of Time Slicing The task of higher priority preempts the low priority tasks and the ISRs preempt the tasks. Therefore, an appropriate precedence is chosen.

The ISRs have higher priorities over the ISTs and tasks.

A mode of scheduling the tasks is assigning them equal priorities and allotting time slice for round robin mode (Section 8.10.2). Time slicing is done in certain specific cases, for example, in a network router when it is routing the packets of multiple clients.

Avoid Task Deletion Create tasks at start-up only and *avoid creating and deleting tasks later*. The only advantage of deleting is the availability of additional memory space. Suppose a task is deleted by an OSTaskDelete () function. Now a situation can be that a task is waiting for a semaphore (to let other tasks finish the critical section) or is waiting for a queue (or mailbox) message for a pointer at the RTOS, and that pointer is for a message to the task that has been deleted. A prolonged blocking or a deadly embrace or a deadlock will then occur. An RTOS may not provide protection for these situations.

Certain RTOS provide an option to make a semaphore *deletion safe*.

Use Idle CPU Time for Internal Functions Often, the CPU may not be running any task. All tasks may be waiting for preemption (for transition from ready place to running place). The CPU at that instant may associate the RTOS for the following. Read the internal queue. Manage the memory. Search for a free block of memory. Delete or dispatch a task. Perform the internal and IPC functions.

Design with Memory Allocation and De-Allocation by the Task If memory allocation and de-allocation are done by the task, the number of RTOS functions is reduced (Example 8.6, Section 8.5). This reduces interrupt latency periods as execution of these functions takes significant time by RTOS whenever the RTOS pre-empts a task.

Further, if fixed sized memory blocks are allocated, then the predictability of time taken in memory allocation is there.

Design with Taking Care of the Shared Resource or Data among the Tasks The ISR coding should be like a reentrant function or should take care of problems from the shared resources or data such as buffer or global variables (Section 7.8). Disabling of running of other tasks for a longer period increases the worst-case interrupt latency periods for all the interrupts. While executing the critical section codes, if possible, instead of disabling the interrupts only the task-switching flag changes should be used. It is done by using semaphore. Thus, only the pre-emption by RTOS should be prevented. Disabling pre-emption may be better than disabling interrupts. However, both increase worst-case interrupt latencies.

Resource locking using the mutex semaphores or spin-locks may be better than disabling preemption or interrupts (Section 8.10.3). A task should take the mutex semaphore only during a short period in which the critical section alone is executed and shared resources (such as the display device driver) are being accessed. Spin locks can also be used in case critical section code is short and executes in time less than the IPC posting and context-switching time (Example 8.20, Section 8.10.3).

Design with Hierarchical and Scalable Limited RTOS Functions Use an RTOS, which is hierarchical as well as scalable so that has only the needed functions are at the ported section of kernel with the rest left outside. This is because the pre-emption scheduling increases the interrupt latency periods because of the time spent in context switching and saving and retrieving pointers for the RTOS functions like memory allocation, and IPCs (Table 8.12).

The functions for the memory management, file system functions, IPC (e.g., pipe, signal, socket and RPC) are provided outside the kernel in a hierarchical and scalable RTOS. MMU is disabled for predictive response time of the tasks (Section 8.8).

Hierarchical RTOS means the RTOS functions portable after extending and interfacing other functionalities and configuring for specific processor and set of devices. Scalable RTOS means portable into the system ROM image after the limited RTOS functions in the kernel space as per the application needs. For example, if queue and pipe functions are not required in an embedded system design, then these functions are not ported in scalable RTOS.

8.9.2 Encapsulation Using the Semaphores and Queues

Semaphores, queues and messages should not be globally shared variables, and each should be shared between a set of tasks only and encapsulated from the rest.

Semaphores A semaphore encapsulates the data during a critical section or encapsulates a buffer from a reading task or writing into the buffer by multiple tasks concurrently. Example 7.14 showed the use of a buffer by producing task (writing into the empty memory addresses buffer) and consuming task (reading from the filled spaces at the buffer). Example 8.15 gives another example.

Example 8.15

Assume that the Task Read_Amount reads the amount and after delivering the chocolate the amount is reduced by a value equal to the cost. The semaphore will insulate the global variable amount.

```

/*****
static int cost; /* The cost of the chocolate selected by the user from user input key*/
static int amount; /* The amount variable */
/* Code for creating semaphore SemAmount with initial value = 1 */
Task Read_Amount {

while (1) {
OSSemPend (SemRead); /* Wait for message from Task User Keypad Input */
OSSemPend (SemAmount); /* Wait for semaphore from amount reducing section in task Chocolate Delivery */
/* Code for action as per reading the Coins, get the value of coins in variable amount */
If (amount >= cost)
OSSemPost (SemChocolate) /* Post a semaphore to let the Task Chocolate delivery start if amount is
equal or more than the cost*/
OSSemPest (SemAmount); /* Post a semaphore to let the Task Chocolate_Delivery reduce the amount by
a value equal to the cost of chocolate after the delivery of the chocolate */
OSSemPost (SemDisplayThanks) /* Post a semaphore to let the Task_Display show the message, Wait
for the nice chocolate*/
}
}
Task Chocolate_Delivery {

while (1) {
OSSemPend (SemChocolate); /* Wait for message from Task Read_Amount */
/* Code for action for chocolate delivery */
OSSemPend (SemAmount) /* Wait for amount ready from Task Read_Amount.*/
amount = amount - cost;
OSSemPost (SemAmount) /* Return the semaphore amount to Task Read_Amount.*/
OSSemPost (SemDisplayCollect) /* Post a semaphore to let the Task_Display show the
message. Thanks you. Collect the nice chocolate. Visit Again. */

}
}
*****/

```

The amount when read by the task read amount, is encapsulated from the task chocolate delivery using the mutex semaphore semAmount. The amount when reduced by the task chocolate delivery, is encapsulated using the semAmount.

Assume that semAmount semaphore is not used for encapsulation of the amount. Let us see the effect. If ISR_KeyInputDevice (Example 8.12) executes on a user input before the amount reduces at task chocolate delivery, the semKB will be posted and then taken by task read amount and it will pre-empt the task chocolate delivery. Even if the user does not insert any coin, the task will wrongly post semChocolate and SemDisplayCollect.

Queues A queue can be used to encapsulate the messages to a task at an instance from the multiple tasks. Assume that a display task is posted a menu for display on a touch screen in a PDA (Example 1.10.6). Multiple tasks can post the messages into the queue for display. When one task is posting the messages and these messages are displayed, another task should be blocked from posting the messages.

We can write a task, which takes the input messages from other tasks and posts these messages to the display task only after querying whether the queue is empty.

Example 8.16

Consider a mobile phone device (Example 1.10.5). Assume that the *Task ReadSMS* and *TaskSelected Menu* posts into a queue to *Task_Display*. The coding can be as follows.

```

/*****
# define QMsgSize = 64 /*Assume that queue can be posted upto 64 pointer variables*/
# define QErrMsgSize = 16 /*Assume that up to 16 error messages can be posted from the queue*/
OS_Event * QMsgPointer /* Create a event control block */
void QMsgPointer [QMsgSize] /* Define a pointer to the queue */
OS_Event * QErrMsgPointer /* Create a event control block for the queue of error messages*/
void *QErrMsgPointer [QErrMsgSize] /* Define a pointer to the queue */
QMsgPointer = OSQCreate (&QMsg [0], QMsgSize) ; /* Create a queue using call to RTOS function OSQCreate */
QErrMsgPointer = OSQCreate (&QMsg [0], QErrMsgSize) ; /* Create a error messages queue using call to RTOS function OSQCreate */
Task_ReadSMS { /* Code for the task which takes the SMS message as input */
while (1) {
OSQQuery (*QMsgPointer, QMsg);
if (QMsg == (void *) 0) {OSQPost (*QMsgPointer, &SMSMsg); } /* Q contains no messages post the SMS message into the queue*/
}
}
Task_Display { /* Code for the task which displays the messages */
while (1) {
DispMsg = (void *) 0
while (QMsg != (void *) 0) {
&DispMsg = OSQPend (*QMsgPointer, 0, &QErrMsg); /* Wait for the messages at the queue till all messages read */
DispMsg ++;
}
}
}

```

The messages are posted into the queue after querying whether it has no messages and thus message pointer points to the null.

9.3 Hard Real-Time Considerations

Hard real time means strict adherence to each task deadline. When an event occurs, it should be serviced within a predictable time at all times in a given hard real-time system. The preemption period for the hard real-time

task in a worst case should be less than a few microseconds. A hard RTOS is one, which has predictable performance with no deadline miss, even in case of sporadic tasks (sudden bursts of occurrence of events requiring attention). Automobile engine control system and antilock brake are the examples of hard real-time systems.

Hard real-time systems provide for the following.

1. Disabling of all other interrupts of lower priority when running the hard real-time tasks.
2. Preemption of higher priority task by lower priority tasks.
3. Some critical code in the assembly to meet the real-time constraint (deadline) fast.
4. Task running in kernel space. This saves the time required to first check whether access is outside the memory space allocated to the kernel functions.
5. Provision of asynchronous IOs.
6. Provision of spin locks.
7. Predictions of interrupt latencies and context switching latencies of the tasks. This is achieved by writing all functions which on execution always take the same time intervals in case of varying rates of occurrences of the events.
8. Response in all the time slots for the given events in the system and thus providing the guaranteed task deadlines even in case of sporadic and aperiodic tasks. Sporadic tasks means tasks executed on the sudden bursts of the corresponding events at high rates, and aperiodic tasks mean tasks having no definite period of event occurrence.

A soft real time is one in which deadlines are mostly met. Soft real time means that only the precedence and sequence for the task operations are defined, interrupt latencies and context switching latencies are small but there can be a few deviations between expected latencies of the tasks and observed time constraints and a few deadline misses are accepted. The preemption period for the soft real-time task in a worst case may be about a few milliseconds. Mobile phone, digital cameras and orchestra-playing robots are examples of soft real-time systems.

8.9.4 Saving of Memory and Power

Methods of Saving and Optimizing the Memory Space

Following are the methods.

1. Use compressed data structure provided the de-compression algorithm plus compressed data structure combined together take less memory in the system compared with the case when only unpacked data structure is used.
2. Make the codes compact and fitted in small memory areas without affecting the code performance. This is called memory optimization. Code means code compiled and assembled executable in the given system. It also reduces the total number of CPU cycles, and thus, the total energy requirements.
3. Use declaration as unsigned byte, especially within the for and while loops, if there is a variable, which always has a value between 0 and 255. When using data structures, limit the maximum size of the queues, lists and stacks size to 256. Byte arithmetic takes less time than integer arithmetic. Follow a rule that uses unsigned bytes especially within the for and while loops for a short integer if possible, to optimize use of the RAM and ROM available in the system. Avoid if possible the use of 'long' integers and 'double' precision floating point value bytes especially within the for and while loops.
4. Avoid use of library functions if a simpler coding is possible. Library functions are the general functions. Use of general function needs more memory in several cases.

Follow a rule that avoids use of library functions in case a generalized function is expected to take more memory especially when its coding is simple.

5. Configure the RTOS functions. For example, if queues are not needed the RTOS queue functions are not ported in the ROM image.

Use a configurable, scalable, hierarchical RTOS which will help the ROM image to execute the needed functions at the kernel.

6. Optimize the RAM use for the stacks. It is done by three methods: (i) reducing the number of tasks that interact with the OS, (ii) reducing the number of nested calls and call at best one more function from a function (one function calling another function and that calling the third and so on means nested calls), (iii) optimize the number of tasks. (Less number of tasks are to be brought first into an initiated task list and there are the frequent interactions with the OS and context savings and retrievals stack on context switching, thus giving more memory and time overheads.) This optimizes the use of the stack. *As a rule reduce the use of frequent function calls and nested calls and thus reduce the time and RAM memory needed for the stacks, respectively.*

7. Optimize the allocation of stacks. A method is that allocated stack areas on allocation are filled with the specific bytes or specific set of bytes. Then find that in worst cases of running of the embedded system, how many filled bytes do not change. Then reduce the allocated stack spaces by rewriting the task, buffer and other memory creation codes.

8. In case the software design can be made fast with the instruction set of the target processor, the assembly codes be used. This also allows the efficient use of memory. The device-driver programs in the assembly especially provide efficiency due to the need to use the bit set-reset instructions for the control and status registers. Only a few assembly codes for using the device I/O port addresses, control and status registers are needed. The best use is made of available features for the given applications. Assembly coding also helps in coding for atomic operations. A modifier *register* can be used in the C program for fast access to a frequently used variable. If portAdata is frequently employed, it is used as follows. `register unsigned byte portAdata`. The modifier *register* directs the compiler to place portAdata in a general purpose register of the processor.

As a rule, use the assembly codes for simple functions like configuring the device control register, port addresses and bit manipulations if the instruction set is clearly understood. Use assembly codes for the atomic operations for increment and addition. Use modifier "register" in C program for a frequently used variable.

9. Calling a function causes context saving on a memory stack and on return the context is retrieved. This involves time and can increase the worst-case interrupt latency. There is a modifier *inline*. When the *inline* modifier is used, the compiler inserts the actual codes at all the places where these operators are used. This reduces the time and stack overheads in the function call and return. But, this is at the cost of more ROM being needed for the codes. If used, it increases the size of the program but gives a faster speed. Using the modifier directs the compiler to put the codes for the function (in curly braces) instead of calling that function.

As a rule, use inline modifiers for all frequently used small sets of codes in the function or the operator overloading functions if the ROM is available in the system. A vacant ROM memory is an unused resource. Why not use it for reducing the worst-case interrupt latencies by eliminating the time taken in the frequent save and retrieval of the program context?

10. When a variable is declared static, the processor accesses with less number of instructions than from the stack. As long as shared data problem does not arise, the use of static (global) variables can be optimized. These are not used as the arguments for passing the values. A good function is one that has no arguments to be passed. The passed values are saved on the stacks in case of interrupt service calls and other function calls. Besides obviating the need for repeated declarations, the use of global variables

will thus reduce the worst-case interrupt latency and the time and stack overheads in the function call and return. But this is at the cost of the codes for eliminating shared data problem.

As a rule, use static (global) variables if shared data problems are tackled and use static variables in case it needs saving frequently on the stack.

11. Combine two functions if possible. For example, the search functions for finding pointers to a list item and pointers of previous list items combine into one. If present is *false* the pointer of the previous list item retrieves the one that has the item.

As a rule, combine whenever feasible two functions of more or less similar codes.

12. Use if feasible, alternatives to the *switch-case* statements, a table of pointers to the functions. This saves the processor time in deciding which set of statements to execute in place of performing the conditional tests all down a chain.

13. When using C++, configure the compiler for not permitting the multi-inheritance, templates, exceptional handling, new style casts, *virtual* base classes and *namespaces*.

As a rule, for using C++, use the classes without multiple inheritance, without template, with run-time identification and with throwable exceptions.

14. When using Java, use the J2ME and configure the device classes.

As a rule, use J2ME with device configurations when programming small-devices code in Java.

Embedded software designers can use various standard ways for optimizing the memory needs in the system.

Methods of Saving and Optimizing the Power Needs Following are the methods of power saving.

Switch to standby and stop modes An embedded system has to perform tasks continuously from power-up and may also be left in power-ON state; therefore, power saving during execution is important. A microcontroller used in the embedded system must provide for executing *Wait* and *Stop* instructions and operation in power-down mode. One way to do this is to cleverly incorporate into the software the *Wait* and *Stop* instructions. For example, a program can be such that it reduces the brightness level of the LCD panel so that it takes less power when the system is used in a fully lighted room. A sensor senses the light level at specific intervals.

An embedded system may need to be run continuously, without being switched off; the system design, therefore, is constrained by the need to limit power dissipation while it is running. Total power consumption by the system in running, waiting and idle states should also be limited. A program can provide for auto-switch over the standby mode in case the system is not used within a specified time interval and stop mode when the system is not used for long intervals. For example, mobile phone auto-switch off the LCD lights when not using for 5 or 10 or 20 seconds. A call attend mode can be switched off if there is no talk for over a minute.

The *current* needed at any instant in the processor for an embedded system depends on the state and mode of the processor. The following are the typical values in six states of the processor.

- 50 mA when only the processor is running; that is, the processor is executing instructions.
- 75 mA when the processor plus the external memories and chips are in a running state; that is, fetching and execution are both in progress.
- 15 μ A when only the processor is in the stop state; that is, fetching and execution have both stopped and the clock has been disabled from all structural units of the processor.
- 15 μ A when the processor plus the external memories and chips are in the stop state; that is, fetching and execution have both stopped and the clock disabled from all system units.
- 5 mA when only the processor is in the waiting state; that is, fetching and execution have both stopped but the clock has not been disabled from the structural units of the processor, such as timers.

6. 10 mA when the processor, the external memories and the chips are in the waiting state. Waiting state now means that fetching and execution have both stopped; but the clock has not been disabled from the structural units of the processor and the external IO units and dynamic RAM refreshing also has not stopped.

Disable cache mode Yet another method is to disable use of certain structural units of the processor – for example, caches – when not necessary and to keep in disconnected state those structure units that are not needed during a particular software portion execution, for example, timers or IO units. The software designer should enable the use of caches in a processor by an appropriate instruction, to obtain greater performance during run of a section of a program, while simultaneously disabling the remaining sections in order to reduce the power dissipation and minimize the system energy requirement. Hardware designers should select a processor with multiway cache units so that only that part of a cache unit gets activated that has the data necessary to execute a subset of instructions. This also reduces power dissipation.

Reduce circuit glitches In a CMOS circuit, power dissipates only at the instance of change in input. Therefore, unnecessary glitches and frequent input changes increase power dissipation. VLSI circuit designs have a unique way of avoiding power dissipation. A circuit design is made such that it eliminates all removable glitches, thereby eliminating any frequent input changes.

Low-voltage operation modes Another is to operate the system at the lowest voltage levels in the idle state by selecting power-down mode in that state.

(1) The processor goes into a stop state when it receives a 'stop' instruction. The stop state also occurs in the following conditions: (i) On disabling the clock inputs to the processor. (ii) On stopping the external clock circuit functions. (iii) On the processor operating in auto-shutdown mode. When in the stop state, the processor disconnects with the buses (buses become in tri-state). The stop state can change to a running state. The transition to the running state is either because of a user interrupt or because of the periodically occurring wake-up interrupts.

(2) The processor goes into a waiting state either on receiving (i) an instruction for *Wait*, which slows or disables the clock inputs to some of the processor units including ALU, or (ii) when an external clock circuit becomes non-functional. The timers are still operating in the waiting state. The waiting state changes to the running state when either (i) an interrupt occurs or (ii) a reset signals.

(3) Power dissipation reduces typically by 2.5 μ W per 100 kHz reduced clock rate. So reduction from 8000 kHz to 100 kHz reduces power dissipation by about 200 μ W, which is nearly similar to when the clock is non-functional. [Remember, the total power dissipated (energy required) may not reduce. This is because on reducing the clock rate the computations will take a longer time at the lower clock rate and the total energy required equals the power dissipation per second multiplied by the time]. The power 25 μ W is typically the residual dissipation needed to operate the timers and few other units. By operating the clock at lower frequency or during the power-down mode of the processor, the advantages are as follows: (i) heat generation reduces. (ii) radiofrequency interference also then reduces due to the reduced power dissipation within the gates. [Radiated RF (radiofrequency) power depends on the RF current inside a gate, which reduces due to increase in 'ON' state resistance between the drain and channel when there is reduced heat generation.]

(4) Low-voltage systems are built using LVCMS (low-voltage CMOS) gates and LVTTTL (low-voltage TTL). Use of 3.3 V, 2.5 V, 1.8 V and 1.5 V systems and IO interfaces other than the conventional 5 V systems results in significantly reduced power-consumption and can be advantageously used in the following cases. (i) In portable or hand-held devices such as a cellular phone (compared with 5 V, a CMOS circuit power dissipation reduces by half, $\sim(3.3/5)^2$, in 3.3 V operation. This also increases the time intervals needed for recharging the battery by a factor of two). (ii) In a system with smaller overall geometry, the low-voltage system processors and IO circuits generate lesser heat and thus can be packed into a smaller space.

Clever real-time programming by using 'Wait' and 'Stop' instructions and disabling certain units when not needed is one method of saving power during program execution. Operations can also be performed at reduced clock rate when needed in order to control power dissipation. Good design must optimize the conflicting needs of low power dissipation and fast and effective program execution.

8.10 RTOS TASK SCHEDULING MODELS, INTERRUPT LATENCY AND RESPONSE TIMES OF THE TASKS AS PERFORMANCE METRICS

Following are the common scheduling models used by schedulers.

1. Cooperative scheduling of ready tasks in a circular queue. It closely relates to function queue scheduling.
2. Cooperative scheduling with precedence constraints.
3. Cyclic and round robin (time slicing) scheduling.
4. Preemptive scheduling.
5. Scheduling using 'earliest deadline first' (EDF) precedence.
6. Rate monotonic scheduling using 'higher rate of events occurrence First' precedence.
7. Fixed times scheduling.
8. Scheduling of periodic, sporadic and aperiodic tasks.
9. Advanced scheduling algorithms using the probabilistic timed Petri nets (stochastic) or multithread graphs. These are suitable for multiprocessors and for complex distributed systems.

An RTOS commonly executes the codes for the multiple tasks as priority-based preemptive scheduler.

8.10.1 Cooperative Scheduling Model

First consider a scheduling by a cooperative scheduler function by a simple example. Consider an embedded system – an automatic washing machine. The system can be partitioned into multiple tasks. First three tasks are task A1, task A2 and task A3 in a set of tasks A₁ to A_N. Figure 8.2(a) shows the first three tasks of the multiple process embedded software. The scheduler first starts the task A1 waiting loop and waits for the message A1 from task A1.

1. **Task A1:** The task is to reset the system and switch on the power if the door of the machine is closed and the power switch pressed once and released to start the system. Task 1 waiting loop terminates after detection of two events – (i) door closed and (ii) power switch pressed by the user. At the end, task 1 sets a flag start_F, which is a message A1 to schedule task A2 to start executing code. This message can be sent using semaphore function OSSemPost (start_F) (Section 7.7.1).
2. **Task A2:** The scheduler waits for the message A1 for start_F setting. The waiting can be by using semaphore function OSSemPend (start_F). If start_F posting event occurs at task 1, the task 2 starts. A bit is set to signal water into the wash tank and repeatedly checks for the water level. When the water level is adequate the flag water-stage1_F is set, which is a message A2 to schedule task A3 to start executing code. This message can be sent using semaphore function OSSemPost (water-stage1_F).
3. **Task A3:** The scheduler waits for the message A2 for the stage1_F setting. The waiting can be by using semaphore function OSSemPend (water-stage1_F). If water-stage1_F posting event occurs at task 2 the task 3 wait ends and starts. A bit is set to stop water inlet and another bit sets to start the wash tank motor. Then a flag, motor-stage1_F is set, which is a message A3, to the schedule the next task to start executing code. This message can be sent using semaphore function OSSemPost (motor-stage1_F).

Figure 8.2(b) shows the cooperative scheduling model. Figure 8.2(c) shows the task program contexts at various instances. Task A1 context has a pointer for task A1, ADDR_A1. Task A2 context has a pointer for task A2, ADDR_A2. Task A3 context has a pointer for task A3, ADDR_A3.

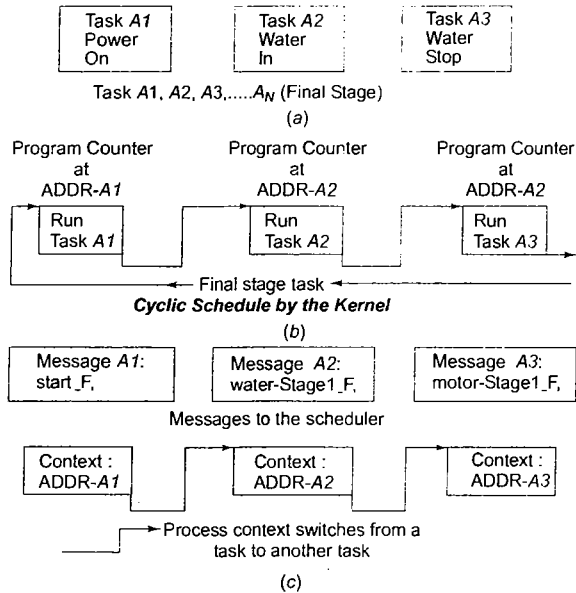


Fig. 8.2 (a) First three tasks in a set of tasks A_1 to A_N into which the embedded software is broken for the example in the text (b) Cyclic scheduling (c) Messages from the scheduler and task program contexts at various instances in washing machine tasks

The Cooperative Scheduling of Ready Tasks List Figure 8.3(a) shows a scheduler in which the scheduler inserts into a list the ready tasks for sequential execution in cooperative model. Program counter PC changes whenever the CPU starts executing another process. Figure 8.3(b) shows how the PC changes on switch to another context. The scheduler switches the context such that there is sequential execution of different tasks, which the scheduler calls from the list one by one in a circular queue.

Cooperative means that each ready task cooperates to let a running one finish. None of the tasks does a block anywhere during the ready to finish states. The service is in the order in which a task is initiated on interrupt and laced in ready list. We can say that the task priority parameter sets as per its position in the queue.

Worst-case latency is the same for each task. It is t_{total} . It is time-period of the circular queue. The longer the queue, the greater is the t_{total} . If a task is running, all other ready tasks must wait. For an i -th task, let the vent detection time when an event is brought into a list be dt_i , switching time from one task to another be st_i , and task execution time be et_i . Then if there are n tasks in the ready list, the worst-case latency with scheduling when including the ISRs execution times will be:

$$T_{worst} = \{(dt_i + st_i + et_i)_1 + (dt_i + st_i + et_i)_2 + \dots + (dt_i + st_i + et_i)_{n-1} + (dt_i + st_i + et_i)_n\} + t_{ISR} = t_{total} + t_{ISR}$$

Here the t_{ISR} is the sum of all execution times for the ISRs. Remember, T_{worst} should always be less than the deadline, T_d for any of the task in the list (Refer to Section 4.6).

The Cooperative Scheduling of Ready Tasks Using an Ordered List as per Precedence constraints Figure 8.4(a) shows a cooperative *priority-based* scheduling of the ISRs executed in the first

layer (top-right side) and *priority-based* ready tasks at an ordered list executed in the second layer (bottom-left), respectively. Figure 8.4(b) shows the PC switch at different times, when the scheduler calls the ISRs and the corresponding tasks at an ordered list one by one. The scheduler using a priority parameter, $taskPriority$, does the ordering of list of the tasks.

The scheduler first executes only the first task at the ordered list, and the t_{total} , equals the period taken by the first task on the list. It is deleted from the list after the first task is executed and the next task becomes the first. The insertions and deletions for forming the ordered list are made only at the beginning of each list.

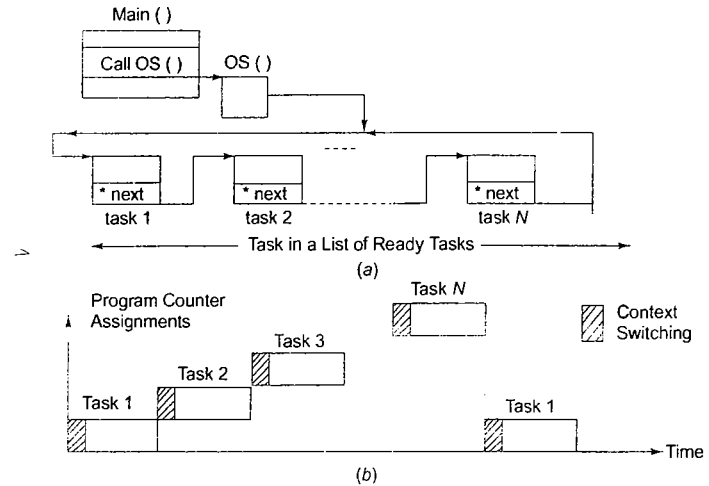


Fig. 8.3 (a) An OS scheduling in which the scheduler inserts into a list the ready tasks for a sequential execution in a cooperative mode (b) Program counter assignments (switch) at different times, when the scheduler calls the tasks one by one in the circular queue from the list

At the first layer, an ISR has a set of short codes that have to be executed immediately. The ISRs run in the first layer (top-right in figure) according to their assigned priorities. It sends a flag(s) or token(s) and its priority parameter for the task to be initiated (serviced). This task inserts into the ready task list. There is cooperative scheduling and each ready task cooperates to let the running one finish. None of the tasks does a block anywhere from the start to finish. Here, however, the next start of scheduling is among the ready tasks that run in turn only from a priority-wise ordered list. The ordering is according to the precedence of the interrupt sources and tasks.

Let p_{min} be the priority of that task which has the maximum execution time. Then worst-case latencies for the highest priority and lowest priority tasks will now vary from:

$$\{(dt_i + st_i + et_i)_{p_{min}} + t_{ISR}\} \text{ to } \{(dt_i + st_i + et_i)_{p_1} + (dt_i + st_i + et_i)_{p_2} + \dots + (dt_i + st_i + et_i)_{p_{m-1}} + (dt_i + st_i + et_i)_{p_m} + t_{ISR}\}.$$

Here, p_1, p_2, \dots, p_{m-1} and p_m are the priorities of the tasks in the ordered list. Also $p_1 > p_2 > \dots > p_m$. With this scheduler, it is easier, but not guaranteed, to meet the requirement that T_{worst} should be $< T_d$ for each task and interrupt source. The programmer assigns the lowest T_d task a highest priority.

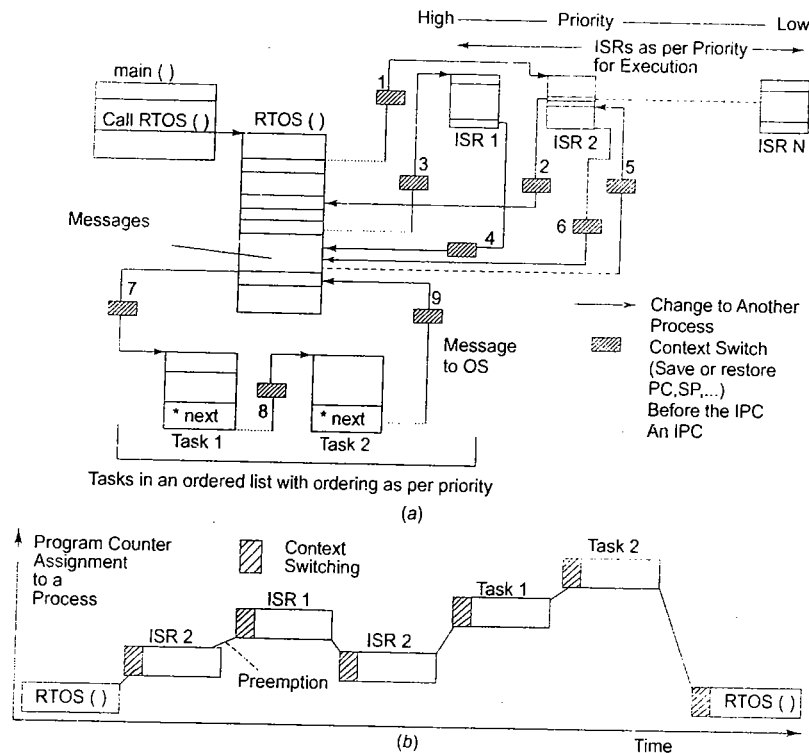


Fig. 8.4 (a) Cooperative priority-based scheduling of the interrupt service routines (ISRs) executed in the first layer (top-right side) and priority-based ready tasks at an ordered list executed in the second layer (bottom-left) (b) Program counter assignments at different times on the scheduler calls to the ISRs and the corresponding tasks

Example 8.17

Consider the ACVM example (Section 1.10.2). First the coins inserted by the user are read, then the chocolate delivers, and then display task displays 'thank you, visit again' message. Each task cooperates with the other to finish. The precedence of the task reading the coins is highest, then of chocolate delivery and display for the ordered list of ready tasks.

8.10.2 Cyclic and Round Robin with Time Slicing Scheduling Models

Cyclic Scheduling An OS scheduler can let the system schedule the various tasks in real time as follows: let us assume that we have periodically occurring three tasks, the need for their service arises after periodically. Let the time-frames be allotted to the first task, the task executes at $t_1, t_1 + T_{\text{cycle}}, t_1 + 2 \times T_{\text{cycle}}, \dots$ second task

frames at $t_2, t_2 + T_{\text{cycle}}, t_2 + 2 \times T_{\text{cycle}}$ and the third task at $t_3, t_3 + T_{\text{cycle}}, t_3 + 2 \times T_{\text{cycle}}, \dots$. Start of a time frame is the scheduling point for the next task in the cycle. T_{cycle} is the cycle for repeating the cycle of execution of tasks in order 1, 2 and 3 and equals start of task 1 time frame to end of task 3 frame. T_{cycle} is the period after which each the task time frame allotted to that repeats.

Each of the N tasks in a cyclic scheduler completes in its allotted time frame when the time frame size is based on the deadline. A cyclic scheduler is clock-driven and is useful for the periodic tasks. It repeats the schedule decided after computations based on the period of occurrences of task instances. Each task has the same priority for execution in the cyclic mode.

Example 8.18

(a) Consider the video and audio signals reaching at the ports in a multimedia system and processed. The video frames reach at the rate of 25 in 1 second. The cyclic scheduler is used in this case to process video and audio with $T_{\text{cycle}} = 40$ ms or in multiples of 40 ms.

(b) Consider the orchestra-playing robots example (Section 1.10.7). First the director robot sends the musical notes. Then, the playing robots receive and acknowledge to the director. In the next cycle, the master robot again sends the musical notes. The cyclic scheduler is used in this case to send and receive signals by each robot.

A cyclic scheduling is very efficient for handling periodic tasks and when the number of tasks is small.

Round Robin Time Slicing Scheduling A task may not complete in its allotted time frame. Round robin means that each ready task runs in turn only in a cyclic queue for a limited time slice $T_{\text{slice}} = T_{\text{cycle}} \div N$, where $N =$ number of tasks. It is a widely used model in traditional OS. Round robin is a hybrid model of the clock-driven model (e.g., cyclic model) as well as event-driven (e.g., preemptive). A real-time system responds to the event within a bound time limit and within an explicit time. A scheduler for the time-constrained tasks in the round robin mode can be understood by a simple example.

Suppose after every 20 ms, there is a stream of coded messages reaching at port A of an embedded system, it is then decrypted and retransmitted to the port after encoding each decrypted message. The multiple processes consist of five tasks: $C1, C2, C3, C4$ and $C5$, as follows:

1. Task $C1$: Check for a message at port A every 20 ms.
2. Task $C2$: Read port A and put the message in a message queue.
3. Task $C3$: Decrypt the message from the message queue.
4. Task $C4$: Encode the message from the queue.
5. Task $C5$: Transmit the encoded message from the queue to port B.

Figure 8.5(a) shows five tasks, $C1$ to $C5$, that are to be scheduled. Figure 8.5(b) shows the five contexts in five time schedules, between 0 and 4 ms, 4 and 8 ms, 8 and 12 ms, 12 and 16 ms and 16 and 20 ms, respectively. Let OS initiate $C1$ to $C5$. Let there be slice-clock tick interrupts at each 4 ms. Task $C1$ is scheduled by OS to bring it to the running state from its blocked state as soon as a timer triggers an event. If it is known that after every 20 ms a byte reaches port A, a timer interrupt triggers an event every 4 ms. Task $C1$ runs within 4 ms, and $C2$ starts running.

Figure 8.5(b) shows at different time slices the real-time schedules, process contexts and saved contexts.

1. At the first instance (first row) the context is $C1$ and task $C1$ is running.
2. At the second instance (second row) after 4 ms, the OS switches the context to $C2$. Task $C1$ is finished, $C2$ is running. As task $C1$ is finished, nothing is saved on the task $C1$ stack.

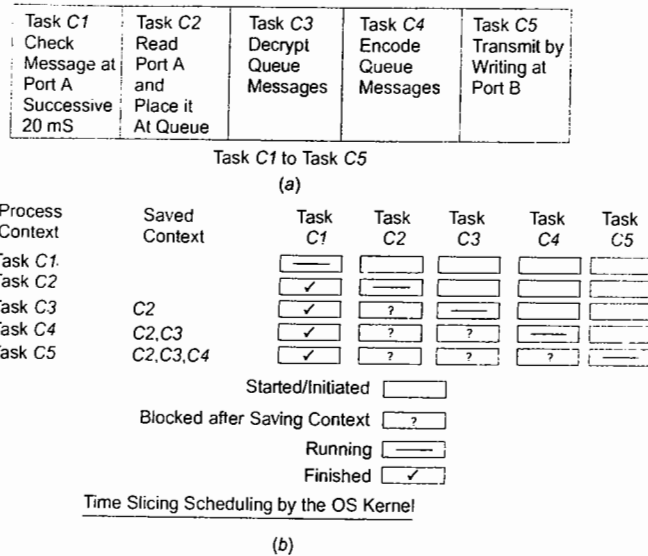


Fig. 8.5 (a) The tasks C1 to C5 round robin (b) Task program contexts at five instances in the round robin (time slice) scheduling scheduler for C1 to C5 with $T_{\text{slice}} = 4 \text{ ms}$

3. At the third instance (third row), the OS switches the context to C3 on next timer interrupt, which occurred after 8 ms from the start of task C1. Task C1 is finished, C2 is blocked and C3 is running. Context C2 is saved on task C2 stack because C2 is in blocked state.
4. At the fourth instance (fourth row), the OS switches the context to C4 on timer interrupt, which occurred after 12 ms from the start of task C1. Task C1 is finished, C2 and C3 are blocked and C4 is running. Contexts C2 and C3 are at the tasks C2 and C3 stacks, respectively.
5. At the fifth instance (fourth row), the OS switches the context to C5 on next timer interrupt which occurred after 16 ms from the start of task C1. Task C1 is finished, C2, C3 and C4 are blocked and C5 is running. Contexts C2, C3 and C4 are at the tasks C2, C3 and C4 stacks, respectively.
6. On a timer interrupt at the end of 20 ms, the OS switches the context to C1. As task C5 is finished, only the contexts C2, C3 and C4 remain at the stack. Task C1 is running as per its schedule.

When a p -th task has high execution time, et_p , the worst-case latency of the lowest priority task can exceed its deadline. To overcome this problem, it is better that the OS defines a lower time slice for each task. Each task has codes in an infinite loop. Cyclic scheduling with time slicing is simple and there is no insertion or deletion into the queue or list. Figure 8.6(a) shows a programming model for cyclic time-sliced round robin scheduling. Figure 8.6(b) shows PC on context switches when the scheduler call to tasks at two consecutive time slices. Each task is allotted a maximum time interval $= t_{\text{slice}}/N$, where t_{slice} is the time after which a timer with the OS) interrupts and initiates a new cycle.

The OS completes the execution of all ready tasks in one cycle within a time slice, $N \times t_{\text{slice}}$ in this mode. Let T_{worst} be the sum of the maximum times for the all the tasks if there are N tasks in all. Then, when $t_{\text{slice}} > T_{\text{worst}}$, the T_{worst} equals:

$$\{(dt_i + st_i + et_i)_1 + (dt_i + st_i + et_i)_2 + \dots + (dt_i + st_i + et_i)_{n-1} + (dt_i + st_i + et_i)_m\} + t_{\text{ISR}}$$

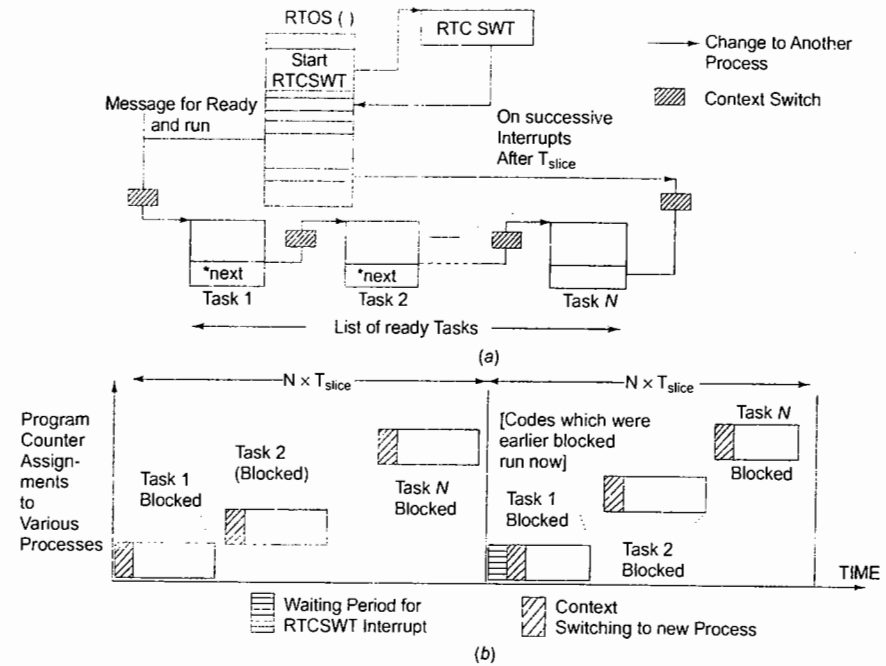


Fig. 8.6 (a) The programming model for the cooperative time-sliced scheduling of the tasks (b) The program counter assignments on the scheduler call to tasks at two consecutive time slices. Each cycle takes a time of t_{slice}

If $N \times t_{\text{slice}}$ equals the sum of the maximum times for each task, then each task is executed once and finishes in one cycle itself. When a task finishes the execution before the maximum time it can take, there is a waiting period between the two cycles. The worst-case latency for any task is $N \times t_{\text{slice}}$. A task may periodically need execution. The period for the required repeat execution of a task is an integral multiple of t_{slice} . For each task to run only once, the $N \times t_{\text{slice}}$ should also be less than the greatest common factor of all the task periods. The estimation of response time for each task is easy in time slice cyclic round robin scheduling. Consider a k -th task. The task responds within its task period plus the sum of the maximum times taken during a time slice from the task 1 to task $(k-1)$. The response time of the m -th task at the end of the list is the maximum.

An alternative model strategy can be the decomposition of a task that takes an abnormally long time to be executed. The decomposition is into two or four or more tasks. Then one set of tasks (or the odd numbered tasks) can run in one time slice, t_{slice} and another set of tasks (or the even-numbered tasks) in another time slice, t_{slice} .

Another alternative strategy can be the decomposition of the long time-taking task into a number of sequential states or a number of node places and transitions as in the FSM. Then one of its states or transitions runs in the first cycle, the next state in the second cycle and so on. This task then reduces the response times of the remaining tasks that are executed after a state.

Example 8.19

Assume a VoIP [Voice Over IP] router. It routes the packets to N destinations from N sources. It has N calls to route. Each of the N tasks is allotted a time slice and is cyclically executed for a routing packet from a source to its destination.



8.10.3 Preemptive Scheduling Model

Cooperative schedulers (described in Section 8.10.1) schedule such that each ready task cooperates to let the running one finish. However, a disadvantage of the cooperative scheduler is that a long execution time of a low-priority task makes a high-priority task wait at least until it finishes. There is a further disadvantage if the cooperative scheduler is cyclic but without a predefined t_{slice} . Assume that an interrupt for service from the first task occurs just at the beginning of the second task. The first task service waits till all other remaining listed or queued tasks finish (Section 8.10.1).

The time-slicing scheduler is simpler in design and extremely valuable in many applications where there is a need to use the resources of the embedded systems sequentially, or none of the tasks has a shorter deadline than the t_{slice} or t_{cycle} . Round robin scheduler (described in Section 8.10.2) also give appropriate time slice to let a task finish with the allotted time frame. Now consider the problem with round robin. Let there be N tasks from task 1 to task N and let the assigned order of priority for interrupt servicing be from 1 (highest) to N (lowest). Assume now that an interrupt occurs in the time-slicing scheduling just after the cycle starts. It means task 1 misses by a flick the chance of running from start to finish as task 1 will not get serviced till the cycle up to task N finishes or till the defined period t_{slice} expires.

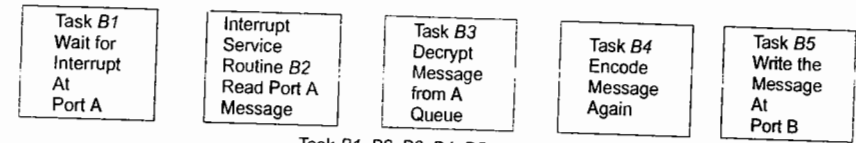
Can the higher-priority task preempt a lower priority by blocking it? If yes, then this can solve the problem of large worst case latency for high priority tasks. The hardware polls to determine whether an ISR or task with a higher priority than the present one needs service at the end of an instruction during execution. If yes, then the higher priority ISR or task is executed. Similarly, the RTOS preemptive scheduler can block a running task at the end of an instruction by a message to the task and let the one with the higher priority take control of the CPU.

Now consider a preemptive scheduler by a simple example. Suppose there is a stream of coded messages reaching at port A of an embedded system. It then decrypts and re-transmits to port B after encoding each decrypted message (recall Example 4.1). Figure 8.7(a) shows the tasks for the multiple processes of this application. Five processes are executed at five tasks, B1, B2, B3, B4 and B5. Now consider preemptive scheduling by a scheduler function by another example. Consider an embedded system for reading a port A input and decrypting the input data, encoding it and sending it to another port B output. The system can be partitioned into multiple tasks. Five tasks are task B1, B2, B3, B4 and B5. Figure 8.7(a) shows the assigned functions to the task and ISR. The order of priorities is as follows.

1. Task B1: Check for a message at port A.
2. Task B2: Read port A.
3. Task B3: Decrypt the message.
4. Task B4: Encode the message.
5. Task B5: Transmit the encoded message to the port.

Figure 8.7(b) gives the symbols used to show the preemptive scheduling by the kernel pre-emptive scheduler actions shown in Figure 8.7(c). A higher priority task takes control from a lower priority task. A higher priority task switches into the running state after blocking the low priority task. The context saves on the pre-emption. Figure 8.7(c) shows the following.

1. At the first instance (first row) the context is B3 and task B3 is running.
2. At the second instance (second row) the context switches to B1 as context B3 saves on interrupt at port A and task B1 is of highest priority. Now task B1 is in a running state and task B3 is in a blocked state. Context B3 is at the task B3 stack.
3. At the third instance (third row) the context switches to B2 on interrupt, which occurs only after task B1 finishes. Task B1 is in a finished state. B2 in a running state and task B3 is still in the blocked state. Context B3 is still at the task B3 stack.
4. At the fourth instance (fourth row) context B3 is retrieved and the context switches to B3. Tasks B1 and B2, both of higher priorities than B3, are finished. Tasks B1 and B2 are in finished states. Task B3 blocked state changes to running state and B3 is now in a running state.
5. At the fifth instance (fifth row) the context switches to B4. Tasks B1, B2 and B3, all of higher priorities than B4, are finished. Tasks B1, B2 and B3, are in the finished states. B4 is now in a running state.
6. At the sixth instance (sixth row) the context switches to B5. Tasks B1, B2, B3 and B4, all of higher priorities than B5, are finished. Tasks B1, B2, B3 and B4, are in the finished states. B5 is now in a running state.



Task B1, B2, B3, B4, B5
(a)

States during different contexts one offer one

State	✓	Finished
State	?	Blocked
State	—	Running

(b)

Preemptive scheduling by RTOS kernel

Process Context	Task B1	Task ISR B2	Task B3	Task B4	Task B5	Saved Context
B3	☐	☐	☐	☐	☐	
B1	☐	☐	?	☐	☐	B3
B2	☐	☐	?	☐	☐	B3
B3	☐	☐	☐	☐	☐	
B4	☐	☐	☐	☐	☐	
B5	☐	☐	☐	☐	☐	
B1	☐	☐	☐	☐	☐	
B2	☐	☐	☐	☐	☐	B5
B3	☐	☐	☐	☐	☐	B5
	☐	☐	☐	☐	☐	B5

(c)

Fig. 8.7 (a) First five tasks B1 to B5 (b) The symbols used for the states in a preemptive scheduling (c) The task program contexts at the various instances

7. At the seventh instance (seventh row) the context switches to *B1* as context *B5* is saved on interrupt at port A, and task *B1* is of highest priority. Now task *B1* is in a running state and task *B5* is in a blocked state. Context *B5* is at the task *B5* stack.
8. At the eighth instance (eighth row) the context switches to *B2* on interrupt, which occurs only after task *B1* finishes. Task *B1* is in a finished state, *B2* in a running state and task *B5* is still in the blocked state. Context *B5* is still at the task *B5* stack.
9. At the last instance (last row) the context is *B3* and task *B3* is running. The tasks *B1* and *B2* are in finished state.

RTOS manages the processes and provides for preemption of lower priority process by higher priority process. (Table 8.11). Let the priority of $\text{task}_1 > \text{task}_2 > \text{task}_3 > \text{task}_4 \dots > \text{task}_N$. Figure 8.8(a) shows the preemptive scheduling of *N* tasks. Figure 8.8(a) also shows the context switching whenever the process switches from a task to the RTOS and from the RTOS to a task. Figure 8.8(b) shows PC assignments on the scheduler call to pre-empt task 2 when the priority of $\text{task}_1 > \text{task}_2 > \text{task}_3$.

Each task has an infinite loop from start (idle state) up to finish (refer to task 1, task 2 and task *N*, three boxes at the bottom of this figure). Last instruction of task 1 points to the next pointed address, *next. In case of the infinite loop, *next points to the same task 1 start. It is unlike a cooperative scheduler (Section 8.10.1), where it signals the next task execution to the OS and OS now initiates and runs the next task in the ready list.

In a preemptive scheduler, there is an RTOS message during the running of task 2 to preempt the task 2. Figure 8.8(a) shows the sequence markings (1), (2) and (3) and Figure 8.8(b) shows program counter assignment. Their meanings are as follows. In step 1, task 2 is run. The higher priority task 1 is initiated as follows:

1. Task 2 blocks and sends a message to the RTOS (step 2).
2. The RTOS now sends a message to task 1 to go to the unblocked state and run (step 3).

After task 1 blocks then RTOS makes the task 2 in the unblocked state. Task 2 now runs. When task 2 blocks then RTOS makes task 3 in the unblocked state. Task 3 will run now.

Each task design is like an independent program, in an infinite loop between the task ready place and the running task place. The task does not return to the scheduler, as a function does. Within the loop, the actions and transitions are according to the events or flags or tokens. The context switching may also occur on an ISR call.

We can define timeout for waiting for the token or event. An advantage of using time out intervals while designing task codes is that worst-case latency estimation is possible. Any task's worst-case latency is the sum of the t_{ISR} and the intervals of all other tasks are of higher priority. Another advantage of using the time outs is the error reporting and handling by the RTOS. Timeouts provide a way to let the RTOS run even the lowest priority task in necessary cases.

Whenever the preemption event takes place, a task switching (a task place transition to its running place) becomes necessary, and the scheduler searches for the highest priority task at that instance. That task only is switched to the running place by the scheduler. Switching occurs when a `taskSwitchFlag` is sent to the highest priority task and not to the task that was running previously.

How can the context switching intervals reduce? The context switching intervals are reduced by the static declaration of the variables, as the static variables are RAM-resident variables and do not save on the stack on a function call. When this is the case, on a call the PC and few must-save registers are saved. Task switching now does not lead to additional stack-saving overheads.

The conditions in which an event (token), the *preemptionEvent*, is generated for a task to undergo transition from the running place to the ready place are as follows.

1. The preemption event takes place when an interrupt occurs and just before the return from the interrupt, there is a service call to the RTOS by the ISR. On this call to the RTOS, a token, the *preemptionEvent*,

is set. The task then undergoes transition to the place, *readyTaskPlace*, and runs only when asked by the scheduler (by sending *taskSwitchFlag*).

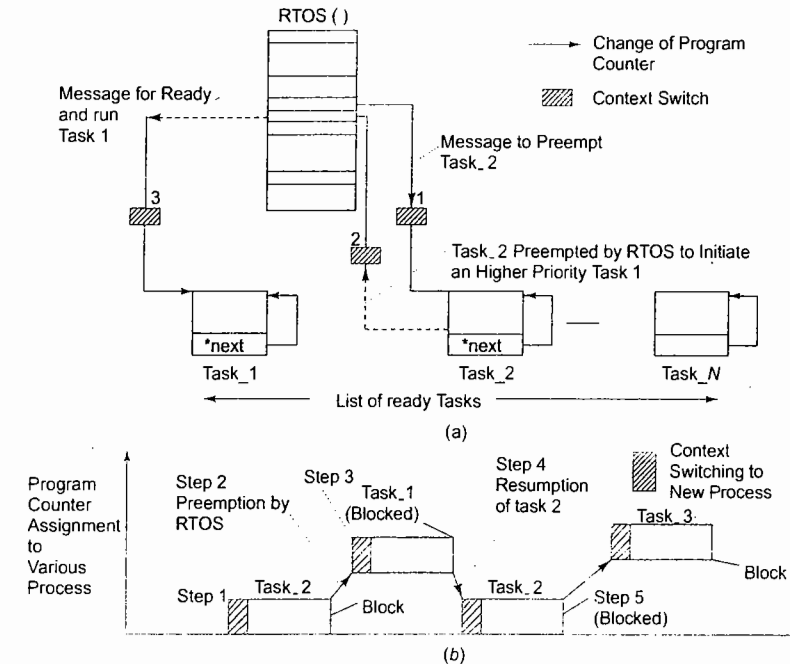


Fig. 8.8 (a) Preemptive scheduling of the tasks (a running task is pre-empted and blocked to let a higher priority task be executed). Note the sequence markings (1), (2) and (3). For meanings of these refer to text (b) Program counter assignments on a scheduler call to preempt task 2. Priority of $\text{task}_1 > \text{task}_2 > \text{task}_3$

2. Each RTOS uses a system clock ticked by a `SysClkIntr` interrupt. The preemption event takes place when the `SysClkIntr` interrupt (real-time clock-driven software timer interrupt) occurs at the RTOS. On this event RTOS takes control of the processor and checks whether it should let currently executing task continue or to preempt it to make way for the higher priority task. This event makes another higher priority task ready to run, on the switch of the flag to the latter.
3. The preemption event takes place when any call to the RTOS occurs to enter the critical section or for sending the task message (outputs) to the RTOS, and if another higher priority task then needs to be serviced (take control of the CPU) (now the preemption is before entering the critical section).

Critical Section Service by a Preemptive Scheduler Critical section is a section in a system call (OS function), where there is no preemption by either ISRs or higher priority tasks. Critical section is also a section in task to prevent preemption. A lock function executes before beginning of critical section and an unlock function executes at exit from the critical section.

Assume that a task I is waiting for a critical section resource, and task J is using a kernel lock at an instance because of the system call for lock by another task j . When task J executes unlock function, the task I waiting for lock will run.

Assume that context switching time is large, for example, 1 ms, while the critical section resource is required by the task for much shorter duration, then kernel lock is an inefficient mechanism to lock a critical section. A **spin lock** (Section 7.11.1) can be used to protect the critical section resources as follows.

Assume that task J is being provided the critical section resource through spin lock, s_{lock} . Assume that task I needs the critical section resource at instance t_i . The spin lock concept provides a busy wait loop for I (on executing lock () function). The I goes at t_i into a busy wait loop for the spin lock, s_{lock} . As soon as J releases the s_{lock} (on executing unlock () function at J), the I gets the critical resource without spending time for the context switch.

An implementation of the spin lock in a task can be by a try. The high priority task tries the lock by a wait loop for the lock for a defined time t_{wait} , else the task un-blocks.

Another implementation of the spin lock can be by trying two or four wait loops for the lock with successive decrements in the time t_{wait} to 0, after which the task un-blocks. After the unblocking, the task will run critical section code without the context switch unlike the case when mutex is used to block or un-block a critical section.

A lock function alternative is execution by taking a mutex and lock by releasing the mutex (Sections 7.7.2 and 7.8.3). Another implementation is by an instruction that disables a specific interrupt at the beginning of a critical section and enables the specific interrupt at the end of the critical section.

A lock function can be used before a critical section. Spin lock is effective for the critical section with a short period of execution, because the busy wait loop will be released in a short time. A mutex can be used for critical section that should run exclusively. Disabling and enabling of interrupts can be used to prevent another ISR or process to run in-between.

Example 8.20

(a) RTOS kernels, for example, Windows CE, provide for pre-emption points. These are the OS function codes in-between the critical sections of kernel processes.

(b) RTOS μ COS-II provides function `OS_ENTER_CRITICAL ()` to stop preemption by any task or ISR, as it disables the interrupt. The RTOS provides function `OS_EXIT_CRITICAL ()` to facilitate preemption by a high-priority task or ISR, as it enables interrupt.

(c) μ COS-II provides `OSSchedLock ()` and `OSSchedUnlock ()` for task critical-section locking to run the section and preventing preemption by other task.

Two tasks may have two sections that share the data or resource and only one section must execute. Before the critical section, a task waits for mutex semaphore from the scheduler and releases the mutex semaphore at the exit from the critical section. Provisioning for priority inheritance permits critical section service by a preemptive scheduler without priority inversion (Section 7.8.5).

Can work be done without the semaphores and/or mutex for the critical sections? Yes, one strategy is disabling and enabling the preemption. The disabling of a preemption means disabling only the task switching flags, or their passing to the task when using shared data and enabling the task switching flags to change and pass again after this. But it should then be ensured that all the ISRs are the reentrant functions. Another

strategy could be the use of resources locking semaphore, *mutex* (Sections 7.7.2 and 7.8.2) for spin-lock (Section 7.11.1).

8.10.4 Model for Critical Section Service by a Preemptive Scheduler

Figure 8.9 shows a Petri net concept-based model which models and helps in designing the codes for a task that has a critical section in its running. The figure shows places by the circles and transitions by the rectangles. The following are the places and transitions.

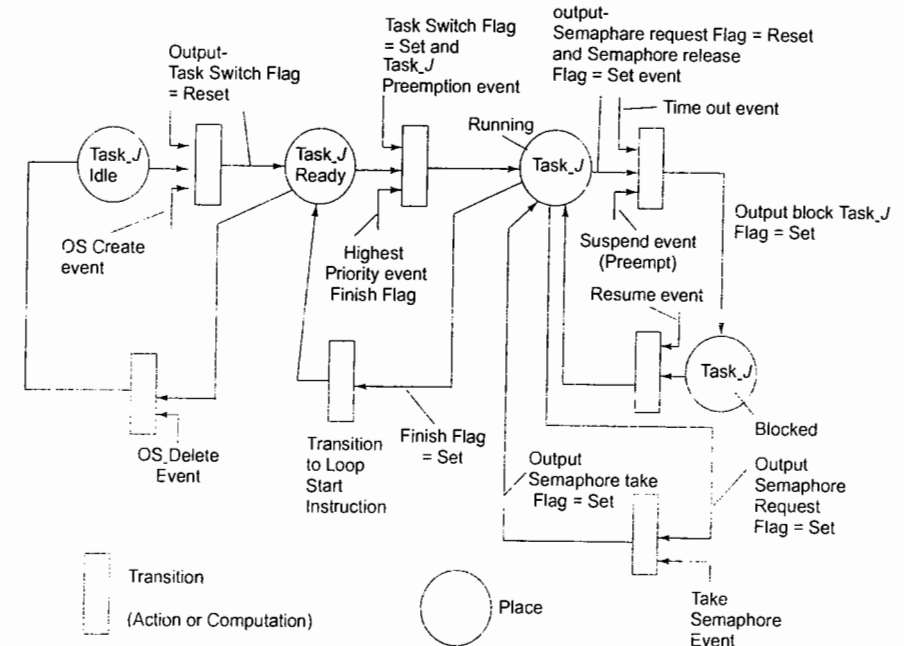


Fig. 8.9 The Petri net model for the task with a preemptive scheduler and one critical section where it takes a semaphore and release on critical section over

1. Each task is in the idle state (at `idleTaskPlace`) to start with, and a token to the RTOS is `taskSwitchFlag = reset`.
2. Consider the `task_J_Idle` place, which currently has highest priority among the ready tasks. When the RTOS creates `task_J`, the place `task_J_Idle` undergoes a transition to the ready state, `task_J_Ready` place. The RTOS initiates `idle` to `ready` transition by executing a function, `task_J_create ()`. A transition from the idle state of the task is fired as follows. RTOS sends two tokens, `RTOS_CREATE` Event and `taskJSwitchFlag`. The output token from the transition is `taskSwitchFlag = true` (refer to the top-left transition in the figure).
3. When after `task_J` finishes, the RTOS sends an `RTOS_DELETE` event (a token) to the task, it returns to `task_J_Idle` place and its corresponding `taskJSwitchFlag` resets (refer to the bottom-left transition in the figure).

4. At task_J_Ready place, the scheduler takes the priority parameter into account. If the current task happens to be of the highest priority, the scheduler sets two tokens, *taskSwitchFlag* = true (sends a token) and *highest Priority Event* = true, for the transition to the running task *J* place, task_J_Running. The scheduler also resets and sends the tokens, task switch flags, for all other tasks that are of lesser priority. This is because the system has only one CPU to process at an instant.
5. From the task_J_Running place, the transition to the task_J_Ready place will be fired when the task finish flag sets (refer to the bottom-middle transition in the figure).
6. At task_J_Running place, the codes of the switched task *J* are executed (refer to the top-right most transition in the figure).
7. At the runningTaskPlace, the transition for pre-empting will be fired when RTOS sends a token, *suspendEvent*. Another enabling token if present, *time_out_event* will also fire the transition. An enabling token for both situations is the semaphore release flag, which must be set. Semaphore release flag is set on finishing the codes of task *J* critical sections. On firing, the next place is task_J_Blocked. Blocking is in two situations. One situation is of preemption. It happens when the *suspendEvent* occurs on a call at the runningTaskPlace asking the RTOS to suspend the running. Another situation is a time out of an SWT that associates with the running task place.
8. On a *resumeEvent* (a token from RTOS) the transition to task_J_Running place occurs (refer to the right-side middle transition, which is between the three transitions that are shown in the figure).
9. At the task_J_Running place, there is another transition that fires so that the task *J* is back at the task_J_Running place when the RTOS sends a token, *take_Semaphore_Event* to ask the task *J* to take the semaphore (the RTOS sets the semaphore request flag, *take_Semaphore_Event*; it resets semaphore release flag; it directs task *J* to run un-interrupted. Do not block).
10. There can be none or one or several critical sections. During the execution of a critical section, the RTOS resets the semaphore release flag and sets the take semaphore event token.

8.10.5 Earliest Deadline First (EDF) precedence and Rate Monotonic Schedulers (RMS) Models

The event-driven schedulers are required for real-time scheduling in case of a number of tasks being large or in case of aperiodic or sporadic tasks. Aperiodic task is one in which the period of occurrence is not known because it may not be known when an event can occur. For example, an event of receiving a phone call is aperiodic event. Sporadic task has periods of bursts when the task events occur.

A deadline is the period in which a task must finish. A task, which has a least deadline that is which has little time left for completion, must be scheduled first. This algorithm of the scheduler is known as EDF algorithm.

EDF precedence When a task becomes ready, it will be considered at a scheduling point. The scheduler does not assign any priority. It computes the deadline left at a scheduling point. Scheduling point is an instance at which the scheduler blocks the running task and re-computes the deadlines and runs the EDF algorithm and finds the task to be run.

An EDF algorithm can also maintain a priority queue based on the computation when the new task inserts. When the number of tasks becomes large, the computation complexity increases for insertion into the queue. Another EDF algorithm can also maintain two or more priority queues based on the relative deadlines and the scheduler inserts the new task into one of the queues.

Precedence Assignment in the Scheduling Algorithms The best strategy is one, which is based on EDF precedence. Precedence is made the highest for a task that corresponds to an interrupt source, which occurs at the earliest and which deadline will finish the earliest.

How is the precedence assigned in the case of variable CPU loads for the different tasks and variable EDFs? One method is as follows.

Let t_1 be the instance when task *J* needs preemption the first time and t_2 be the next instance. A task with minimum $(t_2 - t_1)$ is inserted at the top of the task priority list. It is assigned the highest precedence. The list is dynamically ordered according to $(t_2 - t_1)$.

First, there is a deterministic or static assignment of the precedence in advanced scheduling algorithms. It means first there is RMS. Later on the scheduler dynamically assigns and fixes the time out delays afresh, and assigns the precedence as per the EDF. The need for the dynamic assignment arises due to the sporadic tasks and the distributed or multiprocessor indeterminate environment.

Resource sharing among the tasks creates a problem. The algorithm has to ensure that none of them misses the deadline.

A task occurring at a higher rate should then get higher precedence in case of a periodic tasks. Assume that the data is being received from multiple channels and some channel receive data at a faster rate than the others. A scheduler uses rate monotonic algorithm (RMA) to schedule the tasks in this case.

Rate Monotonic Scheduler RMS computes the priorities, p , from the rate of occurrence of the tasks. The i -th task priority, p_i is proportional to $(1/t_i)$ where t_i is the period of the occurrence of the task event. RMA gives an advantage over EDF because most RTOSes have provisions for priority assignment. Higher-priority tasks always get executed.

RMA disadvantage is that it does not support aperiodic and sporadic tasks. When a burst occurs, even due to higher rate of arriving of the sporadic task in the burst period, it cannot be assigned high priority. The aperiodic and sporadic tasks can be assigned the tickets by aperiodic and sporadic servers in the scheduler. Ticket means the periods in which the events from them will be scheduled.

RMA disadvantage is that a task may have long periods, but can be very critical. It will be assigned least priority. A solution is to divide the very critical task into two or more tasks to raise their allocated priority by the RMA.

8.10.6 Fixed (Static) Real-Time Scheduling Model

The slice-time scheduling method described in Section 8.10.2 is a special case of 'fixed real-time scheduling'. Every task is allotted fixed schedules to run. Let there be m tasks and m real-time clock interrupts, the scheduler can thus assign each task a fixed schedule. Each task undergoes a ready place to running place transition on the timeouts of the corresponding timer. The OS is supposed to define hard real-time schedules for each task.

A scheduler is said to be using a fixed-time scheduling method when the schedule is static and deterministic. The working environment is unaltered when processes are scheduled on the single CPU of the system. Schedules are deterministic as the worst-case latencies for all the interrupts and the tasks are predetermined. The OS scheduler can thus schedule each task at fixed times so that none misses its deadline (this is when the worst-case latency of each task is less than its deadline for its service). The 'no deadline miss' advantage is feasible only in deterministic situations. Coding for the tasks are such that execution times do not vary under the different inputs or different conditions.

Schedules once defined remain static in a fixed-time scheduler. Fixed schedules can be defined by one of the three methods.

1. *Simulated annealing method.* Here the different schedules can be fixed and the performance simulated. Now, schedules for the tasks are gradually incremented by changing the interrupt timer settings (using a corresponding OS function) till the simulation result shows that none is missing its deadline.
2. *Heuristic method.* Here, reasoning or past experience helps to define and fix the schedules.
3. *Dynamic programming model.* This is as follows: a specific running program first determines the schedules for each task and then the timer interrupt loads the timer settings from the outputs from that program.

If the scheduler cannot fix the schedules, it is a non-deterministic situation. An example is a situation in which a message for a task is expected in a network, from another system and the minimum and maximum periods for receiving it are unknown. Another example is when the inputs for a task are expected from another system and the minimum and maximum periods when the inputs will be received are not known.

A dynamic scheduling model is as follows: the software design may be such that the priorities can be rescheduled and fixed times redefined when a message or error message is received during the run.

8.10.7 Latency and Deadlines as Performance Metric in Scheduling Models For Periodic, Sporadic and Aperiodic Tasks

An RTOS should quickly and predictably respond to the event. It should have minimum interrupt latency and fast context switching latency.

Different models have been proposed for measuring performances. Three performance metrics are as follows.

1. Ratio of the sum of interrupt latencies with respect to the sum of the execution times.
2. CPU load.
3. Worst-case execution time with respect to the mean execution time.

'Interrupt latencies' in various task models can be used for evaluating performance metrics. The latencies for various task scheduling models are described in Sections 8.10.1 to 8.10.3. The CPU load is another way to look at the performance. It is explained in Section 8.10.8. Worst-case performance calculation for a sporadic task is explained in Section 8.10.9. 'Refer Real Time Systems' by Jane W. S. Liu, Pearson Education, 2000, for details of many models available for evaluating the performances.

8.10.8 CPU Load as Performance Metric

Each task gives a load to the CPU that equals the task execution time divided by the task period. [Task period means period allocated for a task.] Recall In_AOut_B intranetwork of Example 4.1. Receiver port A expects another character before 172 μ s. Task period is 172 μ s. If the task execution time is also 172 μ s, the CPU load for this task is 1 (100%). The task execution time when a character is received must be less than 172 μ s. The maximum load of the CPU is 1 (less than 100%).

The CPU load or system load estimation in the case of multitasking is as follows. Suppose there are m tasks. For the multiple tasks, the sum of the CPU loads for all the tasks and ISRs should be less than 1. The time outs and fixed-time limit definitions for the tasks reduce the CPU load for the higher-priority tasks so that even the lower-priority tasks can be run before the deadlines. What does it mean when the sum of the CPU loads equal to 0.1 (10%)? It means that the CPU is underutilized and spends 90% of its time in a waiting mode. As the execution times and the task periods vary, the CPU loads can also vary.

When a task needs to run only once, then it is aperiodic (one shot) in an application. Scheduling of the tasks that need to run periodically with the fixed periods can be periodic and can be done with a CPU load very close to 1. An example of a periodic task is as follows. There may be inputs at a port with predetermined periods, and the inputs are in succession without any time gap.

When a task cannot be scheduled at fixed periods, its schedule is called sporadic. For example, if a task is expected to receive inputs at variable time gaps, then the task schedule is sporadic. An example is the packets from the routers in a network. The variable time gaps must be within defined limits.

A preemptive scheduler must take into account three types of tasks (aperiodic, periodic and sporadic) separately.

1. An aperiodic task needs to be preempted only once.
2. A periodic task needs to be preempted after the fixed periods and it must be executed before its next preemption is needed.
3. A sporadic task needs to be checked for preemption after a minimum time period of its occurrence. Usually, the strategy employed by the software designer is to keep the CPU load between (0.7 ± 0.25) for sporadic tasks.

8.10.9 Sporadic Task Model Performance Metric

Let us consider the following parameters.

T_{total} is the total length of the periods for which sporadic tasks occur; e is the total task execution time; T_{av} is the mean periods between the sporadic occurrences; T_{min} is the minimum period between the sporadic occurrences.

Worst-case execution time performance metric, p is calculated as follows for the worst case of a task in a model.

$$p = p_{\text{worst}} = (e * T_{\text{total}} / T_{\text{av}}) / (e * T_{\text{total}} / T_{\text{min}}).$$

It is because the average rate of occurrence of sporadic task is (T_{total} / T_{av}) and the maximum rate of sporadic task burst is T_{total} / T_{min} .

There are various models to define a performance metric. Three performance metrics for schedule management by the RTOS are: (i) interrupt latencies with respect to the execution times, (ii) CPU load and (iii) worst-case execution time.

8.11 OS SECURITY ISSUES

When a doctor has to dispense to multiple patients, protection of the patients from any confusion in the medication becomes imperative. When an OS has to supervise multiple processes and their access to the resources, protection of memory and resources from any unauthorized writes into the PCB or resource, or mix up of accesses of one by another becomes imperative. The OS security issue is a critical issue.

Each process determines whether it has a control of a system resource exclusively or whether it is isolated from the other processes, or whether it shares a resource common to a set of processes. For example, a file or memory blocks of a file will have exclusive control over a process and a free memory space will have the access to all the processes. The OS then configures when a resource is isolated from one process and a resource is shared with a defined set of processes.

The OS should also have the flexibility to change this configuration when needed, to fulfil the requirements of all the processes. For example, a process has control of 32 memory blocks at an instance and the OS configures the system accordingly. Later when more processes are created, this can be reconfigured.

The OS should provide protection mechanisms and implement a system administrator(s)-defined security policy. For example, a system administrator can define the use of resources to the registered and authorized users (and hence their processes).

What about issues of an *application* changing the OS configuration? The OS needs a protection mechanism for itself. An application software programmer can find a hole in the protection mechanism and gain an unauthorized access. Thus the implementation of protection mechanisms and enforcement of security policy for resources is a challenging issue before any OS software designer. The network environment complicates this issue.

Table 8.13 gives the various activities for implementing important security functions.

Table 8.13 Important Security Functions

Function	Activities
Controlled resource sharing	Controlling read and write of the resources and parameters by user processes. For example, some resources write only for a process and some read only for a set of processes.
Confinement mechanism	Mechanism that restricts sharing of parameters to a set of processes only.
Security policy (strategy)	Rules for authorizing access to the OS, system and information. A policy example is a communication system having a policy of peer-to-peer communication (connection establishment preceding the data packets flow).
Authentication mechanism	External authentication mechanism for the user and a mechanism to prevent an application run unless the user is registered and the system administrator has (software) authorized. Internal authentication for the process, and the process should not appear (impersonate) like other processes. User authentication can become difficult if the user disseminates passwords or other authentication methods.
Authorization mechanism	User or process allowed using the system resources as per the security policy.
Encryption	A tool to change information to make it unusable by any other user or process without the appropriate key for deciphering it.

The OS security issues are important considerations. Protection of memory and resources from any unauthorized write into the PCB or resource, or mix up of accesses of one by another, becomes imperative for an OS security and protection mechanism.



Summary

- A kernel is a basic unit of any OS that includes the functions for memory allocation and de-allocation, preventing unauthorized memory access, tasks scheduling, IPC, I/O management, interrupts-handling mechanism and device drivers and management. The OS also controls I/O and network subsystems. An OS kernel may also have file management functions and functions for network subsystems (these functions may also be separate from the OS kernels in certain OSES).
- An RTOS has functions for real-time task scheduling and interrupt latency control. The RTOS uses the time, μs and system clocks. Its functions include time allocation and de-allocation to attain the best utilization in presence of timing constraints to the tasks.
- RTOS uses asynchronous IO functions so that the tasks do not block during the IOs.

- RTOS uses the *preempting scheduling* so that a lower-priority task is forced to block by the scheduler to let a higher-priority task run. Time slicing means that each task is allotted a time slice after which it blocks and waits for its turn on the next cycle. In certain cases, other strategies for scheduling, for example, the cooperative scheduling of the multiple tasks as per interrupt sequences, and time slicing can be used.
- An RTOS has functions for handling interrupts. There are three alternative ways used in the RTOSes for response to hardware source calls from the interrupts. An RTOS has functions for registering and deregistering a device driver and also facilitates concurrent processing of the modules, devices, ISRs and tasks.
- An RTOS has mutex and spin lock functions for critical section handling in priority scheduling cases. RTOS may also provide for fixed real-time scheduling.
- RTOS has synchronization mechanism for the tasks using the IPCs and predictable timing and synchronization behaviour in the system. RTOS provide for IPC functions (signals, semaphores, mutexes, queues, mailboxes, pipes and sockets). There is standardization (e.g., POSIX 1003.b) of the RTOS and IPC functions.
- Relative *timing* of the various actions in a preemptive scheduler helps us to optimize the process timings in an application developed using an RTOS.
- RTOS uses the task running in the kernel space to let the code execute in the supervisory mode and thus execute fast (due to no instructions for memory leak and stack overflow checks and kernel space protection check) and reduce the interrupt latencies.
- RTOS uses fixed memory block allocation with predictable memory allocation and de-allocation time.
- OS security issues are important considerations in a system and the protection of memory and resources or PCB from any unauthorized write is essential.



Keywords and their Definitions

- Asynchronous IOs* : IOs in which a process is not blocked due to IO.
- Cooperative scheduling* : A waiting task lets another task run till it finishes.
- Critical section run* : In spite of higher priority, a critical section is allowed to run by a scheduler using the semaphore or spin-lock or lock functions. The critical section is used for shared resources and data between multiple tasks.
- Cyclic round robin scheduling* : A scheduling algorithm in which the tasks are cyclically scheduled in sequence from a list of ready tasks. A time slice is provided in case of round robin cycle.
- Fixed real-time scheduling* : A scheduling strategy in which the time for each task is fixed.
- Hard real time system* : A system in which no task should delay and miss the deadline, the system have minimal interrupt latencies and well-defined time-constraints for each task.
- Kernel* : A basic unit of any OS that includes the functions for processes, memory, task scheduling, IPC, management of devices, IOs and interrupts and may include the file systems and network subsystems in certain OSES.
- Lock* : A function to lock the availability of resources to other tasks at beginning of critical section codes.
- OS* : A system having basic kernel functions of process and memory management, file, IO, device and network management functions and many other functions also.
- Pre-empting scheduling* : A scheduling algorithm in which a higher-priority task is forced (pre-empted) to block by the scheduler to let a higher-priority task run.
- Protection mechanism* : A mechanism at an OS to protect against unauthorized accesses to the resources.

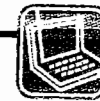
- Rate monotonic scheduling** : A scheduling in which tasks are assigned priorities in accordance with their rate of occurrences of need of their services.
- RTOS** : OS for soft or hard real time tasks with task scheduling with real-time constraints (deadlines) using priority based scheduling, interrupt-latency control, synchronization of tasks with IPCs, and predictable timing and synchronization behaviour of the system.
- Round robin** : A scheduling algorithm in which tasks are scheduled with each one allotted a time slice and run in cyclic in sequence.
- Soft real-time system** : A system in which most not all tasks meet the time constraints and do not miss deadline and a miss can be handled with some delay.
- Spin lock** : An implementation of the spin lock in a task can be by a try. The task tries the lock by a wait loop for the s_{lock} for a defined time t_{wait} , else the task un-blocks for the s_{lock} .
- Time slicing scheduling** : It is also called round robin scheduling. A scheduling algorithm in which each task is allotted a time slice after which it is blocked and waits for its turn on the next cycle.
- Unlock** : A function to release the lock at end of the critical section.



Review Questions

1. What should be the goal of an OS?
2. List the layers between *application* and *hardware*.
3. Why does an OS function provide two modes, *user mode* and *supervisory mode*?
4. List the functions of a kernel. What can be the functions outside the kernel?
5. Explain the terms process descriptor and process control block (PCB). What are the analogies in a PCB and TCB?
6. When is a message used and when does a system call for seeking access to system resources?
7. Process or task creation and management are the most important functions of the kernel. Why?
8. A strategy is that the tasks are created at start-up only and creating and deleting tasks later is avoided. Why should it be adopted?
9. Memory allocation and management are the most important functions of the kernel. Why? How does memory allocation differ in RTOS and OS? What is memory locking?
10. List the advantages and disadvantages of fixed and dynamic block allocations by the OS.
11. The kernel controls the access of system resources, CPU, memory, IO subsystems and devices. Why is it needed? Explain the critical section handling with mutexes and spin locks.
12. What is the importance of device management in an OS for an embedded system?
13. Give examples of IO subsystems. Explain the use of asynchronous IOs.
14. Define a network OS. How does a network OS differ from a conventional OS?
15. What are the uses of OS interoperability and portability?
16. How do you choose scheduling strategy for the periodic, aperiodic and sporadic tasks?
17. What are the OS functions at an RTOS kernel?
18. When do you use cooperative scheduling and when preemptive?
19. Compare two scheduling strategies for the real-time scheduling – preemptive mode and round robin scheduling.

20. What are the cases in which time-slice scheduling helps?
21. List three ways in which an RTOS handles the ISRs in a multitasking environment. What is the advantage of two- or three-level handling of the interrupts? Explain IST.
22. How does a preemption event occur?
23. Real-time system performance metrics are throughput, interrupt latencies, average response times and deadline misses. Explain the importance of each of these metrics.
24. Why should you estimate worst-case latency?
25. Explain the applications of simulation annealing method.
26. What should be the OS security policy?
27. What is the protection mechanism for the OS?
28. OS security issues are important considerations. Protection of memory and resources from any unauthorized write into PCB or resource or mix up of accesses of one by another becomes imperative for an OS security and protection mechanism. Explain each of these considerations.
29. What do you mean by hierarchical RTOS?
30. How is the precedence assignment done for the tasks? How is the precedence assignment algorithm used in dynamic programming?
31. List the best strategies for synchronization between the tasks and ISRs.
32. What is dynamic program scheduling?



Practice Exercises

33. Give two examples when the scheduler cannot fix the schedules and there is a non-deterministic situation.
34. How do you estimate CPU load in a multitasking system handling sporadic tasks?
35. When do you use CPU load for performance metrics of a real-time system?
36. Give a table showing the differences between traditional OS and the RTOS.
37. Show how the timer functions can be used: (i) to reduce the light level in a mobile phone with full brightness, (ii) to switch off the LCD display in a mobile phone after 15 seconds from the time it was switched on.
38. Show how will you use the mailboxes between display task and other tasks. Which one should you prefer, use of semaphore as shown in the example or use of mailbox.
39. Consider a mobile phone cum PDA device and look at the main menu. Explain how the events of touching the screen at different points on the screen are handled by an RTOS using two-level ISR handling.
40. Give a table showing the differences between three methods of ISR handling in the RTOSs.
41. Show the use of 15 points for the principles of RTOS-based design by taking the example of ACVM.
42. List the priority allocations in ACVM tasks.
43. Show the use of 15 points for the principles of RTOS-based design by taking the example of digital camera.
44. Give the priority allocations in camera tasks.
45. Show the use of 15 points for the principles of RTOS-based design by taking the example of mobile phone device.
46. Give the priority allocations in phone tasks.
47. Show the scheduling method that RTOS can use in case of the VoIP router.
48. Give the priority allocations in smart card tasks.
49. Show the use of semaphores for synchronizing the tasks as cooperative scheduled tasks in a preemptive RTOS.
50. Show the use of semaphores and timer functions for synchronizing the tasks as round robin time-sliced scheduled tasks in a preemptive RTOS.

REAL-TIME OPERATING SYSTEM PROGRAMMING-I: MicroC/OS-II and VxWorks

9

R

We have learnt the following important points relating to the traditional OS and RTOS.

e

• Process is that computational unit which an OS schedules and on request, either by system call or by message passing, the OS lets the process use the resources: CPU, memory, IO subsystems, flash-memory file system, network subsystems and device drivers. Process also means 'task' in a multitasking model of the processes and means 'thread' in a multithreading model of the processes, both controlled by the OS. A process can also consist of several threads, which share a common process structure.

c

• System structure consists of application software, APIs, additional system software than the one provided at the OS, OS interface, OS, hardware-OS interface and hardware, and an OS or RTOS.

a

• The application software and APIs are programmed for processes (tasks) and ISRs, ISTs.

[

[

R

• The priorities when executing codes are first the ISRs, then ISTs and then threads of the processes.

e

• The basic functions (services) of the OS are process management (also means thread management or task management) from creation to deletion, processing resource requests, memory management from allocation and de-allocation, process scheduling, processing and managing IPC (communication among the ISRs, tasks and OS functions), IO subsystems management, management of the file, IO, device, and device drivers and functions for enabling sharing of resources and data.

c

• Handling of interrupts and scheduling of tasks by the RTOS.

• RTOS has the basic functions of the OS plus functions for real-time task scheduling and interrupt latency control. RTOS uses the timers and system clocks, time allocation and de-allocation to attain best utilization of the CPU time under the given timing constraints for the tasks.

a

• RTOS provides a predictable timing behaviour of the system (in most cases) and a predictable task synchronization using the priorities allocation and priorities inheritance. RTOS provides for synchronization of ISRs, ISTs and tasks using the IPCs for the hard and soft real-time operations. RTOS provides for asynchronous IOs.

[

• Interprocess synchronization during concurrent processing of the tasks takes place through signals, semaphores, queues, mailboxes, pipes, sockets, RPCs, timer and event functions.

• Basic strategies for scheduling the multiple tasks are pre-empting, round robin time slice and cooperative scheduling. The RTOS basic strategy is preemptive scheduling.

• Principles of basic design using the RTOS, and important points that are taken care of during coding for synchronization between the processes (ISRs, functions, tasks and scheduler functions).

[

The goals of any embedded software, and hence of RTOS, are perfection and correctness. The reader must have now realized that there is a great deal of functions involved in real-time programming. The objective of this chapter is to explain thoroughly the two popular RTOSes that are used for programming and provide the OS functions which, significantly reduce the time required to design an embedded system.

L
E
A
R
N
I
N
G
O
B
J
E
C
T
I
V
E
S

We will learn the following in this chapter.

- (i) Basic functions and types of RTOSes.
- (ii) RTOS μ COS-II (referred as MUCOS in the text) through 20 examples— Examples 9.1 to 9.20. What arguments are passed and what values are returned for each given MUCOS function will be explained. Learning the use of functions in the MUCOS is important for a reader even if another RTOS is used later. This will help greatly in understanding the advanced, sophisticated embedded RTOSes later on.
- (iii) VxWorks from Wind River[®] Systems is also an RTOS for sophisticated embedded systems. It has powerful tool support. The features in VxWorks are explained by seven examples— Examples 9.21 to 9.27. Differences between the VxWorks semaphores, mailboxes and queues with respect to that of MUCOS will be made clear.

Chapter 10 will describe the Windows CE, OSEK and real-time Linux (RTLlinux).

9.1 BASIC FUNCTIONS AND TYPES OF RTOSes

A complex multitasking embedded system design requires the following:

1. Integrated development environment
2. Task functions in embedded C or embedded C++
3. Real-time clock-based hardware and software timers
4. Scheduler
5. Device drivers and device manager
6. Functions for IPCs using the signals, event flag group, semaphore-handling functions and functions for the queues, mailboxes, pipe and sockets.
7. Additional functions, for example, TCP/IP or USB or Bluetooths or WiFi or IrDA and GUIs.
8. Error and exception handling functions.
9. Testing and system debugging software for testing RTOS as well as developed embedded application.

Figure 9.1(a) shows the basic functions expected from the kernel of an RTOS. The RTOS's have the following features in general.

1. Basic kernel functions and scheduling: pre-emptive or pre-emptive plus time slicing.
2. Priorities definitions for the tasks and IST.
3. Priority inheritance feature with option of priority ceiling feature.
4. Limit for number of tasks.
5. Task synchronization and IPC functions.
6. IDE consisting of editor, platform builder, GUI and graphics software, compiler, debugging and host target support tools.
7. Device imaging tool and device drivers.

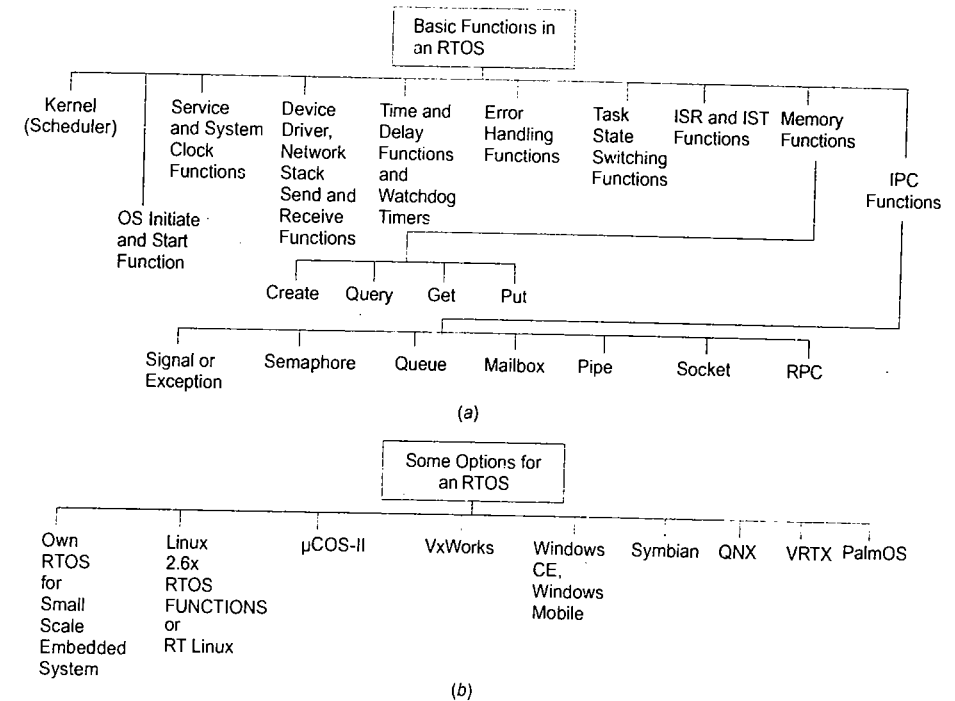


Fig. 9.1 (a) Basic functions expected from the kernel of an RTOS (b) Common options available for selecting a real-time operating system (RTOS)

8. Support to the clock, time and timer functions, POSIX, asynchronous IOs, memory allocation and deallocation system, file systems, flash systems, number of processors, TCP/IP, network, wireless and bus protocols, development environment with Java and componentization (reusable modules for different functions), which leads to small footprint (small-sized RTOS codes placed in the ROM image).
9. Support to number of processor architectures.

Section 9.1.1 gives the host-target and self-host approaches to development of an application. Section 9.1.2 gives the types of RTOSes.

9.1.1 Host and Target-Based, and Self-Host-Based Development Approaches

A real-time or non-real-time application-development approach is the **host target approach**. A host machine (computer) for example, a PC uses a general purpose OS, for example, Windows or Unix for system development. The target connects by a network protocol, for example, TCP/IP during the development phase. The developed codes and the target RTOS functions first connect a target. The target with downloaded codes finally disconnects and contains a small-size footprint of RTOS. For example, the target does not download host machine resident compiler, cross-compiler, editor for programs, simulation and debugging programs and MMU support.

The **self-host development approach** is that the same system with full RTOS is used for development on which the application runs. This also does not require cross-compilation. When application codes are ready, the required RTOS function codes and application codes are downloaded into the ROM of the target board.

9.1.2 Types of RTOSes

Some options for selecting an RTOS are shown in Figure 9.1(b). Followings are the types of RTOSes.

In-House Developed RTOS In-house RTOS has the codes written for the specific need, and application or product and customizes the in-house design needs. Generally either a small-level application developer uses the in-house RTOS or a big research and development company uses the codes built by the in-house group of engineers and system integrators.

Broad-based Commercial RTOS A readily available broad-based commercial RTOS package offers the following advantages.

1. Provides an advantage of availability of off the self thoroughly tested and debugged RTOS functions.
2. Provides several development tools. Development tools consist of tools for the source-code engineering, testing, simulating and debugging are also available with the RTOS package. When designing a mission critical real-time application, lack of appropriate error-handling capability or an appropriate RTOS or a testing and debugging tool causes data loss. Even hardware loss may be caused.
3. Support to many processor architectures, for example, ARM as well as x86, MIPS and SuperH.
4. Support to development of GUIs in the system.
5. Support to many devices, graphics, network connectivity protocols and file systems.
6. Support to device software optimization (DSO). It is a recently available concept in a few RTOSes.
7. Provides error and exceptional handling functions, which can be ported directly as these are already well tested by thousands of users.
8. Not only simplifies the coding process greatly for a developer but also helps in building a product fast: it aids in building robust and bug-free software by thorough testing and simulation before locating the codes into the hardware.
9. Saves large amount of development time for RTOS tools and in-house documentation. Saving of time results in little time to market an innovative and new product.
10. Saves maintenance cost.
11. Saves cost of keeping in-house engineers.

General Purpose OS with RTOS Embedded Linux or Windows XP is general purpose OS. They are not componentized. Footprint (the code that goes as ROM image) is not reducible. The tasks are not assignable priorities. They offer powerful GUIs, rich multimedia interfaces and have low cost.

The general purpose OS can be used in combination with the RTOS functions. For example, RTLinux is a real-time kernel over the Linux kernel. Another example is 'Windows XP Embedded' for x86 architecture.

Special Focus RTOS Special focus RTOS is used with specific processors like ARM or 8051 or DSP, for example, OSEK for automotives or Symbian OS for the mobile phones.

9.2 RTOS μ COS-II

One popular RTOS for the embedded system development is μ COS-II. For non-commercial use, RTOS μ COS-II is also a freeware. Jean J. Labrosse designed it in 1992 and nowadays μ COS-II is well developed for

a number of applications. It is available from Micrium (www.micrium.com). Its name μ COS-II is derived from Micro-Controller Operating System. It is also popularly known as MUCOS or MicroCOS or UCOS. (*Note: MUCOS is pronounced as MU-C-OS*)

Micrium describes MUCOS as portable, ROMable, scalable, preemptive, real-time and multitasking kernel. MUCOS has been used in over thousands of applications, including automotive, avionics, consumer electronics, medical devices, military, aerospace, and networking, and systems-on-a-chip development. MUCOS has an elegant code and is said to offer best high quality/performance ratio. Its source code has been certified by Department of Defense, USA for use in Avionics and in medical applications.

It has a precertifiable software component for safety critical systems, including avionics system ADO-178B and EUROCAE ED-12B, medical FDA 510(k) and IEC 61058 standards for transportation and nuclear systems, respectively.

Using this RTOS has another advantage. It has full source code availability and has been elegantly and very well documented in the book by its designer (refer to the printed book references in Appendix 2).

MUCOS codes are in C and a few CPU-specific modules are in the assembly. Its code ports on many processors that are commonly used in the designing of embedded systems. MUCOS is real-time kernel with additional support as follows.

1. μ C/BuildingBlocks [an embedded system building blocks (software components) for hardware peripherals, e.g., clock (μ C/Clk) and LCD (μ C/LCD)].
2. μ C/FL (an embedded flash memory loader).
3. μ C/FS (an embedded memory file system).
4. μ C/GUI (an embedded GUI platform).
5. μ C/Probe (a real-time monitoring tool).
6. μ C/TCP-IP (an embedded TCP/IP stack).
7. μ C/CAN (an embedded controller area network bus).
8. μ C/MOD (an embedded modbus).
9. μ C/USB device and μ C/USB host (embedded USB-devices framework).

Source Files MUCOS has 10,000 plus lines of codes. There are two types of source files. Master header file includes the '#include' preprocessor commands for all the files of both types. It is referred as 'includes.h'. Every C file has the command '#include 'INCLUDES.H'.

1. *Processor-dependent source files:* Two header files at the master are the following: (i) `os_cpu.h` is the processor definitions header file. (ii) The kernel building configuration file is `os_cfg.h`. Further, two C files are for ISRs and RTOS timer, specifying `os_tick.c` and processor C codes `os_cpu_c.c`. Assembly codes for the task switching functions are at `os_cpu_a.s12` (for 68HC12 microcontroller). For other microcontrollers, there are similar assembly code files, for example, `os_cpu_a.s51` for 8051.
2. *Processor-independent source files:* Two files, MUCOS header (included in master) and C files, are `ucos_ii.h` and `ucos_ii.c`. The files for the RTOS core, timer and task files are `os_core.c`, `os_time.c` and `os_task.c`. The memory-partitioning, semaphore, queue and mailbox codes are in `os_mem.c`, `os_sem.c`, `os_q.c` and `os_mbox.c`, respectively.

A feature of MUCOS is adaptation of a systematic naming convention that helps program design with clear understanding of the code. The naming convention is as follows.

- a. OS or OS_ (OS followed by underscore) when used as a prefix denotes that the function or variable is a MUCOS-operating system function or variable. For example, `OSTaskCreate()` is a MUCOS function that creates a task. `OS_NO_ERR` is a MUCOS macro that returns true in case no error is reported from an OS function. `OS_MAX_TASKS` is a constant for the maximum number of tasks in the user application (The user in the preprocessor definitions defines this constant).

- b. MUCOS is scalable OS. (Only the OS functions that are necessary become part of the application codes, thus having reduced memory requirements). The functions needed for task servicing, IPC and so on must be predefined in a configuration file included in the user codes (Refer to Example 9.7 Steps 1 and 2 for a clear understanding of configuration-setting codes).
- c. For multitasking, MUCOS employs a preemptive scheduler (Section 8.10.3).
- d. MUCOS has system-level functions. These are for RTOS initiation and start, system clock ticks (interrupts) initiation and the ISR enter and exit functions. (Section 9.2.1 Table 9.1) For the critical section, MUCOS has (i) interrupts disabling and enabling functions that execute at entering and exiting the section, respectively, (ii) lock and unlock functions in kernel that execute at entering and exiting the section, respectively, and that do not disable the interrupts, and (iii) semaphore functions, which can be used as mutex functions that execute at entering and exiting the critical section, respectively, and that disables the preemption by a higher priority task, which is sharing that mutex in its critical section (Refer to Sections 7.7.2 and 8.10.3).
- e. MUCOS has task service functions (e.g., task creating, running, suspending and resuming) (Table 9.2).
- f. MUCOS has task delay functions (Table 9.3).
- g. MUCOS has memory allocation functions for creating and partitioning into blocks, getting a block, putting into the block and querying during debugging at a block (Table 9.4).
- h. MUCOS has IPC functions. These are as per Tables 9.5, 9.6 and 9.7, respectively. MUCOS IPCs use the semaphores, queues and mailboxes (Sections 7.11 to 7.13).
- i. MUCOS has semaphore functions, which are usable like the event-signalling flags, shared resource acquiring keys or counting semaphores (Section 7.7.5). Table 9.5 lists these.
- j. MUCOS has event flag-group functions (Section 8.4) also, where a task waits for any or all of the flags setting in an event flag groups. These functions enable event signalling from OR or AND operation for WAIT_ANY or WAIT_ALL operation on number of flags set in the group. An ISR or task can set a flag in the group. An ISR is not allowed to wait (pend) for the event(s).
- k. MUCOS has mailbox functions. A MUCOS mailbox has one message pointer per mailbox. (Section 7.13). There can be any number of messages as MUCOS sends only the pointer (start address of the message) into the mailbox. Table 9.6 lists these.
- l. MUCOS has queue functions. A queue permits an array of message pointers per queue from which messages retrieve in the FIFO method (Section 7.12). There can be any number of messages in a queue element as MUCOS sends only the address of message into queue. Table 9.7 lists these.

The following seven subsections 9.2.1 to 9.2.7 describe the aforementioned MUCOS functions. For functions in each of the seven Tables, 9.1 to 9.7, these sections give the details of values that are returned by the MUCOS functions and the details of parameters (arguments) that are passed by value or reference to a MUCOS function.

9.2.1 System-Level Functions

We first use initiate function for initiating the use of MUCOS, and then use start function for starting the MUCOS multitasking functions. These functions are OSInit and OSStart, respectively. The start function is used after the creation of at least one task, which can be called Start_Task or First_Task. (As we will read later, a strategy is that remaining tasks are created in the First_Task.)

Recall Section 8.3. We first initiate the system time, clock ticks (and interrupts). MUCOS RTOS has system functions that should be executed when entering and exiting the ISR.

Recall Table 7.1 and Section 7.8.2. MUCOS RTOS has system functions for disabling and enabling interrupts that can be executed when entering a critical section of a task or ISR and exiting the critical section or ISR (Sections 7.7.2 and 8.10.3). Table 9.1 gives these RTOS system-level functions.

Table 9.1 Real-time Operating System (RTOS) and System Clock Initiate, Start, Interrupt Service Routine (ISR) and Disable-Enable Interrupt Functions¹

Prototype Functions	When is this OS Function Called?
void OSInit (void)	At the beginning prior to OSStart ()
void OSStart (void)	After the OSInit () and task-create functions
void OSTickInit (void)	In first task function, which executes once only, this function is to initialize the system timer ticks (system clock interrupts)
void OSIntEnter (void)	Recall Section 8.7.1. Just after the start of the ISR codes OSIntExit must call just before the return from the ISR. ² (enter and exit functions form a pair)
void OSIntExit (void)	After the OSIntEnter () is called just after the start of the ISR codes and OSIntExit is called just before the return from the ISR (enter and exit functions form a pair) ³
OS_ENTER_CRITICAL	Macro to disable interrupts (Section 7.8.2)
OS_EXIT_CRITICAL	Macro to enable interrupts (enter and exit functions form a pair in the critical section)
OSSchedLock ⁴ ()	To lock scheduling of the task (Sections 7.11.1 and 8.10.3)
OSSchedUnlock ⁴ ()	Unlock scheduling of the tasks (Sections 7.11 and 8.10.3)

¹ Functions in this table pass no arguments and returns void.

² There is a global variable, OSIntNesting. It increments after the enter call. (We should not increment directly though it can be done. Let it increment automatically on enter to an ISR).

³ Global variable OSIntNesting decrements on exit call. (We should not decrement directly though it can be done. Let it decrement automatically on exit from an ISR).

⁴ OSSchedlock () enables any task critical section run without preemption if it executes before entering critical section and runs OSSchedUnlock () after end of the task critical section. Task can however be interrupted by an ISR. Task locking the scheduler should not suspend itself before unlocking.

1. *Initiating the OS before starting the use of the RTOS functions.* Function void OSInit (void) is used to initiate the OS. Its use is compulsory before calling any OS kernel functions. It returns no parameter. An exemplary use as a function is as follows:

Example 9.1

```
1. /* Start executing the codes */
void main (void) { 2. /* Initiate MUCOS RTOS to let us use the OS kernel functions */
OSInit ( );
3. /* Create (Define Identity, stack size and other TCB parameters for the tasks using RTOS Functions */
.
.
.
4. /* Create semaphore, queue and mailboxes, etc. */
.
.
.
}
```

2. *Starting the use of RTOS multitasking functions and running the tasks.* Function `void OSStart (void)` is used to start the initiated OS and created tasks. Its use is compulsory for the multitasking OS kernel operations. It returns no parameter. An exemplary use as a function is as follows:

Example 9.2

```
1. /* Start executing the codes from Main*/
void main (void) {2. /* Initiate MUCOS RTOS to let us use the OS kernel functions */
OSInit ( );
3. /* Create tasks and inter-process communication variables by defining their identity, stack size and other
TCB parameters. */
.
.
4. /* Start MUCOS RTOS to let us use RTOS control and run the created tasks and inter-process
communication. */
OSStart ( );
/* An infinite while-loop follows in each task. So there is no return to the main ( ) from the
RTOS. */
}/* End of the Main function. */
```

3. *Starting the use of RTOS System clock.* Function `void OSTickInit (void)` is used to initiate the system clock ticks and interrupts at regular intervals as per `OS_TICKS_PER_SEC` predefined during configuring the MUCOS. Its use is compulsory for the multitasking OS kernel operations when the timer functions are to be used. It returns or passes no parameter. An exemplary use will be shown in steps 2 and 10 of Example 9.7.
4. *Sending message to RTOS kernel for taking control at the start of an ISR.* Function `void OSIntEnter (void)` is used at the start of an ISR. It is for sending a message to RTOS kernel for taking control (Section 8.7.1). Its use is compulsory to let the multitasking OS kernel, control the nesting of the ISRs in case of occurrences of multiple interrupts of varying priorities. It returns no parameter. An exemplary use as a function is as follows:

Example 9.3

```
1. /* Start executing the codes of an ISR*/
ISR_A ( ) {
2. /* sending message to RTOS kernel for taking control of ISR_N from nested ISRs loop. Increment
OSIntNesting, a global variable */
OSIntEnter ( );
3. /* Codes for servicing of the ISR by calling a task. */
.
.
.
5. Sending a message to RTOS kernel for quitting the control at return from an ISR. Function void OSIntExit (void) is used just before the return from the running ISR. It is for sending a message to RTOS kernel for quitting control from the nesting loop. Its use is compulsory to let the OS kernel quit the ISR from the nested loop of the ISRs. It returns no parameter. An exemplary use as a function is as follows:
```

Example 9.4

1. to 3. As in Example 9.3

```
.
.
4. /* Sending message to RTOS kernel for quitting the control of ISR_A from the nested loop. Decrement
OSIntNesting, a global variable */
OSIntExit ( );
}/* End of the ISR function. */
```

6. *Sending a message to RTOS kernel for taking control at the start of a critical section.* Recall Example 8.21 in Section 8.10.3. A macro-function `OS_ENTER_CRITICAL` is used at the start of critical section in a task or ISR. It is for sending a message to MUCOS kernel and disabling the interrupts. Its use is compulsory to let the OS kernel take note of and disable the interrupts of the system. It returns no parameter. An exemplary use as a function is as follows.

Example 9.5

```
1. /* Start executing the codes of a task or an ISR*/
task_A ( ) {
2. /* Codes for servicing of the task. */
.
.
3. /* Sending a message to RTOS kernel and disabling the interrupts. */
OS_ENTER_CRITICAL;
4. /* Run critical section codes as follows. */
.
.
5. /* Codes for Exiting the service*/
}
```

7. *Sending a message to RTOS kernel for quitting the control at the return from a critical section.* Macro-function `OS_EXIT_CRITICAL` is used just before the return from the critical section. It is for sending a message to RTOS kernel for quitting control from the section. Its use is compulsory to let the OS kernel quit the section and enable the interrupts to the system. It returns no parameter. An exemplary use as a function is as follows:

Example 9.6

1. to 4. As in Example 9.5

```
.
.
5. /* Sending a message to RTOS kernel for quitting the control of critical section and enabling the
interrupts. */
OS_EXIT_CRITICAL;
}/* End of the ISR function. */
```

8. *Locking OS scheduler.* OSSchedlock () disables preemption by a higher-priority task. This function inhibits preemption by higher-priority task, but does not inhibit interrupt. If an interrupt occurs then locking enables return of OS control to that task, which executes this function. The control returns to the task after any ISR completes.
9. *Unlocking OS scheduler.* OSSchedUnlock () enables preemption by higher priority task. Enables return of OS control to the high priority task after the execution of OSSchedUnlock. In case of any interrupt occurring after executing OSSchedUnlock and after the end of the ISR, the higher-priority task, which is ready will execute on return from the ISR.

9.2.2 Task Service and Time Functions and their Exemplary Uses

MUCOS service functions for the tasks and time are as per Table 9.2. Service functions mean the functions to create task, suspend and resume, and time setting and time retrieving (getting) functions. Time functions set time and get time in terms of the number of system clock ticks.

The declarations for the variable and prototype assignments for task functions are done in the preprocessor commands. Steps 1 and 2 of Example 9.7 show these preprocessor commands. The codes in Steps 1 and 2 codes are saved in a configuration file, which is included in the source code before compilation. These steps configure the MUCOS before they are used.

We shall see in the following examples that there is an infinite loop in every task function. This is a characteristic way of coding the tasks for preemptive scheduling. From the infinite loop, how will the CPU control return to MUCOS? In other words, how does context switching occur in the OS and how does the OS then activate the task switch to a higher-priority task? The CPU control returns to MUCOS (or in any other preemptive scheduler) as soon as one of the following situations arises.

1. Any interrupt event including the occurrence of the timer tick interrupt. Refer to Example 9.7, Step 10 (time set for the interrupt every 10 ms in Step 2).
2. On suspending the presently running task by calling OSTaskSuspend as in Example 9.8, Step 12.
3. As soon as any OS function, say a time delay function in Table 9.3 or a semaphore-pending function in Table 9.4 is called, the scheduler switches context, preempts and thus passes the control to other higher priority assigned task by activating task switch.

How does the control of CPU return to a preempted task because of an infinite loop existing in pre-empting higher-priority task also? It must return by an appropriate coding. For example, refer to the code in Step 12 in Example 9.8. Here, the FirstTask is of priority 8 (the highest available to a user task). It suspends itself from the loop at Step 12.

1. *Creating a task.* Function unsigned byte OSTaskCreate (void (*task) (void *taskPointer), void *pdata, OS_STK *taskStackPointer, unsigned byte taskPriority) is explained as follows.

A preemptive scheduler preempts a task of higher priority. Therefore, each user-task is to be assigned a *priority*, which must be set between 8 and OS_MAX_TASKS - 9 (or 8 and OS_LOWEST_PRIORITY - 8). OS reserves eight highest and eight lowest priority tasks for its own functions. Total number of tasks that MUCOS manages can be up to 64. Mucos task *priority* is also the task identifier. There is no task ID-defining function in MUCOS because each task has to be assigned a distinct *priority*.

If the maximum number of user tasks is eight, then OS_MAX_TASKS is 24 (including eight system-level tasks and 8 lowest priority system-tasks), the priority must be set between 8 and 15. The OS_LOWEST_PRIORITY must be set at 23 for eight user tasks of priority between 8 and 15, because MUCOS will assign priorities 16 to 23 to the 8 lowest priority system level tasks. The *priorities* 0 to 7 or 16 to 23 will then be for MUCOS internal uses.

Table 9.2 Service and System Time Functions for the Tasks

Prototype of Functions	What are the Parameters Returned?	What are the Parameters Passed?	When is this Operating System (OS) Function Called?
unsigned byte OSTaskCreate (void (*task) (void *taskPointer), void *pdata, OS_STK *taskStackPointer, unsigned byte taskPriority)	RA	PA	Must call before running a task
unsigned byte OSTaskSuspend (unsigned byte taskPriority)	RB	PB	Called for blocking a task
unsigned byte OSTaskResume (unsigned byte taskPriority)	RC	PC	Called for resuming a blocked task
void OSTimeSet (unsigned int count)	None	PD	Each <i>count</i> represents the system clock ticks. When system time is to be set it is set by an initial <i>count</i> value
unsigned int OSTimeGet (void)	RE	None	Find the present <i>count</i> so that the system time is read

Unsigned int means a 32-bit unsigned integer. Abbreviations used in columns 2 and 3 are explained in text.

OS_LOWEST_PRIORITY and OS_MAX_TASKS are user-defined constants in preprocessor codes that are needed for configuring the MUCOS for the user application. Defining unnecessarily 20 user tasks when actually 4 tasks are created by the user is to be avoided because more OS_MAX_TASKS means unnecessarily higher memory space allocation by the system to the user tasks.

Task parameters passing PA:

- (a) *taskPointer is a pointer to the codes of task being created.
- (b) *pdata is pointer for an optional message data reference passed to the task. If none, we assign it as NULL.
- (c) *TaskStackPointer is a pointer to the stack of task being created.
- (d) TaskPriority is the task priority and must be within 8 to 15, if macro OS_LOWEST_PRIORITY sets the lowest priority equal to 23.

Returning RA: The lowest priority of any task OS_LOWEST_PRIORITY is 23. For the application program, task priority assigned must be within 8 to 15. The function OSTaskCreate () returns the following: (i) OS_NO_ERR, when creation succeeds; (ii) OS_PRIORITY_EXISTS, if priority value that passed already exists; (iii) OS_PRIORITY_INVALID, if priority value that passed is more than the OS_PRIORITY_LOWEST; (iv) OS_NO_MORE_TCB returns, when no more memory block for the task control is available.

A task can create other tasks, but an ISR is not allowed to create a task. An exemplary use is in creating a task, TaskI_Connect, for a connecting task. OSTaskCreate (TaskI_Connect, void (*) 0, (void *) *TaskI_ConnectStack [100], 10)

Task parameters passed as arguments are as follows.

- (a) TaskI_Connect, a pointer to the codes of TaskI_Connect for task being created.
- (b) The pointer for an optional message data reference passed to task is NULL.

(c) * `Task1_ConnectStack` is a pointer to the stack of `Task1_Connect` and it is given size = 100 addresses in the memory.

(d) `TaskPriority` is task priority allotted at 10, the highest but two that can be allocated.

It will generate error parameters. `OS_NO_ERR` = true in case creation of `Task1_Connect` task succeeds. `OS_PRIO_EXIST` = true, if priority 8 task is already created and exists. `OS_PRIO_INVALID` = true, if passed priority parameter is higher than `OS_LOWEST_PRIO` - 8. `OS_NO_MORE_TCB` = false, when TCB is available for `Task1_Connect` (TCB definition is given in Section 7.3).

Example 9.7

```
1. /* Preprocessor MUCOS configuring commands to define OS tasks service and timing functions as
enabled and their constants*/
#define OS_MAX_TASKS 24 /* Let maximum number of tasks in user application be 8. */
#define OS_LOWEST_PRIO 23 /* Let lowest priority task in the OS be assigned priority
= 23 for 8 user application tasks of priorities between 8 and 15. */
#define OS_TASK_CREATE_EN 1/* Enable inclusion of OSTaskCreate ( ) function */
#define OS_TASK_DEL_EN 1/* Enable inclusion of OSTaskDel ( ) function */
#define OS_TASK_SUSPEND_EN 1/* Enable inclusion of OSTaskSuspend ( ) function */
#define OS_TASK_RESUME_EN 1/* Enable inclusion of OSTaskResume ( ) function */

/* End preprocessor MUCOS configuring commands */
2. /* Specify all user prototype of task-functions to be scheduled by MUCOS */
/* Remember: Static means permanent memory allocation */
static void FirstTask (void *taskPointer);
static void Task1_Connect (void *taskPointer);
static OS_STK FirstTaskStack [FirstTask_StackSize];
static OS_STK Task1_ConnectStack [Task1_Connect_StackSize];
/* Define public variable of the task service and timing functions */
#define OS_TASK_IDLE_STK_SIZE 100 /* Let memory allocation for an idle state task stack size be
100*/
#define OS_TICKS_PER_SEC 100 /* Let the number of ticks be 100 per second. The system clock will
interrupt and thus tick every 10 ms to update the set counts and to transfer control to the MUCOS. */
#define FirstTask_Priority 8 /* Define first task in main priority */
#define FirstTask_StackSize 100 /* Define first task in main stack size */
#define Task1_Connect_Priority 10 /* Define Task1_Connect priority */
#define Task1_Connect_StackSize 100 /* Define Task1_Connect stack size */

3. /* The codes of the application starts from main*/
void main (void) {
4. /* Initiate MUCOS to let us use the OS kernel functions */
OSInit ();
5. /* Create first task that must execute once before any other. Task creates by defining its
identity as FirstTask, stack size and other TCB parameters. */
```

```
OSTaskCreate (FirstTask, void (*) 0, (void *) &FirstTaskStack [FirstTask_StackSize],
FirstTask_Priority);
/* In this example, FirstTask will be creating other tasks. May create other main tasks and
inter-process communication variables if these must also execute at least once after the
FirstTask. */

6. /* Start MUCOS RTOS to let us use RTOS control and run the created tasks */
OSStart (); /* Infinite while-loop is there in each task. So there is no return to main from the RTOS
function OSStart ( ). */
} /* End of the Main function */
7. /* The codes of the application first task that creates in Main*/
static void FirstTask (void *taskPointer) {
8. /* Create a Task as per Step 2 defined by task identity Task1_Connect, stack size and other TCB parameters.
*/
OSTaskCreate (Task1_Connect, void (*) 0, (void *) &Task1_ConnectStack [Task1_Connect_StackSize],
Task1_Connect_Priority)
9. /* Create other tasks and inter-process communication variables. */

10. /* Start Timer Ticks for using timer ticks later. */
OSTickInit (); /* Function for initiating system clock ticks at the configured time in the MUCOS
configuration preprocessor commands in Step 1 */
11. while (1) /* Infinite loop of FirstTask */

12. }; /* End infinite loop */
13.} /*End of FirstTask Codes. */
The FirstTask is the main and only task created in step 5. The first task calls a function for the timer
initiation (step 10). This is required as the RTOS timer functions are needed in the application. A task
function Task1_Connect codes creates in step 8. All other tasks that are created are the private tasks using
OSTaskCreate function within the first task function.
14. /* The codes for the Task1_Connect*/
static void Task1_Connect (*taskPointer) {
15. /* Initial assignments of the variables and pre-infinite loop statements that execute once
only*/

16. /* Start an infinite while-loop. */
while (1) {
17. /* Codes for Task1_Connect*/

18.}; /* End of while loop*/
19.} /* End of the Task1_Connect function */
```

2. *Suspending (blocking) a task.* Function unsigned byte OSTaskSuspend (unsigned byte taskPriority) Task parameters passing *PB*: taskPriority is the task priority and must be within 8 to 15 for 8 user-tasks.

Returning *RB*: The function OSTaskSuspend () returns the error parameters OS_NO_ERR when the blocking succeeds. OS_Prio_INVALID, if priority value that passed is more than 15, the OS_Prio_LOWEST constant value. OS_TASK_SUSPEND_Prio, if priority value that passed already does not exist. OS_TASK_SUSPEND_IDLE, if attempting to suspend an idle task that is illegal.

An exemplary use is in blocking the task TaskI_Connect of priority = TaskI_Connect_Priority is as follows: OSTaskSuspend (TaskI_Connect_Priority). Task parameter passed as argument is 6. Recall TaskI_Connect_Priority was assigned 10 earlier in the step 2 of Example 9.7. The following error parameters will be returned by this function.

- OS_NO_ERR = true, when the blocking succeeds.
- OS_Prio_INVALID = false, as 8 is a valid priority and is not more than OS_Prio_LOWEST.
- OS_Prio_LOWEST = 23.
- OS_TASK_SUSPEND_Prio = false, as priority value that passed already does exist.
- OS_TASK_SUSPEND_IDLE = false, when attempting to suspend a task that was not an idle task. MUCOS executes idle task OSTaskIdle () when all tasks are either waiting for timer expiry or for an IPC, e.g., semaphore IPC.

Example 9.8

```
1 to 11. /* Steps Codes as in Example 9.7 */
12 /* Suspend FirstTask, as it was for initiating the timer ticks (interrupts), creating the user application
tasks, and was to be run only once */
OSTaskSuspend (FirstTask_Priority); /*Suspend First Task and control of the RTOS passes to other
tasks waiting execution, for example to TaskI_Connect. */
} /* End of while loop */
13. ) /* End of FirstTask Codes */
14 to 19. /* Steps Codes as in Example 9.7 */
```

3. *Resuming (enabling unblocking) a task.* Function unsigned byte OSTaskResume (unsigned byte taskPriority) resumes a suspended task.

Task parameters passing *PC*: taskPriority is the task priority of that task which is to resume and must be within 8 to 15 when OS_LOWEST_Prio is 23 and number of user-tasks = 8.

Returning *RC*: The function OSTaskResume () returns the OS_NO_ERR when the blocking succeeds. OS_Prio_INVALID, if priority value that passed is more than 23, the OS_LOWEST_Prio constant value. OS_TASK_RESUME_Prio, if priority value that passed already resumed. OS_TASK_NOT_SUSPENDED, if attempting to resume a not suspended (blocked) task.

An exemplary use is in un-blocking TaskI_Connect of priority = TaskI_Connect_Priority is as follows: OSTaskResume (TaskI_Connect_Priority). Task parameter passed as argument is 10, as TaskI_Connect_Priority = 10. The following error parameters will be returned by the task-resuming function.

- OS_NO_ERR = true, when the un-blocking succeeds and task of priority 8 reaches the running state.
- OS_Prio_INVALID = false, as 8 is a valid priority and is not more than OS_LOWEST_Prio.
- OS_LOWEST_Prio = 23.

- OS_TASK_RESUME_Prio = false, as priority value that passed already resumed.
- OS_TASK_NOT_SUSPENDED = false, when attempting to resume a task that was not suspended.

Example 9.9

```
1. to 19. /* Steps as per Example 9.7 codes for TaskI_Connect Function. */
20. /* Codes of other task, Task_N */
```

```
Step n. /* Resume TaskI_Connect. Control. */
OSTaskResume (TaskI_Connect_Priority);
Step n + 1. }; /* End of while loop*/
Step n + 2.} /* End of the Task_N function.*/
```

4. *Setting time in system clock.* Function void OSTimeSet (unsigned int count) returns no value. Passing parameter, PD as argument is given next.

PD: It passes a 32-bit integer for the *count* (set the number of ticks for the current time that will increment after each system clock tick in the system).

An exemplary use is a function OSTimeSet (to preset the time). The function OSTimeSet (0) sets the present *count* = 0. *Caution*: it is suggested that OSTimeSet function should be used before the OSTickInit function and only once then never be used within a task function, as some other functions that rely on the timer will malfunction. Let the OS timer clock *count* continue to be used as in a free running counter. There is little need later on of using the set time function. This is because at any instant, the time can be read using a get function (Example 9.11) and at any other instant, it can be defined again by adding a value to this time. Example 9.10 uses the set function in the FirstTask.

Example 9.10

```
void FirstTask (*taskPointer) {
1. to 9. /* The codes up to OSTickInit () in Example 9.7*/
10. /* Set the timer number of ticks to 0 */
unsigned int presetTime = 0;
OSTimeSet (presetTime);
OSTickInit ();
11. /* Other codes of FirstTask */

} /* End of the FirstTask function.
```

5. *Getting time of system clock.* Function unsigned int OSTimeGet (void) returns current number of ticks as an unsigned integer. The passing parameter as argument is none.
RE: Returns 32-bit integer, current number of ticks at the system clock.

Example 9.11

1: to 20. /* The codes as per steps i to 19 in Example 9.7 and the codes as for another task function */

```
unsigned int currentTime = OSTimeGet ( );
2. /* Other codes of the task after determining current time */
```

```
/* End of Task-function. */
```

3.2.3 Time Delay Functions

MUCOS time delay functions for tasks are as per Table 9.3.

Table 9.3 Time Delay Functions for Tasks

Prototype Function	What are the Parameters Returned?	What are the Parameters Passed?	When is this Operating System (OS) Called?
void OSTimeDly (unsigned short delayCount)	None	PF	When a task is to be delayed by count inputs equal to delayCount - 1. The task, which delays is the one in which this function executes. ¹
unsigned byte OSTimeDlyResume (unsigned byte taskPriority)	RG	PG	When a task of priority = taskPriority is to resume before the preset delay, which was by an amount defined by delayCount or (hr, mn and ms) and presently is in blocked state
oid OSTimeDlyHMSM (unsigned byte r, unsigned byte mn, unsigned byte ec, unsigned short ms)	RH	PH	When need is to delay and block a task for hr hours, mn minutes, sec seconds and ms milliseconds ²

Abbreviations used in columns 2 and 3 are explained in text.

¹ Task cannot be delayed than 65,535 system clock count inputs (ticks) by function OSTimeDly.

² Task cannot resume later by OSTimeDlyResume () if delay (in hours, minutes, seconds and milliseconds) is set more than 65,535 system clock count inputs (ticks).

1. *Delaying by defining a time delay by number of clock ticks.* Function void OSTimeDly (unsigned short delayCount) delays task by (delayCount - 1) ticks of system clock. It returns no parameter. Task parameters passing PF: A 16-bit integer, delayCount, to delay a task at least till the system clock count inputs (ticks) equals to (delayCount - 1) + count, where count is the present number of ticks at time-clock.

An exemplary use as a function in a task is OSTimeDly (10,000). It delays that task for at least 100,000 ms if system clock ticks after every 10 ms.

Example 9.12

1. to 18. /* Steps as per Example 8.7 codes for Task1_Connect Function. */

```
19. /* Time delay for 1 second = period of 100 system ticks if system tick is set at every 10 ms.*/
```

```
OSTimeDly (100);
```

```
20. /* Resume Task1_Connect by a function defined in next subsection and execute other codes within the loop. */
```

```
21. /* End of while loop*/
```

```
22. /* End of the Task1_Connect function. */
```

2. *Resuming a delayed task by OSTimeDly.* Function unsigned byte OSTimeDlyResume (unsigned byte taskPriority) resumes a previously delayed task, whether the delay parameter was in terms of the delayCount ticks or hours, minutes and seconds. *Note:* In case, defined delay is more than 65,535 system clock ticks, OSTimeDlyResume will not resume that delayed task.

Returning RG: Returns the following error parameters.

(a) OS_NO_ERR = true, when resumption after delay succeeds.

(b) OS_TASK_NOT_EXIST = true, if task was not created earlier.

(c) OS_TIME_NOT_DLY = true, if task was not delayed.

(d) OS_PRIO_INVALID, when taskPriority parameter that was passed is more than the OS_PRIO_LOWEST (=23).

Task parameters passing PG: taskPriority is the priority of that task that is delayed before resumption.

An exemplary use is OSTimeDlyResume (Task_ReadPortPriority). It resumes a delayed task that the OS identifies by priority Task_ReadPortPriority.

Example 9.13

1. to 19. /* Steps as per Example 9.12 codes for Task1_Connect Function. */

```
20. /* Time delay for 1 second = period of 100 system ticks if system tick is set at every 10 ms.*/
```

```
OSTimeDly (100);
```

```
21. /* Other codes */
```

```
22. /* Resume Task1_Connect Control and execute other codes within the loop. */
```

```
OSTimeDlyResume (Task1_Connect_Priority);
```

```
23. /* End of while loop*/
```

```
24. /* End of the Task1_Connect function. */
```

3. *Delaying by defining a time delay in units of hours, minutes, seconds and milliseconds.* Function void OSTimeDlyHMSM (unsigned short hr, unsigned short mn, unsigned short sec, unsigned short mils) delays up to 65,535 ticks a task with delay time defined by hr hours between 0 and 55, mn minutes between 0 and 59, sec seconds between 0 and 59 and mils milliseconds between 0 and 999. The ms adjusts to the integral multiple of number of system-clock ticks. The task in which this function is defined is delayed.

Returning RH: The function OSTimeDlyHMSM () returns an error code as following.

- OS_NO_ERR, when four arguments are valid and resumption after delay succeeds.
- OS_TIME_INVALID_HOURS, OS_TIME_INVALID_MINUTES, OS_TIME_INVALID_SECONDS and OS_TIME_INVALID_MILLI, if the arguments are greater than 55, 59, 59 and 999, respectively.
- OS_TIME_ZERO_DLY, if all the arguments passed are 0.

Task parameters passed PH: (a) to (c) hr, mn, sec and ms are the delay times in hours, minutes, seconds and milliseconds by which task delays before resuming.

An exemplary use is using OSTimeDlyHMSM (0, 0, 0, 999) function in the codes of a task in step 8 in Example 9.12. It delayed that task by 9990 ms. The function delays that task for at least 10 ms if system clock ticks after every 10 ms. (If delay is defined as 9,000,000 ms, the OSTimeDlyResume shall not be able to resume this task when asked. Number of ticks must be less than 65,535, which means maximum delay can be 655,350 ms if system clock ticks every 10 ms).

9.2.4 Memory Allocation-Related Functions

Memory functions are required to allocate fixed-size memory blocks from a memory partition having integer number of blocks. The allocation takes place without fragmentation. The allocation and de-allocation take place in fixed and deterministic time (Example 8.6). MUCOS memory functions for the tasks are as per Table 9.4.

Table 9.4 Real-Time Operating System (RTOS) Memory Functions for Querying, Creating, Getting and Putting

Prototype Functions	What are the Parameters Returning and Passed?	When is this OS Call?
OSMem *OSMemCreate (void *memAddr, MEMTYPE numBlocks, MEMTYPE blockSize, unsigned byte *memErr)	RJ and PJ	To create and initialize a memory partition. The memory blocks are then allotted from the partition
void *OSMemGet (OS_MEM *memCBPointer, unsigned byte *memErr)	RJ and PJ	To find pointer of the memory control block allocated to the memory blocks. NULL if no blocks. OSMemGet is used when an interrupt service routine (ISR) or task needs to get the memory block(s)
unsigned byte OSMemQuery (OS_MEM *memCBPointer, OS_MEM_DATA *memData)	RK and PK	To find pointers of the memory control block and OS_MemData data structure
unsigned byte OSMemPut (OS_MEM *memCBPointer, void *memBlock)	RL and PL	To return a pointer of memory block in the memory partitions from the memory control block pointer. OSMemPut is used when the application no longer needs the memory block

MEMTYPE is unsigned int of 16 or 32 bits.

1. *Creating memory blocks at a memory address.* Function OSMem *OSMemCreate (void *memAddr, MEMTYPE numBlocks, MEMTYPE blockSize, unsigned byte *memErr) is an OS function, which partitions the memory from an address with partitions in the blocks. The creation and initializing of the memory partitions into the blocks helps the OS in resources allocations.

Returning RJ: The function *OSMemCreate () returns a pointer to a control block for the created memory partitions. If none created, the create function returns NULL pointer.

Task parameters passing PJ: MEMTYPE is the data type according to the memory, whether 16-bit or 32-bit CPU memory addresses are there. For example, 16-bit in 68HC11 and 8051. (i) *memAddr is pointer for the memory-starting address of the blocks. (ii) numBlocks is the number of blocks into which the memory must be partitioned (must be 2 or more). (iii) The blockSize is the memory size in bytes in each block. (iv) *memErr is a pointer of the address to hold the error codes. At the address *memErr the following global error code variables change from false to true. OS_NO_ERR = true when creation succeeds. OS_MEM_INVALID_BLKS = true, when at least two blocks are not passed as arguments. (v) OS_MEM_INVALID_PART = true, when memory for partition is not available. (vi) OS_MEM_INVALID_SIZE = true, when block size is smaller than a pointer variable.

Example 9.14 shows the creation of four blocks of memory each block being of 1 kB. Let memory start address be 0x8000.

Example 9.14

1. /* Definition in Pre-Processor for a 16-bit unsigned number, MEMTYPE to define number of blocks which can be between 0 and 65535 and to define the number of bytes that store at a block. Maximum number of bytes at a block can be 65535. */

```
typedef unsigned short MEMTYPE;
```

3. /* Codes for main function or for a task function */

4. /* Codes for creating the blocks of memory */

```
memAddr = 0x8000;
```

```
numBlocks = 4;
```

```
blockSize = 1024; /* Each block is of 1kB memory */
```

```
*OSMemCreate (*memAddr, numBlocks, blockSize, *memErr);
```

5. /* Other Codes for the function. */

```
} /* End of the function */
```

2. *Getting a memory block at a memory address.* Function void *OSMemGet (OS_MEM *memCBPointer, unsigned byte *memErr) is to retrieve a memory block from the partitions created earlier.

Returning RJ: The function OSMemGet () returns a pointer to the memory control block for the partitions. It returns NULL if no blocks exist there.

Task parameters passing PJ: (i) Passes a pointer as argument for the control block of a memory partition. (ii) The function OSMemGet () passes the error code pointer *memErr so that later it returns one of the

following. OS_NO_ERR, when memory block returns to the memory partition, or OS_MEM_FULL, when memory block cannot be put into the memory partition as it is full.

Example 9.15 shows how to get a pointer to the memory block, which has been created earlier.

Example 9.15

```
1. to 5. /* Codes as per Example 9.14 */
6. /* Codes for retrieving the pointer to memory block in a partition created by step 5 in Example 9.14 */
memPointer = 0xA000;
memErr = OS_MEM_NO_FREE_BLK;
*OSMemGet (*memPointer, *memErr);
.
.
5. /* Other Codes for the function. */
.
.
} /* End of the function */
```

3. *Querying a memory block.* Function unsigned byte OSMemQuery (OS_MEM *memCBPointer, OS_MEMDATA *memData) is to query and return the error code and pointers for the memory partitions. OS_NO_ERROR as 1 if a memory address *memPointer exists at *OS_MEMDATA, else returns 0.

Returning *RK*: The function OSMemQuery () returns an error code, which is an unsigned byte. The code is OS_NO_ERR = 1 when querying succeeds, else 0.

Task parameters passing *PK*: (i) The function OSMemQuery () passes (i) a pointer memPointer of the memory created earlier, and (ii) a pointer of data structure, OS_MEM_DATA. As pointers are passed as references, the information about memory partition returns with the memory control block pointer.

4. *Putting a memory block into a partition.* Function unsigned byte OSMemPut (OS_MEM *memCBPointer, void *memBlock) returns a memory block pointed by *memBlock, which memory control block points by *memCBPointer.

Returning *RL*: The function OSMemPut () returns error codes for one of the following: either (i) OS_NO_ERR, when the memory block returned to the memory partition or (ii) OS_MEM_FULL, when the memory block cannot be put into the memory partition as it is full.

Task parameters passing *PL*: (i) The function OSMemPut () passes a pointer *memCBPointer of the memory control block for the memory partitions. It is there that the block is to be put. (ii) A pointer of the memory block *memBlock is to be put into the partition.

9.2.5 Semaphore-Related Functions

MUCOS semaphore functions for the tasks are as per Table 9.5. MUCOS also provides for event functions for an event flags group to handle task-pending action on occurrence of any or all events. These are not discussed in this section.

When a semaphore created by this OS is used as a resource-acquiring key (as mutex), the semaphore value to start with is 1, which means that resource is available and 0 will mean not available (Section 7.7.2). When a semaphore created by this OS is used as an event-signalling flag or as counting semaphore, the semaphore value to start with = 0 or N when using the semaphore as event-signalling flag or counting, respectively (Sections 7.7.1 and 7.7.4).

1. *Creating a semaphore for the IPCs.* Function OS_Event OSemCreate (unsigned short semVal) is for creating an OS's ECB (Event Control Block) for an IPC with semVal returning a pointer, which points to the ECB. A semaphore creates and initializes with the value = semVal.

Returning *RM*: The function OSemCreate () returns a pointer *eventPointer for the ECB allocated to the semaphore. Null if none available.

Task parameters passing *PM*: A semVal between 0 and 65535 is passed. For IPC as an event-signalling flag, SemFlag must pass 0 and as a resource-acquiring key, SemKey must pass 1. For IPC as a counting semaphore, SemCount must be either 0 or a count-value to be passed in the beginning.

Refer to Examples 9.16, 9.17 and 9.18 for understanding the use of OSemCreate.

2. *Waiting for an IPC for semaphore release.* Function void OSemPend (OS_Event *eventPointer, unsigned short timeOut, unsigned byte *SemErrPointer) is for letting a task wait till the release event of a semaphore: SemFlag or SemKey or SemCount. The latter is at the ECB pointed by *eventPointer. SemFlag or SemKey or SemCount becoming greater than 0 is an event that signals the release of the tasks in the waiting states. The tasks now become ready for running (They run if no other higher-priority task is ready). The tasks also become ready after a predefined timeout, timeOut. SemFlag or SemKey or SemCount decrements and if it becomes 0 then it makes the semaphore pending again and the other tasks using OSemPend () (have to wait for its release).

Returning *RN*: The function OSemPend () when a semaphore is pending, then suspends till >0 (release) and decrements the semVal on unblocking that task SemVal = 1. The following macros return true. (i) OS_NO_ERR returns true, when semaphore search succeeds (SemVal > 0). (ii) OS_TIMEOUT returns true, if semaphore did not release (did not become >0) during the ticks defined for the timeout. (iii) OS_ERR_PEND_ISR returns true, if this function call was by an ISR and which is an error, since an ISR should not be blocked for taking the semaphore. (iv) OS_ERR_EVENT_TYPE returns true, when *eventPointer is not pointing to the semaphore.

Task parameters passing *PN*: (i) The OS_Event *eventPointer passes as a pointer to ECB that associates with the semaphore: SemFlag or SemKey or SemCount. (ii) Passes argument for the number of timer ticks for the timeOut. Task unblocks after the delay is equal to (timeOut - 1) ticks even when the semaphore is not released. It prevents infinite wait. It must pass 0 if this provision is not used. (iii) Passes *err, a pointer for holding the error code.

Examples 9.16, 9.17 and 9.18 explains use of OSemPend ().

Table 9.5 Real-Time Operating System (RTOS) Semaphore Functions for Inter Tasks Communications

Prototype of Functions ¹	What are the Parameters Returning and Passed? ¹	When is this OS Function Called?
OS_Event OSemCreate (unsigned short semVal)	<i>RM</i> and <i>PM</i>	To create and initialize ECB and a semaphore to SemVal.
void OSemPend (OS_Event *eventPointer, unsigned short timeOut, unsigned byte *SemErrPointer)	<i>RN</i> and <i>PN</i>	Only a task and not an interrupt service routine (ISR) can accept the semaphore. The function is to check whether semaphore is pending or not pending (0 or >0). If pending (=0), then suspend the task till >0 (released). If >0, decrement the value of semaphore and run the waiting codes. Decrement makes the semaphore pending again for some other

(Contd)

Prototype of Functions ¹	What are the Parameters Returning and Passed? ¹	When is this OS Function Called?
unsigned short OSSemAccept (OS_EVENT *eventPointer)	RO and PO	An ISR or task can accept the semaphore. The function checks whether semaphore value > 0 and if yes, then retrieve and decrement. Used when there is no need to wait by the task, only decrease it to 0 if the value is not already zero.
unsigned byte OSSemPost (OS_EVENT*eventPointer)	RP and PP	An ISR or task can post the semaphore. SemVal if 0 or more, increments. Increment makes the semaphore again not pending for the waiting tasks. If tasks are in the blocked state and waiting for the SemVal semaphore to acquire value >0 then make those also ready to run as and when scheduled by the kernel. The kernel finds the priority of the running and ready tasks and runs the one that has the highest priority first.
unsigned byte OSSemQuery (OS_EVENT *eventPointer, OS_SEM!_DATA *SemData)	RQ and PQ	To get semaphore information.

Column 2 refers to the corresponding explanatory paragraph in the text.

3. *Check for availability of an IPC after a semaphore release.* Function unsigned short OSSemAccept (OS_Event *eventPointer) checks for a semaphore value at ECB and whether it is greater than 0. An unassigned 16-bit value is retrieved and then decremented.

Returning RO: The function OSSemAccept () decrements the semVal if >0 and returns the predecremented value as an unsigned 16-bit number. It returns 0 if semVal was 0 and semaphore was not pending when posted (released). After this, the task codes run further.

Task parameters passing PO: The OS_Event *eventPointer passes a pointer for the ECB that associates with semaphore, *semVal*.

4. *Sending an IPC after a semaphore release.* Function unsigned byte OSSemPost (OS_Event *eventPointer) is for letting another waiting task not wait now afterwards and an IPC is sent for the release event of the semaphore, *SemFlag* or *SemKey* or *SemCount* (Example 9.16). The IPC is at the ECB pointed by *eventPointer *SemFlag* or *SemKey* or *SemCount*, it increments and if it becomes greater than 0, it is an event that signals the release of a task waiting state. The task now become ready for running (runs if no other higher-priority task is ready). *SemFlag* or *SemKey* or *SemCount* decrements on running that task and if it becomes < 0 then it makes semaphore pending again and the other tasks have to wait for its release.

If the IPC is posted from an ISR, then the pending task can run only after OSIntrExit () executes and return from the ISR. If the presently running task is of higher priority than the task pending for the want of the IPC, then the present task will continue to run unless blocked or delayed by executing some function.

Returning RP: The function OSSemPost () increments the semVal if it is 0 or > 0, and later following macros return true from the error code macros as follows: (i) OS_NO_ERR returns true, if semaphore signaling succeeded (SemVal > 0 or 0). (ii) OS_ERR_EVENT_TYPE returns true, if *eventPointer is not pointing to the semaphore. (iii) OS_SEM_OVF returns true, when semVal overflows (cannot increment and is already 65,535).

Task parameters Passing PP: The OS_Event *eventPointer passes as pointer to ECB that associates with the semaphore.

5. *Retrieve the error information for a semaphore.* Function unsigned byte OSSemQuery (OS_EVENT *eventPointer, OS_SEM!_DATA *SemData) puts the data values for the semaphore at the pointer, SemData.

Returning RQ: After the OSSemQuery () runs the SemData function and gets the OSCnt, which is the semaphore present value (count). The Semdata also gets the list of the tasks, which are waiting for the semaphore. The list is at pointers OSEventTbl [] and OSEventGrp. The semaphore error information parameters we can find on running the macros, OS_NO_ERR and OS_ERR_EVENT_TYPE. (i) OS_NO_ERR returns true, when querying succeeds or (ii) OS_ERR_EVENT_TYPE returns true, if *eventPointer is not pointing to the semaphore.

Task parameters passing PQ: The function OSSemQuery () passes a pointer of the semaphore created earlier at *eventPointer and a pointer of the data structure at *SemData for that created semaphore.

Example 9.16

The use of OSSemPost and OSSemPend as an event-signalling flag is as follows. Let the initial value of an event-signalling *SemFlag* be 0 on creating a semaphore by OSSemCreate. A task must first execute OSSemPost, which increases the *SemFlag* to 1 and thus notifies the event. When *SemFlag* becomes 1 released (not taken), the waiting task (task that executed OSSemPend function) on posting of the semaphore as 1 can start running (it runs when no other higher-priority task is ready to run). The semaphore *SemFlag* decreases to 0 (again pending or taken) on return from the OSSemPend function. The waiting codes of the task now run.

Consider an example of reading bytes on a network. Assume that an ISR executes on a character reaching at a port. Another task is read port A. Third task is to decipher the port message. This example shows how the steps *a*, *b* and *c* synchronize the ISR and two tasks using semaphore as event-signalling flag for waiting and sending an IPC.

- For step *a*, let the task be ISR_CharIntr. It executes on interrupt and writes the character into PortA buffer. It signals for availability character at port A buffer using semaphore semFlag.
- For step *b*, let the task signalled to run by the ISR be Task_Read_Port_A. It is for reading the character when available at port A.
- For step *c*, let the task be Task_Decrypt_Port_A. It is for decrypting the message.

The codes to create the ISR and two tasks and synchronize these will be as follows.

1. /* Codes as per Example 9.7 Step 1 except last comment line*/

2. /* Preprocessor definitions for maximum number of inter process events to let the MUCOS allocate memory for the Event Control Blocks */

```
#define OS_MAX_EVENTS 8/* Let maximum IPC events be 8 */
```

```
#define OS_SEM_EN 1/* Enables inclusion of semaphore functions in applications using MUCOS */
```

```
/* End of preprocessor commands */
```

3. /*Codes as per Example 9.7 Step 2 */

```

4. /* Prototype definitions for ISR and two tasks, stacks and priorities. */
static void ISR_CharIntr (void *IntrVectorPointer);
static void Task_Read_Port_A (void *taskPointer);
static void Task_Decrypt_Port_A (void *taskPointer);
static OS_STK Task_Read_Port_AStack [Task_Read_Port_AStackSize];
static OS_STK Task_Decrypt_Port_AStack [Task_Decrypt_Port_AStackSize];
#define Task_Read_Port_AStackSize 100 /* Define task 2 stack */
#define Task_Decrypt_Port_AStackSize 100 /* Define task 3 stack */
#define Task_Read_Port_APriority 12 /* Define task 2 priority */
#define Task_Decrypt_Port_APriority 13 /* Define task 3 priority */
5. /* Prototype definitions for the semaphores */
OS_EVENT: SemFlag1; /*Needed when using Semaphore as flag for inter-process communication between
port check and port read tasks. Port read has to wait for check O.K.*/
OS_EVENT: SemFlag2; /*Needed when using Semaphore as flag for inter-process communication between
port read and port read decipher task. Port decrypting has to wait for port read */
OS_EVENT: SemKey1; /*Needed when using Semaphore as resource key as in Example 9.17*/
OS_EVENT: SemCount; /*Needed when using Semaphore as Counting as in Example 9.18*/
6. /* Codes as per Example 9.7 Step 3 to 5 */

7. /* Create Semaphores and Start MUCOS RTOS to let us RTOS control and run the created tasks */
SemFlag1 = OSSemCreate (0); /*Declare initial value of semaphore = 0 for using it as an event signaling
flag*/
SemFlag2 = OSSemCreate (0); /*Declare initial value of semaphore = 0 for using it as an event signaling
flag*/
OSSStart ( ); /* Infinite while-loop is there in each task. So there is no return from the RTOS function
OSSStart ( ). */
} /* End of while loop*/
} /* *** End of the Main function *** */
/* Codes as per Example 9.7 Step 7 and 8 */
8. /* Create two tasks as per Step 2 by defining two task identities. Task_Read_Port_A and
Task_Decrypt_Port_A and the stack sizes and other TCB parameters. */
OSTaskCreate (Task_Read_Port_A, void (*) 0, (void *) & Task_Read_Port_AStack
[Task_Read_Port_AStackSize], Task_Read_Port_APriority);
OSTaskCreate (Task_Decrypt_Port_A, void (*) 0, (void *) & Task_Decrypt_Port_AStack
[Task_Decrypt_Port_AStackSize], Task_Decrypt_Port_APriority);
9. while (1) { /* Infinite loop of FirstTask */

10. /* Suspend, with no resumption later, the First task as it must run once only for initiation of timer ticks
and for creating the tasks that the scheduler controls by preemption. */
11. OSTaskSuspend (FirstTask_Priority); /*Suspend First Task and control of the RTOS passes to other
tasks waiting execution*/
12. } /* End of while loop */

```

```

13. } /* End of FirstTask Codes */
/*****
14. /* The codes for the ISR_CharIntr */
static void ISR_CharIntr (void *BufferPointer) {
15. OSIntEnter ( ); /*
/* Initial assignments of the variables and pre-infinite loop statements that execute once
only. */
.
.
16. /* Code for resetting interrupt pending flag, in case, it does not automatically reset in the given interrupt
service start */
17. /* Codes for ISR_CharIntr to put the received character into Port A buffer at *BufferPointer */
.
.
18. /* Code for readying for next interrupt at the port */
19. /* Release semaphore to a task waiting for the read at Port A */
OSSemPost (SemFlag1);
20. OSIntExit ( );
21. } /* End of the ISR_CharIntr function */
22. /*****The codes for the Task_Read_Port_A *****/
static void Task_Read_Port_A (Void *BufferPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that execute
once only*/
.
.
23. /* Start an infinite while-loop. */
while (1) {
24. /*Wait for SemFlag1 =1 by OSSemPost function of character availability in buffer after interrupt at
port */
OSSemPend (SemFlag1, 0, SemErrPointer);
25. /* Codes for reading from Port A and storing message at a new buffer*/
.
.
26. /* Release semaphore to a task waiting for the decrypting*/
OSSemPost (SemFlag2);
OSTimeDly (100); /* Block the task for 100 clock ticks to enable the lower priority decryption task
start */
27. }; /* End of while loop*/
28. } /* End of the Task_Read_Port_A function */
/*****
29. /* Start of Task_Decrypt_Port_A codes */
static void Task_Decrypt_Port_A (void *taskPointer) {
30. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/

```

```

31. /* Start an infinite while-loop. */
while (1) {
OSSemPend (SemFlag2, 0, *SemErrPointer); /* Wait for unlimited time for SemFlag2 =1 by OSSemPost
function for a character read at Port A */
32. /* Codes for Task_Decrypt_Port_A that read the new buffer and put the decrypted data back into new
buffer */

OSTimeDlyResume (Task_Read_Port_Priority); /* Resume the Task_Read_Port delayed earlier */
33. /* End of while loop*/
34. } /* End of the Task_Decrypt_Port_A function */
/*****

```

Example 9.17

Use of OSSemPost and OSSemPend as resource-acquiring keys is as follows. The resource may be a shared memory buffer or commands that use global variables or touch screen or flash memory or print device control registers and buffers. Let a resource key available *SemKey*'s initial value be 1. A task must first execute OSSemPend, which decreases the *SemKey* value to 0 and the codes of the critical section *C* of task run. The section is one in which that resource is used. The same task must execute OSSemPost after its codes finish at *C* and thus signal the resource key availability to other tasks. The *SemKey* becomes 1 and released (not taken) on return from OSSemPost function. If no other higher-priority task is ready to run, another task that shares the resource with the earlier task and executes OSSemPend function on entering shared data section *C*. The *C* executes OSSemPost function on exit. Making *SemKey* 0 and then 1 in task sections *C* and *C* lets one task only acquire the resource in a specific running state of a task.

Recall Example 9.16, steps *b* and *c* for reading and then decrypting the bytes from a network. If there is no message, how can it be deciphered? Let us revisit Example 9.16. The present example will show how the steps *b* and *c* synchronize using a semaphore key for key waiting and key sending IPC, and how steps *a* and *b* synchronize using a semaphore as event signaling flag.

```

1. /* Codes as per Example 9.16 Step 1 to 5 */

2. /* Prototype definition for the semaphore used as resource key for inter-process communication between
port read and port message encrypt read tasks. */
OS_EVENT *SemKey1; /* Needed when using Semaphore as resource key*/
/* Codes as per Example 9.16 Steps 6 to 21. However, create the semaphore SemKey1 before calling
OSStart () in main*/
SemKey1 = OSSemCreate (1); /* Declare initial value of semaphore = 1 for using it as a resource acquiring
key*/
/*****

```

```

4. /* After the end of codes for ISR_CharIntr, the codes for the Task_Read_Port_A redefined to show a
use of the key*/
static void Task_Read_Port_A (void *taskPointer) {
5. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/

6. while (1) { /* Start an infinite while-loop. */
7. OSSemPend (SemFlag1, 0, SemErrPointer); /* Wait for SemFlag1 =1 by OSSemPost function of
character availability check task */
/* Acquire resource as SemKey presently > 0 and decrement it and not allow any other task to use this
key*/
8. OSSemPend (SemKey1, 0, SemErrPointer);
9. /* Codes for reading from Port A buffer and storing at new buffer, which will be shared with
Task_Decrypt_Port_A and the tasks for transmitting decrypted data to another device */

10. /* Release the key to a task waiting for the decrypting*/
OSSemPost (SemKey1);
11. /* To exit the infinite loop at a task that has been assigned a higher priority and to let the
lower priority task run call OS delay function for wait of 100 ms (ten OS timer tick. This is the
method to let the other task of lower priority execute Port A decrypt. */
OSTimeDly (10);
/* End of while loop*/
12. } /* End of the Task_Read_Port_A function */
/*****

13. /* Start of Task_Decrypt_Port_A codes */
static void Task_Decrypt_Port_A (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that execute once only*/

14. while (1) { /* Start the infinite loop */
15. /* Acquiring the resource as SemKey1 > 0 and decrement it to not to let port read task use this key*/
OSSemPend (SemKey1, 0, SemErrPointer);
16. /* Codes for Task_Decrypt_Port_A or deciphering the message read at new buffer */

17. /* Release the key to a task waiting for the new buffer data transmission to a device */
OSSemPost (SemKey1);
OSTimeDlyResume (Task_Read_Port_APriority); /* Resume the delayed task */
18. /* End of while loop*/
19. } /* End of the Task_Decrypt_Port_A function */
/*****

```


Example 9.18

Use of counting semaphore helps in programming for the printer or other bounded buffer problem (producer consumer problem) (Refer to Section 7.5). The use of OSSemPost to increase the count and OSSemPend to decrease the count in a counting semaphore is as follows. Recall Example 9.16. Let us first modify this example as follows.

1. For step *a*, on the interrupt let ISR_CharIntr executes. It posts a semaphore to task, Task_Read_Port_A.
2. For step *b*, let the task be Task_Read_Port_A. It is for reading the characters when available at port buffer. Let Task_ReadPortA read a stream of the characters from the buffer. Let the task put the characters, as it reads one by one, into a bounded buffer (it is a producing task; buffer is bounded by a limit like a printer or display buffer).
3. For step *c*, let the task be Task_Decrypt. It is for decrypting and displaying the maximum 160 characters (it is a consuming task; it is like printing from the print buffer).
4. For step *d*, let the task be Task_Display. Task display the decrypted data at display device.
5. Let the display buffer of 160 characters be shared by Task_Decrypt and is bounded upto 160 addresses in memory.

This example shows how the steps *b* and *c* synchronize using counting semaphore and how the steps *c* to *d* synchronize.

1. Let there be a counter *SemCount*, which counts the number of times a task posting the semaphore ran. Let *SemCount*'s initial value be 0. A first task section must first execute OSSemPost, which increases the *SemCount* to 1. Every time this task section runs, *SemCount* increases by 1. Every time the task deciphers a character in another task, it decreases by 1.
2. When *SemCount* reaches a specific preset value, then a semaphore event-signalling flag *SemCountLimitFlag* sets, and the count resets to 0. As there is an OS call by a delay function, the lower-priority task for deciphering starts running and it acquires the key. The semaphore resource key *SemKey* becomes unavailable to the reading task and further reading stops till the deciphering task releases the key and also executes OSSemPend to decrease *SemCount* to let the task that reached the limit run again.

1. /* Codes as per Example 9.17 Steps 1 and 2*/

2. /* Prototype definitions for one ISR and three tasks*/

```
static void ISR_CharIntr (void *IntrVectorPointer);
```

```
static void Task_Read_Port_A (void *taskPointer);
```

```
static void Task_Decrypt (void *taskPointer);
```

```
static void Task_Display (void *taskPointer);
```

3. /* Definitions for three task stacks */

```
static OS_STK Task_Read_Port_AStack [Task_Read_Port_AStackSize];
```

```
static OS_STK Task_DecryptStack [Task_DecryptAStackSize];
```

```
static OS_STK Task_DisplayStack [Task_DisplayStackSize];
```

4. /* Definitions for three task stack size */

```
#define Task_Read_Port_AStackSize 100 /* Define task 2 stack size*/
```

```
#define Task_DecryptStackSize 100 /* Define task 3 stack size*/
```

```
#define Task_DisplayStackSize 100 /* Define task 4 stack size*/
```

5. /* Definitions for four task priorities. */

```
#define Task_ReadPortAPriority 11 /* Define task 2 priority */
```

```
#define Task_DecryptPriority 12 /* Define task 3 priority */
```

```
#define Task_DisplayPriority 14 /* Define task 5 priority */
```

6. /* Prototype definitions for the semaphores */

```
OS_EVENT *SemFlag1; /* Needed when using semaphore as the flag for inter-process communication between ISR and port read tasks. Port A read task has to wait for ISR notifying the character receipt at port A */
```

```
OS_EVENT *SemCountSend; /* Needed when using semaphore for sending task semaphore count value in the inter-process communication between read and decipher tasks. Port deciphering has to wait for port receive from sending task */
```

```
OS_EVENT *SemKey; /* Needed when using semaphore as resource key */
```

```
OS_EVENT *SemCountRecv; /* Needed when using semaphore for counting the deciphered for display */
```

8. /* Codes as per Example 9.7 Step 3 to 8. However, the semaphores are to be created and initialised as done earlier in Step 5 Example 9.16 */

```
SemFlag1 = OSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as an event signaling flag*/
```

```
SemCountSend = OSSemCreate (0); /* Declare initial value of semaphore = 0 for an event counter for sender */
```

```
SemCountDisp = OSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as a flag for display */
```

```
/* Declare initial value of semaphore count = 0 for using as a counter that gives the number of times a task, which sends into a buffer that stores a character stream, ran minus the number of times the task which used the character from the stream ran from the buffer */
```

```
SemCountRecv = OSSemCreate (160); /* Declare initial value of semaphore = 160 for 160 free addresses count for receiving the bytes for display */
```

9. /* Create four tasks as per Step 3, defined by four task identities, Task_Read_Port_A, Task_Decrypt_Port_A, Task_EncryptPortB and Task_SendPortB and the stack sizes and other TCB parameters. */

```
OSTaskCreate (Task_Read_Port_A, void (*) 0, (void *) & Task_Read_Port_AStack
```

```
[/Task_Read_Port_AStackSize], Task_ReadPortAPriority);
```

```
OSTaskCreate (Task_Decrypt, void (*) 0, (void *) & Task_DecryptStack
```

```
[/Task_DecryptStackSize], Task_DecryptPriority);
```

```
OSTaskCreate (Task_Display, void (*) 0, (void *) & DisplayStack [/Task_DisplayStackSize], Task_DisplayPriority);
```

10. /* Codes same as at Steps 9 to 21 in Example 9.16 */

11. /* The codes for the Task_Read_Port_A redefined to use the semaphore as counter*/

```
static void Task_Read_Port_A (void *taskPointer) {
```

```
12. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
```

```

unsigned short countLimit = 160; /* Declare the buffer-size for the characters countLimit = 160 */
.
13. while (1) { /* Start an infinite while-loop. */
14. /* Wait for SemFlag1 1 by OSSemPost function of character availability check task */
OSSemPend (SemFlag1, 0, SemErrPointer);
15. OSSemPend (SemCountRecv, 0, *SemErrPointer); /* wait for available space in receiving buffer */
CountLimit = 160; /* CountLimit reset to 160 */
16. /* Read port A buffer byte and write byte into decipher buffer */
CountLimit -- ; if (CountLimit > 0){
OSSemPost (SemCountSend); /* Release the SemCountSend to let the decipher task start */
17. OSTimeDly (10); /* To exit the infinite loop of this assigned higher priority task to let the lower
priority task run call the OS delay function for a wait of 100 ms (ten OS timer tick). This is the method to
let the other task of lower priority execute Port A's message deciphering task */
18. } /* End of codes for the action on reaching the limit of putting characters into the buffer */
19. }; /* End of while loop */
20. } /* End of the Task_Read_Port_A function */
/*****
21. /* Start of Task_Decrypt */
static void Task_Decrypt (void *taskPointer) {
22. /* Initial assignments of the variables and pre-infinite loop statements that execute only once */
.
23 while (1) { /* Start the infinite loop */
24 /* Take the key to not letting the Task_Read_Port_A run before at least one cycle of this while loop */
OSSemPend (SemCountSend, 0, *SemErrPointer);
25. /* Codes for Task_Decrypt_Port_A or deciphering for displaying the message when placed
in new buffer fort */
.
OSSemPost (SemCountdisp, 0, *Err);
OSTimeDly (10); /* to delay for transfer control to display task */
26. OSTimeDlyResume (Task_ReadPortAPriority); /* Resume task read port A */
27. }; /* End of while loop */
28. } /* End of the Task_Decryptfunction */
/*****
29. /* The codes for the Task_Display */
static void Task_Display (void *taskPointer) {
30. /* Initial assignments of the variables and pre-infinite loop statements that execute only once */
.
.
31. while (1) { /* Start an infinite while-loop. */
32. /* Codes for displaying the deciphered characters */
OSSemPend (SemCountdisp, 0, *Err); /* wait for deciphered character */

```

```

33. /* Code for displaying character */
OSSemPost (SemCountRecv);
34. OSTimeDlyResume (Task_DecryptPriority);
35. }; /* End of while loop */
36. } /* End of the Task_Display function */
/*****

```

9.2.6 Mailbox-Related Functions

We have seen in Examples 9.16, 9.17 and 9.18 that the semaphores communicate one of the following.

1. The occurrence of an event to other task, which is waiting for the event before running.
2. The availability of a resource in a task to let a section of codes in the task run.
3. The occurrences of an event number of times before they are taken note by other task.
4. The availability of a resource in a task to let a section of codes in the task run number of times.

However, suppose the message is a string or is in a data structure or is in a buffer or array. *The mailbox IPC can be used to communicate a pointer for that information.* Refer to Figure 7.6(a), which showed multiple types of mailboxes. In MUCOS, the mailbox type is one message pointer per mailbox.

Let there be a pointer *msg to the message to be sent in the mailbox, and another *mboxPointer for the message sending event and retrieving the message itself. MUCOS mailbox IPC functions for the tasks are as per Table 9.6.

1. *Creating a mailbox for an IPC.* Function *OS_Event *OSMboxCreate (void *msg)* is for creating an ECB at the RTOS and thus initializing a pointer *mboxPointer to msg. The msg pointer is NULL, if the created mailbox initialized as empty mailbox.

Task parameters passing M1: *msg is message pointer to which *mboxPointer will initialise. For an IPC, sending the message-pointer *mboxPointer communicates the msg.

Returning M1: The function OSMboxCreate () returns a pointer to the ECB at the MUCOS and mboxPointer at ECB points to msg.

Step 8 in Example 9.19 shows how to use OSMboxCreate function.

2. *Check for availability of an IPC after a message at mailbox.* Function *void *OSMboxAccept (OS_EVENT * mboxPointer)* checks for a mailbox message at ECB at mboxPointer (an event pointer). The pointer for the message msg returns from the function, if message is available, mboxPointer not pointing to NULL but to the msg. After returning, the mailbox empties, and mboxPointer will point to NULL on emptying of mailbox. The difference with OSMboxPend function is that OSMboxPend suspends the task if the message is not available and waits for mboxPointer not equal to NULL.

Task parameters passing M2: The OS_Event * mboxPointer passes as pointer to ECB that associates with the mailbox.

Returning M2: The function OSMboxAccept () checks the message at *mboxPointer and returns the message pointer *msg presently at MsgPointer. The function then returns NULL pointer if message pointer is not available at mboxPointer. Later *mboxPointer will point to NULL, because mailbox empties.

Step 40 in Example 9.19 shows how to use *OSMboxAccept* to retrieve an error string, if any, available in the mailbox without specifically waiting and blocking the task.

Table 9.6 Mailbox Real-Time Operating System (RTOS) Mailbox Functions for the Intertask Communications

Prototype of Service and System Clock Function	What are the Parameters Returning and Passed? ¹	When is this OS Function Called?
OS_Event *OSMboxCreate (void *msg)	<u>M1</u> and M1	To create and initialize a mailbox message pointer for the ECB of a mailbox message.
void *OSMboxAccept (OS_EVENT *mboxPointer)	<u>M2</u> and M2	To check if mailbox message *msg is pointed by *mboxPointer. Unlike OSMboxPend function, it does not block (suspend) the task if message is not available. If available, it returns the pointer *msg and *mboxPointer again points to NULL. On the return mailbox empties
void *OSMboxPend (OS_Event *mboxPointer, unsigned short timeOut, unsigned byte *MboxErr)	<u>M3</u> and M3	To check if mailbox-message pending is available; then the message pointer is read and the mailbox emptied and *mboxPointer again points to NULL. If message is not available (*mboxPointer points to NULL) it waits, suspends the task and blocks further running (or till the number of ticks = <i>timeOut</i> - 1 occurs at the system-timer). If pending, then the task is resumed on availability, *mboxPointer now NULL. Resumes on timeOut also.
unsigned byte OSMboxPost (OS_EVENT *mboxPointer, void *msg)	<u>M4</u> and M4	Sends a message for a task presently at address msg by posting the address pointer of it to the mboxPointer. Context switch to that task or any another task if of higher priority will also occur. If the box is already full, then the message is not posted and the error information is given.
unsigned byte OSMboxQuery (OS_EVENT *mboxPointer, OS_MBOX_DATA *mboxData)	<u>M5</u> and M5	See text.

¹Column 2 refers to the corresponding explanatory paragraphs in the text for the intertask communications using a mailbox.

3. *Waiting for availability of an IPC for a message at mailbox.* Function *void *OSMboxPend (OS_Event *mboxPointer, unsigned short timeOut, unsigned byte *MboxErr)* checks a mailbox message pointer *msg* at ECB event pointer, *mboxPointer*. A pointer for the message retrieves on return, if message is available, *mboxPointer* not pointing to NULL but pointing to *msg* else waits till available or till time out, whichever is earlier. If *timeOut* argument value is 0, it means wait indefinitely till the message is available.
- Task parameters passing M3: (i) The OS_Event *mboxPointer passes as a pointer to ECB that is associated with the mailbox. (ii) Passes argument timeOut. This resumes that the blocked task after the delay is equal to timeOut - 1 count inputs (ticks) at the system-clock timer. (iii) Passes reference *MboxErr, a pointer that hold the error codes.

Returning M3: The function OSMboxPend checks as well as waits for the message at *mboxPointer and the function returns *msg*. After returning, the mailbox empties. *mboxPointer will later point to NULL, because mailbox empties. When message is not available, it suspends the task and blocks as long as *msgPointer is not NULL. It returns NULL pointer if the message is not available (msgPointer pointing to NULL). The following macros will then return true. (i) OS_NO_ERR returns true, when mailbox message search succeeds; (ii) OS_TIMEOUT returns true, if mailbox message does not succeed during the ticks defined for the *timeout* > 0; (iii) OS_ERR_PEND_ISR returns true, if this function call was from the ISR; (iv) OS_ERR_EVENT_TYPE returns true, when *mboxPointer is not pointing to the pointer type variable for *msg*.

Step 39 of Example 9.19 shows how to use *OSMboxPend* to retrieve an error string, if any, available in the mailbox by *Task_Err*, how to retrieve the read string at *Task_OutPortB* and to specifically wait and block the task.

4. *Send a message for an IPC through mailbox.* Function unsigned byte OSMboxPost (OS_EVENT *msgPointer, void *msg) sends mailbox message at ECB event pointer, *msgPointer. The message sent is at *msg*, as well as at *mboxPointer* after the posting.

Task parameters passing M4: The OS_Event *msgPointer passes as a pointer to ECB that associates with the message. The pointer *msg* passes to the mailbox address *msgPointer.

Returning M4: The function OSMboxPost () sends the message and then returns the error code on running the macros as follows: (i) OS_NO_ERR returns true, if mailbox message signalling is succeeded, or (ii) OS_ERR_EVENT_TYPE returns true, if *msgPointer does not point to the mailbox message type, or (iii) OS_MBOX_FULL returns true, when mailbox at msgPointer already has a message that is not accepted or returned.

Steps 24, 30 and 40 in Example 9.19 show OSMboxPost by the tasks to post the message to the waiting tasks for the messages.

5. *Finding mailbox data and retrieving the error information for a mailbox.* Function unsigned byte OSMboxQuery (OS_EVENT *msgPointer, OS_MBOX_DATA *mboxData) checks for a mailbox data and places that at *mboxData*. It also finds the error information parameters, OS_NO_ERR_EVENT_TYPE for the ECB.

Task parameters passing M5: The function OSMboxQuery () passes (i) a pointer of the mailbox message created earlier at *msgPointer and (ii) a pointer of data structure at *mboxData*.

Returning M5: Function OSMboxQuery () returns information in the pointer *mboxData* is a data structure with current contents of the message (OSMsg) and the list of waiting tasks for the message. The error code macro OS_NO_ERR returns true, when mailbox message querying succeeds or OS_ERR_EVENT_TYPE returns true, if msgPointer does not point to mailbox type *msg*.

Example 9.19

Let a task, *Task_Read_Port_A*, after it receives a message string (reading an array of character received at a port A), use OSMboxPost to send an IPC to another task waiting (blocked) for that message. An exemplary situation is when receiving a called party telephone number from the keypad at port A in mobile phone by a task and the task is waiting for dialing and transmitting of the number after ascertaining that the number does not have an invalid character. Let the waiting tasks be *Task_OutPortB* and *Task_SendPortB*. The latter sends the string for the telephone number to port B after the wait is over.

To *Task_OutPortB*, not only *Task_Read_Port_A* but also another task, *Task_Err*, can send an error message on detecting an invalid character or if the limit of characters expected in the string is exceeded. In the present example, the application of OSMboxPend function is for a task-wait for a message as well as for the error message string also (Refer to wait by OSMboxPend in task, *Task_SendPortB*, which executes

a service routine in case of error string detect). For example, in a mobile phone, *Task_OutPortB* can be used as follows. When there is no error message, then establish the connection with the cellular service and then dial and transmit the called number using *Task_SendPortB*. When there is an error message, *Task_OutPortB* directs the message to another task, *Task_ErrSR*. Another task displays the error message string warning the user to redial the number. The steps in this operation are as follows.

1. Step *a*: Task *ISR_CharIntr* interrupts on the port A status ready on availability of a character. For example, the ISR executes if a key is pressed in a mobile phone keypad (Section 1.10.5). The use of semaphore *SemFlag1* as in Example 9.16 suffices because an IPC will be just for an interrupt event flag.
2. Step *b*: Task *Task_Read_Port_A* waits for the *SemFlag1* and executes the codes that accumulate the characters into an array to obtain a string, *str*. *OSMboxPost* posts a message pointer for the *str* if no other key is pressed within a time-out period.
3. Step *c*: Task *Task_Err* checks each *message* read at port A and sends a string, *errStr*, into the mailbox when the character is not a valid character or if the number of characters has exceeded the limit. It posts the message back into the mailbox if it does not have invalid characters. For example, character is not a number in case of a telephone number, which is read by the task at step *b*.
4. Step *d*: Task *Task_OutPortB* waits for *str* and *errstr* in the mailbox. The use of mailbox for the IPCs is between steps *b* and *d*, and *c* and *d*.
5. Step *e*: Task *Task_SendPortB*. If there is no error, the task sends the message of port B.
6. Step *f*: Task *Task_ErrSR* to execute a service routine in case of error.

This example shows how the steps *a* and *b* synchronize by the IPC *SemFlag1*, how tasks at the steps *b* and *c* synchronize and how steps *b* to *d* and *c* and *d* synchronize using the mailbox functions of the MUCOS.

1. /* Define Boolean variable and NULL pointer. Define codes as per Example 9.17 Steps 1*/

```
typedef char int8bit;
#define int8bit boolean
#define false 0
#define true 1
/* Define a NULL pointer */
#define NULL (void*) 0x0000
```

```
#define unsigned byte inputCharsMaxSize 16 /* Let Maximum size of telephone-number string be 16 characters. */
```

2. /* Preprocessor definitions for maximum number of inter-process events to let the MUCOS allocate memory for the Event Control Blocks */

```
#define OS_MAX_EVENTS 12 /* Let maximum IPC events be 12 */
#define OS_SEM_EN 1 /* Enable inclusion of semaphore functions in application. */
#define OS_MBOX_EN 1 /* Enable inclusion of mailbox functions in application. */
/* End of preprocessor commands */
```

3. /* Prototype definitions for ISR and five tasks for steps *a* to *f* above. */

```
static void ISR_CharIntr (void *IntrVectorPointer);
static void Task_Read_Port_A (void *taskPointer);
static void Task_Err (void *taskPointer);
static void Task_OutPortB (void *taskPointer);
static void Task_SendPortB (void *taskPointer);
```

```
static void Task_ErrSR (*taskPointer);
/* Definitions for five task stacks */
static OS_STK Task_Read_Port_AStack [Task_Read_Port_AStackSize];
static OS_STK Task_ErrStack [Task_ErrStackSize];
static OS_STK Task_OutPortBStack [Task_OutPortBStackSize];
static OS_STK Task_SendPortBStack [Task_SendPortBStackSize];
static OS_STK Task_ErrSRStack [Task_ErrSRStackSize];
/* Definitions for five task stack size */
#define Task_Read_Port_AStackSize 100 /* Define task 2 stack size*/
#define Task_ErrStackSize 100 /* Define task 3 stack size*/
#define Task_OutPortBStackSize 100 /* Define task 4 stack size*/
#define Task_SendPortBStackSize 100 /* Define task 5 stack size*/
#define Task_ErrSRStackSize 100 /* Define task 3 stack size*/
4. /* Definitions for five task priorities. */
#define Task_ReadPortAPriority 10 /* Define task 2 priority */
#define Task_ErrPriority 11 /* Define task 3 priority */
#define Task_OutPortBPriority 12 /* Define task 4 priority */
#define Task_SendPortBPriority 13 /* Define task 5 priority */
#define Task_ErrSRPriority 14 /* Define task 6 priority */
5. /* Prototype definitions for semaphores */
OS_EVENT SemFlag1; /* Needed when using semaphore as a flag for inter-process communication from
ISR_CharIntr and to Task_Read_Port_A (Port A read task) on port A interrupt */
OS_EVENT SemCharInvalid; /* Needed when using semaphore as a flag for inter-process communication
from Task_OutPortB task and to number transmitting task Task_SendPortB. */
OS_EVENT semCharCountLimitFlag; /* Needed when using semaphore as the flag for limiting the character
count value in the inter-process communication between Task_Read_Port_A and Task_Err */
6. /* Prototype definitions for the mailboxes */
OS_EVENT MboxStrPointer; /* Needed when using the mailbox message between steps b and d and
d and e */
OS_EVENT MboxErrStrPointer; /* Needed when using the mailbox message between steps c and d */

7. /* Codes as per Example 9.7 Step 3 to 8. However, before the 'OSStart ( )', the semaphore and mailbox
must be created and initialised as in Step 6 Example 9.18. */
SemFlag1 = OSSemCreate (0) /* Declare initial value of semaphore = 0 for using it as an event signaling
flag*/
SemFlag2 = OSSemCreate (0) /* Declare initial value of semaphore = 0 for using it as an event signaling
flag*/
semCharCountLimitFlag = OSSemCreate (0) /* Declare initial value of semaphore = 0 as an event signaling
flag*/
SemKey = OSSemCreate (1) /* Declare initial value of semaphore = 1 for using it as a resource key*/
/* Create Mailboxes for the tasks. */
8. MboxStrPointer = OSMboxCreate (NULL); /* Needed when using mailbox message between steps
b and d to pass a string message pointer*/
```

```
MboxErrStrPointer = OSMboxCreate (NULL); /* Needed when using mailbox message
between steps b and c to pass a error string message pointer*/
```

9.

```
10. /* Create five tasks as shown in Step 9 Example 9.7 defining five task identities, Task_Read_Port_A,
Task_Err, Task_OutPortB and Task_SendPortB and the stack sizes, other TCB parameters. */
```

```
OSTaskCreate (Task_Read_Port_A, void (*) 0, (void *) &
```

```
Task_Read_Port_AStack [Task_Read_Port_AStackSize], Task_ReadPortAPriority);
```

```
OSTaskCreate (Task_Err, void (*) 0, (void *) & Task_ErrStack [Task_ErrStackSize], Task_ErrPriority);
```

```
OSTaskCreate (Task_OutPortB, void (*) 0, (void *) & Task_OutPortBStack [Task_OutPortBStackSize],
Task_OutPortBPriority);
```

```
OSTaskCreate (Task_SendPortB, void (*) 0, (void *) & Task_SendPortBStack [Task_SendPortBStackSize],
Task_SendPortBPriority);
```

```
OSTaskCreate (Task_Err, void (*) 0, (void *) & Task_ErrStack [Task_ErrStackSize], Task_ErrPriority);
```

```
/******
```

```
11-13. /* Codes same as those in Steps 9 to 21 in Example 9.16. The ISR executes on each key-press
interrupt at port A */
```

;

.

```
/* End of the ISR_CharIntr function */
```

```
/******
```

```
14. /* Example 9.16 codes for the Task_Read_Port_A redefined. To use the mailbox*/
```

```
static void Task_Read_Port_A (void *taskPointer) {
```

```
/* Initial assignments of the variables and pre-infinite loop statements that execute only once*/
```

```
char *portAdata: boolean findChrInvalid (chr [ ]):
```

```
char [ ] portAinputStr: /* Let port A input string be an array to hold the data from port A*/
```

```
unsigned byte charCount = 0: /* To count the number of characters read from the port. The counter that gives
the number of times the data sent into a buffer (portAinputStr in present case) that stores a character stream,
ran minus the number of times the task, which used the character from the stream, ran from the buffer */
```

```
boolean charInvalid = false: /* Initialize charInvalid flag as false */
```

```
while (1) { /* Start an infinite while-loop and Wait for SemFlag1 posting by OSSemPost function on a
character-availability */
```

```
OSSemPend (SemFlag1, 0, SemErrPointer);
```

```
15. Port_AInputstr [chrCount] = PortAdata;
```

```
16. /* Code for reading a byte from buffer that the ISR_CharIntr wrote before posting SemFlag 1 */
```

```
17. /* Actions on maximum size exceeding the string buffer size and on character found invalid */
```

```
if (charCount > inputCharsMaxSize) {OSSemPost (semCharCountLimitFlag);} /* Post the
semCharCountLimitFlag */
```

```
findChrInvalid (Port_AInputStr) {/* Code for check */ return (chrInvalid = true);}
```

```
if (chrInvalid == true) {OSSemPost (SemcharInvalid);
```

```
/* To exit the infinite loop of this higher priority assigned task to let the lower priority Task_Err, we call the
OS delay function that forces a wait of 20 ms (two OS timer ticks) or until delay resume function executes.
This is the method to let the other task of lower priority execute */
```

```
OSTimeDly (2); /* Delay by 20 ms (two timer ticks) to let lower priority Task_Err run */
```

```
charCount = 0; /* Reset the character counter to 0. */
```

```
18. } /* End of Codes for the action on character invalid or reaching the limit of putting characters into the
buffer */
```

```
19. /*Le: the charCount increase after one character has been put into the string holding the character
stream*/
```

```
charCount ++; /* Increment the character count */
```

```
/* If an ASCII code for start of text is found then initialize charCount = 0. */
```

```
20. If (portAinputStr [chrCount] == 0x02) {charCount = 0;};
```

```
21. /* Other codes in case required */;
```

```
22. For sending string into mailbox if maximum characters reading over*
```

```
23. /* codes for
```

```
If (chrCount == input CharsMaxSize) {OSMboxPost (MboxStrPointer. port A InputStr); charCount = 0;
}
```

```
24. /* When the character is equal to End of Text, ASCII code at Port A (for example, the Enter key
pressed) is found send the message pointer String to the waiting mail box at Task_OutPortB and make
initial charCount = 0 again for next string*/
```

```
if (PortAinputStr [charCount] == 03)
```

```
{OSMboxPost (MboxStrPointer, portAinputStr); charCount = 0};
```

```
25. /* End of while loop*/ OSTimeDly (2); /* Delay to let Task_Err start */
```

```
26. } /* End of the Task_ReadPortA function */
```

```
/******
```

```
27. /* Codes of Task_Err */
```

```
static void Task_Err (void *taskPointer) { / Initial assignments of the variables and pre-infinite
loop statements that execute once only*/
```

```
unsigned byte [ ] msgBuffer = 0 /* Declare initial value of msgBuffer = 0 */
```

```
/*Declaration for an error string for using when at the Step d an error invalid-character is found at mailbox.
char [ ] ErrStr1 = "Invalid Character Found";
```

```
/* Declaration for an error string message for task at Step d when the limit exceeds. */
```

```
char [ ] ErrStr2 = "Characters in the message exceeded the limit declared at task for read";
```

```
boolean invalid = false; /* Declare invalid variable 'false' and will be assigned 'true' in case a character is
found invalid. */
```

```
28. while (1) { /* Start the infinite loop */
```

```
29. /* Post Limit of Message Exceeded Message to task at step d. */
```

```
OSSemQuery (semCharCountLimitFlag, SemData)
```

```
30. if (SemData -> OSCnt == 1) {OSMboxPost (MboxErrStrPointer, ErrStr2); OSSemAccept
(semCharCountLimitFlag, 0, *SemErrPointer); OSTimeDly (2);};
```

```
31. /* Codes for reading Msg */
```

```
msgBuffer = OSMboxPend (MboxStrPointer, 0, MboxErrData);
```

```
32. /* Post Mailbox message to task at step d if an invalid character is not detected else Post invalid
character error message */
```

```

33. OSSemQuery (SemCharInvalid, SemData);
34. /* Take semCharInvalid (if >0) by accepting the semCharInvalid semaphore. Task does not suspend
even if semaphore not available (not > 0). This task has to run whether count is invaled not.
if (SemData -> OSCnt == 0 && chrInvalid == true) {OSMboxPost (MboxErrStrPointer, ErrStr1);
OSSemAccept (semCharInvalid);} else {OSMboxPost (MboxStrPointer, msgBuffer); C3TimeDly (2)};
/* If data invalid post error string or message in a error mailbox else post message in the mailbox */
OSTimeDlyResume (Task_ReadPortAPriority; /* Resume the task Port A Road*
*/ /* End of while loop*/
35. } /* End of the Task_Err function */
/*****
36. /* Codes for the Task_OutPortB */
static void Task_OutPortB (void *taskPointer) {
37. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
char [ ] message; /* Declare error free message string pointer*/
char [ ] errMessage; /* Declare error message pointer. */
38. while (1) { /* Start an infinite while-loop. */
39. /* Wait for Mailbox Message available (not NULL) */
message = OSMboxPend (MboxStrPointer, 0, MboxErrPointer);
40. /* Check for Mailbox Error Message available (not NULL) */
errMessage = OSMboxAccept (MboxErrStrPointer);
if (errMessage != NULL) {OSMboxPost (MboxErrStrPointer, errMessage);}
else {
/* Codes for again sending the Port B string of characters to task for transmission; the message is tested to
see that it has no invalid character or that it never exceeds the limits of its size.*/
OSMboxPost (MboxStrPointer, message);
};
/* To exit the infinite loop at higher priority assigned task to let the lower priority task run, call the
OS delay function for wait of 20 ms (two OS timer ticks). This is the method to let the other task
at lower priority execute Port B sending the characters. */
OSTimeDly (2);
41. OSTimeDlyResume (Task_ErrPriority); /* Let delayed higher priority task err resume. */
/* End of while loop*/
42. } /* End of the Task_OutPortB function */
/*****
43. /* Codes of Task_SendPortB */
static void Task_SendPortB (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that execute once */
unsigned char [ ] message;
.
.
44. while (1) { /* Start the infinite loop */
45.
46. /* Wait for error free message from Port B. If available, retrieve it. */
message = OSMboxPend (MboxStrPointer, 0, MboxErrPointer);

```

```

47. /* Codes for sending the valid message to a memory buffer where it is saved or to a network for
transmission */
.
.
48. OSTimeDly (2); /* Delay for low priority task start */ OSTimeDlyResume (Task_OutPortBPriority);
/* Resume Delayed Task_OutPortB */
}; /* End of while loop*/
49. } /* End of the Task_SendPortB function */
/*****
50. /* Codes of Task_Err */
static void Task_Err (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that execute only once */
char [ ] errMessage; /* Declare error message pointer. */
51. while (1) { /* Start an infinite while-loop. */
52. /* Check for Mailbox Error Message available (not NULL) */
errMessage = OSMboxAccept (MboxErrStrPointer);
53. /* Codes for the action on error message. */
if (strcmp (errMessage, "Invalid Character Found") == 0) {
/* Codes for the actions that need to be taken when invalid characters are found. For example, codes for
displaying "Invalid Number Dialed. Dial Again" on an LCD. */
.
.
};
if (strcmp (errMessage, " Characters in the message exceeded the limit declared at task for
read") ==0) {
/* Codes for actions needed on limit exceeded. Codes for displaying on an LCD "Message too
long to Accept. Dial again". */
.
.
}
OSTimeDlyResume (Task_SendPortB); /* Resume task_SendPortB
54. }; /* End of while loop*/
55. } /* End of the Task_Err function */
/*****

```

9.2.7 Queue-Related Functions

It is learnt from Example 9.18 that the semaphore communicates an IPC for an event occurrence. It is learnt from Example 9.19 that a mailbox communicates a pointer for a message, which may be larger or smaller. By using a queue, we can communicate an array of message pointers from the tasks. The message pointers can be posted into a queue by the tasks either at the back as in a *queue* or at the front as in a *stack*. (A priority message is posted in front.) A task can thus insert a given message for deleting either in the FIFO mode or in the LIFO mode. (Note: The IPC queue differs from the data structure queue with respect to the available methods for inserting an element into a queue. The OS controls the IPC queue.)

Refer to Figure 7.12. MUCOS permits a queue of an array of pointers. Let there be a pointer, ***QTop*, to a queue of message pointers, and there be two pointers, **QheadPointer* and **QtailPointer*, which send and retrieve, respectively, the message pointer for the message. MUCOS queue functions for the tasks' IPCs are as per Table 9.7. MUCOS permits up to 65,536 message-pointers into a queue (i.e., the MUCOS queue size can be 65,536). The post-front function enables insertion such that the waiting task does LIFO retrieval of the message-pointer, hence of the message.

1. *Creating a Queue for an IPC.* *OS_Event QMsgPointer = OSQCreate (void **QTop, unsigned short qSize)* is used for creating an OS's ECB for the *QTop* and queue is an array of pointers at *QMsgPointer*. The array size can be declared as maximum 65,536 (0th to 65,535th element). Initially, the array *QMsgPointer* points to NULL.

Table 9.7 Queue Functions for the Intertask Communications

Prototype of Service and System Clock Function	Parameters Returned and Passes	When is this Operating System (OS) Called?
<i>OS_Event OSQCreate (void **QTop, unsigned short qSize)</i>	<u>R</u> and R	OS creates a queue ECB. This creates and initializes an array of pointers for the queue at <i>QTop</i> . Queue can be of maximum size = <i>qSize</i> . <i>QTop</i> should point to top (zeroth element of an array). ECB points at the <i>QMsgPointer</i> .
<i>void *OSQPend (OS_Event *QMsgPointer, unsigned short timeOut, unsigned byte *Qerr)</i>	<u>S</u> and S	Refer to text.
<i>unsigned byte *OSQFlush (OS_EVENT *QMsgPointer)</i>	<u>T</u> and T	To eliminate all the messages in the queue that have been sent. This function checks if a queue has a message pending at <i>QMsgPointer</i> (the queue front pointer at the ECB is not pointing to NULL). Function then returns all the message pointers between the queue front pointer and queue back pointer at the ECB. It returns error codes and <i>QMsgPointer</i> will point to NULL.
<i>unsigned byte OSQPost (OS_EVENT *QMsgPointer, void *QMsg)</i>	<u>U</u> and U	Sends a pointer of the message <i>QMsg</i> to the <i>QMsgPointer</i> at the queue back. The message inserts at a queue tail pointer in the ECB.
<i>unsigned byte OSQPostFront (OS_EVENT *QMsgPointer, void *QMsg)</i>	<u>V</u> and V	Sends <i>QMsg</i> to the <i>QMsgPointer</i> at the queue. It points to the queue head pointer in the ECB where pointer for <i>QMsg</i> now stores pushing other message pointers backward. ¹
<i>unsigned byte OSQQuery (OS_EVENT *QMsgPointer, OS_Q_DATA *QData)</i>	<u>X</u> and X	To get queue message's information and error information.

Column 2 refers to the corresponding explanatory paragraph in the text.

¹Use *OSQPostFront* or *OSQPost* functions according to the message priority, if the message has a higher priority, post it at the front; else, post as usual in a queue. We can use post-front and post functions to build a queue in which an array of message pointers are stored and ordered according to their priorities. Queue is then called a prioritized ordered queue. Use of *OSQPostFront* and then *OSQPend* enable indirectly a LIFO mode of retrieval of a message pointer to priority message.

Task parameters passing R: The ***QTop* passes as pointer for an array of voids. The *qSize* is the size of this array (number of message pointers that the queue can insert before a read is within 0 and 65,535).

Returning R: The function *OSQCreate ()* returns a pointer to the ECB allocated to the queue. It is an array of voids initially. NULL if none is available.

Example 9.20 explains the use of *OSQCreate* function.

2. *Waiting for an IPC message at a queue.* Function *void *OSQPend (OS_Event *QMsgPointer, unsigned short timeOut, unsigned byte *Qerr)* checks if a queue has a message pending at ECB *QMsgPointer* (*QMsgPointer* is not pointing to NULL). The message pointer points to the queue front (head) at the ECB for the queue defined by *QMsgPointer*. It suspends the task if no message is pending [until either the message received or the wait period, passed by argument *timeOut*, finishes after (*timeOut - 1*) ticks of the system timer]. The queue head pointer at the ECB will later increment to point to the next message after returning the pointer for the message.

Returning S: The function returns pointer to a queue at ECB. It also returns the following on running the macros as under: (i) *OS_NO_ERR* returns true, when the queue message search succeeds; (ii) *OS_TIMEOUT* returns true, if queue did not get the message during the ticks defined by the *timeOut*; (iii) *OS_ERR_PEND_ISR* returns true, if this function call was from the ISR; (iv) *OS_ERR_EVENT_TYPE* returns true, when **QMsgPointer* is not pointing to the queue message.

Task parameters passing S: (i) The *OS_Event *QbackPointer* passes as pointer to the ECB that is associated with the queue. (ii) It passes 16-bit unsigned integer argument *timeOut*. It resumes the task after the delay equals (*timeOut - 1*) count inputs (ticks) at the system clock. (iii) It passes **err*, a pointer for holding the error code.

Example 9.20 explains the use of *OSQPend* function.

3. *Emptying the queue and eliminating all the message pointers.* Function *unsigned byte *OSQFlush (OS_EVENT *QMsgPointer)* checks if a queue has a message pending at *QMsgPointer* (the queue front pointer at the ECB does not point to NULL). The function returns all the message pointers between queue front pointer and queue back pointer at the ECB. It returns an error code and *QMsgPointer* at ECB. These will later point to NULL on return from the function.

Task parameters passing T: The *OS_Event *QMsgPointer* passes as pointer to the ECB that is associated with the queue.

Returning T: After the function *OSQFlush ()* executes, the error macros returns as follows: *OS_NO_ERR* returns true, if the message queue flush succeeds, or *OS_ERR_EVENT_TYPE* returns true, if *QMsgPointer* is not pointing to the queue message queue.

4. *Sending a message pointer to the queue.* The function *unsigned byte OSQPost (OS_EVENT *QMsgPointer, void *QMsg)* sends a pointer of the message **QMsg*. The message pointer *QMsgPointer* (queue tail pointer) points to the *QMsg*.

Task parameters passing U: The *OS_Event *QMsgPointer* passes as pointer to the ECB that is associated with the queue tail.

Returning U: After the function *OSQPost ()*, the message pointer **QMsg* is passed for the message to **QMsgPointer* and the error macros return the error code as follows: (i) *OS_NO_ERR* returns true, if queue signalling succeeded, (ii) *OS_ERR_EVENT_TYPE* returns true, if **QtailPointer* is not pointing to the queue and (iii) *OS_Q_FULL* returns true, when queue message cannot be posted (*QSize* cannot exceed a limit set on creating the queue).

Example 9.20 explains the use of *OSQPost* function.

5. *Sending a message pointer and inserting it at the queue front.* The function *unsigned byte OSQPostFront (OS_EVENT *QMsgPointer, void *QMsg)* sends *QMsg* pointer to the *QMsgPointer* at the queue, but it is at the queue front pointer in the ECB where pointer for *QMsg* now stores, pushing other message pointers backwards.

Task parameters passing V: The OS_Event *QMsgPointer passes as pointer to the ECB that is associated with the queue. The second argument is the message QMsg address that is the queue front address.

Returning V: After the function OSQPostFront () executes the following error macros returns as under: (i) OS_NO_ERR returns true, if the message at the queue front is placed successfully; (ii) OS_ERR_EVENT_TYPE returns true, if pointer QtailPointer is not pointing to message queue; or (iii) OS_Q_FULL returns true, if qSize was declared n and queue had n messages waiting for the read.

Example 9.20 explains the use of OSQPostFront function.

6. Querying to find the message and error information for the queue ECB. The function unsigned byte OSQQuery (OS_EVENT *QMsgPointer, OS_Q_DATA *QData) checks for a queue data and places that at QData. It also finds the error information parameters, on executing the following macros: OS_NO_ERR and OS_ERR_EVENT_TYPE.

Task parameters passing X: The function OSQQuery passes (i) a pointer of the queue at *QMsgPointer ECB and (ii) a pointer of the data structure at *QData.

Returning X: QData has pointer to the message at OSMsg, number of messages at OSNMsgs, OSQSize as queue size in terms of the number of entries permitted and list of the tasks waiting for the message. After the function, the following macros returns true: (i) OS_NO_ERR, when querying succeeds or (ii) OS_ERR_EVENT_TYPE, if *QMsgPointer is not pointing to queue message.

Example 9.20

Let a task, *Task_ReadPortA*, receive characters *QMsg* and put these into a queue. The task uses OSQPost to send an IPC for another task waiting (blocked) for these characters. An advantage is that the messages can be used as soon as available by another task without the completion of the whole message string as was the case in Example 9.19. The mailbox permitted only one message through a message pointer. Queue permits any number of messages till the queue gets full. Full means that the maximum array size defined for the queue is reached. Another advantage is that port A need not be the one sending characters or bytes, but can be an NIC (network input card) or any other input device sending a word, frame or a segment of a message that needs to be posted to another waiting task in a sequence. Further, any number of error messages sent by the *Task_Err* can also be posted into the same queue as priority messages. The codes get simplified.

To *Task_MessagePortA*, not only *Task_ReadPortA* but also another task, *Task_Err*, can send an error message on detecting an invalid character or if the limit of characters expected in the string is exceeded. The use of OSQPostFront is to send the errors as the priority message. OSQPend is used to wait for an IPC for the message as well as error message string. The steps in this operation are as follows.

1. Step a: ISR, *ISR_CharIntr* interrupt on the port A status for the availability of a message. (For example, the task is activated if, at port A, a key is pressed for sending the character or a network input data or a set of numbers are keyed on a keypad of mobile.) It puts the characters for message at portAData buffer. Semaphore *SemFlag1* (as in Example 9.16) posts an event occurrence as an IPC.
2. Step b: *Task_ReadPortA* waits for the *SemFlag1* and executes the codes that post the message, checks and, posts in front the error in the input message into a common queue.
3. It checks each character or message read at port A and sends a string, *errStr*, into a general message queue when the character or message has invalid character or message queue is full. For example, if the character or message is not a number in the case of a telephone number.
4. Step c: *Task_MessagePortA* waits for the characters or messages as an array of pointers. The task also sends the messages for display.
5. Step d: *Task_ErrLogins* also waits message for the error posted in a queue for error logins.

This example shows how the steps *a* and *b* synchronize by the IPC *SemFlag1*, how tasks at the steps *b* and *c* use queue to synchronize and how the steps *b* to *d* synchronize using the queue functions of the MUCOS.

1. /* Codes are the same as in Step 1 Example 9.19, except that the statements are shown in bold for the queues. The mailbox-related statements are replaced by the queue-related messages. */

```

2. /* Preprocessor definitions for maximum number of inter-process events to let the MUCOS allocate
memory for the Event Control Blocks */
#define OS_MAX_EVENTS 12 /* Let maximum IPC events be 12 */
#define OS_SEM_EN 1 /* Enable inclusion of semaphore functions in applications using MUCOS */
#define OS_Q_EN 1 /* Enable inclusion of queue functions in applications using MUCOS */
/* End of preprocessor commands */

3. /* Prototype definitions for ISR and tasks for steps b to d above. */
static void ISR_CharIntr (void *IntrVectPointer);
static void Task_ReadPortA (void *taskPointer);
static void Task_MessagePortA (void *taskPointer);
static void Task_ErrLogins (*taskPointer);

4. /* Definitions for task stacks */
static OS_STK Task_ReadPortAStack [Task_ReadPortAStackSize];
static OS_STK Task_MessagePortAStack [Task_MessagePortAStackSize];
static OS_STK Task_ErrLoginsStack [Task_ErrLoginsStackSize];

5. /* Definitions for task stack size */
#define Task_ReadPortAStackSize 100 /* Define task stack size*/
#define Task_MessagePortAStackSize 100 /* Define task stack size*/
#define Task_ErrLoginsStackSize 100 /* Define task stack size*/

6. /* Definitions for task priorities. */
#define Task_ReadPortAPriority 10 /* Define task 2 priority */
#define Task_ErrPriority 11 /* Define task 3 priority */
#define Task_MessagePortAPriority 12 /* Define task 4 priority */
#define Task_ErrLoginsPriority 14 /* Define task priority */

7. /* Prototype definitions for semaphore */
OS_EVENT SemFlag1; /* Needed for using the semaphore as a flag for inter-process communication between
port status interrupt, ISR_CharIntr and port read task. Task_ReadPortA Port A read task waits for semaphore
till port A interrupts and puts the message in port A data buffer. */
OS_EVENT SemCountLimitFlag; /* Needed when using the semaphore as flag for reaching the limits of
semaphore count in the inter-process communication between port read and port read decipher task. Port
reading has to wait for port read */
OS_EVENT *SemKey; /* Needed when using the semaphore as resource key by Task_ReadPortA and
Task_Err */
OS_EVENT *QMsgPointer; /* Needed when using queued message between steps b and d and steps d
and e */
void QMsgPointer [QMessagesSize]; /* Let the maximum number of message-pointers at the queue be
QMessagesSize. */

```



```

OS_EVENT *QErrMsgPointer; /* Needed when using a queued message between steps c and d */
void *QErrMsgPointer [QErrMsgMessagesSize]; /* Let the maximum number of error message-pointers at
the queue be QErrMsgMessagesSize. */
9. /* Define both queues array sizes. */
#define QMessagesSize = 64; /* Define size of message-pointer queue when full */
#define QErrMsgMessagesSize = 16; /* Define size of error message-pointer queue when full */
10. /* Codes for port input reading from Port A and storing a character or message at a queue or
buffer. Alternatively, modify the code for reading from a port at NIC or any other device or
peripheral. */
.
11. /* Codes as per Example 9.7, Steps 3 to 8. However, before the 'OSStart ( );', the semaphore and queue
must be created and initialised as under: */
SemFlag1 = OSSemCreate (0) /* Declare initial value of semaphore = 0 for using it as an event signaling
flag */
SemCountLimitFlag = OSSemCreate (0) /* Declare initial value of semaphore = 0 as an event signaling
flag */
SemKey = OSSemCreate (1) /* Declare initial value of semaphore = 1 for using it as a resource key */
12. /* Declare initial count as 0 as a counter that gives the number of times a task, which sends into a buffer
that stores a character or message stream, ran minus the number of times the task which used the character
or message from the stream ran from the buffer */
SemCount = OSSemCreate (0);
13. /* Create Two queues for the tasks. one general purpose queue and another for error logins only. The
queue for errors is posted messages after the Task_messagePortA selects error messages from the general
queue. */
/* Define a top of the message pointer array. QMsgPointer points to top of the Messages to start with. */
QMsgPointer = OSQCreate (&QMsg [0], QMessagesSize);
/* Define a top of the message pointer array. QMsgPointer points to top of the Messages to start with. */
QErrMsgPointer = OSQCreate (&QErrMsg [0], QErrMsgMessagesSize); /* Needed when using mailbox
message between steps c and d to pass a string message pointer */
14.
.
15. /* Create Tasks as per Step 3 defining by tasks—Task_ReadPortA, Task_Err, Task_MessagePortA and
Task_ServiceMessage and the stack sizes, other TCB parameters. */
OSTaskCreate (Task_ReadPortA, void (*) 0, (void *) & Task_ReadPortAStack [Task_ReadPortAStackSize],
Task_ReadPortAPriority);
OSTaskCreate (Task_MessagePortA, void (*) 0, (void *) & Task_MessagePortAStack [Task_
MessagesPortAStackSize], Task_MessagePortAPriority);
OSTaskCreate (Task_ErrLogins, void (*) 0, (void *) & Task_ErrLoginsStack [Task_ErrLoginsStackSize],
Task_ErrLoginsPriority);
16. /* Codes same as at Steps 9 to 21 in Example 9.16 */
;

```

```

17. /* End of the ISR_CharIntr function */
/*****
18. /* Codes for Task_ReadPortA redefined to use the key, flag and 16-bit value and mailbox */
static void Task_ReadPortA (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that execute only once */
char [ ] portAdata;
Char [ ] msgBeginStr = "Messages Begin";
boolean QFull = false,
unsigned short msgCount = 0;
unsigned byte [ ] msgBuffer /* Declare initial value of msgBuffer */
/* Declare an error string for step d error invalid message data found to the queue. */
char [ ] ErrStr1 = "Invalid Message Data Found";
/* Declare an error string message for task at step d when the limit exceeds. */
char [ ] ErrStr2 = "Array Size exceeded the Limit. Queue Full";
boolean invalid = false; /* Declare invalid variable 'false' and will be assigned 'true' when character or
message read is found invalid. */
19. /* Start an infinite while-loop and Wait for SemFlag1 1 by OSSemPost function of character or message
availability from ISR_CharInt */
while (1) {
OSSemPend (SemFlag1, 0, SemErrPointer);
/* Post the string to initiate start of the message */
if (msgCount == 0) {OSQPostFront (QMsgPointer, MsgBeginStr);}
20. /* Actions on maximum size exceeding the string buffer size */
if ( msgCount >= QMessagesSize) {OSQPostFront (QMsgPointer, ErrStr2);
Qfull = true; msgCount = 0;}
/* Code for checking any invalid character or message in port A buffer */
if (invalid == true) {msgCount = 0; OSQPostFront (QMsgPointer, ErrStr1);}
} /* End of Codes for the actions on reaching the queue array limit of the putting the message pointers into
the buffer or on finding an invalid message character or data */
21. /* Write the array element that returned as Port A data into the port A input string */
if (Qfull == false || invalid == false)
{OSQPost (QMsgPointer, &portAdata);
/* Let message counter value increase after one character or message has been put into the String, holding
the character or message stream */
msgCount ++;}
22. /* Let Task_MessagePortA start by delay */
23. OSTimeDly (2);
24. /* Other codes for read port task */
25. /* End of while loop */
26. /* End of Task_ReadPortA function */
27. /*
/*****

```

```

28. /* Codes for the Task_MessagePortA */
static void Task_MessagePortA (void *taskPointer) {
29. /* initial assignments of the variables and pre-infinite loop statements that execute only once */
void * message;
30. while (1) { /* Start an infinite while-loop. */
31. /* Wait for Queue Message Pointer available (not NULL) */
&message = OSQPend (QMsgPointer, 0, QErrPointer);
32. /* Find if the message has invalid character or the messages found as Error Messages. Check for queue
Error Message available (not NULL) */
if (strcmp ((char *) message, "Invalid Message Data Found ")==0) {OSQPost (QErrMsgPointer,
message); OSTimeDly (2)};
33. if (strcmp ((char *) message , "Array Size exceeded the Limit. Queue Full") == 0){OSQPost
(QErrMsgPointer, message); OSTimeDly (2)}; OSTimeDlyResume (Task_ErrPriority); /* Let delayed
higher priority task resume. */
&message = OSQPend (QMsgPointer, 0, QErr);
34. /* while (Strcmp (char *) message, "Messages Begin"){
/* Codes for servicing as per the valid message to a memory buffer for saving or to a network or to dial and
transmit */}
35. /* To exit the infinite loop at the task that has been assigned a higher priority and to let the next priority
ErrLogins task run, let us call the OS delay function for wait of 20 ms (two OS timer ticks). This is the
method to let the other task of lower priority execute. */
OSTimeDly (2);
36. OSTimeDlyResume (Task_ReadPortAPriority); /* Resume Delayed Task_ReadPortA */
}; /* End of while loop*/
37. } /* End of the Task_MessagePortA function */
/*****

38. /* Codes of Task_ErrLogins */
static void Task_ErrLogins (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that execute only*/
void errorLogged; /* Declare error message pointer. */
39. while (1) { /* Start an infinite while-loop. */
40. /* Check for Mailbox Error Message available (not NULL) */
&errorLogged = OSQPend (QErrMsgPointer, 0, QErrPointer);
41. /* Codes for the action as per the error logged in */
42. if (errorLogged == " Invalid Message Data Found" ) {
/* Codes for actions needed on invalid character or message found. For example,
codes for displaying "Invalid Number Dialed. Dial Again" on an LCD. */
.
.
.
};
43. if (errorLogged == " Array Size exceeded the Limit. Queue Full ") {

```

```

44. /* Codes for actions needed on limit exceeded. Codes for displaying on an LCD "Message too
long to Accept. Dial again". */
.
.
.
}
OSTimeDlyResume (Task_MessagePortAPriority); /* Resume the ServiceMessage */
}; /* End of while loop*/
45. } /* End of the Task_MessagePortA function */
/*****

```

9.3 RTOS VxWorks

For sophisticated embedded systems, there is a popular RTOS, VxWorks from WindRiver® (<http://www.windriver.com/>). VxWorks is a high-performance, Unix-like, scalable RTOS, and support to ARM, ColdFire, MIPS, Pentium, Intel X-Scale, Super H and other popular processors for embedded system design. VxWorks RTOS design is hierarchical (Section 8.9) and is for hard real-time applications. It supports kernel mode execution of tasks for fast execution of application codes.

VxWorks is supported with powerful development tools that make it easy and efficient to use. VxWorks supports many advanced processor architectures. VxWorks supports 'Device Software Optimization', which is said to be a new methodology that enables the development and the running of the device software faster, better and more reliably. VxWorks has been found to be the most popular RTOS in a survey in 2006 (<http://www.embedded.com/columns/showArticle.jhtml?articleID=187203>). VxWorks 6.x is its latest version.

VxWorks provides for the following.

1. Multitasking environment using scheduler which supports IEEE standard POSIX scheduler and which also supports the in-house developed scheduler.
2. Supports ability to run two concurrent OSs on a single processing layer.
3. Multiple file systems (Section 8.6.2) and systems that enable advanced multimedia functionality.
4. Synchronization using a full range of IPC options (Section 7.9) that includes. (i) event-signalling flag, (ii) mutually exclusive access using resource key (mutex), (iii) counting mechanism using three types of semaphores in the tasks, (iv) queue, (v) socket and ISRs and includes POSIX standard semaphore and other IPCs (Sections 7.8.3, 7.9 to 7.15 and 8.12). Also supports real-time processes IPCs. It also support openSource TIPC (transparent inter-process-communication) protocol for network and clustered systems environment. [PTTS 1.1 is latest release in December, 2007 for Linux and VxWorks.]
5. Different context saving mechanism for the tasks and ISRs (tasks have separate TCBs and stacks, and ISRs use a common stack due to nesting of the calls).
6. Watchdog timers.
7. Virtual IO devices including pipes and sockets (Sections 7.14 and 7.15).
8. Virtual memory management functions.
9. Power management functions that enhance the ability to control power consumption, and automatic detection and reporting of common memory and other errors.
10. Interconnect functions that a support large number of protocols, including IPv4/IPv6 dual mode stack ready APIs.

VxWorks TCB saves the following task information for each task.

1. Control information for the OS that includes *priority*, *stack size*, *state* and *options*.
2. CPU context of the task that includes PC, SP, CPU registers and task variables.

VxWorks also provides:

1. Pipe drivers for IPCs and pipe is an IO virtual device.
2. Network transparent sockets.
3. Network drivers for shared memory and Ethernet.
4. RAM 'disk' drivers for memory-resident files.
5. Drivers for SCSI, keyboard, VGA display, disk and parallel port of a computer system, HDD, diskette, tapes, keyboard and displays.
6. VxWorks 6.x provisions for processor abstraction layer. It enables application system design by user when using new versions of a processor architecture.

VxWorks IO system also includes the POSIX standard asynchronous IOs and UNIX standard buffered IOs. It also provides for simulator (VxSim) (Section 14.2.3), software logic analyser (WindView), Code coverage study tool, MemScope, StethoScope (Section 1.4.7 Table 1.2) network facilities between VxWorks and TCP/IP network systems. For many other facilities, we can refer to VxWorks programmer's guide and VxWorks Network Programmer's Guide provided with the product.

9.3.1 Basic Features

A summary of the important features of VxWorks that are essential in a sophisticated embedded system design are as follows:

1. VxWorks is a scalable OS (only the necessary OS functions become part of the applications codes, thus having reduced memory requirements). The run-time configurable feature gives a higher performance in VxWorks. The functions needed for task servicing, IPC and so on must be predefined in a configuration file included in the user codes. Pre-emptive latency minimization is there as not all the functions are at the kernel (Section 8.1).
2. RTOS hierarchy includes timers, *signals*, TCP/IP *sockets*, queuing functions library, NFS, RPCs, Berkeley Port and sockets (Section 7.15), pipes (Section 7.14), Unix-compatible loader, language interpreter, shell, debugging tools and linking loader for Unix. (These are similar to system tasks. The scheduler runs these, as it runs the ISRs.)
3. For multitasking, like MUCOS, VxWorks employs a preemptive scheduler (Section 8.10.3). VxWorks is a preemptive priority based scheduler. It provisions for 256 priority levels within 0 to 255. A task context saves fast when the CPU access changes to a higher priority. However, VxWorks offers a flexibility that there can be a set of tasks (different tasks of the same priority), which run in time-slice (round robin) mode (Section 8.10.2). Each task in a set of tasks executing round robin runs for a given number of system-clock ticks and after timeout becomes the last in a queue of the set. We can use preemptive priority and time-slicing scheduling simultaneously. (*Note:* The *preemption priority* and POSIX FIFO scheduling are identical).
4. Refer to Examples 9.19 and 9.20. The ISR interrupt flag was checked and reset for new interrupt in `ISR_CharIntr` and the ISR passed a semaphore (message) to run the waiting task `Task_ReadPortA`. *VxWorks RTOS schedules the ISRs separately and has special functions for interrupt handling* [Refer to Section 9.3.3 Table 9.10].
5. VxWorks has system-level functions for RTOS initiation and start, system clock ticks (interrupts) initiation and the ISR functions, ISR connecting to interrupt vector and masking functions. Recall Sections 7.8.4

and 8.10.3. For the critical section (section of codes which access the shared resources or variables) in the ISRs, VxWorks has the interrupts disabling and enabling functions that execute at entering and exiting the section, respectively (semaphore pending functions as event-signalling flag, and counting should not be invoked in the ISRs).

6. If a task is expecting a message from another task, which is being deleted by using the task delete function, then RTOS inhibits the deletion when an option called 'Deletion Safe' is used.
7. VxWorks has task service functions (Table 9.8). VxWorks *task creation* (initiation) by itself does not make a task in a list of active tasks. Active task means that it is in one of the three states: ready, running, or waiting (blocking or pending). VxWorks not only has the task creating, running, waiting (blocking or pending till a time out or till resource available), suspending (inhibiting task execution) and resuming, but also the functions for task spawning (creating followed by activating). VxWorks also includes the task-pending cum suspending and pending cum suspension with timeout functions. VxWorks also has the tasks, which have a state and an inherited priority (Section 7.8.5).
8. VxWorks has task delay functions and task delaying cum suspending function (Section 9.3.2).
9. VxWorks has the shared memory allocation functions and bounded ring buffer allocation for sharing the memory and buffers between the tasks and ISRs. To improve the performance of RTOS, VxWorks provides a shared address in memory to all the tasks. This helps in fast access through the pointers. A pipe need not be allocated a separate memory space. Of course, there is an attendant risk due to a possible illegal access.
10. VxWorks has IPC functions that are more sophisticated than MUCOS functions. Recall that MUCOS has identical semaphore functions for event-signalling flags, resource-acquiring keys and counting semaphores. Recall the use of the semaphore `SemKey` with `OSSemPend` and `OSSemPost` functions on `SemKey`. `SemKey` was used as a resource-acquiring key by the various tasks in Examples 9.17 to 9.20. VxWorks provides for three types of semaphores separately (POSIX-IPCs and TPCs are additional).
11. VxWorks has special features for mutual exclusiveness in a critical region. We use a mutex semaphore for the resource key when using VxWorks. [Only the task that takes the resource key through a mutex semaphore can release (give or post) the key. No other task can release it. This provides mutual exclusiveness.] One type of semaphore used as mutex has the following special features: (i) One task can be protected from being deleted by any other task. Thus, unprotected deletion cannot occur when using a mutex semaphore function with the deletion safe option. At mutex creation the option is selected to include the deletion protection. (ii) When a task acquires the key using the mutex priority inversion can be prevented with the priority inversion safe option. The priority assignment of high-priority task can now inherit (during execution of critical section) so that in case of pre-emption by an intermediate priority task, the high-priority task does not get blocked (Section 7.4). This prevents priority inversion situations during execution of the critical section.
12. The `unlock ()` and `lock ()` functions are available for the tasks and interrupts, for disabling other task but not interrupts or enabling pre-emption (task switching) as alternative to resource locking by mutex semaphore.
13. The `lock` and `unlock` functions in VxWorks do not cause the priority inversion problem (Sections 7.8.5 and 8.10.3). The priority first inherits and then returns to the original ones.
14. Let us recall Figure 7.4(a) in Section 7.7.6 to understand P and V mutex semaphores used for locking the resources. VxWorks provides for P and V semaphore functions also.
15. Unlike MUCOS, VxWorks has no separate functions for the mailbox that distinguish the mailbox from the message queue. VxWorks messages can be queued. It provides for sending messages of variable length into the queues. (MUCOS queue functions permit a pointer only for a

message and a queue is an array of message pointers.) MUCOS messages can insert such that these retrieve in the FIFO method or in the LIFO if message pointer is posted in the front instead of at the back when a message is of higher priority. VxWorks also supports this post and post-front feature as in MUCOS. There are additional special features with the message queues in VxWorks.

16. In addition to queues, VxWorks provides the IPCs through the pipe (Section 7.14) into which an ISR or task can write by invoking functions for pipe open (). Task can read from a pipe by using open () and read () functions. A VxWorks pipe is a virtual IO device functions (Refer to Section 9.3.4).
17. The features of scheduler design compatible with POSIX 1003.1b can also be used. The POSIX library can be included in VxWorks.
18. The timings taken by the various RTOS functions are similar to the ones given in Table 8.12. The following subsections describe the specific VxWorks functions.

9.3.2 Task Management Library at the System Library Header File

Each task divides into eight states (places). First four of these are also available in MUCOS tasks.

1. Suspended (idle state just after creation or state where execution is inhibited). [Refer to the use of OSTaskSuspend function for FirstTask in the step 12 of Example 9.8.]
2. Ready (waiting for running and CPU access in case scheduled by the scheduler but not waiting for a message through IPC).
3. Pending (the task is blocked as it waits for a message from the IPC or from a resource; only then will the CPU be able to process further). [Refer to OSSemPend, OSMboxPend and OSQPend functions in Examples 9.16 to 9.20].
4. Delayed (sent to sleep for a certain time interval). [Refer to the use of OSTimeDly in Examples 9.16 to 9.20].
5. Delayed + suspended (delayed and then suspended if it is not pre-empted during the delay period).
6. Pended for an IPC + suspended (pended and then suspended if the blocked state does not change).
7. Pended for an IPC + delayed (pended and then pre-empted after the delayed time interval).
8. Pended for an IPC for an IPC + suspended + delayed (pended and then suspended after the delayed time interval).

VxWorks and kernel library functions are in the header files, 'vxWorks.h' and 'kernelLib.h'. Task and system library functions are in 'taskLib.h' and 'sysLib.h'. For logging, the library function is 'logLib.h'. Library functions are given in Tables 9.8, 9.9 and 9.10. Table 9.8 lists the task-state function transitions. Table 9.9 gives the task creation, naming and control functions. Table 9.10 gives the interrupt service functions.

1. *Creating and activating a task by TaskSpawn function.* The function for task creating and activating is taskSpawn (). Prototype use is unsigned int taskId = taskSpawn (name, priority, options, stacksize, main, arg0, arg1, ..., arg8, arg9). A memory is allocated to the stack as well as to the TCB. The task identified by taskID will be assigned a stack of size stacksize with arg0 to arg9 passed to the stack. It is also assigned a TCB pointer, which points to the entry point of function main that the system executes first.

Recall that in MUCOS we were accessing a task by its priority argument, for example, OSTimeDlyResume (Task_ReadPortPriority). We access a task in VxWorks by its ID, taskID. [Refer to taskID argument at the functions in the following subsections]. taskID is a 32-bit positive number. Further, when a task of taskID = 0 is referred, VxWorks assumes that the calling task is being referred.

New task taskID gets the name, priority, options and stack size on spawning. If the NULL pointer is used as the argument for name, then conventionally the name has two characters tN (character t followed by number N) prefixed with the name.

Table 9.8 Functions for the Task State Transitions

Function	Present State	Next State	Previous Function Call before the Call or Previous States
taskResume ()	Suspended	Delayed or pended	taskInit () (task must have been initiated from the idle state)
taskResume () or taskActivate ()	Suspended	Ready	taskInit ()
taskSuspend ()	Delayed	Suspended	taskSpawn () or taskActivate ()
taskSuspend () After a timeout	Delayed	Ready	Suspended
taskSuspend () After a wait for a resource	Pended	Suspended	Ready
semGive () or msgQSend ()	Pended	Ready	Suspended
semTake () or msgQReceive ()	Ready	Pended	Delayed or suspended
taskDelay ()	Ready	Delayed	Pended or suspended
taskSuspend ()	Ready	Suspended	Delayed or pended
taskInit ()	Unknown	Suspended	
exit () ¹	Suspended	Terminated	
taskDelete () ²	Suspended	Eliminated	

¹Terminate the task.

²Terminate the task and free the memory.

Table 9.9 Task Creation, Naming and Control Functions

Function	Description
taskSafe ()	Protects the calling task from deletion
taskUnsafe ()	Permits deletion of the task protected earlier
taskDelete ()	Deletes a task
taskRestart ()	Restarts (create again) ¹ the running task as the earlier run returned error
taskActivate ()	Task activates if initialized earlier
taskSpawn ()	Creates as well as activates
taskName ()	Returns the task name that associates with the taskID passed as argument
taskNameID ()	Returns the taskID that associates with the task name passed as argument
taskIDVerify ()	Verifies if a task of taskID in the argument is available
taskIDSelf ()	Returns the taskID of the task
taskIDListGet ()	Returns an array of all ready taskIDs
taskInfoGet ()	Returns information (parameters of the task)
taskRegsGet ()	Returns the registers of the task
taskRegsSet ()	Sets the registers of the task
taskOptionsSet ()	Sets the task options
taskOptionsGet ()	Returns task options defined earlier
taskIsSuspended ()	Checks if the task is in a suspended state
taskIsReady ()	Checks if the task is in a ready state
taskTcb ()	Returns the pointer to the task control block

(Contd)

Function	Description
taskPriorityGet ()	Returns the task priority
taskLock ()	It is used at the beginning of critical section. It disables other tasks (not ISRs) and thus rescheduling. ² Priority pre-emption when the task is running
taskUnlock ()	It is used at end of critical section in a task and it enables other tasks and thus rescheduling. ³ Priority pre-emption when the task is running after the critical section
taskPrioritySet ()	Sets the priority within 0 and 255
kernelTimeSlice (int numTicks)	Defines time slice per task after enabling round robin running of the tasks. When numTicks = 50, after 50 system clock interrupts define the time-slicing period

¹Allocate the memory with the allocated stack and control blocks at the beginning.

²Scheduler cannot block when the task is running, although a higher-priority task needs to be scheduled.

³Scheduler can block when the task is running and when a higher-priority task needs to be scheduled.

1. Function 'unsigned int taskIdSelf ()' returns the identity of the calling task.
2. Using a function 'unsigned int [] listTasks = taskIdListGet ()' will return the tasked list of all existing tasks needed in the array, *listTasks*.
3. Function *taskIdVerify (taskId)* verifies whether the task *taskId* exists.

System-task has the priorities upto 99 and task-highest priority is 100 by default. User task priorities are between 101 and 255. Lowest priority means task of priority 255. In VxWorks, the priority numbering scheme is lower the number, the lower the priority. In POSIX, the priority numbering scheme is the reverse of VxWorks (the lower the number, the lower the priority). Priority numbers below 100 are used for the system-level and scheduler-level processes in VxWorks (MUCOS system-level and scheduler-level processes use the highest eight and lowest eight priorities reserved for them).

A task may use priority-functions. For example, there are three functions in *taskLib*. A function can find the *priority* using 'taskPriorityGet (*taskId*, &*priority*)'. We can change the task priority dynamically. Function 'taskPriorityPut (*taskId*, *newPriority*)' will reassign the *priority = newPriority* for the task that *taskId* identifies.

The options definable on spawning are the following.

1. An option is `VX_PRIVATE_ENV`. It means that the task must be executed in the private environment. The task is then not a public environment task.
2. An option is `VX_NO_STACK_FILL`. It means no stack fills with the hexadecimal bytes `0xEE`. (Stack-filling option allocates stack with filling of stack with byte, `0xEE` at each stack address. Use of the option facilities later finding of unused stack in memory. We can use `VX_NO_STACK_FILL` after the developed program has been debugged and unused memory spaces has been compacted by reallocation of stack sizes.) A sufficient stack size should be declared by *stacksize*. VxWorks reserve a part of the stack as a buffer to protect the stack from overflow, which may lead to unpredictable behaviour of the system. To start with, task stack fills the bytes `0xEE`. (When using simulator VxSim for the VxWorks application, VxSim adds an additional 8000 bytes to the *stacksize*. Thus, the stacks for interrupts that it simulates also become available.) A library function, 'unsigned int checkStack (*taskId*)' returns the stack usage. It first finds the unused stack area by counting the number of bytes from the end with `0xEE` and then subtracts counts from the *stacksize*.
3. An option is `VX_FP_TASK`. It means that the task must be executed with the floating-point mode of processor. (The precision is higher when using floating-point processor. Time taken is smaller when using integer number processing for floating-point operations.)

4. An option is `VX_UNBREAKABLE`. It means disable the breakpoint at the task during execution. (Breakpoints are provided for help during debugging.)

In place of option names, the hex values, `0x80`, `0x100`, `0x8` and `0x2`, respectively, can be used for the four options described above. Multiple options can be given as an argument. For example, `VX_PRIVATE_ENV | VX_NO_STACK_FILL` selects both options in a task when spawning. Like the priority task, options can also be reassigned using *taskOptionsSet ()* and *taskOptionsGet ()* functions. (The `|` sign is used between multiple options because `&` sign in C refers to the address of the succeeding variable.)

The argument *main* is the main routine address. MUCOS and for many RTOSes, *main ()* is the function, which is called by the RTOS first. Refer to Example 9.7. The *main* function is used to create a task that executes first, *FirstTask*. *FirstTask*, when it executes later, initiates the system timer. It then creates (activates as well) the application tasks and suspends itself to let the OS schedule and run the application tasks. In VxWorks, the *main ()* function analogue may be used. It is *schedule ()*.

When using VxWorks, unlike the Unix-operating system or MUCOS (), all tasks can be spawned as peers (i.e., every task is independent and no task calls or spawns another task). This means that a task can be similar to *FirstTask* in Example 9.7; a starting task (parent task) need not be spawned first. The starting task spawns daughter tasks and then suspends itself to prevent scheduling of the parent in MUCOS. The daughters in a parent task are spawned in VxWorks only when the parent is a server task that concurrently processes the daughter task.

Ten arguments from *arg0* to *arg9* can be passed into the main routine. These arguments give the start-up parameters.

An important point to note is that MUCOS *OSTaskCreate* creates the task as well as activates it (puts it in the list of tasks to be scheduled). These functions are separate in VxWorks. The *taskInit ()* and *taskActivate ()* can also be used separately in place of *taskSpawn* function. However, unless a greater control for scheduling is needed first by creation and then later by activation in an application, we prefer to use *taskSpawn* function in first instance itself.

Example 9.23 will explain the use of *taskSpawn* function.

2. *Task suspending and resuming functions*. Function *taskSuspend (taskId)* inhibits the execution of task identified by *taskId*. Function *taskResume (taskId)* resumes the execution of task identified by *taskId*. Function *taskRestart (taskId)* first terminates a task and then spawns again with its original assigned arguments. This function is used in certain situations. The priority might have been reassigned in between, and now the original priority is to be restored. Similarly, the start-up parameters might have to be restored.

3. *Deleting and protecting from deletion*. Function *taskDelete (taskId)* not only inhibits permanently the execution but also cancels the allocation of the memory block for the task stack and TCB. Deletion thus frees the memory. The task deleted is one identified by argument *taskId*. Function 'exit (*code*)' deletes the task itself but stores the code at a TCB field *exitcode*. The debugger can examine the TCB using the code.

A task should not be deleted when it has a resource key because the key can then never be released in case deleted. Protection is available to the task by using a function *taskSafe ()* before entering its critical region and using function *taskUnsafe ()* function at the end of the region. When using the mutex semaphore, we can alternatively select an option, `SEM_DELETE_SAFE` when creating it (Refer to Section 9.3.4).

Why do we use the task delete or exit function? It is because that many times system resources have to be reclaimed for reusing; memory may be a scarce resource for the given application. TCB and stack are the only resources that are automatically reclaimed. There is no saving of tasks spawned by other tasks by the kernel. Each task should itself execute the codes for the following.

1. Memory de-allocation.
2. Ensure that the waiting task gets the desired IPC.
3. Close a file, which was opened before.
4. Delete daughter tasks when the parent task executes the *exit ()* function.

4. *Delaying a task to let a lower-priority task get access.* The function 'int sysClkRateGet ()' returns the frequency (system ticks and thus system-clock interrupts per second). Therefore to delay 0.25 seconds, the function taskDelay (sysClkRateGet ()/4) is used. Recall the use of OSTimeDly and later OSTimeDly to resume in MUCOS (see Examples 9.16 to 9.20). This lets a task of lower priority to run. VxWorks taskDelay (NO_Wait) will allow other tasks of the same priority or lower to run (because the timeout interval is zero). No delayed task resumption is needed in the other priority task that runs after this function. Function nanosleep (1,000,000) will delay the task by 1ms. It is a POSIX function. Integer arguments define the number of nanoseconds for delay (sleeping).

9.3.3 VxWorks System Functions and System Tasks

The first task that a scheduler executes is UsrRoot from the entry point of usrRoot () in file install/Dir/target/config/all/usr/Config.C. It spawns VxWorks tools and does following tasks: The root terminates after all the initializations. Any root task can be initialized or terminated. The set of functions, tLogTask, logs the system messages without current task context IO. The daemon (a set of large number of functions) supports the task-level network functions. The exception-handling functions are at tExcTask. It has the highest priority. It should not be suspended, deleted or assigned lesser priority. By using it, the system reports exceptional conditions that arise during running the scheduler and tasks.

An important set of functions that are also target-specific is tWdbTask. The user creates it to service the requests from Tornado target server. It is a target agent task.

1. *System clock and watchdog timer-related functions.* VxWorks sysLib is the system library and is in a header file kernelLib.h. The following important functions are present.

1. Function sysClkDisable () disables the system clock interrupts and sysClkEnable () enables the system clock interrupts.
2. Function sysClkRateSet (TICK numTicks) sets the number of ticks per second. It thus defines the number of system clock interrupts per second. Function sysClkRateGet () returns the system ticks (system clock interrupts) per second. sysClkRateSet (100) will set the system clock tick after every 1/100 second (10 ms). This function should be called in the main () or start up FirstTask or as a starting function. There is a 64-bit global variable, 'vxAbsTicks.lower'. Variable lower that increases after each tick and vxAbsTicks. Variable 'vxAbsTicks.upper' that increments after each 2^{32} ticks. 'TICK' is defined by type def as following.

```
typedef struct (
unsigned long lower;
unsigned long upper;
) TICK
TICK vxAbsTicks;
/* Function' unsigned long tickGet ( )' returns vxAbsTicks.lower. */
```

3. Function sysClkConnect () connects a C function to the system clock interrupts.
4. Function sysAuxClkDisable () disables the system auxiliary clock interrupts and sysAuxClkEnable () enables the system auxiliary clock interrupts.
5. Function sysAuxClkRateSet (numTicks) sets the number of ticks per second for an auxiliary clock. It thus defines the number of system auxiliary clock interrupts per second. Function sysAuxClkRateGet () returns the system auxiliary clock ticks (system clock interrupts) per second.
6. Function sysAuxClkConnect () connects a C function to the system auxiliary system clock interrupts.

7. Function 'WDOG_ID wdCreate ()' creates a watchdog timer. Statement 'wdtID = wdCreate ();' creates a watchdog timer, which later identifies by wdtID. There is a function STATUS wdStart (wdtID, delayNumTicks, wdtRoutine, wdtParameter). The timer created starts on calling this function. The parameters that should pass as the arguments of this function are the following: (i) wdtID to define the identity of the watchdog timer; (ii) delayNumTicks to let the timer interrupts after the number of system clock interrupts of the system equal to delayNumTicks; (iii) wdtRoutine, a function called (not task or ISR) on each interrupt; (iv) wdtParameter an argument, which passes to wdtRoutine. A started watchdog timer, wdtID cancels on calling STATUS wdCancel (wdtID). A watchdog timer, wdtID de-allocates the memory on calling STATUS wdtDelete (wdtID).
2. *Defining time-slice interval for round robin time-slice scheduling.* Function kernelTimeSlice (int numTicks) controls the round robin scheduling and time slicing turns on and preemptive priority scheduling turns off for the tasks of equal priority. Suppose there is a system clock tick every millisecond, kernelTimeSlice (50) will set the time slice period as 50 ms. This is the time each task is allowed to run before CPU control relinquishes for another equal priority-task.

The codes #define TIMESLICE sysClkRateGet () and later sysClkRateSet (1000); kernelTimeSlice (TIMESLICE); TIMESLICE = TIMESLICE/60;' will set time slice = 1000/60 ms.
3. *Interrupt-handling functions.* Table 9.10 gives the interrupt service-related functions. Refer to Section 4.4.1. An internal hardware device (interrupt source or interrupt source group) auto-generates an interrupt vector address, ISR_VECTADDR as per the device, for example, timer. Exceptions are defined in the user software. For exceptions, ISR_VECTADDR needs to be defined by intVecSet () function. Function intConnect () connects the ISR_VECTADDR to a C function for that ISR. Device driver uses this function as follows: a lock function used as 'int lock = intLock ();' disables the interrupts. It returns an integer lock. Using the same integer as argument in the unlock function, we enable the interrupts. An unlock function used as 'intUnlock (lock);' enables the interrupts.

VxWorks provides for an ISR design that is different from a task design.

 1. *ISRs have the highest priorities and can pre-empt any running task.* It arises because ISR is needed because of internal device events (e.g., from on-chip timers) and because of exceptions and signals (user-defined software interrupts on certain error conditions).

Table 9.10 VxWorks Interrupt Service Functions

Function	Description	Function	Description
intLock ()	Disables Interrupts ¹	intUnlock ()	Enables Interrupts ²
intVecSet ()	Set the interrupt vector ³	intCount ()	Counts number of interrupts nested together
intVecGet ()	Get interrupt vector	intVecBaseSet ()	Sets base address of interrupt vector
intVecBaseGet ()	Get interrupt vector base address	intLevelSet ()	Sets the interrupt mask level of the processor
intContext ()	Returns true when calling function is an ISR	intConnect ()	Connects a C function to the interrupt vector

^{1,2} The meanings of these has been explained in the text. Also refer to Section 7.7.2. It can be used in a task critical region as a last option, because it increases the interrupt latency periods of all sources.

³ For 'exception' only. For hardware internal device interrupts, the interrupt vectors are fixed, cannot be set.

2. An ISR inhibits the execution of tasks till return.
3. An ISR does not execute like a task and does not have regular task context. It has a special ISR context.
4. While each task has its own TCB that includes its own stack pointer, unless and otherwise not permitted by a special architecture of a system or processor, all ISRs use same interrupt stack. In case of such special architecture, in place of interrupt-servicing support functions, the VxWorks stacks of ISRs can be used similar to task-stacks and codes can be defined similar to ones used in Examples 9.19 and 9.20 for the MUCOS tasks. CPUs 80x86 and R6000 are examples of the special architectures.
5. An ISR should not wait for taking the semaphore or other IPC (an ISR cannot use semTake function). An ISR should not call 'melloc ()' for memory allocation as that the function uses semaphores. ISR should not use mutex semaphore. ISR can use counting semaphore for giving (posting) semaphores.
6. ISR should just write the required data at the memory or buffer or post (send or give) an IPC or make a non-blocking write to a message queue (Section 7.12) so that it has short codes and most of its codes, which are non-critical and long time-taking, execute at the tasks.
7. ISR should not use floating-point functions as these take longer times to execute. Let these functions be passed onto the task that runs the codes later.

3.3.4 IPC Functions

Table 9.11 gives a list and description of the interprocess functions. Recall Sections 7.7.1, 7.7.2.1 and 7.7.5. The table gives the functions for semaphore, message queue and pipe.

Signals and software interrupt functions are as follows: Function 'void sigHandler (int sigNum);' declares signal servicing routine for a signal identified by sigNum and a signal servicing routine registers a signal as follows: signal (sigNum, sigISR). The parameters that pass are sigNum (for identifying the signal) and signal-servicing routine name, sigISR. Function sigHandler passes sigNum as well as an additional code. The sigHandler.codes associates with sigHandler. A pointer *pSigCtx associates with the signal context. The signal context saves CPU registers including PC and SP like an ISR context. The return from sigHandler restores the saved context.

Let sigISR be a C function that services the signal interrupt. Let its address be ISR_ADDR. Let the signal be identified by sigNum. The function 'intConnect (I_NUM_TO_IVEC (sigNum), sigISR, sigArg)' will connect the signal interrupt service routine (sigISR) for the signal identified by sigNum to the ISR_ADDR. I_NUM_TO_IVEC (sigNum) is a function that uses the argument sigNum to find the program counter (PC) assignment from the interrupt vector and uses it for ISR_ADDR. The argument sigArg passes for use by the C function.

The sigISR may call the following functions.

1. Call 'taskRestart ()' to restart the task, which generated the sigNum. Restarting assigns the original context on creation. Original PC, SP, arguments and options to a task restore now.
2. Call 'exit ()' to terminate the task, which generated the sigNum.
3. Call 'longjump ()'. This results in starting the execution from a memory location. The location is the one that was saved when function setjump () was called.

VxWorks provides three kinds of semaphores, binary (flag), mutex and counting. Mutex semaphore also takes care of the priority inversion problem on selecting an OPTION when creating it. Use of a binary semaphore provides an advantage over disabling of interrupts is that it limits (blocks) the use of the associated resources needed in sections within which that semaphore is required to be taken only.

A queue is used when the messages are put in queue for one or more waiting tasks. The VxWorks queue functions are in a library, msgQLib, which the user includes before using those. For full duplex communication between the two tasks, we should create two queues, one for each task. The mqPxLib functions are compatible with POSIX 003.1b. A detailed description of the three types of VxWorks semaphores and message queues is given next.

Table 9.11 VxWorks Interprocess Communication Functions

Function	Description
semBCreate ()	Creates a binary semaphore ¹
semMcreate ()	Creates a mutex semaphore ¹
semCCreate	Creates a counting semaphore ¹
semDelete ()	Deletes a semaphore
semTake ()	Takes a semaphore
semGive ()	Releases a semaphore
semFlush ()	Resumes all waiting blocked tasks
msgQCreate ()	Allocates and initializes a queue for the messages
msgQDelete ()	Eliminates the message queue by freeing the memory
msgQSend ()	Sends into a queue
msgQReceive ()	Receives a message into the queue ²
pipeDevCreate ()	Creates a pipe device ³
select ()	A task waits for several kinds of messages, from pipes, for sockets and serial IOs ⁴

¹We specify an option SEM_Q_PRIORITY for the order in which the semaphore should be taken if there are a number of waiting tasks for same semaphore. Specify SEM_Q_FIFO for defining to take the semaphore in FIFO mode. The task waiting since longest gets that first.

²The calling task blocks if no message is available, else the message is read by the task. According to the option parameter, insertions into a queue can be an ordered one with priority as ordering parameter or for a FIFO based read.

³Statement STATUS = pipeDevCreate ('/pipe/name', max_msgs, max_length) will create a named pipe with maximum number of max_msgs messages in maximum pipe length max_length bytes.

⁴A task blocks if the message is not available at queue or socket and pipe and buffer are empty, when the task attempts to read the queue pipe, socket and IO buffer.

1. *Creating a binary semaphore for the IPCs.* The function 'SEM_ID semBCreate (options, initialState)' creates an ECB pointed by the SEM_ID. One of the two options mentioned next must pass on calling the function.

Passing parameters: (i) *Option(s)*: One option, which can be selected is SEM_Q_PRIORITY. (ii) The other is SEM_Q_FIFO. Let us assume that at an instant, several tasks are in the blocked state and are waiting (pending) for a binary semaphore for its posting. A waiting task can take the semaphore in one of the two ways: (a) a task higher in priority than the other waiting ones takes the semaphore first and this becomes possible by SEM_Q_PRIORITY option, or (b) a task that was first blocked and reached the waiting state takes the semaphore first among the waiting ones and this becomes possible by the SEM_Q_FIFO option. The initial state of the binary semaphore passes by argument initialState. It is SEM_EMPTY when using the binary semaphore as an event-signalling flag. For the initialState, two options can be chosen: SEM_FULL in which the created semaphore's initial state is initialized as not available and already taken; and SEM_EMPTY, in which the created semaphore's initial state is initialized as available not taken. (Recall the use of semaphores SemKey and SemFlag in MUCOS.) There is mutex semaphore provision is different than the SemKey of MUCOS. In MUCOS, SemKey and SemFlag differ only initial values defined for them, the rest of the operations are identical. The semaphore initialState option defines the initial state when it was created.

Returning parameter: The function semBCreate () returns a pointer, *SEM_ID. It returns NULL in case of an error for the ECB allocated to the binary semaphore, if none is available.

Example 9.21 explains the use of semBCreate. Let us assume that ISR_CharIntr is service routine on an interrupt when a byte becomes available at a port A and a task reads that byte after waiting for the semaphore availability from ISR_CharIntr.

Example 9.21

```

1. /* Include the VxWorks header file as well as semaphore functions from a library. */
# include "vxWorks.h"
# include "semLib.h"
# include "taskLib.h"
2. /* Task parameters declarations */
.
.
.
3. /* Declare a binary semaphore to be used as flag. */
SEM_ID semBCharIntrFlagID;
4. /* Create the binary semaphore and pass the options chosen selected to it. */
semBCharIntrFlagID = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY); /* Higher priority waiting tasks
can take it first. Its initial state is not available. */
.
.
.
5. /* ISR creation codes */
6. /* Codes for ISR_CharIntr, for example, for putting the port byte into a buffer */
.
.
.
7. /* At the end, make the binary semaphore SEM_FULL from SEM_EMPTY using semGive ( ) */
semGive (semBCharIntrFlagID); /* Section 9.34 explains SemGive */
/* Other remaining codes for the ISR. */
.
.
.
8. /* End of ISR_CharIntr Codes */

```

For using binary semaphore, the following points are taken care.

- Declare initial value as SEM_EMPTY (not available)
 - Use semTake () in a task, which makes SEM_FULL to signal an event to another task
 - Use semGive () in waiting task for the event Example 9.21 showed the codes when using semBCreate ().
2. *Waiting for an IPC for binary or other type of semaphore release or check for availability of an IPC for release.* The function 'STATUS semTake (semId, timeOut)' is for letting a task wait till the release event of posting the binary or other type of semaphore. Wait till either SemId is posted (given) by a task or till time out occurs, whichever happens first. An exemplary use is semTake (semBCharIntrFlagID, WAIT_FOREVER). WAIT_FOREVER means timeout = -1 and the period is thus infinity. semTake is like OSSemPend function of MUCOS. (In MUCOS, time out = 0 means wait for ever.)

Passing parameters: (i) *semId* is the semaphore for which a suspended task waits; (ii) *timeOut* is the timeout period. One option that can be selected is WAIT_FOREVER if wait must be done for the posting of *semId*. The other option is NO_WAIT. Recall OSSemAccept function in MUCOS. Whenever this task is scheduled by the scheduler and this function is called, take the semaphore identified by *semId* if set as available SEM_FULL. Third option, wait for timeout interval.

Returning parameter: The function semTake () returns STATUS. It returns STATUS = OK in the case of success in taking the semID, else returns ERROR in the case of an error. After the semTake () function unblocks a task, it again becomes available (empty or not taken).

3. *Sending an IPC after a binary or mutex or counting semaphore release (posting).* The function 'STATUS semGive (semId)' is for letting a task post (release) the binary or other type of semaphore. After this, a waiting-task can unblock. Which task unblocks depends on the option defined while creating the semaphore posted. Unblocking can be as per option SEM_Q_FIFO or as per SEM_Q_PRIORITY.

Passing parameter: *semId*, for which there is a wait by this or another task.

Returning parameter: The function semGive () returns STATUS. It returns STATUS = OK in case of success in taking the semID, else returns 'ERROR' in case of an error that *semId* is invalid.

4. *Taking the semaphore multiple times till unavailable before the next posting.* Function STATUS semFlush (semFlagID) is for flushing. It will take the semaphore multiple times till unavailable before the next posting. It lets any waiting task not wait any further. It unblocks not only the calling but also all the other tasks waiting for taking the semaphore, *semFlagID*.

Parameter passing: The *semFlagID* passes as SEM_ID pointer at ECB that associates with the semaphore.

Returning parameter: The function semFlush () returns the STATUS and makes the semaphore state from SEM_FULL (available) to SEM_EMPTY meaning unavailable. It returns STATUS = OK in case of the success in flushing of the semaphore and making semFlagID state SEM_EMPTY. Else, it returns 'ERROR' in case of an error that on time out the semaphore is unavailable or the semaphore identity is invalid. The semFlush () function unblocks all the waiting tasks waiting for this *semFlagID* (semFlagID state = SEM_FULL earlier).

5. *Creating a mutex semaphore for the IPCs.* Mutex semaphore is needed when there is a critical section, which shares a data structure or uses a resource shared with the other tasks e.g., the bytes in a buffer between a sending task and a receiving task or using the flash memory for write operation or writing to a display device). There may also be sharing of the hardware devices or files between two tasks.

(a) An exemplary use in which we are using binary semaphore for mutual exclusion is as follows:

- SEM_ID semMKeyID;
 - semMKeyID = semMCreate (SEM_Q_PRIORITY, SEM_FULL).
- (b) The function 'SEM_ID semMCreate (options)' is for creating an ECB pointed by the SEM_ID. The uses of semTake and semGive functions are as explained earlier. Let us assume that when entering a critical region in a task, semTake (semMReadPortAKey) executes and on leaving the critical region, semGive (semMReadPortAKey) for using the mutex semaphore, semMReadPortAKey. Here, we prevent the priority inversion situation (Section 7.8.5) by choosing an option. Another option is for selecting either SEM_Q_FIFO or SEM_Q_PRIORITY. However, when selecting SEM_INVERSION_SAFE we must select the option SEM_Q_PRIORITY. (Reason for using | sign in place of & in case of passing multiple options by a single argument was given before in Section 9.3.2.)
- (c) Another example of using three options in semMCreate function argument is as follows. Here, the option prevents the priority inversion situation as well as protects the task from deletion by any other task until the semaphore is made empty (not taken) at the end of the critical region of a task. Three options are selected as follows.

```

SEM_ID semMReadPortAKey;
semMReadPortAKey = semMCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE
| SEM_DELETE_SAFE);

```

Passing parameters: (i) The use of option between SEM_Q_PRIORITY and SEM_Q_FIFO is identical to the binary semaphore, which was described earlier. (ii) The use of SEM_INVERSION_SAFE makes the critical section using the mutex safe from priority inversion situation (Section 7.8.5). It means that the created semaphore initial state is initialized as SEM_FULL (available). (iii) Recall the use SEM_DELETE_SAFE. It protects deletion of this task when in the critical region.

Returning parameter: The function semMCreate () returns a pointer, *SEM_ID for the ECB allocated to the *mutex semaphore*. It returns NULL in case of an error if none available.

Example 9.22 shows the codes using semMCreate (), semGive () and semTake ().

Example 9.22

```

1. /* Include the VxWorks header file as well as the task and semaphore functions from a library. */
# include "vxWorks.h"
# include "semLib.h"
# include "taskLib.h"
/* Declare a semaphore key to be used as mutex. */
2. SEM_ID semMReadPortAKey;
/* Create the mutex and pass the options chosen selected to it. */

semMReadPortAKey = semMCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE); /* This makes
the mutex semaphore full and available like SemKey set as 1 in MUCOS. */
semGive (semMReadPortAKey);
3. /* Task1 creation codes */
.
.
4. /* Initial Codes for the Task 1 */
.
.
/* Task 1 Initial Codes ends */
5. /* Task while loop codes */
.
.
6. semTake (semMReadPortAKey, WAIT_FOREVER); /* Critical Section (shared resource or data
section starts) */
.
.
7. semGive (semMReadPortAKey); /* Critical Section (shared resource or data section ends) */
8. /* Remaining task 1 codes */
9. /* Task2 creation codes */
.
.
10. /* Initial Codes for the Task 2*/
.
.
/* Task 2 Initial Codes end */

11. /* Task 2 while loop codes */
.
.
12. semTake (semMReadPortAKey, WAIT_FOREVER); /* Critical Section
(shared resource or data section starts) */
.
.
13. semGive (semMReadPortAKey); /* Critical Section (shared resource or data
section ends) */
14. /* Remaining task 2 codes */
/*****

```

For using mutex, the following points are taken care.

- The critical section that uses mutex for the resources protection should not be unnecessarily long and should be as short as possible.
- Declare initial value as SEM_EMPTY (available) and use options for SEM_DELETE_SAFE if some task uses taskDelete () function when using semBCreate.
- Use option SEM_INVERSION_SAFE if some priority inversion situation is likely to arise and affect the system functioning.
- Use semTake function in the same task at the beginning and semGive at the end of a critical region in which there are shared resources. semTake can be used recursively but the total number of times a semTake executes should be the same as the number of times semGive executes.
- Do not use semGive () for the mutex posting outside the critical region.
- Do not use semFlush () (its use is illegal when using the mutex semaphore).

6. *Creating a counting semaphore for the IPCs.* VxWorks counting semaphore (Section 7.7.5) is similar to the POSIX semaphore (Section 7.8.3). It increments on posting (giving) and decrements on taking (on wait-over) the semaphore. Posting this semaphore up to 256 times is permitted before it is taken. The status becomes equal to the initial value of counting semaphore only when the number of times semaphore-given equals to the number of times it is taken. The counting semaphore helps in bounded buffer problem, ring-buffer problem and consumer-producer problem (Section 7.8.3). We have seen this in Example 9.18. If initial count = 0, then a task waiting for the semaphore blocks.

The function SEM_ID semCCreate (*options*, unsigned byte *initialCount*) is for creating an ECB pointed by the SEM_ID. One of the two options must be passed on calling a function. An exemplary use is as follows: 'semCCharIntrFlagID = semCCreate (SEM_Q_PRIORITY, SEM_EMPTY);'

- SEM_ID semCID;
- SEM_ID = semCCreate (SEM_Q_PRIORITY, 0); /* To initial count = 0. */

Passing parameters: (i) One option that can be selected is SEM_Q_PRIORITY and the other is SEM_Q_FIFO. For the initial state, two options can be chosen: either initialCount should pass as 0 or it should be a fixed value. It depends on whether the semaphore is to be used for decrementing count for the tasks that are already blocked or (b) for incrementing counting.

Returning parameter: The function semCCreate () returns a pointer, *SEM_ID. It returns NULL in case of an error for the ECB allocated to the *counting semaphore*. Null if none is available.

Recall MUCOS of Example 9.18, Steps 11 to 21. Example 9.23 shows how to use VxWorks semCCreate function. It also shows the use of the other VxWorks functions for task spawning and semaphores.

Example 9.23

```

1. /* Same as Steps 1 and 2 of Examples 9.21 and 9.22. */
2. /* Declare and Create Semaphores function, its identifying variables. */
/* Declare SemFlagIID as the argument that passes to the task whenever called. Declare SemMKeyID and
SemCCountID as the mutex and counting semaphores. */
SEM_ID SemFlagIID, SemMKeyID, SemCCountID;
3. /* Create Semaphore flag and declare unblocking of the tasks priority wise. Declare initially semaphore
flag unavailability. */
SemFlagIID = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY);
4. /* Create Semaphore mutex and declare unblocking of the tasks priority wise. Initially semaphore
mutex is available by default. */
SemMKeyID = semMCreate (SEM_Q_PRIORITY); /* SEM_Q_PRIORITY | SEM_DELETE_SAFE

```

```

two options can also be used. However that prolongs the execution time. We are not using safe option as
taskDelete ( ) function is not used. Initially semaphore (mutex) is available by default. */
5. /* Create Semaphore for counting and declare unblocking of the tasks priority-wise.
unsigned byte initialCount = 0;
SemCCountID = semCCreate (SEM_Q_FIFO, initialCount);
unsigned short COUNT_LIMIT = 80; /* Declare limiting Count = 80 */
6. /* Declare and Create Semaphores task function, its variables and parameters. */
void Task_ReadPortA (SEM_ID SemFlagIID);
int readTaskID = ERROR; /* Let initial ID till spawned be none */
int Task_ReadPortAPriority = 105; /* Let priority be 105 */
int Task_ReadPortAOptions = 0; /* Let there be no option. It waits for the SemFlagIID from the ISR
(Example 3.21). */
int Task_ReadPortAStackSize = 4096; /* Let stack size be 4 kB memory */
4. /* Create and initiate a task for reading at Port A. Task name starts with 't'. The task calling-function is
Task_ReadPortA */
readTaskID = taskSpawn ("tTask_ReadPortA", Task_ReadPortAPriority, Task_ReadPortAOptions,
Task_ReadPortAStackSize, void (* Task_ReadPortA) (SEM_ID SemFlagIID), SemMKeyID,
SemCCountID, &initialCount, COUNT_LIMIT, 0, 0, 0, 0, 0, 0); /* Pass SemFlagIID as the argument of
task function and pass other arguments SemMKeyID and SemCCountID as arg0 and arg1. Remaining
arguments are 0s. */

/* Other Codes */

5. /* The codes for the Task_ReadPortA redefined to use the key, flag and counter*/
static void Task_ReadPortA (SEM_ID SemFlagIID) {
6. /* Initial assignments of the variables and pre-infinite loop statements that execute only once */
/* Declare the buffer-size for the characters countLimit = 80 */
intCount = 0;

7. while (1) { /* Start an infinite while-loop. We can also use FOREVER in place of while (1). */
8. /* Wait for SemFlagIID state change to SEM_FULL by semGive function of character availability
check task */
semTake (SemFlagIID, WAIT_FOREVER);
9. /* Take the key so that another task, port decipher does not unblock. That task needs SemMKeyID to
unblock and run */
semTake (SemMKeyID, WAIT_FOREVER); /* SemMKeyID is now not available and the critical region
starts */
10. if (Count >= COUNT_LIMIT) {

}; /* End of Codes for the action on reaching the limit of putting the characters into the buffer */

```

```

11. /* Codes for reading from Port A and storing a character at a queue or buffer*/
.
.
.
12. semGive (SemCCountID); Count++; /*Let counting semaphore value increase because one character
has been put into the buffer holding the character stream. initialCount incremented because of the need to
compare later with the COUNT_LIMIT*/
13. semGive (SemMKeyID); /* Critical region ends. Release the mutex SemMKeyID to let next cycle of
this loop start */
14. ; /* End of while loop*/
15. } /* End of the Task_ReadPortA function */
A point to be noted is that there is no provision for setting the limit up to which a
counting semaphore can be given (posted). It is fixed at 65,535 in MUCOS. Therefore, we
have to use COUNT_LIMIT variable, and compare with the Count variable, which
increments after each function 'semGive (SemCCountID)' call.
/*****

```

7. Using POSIX semaphores. POSIX semaphore functions can also be used for the VxWorks counting semaphores. The function 'semPxlLibInit ()' initializes the VxWorks library to permit use of these. The functions are sem_open (), sem_close () and sem_unlink () to initialize, close and remove a named semaphore, respectively.

The functions sem_post () and sem_wait () unlock and lock a semaphore. The actions of these two are similar to semGive and semTake in the VxWorks counting semaphore or OSSemPost and OSSemPend functions of MUCOS semaphores. The function sem_getvalue () retrieves the value of a POSIX semaphore. The actions of the function is similar to OSSemQuery in MUCOS.

POSIX semaphore functions sem_init () and sem_destroy () initialize and destroy. Destroy means de-allocate associated memory with the semaphore ECB. (This effect is not the same as first closing a semaphore and then unlinking it by sem_close and sem_unlink). Remember that no deletion safety like VxWorks mutex is available before using these functions. VxWorks semaphores have the additional following features. (i) Options of protection from priority inversion and task deletion. (ii) Single task may take a semaphore multiple times and recursively. (iii) Mutually exclusive ownership can be defined. (iv) Two options, FIFO and task priority for semaphores wait by multiple tasks.

sem_trywait () is to try to lock a semaphore if not available and locked by other task.

8. Creating a message queue for the message IPCs. The function 'MSG_Q_ID msgQCreate (int maxNumMsg, int maxMsgLength, int qOptions)' is used for creating an ECB pointed by the MSG_Q_ID. One of the two options mentioned next must pass on calling the function. The memory allocation to the buffer, which holds the bytes for messages is according to maxNumMsg and maxMsgLength (parameters for maximum number and length).

A point to be noted is that the message pointer passed into an array of message pointers in MUCOS (Example 9.20).

In a message queue, the maximum number of messages = $2^{31} - 1$ and the maximum number of bytes in each message is $2^{31} - 1$ bytes.

Passing parameters: (i) To the function, maxNumMsg passes the maximum number of messages that can be sent to the queue. (ii) maxMsgLength passes the maximum number of bytes permitted to be sent as a message. (iii) One option that can be selected is MSG_PRI_NORMAL, when the message is sent into the queue for receiving as a FIFO. The first message sent is then read first. The other option is MSG_PRI_URGENT. When the message is sent into the queue with this option, the message is received as LIFO. Urgent messages

like error logins are sent with this option selected. The last message sent is then read first. (iv) Another option that can be selected is MSG_Q_PRIORITY. The other is MSG_Q_FIFO. Let us assume that at an instant, several tasks are in the blocked state and are waiting (pending) for a message from the same queue for its posting (sending). A waiting task can take from the queue in one of two ways. (a) A task higher in priority than the other waiting ones, takes the message from the queue first. This becomes possible by MSG_Q_PRIORITY option. (b) A task, which first blocked and reached the waiting state, takes the message from queue first among the waiting ones. This is possible by MSG_Q_FIFO option.

Returning parameter: The function msgQCreate () returns a pointer *MSG_Q_ID. It returns NULL in case of an error for the ECB allocated to the *message queue*. Null if none is available or on error.

Consider the *tasks* in a hand-held device. The *tasks* post the messages into a queue using msgQsend (). A buffer for writing into the flash memory in a task receives the bytes using msgQReceive ().

9. *Sending an IPC after a message is sent into the queue.* The function 'STATUS msgQSend (*msgQId*, &*buffer*, *numBytes*, *timeOut*, *msgPriority*)' is used for letting a task send into the queue.

Passing parameters: (i) The queue identifies by *msgQId*. (ii) Message posts to an addressed *buffer*. The number of bytes sent into the buffer = *numBytes*. (iii) The *timeOut* is the period till posting of the message is awaited in case the queue is full. (iv) *msgPriority* is specified as MSG_PRI_NORMAL or MSG_PRI_URGENT, depending upon whether the message is inserted later on to retrieve in the FIFO or LIFO mode, respectively.

Returning parameter: The function msgQSend () returns STATUS. It returns STATUS = OK in case of success in taking the *msgQId*, else returns 'ERROR' in case of an error that *msgQId* is invalid.

Example 9.24 shows how to use the queue functions for create and send.

Example 9.24

```

1. /* Include the header files in Example 9.22 Step 1 as well as queue functions from a library. */
2. # include "msgQLib.h"
3. /* Declare message queue identity and message data type or structure. */
4. MSG_Q_ID portAInputID;
5. unsigned byte portAdata;
6. void * message; /* Pointer for the message buffer */
7. /* Create the message queue identity and pass the parameters and options chosen selected to it and let
   the maximum number of messages be 80 and message be of 1 byte each.
   Let us assume that Task_ReadPortA reads a byte from port A and sends it to another task that receives
   the messages from a queue after waiting for the queue message availability. */
8. PortAInputID = msgQCreate (80,1, MSG_Q_FIFO | MSG_PRI_NORMAL)
9. /* Task Creation Codes as in Example 9.23. */
10. .
11. .
12. /* Start Codes for Task_ReadPortA. */
13. void Task_ReadPortA {
14. .
15. while ( ) {
16. semTake (SemFlagIID, WAIT_FOREVER);
17. /* Take the key to not let port-decipher task unblock and run. Therefore, that task also needs
   SemMKeyID for running. */
18. semTake (SemMKeyID, WAIT_FOREVER); /* SemMKeyID is now not available and
   the critical region starts */

```

```

19. .
20. .
21. /* At the end, the send the byte, which is read at Port A. It is sent as a message to queue,
   portAInputID.
22. *message = portAData;
23. msgQSend (portAInputID, &message, 1, NO_WAIT, MSG_PRI_NORMAL);
24. /* Other remaining codes for the task. */
25. .
26. .
27. }
28. /* End of Task_ReadPortA Codes */

```

10. *Waiting in a queue for availability of message.* The function 'int msgQReceive (*msgQId*, &*buffer*, *maxBytes*, *timeOut*)' is used for letting a task wait till sending (posting) of a message. Wait till either *msgQSend* function sends the message in a task or till a time out occurs, whichever happens first.

Passing parameters: (i) Whenever the scheduler schedules this task and this function is called, there is a wait for a message pointed by *msgQId*. (ii) *buffer* is the address of the buffer. (iii) *maxBytes* is the maximum number of bytes acceptable per message by msgQReceive function. (iv) *timeOut* is the time out period. One option that can be selected is WAIT_FOREVER if wait must be done for sending the message by msgQSend function. The other option is NO_WAIT. Recall OSQFlush function in MUCOS. NO_Wait option is simply used for checking the availability of a message. Message is received from buffer if available else task runs other succeeding codes. In case of error message, for example, there is no waiting especially for the message. Only check for error message is needed.

Returning parameter: The function msgQReceive () returns an integer for the number of bytes retrieved from the buffer address.

Example 9.25 shows how to use msgQReceive function.

Example 9.25

```

1. to 9. /* Codes as per Steps 1 to 9 in Example 9.24 */
10. /* The codes for the Task_MessagePortA */
static void Task_MessagePortA (void *taskPointer) {
11. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
int maxBytes = 80;
12. while (1) { /* Start an infinite while-loop. */
13. /* Wait for a Queue Message sending or availability. */
msgQReceive (msgQId, message, maxBytes, WAIT_FOREVER); /* WAIT_FOREVER means
timeout = -1 and the period is thus infinity. */
14. /* Other remaining Codes */
.
.
15. }; /* End of while loop*/
16. } /* End of the Task_MessagePortA function */
/*****

```

11. Using POSIX queues. Important points in using the POSIX queues are as follows.

- The function `mqPxLibInit ()` initializes the VxWorks library to permit use of the POSIX Queues.
- The functions `mq_open ()`, `mq_close ()` and `mq_unlink ()` initialize, close and remove a named queue.
- The function `mq_setattr ()` sets the attribute of a POSIX queue.
- The functions `mq_send ()` and `mq_receive ()` unlock and lock a queue.
- The function `mq_notify ()` signals to a single waiting task that the message is now available. The notice is exclusive for a single task, which has been registered for a notification (registered means later on takes note of the `mq_notify`). This provision is extremely useful for a server task. A server task receives the notification from a client task through a signal-handler function (like an ISR).
- The function `mq_getattr ()` retrieves the attribute of a POSIX queue.
- The POSIX queue function `mq_unlink ()` does not destroy the queue immediately but prevents the other tasks from using the queue. The queue will get destroyed only if the last task closes the queue. Destroy means to de-allocate the memory associated with queue ECB.

VxWorks queues have the additional following features. (i) Time out option can be used. (ii) Two options, FIFO and task priority for queues wait by multiple tasks. POSIX queues have the additional following feature: task notification in case a single waiting task is available and there can be 32 message priority levels in place of one priority level URGENT in VxWorks.

12. Creating a pipe device for read-write in IPCs. When a task creates, a taskID allocates (Section 9.3.1). When using the task-related functions, the number facilitates task identity. Similarly, a pipe (Section 7.14) or socket (Section 7.15) or file (Section 8.6.2) when creates, a file descriptor data structure is created and a number, for example, `fd` is assigned to identify the device created. The number is assigned after examining a set of the numbers already allocated. When using the device-related functions, the number facilitates the device identity. The device function examples are open or read or write or get attribute or set or attribute close (Section 8.6).

A pipe in VxWorks is a FIFO queue, which is managed not by queue IPC functions but by the device-driver functions. VxWorks has management functions for a pipe-driver (like a device driver) `pipedr`. This is analogous to the named pipe driver in Unix. Pipes also implement the unidirectional link between a set of tasks.

Function `pipeDevCreate ('/pipe/pipeName', maxMsgs, maxMsgBytes)` creates a pipe device named `pipeName` for maximum `maxMsgs` messages. Each message can be of maximum size `maxMsgBytes`. It enters into a list of devices on creation. `devs ()` function retrieves the list of devices with the device number allotted to each device including pipe devices.

Consider an example for creating a pipe named as `pipeUserInfo`. Assume that it can have a maximum of four messages: user name, password, telephone number and e-mail ID. Each of these can be of a maximum size of 32 bytes only. A global variable `fd` is an integer number for a file descriptor that identifies a device among a number of devices at the IO system. The device can be a file or pipe or socket or other device. Example 9.26 explains the codes for creating, writing and reading.

Example 9.26

```
1. # include "fioLib.h" /* Include the IO library functions. */
   pipeDrv ( ); /* Install a pipe driver. */
2. /* Declare file descriptor. */
   int fd;
3. /* Mode refers to the permission in an NFS (Network File Server). Mode is reset as 0 for unrestricted
   permission. */
   int mode;
4. /* Create pipe named as pipeUserInfo for 4 messages, each 32 bytes maximum. */
```

```
pipeDevCreate ("/pipe/pipeUserInfo", 4, 32); mode = 0x0;
Messages can be written into a pipe by the function by first opening a pipe, device and then writing into
that. The function for opening is open ('/pipe/pipeUserInfo', rdwrFlag, mode). We define flag = O_RDWR,
which permits both read and write. Flag O_RDONLY permits the read only option and flag O_WRONLY
permits the write only option. Remember that after opening a pipe, when we finish using it, we must use
the function 'STATUS close ( )'. Writing to a pipe is analogous to writing on an IO device. To write, the
coding is as follows.
```

```
5. /* Open read-write device using, a pipe named as pipeUserInfo with mode = 0 for unrestricted permission.
   */
```

```
fd = open ("/pipe/pipeUserInfo", O_RDWR, 0); /*A file descriptor is used for a file or pipe or socket or
serial device or other type of device. */
```

```
6. /* Write a message, info of lBytes. */
```

```
char [ ] info; Let the message be a string of characters. */
int lBytes;
```

```
lBytes = 12;
```

```
write (fd, info, lBytes);
```

The message can be read from an open pipe by the function 'int read (fd, &buffer, lBytes)'.
Reading from a pipe is analogous to reading from an IO device as per the file descriptor.

To read, the coding is as follows.

```
7. int numRead; /* An integer to indicate the number of bytes successfully read. */
```

```
lBytes = 12; /* Let the Message bytes to read = 12*/
```

```
numRead = read (fd, info, lBytes);
```

13. Finding the set of opened devices at an instance from the number of devices in the system. Recapitulate event functions in Section 8.4. Assume that all bits are cleared at the time of creation of an event flag group. A register saves the bits, which set on the occurrences of the events from the number of sources. Each event sets one bit. A task can wait for setting any or all events in the group.

Similarly, there is `FD_SET`. `FD_SET` sets a file descriptor function. Each device in a system having a number of devices set a bit in the `fdSet`. (A file descriptor is used for a pipe or socket or serial device or other type of device. Let a file descriptor `fd = n`; there is an array of bits in which the `n`th bit corresponds to `fd = n`.)

Now function, `FD_SET (n, &fdSet)` defines the data structure `fdSet`, when executed will make `n`th bit = set. `FD_SET (n, &fdSet)` makes the `n`th bit = set. `FD_CLR (n, &fdSet)` will make `n`th bit = clear. `FD_ZERO (&fdSet)` makes all bits of array = 0. `FD_ISSET (n, &fdSet)` returns true if `n`th bit in the array is set and false if reset at `fdSet`.

Now, let us examine how a task selects and finds the number of active devices at an instance. Task finds whether a pipe, `pipeUserInfo` is active or a `pipeResponse` is active. The function to select is 'int select (`numBitWidth`, `pointerReadFds`, `pointerWriteFds`, `pointerExceptFds`, `pointerTimeOut`). The arguments passed are the following: `numBitWidth` = number of bits to examine in the array of bits at two pointers, `pointerReadFds` and `pointerWriteFds`. Examination is as per the value at a structure that stores NULL if wait forever or a value for time out. Timeout is the number of system clock interrupts up to which the wait is done. The function select () blocks till at least one device in the array of devices is ready or till time out, whichever happens first. Select clears all the bits that correspond to the devices that are not ready and returns the number of active devices. It returns `ERROR` on an error.



Summary

The following is summary of what we learnt in this chapter.

- The basic functions in the RTOSes and types of RTOSes.
- It is a necessity to use a well-tested and debugged RTOS in a sophisticated multitasking embedded system. MUCOS and VxWorks are the two important RTOSs.
- Code elegance is one of the best in MUCOS and the provision of powerful functionalities is one of the best in VxWorks.
- MUCOS task creating and deleting, suspending and resuming functions are used for the task controlling and scheduling functions.
- There are functions for initiating the system timer in MUCOS. Starting a multitasking system by a first task and later suspending it forever is shown as a technique in programming for a multitasking system.
- MUCOS handles and schedules the tasks and ISRs and handles pre-emptive scheduling.
- There are delay and delay resume functions in MUCOS. These are shown to be useful for letting a low priority task run.
- MUCOS has the IPC functions for the event flag group, semaphore, mailbox and queue. The simplicity feature of MUCOS is that the same semaphore functions are used for binary semaphore, for event-signalling flag, for resource key and counting.
- MUCOS has mailbox functions and a simple feature that a mailbox has one message pointer per mailbox. There can be any number of messages or bytes, provided the same pointer accesses them.
- MUCOS has queue functions. A queue receives from a sender task an array of message pointers. Message pointers' insertion can be such that later on it can retrieve in the FIFO method as well as in the LIFO method from a queue. It depends on whether the post was used or post-front function was used, respectively. This helps in taking notice of a high-priority message at the queue.
- VxWorks is a popular broadly focused RTOS because of its powerful development tools, support to advanced processor architectures and device software optimization.
- VxWorks supports the multiple file systems, systems that enable advanced multimedia functionality and multitasking environment using VxWorks scheduler, POSIX scheduler or in-house developed scheduler.
- VxWorks supports ability to run two concurrent OSEs on a single processing layer (e.g., VxWorks and Windows or VxWorks and embedded Linux).
- Instead of one create function, VxWorks has three functions: task create, task activate and task spawn (create and activate).
- VxWorks also provides for system timer functions, system auxiliary clock functions, watch dog timer functions, delay and delay resume functions.
- VxWorks handles and schedules the functions for the tasks and ISRs differently. It allocates highest priorities for the ISRs over the tasks and provides nested ISRs, and thus a common stack of the ISRs.
- VxWorks has an IPC called *signal*. It is used for exception handling or handling interrupts from the tasks. VxWorks has signal-servicing routines. A signal-servicing routine is a C function. It executes on occurrence of an interrupt or exception. A connect function connects the function with the interrupt vectors.
- Exceptions are the software interrupts. A signal setting is equivalent to a flag setting in case of hardware interrupts.
- VxWorks provides for pre-emptive scheduling as well as round robin time-sliced scheduling of tasks assigned equal priority.
- VxWorks provides for two ways in which a pending task among the pending tasks can unblock. One is as per the task priority and another is as a FIFO when accepting (taking) an IPC.
- VxWorks has three different semaphore functions for use as IPC for the event-signalling flag, resource key and counting semaphore. VxWorks also supports POSIX semaphores. VxWorks, instead of queuing the message pointers only, provides for queuing of the messages. Queues can be used as LIFO as in MUCOS. VxWorks also supports use of pipes and POSIX queues. VxWorks pipes are the FIFO queue that can be opened and closed like a file device. Pipes are like virtual IO devices that store the messages as FIFO.



Keywords and their Definitions

- Counting semaphore** : It is a semaphore that increments when an IPC is given by a task or a section of the task. It decrements when a waiting task unblocks and starts running.
- Event-signalling flag** : A flag, which sets on occurrence of an event and resets on response to the event. A binary semaphore or event flag group bit can be used as the *event-signalling* flag.
- Exception handling** : Executing a function on receiving a signal. Error is also handled by using an exception-handling function.
- FD set** : The file descriptors of all devices exist into a data structure FD set. The bits corresponding to active devices are set and inactive devices are cleared.
- File descriptor** : A pipe or socket or file (Section 8.6.2) when creates, a file descriptor a data structure is created and a number, for example, fd is assigned to identify the device created.
- File device** : Memory block(s) in which the file read, file write, file open and file close functions operate as in case of file on a disk.
- Mailbox** : An IPC in the event control block into which a task or ISR posts a message pointer, which is retrieved by another task waiting for that.
- Message queue** : An IPC in the event control block into which a task posts the messages at the tail pointer or urgent messages at the front pointer, which are retrieved by another task waiting for that.
- Pipe** : A device from which one task gets the messages and the other task puts the messages. VxWorks *pipe* is a FIFO queue in which the IO device functions operate. Putting and getting messages from a pipe is like the one from a file.
- POSIX queues** : IPC queue functions as per POSIX standard functions.
- POSIX semaphores** : Semaphore functions as per the IEEE POSIX standard functions.
- Resource key** : A semaphore that resets on the start of execution of a critical region code and sets on finishing these.
- Signal** : Flag-like intimation to RTOS for development of certain situations during a run that need urgent attention by executing an ISR function-like signal handler.
- Sophisticated multitasking embedded system** : A system that has multitasking needs with multiple features and in which the tasks have deadlines that must be adhered to.
- System timer** : A system clock that can be set to interrupt at preset intervals. The time is updated regularly and the system interrupts regularly. RTOS also gets control of CPU to examine if any pre-emption or rescheduling is needed. Task priority provides priority for system timer functions, delay functions and delay resume functions.
- Task delay** : Let a task wait for a minimum time defined by the number of system ticks passed as an argument to the delay function.
- Task spawning** : Task creation and activation.
- MUCOS** : An RTOS μ C/OS-II from Micrium of Jean J. Labrosse.
- VxWorks** : An RTOS from Wind River@ Systems.
- Task creation** : Task is allotted a TCB and an identity. Creation also *initiates* and *schedules* on creation in MUCOS.
- Task deletion** : Task no longer has the TCB and is ignored till created again.
- Task resumption** : Task, which was delayed or suspended, can now be scheduled when the turn comes.
- Task suspension** : A task unable to run its codes further.

TCB : Task control block, which has the task parameters so that on task switching the parameters remain saved and when RTOS re-switches it back, the task can run from the point at which it left. Task is thus an independent process.

Well-tested and debugged RTOS : An RTOS, which is thoroughly tested and debugged in a number of situations.



Review Questions

1. What are the advantages of a well-tested and debugged broad-focussed RTOS, which is also well trusted and popular? (*Hint*: Embedded software has to be of the highest quality and there should be faster software development. Need for complex coding skills required in the development team for device drivers, memory and device managers, networking tasks, exception handling, test vectors, APIs and so on.)
2. How does a mailbox message differ from a queue message? Can you use message queue as a counting semaphore?
3. Explain ECB.



Practice Exercises

Note: Exercises 5 to 12 pertain to MicroC/OS-II and 14 to 23 to VxWorks.

4. Search the web (e.g., www.eet.com) and find the latest top RTOS products.
5. Draw five figures showing models for five examples 9.16 to 9.20 in Section 9.2 for event-flag semaphore, mutex, counting semaphore, mailbox and queue interprocess communication.
6. Draw the figures to show the models for interprocess communication at processes in digital camera, ACVM and orchestra playing robot examples in Sections 1.10.4, 1.10.2 and 1.10.7, respectively.
7. Classify and list the source files, which depend on the processor and those that are processor-independent?
8. Design a table that gives MUCOS features.
9. MUCOS has one type of semaphore for using as resource key, as flag, as counting semaphore and mutex. What is the advantage of this simplicity?
10. How do you set the system clock using function void OSTimeSet (unsigned int counts)?
11. When do you use OS_ENTER_CRITICAL () and OS_EXIT_CRITICAL ()?
12. How do you set the priorities and parameters, OS_LOWEST_PRIO and OS_MAX_TASKS, for pre-emptive scheduling of the tasks?
13. A starting task is first created, which creates all the tasks needed, initiates the system clock and then that task is suspended. Why must this strategy be used?
14. VxWorks kernel includes both POSIX standard interfaces and VxWorks special interfaces. What are the advantages of special interfaces for the semaphores and queues?
15. How do you initiate round robin time-slice scheduling? Give 5 examples of the need for round robin scheduling.
16. How do you initiate pre-emptive scheduling and assign priorities to the tasks for scheduling? Give 10 examples of the need for pre-emptive scheduling.
17. How do you use signals and use function void sigHandler (int sigNum), signal (sigNum, sigISR) and intConnect [I_NUM_TO_IVEC (sigNum), sigISR, sigArg]? Give five examples of their uses.
18. How do you create a counting semaphore?
19. OS provides that all ISRs share a single stack. What are the limitations it imposes?
20. How do you create, remove, open, close, read, write and IO control a device using RTOS functions? Take an example of a pipe delivering an IO stream from a network device.
21. Explain the use of file descriptor for IO devices and files.
22. How do you let a lower priority task execute in a pre-emptive scheduler? Give four coding examples.
23. How do you spawn tasks? Why should you not delete a task unless memory constraint exists?
24. Write exemplary codes for using the POSIX functions for timer, semaphores and queues.

REAL-TIME OPERATING SYSTEM PROGRAMMING-II: WINDOWS CE, OSEK AND REAL-TIME LINUX FUNCTIONS

10

R

e

c

a

f

f

We have discussed the following important points relating to the RTOSes in the previous chapters.

- An RTOS has basic functions (services) of process (thread or task) and memory management, enables sharing of resources and data, enables use of timers and system clock, does time allocation and de-allocation to attain best utilization of the CPU time under the given timing constraints for the tasks, manages interprocess communication (IPC) (communication between the ISRs, tasks and OS functions) and IO subsystems, manages devices and device drivers and provides for real-time task-scheduling and interrupt-latency control. RTOS enables hard and soft real-time operations. RTOS provides a predictable timing behaviour of the system (in most cases in case) and a predictable task synchronization using the priorities allocation RTOS provisions for priorities inheritance.
- A programmer uses RTOS functions in application software and APIs. RTOS also enables asynchronous IOs. RTOS functions synchronize the concurrent

running of processes (tasks or threads), fast-level ISRs, slow-level interrupt service threads (ISTs).

3. MUCOS and VxWorks are the two important RTOSes. MUCOS and VxWorks functions provide programming for the ISRs and tasks (processes).
4. Code elegancy and reliability is one of the best in MUCOS and provision of powerful functionalities is one of the best in VxWorks. VxWorks is a popular broadly focused RTOS because of its powerful development tools, support to advanced processor architectures and device software optimization. VxWorks supports the multiple file systems, systems that enable advanced multimedia functionality and multitasking environment using VxWorks scheduler, POSIX scheduler or in-house developed scheduler. VxWorks supports ability to run two concurrent OSes on a single processing layer.

We will discuss the following popular RTOSes in this chapter:

1. Windows CE for consumer electronic systems and devices
2. OSEK—a reliable RTOS for the automotive electronic system
3. Open source real time Linux
4. RTLinux

10.1 WINDOWS CE

Windows CE (WCE) is an RTOS for handheld computers and mobile systems, developed by Microsoft. Microsoft designer perception for using the word CE is that CE stands for the properties that it is *compact, connectable, compatible, companion* and *efficient*. WCE can also be perceived as Windows for consumer electronics systems, however applications do not limit to consumer electronics systems.

WCE is nowadays one of the most popular OSes for the handheld systems.

An enhancement of WCE is Windows CE.NET. [Dot NET framework provides for compiling the managed code. Managed code is one that is compiled in CIL (common intermediate language). It gives platform-independent CPU neutral compilation as the byte codes. A run-time environment converts the byte code instructions into the native machine and platform instructions. When different CPU embeds into the system, the different run-time environment is used. Therefore, bytes code can run on different platforms and be distributed. At run time, the .NET run-time verifies the executing native environment, data source and destination types, within range array indices and other functionalities. The code becomes robust.]

Windows CE.NET is used as a real-time operating system for handheld computers and mobile systems. WCE.NET is described in detail in Douglas Boling *Programming Microsoft WINDOWS CE.NET*. Microsoft, USA, 2003. Section 10.1.1 describes in brief the basic features and functions in WCE. Figure 10.1 shows the basic features.

10.1.1 Windows CE Features

WCE platform provides the following features.

1. Provides a Windows platform for the systems, which have resource constraints of power, memory, touch screen or display screen size and processing speeds. Windows platform enables a user to feel, look and interact with the system using GUIs in a manner similar to a PC running on Windows.
2. It is an open, scalable and small-footprint 32-bit OS.
3. Enables running of PocketPC applications such as Outlook, Explorer, Pocket Power-Point, Pocket-Word and Pocket-Excel for mailing, Internet, PPT and slide shows and office applications. PocketPC is a handheld PC based on WCE. Latest version CE 6.0 is for home as well as office systems and gives cellular networks connectivity. WCE using systems enable running of multimedia, voice user interfaces (VUIs), smartphone and game applications. (Voice user interfaces facilitate interaction and command inputs using stored voice or tunes, and voice-command inputs from user.)

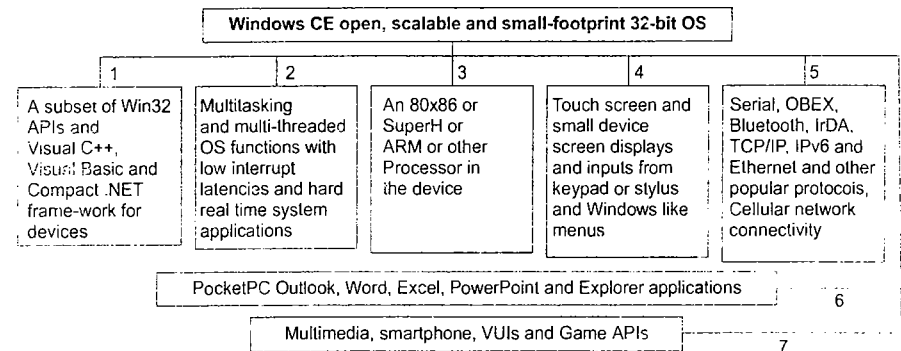


Fig. 10.1 Windows CE basic features

4. Processor of a WCE using system may be an 80x86 or SuperH or ARM or SH4 or MIPS. WCE system-performance fine tunes to the processor.
5. Functions as multitasking and multithreaded OS. Multitasking means that there are a number of processes which can run concurrently. Each process can have multiple threads and has at least one thread. A thread is a basic unit of computation using the resources which are provided by WCE. WCE provides support to 256 levels of thread-priorities. WCE also provides for adjustable time quantum for the threads. Threads having equal priorities are assigned a time slice (Section 8.10.2) each during the system run.
6. WCE has low interrupt latencies due to use of ISTs in addition to ISRs. The ISTs are put in priority queue of threads waiting for execution. (Sections 4.2.3 and 8.7.3). An IST is the slow-level interrupt service thread of a fast-level ISR. This gives WCE functionalities for hard real-time scheduling and interrupt latency control. WCE supports nested ISRs.

7. Enables use of a subset of Win32 APIs, Visual C++, Visual Basic and a .NET Compact Framework (in Windows CE.NET). Software can be created using Microsoft Visual Studio 2005. The code is developed in Visual C++. Software for mobile devices (systems) with smartphone and cellular connectivity are developed using Windows Mobile SDK.
8. The programs can be tested on the familiar PC having 80x86 processor before embedding into the system. Emulation edition can be used to emulate an application on a PC. Evaluation kit is used to test the program before embedding into the actual system hardware with it or another processor.
9. Supports touch screen. A touch screen displays as well as accepts input through a stylus. (Stylus is a writing pencil-core-shaped object for a user of device or system as an alternative to mouse and keyboard. The user touches the tip at the displayed menu or displayed keypad on the screen to enter the commands or text, respectively.)
10. Network and communication protocols support, for example, OBEX (object exchange), Bluetooth, IrDA, UDP, TCP/IP, IPv6 and ethernet and many other popular protocols.
11. Shared source and source code access are provided by Microsoft. There is *componentization*. There are two software layers. One sublayer consists of Microsoft developed source codes of WCE kernel and is shared with the system or device manufacturer. Then the manufacturer adds the remaining part of the kernel according to the system hardware. The remaining part is the hardware abstraction layer. Further, Microsoft gives freedom to modify kernel-level objects also without sharing them with the Microsoft.
12. Supports power manager, virtual memory, file-based registry and several file systems (e.g., flash memory-based file system). (Power manager is software to reduce the power dissipation by reducing clock speed or running Wait or Stop instruction or optimizing use of caches or stopping screen or reduced intensity displays after limited wait for user input. Virtual memory are the addresses allocated for stored programs which may be of size more than the physical memory size. Section 8.6.2 explained the file system concept.)

Table 10.1 gives the new enhanced features of WCE in Windows CE 6.0, Windows Mobile 6 and Windows Automotive 5.0.

10.1.2 Windows CE Programming

Following are the differences in programming with WCE and Windows. WCE provisions for the following.

1. Win32 APIs subset only (e.g., no environment-related functions and environment blocks, no current directory information at the subset).
2. Small screen system.
3. Touch screen system.
4. No hard-disk, low RAM memory and use of ROM and flash memory in the system.
5. System processor can be x86 or ARM or SuperH, or any other.
6. Unicode 16-bit characters (unsigned *short*) so that international characters and languages can be used.
7. Reduced number of Windows controls in WCE compared with the personal computer.
8. New format Windows Controls (classes which support a number of GUI functions of command, menu, tool bars provided in one line due to small screen) and the new Controls [e.g., for date and time picker, calendar picker, edit to auto capitalize first character of a word when keying-in, virtual keyboard and organizer (e.g., task-to-do)].
9. Device drivers imported as the DLLs (Section 10.1.7).
10. Not support for the Handle inheritance and certain security attributes.
11. Componentization (Section 10.1.1).

Table 10.1 Windows CE 6.0, Windows Mobile 6 and Windows Automotive 5.0 Enhanced Features

S.No.	Feature	Description
1	Windows CE 6.0	Number of processes 2 ¹⁶ (earlier 32), lower virtual memory (VM) 2 GB (earlier 32 MB) addresses per process, upper 2 GB of the kernel VM space, device drivers running in both user mode and kernel mode (Section 8.1.2), system components which now run in kernel space later at run time, new security infrastructure, 802.11i and 802.11e Wireless LAN support, new cell core for cellular networks and easy data connections, UDF 2.5 and exFAT file system, IDE integrated with Visual Studio 5.0 and targeted to enterprise specific tools such as industrial controllers and consumer electronics devices.
2	Windows Mobile, 6 second edition	Office Mobile 2007, smartphone with touch screen, improved Bluetooth stack, VoIP with AEC (acoustic echo cancellation), support to encryption of data stored in external removable storage cards, uses smart filter for fast files, e-mail, contacts and songs search, and can be set as modem for laptop.
3	Windows Automotive 5.0	Based on Windows CE 5.0 and building blocks for automobile off board service, automotive user interface toolkit (AUITK), expanded virtual memory support to enable the creation of complex 3-D graphics, and advanced navigation displays, enhanced power management and faster cold-boot times, real-time traffic updates, directions to the cheapest gas and improved performance.

A Windows-based application program is written to respond or activate or changes from the current state on pushing of notification(s) from the OS. A notification occurs on an event. The notification sends the message (Sections 8.1.2 and 8.4) to the Windows application program. Messages are placed in queue (Section 7.12) for the Windows of the application program. The OS monitors all input sources [e.g., stylus tap, virtual (on-touch screen) or physical key press]. The OS notifies that a key has been pressed or a button has been clicked or command has been received for redrawing the Windows screen. [In Unix: it is the other way round. The application program asks for the input(s) from the OS for a character or commands or inputs from the keyboard.]

Window class instance defines a Window (object). The Window has basic coordinates x and y , and z -parameter. The z specifies whether Window is over and below other windows. The Window has specification for visibility (show or hide or no activate). The Window has specification for parent-child hierarchy. Windows procedures share the attributes, for example, Commandshow. Windows procedures are there to respond to requests and all notifications sent to the Windows.

WCE does not support Handle inheritance. [Windows uses Handle in many procedures (functions). The Handle provides reference to an interface, for example, for a Window, file or thread or port. An example is INSTANCE of a Window. It is an object, which is used as a Handle. An interface is an unimplemented procedure (function or method), the codes for which are defined in the class, which uses that interface. Handle is also used as a pointer, called *option pointer*. The option pointer is pointer which points to a pointer of one of the several sets of the codes, which run on selecting the option. Windows support (WCE does not) Handle inheritance, which means a Handle can be extended to create a new Handle, which inherits the variables, properties and procedures of parent handle and adds, overrides and overloads new variables, properties and procedures.]

10.1.3 Windows and Windows Management

There are many Windows on a screen. A screen top (desktop) is a Window. A command-tool-task bar is a Window. A button is a Window. The Windows are related to each other. There may be a hierarchical (parent child) relationship in the Windows. There may be a sibling relationship or owner-owned relationship.

There is a top-level main Window. The main Window does not have a parent. The main Window can have child Windows. When a parent is moved or deleted, the all child Windows shall also move or delete. Child Window is invisible except at the edges. CreateWindowEx or CreateWindow creates a Window and uses the same messages and procedures as the main. A 32-bit style parameter dwStyle when set as WS_Child the child Window is created. An 8-bit style parameter bMenu parameter is used in child Window and equals the ID of that Window. (Prefix dw means double word and b means byte as data-types.)

Examples of management functions for the Windows are FindWindow (to find a Window and get Handle for that), GetParent to find the parent and GetWindow to query and get the owner, children and siblings.

10.1.4 Memory Management

Windows CE 6.0 permits virtual memory (VM) limit of 2 GB (earlier 32 MB) for each process and upper 2 GB VM space as the kernel VM space. Extended VM support enables the creation of complex 3-D graphics on WCE devices and therefore animation and gaming applications.

WCE provides for system memory between 1 MB and 64 MB and OS needs minimum 512 kB of memory and 4 kB RAM. WCE also provides for managing the low memory conditions.

WCE considers the RAM in two sections: *program memory* (called system heap) and *object store*. The memory is allocated to the program from a pool of unused memory area called the heap. The application program that runs uses the heap and stacks. An application is allocated memory blocks (in place of the pages) from the heap and is in reserved virtual memory space region. A block in heap can also be freed later when not required. A heap can be a local heap of 188 kB or a separate heap in case of requirement of bigger number of memory blocks.

Object store (256 MB) is virtual RAM disk for permanent store, which is protected from turning off power. Individual file can use up to 32 MB in case of RAM as *object store*. PIM (personal information manager) data is also stored at the *store*. PIM includes data of the contacts, calendar and task-to-do. A contact includes name, address, e-mail ID, phone numbers of home, office and mobile. A handheld PocketPC has a backup battery, which provides power to *object store* data and files. WCE at power-on searches the previously loaded *object store* at RAM and uses that object if available. The *object store* stores files, registry and WCE databases (Sections 10.1.5 and 10.1.6).

WCE saves in the ROM execute-in-place files. Execute-in-place file is a file in ROM for execution that cannot be opened and read by standard file functions open and read (Section 8.6.2).

WCE supports virtual and page memory. Virtual memory may be at the flash or disk. The application program uses the physical addresses at RAM. A virtual memory management system maps the virtual addresses of pages with the physical addresses of pages after the pages of the program has been loaded at RAM. A page is a fixed-sized memory unit, which is loaded from disk or flash to the RAM. WCE uses page size of 1 kB or 4 kB. It depends on the system processor. Three types of virtual pages are supported in WCE. Committed page is a page reserved for application and directly maps to the RAM address. *Reserved page* at virtual address cannot be used in the application. A free page can be used and is allocated during the run.

Static Allocations WCE allocates two allocations, one for read only and other for read/write data.

Stacks Stack stores the temporary variables and processor registers for the application and OS functions. WCE provides for separate stack for each thread (Section 7.2). WCE provides for 58 kB maximum stack size and 6 kB of stack for guarding the stack for underflow or overflow. An application can also specify the thread stack size.

10.1.5 Files and Registry

Files are created by a CreateFile function. It has the following arguments:

1. long pointer for character string,
2. 32-bit desired access parameter,
3. 32-bit shared mode specification,
4. long pointer for security attributes,
5. 32-bit to specify creation and distribution,
6. 32-bit specify flags and attributes and
7. Handle for template file.

The arguments are assigned as follows:

1. character string used for file name,
2. WCE file access parameter GENERIC_READ or GENERIC_WRITE or both,
3. WCE create file can set 32-bit shared mode specification as 0 or FILE_SHARE_READ, FILE_SHARE_WRITE.
4. WCE create file must use long pointer for security attributes as NULL.
5. WCE file creation and distribution specified by 32-bits CREATE_NEW, OPEN_EXISTING, CREATE_ALWAYS (new file after truncating existing), OPEN_ALWAYS (create new file if not existing else open) or TRUNCATE_EXISTING.
6. 32-bit specify WCE flags and attributes FILE_FLAG_WRITE_THROUGH, FILE_FLAG_RANDOM_ACCESS, FILE_ATTRIBUTE_NORMAL, FILE_ATTRIBUTE_READONLY, FILE_ATTRIBUTE_ARCHIVE, FILE_ATTRIBUTE_SYSTEM, FILE_ATTRIBUTE_HIDDEN, FILE_ATTRIBUTE_NORMAL and
7. Handle for template file is NULL as WCE does not support file template.

WCE files differ and have similarities in following respects for other Windows OS.

- (1) Uses most of the Win32 file APIs.
- (2) Uses standard file IO procedures CreateFile, OpenFile, ReadFile, WriteFile, SetEndOfFile and CloseFile (Section 8.6.2). These functions create, open, read, write, truncate and close the files (not for execute-in-place files in ROM).
- (3) WCE files use the same attribute flags as in Windows. Examples are flags for read only, compressed, archive and system hidden.
- (4) WCE used up to 256 storage devices or partitions on the storage devices. A installable file system driver can be installed for flash and other file systems.
- (5) Uses object store as a default RAM-based file system.
- (6) Current directory concept missing. A file name and complete path specification (maximum 260 bytes of MAX_PATH length) is required in WCE. There is no mention like C or D drive when using the files. A file has three-character extension format after the dot sign. The extension defines the file type. For example, .txt for text file.
- (7) Memory-mapped files and objects are supported for reading the files as byte streams.
- (8) Uses compact flash and RAM with backup battery.

WCE registry uses standard registry API. Registry API is an API for system database. It uses standard file registry functions. The functions are as follows: RegCreateKeyEx, RegOpenKeyEx, RegSetValueEx, RegQueryValueEx, RegDeleteKeyEx, RegDeleteValueEx and RegCloseKey.

Registry system has keys and their values as in a hash table. Keys can contain the keys. There are multiple-level keys. Permitted data types for registry are 32-bit numbers, string or free (for binary data).

10.1.6 Windows CE Databases

Table 10.2 gives the procedures (functions) and properties of WCE databases.

Table 10.2 Windows CE Functions in WCE Databases

S.No.	Feature	Description
1	Database format	Series of un-lockable records with saving of the property(ies) and data together in a record. No record contains another record within it. Each record has four-level indices for sorting.
2	Maximum number of records and record size	$2^{16} - 1$ and 2^{17}
3	Database record properties data types	Double 64-bit signed, boolean, collection of bytes, 0-temined unicode string, 16-bit signed, 16-bit unsigned, 32-bit signed, 32-bit unsigned, time and date structure
4	CeMountDBVol Windows CE function	To mount an external flash or media on a database volume, which stores the database files (not object-store files)
5	CeCreateDatabaseEx Windows CE function	To create new database
6	CeOpenDatabaseEx Windows CE function	To open created database
7	CeSeekDatabaseEx Windows CE function	To set pointer to database record
8	CeSetDatabaseInfoEx Windows CE function	To set the sort order of opened database
9	CeReadrecordPropsEx Windows CE function	To read a record of given property
10	CeWriteRecordProps function	To write new property of record
11	CeDeleteDatabaseEx function	To delete records, properties and complete database
12	CloseHandle ()	To close the Handle
13	CeUnmountDBVol Windows CE function	To un-mount an external flash or media form a database volume, which was mounted earlier
14	Windows CE notification	Notify to a process that a thread has modified the database.

10.1.7 Processes, Threads and IPCs

WCE has Win32 executable files, called modules in two forms, one .exe files and the other the DLLs (dynamic-linked libraries). The .exe files are compiled files and are first loaded in system memory for execution. The DLLs are loaded at run time on request from the .exe file or another DLL. The loading is by using a list of DLLs and the functions list within the DLLs at an import table in the .exe file. The DLL also has import table for other DLLs and their functions.

WCE is a multitasking multithreaded (Sections 7.1, 7.2 and 7.3) OS. In a multitasking OS, a process consists of the number of processes. The process is an instance of an application. There can be multiple copies of the same processes. In a multitasking multithreaded OS, a process can create multiple threads which execute concurrently. There is at least one thread which is created per process. There is communication between the threads of the same processes or from one process thread to another process thread. Each thread has separate execution under control of kernel is thread. Each thread has separate context (for CPU register, program counter and stack pointer) and stack for saving context when thread blocks and for retrieving on thread resume.

At least four WCE processes load on start up. These are FileSys.exe (for file system functions), User.exe (for GUI functions), Device.exe (for loading and maintaing device drivers) and NK.exe (for kernel functions).

Table 10.3 gives the procedures and properties of processes and threads.

Table 10.3 Properties and Windows CE Functions for Threads and Processes

S.No.	Feature	Description
1	Thread properties	Threads of the same process share the memory space and manage access to shared resources. The Handles are used for synchronization objects (interprocess communication objects), file and memory objects. A primary thread can create secondary threads.
2	Thread priorities	Each thread assigned priorities one among eight levels: system-level threads and device drivers (ISTs) use upper 248 levels of priorities. Total number of priority levels is 256 and priority 0 value is highest and 255 is lowest. Higher priority thread pre-empts lower priority thread. Priority inversion (Section 7.8.5) is taken care of during execution.
3	Maximum number of processes and memory size	Windows CE 6.0 supports number of processes 2^{16} (earlier 32) with each process virtual memory limit of 2 GB (earlier 32 MB) and up to lower 2 GB (earlier 32 MB). Also there is upper 2GB (earlier 32 MB), which is the kernel VM space.
4	CreateProcess WCE function	To create a new process with four parameters required in the arguments (long pointers for the ApplicationName, Commandline (a unicode string), and ProcessInformation ² and 32-bit CreationFlags ³ (to specify initial state on loading).
5	TerminateProcess Windows CE function	To terminate a process. Two arguments are Handle for process and 32-bit exit code for the process. The exit code for the process is obtained from GetExitCodeProcess function.
6	OpenProcess Windows CE function	To open a process using processId as argument. ProcessId is obtained from GetWindowThreadProcessId .
7	CreateThread WCE function	To create new process with following parameters required in the arguments: 32-bit StackSize, long pointer for THREAD_START_ROUTINE (an address of routine for starting thread execution), long pointer Parameter for application-specified thread parameter(s), 32-bit CreationFlags ⁴ (to specify initial state on loading) and 32-bit ThreadId (thread ID).
8	ExitThread Windows CE function	To terminate a thread using one argument, which is 32-bit exit code for the thread. The exit code for the thread is obtained from GetExitCodeThread function.

(Contd)

S.No.	Feature	Description
9	SetThread-Priority	To set thread priority among eight levels. Handle for thread and <i>priority</i> value to be set are the arguments. Value = 250 for thread set to normal priority level.
10	CeSetThread-Priority	To set thread priority value among 256 values between 0 and 255. Handle for thread and <i>priority</i> value are the arguments.
11	GetThread-Priority	To get thread priority level. Handle for thread is the argument.
12	CeGetThread-Priority	To get thread priority value among 256 levels between 0 and 255. Handle for thread is the argument.
13	CeSetThread-Quantum	To set thread time slice value. Handle for thread and 32-bit <i>time slice</i> in milliseconds are the arguments.
14	CeGetThread-Quantum	To get thread time slice value. Handle for thread is the argument.
15	Sleep	To delay the thread execution for a period. 32-bit time slice in milliseconds is the argument.
16	SuspendThread	To suspend a thread. Handle for thread is the argument.
17	ResumeThread	To resume a suspended thread. Handle for thread is the argument.

Eight levels are idle (= normal - 4), above idle (= normal - 3), lowest (= normal - 2), below normal (= normal - 1), normal, above normal (= normal + 1), highest (= normal + 2) and time critical (= normal + 3).

ProcessInformation consists of a Handle object for process, a Handle object for thread, 32-bit processId and 32-bit threadId. CreationFlags = 0 for standard process. = CREATE_SUSPEND for create and suspend. = CREATE_NEWCONSOLE for creating a new console. = DEBUG_PROCESS for process passing debug information to calling process and = DEBUG_ONLY_THIS_PROCESS for process passing debug information only from this process and not from CHILD PROCESSES.

CreationFlags = 0 for standard thread. = CREATE_SUSPEND for create and suspend (Must use ResumeThread if created thread is suspended) and = STACK_SIZE_PARAM_IS_A_RESERVATION for creating thread with reserved stack size.

Exceptions, Notifications and IPC Objects Synchronization WCE provides for exception handling signals and notification signals (Section 7.10). Notification examples are notifications from timer, serial device detect, power up, system event. Notification can generate a dialog and a sound or run a notification responding function.

WCE also provides Handles for the IPC objects. An IPC object in a multitasking or multithreading system is used to generate a synchronization object and the object gives the information about certain sets of computations finishing one process or thread and to let the other process or thread waiting for that object to get information about finishing the computations that take note of the information. An IPC object is released (sent) which means that a process (thread or scheduler, task or ISR) generates some information by or value *r* generates an output so that it lets another process waiting for that object in order to take note or use the object. A kernel provisions for the functions for creating, releasing and waiting the IPC objects (Section 7.9).

IPC objects in a multitasking or multithreading system are events (Section 8.4), semaphores including mutex semaphores (Section 7.11) and message queues (Section 7.12). WCE provides for the events, semaphores including mutex semaphores and message queues for threads synchronization. Table 10.4 gives these for exceptions, notifications, events, semaphores, mutex and message queues.

0.1.8 Inputs from Keys, Touch Screen or Mouse

Keyboard is used to enter many characters, commands or large text. Physical keyboard is inconvenient in a handheld device. There is a soft keyboard. It controls and simulates the virtual keyboard on touch screen. An

application can get the input either from physical keyboard or from soft keyboard. A function SetFocus is used to specify the focused Window so that the input directs to that Window. Windows sends a series of messages for the Window in focus. Every key or action has an assigned value. For example, a virtual key value is VK_LBUTTON which passes a value 01 on a stylus tap. A virtual key value is VK_RETURN, which passes a value 0D when the Enter key is pressed.

Table 10.4 Windows CE Functions for Exceptions, Notifications, Events, Semaphores, Mutex and Message-Queues

S.No.	Feature	Description
1	Signalling (exception) functions	RaiseException using the 32-bit exception code, exception flags, number of arguments and 32-bit constant for array of long pointer for the arguments. Each argument passes the data to the exceptional responding routine (like ISR).
2	Signalling (notification) functions	<ol style="list-style-type: none"> CeSetUserNotificationEx using the Handle for notification, CE_NOTIFICATION_TRIGGER object¹ and CE_USER_NOTIFICATION object (object pointer defines action flags, dialog title, dialog text, sound and other details). CeGetUserNotificationEx using the Handle for notification and long pointer for CE_USER_NOTIFICATION object. CeClearUserNotificationEx using the Handle for notification to acknowledge a notification by notification responding function.
3	Critical section functions	InitializeCriticalSection (to initialize a critical section), EnterCriticalSection (to enter a critical section), LeaveCriticalSection (to exit a critical section), TryEnterCriticalSection (to try to enter a critical section) and DeleteCriticalSection (to delete a critical section).
4	Semaphore functions (for threads of a process)	CreateSemaphore (to create the semaphore), ReleaseSemaphore (release semaphore to let waiting thread code unblock), CreateMutex (to create the mutex), ReleaseMutex (release mutex to let waiting thread code unblock).
5	Message queue functions	CreateMsgQueue (to create the message queue), OpenMsgQueue (to open a message queue), ReadMsgQueue (to read from the queue), GetMsgQueueInfo (to query the queue), WriteMsgQueue (to write into the queue), CloseMsgQueue (to close an open message queue).
6	Event functions (for threads of a process)	CreateEvent (to create the event), SetEvent (event set to signal occurrence of the event and do not auto-reset till waiting thread unblocks) (event auto-resets on unblocking of thread), ResetEvent (to force reset of the event and unblock the thread waiting for it), PulseEvent (to set the event and then reset the event by unblocking all waiting threads for that event), SetEventData using event-handle and 32-bit data in the arguments, GetEventData using event-handle to get the event data.
7	Wait-Single object functions	WaitForSingleObject using object Handle and 32-bit waiting time value in milliseconds in the arguments.

(Contd)

S.No.	Feature	Description
8	Wait for multiple objects	WaitForMultipleObjects using count number for objects (events or mutexes), pointer to array of object Handles, boolean WaitAll (if true then wait for all, in WCE must be set to false) and 32-bit waiting time value in milliseconds in the arguments. Each object handle is a long pointer. Waiting time value = INFINITE disables the timeout specification For wait for the multiple objects.
9	Wait for message objects	MsgWaitForMultipleObjectsEx using count number for message objects, long pointer to array of object Handles, boolean WaitAll (if true then wait for all, in WCE must be set to false), 32-bit waiting time value in milliseconds in the arguments and 32-bit flags for WakeMask. ²

²CE_NOTIFICATION_TRIGGER object pointer defines type and notification details of notification type, notification size, notification event, notification application, notification arguments, notification start time and notification end time. WakeMaskFlags = QS_ALLINPUT for any message received, QS_TIMER for a WM_TIMER (Windows Manager timer) message, QS_PAINT for a WM_PAINT Windows manager paint, QS_SENDMESSAGE (a sent message outside the list received), QS_POSTMESSAGE (a posted message outside the list received), QS_MOUSE a mouse move or click or stylus tap received, QS_MOUSEMOVE a mouse move or stylus move received, QS_MOUSEBUTTON a mouse click or stylus tap received.

A keyboard function example is SHORT GetKeyState (int iVirtKey) for querying a keyboard key. An application can simulate key-event.

Inputs from Touch Screen or Mice Touch screen for input is equivalent to a single button mice input. Further, the mice has a cursor. When a mice is pressed, the window is sent the message WM_LBUTTONDOWN on left button down and release of WM_LBUTTONUP on left button up event.

WM_MOUSEMOVE message is sent when the stylus is moved within the same window. When the stylus is dragged outside the in-focus window, the WM_MOUSEMOVE messages stop. If SetCapture procedure is called then, WM_MOUSEMOVE messages continue. ReleaseCapture stops sending the messages of WM_MOUSEMOVE.

GetMouseMovePoints sends the messages for each point traced by stylus on the screen from a start to end. GetMouseMovePoints integrated with handwriting recognizer application can be used handwriting on the PocketPC to write the text or commands or messages.

WM_LBUTTONDOWNBLCLK message is sent on the double tap of the stylus. For each message the parameters lParam = two 16-bit screen tap horizontal and vertical position values x and y, and wParam = 16 bits for the flags corresponding to which key shift or control held down or not.

Right button click of mice is simulated using stylus when ALT key is held down while tapping.

Windows Controls Each Window uses a number of classes, called *Controls*. A control has a number of user-interface elements. The user-interface examples are *button*, *radio* and *checkbox*. The user-interface elements are predefined for a Control and there exists a Windows Control library. Predefinition and library help in each application window has same feel and look.

A Control is also a Window and is created by CreateWindowEx or CreateWindow. A control may be static control. It displays a text (as per defined alignment) or icon or bitmap. A control is scroll bar control.

Most powerful Control for user interface is *Button*. Button appearance can be set. An owner window can also draw the owned button.

1. Simple button is a push-button. When the stylus taps the button, it generates a WM_ message with a 16-bit parameter wParam for the flag BN_CLICKED. BN_CLICKED that button is clicked.
2. Checkbox button is a square box with blank or filled circle or a label using which the user choice by tap stylus at that point and which toggles between the blank and the filled. It toggles between two states.
3. Radio button is a button to allow the user to select among the interrelated choices and selects the other may unselect. Application checks or unchecks a radio button.
4. Group box button. It is an empty box with a text label. The text in the box gives the interface programming.

A Control is list box control. It is used for selecting among the list of items displayed by text. WCE supports a constant string data style in list box control. The style is called LBS_EX_CONSTSTRINGDATA. Only pointer to the string saves at the Window and not the string. The application is supposed to manage that string.

A Control is Edit control. It is used for keying in the text and editing it. The keyed text is in upper cases when ES_UPPERCASE style is set. When ES_LOWERCASE is the style, the edit text appears as lower cases. The keyed text is visible as *** when ES_PASSWORD style is set.

A Control is combo-box control. One can use two or more controls in the combo box. A combo box in WCE is drop-down or drop-down list. Drop-down is an edit-text field control with a button on the right side. When this button is clicked a list box for selection appears. Drop-down list is a list of texts each with a button on the right side. The stylus taps at any one of it to choose.

Windows Menus WCE menus are at the menu bar or command bar control. The CreateMenu, AppendMenu, InsertMenu are the procedures in WCE to create, append or insert a menu item. CreatePopupMenu is a procedure to nest the menus. Window generates WM_COMMAND message and the ID parameter of the menu item is sent.

10.1.9 Communications and Networking

WCE serial port is a stream device driver, which is also opened by CreateFile (Section 10.1.4). WCE has the functions for clearing errors, querying status, timeout values setting and getting, querying the serial driver and controlling the communication. Table 10.5 gives the Windows CE serial communication functions.

WNet API is an API for networking support to Windows. It has a feature of accessing network resources that does not depend on platform and implementation of network functions. WCE supports a subset of WNet. Table 10.6 gives the WNet API subset network connection functions.

10.1.10 Device-to-Device Socket and Communication Functions

Devices, such as mobile phone or PocketPC establish synchronization with the neighbouring devices and computers, and form a personal area network (PAN). Examples of protocols used in PAN are Bluetooth and IrDA (Infrared Data Association).

An API is Winsock API for sockets programming support to Windows. The TCP/IP, Bluetooth and IrDA network sockets are programmed using Winsock. Winsock supports streaming sockets and datagram (Section 3.11.3) connections. (Streaming sockets and datagram difference is that there is connection between two APIs at different devices, and between two specific addresses at two APIs at different devices, respectively.) Winsock has a feature of accessing PAN resources through the sockets that does not depend on platform and implementation of socket functions (Section 7.15). WCE supports a subset of Winsock 1.1 and 2.0 API. Table 10.7 gives the Winsock API subset in WCE for device-to-device socket communication functions.

Table 10.5 Windows CE Serial Communication Functions

S.No.	Feature	Description
1	CreateFile WCE function	Creates the port for communication. Returns a Handle for serial COM1 port. The arguments used are TEXT ("COM1"), GENERIC_READ GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, and NULL.
2	ReadFile	Reads from the port for communication. Returns an integer. The arguments used are Handle returned on creation, pointer to character, pointer to 8-bit number of bytes read, NULL.
3	WriteFile	Writes to the port for communication. Returns an integer. The arguments used are Handle returned on creation, pointer to character, pointer to 8-bit number of bytes read, NULL.
4	TransmitCommChar	Send character into queue for port transmission. (Control characters can be inserted into the stream). Returns a <i>boolean</i> for successful or unsuccessful transmission. The arguments used are Handle returned on creation and character for transmission.
5	Set CommMask	To set communication mask. The arguments used are Handle returned on creation and 32-bit for event mask to specify clear to send, break, data set ready, error, receive line signal detect, character received, a receive-event's flag received, transmit buffer empty.
6	Get CommMask	To get communication mask. The arguments used are Handle returned on creation and long pointer for 32-bit event mask.
7	WaitCommEvent	To wait for event. Handle for file, long pointer for 32-bit event mask and NULL (for long pointer for overlap) are the arguments.
8	SetCommState	To set communication state. Handle for file and long pointer to device control block (DCB) structure are the arguments. DCB defines 32 bits for DCB length, baud rate, binary flag, parity flag and 24 other flags.

Table 10.5 WNet API Network Connection Functions

S.No.	Feature	Description
1	WNetAdd-Connection	Maps the network (remote) resource. Returns a 32-bit code for no error or error. The arguments used are one Window handle, three long pointers for <i>network resource</i> and string for password and user names and one 32-bit value for the flags. The <i>network resource</i> is a structure, which contains long pointers for remote name and local name.
2	WNetConnection-Dialog	To dialog. The argument used is a long pointer for connection dialog structure.
3	WNetCancel-Connection	Disconnects the network (remote) connection added earlier. Returns a 32-bit code for no error or error. The arguments used are one long pointer for name (local or remote), 32-bit value for the flags, boolean to specify forced disconnection when files or devices are open and not closed.

(Contd)

S.No.	Feature	Description
4	WNetDis-connectDialog	To dialog on disconnection. The arguments used are a Window handle and 32 bits for resource type. Resource type may be printer or disk or any other that is available. There is another overloading WNetDisconnectDialog, which has one argument, a long pointer for disconnection dialog structure.
5	WNetGet-Connection	Queries the network (remote) resource connection. Returns a 32-bit code for no error or error. The arguments used are long pointers for strings for name (local or remote) and user name and long pointer for 32-bit value, which specifies the length of remote buffer characters.
6	WNetGetUser	Queries the user name. The arguments used are long pointers for strings for local and remote names and long pointer for 32-bit value, which specifies the length of remote buffer.
7	WNetGetUniversalName	Queries the name as per universal naming convention. The arguments used are long pointer for string for local path, 32-bit info-level, long pointers for buffer address and 32-bit buffer size.
8	SetCommState	To set communication state. Handle for file and long pointer to device control block (DCB) structure are the arguments. DCB defines 32 bits for DCB length, baud rate, binary flag, parity flag and 24 other flags.

Table 10.7 Winsock API Subset in WCE for Device-to-Device Socket Communication Functions

S.No.	Feature	Description
1	Socket function	To create new Socket, is like a Handle. The function has three parameters required as the arguments: three integers, one for addressed family specification (e.g., AF_BT, AF_IRDA or AF_INET for Bluetooth or IrDA or TCP/IP, respectively), second for addressed socket type specification (e.g., SOCK_STREAM or SOCK_DGRAM for stream or datagram socket, respectively) and third for protocol (e.g., BTHPROTO_RFCOMM for Bluetooth RF communication).
2	Bind function	To find the desired server. Three arguments are SOCKET (of server), constant structure for addressed socket information (e.g., SOCKADDR_BT, SOCKADDR_IRDA or SOCKADDR_INET for Bluetooth or IrDA or TCP/IP, respectively) and integer for name length.
3	Accept function	To accept client connection at the server in listen mode. Three arguments are SOCKET (of server) already in listen mode, structure for addressed buffer information (for SOCKADDR_BT, SOCKADDR_IRDA or SOCKADDR_IN for Bluetooth or IrDA or TCP/IP, respectively) and integer for buffer length.

(Contd)

S.No.	Feature	Description
4	Connect function	To connect a newly created client socket to server (client does not call bind and accept function). Three arguments are connecting (client) SOCKET, constant structure for addressed socket information (e.g., SOCKADDR_BTH, SOCKADDR_IRDA or SOCKADDR_INET for Bluetooth or IrDA or TCP/IP, respectively) and integer for name length.
5	Listen function	To connect to server for data after binding. Two arguments are SOCKET and integer for queue size (= SOMAXCONN for maximum size) for pending connection.
6	Send function	To send from a socket. Four arguments are SOCKET (of sender), constant char for addressed buffer information (for SOCKADDR_BTH, SOCKADDR_IRDA or SOCKADDR_INET for Bluetooth or IrDA or TCP/IP, respectively) and two integers one for length and other for flags.
7	Recv function	To receive from a SOCKET. Four arguments are SOCKET (of receiver), constant char for addressed buffer information (for SOCKADDR_BTH, SOCKADDR_IRDA or SOCKADDR_INET for Bluetooth or IrDA or TCP/IP, respectively) and two integers one for length and other for flags.
8	Shutdown	To shut the send and receive functions on a close of the socket connection. Two arguments are SOCKET and integer for closing how (= SD_BOTH or SD_SEND or SD_RECEIVE for shutdown of both send or receive or send function only or receive function only, respectively)
9	Close socket	To close the socket connection. The argument is SOCKET to be closed.

10.1.11 Win32 API Programming

The development of Windows for the GUIs is often the most important part of application development in a computer or embedded system or handheld system which has a screen or touch screen for interaction with a user. GUIs facilitate interaction and inputs from user after graphic screen displays of menus, buttons, dialog boxes, text fields, labels, check box and radio buttons and others. Win32 API programming is thus a very important part of any application development. Win32 has large number of APIs in a PC. However, only a subset is required for handheld devices and small screen size systems. A subset of Win32 APIs is provided in WCE.

When an application is developed, a Windows displays the messages in the central region, title, command, tool and status bars. The Windows also displays commands (buttons) so that a stylus tap (or mouse click) sends the selected command using menu and buttons. The Windows also show the icons for maximizing, minimizing and closing at right-hand side top corner. Windows also show icon for Help (to help the user) and a ? sign icon (to show more buttons on a tap there). WCE has single-line controls for command, tool and status bars. A stylus tap or mouse click sends the menu choice to the application. WCE has new format for the Windows Controls (command, menu, toolbar bars) and new Controls (data, time, calendar) and organizer (e.g., task-to-do).

The following example shows a simple application of Win32 API. The example shows how simple it is to create the screen windows and show the commands (buttons) for further action in an application.

Example 10.1

```
int WINAPI WinMain (HINSTANCE hPresentinstance; HINSTANCE hPreviousinstance, LPWSTR lpCommandline, int iCommandshow) {MessageBox (NULL, TEXT ("Welcome"), TEXT ("WelcMsg"), MB_DEFBUTTON1, MB_DEFBUTTON2, MB_ICONQUESTION); return 0; } /* After third argument the last argument(s) is one or more among the series of flags which can be used for showing the buttons or icons in MessageBox Windows bar. The buttons and icons must be those as provided for in the procedure MessageBox. */
```

1. The Presentinstance is a parameter to identify the present instance. Previnstance is a parameter to identify the previousinstance (WCE always assumes it to be zero).
2. Commandline is a unicode string (it specifies the functions of the program).
3. Commandshow is an integer to specify state of the program, which defines a configuration of main window. The state parameter is passed from the parent application to a new application. The state configuration in a personal computer can be the one which shows minimized, maximized or normal icons. WCE allows only three states and configuration of WCE Windows is as per variables show without activate (SW_SHOWNOACTIVATE), show hidden (SW_HIDE) and show normal (SW_SHOW). Default value of Commandshow is used as per the value for the main Window show command.
4. MessageBox creates a window over the main window. The window shows messages in the box until window is closed. It shows: (i) no other Windows because first argument is NULL; (ii) text message Welcome in the unicode message window (at center) and text Unicode message caption (title) WelcMsg at left corner in the command-cum-tool-status bar; (iii) buttons as per definitions MB_DEFBUTTON1, MB_DEFBUTTON2 in the middle of command bar and icon of ?; (iv) at the end of bar, a sign X icon is created at the right corner in the bar. The X enables the closing of the window by the user on tapping on the touch screen or mouse click. [MessageBox is used here in place of printf otherwise a driver console.dll needs to be added to enable printing on console (screen).]

This example uses Handle, INSTANCE is a Handle object. In the present case, a handle is a reference to an interface, which handles a Window instance.

This example uses prefixes before the objects and variables as follows. Prefix H before object INSTANCE indicates that it is a Handle object. Prefix LP before WSTR indicates that it is a long pointer. Prefix W before STR indicates that the string is a 16-bit unsigned word. This method or prefixing helps in easier understanding of the objects and variables by a programmer.

10.1.12 Creating Windows

An application can create its own Windows instead of creating the Windows using MessageBox as in the previous example or a similar function in the Win32 subset of WCE. There are several Windows procedures that can create its own Windows. Following are the examples.

1. CreateWindowEx, which creates main window.
2. MainWndProc, which creates application window.
3. For example, WM_Paint to draw the window background and put text within it at the specified position after first creating a client rectangle.

Drawing on Screen WCE does not support full Win32 graphics API and different mapping modes in Windows. WCE does not support coordinate transformations. A text is written using DrawText procedure. WCE always sets device context in MM_TEXT mapping mode.

Windows application does not write directly to the screen. It requests a Handle. The handle draws and displays device context. Device context specifies the application Windows. Windows sends the pixels to screen using the device context. A device context is a tool, which Windows use to manage the access to the display and printer. Two attributes of device context are colours for background and foreground. Text alignment attributes of device context are left, right, top, centre, bottom. no update and update of current point of device context and baseline alignment. Font of the displayed text from the device context can be specified. Font can also be created for an application as alternative to WCE default fonts.

A bitmap is a graphical object. Bitmaps can be drawn. The bitmaps are used to create, retrieve images, manipulate and draw at the device context. WCE supports format of bitmap in four colours. WCE permits 1, 2, 4, 8, 16 and 24 values and provides for compaction.

WCE provides for drawing lines, rectangle, circle, ellipse, round rectangle and polygon shapes and has a pen tool. WCE provides for fill functions for the draw, for example, gradient fill (shade changing on moving vertically up to down or horizontally from left to right) and hatched filling.

10.2 OSEK

RTOSes described in Sections 9.1 to 9.3 do not suffice for automotive systems, which require other necessary features. Embedded software in the automotive system needs special features in its OS over and above the MUCOS or VxWorks features and MS DOS and UNIX. Special OS features needed are as follows.

1. *Language* can be application-specific, need not be just C or C++ and *data types* should also be application-specific and not RTOS-specific. In VxWorks, for example, STATUS is RTOS specific. This is not permitted, as it could be the source of a bug and thus unreliable.
2. *OS*, every method, class and run-time library should be scalable. This optimizes the memory needs.
3. *Tasks* can be classified into four types. This provides a clear-cut distinction to a programmer: which class to use for what modules in the system.
 - (a) Basic with one task of each priority and single activation. It is called BCC 1 (Basic Conformance Class 1).
 - (b) Extended with one task of each priority and single activation. It is called ECC 1 (Extended Conformance Class 1). Extended task means, for example, a task created by FirstTask in Example 9.8.
 - (c) Basic with multiple tasks of each priority and multiple times activation during run. It is called BCC 2.
 - (d) Extended with multiple tasks of each priority and multiple times activation during run. It is called ECC 2.
4. OS can schedule ISRs and tasks in distinct ways. (VxWorks scheduler also does so.)
5. Interrupt system disables at the beginning of the service routine and enables on return. This lets the task run in real-time environment.
6. Task can be scheduled in real-time.
7. Task can consist of three types of objects, *events* (semaphore), *resources* (statements and functions) and *devices*. There are port devices also. An exemplary device is *alarm*. It displays the pictograms, messages and flashing messages. It sounds buzzer and beeps.
8. Timer, task or semaphore objects creation and deletion cannot be allowed. A run-time bug may lead to uncalled deletion of a timer or semaphore. That is the potential source of a problem and thus unreliable.
9. IPC, message queue posting by a task, is not allowed as a waiting task may wait indefinitely for its entire message needs. RTOS queue types, waiting infinitely or for a time out for a message can be a potential source of trouble and thus unreliable. Similar risks may arise with semaphore as a resource key or counter. These are therefore not used.

10. Before entering a critical section and on executing a service routine, all interrupts must disable and enable on return only (Refer to Section 7.8).

There is incompatibility of control units made by different automobile manufactures due to different interfaces and protocols. Software in the automotive electronics must also be standard.

A structured and modular software implementation based on standardized interfaces and protocols as proposed by OSEK/VDX is a necessity. This gives the portability and extendibility, and thus the reusability of existing software. Presently, the important software standards and guidance are AMI-C (Automotive Multimedia Interface Collaboration) [<http://www.ami-c.org>], MISRA-C (Motor Industry Reliability Association standard for C language software guidelines for automotive systems) [www.misra.org.uk] and OSEK/VDX for RTOS, communication and network management. (Refer to website <http://www.osek-idx.org> and also to a book *Programming in the OSEK/VDX Environment* by Joseph Lemieux from CMP Books, Oct. 2001.)

OSEK is an acronym for Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen (eng., "Open Systems and their interfaces for the Electronics in Motor vehicles"). A German automotive company consortium (BMW, Robert Bosch GmbH, DaimlerChrysler, Opel, Siemens and Volkswagen Group) and the University of Karlsruhe founded OSEK in 1993. Later VDX (Vehicle Distributed eXecutive) from Renault and PSA Peugeot Citroën joined the consortium.

OSEK/VDX is a body for defining and specifying three standards for an embedded OS.

1. One is for real-time execution of ECUs (electronic control units) software and base for the other OSEK/VDX modules using *MODISTARC* (methods and tools for the validation of OSEK/VDX-based distributed architectures and for conformance testing of the OSEK/VDX implementations).
2. Second is for *communications stack* for data exchange within and between control units.
3. Third is for *network management* protocol for automotive embedded systems configuration determination and monitoring.

OSEK/VDX has also produced other related specifications. There are two standard alternatives for OS and communication systems. One is for normal requirements (standard operating system/communication specifications). Second is for the special requirements. Special globally synchronized fault-tolerant architectures are covered using OSEK time specifications. Figure 10.2 shows OSEK basic features. OSEK provides for functional extendibility. One can integrate new application functions into a single control unit together with other APIs. OSEK provides for APIs porting. There is easy transfer of application functions from one hardware ECU platform to another with only minor modifications.

OSEK specifies that extendibility and portability should be independent of the source of APIs and co-existence of software from different sources must be possible. It shall be remarked that OSEK/VDX does not prescribe the implementation of OSEK/VDX modules, that is, different ECUs may have the same OSEK/VDX interfaces, but different implementations, depending on the hardware architecture and the performance required.

OSEK defines three standards.

1. OSEK-OS for OS, which has greater reliability. It is because, in an OS based upon OSEK, the previous ten points are taken care of.
2. OSEK-NM architecture for network management. As in OS, tasks are divided into four types, and the NM divides the architecture into two types. (i) Direct transfer and interchange of network messages; (ii) indirect transfer and interchange, both between the nodes.
3. OSEK-COM architecture for IPCs between the same CPU control unit tasks and between the different CPU control unit tasks. Between different unit tasks, the data link and physical layers exists. Different CPU physical layers connect by CAN bus architecture.

OSEK OS standard provisions for greater reliability compared with VxWorks or MUCOS. The MUCOS provides OSEK/VDX extension and provides the programmer the user a certified OSEK/VDX application programming interface. Reliability is introduced by the interface because the extension supports the OS conformance classes BCC1 and ECC1. The COM conformance classes CCCA and CCCB are provided for

internal communication. Extension does not permit creation and deletion of tasks during run. Extension defines each task of different priority and activates it only once in the codes. Extension does not use message queues and uses semaphores as event flag only with no task having run-time deletion or creation of these.

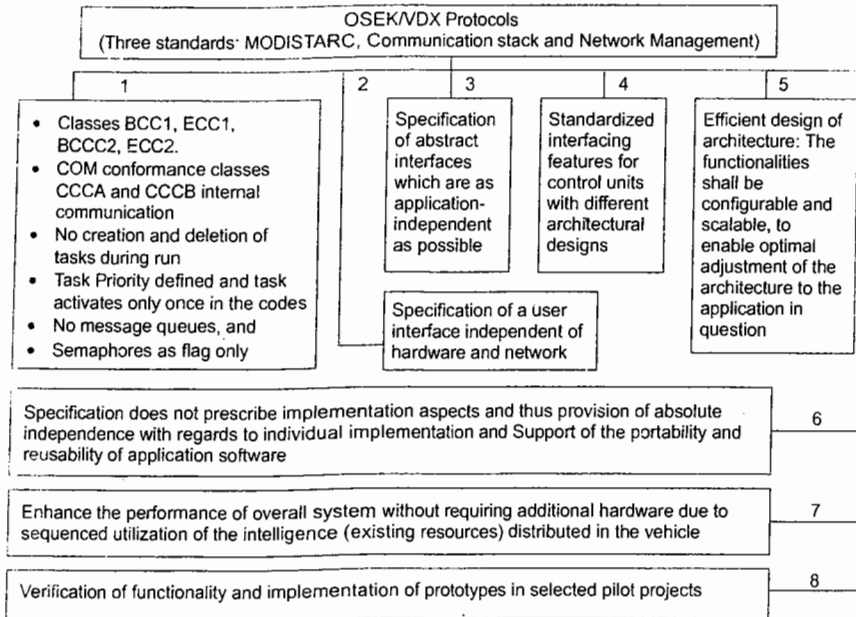


Fig. 10.2 OSEK basic features

10.3 LINUX 2.6.x AND RTLINUX

Linux is increasingly used in embedded systems and real-time enhancements of that have been introduced as Linux 2.6.x, the enhancements provide preemptive scheduling, high resolution timers and preemptive interrupt service (handler) threads. Linux latest version is Linux 2.6.24, released in January 2008. Reasons for the OS popularity are that it is a freeware, has device driver features, has expandability of the kernel codes at run time and has provision of registering and deregistering device-driver modules. The modules are scheduled like the processes. Section 10.3.1 and 10.3.2 describe the open-source real-time Linux and RTLinux (a real-time Linux) (both open source and profession).

10.3.1 Real Time Linux Functions

The OS named Linux is after Linus Torvalds, father of the Linux OS. Embedded Linux systems combine the Linux kernel with a small set of free software utilities. The glibc is often replaced as the C standard library by less resource-consuming alternatives such as dietlibc, uClibc or Newlib.

Embedded Linux application program (tasks) makes the system calls or message passing (Section 8.1.2) for the functions at a kernel for: 1. Process management, 2. Memory management (e.g., allocation, de-allocation, pointers, creating and deleting the tasks), 3. File system, 4. Shared memory, 5. Networking system functions,

6. Device control functions for any peripheral present into a system (computer), 7. Graphic editors for kernel configuration, 8. Soft real time scheduling. The following are three important classes of devices in embedded Linux.

Real-time Linux scheduler has all functions as $O(1)$ functions, which means that size of input to a function and function-run time are linearly correlated. The task scheduler supports spin lock (Section 7.11), and it is a preemptive scheduler. Linux supports character and block devices. A device type can be a char device for example, a parallel port, LCD matrix or serial port or keypad or mice. The character access is byte-by-byte and analogous to the access from and to a printer device. System call functions are also like an IO device open (), close (), write () and read () functions (Section 8.6.1).

The Linux kernel maps a device to a node of a file system. The network device drivers for the network devices also support the address resolution protocol (ARP) and reverse address resolution support (RARP). Recall Section 4.9.4 for the description of Linux device drivers. The important aspects of Linux are explained next.

Linux has the following features useful for embedded system design.

1. Linux is multi-user OS and supports user groups (2^{32}).
2. Linux has root directory (represented by / sign) and all files are subdirectories to it. For example, a device module file in the subdirectory is /dev. User files directory is /user.
3. Linux has large number of editor, file, directory, IO commands. Each command can consist of a number of words, first word is command and remaining words are arguments. A word may be a composite word for several functions and actions.
4. Linux has number of interfaces for user. For example, X-Windows for GUI and csh (for C shell). [Shell runs a parent process. Shell is an interface for a user to enable entering the commands for the OS. Shell reads, interprets, checks for errors and executes the command(s). For executing a command, a child process is created. (this action is called fork). After execution, there is return to the parent process, which issues a prompt sign \$. The shell waits for another command from user.]
5. Linux uses POSIX processes and threads (Sections 8.12). Linux header files Linux/types.h and Linux/shm.h, if included support the system programming for forking processes and shared memory functions in the kernel. For shared memory functions the POSIX map are used in Linux.
6. A process always creates as child process in the process that uses fork () function. The child creates as the copy of the parent with a new process ID (Section 7.1). There can be 2^{30} process IDs. The fork () returns a different process structure *pid* for the parent and *pid* number of child becomes = 0. The child process is made to perform different functions than the parent by overloading function of the parent process using execv () function. The function has two arguments, the *function_name* for the child function and *function_arguments* for the child functions. Each *process* has its own memory and cannot directly call another *process*. There is no lightweight process as in UNIX.
7. Linux has modules for the character, block, socket, network device drivers and others (Section 4.9.4) and Linux 2.6.x supports 4095 device-types. Each type of device can have 2^{30} device addresses. A *character device* (for example, parallel port, LCD matrix port, keypad or mice) receives or sends a sequential access byte stream. A *block device* (for example, file system or RAM disk) is a device that handles a block or part of a block of data. For example, 1 kB data handled at a time. Each block device receives or sends through a file system node (Section 8.6.2). A *net device* (for example, Telnet, FTP, TCP, IP or UDP protocol stacks networking device) is a device that handles network interface device (card or adapter) using a line protocols, for example, tty or PPP or SLIP. A *network interface* receives or sends packets using a protocol and sockets, and the kernel uses the modules related to packet transmission. An *input device* is a device that handles inputs from a device, for example, keyboard. An *input device driver* has functions for the standard input devices. The *media* device drivers have functions for the voice and video input devices. Examples are *video-frame grabber device*, *teletext device*, *radio device* (actually a streaming voice, music or speech device). A *video device* is a device that handles the frame buffer from the system to other systems as a char device does or UDP network packet-sending device

The video driver has functions for the standard video output devices. A sound device driver has functions for the standard audio devices. A sound device is a device that handles audio in standard format.

8. Linux supports a module initialization, handling of the errors, prevention of unauthorized port accesses, usage counts, root-level security and clean up. A module creates by compiling without main (). A module is an object file. For example, object module1.o creates from module1.c file by command \$ gcc -c {flags} module1.c. The Linux OS supports registering of the driver configurations and functions. Table 10.8 gives these functions. Linux scheduler not only schedules the tasks and device-driving ISRs but all other device modules also.

A Linux kernel can insert a module by registering and removing it by de-registering. (Registering means that it is scheduled later when its turn comes. Deregistering means the module will be ignored. It is similar to task spawning and deleting.) MUCOS (Section 10.2) registers and deregisters the tasks and ISR interrupts the tasks (for example, by a software interrupt instruction) and passes messages to the tasks. VxWorks (Section 9.3) kernel registers and deregisters the tasks (pre-empt mode scheduling) as well as nested ISRs. Linux kernel provides registering and deregistering of the device-driver modules as well. Thus, the Linux kernel permits the scheduling of device drivers and modules. Therefore, the different tasks or programs can send the bytes concurrently or sequentially to a device through its driver registered at the kernel. Further, the Linux kernel enforces the use of sequential accesses, an to specific memory addresses only, by the registering and de-registering mechanism.

Table 10.8 Registering and De-Registering and Related Functions of Linux Modules

Function	Action(s)
init	The <code>init_module()</code> is called before the module is inserted into the kernel. The function returns 0 if initialization succeeds and negative value if does not. The function registers a handler for something with the kernel. Alternatively it replaces one of the kernel functions by overloading.
insmod	Inserts module into the Linux kernel. The object file <code>module1.o</code> , inserts by command \$ <code>insmod module1.o</code> .
rmmod	A module file <code>module1.o</code> is deleted from the kernel by command \$ <code>rmmod module1</code> .
cleanup	A kernel-level void function, which performs the action on an <code>rmmod</code> call from the execution of the module. The <code>cleanup_module()</code> is called just before the module is removed. The <code>cleanup_module()</code> function negates whatever <code>init_module()</code> did and the module unloads safely.
register_capability	A kernel-level function for registering.
unregister_capability	A kernel-level function for deregistering.
register_symtab	A symbol table function support, which exists as an alternative to declaring functions and variables static.

9. Linux header file `Linux/time.h`, if included supports the timing function in the kernel for scheduling. Linux header files `Linux/delay.h` and `Linux/tqueue.h`, if included support delay functions and task queues functions. The task queues supported are the timers, disk, immediate and scheduler.
10. Linux supports signals (Section 7.10) on an event. Linux header file `Linux/signal.h` is included which supports the signalling function. Linux supports multithreading (Section 7.2), semaphores, mutex. Linux header files `Linux/pthread.h`, `Linux/ipc.h` `Linux/sem.h`, `Linux/msg.h` are included to support

the POSIX (Sections 8.12) thread, IPC, semaphore (including mutex) and message queue functions, respectively (Sections 7.10 to 7.16). Table 10.9 gives the functions for the signal, multithreading and semaphore and message queue IPCs.

11. Linux 2.6.x all tasks are assigned static priorities between -19 to +20 (highest to lowest). Time slices are varied as per priority.
12. Real-time Linux 2.6.24—a latest release supports a new set of group-scheduling functions, which improves CPU load.
13. Linux 2.6.24 provides high resolution timers as well as tick-less timers. (Tick-less mean does not result in clock-interrupts.)
14. Linux 2.6.24 supports functions to improve system performance by restricting the block device from taking larger CPU load.

Recently Wind River of VxWorks fame has optimized its Linux device software platform.

Table 10.9 Linux Functions for the Signal, Multithreading and Semaphore and Message Queue Interprocess Communication

S.No.	Feature	Description
1	Thread properties	Threads of same process share the memory space and manage access permissions. The IPCs are used for synchronization objects (interprocess communication objects) and threads.
2	Signal functions	<ol style="list-style-type: none"> 1. <code>struct sigaction signal_1; /* statement in C defines a structure signal_1. */</code> 2. <code>signal_1.sa_flags = 0; /* Sets flags = 0 */</code> 3. <code>signal_1.sa_handler = handlingfunction_1; /* Defines signal handler as a user defined function handlingfunction_1. */</code> 4. <code>sigemptyset (&signal_1.sa_mask); /* Defines signal as empty and sa_mask masks the execution of signal handler. */</code> 5. <code>sigaction (SIGINT, &signal_1, 0); /* Initiates signal function handlingfunction_1() using the structure signal_1. */</code>
3	Thread functions	<ol style="list-style-type: none"> 1. <code>struct pthread_t clientThd_1, clientThd_2, ServingThd; /* statement in C defines three structures for POSIX threads clientThd_1, clientThd_2, ServingThd. */</code> 2. <code>pthread_create (&clientThd_1, NULL, 0, thread_1, NULL) /* creates thread with structure clientThd_1 and calling function thread_1. NULL is the parameter passed. Similarly the threads clientThd_2, ServingThd and others can be created*/ The function returns an integer <i>createdstate</i> which is not 0 in case thread creation fails.</code> 3. <code>pthread_self () : /* Returns (gets) ID of the function pthread_self calling thread (self) */</code> 4. <code>pthread_join (&clientThd_1, &threadNew) /* joins thread with structure clientThd_1 and threadNew. */ The function returns an integer <i>createdstate</i> which is not 0 in case thread creation fails.</code> 5. <code>pthread_exit ("text") /* exits the thread under execution and text displays on console on exit. */</code> 6. <code>sleep (sleepime); /* The thread sleeps for a period = <i>sleepime</i> nanoseconds. Other thread gets the CPU in sleep interval*/</code> 7. <code>pthread_kill (); /* Sends 'kill' signal to the thread */</code>
4	Semaphore functions	<ol style="list-style-type: none"> 1. <code>struct sem_t sem1, sem2; /* statement in C defines three structures for semaphore structure (event control block), sem1 and sem2. */</code>

(Contd)

S.No.	Feature	Description
5	Mutex functions	<ol style="list-style-type: none"> 2. <code>sem_init ()</code>: The arguments are semaphore to be initiated and <i>options</i>. The function returns an integer <i>createdstate</i> which is not 0 in case semaphore initiation fails 3. <code>sem_wait (&sem1)</code>; /* wait for the semaphore sem1 */ 4. <code>sem_post (&sem1)</code>; /* post the semaphore sem1 */ 5. <code>sem_destroy (&sem1)</code>; /* destroy the semaphore sem1 */ <ol style="list-style-type: none"> 1. <code>struct pthread_mutex_t ms1, ms2</code>; /* statement in C defines three structures for semaphore structure (event control block). ms1 and ms2. */ 2. <code>pthread_mutex_init ()</code>: Initiates a mutex. The arguments are semaphore to be initiated (ms1 or ms2 or other) and parameter <i>options</i>. The function returns an integer <i>createdstate</i> which is not 0 in case mutex initiation fails 3. <code>pthread_mutex_lock ()</code>; The argument is mutex ms1 or ms2 or other*/ 4. <code>pthread_mutex_unlock ()</code>; The argument is mutex ms1 or ms2 or other*/ 5. <code>pthread_mutex_destroy ()</code>; The argument is mutex ms1 or ms2 or other*/
6	Message queue functions	<ol style="list-style-type: none"> 1. <code>pthread_mq_open ()</code>, <code>mq_close ()</code> and <code>mq_unlink ()</code> initialise, close and remove a named queue. <code>unlink ()</code> does not destroy the queue immediately but prevents the other tasks from using the queue. The queue will get destroyed only if the last task closes the queue. Destroy means to de-allocate the memory associated with queue ECB. 2. <code>pthread_mq_setattr ()</code> sets the attributes. 3. <code>pthread_mq_lock ()</code> and <code>pthread_mq_unlock ()</code> unlock and lock a queue. 4. <code>pthread_mq_send ()</code> and <code>pthread_mq_receive ()</code> to send and receive into a queue. <code>mq_send</code> four arguments are <i>msgid</i> (message queue ID), (void *) &<i>qsendingdata</i> (pointer to address of user data), <i>sizeof</i> (<i>qsendingdata</i>) and 0. <code>mq_receive</code> five arguments are <i>msgid</i> (message queue ID), (void *) &<i>qreceivedata</i> (pointer to address of bufferdata), <i>sizeof</i> (<i>qreceivedata</i>), 0 and 0. 5. <code>pthread_mq_notify ()</code> signals to a single waiting task that the message is now available. The notice is exclusive for a single task, which has been registered for a notification. (Registered means later on takes note of the <code>pthread_mq_notify</code>.) 6. The function <code>pthread_mq_getattr ()</code> retrieves the attribute of a POSIX queue.
7	Queues of functions	The queue of functions are equivalent to character devices, accessed via POSIX <code>read/write/open/ioctl</code> system calls.
8	Time functions	<ol style="list-style-type: none"> 1. <code>nanosleep (sleepime)</code>; /* Sleep for nanosecond period defined in argument <i>sleepime</i> */ 2. <code>clock_gettime ()</code>; /* <code>getthrtime</code> returns high resolution time (in ns) of the clock named clock. The clock is name of the clock (system clock) specified earlier. */ 3. <code>getthrtime ()</code>; /* <code>getthrtime</code> returns high resolution time (in ns) in a thread in which it is used as argument. */ 4. <code>clock_gettime ()</code>; /* <code>gettime</code> returns time of the clock named clock. */ 5. <code>clock_settime ()</code>; /* <code>settime</code> sets time of the clock named clock. */
9	Shared memory functions	Used as an alternative to using the IPCs. <ol style="list-style-type: none"> 1. <code>include <shm.h></code> /* Includes shared memory header*/. 2. <code>include "common.h"</code> /* Includes common memory file header*/.

(Contd)

S.No.	Feature	Description
		<ol style="list-style-type: none"> 3. <code>struc common_struct sharedblk</code>; /* Specify a new structure <i>sharedblk</i> */. 4. <code>int shmID = shmget ((key_t) num, sizeof (struc common_struct), 0666 IPC_CREAT)</code>; /* <code>shmget ()</code> function arguments are (key_t) num (= 2377), common structure size, and a option for shared memory creation or use of IPC. Return ID of shared memory structure*/. 5. <code>sharedMem1 = shmat (shmID, (void *)0, 0)</code> /* Pointer to shared memory block <i>sharedMem1</i> is found by function <code>shmat</code>. The arguments are ID of shared memory block, null pointer function and option 0. */. 6. <code>int shmem_status = shmctl (shmID, IPC_RMID, 0)</code>; /* <code>shmctl ()</code> function arguments are The arguments are ID of shared memory block, <code>IPC_RMID</code> pointer and option 0. Return shared memory status <i>shmem_status</i> */ 7. <code>int shmmdetect_status = shmdt (sharedMem1)</code>; /* <code>shmdt ()</code> function argument is pointer to shared memory block <i>sharedMem1</i>. Returns -1 if not detected. */

10.3.2 RTLinux

An extension of Linux version in which earlier there was no real-time support is a POSIX hard real-time environment using a real-time core. The core is called RTLinuxFree and RTLinuxPro, freeware and commercial software respectively. V. Yodaiken developed RTLinux.

There are relatively simple modifications, which converts the non real time Linux kernel with round-robin scheduler into a hard real-time environment. The deterministic interrupt latency ISRs execute at RTLinux core and other in-deterministic processing tasks are transferred to Linux. The forwarded Linux functions are placed in FIFO with sharing of memory between RTLinux threads as highest priority and Linux functions running as low priority threads. Figure 10.3 shows RTLinux basic features.

Running the task in the following configuration gives hard real-time performance.

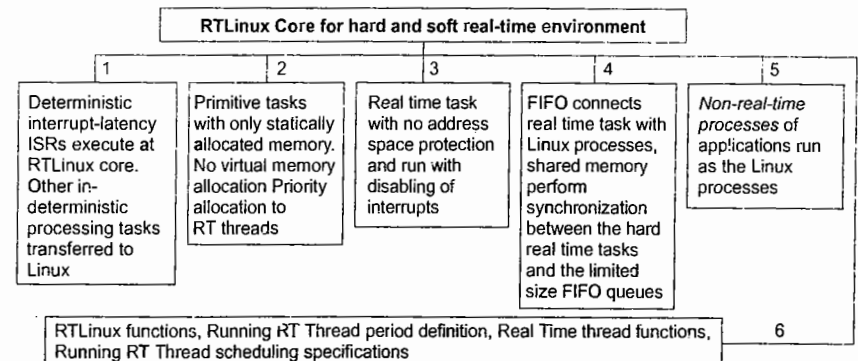


Fig. 10.3 RTLinux basic features

- Run the primitive tasks with only statically allocated memory. The dynamic memory allocation or virtual memory allocation introduces unpredictable allocation and load timings.

- Run the real-time task with no address space protection. The memory address protection involves additional checks, which also introduce the unpredictable allocation and load timings.
- Run with disabling of interrupts so that other interrupts do not introduce the unpredictability.
- Run a simple fixed priority scheduler.
- When the FIFO connects real-time task with Linux processes, perform the synchronization between hard real-time tasks and limited-size FIFO-queues that is achieved through use of shared memory (not through IPCs).

Soft real time-task of applications can be configured to run differently. This is because the RTLinux allows flexibility in defining real-time task behaviour, synchronization and communication because the RTLinux kernel has been designed with modules, which can be replaced to make the behaviour flexible wherever possible.

Non-real-time processes of applications run as the Linux (before 2.6.x version arrived) processes. RTLinux has the following features useful for real-time embedded system design.

1. Small footprint core. The core is like a virtual machine layer. The RTLinux core provides kernel as preempting kernel as well as non-preempting kernel.
2. RTLinux executive cannot be preempted.
3. Real-time tasks run in privileged mode and therefore directly access the hardware and do not use virtual memory. Real-time tasks are written as special Linux modules that can be dynamically loaded into memory.
4. Deterministic worst-case latency for number of processors, x86, ARM, MIPS tested for extended periods of extreme loads is very small.
5. CPU can be assigned to specific real-time threads, for example, during media processing. For a task, priority is introduced and set, and repetition rate and scheduling algorithm can be selected.
6. RTLinux gives high average case performance and the real-time threads meet the strict worst-case limited deadlines.
7. POSIX threads APIs has POSIX 1003.13 PSE51 specification. The latter is a minimal real-time system model specification. Linux or Unix functions are the lowest priority threads with POSIX IPCs using the shared memory and lock-free queues (queuing even when interrupts disabled) and shared memory.

Programming with RTLinux Steps in running a program is to first boot RTLinux. Following examples gives a way to compile the module in RTLinux. It is similar to the one in Linux.

Example 10.2

1. `include rtl.mk` /* Include RTLinux make file. The rtl.mk file is an include file which contains all the flags needed to compile the code. */
2. `all: module1.o` /* Object file at module1.o */
3. `clean: rm -f *.o` /* Remove using function rm object files inserted before this file */
4. `module1.O: module1.c` /* module1.0 is object file of source file module1.c */
5. `$(cc) $(include) $(cflags) -c module1.c` /* Compile, include, Cflags C module module1.c */

Now command \$ compiles the files. Command \$ `rtlinux start module1` inserts (start) object binary file module1.o. Command \$ `dmesg` displays the messages. Command \$ `rtlinux stop module1` removes (stops) object binary file module1.o.

RTLinux supports a module as well as threads initialization, handling the errors, prevention of unauthorized port accesses, usage counts, root-level security and clean up. RTLinux scheduler not only schedules the

threads and modules. The RTLinux supports the registering of the modules, threads, configurations, functions, thread priority allocation and scheduling. Table 10.10 gives the registering, de-registering, priority and scheduling of real-time thread and FIFO-related functions in RTLinux.

Table 10.10 Registering, De-Registering, Priority and Scheduling of Real-Time Thread and FIFO-Related Functions in RTLinux

Function	Action(s)
init insmod, claenup, and rmmod	The <code>init_module()</code> , <code>insmod_module()</code> , <code>cleanup()</code> and <code>rmmod_module()</code> functions same as Linux
RT Linux functions	<ol style="list-style-type: none"> 1. <code>rtl_hard_enable_irq()</code> /* Enables hard real time interrupts */ 2. <code>rtl_hard_disable_irq()</code> /* Gets current clock schedule */ 3. <code>rtlinux_sigaction()</code> /* to RTLinux signal handling function of the application */ 4. <code>rtl_getschedclock()</code> /* Gets current clock schedule */ 5. <code>rtl_request_irq()</code> /*to install real time interrupt handler */ 6. <code>rtl_restore_interrupts()</code> /* Restores the CPU interrupts state */ 7. <code>rtl_stop_interrupts()</code> /* Stops the CPU interrupts */ 8. <code>rtl_printf()</code> /*to print text from real time thread */ 9. <code>rtl_no_interrupts()</code> /* /* No CPU interrupts permitted*/
Thread creation and getting thread ID	Creation and getting ID is same as Linux threads <code>pthread_create()</code> and <code>pthread_self()</code> .
Semaphore and mutex functions	Same as in Linux.
Thread wait	<code>pthread_wait_np()</code> /*thread waits for other thread to complete*/.
Running RT thread period definition	<code>pthread_make_periodic_np(pthread_self(), gethrtime(), 800000000)</code> /* defines periodicity of thread with a thread self. First argument is reference to self thread (thread itself). Second argument is <code>gethrtime()</code> is a function to get thread time. Third argument is to define period of thread run = 800000000 ns. Any other period can be defined in third argument.*/
RT thread deletion	<code>pthread_delete_np(thread_1)</code> /* Delete the thread thread_1*/.
Thread priority	<ol style="list-style-type: none"> 1. <code>struct sched_param thrd1, thrd2</code> /* Defines two structures for threads thrd1, thrd2 assignment parameters. */ 2. <code>thrd1.sched_priority = 1</code> /* Defines thrd1 scheduled priority parameter = 1 (=high). Similarly thrd2 priority can be defined */ 3. <code>pthread_setschedparam(pthread_self(), SCHED_FIFO, &thrd1)</code> /* Function <code>setschedparam</code> arguments are self function, <code>schedule</code> defines as FIFO and thread structure thrd1. Similarly thrd2 arguments for Function <code>setschedparam</code> can be defined */
Put into the FIFO from thread	<code>thrd1_put(fid, &queuebuff, 1)</code> /* Function <code>thrd1_put</code> arguments are <code>fid</code> (= an integer for FIFO descriptor ID, say, 1, 2, 3, ..), address of queue buffer <code>queuebuff</code> and an option = 1 */
Real-time thread functions	<ol style="list-style-type: none"> 1. <code>pthread_attr_setcpu_np()</code> /* Set CPU pthread attributes. */ 2. <code>pthread_attr_getcpu_np()</code> /* Get CPU pthread attributes. */ 3. <code>pthread_attr_setfp_np()</code> /* Set CPU pthread floating point enable attributes.*/ 4. <code>pthread_suspend_np()</code> /* Suspends thread run.*/

(Contd)

Real-time FIFO functions

5. pthread_wakeup_np () : /* Wakes up the thread .*/
6. pthread_wait_np () : /* Wait for start (message or signal) the thread .*/
7. pthread_setfp_np () : /* Allows floating point arithmetic.*/
8. pthread_delete_np () : /* Delete the thread.*/
1. rtf_create (), rtf_create_rt_handler (), rtf_open (), rtf_close () and rtf_unlink () are the functions to create FIFO device, create real-time handler, initialize, close and remove a named RT FIFO. unlink () does not destroy the queue immediately but prevents the other tasks from using the queue. The queue will get destroyed only if the last task closes the queue. Destroy means to de-allocate the memory associated with queue ECB.
2. rtf_setattr () sets the attributes.
3. rtf_lock () and rtf_unlock () lock and unlock a queue.
4. rtf_put (fid, &queuebuff, 1); /* Function rtf_put arguments are fid (= an integer for FIFO descriptor ID, say, 1, 2, 3, ..), address of queue buffer queuebuff and an option = 1 */.
5. rtf_get (fid, &queuebuff, 1); /* Function rtf_get arguments are fid (= an integer for FIFO descriptor ID, say, 1, 2, 3, ..), address of queue buffer queuebuff and an option = 1 */.
6. rtf_send () and rtf_receive () to send and receive into a queue. rtf_send four arguments are fid (FIFO device ID), (void *) &qsendingdata (pointer to address of user data), sizeof (qsendingdata) and 0. mq_receive five arguments are msgid (message queue ID), (void *) &qreceivedata (pointer to address of bufferdata), sizeof (qreceivedata), 0 and 0.
7. rtf_notify () signals to a single waiting task that the message is now available. The notice is exclusive for a single task, which has been registered for a notification (registered means later on takes note of the rtf_notify).
8. rtf_getattr () retrieves the attribute of a RT FIFO.
9. rtf_flush () flushes the data of a RT FIFO.
10. rtf_allow_interrupts : () /* Controls real time interrupt handler */.
11. rtf_free_irq () : /*Frees real time interrupt handler */.

IPC is performed by using a FIFO operation as follows. Real-time FIFOs use one source process (a real-time thread) and one end process (user space process). As one source process sends into the FIFO, another process at the other end of the FIFO receives. RTLinux FIFOs are character devices, (/dev/rtf*). Real-time threads use integers to refer to each FIFO. Integers 1, 2, 3 for FIFO 1, 2 and 3, respectively. There is a limit to the number of FIFOs (150). The RTF (RTLinux-FIFO) devices are rtf1, rtf2, rtf3 and so on. Linux character devices use create(), put(), get () and destroy() functions to device create, send from thread and get into process and device destroy. The RTF device uses rtf_create (), rtf_put(), rtf_get () and rtf_destroy() functions to RTF device create, send from thread and get into process and device destroy. Following is the code example.

Example 10.3

This example shows how to create a thread and defines its real-time periodicity and use RT thread FIFO device to put into the buffer.

```

1. #include rtl.h          /* Include RTLinux header file.*/
2. #include pthread.h     /* Include POSIX thread header file.*/
3. #include rtl_fifo.h    /* Include RTLinux FIFO header file.*/
4. #include time.h        /* Include timer header file.*/

```

```

5. #define fid 5 /* Define FIFO ID = 5 */
6. int open ("/dev/rtf5", O_WRONLY); /* Open FIFO device rtf5*/
7. struct pthread_t clientThd_1 /* Create a client thread 1*/
8. int init_module (void) {
9.     return pthread_create(&clientThd_1, NULL, thread_1, NULL); /* creates thread with structure
clientThd_1 and calling function thread_1. Returns the thread ID return. */
10. }
11. void cleanup_module (void)
12. { pthread_delete_np (clientThd_1); /* Delete the thread function */
13. }
14. void * thread_1 (int fid) { /* Thread1 function */
15.     static char queuebuff = 0; /* Define FIFO buffer address */
16.     pthread_make_periodic_np (pthread_self(), gethrtime(), 800000000); /* Define thread period
as 800000000 ns */
17.     while (1)
18.     {pthread_wait_np(); /* wait for RT thread */
19.         queuebuff = (int) queuebuff ^0xff; /* Define new FIFO buffer address */
20.         rtf_put(fid, & queuebuff, 1); /* Put into the queuebuff FIFO defined by fid */
21.     } return 0; }

```

**Summary**

The following is the summary of what we learnt in this chapter.

- Windows CE (WCE) is an operating system for systems, handheld and mobile devices, which have resource constraints of power, memory, touch screen or display screen size and processing speeds. It has extensions for pocket PCs and automobiles.
- WCE is an open, scalable and small-footprint 32-bit OS.
- An enhancement of WCE is Windows CE.NET. .NET framework provides for compiling the managed code. Managed code is one that is compiled in CIL (common intermediate language). It gives platform-independent CPU neutral compilation as the byte codes.
- WCE provides a Windows platform for the systems and it gives a user to feel, look and interact with the system using GUIs in a manner similar to a PC running on Windows.
- 80x86 or SuperH or ARM, SH4 and MIPS-based processor architectures are supported by WCE and the WCE fine tunes the processor performance.
- There is *componentization* of OS. OS has two layers: one is the source code and the other is the shared code with the devices manufacturer.
- A Windows-based application program is written to respond or activate or changes from the current state on pushing off notification(s) from the OS. A notification occurs on an event. The notification sends the *message* to the Windows application program. Messages are placed in queue for the Windows of the application program.
- Win32 API subset is used for GUIs programming.
- WCE supports the ISRs which pass the messages to ISTs, which run as lower-priority threads than the ISRs and ISTs run as priority queue of threads.
- Windows uses Handle in many procedures (functions). WCE does not support inheritance of Handles.
- WCE thread assigned priorities to each one among the eight levels: System-level threads and device drivers (ISTs) use the upper 248 levels of priorities.

- Windows CE 6.0 supports 2^{16} number of processes (earlier 32) with each process having a virtual memory limit of 2 GB (earlier 32 MB) and up to lower 2 GB (earlier 32 MB). There is also upper 2 GB (earlier 32 MB) on the kernel VM space.
- Windows Mobile 6 supports Office Mobile 2007, smartphone with touch screen, improved Bluetooth stack, VoIP with AEC support to encryption of data stored in external removable storage cards, uses smartfilter for fast files, e-mail, contacts and songs search and can be set as modem for laptop.
- WCE provides for system memory between 1 MB and 64 MB and OS needs minimum 512 kB of memory and 4 kB of RAM. WCE also provides for managing the low memory conditions.
- WCE considers the RAM in two sections: *program memory* (called system heap) and *object store*.
- WCE has file systems and databases. The database has a series of un-lockable records with saving of a property(ies) and data together in a record. No record contains another record within it. Each record has four-level indices for sorting.
- WCE is a multitasking multithreaded program. There is at least one thread which is created per process. The basic unit of execution under control of the kernel is the thread. Each thread has a separate context (for CPU register including PC and stack pointer) and stack for saving the context when thread blocks and for retrieving on activating the thread.
- WCE provides for exception handling signals, notification signals, event functions (for threads of a process), semaphores, message queues, wait single object, multiple object and multiple message object functions.
- WCE provisions for GUIs based on Windows, menus, dialog boxes, radio and check buttons.
- Subset of Win32 APIs in WCE provision for inputs from keys, touch screen or mouse, communication with serial port, Bluetooth, IrDA, WiFi, networking, device-to-device socket and communication functions.
- OSEK is a structured and modular software implementation based on standardized interfaces and protocols for the automobile distributed ECUs (electronic control units).
- OSEK/VDX specifies three standards, which give the portability and extensibility features, and thus the reusability of the existing software. OSEK specified three standards: one for embedded OS using *MODISTARC* (methods and tools for the validation of OSEK/VDX-based distributed architectures and for conformance testing of the OSEK/VDX implementations). The second is for communication and the third is for network management.
- OSEK uses abstract interfaces and implementation can be as per hardware and network. OSEK has standardized interfacing features for control units with different architectural designs. OSEK has an efficient design of architecture. The functionalities in the OSEK standard shall be configurable and scalable, to enable optimal adjustment of the architecture to the application in question.
- OSEK defines four types of classes BCC1, ECC1, BCCC2, ECC2COM and conformance classes CCCA and CCCB for internal communication. It specifies that there should be no creation and deletion of tasks during run. Task Priority must be defined and task activates only once in the codes. There must not be use of message queues, and semaphores must be used as flag only.
- Linux uses POSIX processes, threads, shared memory, POSIX map and POSIX queues.
- Linux has a number of interfaces for the user. For example, Graphic editors for configuring the kernel, X-Windows for GUI and csh (for C shell). Linux has a number of characters, blocks and network interface devices and has device drivers for the user-programming environment.
- Real-time Linux besides support to the module initialization, handling the errors, prevention of unauthorized port accesses, usage counts, root-level security, insert, remove and clean-up functions, also supports preemptive scheduling. Latest version of real-time Linux is Linux 2.6.24.
- RTLinux has real-time thread wait, thread period definition, thread deletion, priority assignment and FIFO device functions. It provides for running of real-time tasks by RTLinux layer and no deterministic non-real-time tasks by Linux. A FIFO connects real-time tasks with Linux processes, performs the synchronization between hard real-time tasks and limited-size FIFO queues through the use of shared memory (not through IPCs).
- RTLinux provides hard real functionalities in a separate layer, which runs the primitive tasks (real-time threads) with only statically allocated memory, no dynamic memory allocation, no virtual memory allocation, no address space protection, runs with disabling of interrupts, runs a simple fixed priority scheduler.
- RTLinux has separate functions. `rtl_hard_enable_irq ()`; `rtl_hard_disable_irq ()`; `rtlinux_sigaction ()`; `rtl_getschedclock ()`; `rtl_request_irq ()`; `rtl_restore_interrupts ()`; `rtl_stop_interrupts ()`; `rtl_printf ()`; and `rtl_no_interrupts ()` and has real-time FIFO functions.



Keywords and their Definitions

- Bitmap** : A bitmap is a graphical object which can be drawn on screen and is used for creating, retrieving images, manipulating and drawing at the device context.
- Block device** : A device which is accessed by a file system (disk) like commands and in which a block is accessed at an instant.
- Button** : A button is a window wherein the bringing the mouse or stylus near it (in-focus) or taking it away (out-of-focus) or clicking or tapping over it on the screen initiates a notification from the OS, which then notifies to an API for running.
- Character device** : A device which accesses byte-by-byte analogous to the access from and to a console or keyboard or printer device using byte streams.
- Child process** : A process created in Linux by `fork ()` function, which sends new process ID to parent and makes self ID = 0, and which is made to perform different functions than the parent by overloading function of the parent process using `execv ()` function.
- Console** : A video terminal or touch screen or display object for GUI and for displaying output of the programs in a computer.
- Control** : An object for controlling program flow, for example, a command, menu or toolbar bar object or a date and time picker control object or calendar picker control or edit control to auto-capitalize the first character of a word when keying in and virtual keyboard or organizer (e.g., task-to-do).
- Componentization** : Provisioning of shared source and source code accesses provided by Microsoft such that there are two software layers. One sublayer consists of Microsoft-developed source codes of the WCE kernel and is shared with the system or the device manufacturer. Then the manufacturer adds the remaining part of the kernel according to the system hardware. The remaining part is the hardware abstraction layer.
- Device context** : A device context specifies the application Window, which sends the pixels to the screen, which is a tool, which Windows use in managing the access to the display and printer, which has two attributes of device context colours for background and foreground, has font of the displayed text, text alignment attributes left, right, top, centre, bottom, no update and update of current point and baseline alignment.
- DLL** : Dynamic link library is file, which is linked at run time of .exe file and which loads on request from the .exe file or another DLL.
- ECU** : An electronic control unit for controlling the unit and its communication with other ECUs in an automobile or other systems.
- Exception** : A signal from API for initiating a Window notification when an exception condition is arrived at run time.
- Execute-in-place file** : A file in ROM for execution that cannot be opened and read by standard file functions for *open* and *read*.
- FIFO** : A device which creates like a thread or file and which is sent.
- GUIs** : Graphic user interfaces for facilitating interaction and inputs from the user after graphic screen displays of menus, buttons, dialog boxes, text fields, labels, checkbox and radio buttons and others.
- Handle** : Windows uses Handle in many procedures (functions). The Handle provides reference to an interface, for example for a Window, file or thread or port. An example is `INSTANCE`, a Window object for use as Handle. An interface is an unimplemented

- procedure (function or method) the codes for which are defined in the class, which uses that interface. Handle is also used as a pointer, called *option pointer*.
- IPC object** : An object in a multitasking or multithreading system for communication events, semaphores including mutex semaphores and messages into queues without using shared memory and which provides for threads synchronization.
- IST** : Interrupt service thread is a thread which is put in the priority queue of threads waiting for execution and which is a slow-level interrupt service thread of a fast-level ISR, which posts the message(s) to that.
- Linux** : A freeware OS named Linux is after Linus Torvalds, father of the Linux OS.
- Managed code** : The code that is compiled in CIL. It gives platform-independent CPU neutral compilation as the byte codes. A run-time environment converts the byte code instructions into the native machine and platform instruction.
- Network interface** : An interface device, which accesses using a network protocol such as Telnet, TCP, UDP, SLIP, PPP, SMTP or Bluetooth.
- Notification** : The OS monitors all input sources (e.g., stylus tap, virtual (on-touch screen) or physical key press). The OS notifies and sends notification when a key has been pressed or a button has been clicked or command has been received for redrawing the Windows screen. A notification as a Windows-based application program is written to respond or activate or changes from the current state on pushing of notification(s) from the OS. A notification occurs on an event. The notification sends the *message* to the Windows application program. Messages are placed in queue for the Windows of the application program.
- Object store** : A virtual RAM disk is a permanent store for storing files, registry, PIM, and WCE databases and is protected from power turning off. Individual file can use up to 32 MB in case of RAM as *object store*. If object is not found in RAM, it is loaded from the *object store*.
- Option pointer** : A pointer which points to a pointer of one of the several sets of the codes, which run on selecting the option.
- OSEK** : OSEK/VDX is a body for defining and specifying three standards for an embedded OS, communication and network management for automotive applications and control of ECUs.
- Thread** : POSIX thread, a thread for which POSIX standard functions are used to create, send, receive, suspend and which uses POSIX IPC semaphores, mutex, message queues and FIFOs.
- PocketPC** : Handheld PC for home as well as office systems and for cellular networks connectivity using Windows CE extension, Windows Mobile, for applications such as Outlook, Explorer, pocket versions of Word, Excel and pocket Power Point, slide shows, mailing, internet and office applications.
- Power manager** : Software to reduce the power dissipation by reducing clock speed or running wait or stop instruction or optimizing use of caches or stopping screen or reduced intensity displays after limited wait for user input.
- Real time core** : A layer of functions for real time threads which has higher priority than other kernel functions.
- Registry** : Registry API for system database using standard file registry functions: RegCreateKeyEx, RegOpenKeyEx, RegSetValueEx, RegQueryValueEx, RegDeleteKeyEx, RegDeleteValueEx and RegCloseKey.
- Signal** : An interrupt or notification object, which initiates a handler function (interrupt service routine), provided signal is not masked and interrupts are enabled.

- Socket** : An API at the streaming sockets or datagram to send and receive between two specific addresses at two APIs at different devices.
- Stylus** : A writing pencil-core-shaped object for a user of device or system as an alternative to the mice and keyboard. The user touches the stylus tip at the displayed menu or displayed keypad on the screen to enter the commands or text, respectively.
- System heap** : The system APIs and OS in the *program memory* area of memory.
- Touch screen** : A device for screen displays as well as for accepting inputs through a stylus.
- VUIs** : Voice user interfaces which facilitate interaction and command inputs using stored voice or tunes, and voice command inputs from the user.
- Win32 APIs** : APIs for 32-bit programming using Window classes, objects, controls and Handles and for managing, displaying and drawing the Windows for the user APIs.
- Window** : An object INSTANCE, which defines a Window (object) to provide Handle(s) for the GUIs and APIs for running the application codes. The object Window has basic coordinates *x* and *y*, and *z*-parameters, specification for visibility (show or hide or no activate), specification for parent-child hierarchy and procedures to share the attributes and to respond the requests and all notifications sent to the Windows.
- Windows CE** : A Win32 API subset-based OS for handheld computers and mobile systems, developed by Microsoft that provides a programming environment using a subset of Win32 APIs, Visual C++, Visual Basic, and that enables a user to feel, look and interact with the system using GUIs in a manner similar to a PC running on Windows.
- Windows CE.NET** : An enhancement of Windows CE deploying .NET framework that provides for compiling the managed code.
- Window Mobile** : OS with extensions for Office Mobile 2007, smartphone with touch screen, improved Bluetooth stack, VoIP with AEC, support to encryption of data stored in external removable storage cards, uses smartfilter for fast files, e-mail, contacts and songs search, can be set as modem for laptop.
- WinSock** : Streaming or datagram socket APIs in Windows, which has a feature of accessing personal area and network resources and that does not depend on platform and implementation of socket functions.



Review Questions

1. Describe the features of Windows CE. What is the advantage in using .NET framework with Windows CE? Why does the Windows CE have low interrupt latencies?
2. Describe the additional features in Windows CE 6.0 and Windows CE extensions to Pocket PC, Windows Mobile 6 and Windows Automobile 5.0.
3. What are the differences in programming with Windows CE with respect to Windows? Explain meaning of Handle and Handle inheritance. What is the advantage of withdrawing reducing Handle inheritance in Windows CE?
4. What do you mean by Windows and relationship between the Windows? List the functions for management of Windows.
5. Describe memory management. Explain the similarity in file management and device management functions.
6. Describe the properties and Windows CE functions for the databases.
7. Windows CE and Linux are multitasking-multithreaded OSes, and MUCOS and VxWorks are multitasking OS. Explain the difference between two concepts. What is the basic unit of computations in each of the OSes? Describe the properties and Windows CE functions for threads and processes.

8. Explain the Windows CE exceptions, notifications and interprocess communication objects and their synchronization.
9. How do the inputs from keys, touch screen or mouse handled in Windows CE?
10. Describe Windows CE serial communication functions. Describe WNet API network connection functions.
11. List WinSock API subset in Windows CE for device-to-device socket-communication functions. How do communication functions for socket differ from the serial port communication?
12. How is Win32 API used for development of Windows for the GUIs in an application?
13. What are the embedded softwares in automotive system that need special features, standards and specifications in its operating system?
14. What is OSEK/VDX? What are three standards specified for an embedded operating system OS by OSEK/VDX?
15. List the OSFK basic features. How does the OSEK provide for functional extendibility and portability?
16. What do you mean by classes BCC1, ECC1, BCCC2 and ECC2? Why are the message queues not used, semaphores used as flags only and tasks are created and defined priorities at the beginning only in OSEK standard?
17. (a) List the features of real-time Linux. (b) Describe the functions for registering and de-registering related functions of Linux modules. (c) How does a child process creates in Linux by fork () function? (d) What are the features added in Linux 2.6.x, which enables real-time system design? (e) What are the features added in Linux 2.6.24?
18. List the Linux functions for the signal, multithreading and semaphore and message queue for IPC.
19. How does RTLinux add a new core for real-time-tasks? How does hard real-time task executes in RTLinux? How does the shared memory help as a fast alternative to the use of IPC?
20. What are new functions added in RTLinux? Describe registering, de-registering, priority and scheduling of real-time thread and FIFO-related functions in RTLinux.



Practice Exercises

21. Write the codes in Win32 API subset of Windows CE for displaying a contact name, telephone number, address and e-mailID at the Window after studying Listing 1-3 in Douglas Boling *Programming Microsoft WINDOWS CE.NET*, Microsoft, USA, 2003.
22. Write the codes for getting messages on Windows using touch screen after studying Listing 3-2 in Douglas Boling *Programming Microsoft WINDOWS CE.NET*, Microsoft, USA, 2003.
23. Write the codes for viewing on a Window a file after studying Listing 8-1 in Douglas Boling *Programming Microsoft WINDOWS CE.NET*, Microsoft, USA, 2003.
24. Write the codes for querying a database in Windows CE.
25. Write the codes for creating two threads with highest and lowest priority, respectively, in Windows CE.
26. Write the codes for sending and receiving bytes between two Windows CE threads using message queues.
27. Write the codes for creating serial port and for sending and receiving bytes in Windows CE.
28. Write the codes for setting user notification and for acknowledging notification in Windows CE.
29. Write the codes for sending and receiving data between the sockets after creating sockets using SOCKET in Windows CE.
30. List the examples of using semaphore flags in an automobile.
31. Write the codes using fork () for creating a child process in Linux.
32. Write the codes for creating the threads for sending and receiving PIM (personal information manager) data. PIM includes data of the contacts, calendar and task-to-do. A contact includes name, address, e-mail ID, phone numbers of home, office and mobile. The data are sent to another thread using synchronization by a POSIX semaphore.
33. Write the codes for creating the threads for sending and receiving Strings data through POSIX message queues in Linux.
34. Write the codes for displaying two texts alternately from two threads using RTLinux.
35. Write the codes for displaying two texts alternately every 8 s from two threads using RTLinux.
36. Write the codes for sending a byte stream into a FIFO in RTLinux.

Design Examples and Case Studies of Program Modeling and Programming with RTOS-1

11

R

We have learnt in chapters 1 and 6, and in an online chapter on 'Software Engineering Approach for Embedded Systems Design' that the following steps are needed as per software engineering and UML modeling approaches.

e

- First, study the *requirements* and be clear of the required purpose, inputs, signals, events, notifications, outputs, functions of the system, design metrics, and test and validation conditions.

c

- Then make the *specifications*, in terms of users, objects, sequences, activity, state and class diagrams for abstracting the component, behavioral and events, and create description in terms of signals, states, and state machine transitions for each task.

a

- Next, define the *system architecture* for hardware and software, extra functionalities and related systems. The architecture specifies the appropriate decomposition of software into modules, components, appropriate protection strategies, and mapping of software.

[

- After defining the design, *implement and test* each component.

[

- Finally, *integrate* components in the system.

Table 11.1 Requirements of an ACVM

Requirement	Description
Purpose	To sell chocolate through an ACVM from which children can automatically purchase the chocolates. The payment is by inserting the coins of appropriate amount into a coin slot. [Adults are also welcome to use the machine!]
Inputs	<ol style="list-style-type: none"> 1. Coins of different denominations through a coin slot. 2. User commands.
Signals, events and notifications	<ol style="list-style-type: none"> 1. A mechanical system directs each coin to its appropriate port—Port_1, Port_2 or Port_5. Each port generates an interrupt on receiving a coin at input. Each port-interrupt starts an ISR, which increases value of <i>amount-collected</i> by 1 or 2 or 5 and posts an IPC to a waiting task. 2. Each selected menu choice sends a notification to the system.
Outputs	<ol style="list-style-type: none"> 1. Chocolate, and a signal to the system that subtracts the cost from the value of <i>amount-collected</i>. 2. Display of the menus for GUIs, time and date, advertisements, and welcome messages.
Functions of the system	A child sends commands to the system using a GUI (graphic user interface). The GUI consists of the LCD display and keypad units. A touchscreen is another alternative for LCD and keypad. The child inserts the coins for the cost of chocolate and the machine delivers the chocolate. If the coins are not inserted as per the cost of chocolate in reasonable times then all coins are refunded. If the coins inserted are of more amount than the cost of chocolate, the excess amount is refunded along with the chocolate. The coins for the chocolates purchased collect inside the machine in a collector channel, so that the owner can get the money through appropriate commands using a GUI. USB wireless modem enables communication to ACVM owner.
Design metrics	<ol style="list-style-type: none"> 1. <i>Power Dissipation</i>: As required by mechanical units, display units and computer system 2. <i>Performance</i>: One chocolate in two minutes and 256 chocolates before next filling of chocolates into the machine (assumed) 3. <i>Process Deadlines</i>: Machine waits for a maximum of 30 s for the coins and the machine should deliver the chocolate within 60 s. 4. <i>User Interfaces</i>: Graphic at LCD or touchscreen display on LCD and commands by children or machine owner through fingers on keypad or touch screen 5. <i>Engineering Cost</i>: US\$ 50000 (assumed) 6. <i>Manufacturing Cost</i>: US\$ 1000 (assumed)
Test and validation conditions	<ol style="list-style-type: none"> 1. All user commands must function correctly. 2. All graphic displays and menus should appear as per the program. 3. Each task should be tested with test inputs. 4. Tested for 60 users per hour.

We will learn embedded systems design from the three case studies discussed in this chapter and four in the next chapter. The first case study is of an **automatic chocolate-vending machine (ACVM)** which was introduced earlier in Section 1.10.2. The second study is of a digital camera, introduced earlier in Section 1.10.4. The third study is of a TCP/IP stack, which was introduced earlier in Section 3.11.

The objectives of these case studies are as follows:

1. To learn how the **requirements** are studied and **specifications** of a system are listed.
2. To learn how **UML modeling** is used to model the design of the system.
3. To learn to define **hardware architecture** using microprocessor or microcontroller or ASIPs or DSPs, and devices.
4. To learn how to define the **software architecture** for software, extra functionalities and related systems and define the decomposition of software into modules, components, appropriate protection strategies, and mapping of software.
5. To learn coding for design **implementation** using MUCOS and VxWorks RTOSes, and also the use of IPCs for task synchronization and concurrent processing.

11.1 CASE STUDY OF EMBEDDED SYSTEM DESIGN AND CODING FOR AN AUTOMATIC CHOCOLATE VENDING MACHINE (ACVM) USING MUCOS RTOS

ACVM was introduced earlier in Section 1.10.2. It listed ACVM functions, hardware and software units. Figure 1.12 showed a diagrammatic representation of ACVM.

Sections 11.1.1 to 11.1.6 give the design steps of an ACVM. Section 9.2 described MUCOS RTOS. It has a portable, ROMable, scalable, preemptive, real time and multitasking kernel. Section 11.1.7 describes coding for ACVM using MUCOS RTOS (Section 9.2) programming environment.

11.1.1 Requirements

The requirements of the machine can be understood through a requirement table given in Table 11.1.

11.1.2 Specifications

The ACVM specifications in brief are as follows:

1. It has an *alphanumeric* keypad on the top of the machine. That enables a child to interact with it when buying a chocolate. The owner can also command and interact with the machine.

2. It has a *three-line* LCD display unit on the top of the machine. It displays menus, text entered into the ACVM, pictograms, and welcome, thank you and other messages. It enables a child as well as the ACVM owner to graphically interact with the machine. It also displays the time and date. [For graphic user interactions (GUIs), the keypad and LCD display units or touchscreen are basic units.]
3. It has a *coin-insertion slot* so that the child can insert the coins to buy a chocolate of his/her choice.
4. It has a *delivery slot* so that the child can collect the chocolate, and coins if refunded.
5. It has an *Internet connection port* so that the owner can know the status of the ACVM sales from a remote location.

Specifications of the required functions in detail are as follows:

1. The ACVM displays a pictogram for the chocolate vending company. It displays the GUI. The ACVM can also collect contact and birthday information. It can answer to FAQs (frequently asked questions).

It displays a welcome message in idle state. It also displays the time and date continuously at the right bottom corner of the display screen. It can also intermittently display advertisements or important information during the idle state.

- When the first coin is inserted, a timer starts simultaneously. The child is expected to insert all the required coins in 30s.
- After 30s the ACVM will display a query in case the child does not insert sufficient coins inside it. If query is not answered, the coins are refunded.
- If within a specific time sufficient coins are collected, it displays a message, 'Thanks, wait for a few moments please!', delivers the chocolate through the delivery slot, and displays the message, 'Collect the chocolate and visit us again, please!'

Figure 11.1(a) shows the basic system of an 'Automatic Chocolate-Vending Machine System' (ACVM) machine. Figure 11.1(b) shows ports at the ACVM. The ports receive the inputs and generate events or notifications for the outputs—chocolate or display messages or coins.

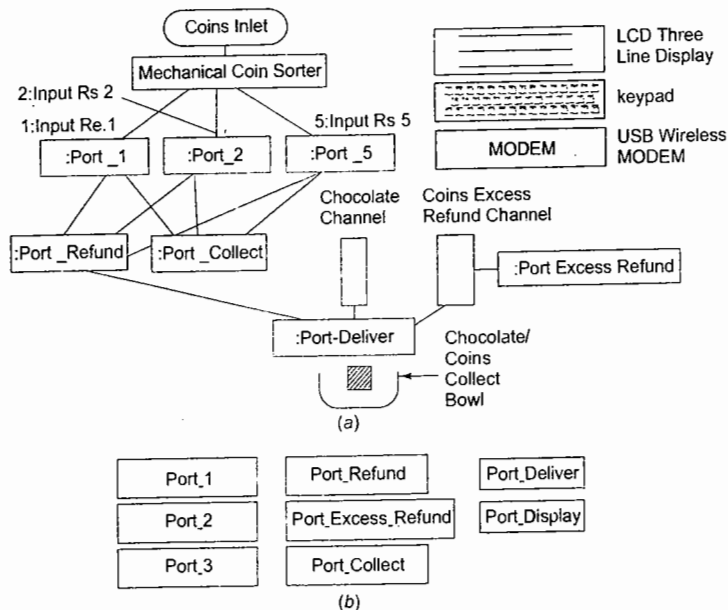


Fig. 11.1 (a) Basic System ACVM (b) Ports of the ACVM

System specifications in detail are as follows:

- There is a slot into which a child (buyer) inserts the coins for buying a chocolate. Suppose the chocolate costs Rs 8. [Children wish that chocolates should be so cheap!] A coin can be in one of three possible denominations: Rs 1, 2 and 5. Whenever a coin is inserted, a mechanical system directs each coin of value Rs 1 or 2 or 5 to Port_1, Port_2 or Port_5, respectively. Each time a port receives a coin, it generates an interrupt, which posts a signal or semaphore to the corresponding task Task_ReadPorts for reading the coin value at the ports to increase the value of a parameter amount_collected.

- The machine should have an LCD for dot matrix display or a touchscreen as 'User Interface'. Let the interface port be called Port_Display. It displays the message strings in three lines with the right-hand last line corner for display of time and date, and top line left-hand corner for pictogram. The pictogram displays the company emblem.
- ACVM has a bowl from where the buyer collects the chocolate through a port for delivery. Let this port be called Port_Deliver. Port_Deliver connects to a chocolate channel. Whenever the channel is left with only a few chocolates, the owner of the machine fills chocolates into the channel. The buyer also collects the full refund or excess amount refund at the bowl through the ports Port_Refund (in the case of short amount) and Port_ExcessRefund (in the case of excess amount) respectively. All ports, Port_Deliver, Port_Refund and Port_ExcessRefund, communicate with Port_Collect by inter-process communication (IPC). Port_Collect is a common mechanical interface to Port_1, Port_2 and Port_5. Port_Deliver is a common mechanical interface to the bowl.
- It should also be possible to reprogram the codes and relocation of the codes in the system ROM or flash or EPROM whenever the following happens: (i) the price of chocolate increases, (ii) the message lines or menus or advertisement or GUI or graphics need to be changed, or (iii) machine features change.
- An RTOS schedules the buying tasks from start to finish. Let MUCOS be the RTOS used in the ACVM.

11.1.3 Specifications Modeling Using UML

Section 6.5 described that UML is a powerful modeling language for (i) software visualizing, (ii) data design(s), (iii) algorithms design, (iv) software design(s), (v) software specifications, and (vi) software development process. The ACVM specification can be modeled using UML.

Class Diagram A class diagram shows how the classes and objects of a class relate, and also hierarchical associations and object interaction between the classes and objects. Rectangular boxes show the classes, and arrows with unfilled triangles at the end show the class hierarchy. Classes in the hierarchy can be joined using a line. Start and end numbers on a line shows how the objects of a class associate with objects of another [Table 6.3].

An ACVM system can be modeled by three class diagrams of abstract classes—ACVM_Devices, ACVM_Output_Ports and ACVM_Tasks. The tasks are the processes or threads that are scheduled by an operating system. Figure 11.2 shows three class diagrams of ACVM_Devices, ACVM_Output_Ports, and ACVM_Tasks. These demonstrate how the class diagrams are shown using UML.

- ACVM_Devices is an abstract class from which the number of extended classes is derived for the devices to handle ACVM mechanism. The devices are keypad, display device, wireless USB modem and coins-input device. Therefore, abstract class ACVM_Devices has four extended classes—User_Keypad_Input, Display_Output, Coins_Input and Wireless_USB_Modem.
- ACVM_Output_Ports is an abstract class from which the number of extended classes is derived for handling output ports at ACVM.
- ACVM_Tasks extends the two classes, ACVM_System_Tasks and ACVM_System_ISR. ACVM_System_Tasks is an abstract class from which we assume that n extended classes Task1-Taskn are derived. Task1 to Taskn are the n tasks (processes or threads) at the ACVM. ACVM_System_ISR is an abstract class from which we assume that m classes are extended. ISR_Task1_ISR_Taskm are extended classes for ISR handling tasks at ACVM. [ISR_Task1 to ISR_Taskm are the m ISRs at ACVM. An ISR_Task initiates and runs on a signal or interrupt or event or notification or exception. A ISR_Task differs from a Task in the sense that ISR_Task has higher priority than the Task and an ISR_Task can only post the IPC object(s) (for example, signal, event semaphores, mailbox message

for notification, message-queue message) to a task(s) [or to an interrupt service thread in Windows CE], while the Task can wait for the IPC(s) as well as post IPC(s) to other tasks. The Task can also run as per preemptive or time slicing or any other scheduling [Section 7.6.2]. MUCOS environment schedules a task in preemptive mode.

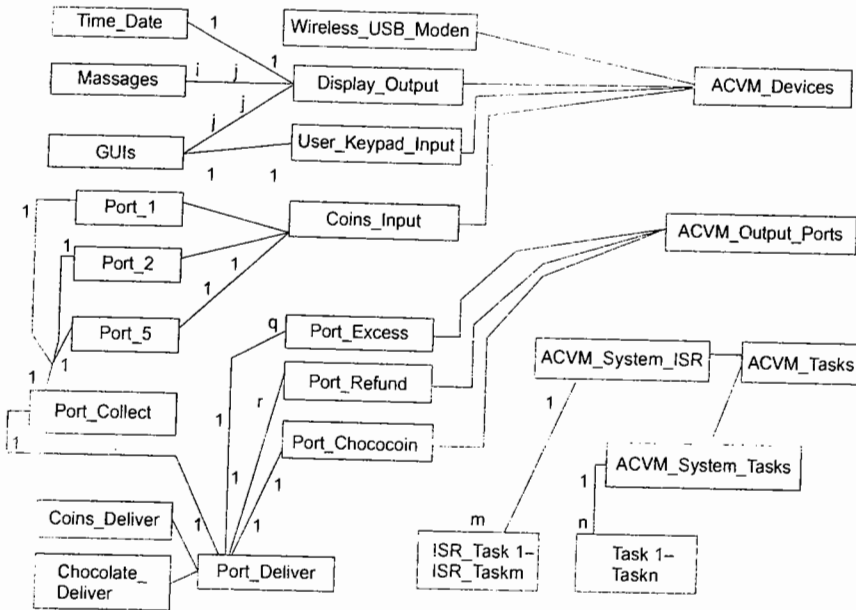


Fig. 11.2 Class diagrams of ACVM devices, ports, and tasks

Class Figure 11.3 shows examples of two classes, Display_Output and User_Keypad_Input, to demonstrate how the classes are shown using UML.

The Display_Output class has the following fields for the following: (i) *picture* for a pictogram, (ii) three strings, *Str1*, *Str2*, *Str3*, for text lines 1, 2 and 3 respectively, (iii) *menuChoiceOffered* string for the choice offered at an instant to graphic user, (iv) *time* that is an array of four integers for the current time in hour and minutes, (v) *amFlag* (= 1 when time is AM else 0), (vi) *pmFlag* (= 1 when *amFlag* = 0), and (vii) *date* is an array of four integers, for two digits each for month, day and year.

Display_Output has four methods (functions in C): *displayPicture* for showing an emblem or other picture, *displayMessage* () for showing a message, *displayTimeDate* () for showing the time and date, *displayMenu* () for showing a menu and *displayAdv* () for showing an advertisement.

The User_Keypad_Input class has the following fields: (i) *inputLine* for the characters received for input line, for example, text during a child keying in the ID or name, address, date of birth, (ii) *advertisement*: array [] of *inputChar* (iii) *inputChar*: character, (iv) eight flags— *choiceAcceptanceFlag*, *upKey*, *downKey*, *leftKey*, *rightKey*, *enterKey*, *YKey* or *NKey* for notifying the acceptance of the selected choice or keying of up or down or left or right or enter or Y or N key, respectively. *YKey* notifies yes and *NKey*, no.

User_Keypad_Input has three methods: *getFlag* (), *getString* () and *getChar* (). These are for getting a Boolean or string or character from the keypad.

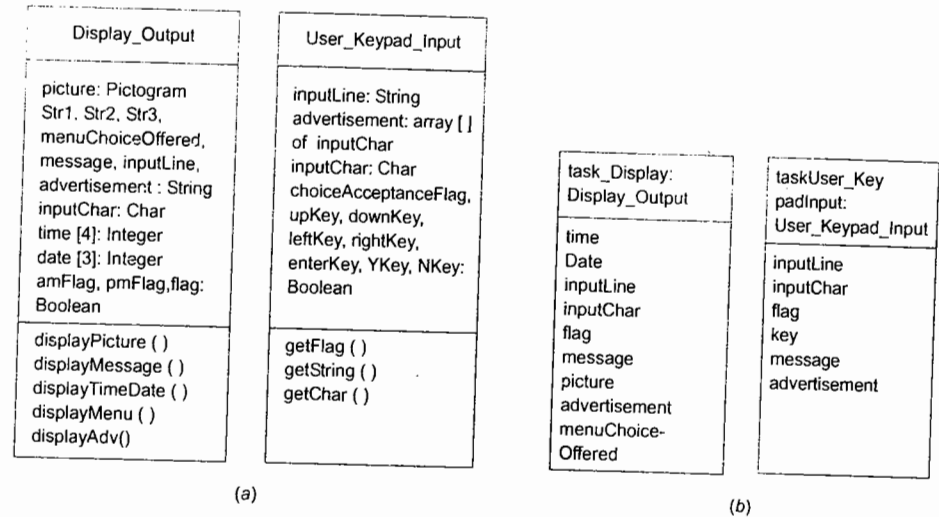


Fig. 11.3 (a) Classes Display_Output and User_Keypad_Input (b) Objects Task_Display and TaskUser_Keypad Input

Object Figure 11.3(b) shows examples of two objects based on classes Display_Output and User_Keypad_Input. It demonstrates how the objects are shown using UML. An object is shown by a rectangular box with the object identity followed by a semicolon and then the class identity of which that object is an instance and functional entity. Task_Display and TaskUser_KeypadInput are the instances of Display_Output and User_Keypad_Input.

State Diagram Figure 11.4 shows a state diagram for ACVM tasks. It demonstrates how the classes are shown using UML. A state diagram shows a model of a structure for its start, end, in-between associations through the transitions and shows event labels (or conditions) with associated transitions. A state diagram is represented as follows: A dark circular mark shows the starting point, and arrows show the transitions. A label over an arrow shows the condition or event, which fires that transition. A dark rectangular mark within a circle shows the end. [Table 6.3] The state transitions take place between the tasks Task_GUI, Task_Display, TaskUser_KeypadInput, Task_Communication, Task_ReadPorts, Task_Refund, Task_ExcessRefund, Task_Collect, Task_Deliver and Task_Display.

11.1.4 ACVM Hardware Architecture

Hardware architecture specifies the appropriate decomposition of hardware into processor(s), ASIPs, memory, ports, devices, and mechanical and electromechanical units. It also specifies interfacing and mapping of these components. Figure 11.5 shows a block diagram of ACVM hardware architecture. Following are the specifications:

1. Microcontroller 8051MX. This version enables use of RAM and ROM larger than 64 kB.
2. 8 MB ROM for application codes and RTOS codes for scheduling the tasks. 64 kB RAM for storing temporary variables and stack.

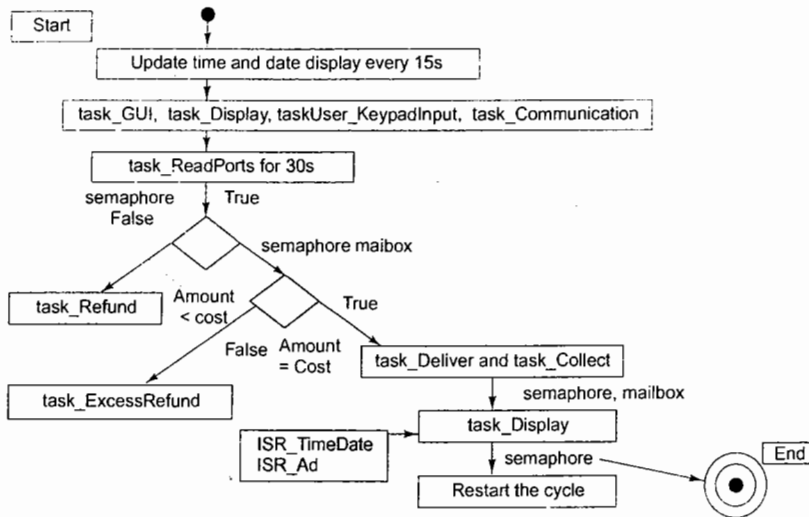


Fig. 11.4 State diagram for ACVM tasks

- 64 kB flash memory part of the ROM stores user preferences, contact data, user address, user date of birth, user identification code, and answers of FAQs.
- A 1 μ s resolution timer is obtained by programming 8051 timer T0 interrupt service routine. Eight hardware-interrupts with 8 interrupt vectors are used for servicing the hardware interrupts.
- A TCP/IP port provides Internet broadband connection through a wireless USB modem, for remotely controlling the ACVM and for retrieving the ACVM status reports by the owner. The ACVM can send warning and error reports to the owner. The Internet port helps in the future updating of the machine to install new applications. For example, an application which sends SMS messages upon arrival of fresh chocolates at the machine or e-mail birthday greetings to users.

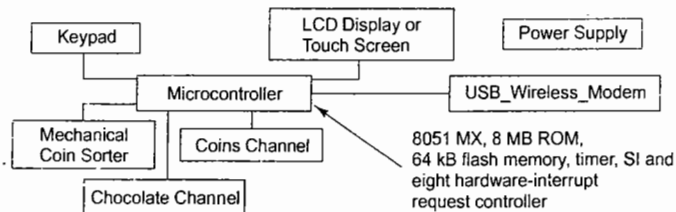


Fig. 11.5 Block diagram of ACVM hardware including Microcontroller

- ACVM specific hardware to *sort the coins of different denominations*. Each denomination coin generates a set of status bits for the coin inputs and a port-interrupt request (notification for hardware event). Using an interrupt service routine for that port, the ACVM processor reads the port status and input bits. The bits give the information as to which type of coin has been inserted. After each read operation, the status bits are reset by the routine.

- Main power supply of 220 V 50 Hz or 110 V 60 Hz. Internal circuits are driven by a supply of 5 V 50 mA for electronic and 12 V, 2 A for mechanical systems.

11.1.5 Software Architecture

Software architecture specifies the appropriate decomposition of software into modules, components, appropriate protection strategies, and mapping of software.

Figure 11.6 shows a design for the software architecture using the classes for Tasks and ISRs. ACVM_Tasks extends to the ACVM_System_ISRs and ACVM_System_Tasks. ACVM_System_ISRs extends to m ISRs, ISR_Task1, ..., ISR_TaskM. ACVM_System_Tasks extends to Task1, ..., TaskN.

ISR_Task1, ..., ISR_TaskM and Task1, ..., TaskN are as follows:

- ISR_KeypadInut, ISR_TimeDate, ISR_Ad, ISR_Port1, and ISR_Port2 and ISR_Port5 are the ISRs, which service the device interrupts.

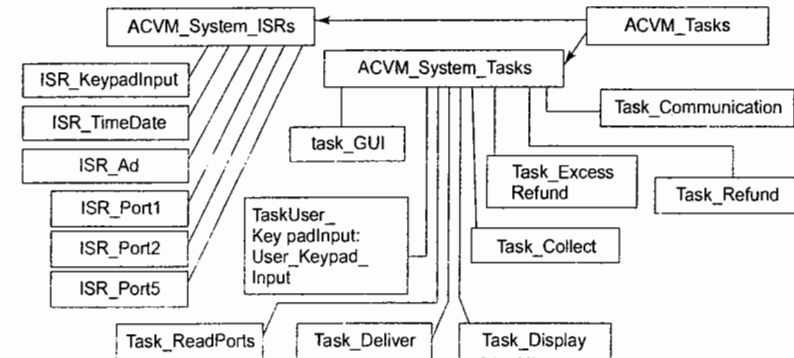


Fig. 11.6 Software architecture of ACVM

- Task_GUI, Task_Display, TaskUser_KeypadInput, Task_Communication, Task_ReadPorts, Task_Refund, Task_ExcessRefund, Task_Collect, Task_Deliver and Task_Display are the tasks, which the kernel schedules.
- ISR_KeypadInput for Keypad input read an interrupt on pressing a key. On getting an interrupt from the keypad, a message is read from the keypad device buffer. The message is as per the sought input at that instance. The message is for one of the following: input_line (for the characters received for input line, for example, text during child keying in the ID or name, address, date of birth), input_character (for example, for the upKey or downKey or leftKey or rightKey or enterKey or YKey or NKey) or an input_choiceAcceptanceFlag. The flag = true (a Boolean when for acceptance of user for the choice marked and shown at display at that instance for one of the offered menu for selection on screen). The input_choiceAcceptanceFlag notifies the acceptance of selected choice. The message input_line, or input_character or input_choiceAcceptanceFlag is notified to the Task User_Keypad_Input.
- ISR_KeypadInput posts the message from the keyboard to the task object of User_Keypad_Input which runs one of the three methods: getFlag(), getString() or getChar(). [Figure 11.3]
- An ISR_TimeDate is for updating the time and date. ISR_TimeDate executes on a timer overflow interrupt, and saves the time and date information. A message is notified to Task_Display. Task_Display executes the method displayTimeDate(). [Figure 11.3]

6. A memory buffer saves the advertisements for display on the ACVM in its idle state at periodic intervals. An ISR_Ad is for selecting an advertisement for display. On a timer overflow interrupt, the ISR_Ad is unmasked at periodic intervals and a software interrupt ISR_Ad is enabled for execution. ISR_Ad posts a message notification to Task_Display. One advertisement is picked at an instant from the memory buffer. The Task_Display executes the method displayAdv ().
7. Task_Display is an object of the class Display_Output [Figure 11.3].
8. Task_Display runs one of the following methods: displayMessage () for messages in mailbox posted from the tasks, and displayTimeDate () or displayAdv() or displayPicture () or displayMenu (). The display method which runs on notification from ISR is according to the message notification posted from ISR_TimeDate or ISR_Ad or ISR_Picture or ISR_Menu.
9. Task_ReadPorts is an object of the class User_Keypad_Input [Figure 11.3]. Task_ReadPorts does the following. (i) It reads the byte (8 bits) at each of the above ports. (ii) If the coin status, as reflected by SemAmtCount, is as per the cost of the chocolate, it sends a flag to Task_Collect. The latter task initiates actions for collecting the coins into a collection unit. (iii) It also resets the bits after reading to keep the ports and all the 24 points in a ready state for the next cycle of the machine. (iv) It does the other actions described in Step 10, if (a) the coins do not accumulate as per the cost within a specific timeout period or are in excess of the cost, or (b) it does other Step 8 actions for displaying messages as per the state of the port just before switching to another task. It sends messages through three mailbox pointers.
10. Task_Collect does the following:
 - (i) It directs the Port_Collect to act and the unit collects all the available coins from the ports. Port_1, Port_2 and Port_5.
 - (ii) After collection is over, it sends an IPC to another task, Task_Deliver, through Port_Deliver. Port_Deliver, on receiving an IPC, initiates action for delivering the chocolate and also posts a signal or semaphore to reset the machine for the next cycle.
 - (iii) Lastly, it sends a message in the mailbox for display in Task_Display.

The following are the details of multiple tasks:

1. Mechanical subsystems also provide a facility, which helps when there are two or more coins of the same type. Thus, there can be a maximum of eight total different points at each port. There are 24 bits, 8 at each port. Port_1, Port_2 and Port_5. These are in reset state (0s) or reset on power-up. The ACVM circuit design is such that at port 0th bit, bit 0 is set (=1) when one coin is available and identified properly. Port bit 1 is set when two are available, bit 2 sets when three are available, and so on. There are 8 bits for 8 points at the port. The number of 1's at the port determines the number of coins at that port.
2. Besides being attached to mechanical subsystems, each of these ports sends 8 input bits for reading at ISR_Port1 or ISR_Port2 or ISR_Port5 using mailbox message pointers. The ISRs send the message pointer for the port bits.
3. Use of a mailbox MboxAmount is such that within a time-out period, a child can thus either insert the chocolate cost by 8 coins of Re 1 for 8 input points at Port_1, or insert a coin of Re 1 for Port_1 point, of Rs 2 for Port_2 point and Rs 5 for the Port_5 point. The child has options for several possible combinations for using the machine. The system recovers the cost of a chocolate before collecting coins and delivering the chocolate.
4. When a port Port_Collect receives a direction (a signal in the form of a flag) from the Task_Collect, then all 24 bits for the 24 points at the three ports release the coins using an electromechanical device. Coins collect at a collection unit. To recover the coins, the machine owner on a convenient time opens

- the unit by a lock and withdraws the money. The machine owner also fills the coins in the unit at Port_ExcessRefund for refunding when the child inserts an extra amount or coin.
5. When a port, Port_Refund, receives a direction (a signal in the form of a flag) from a task, Task_Refund, it directs all the eight points at each of the ports to release the coins if any, using an electromechanical device. The coins drop in a bowl if the amount at the three ports is found to be less than the cost. When a port, Port_ExcessRefund, receives a direction (a signal in the form of a flag) from Task_ExcessRefund, it directs the eight points at another port to release the excess amount, using an electromechanical device, and the coins drop in the bowl, provided the amount at the three ports is found to be more than the cost. [To recover the coins, the child looks at a bowl to withdraw the refunded money.]
 6. Task_Refund does the following: (i) It directs the Port_Refund to act and the unit sends the coins from three ports to a bowl when the amount is short. (ii) It does the other required actions. It sends a display message for the mailboxes, and the mails in which Task_Display waits.
 7. Task_ExcessRefund does the following: (i) It directs the excess refunding on the unit to refund the excess amount. (ii) It does the other actions described in Step 8 and messages to the mailboxes.
 8. At this step, an LCD port gets a message from a Task_Display. The display is as per the state of the machine or time-date mailed to the task waiting mailboxes. The displayed messages are as follows: (i) When the machine resets or on the start of a cycle, in the first line, a welcome message, "Welcome to Sweet Memories Chocolates" is displayed. The second line display is "Insert amount, please". The third line on the right corner displays "time and date" from a Task_TimeDateDisplay which sends the message every second. (ii) After a mail from Task_Collect, the first line shows a message, "Wait for a moment", "Collect a nice chocolate soon" is displayed in the second line. The message clears after a timeout. (iii) After a mail from Task_Deliver, the first line message is "Collect the nice chocolate". The second line message is "Insert coins for more". The message clears after a timeout. (iv) After an event flag from Task_Refund, the message in the first line is "Sorry"! The second line displays "Please collect the refund". The messages clear after a timeout. (v) After the mails, Task_ExcessRefund displays in the first line, "Collect the chocolate and money". "Do not forget to collect the excess" is displayed in the second line. The message clears after a timeout. (vi) After a timeout or on machine reset for the next cycle, the display is repeated as in step (i).
 9. OSSemPost and OSSemPend semaphore functions at MUCOS synchronize such that Task_ReadPorts waits for execution of the codes till the necessary amount, indicated by SemAmtCount, are collected within a specified timeout period. Task_Deliver sends a signal to Port_Deliver that delivers a chocolate from ACVM and it must deliver only on collection of specific coin combinations such that the amount received is equal or more than the chocolate cost.

It is a must that multiple tasks need to be synchronized with respect to each other, using the IPCs. An RTOS is thus required and the binary-semaphores, resource-key semaphores, counting semaphores and mailboxes are used for synchronizing and concurrent processing.

Synchronization Diagram Let MUCOS RTOS be the choice for an embedded software development for ACVM. Figure 11.7 shows multiple tasks and their synchronization model. It demonstrates how to draw synchronization diagrams. The figure mentions synchronization objects near the line connecting a task with another task. The synchronization object(s) is a semaphore(s) or mailbox message(s) for notification. The steps for synchronization are as follows:

1. Task_ReadPorts starts execution only when a semaphore SemFlag1 is posted from the ports. It also accept the semaphore timeout in case available. Timeout semaphore is released by timer service routine for tick after 30s. It accepts amount messages from Port_1, Port_2 and Port_5. It posts message pointers for mailbox waiting for the mail at Task_Collect.

- Task_Collect waits for taking SemFlag1 and MboxAmount. It releases SemFlag2 to let a Task_Deliver provide the chocolate. Task_Collect also releases SemFlag2 again which Task_ExcessRefund takes. This is because the child has already paid extra, so s/he must get a chocolate.

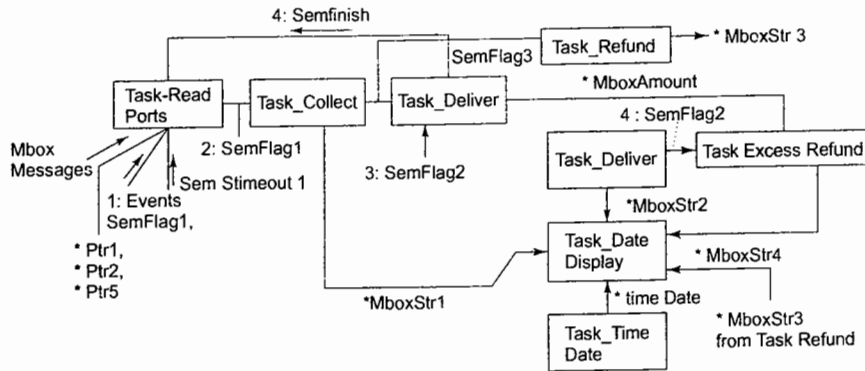


Fig. 11.7 Multiple tasks and their synchronization model using semaphores and mailbox messages

- Task_Refund waits for taking SemFlag3. It sends message to *MboxStr3.
- Task_ExcessRefund waits for taking SemFlag2 and posts mailbox message Str4 for display. Task_Display waits for taking SemFlag4. It takes mutex, SemMKey2 before passing the bytes to a stream for Port_Display and releases it after sending. It displays the mailbox messages at the message pointers. *Collect, *delivered, *refund, and *ExcessRefund.
- Method displayTimeDate () displays at Task_Display gets a timeout notification through a mailbox message for time and date. The time outs occur from ISR_TimeDate after every 1000 ticks of the system clock. A timeout updates the time and date values at a pointer *timeDate. It posts into the mailbox *timeDate to Task_Display and displayTimeDate () uses it to display in the third line, at the right corner of the LCD.

11.1.6 Creating a List of Tasks, Functions and IPCs

For design implementation using a MUCOS RTOS environment, let us create a table. Table 11.2 gives the design table. Task synchronization model is used for creating the table. The table gives name, priority and actions expected from the task in columns 1, 2 and 3, respectively in each row. IPCs pending and IPCs posted are given in columns 4 and 5. Mechanical or other system inputs and outputs are given in the last columns, 6 and 7.

11.1.7 Exemplary Coding Steps

The task objects in Figure 11.6 can be implemented as C functions in a MUCOS environment with C as the programming language. MUCOS IPC functions can implement the synchronization model shown in Figure 11.7. The following are the MUCOS coding steps for the task objects listed in Table 11.2.

Table 11.2 List of Six Tasks, Functions and IPCs

Task Function	Priority	Action	IPCs pending	IPCs posted	ACVM input	ACVM Output
Task_ReadPorts	9	Waits for the coins and action as per coins collected	MboxPtr1Msg, MboxPtr2Msg, MboxPtr5Msg, SemFlag1 messages and Event signals from Port_1, Port_2 and Port_5; Stimeout from timer ISR	Message Pointer *MboxAmount	Coins at Port_1, Port_2 and Port_5.	—
Task_Collect	11	Waits for coins = or > cost till timeout and acts accordingly	SemFlag1, *MboxAmount	SemFlag2, SemFlag3, Message Pointers *MboxAmount, *Str1	—	Coins at Port_ Collect it amount >= cost
Task_Deliver	12		SemFlag2	SemFlag2, Message Pointer *Str2	Chocolate from a channel	Delivers Chocolate into the bowl.
Task_Refund	17	Waits for refund event and refunds the amount	SemFlag3	Message Pointer *Str3	Coins at Port_1, Port_2 and Port_5	Flushed back to the bowl.
Task_Excess Refund	13	Refunds the excess amount	SemFlag2, *MboxAmount	Message Pointer *Str4	Coins at the Ports Excess Refund channel	Excess Coins from Port_Excess Refund
Task_Display	15	Waits for the message mails	Message pointers: Collect, *delivered, *refund, *ExcessRefund (Str2, Str3, Str4) and *timeDate		Strings for line 1, 2 and 3 and time date.	Bytes for the LCD display to lines 1, 2 and 3.

Example 11.1

```
1. /* Define Boolean variable, define a NULLpointer to point in case mailbox is empty. */
typedef unsigned char int8bit;
#define int8bit boolean
#define false 0
```

```

#define true 1
/* Define a NULL pointer; */
#define NULL (void*) 0x0000
/* Preprocessor commands define OS tasks service and timing functions as enabled and their constants;
similar to Example 9.7. */
#define OS_MAX_TASKS 10
#define OS_LOWEST_PRIO 28 /* Let lowest priority task be 19. */
#define OS_TASK_CREATE_EN 1 /* Enable inclusion of OSTaskCreate ( ) function */
#define OS_TASK_DEL_EN 1 /* Enable inclusion of OSTaskDel ( ) function */
#define OS_TASK_SUSPEND_EN 1 /* Enable inclusion of OSTaskSuspend ( ) function */
#define OS_TASK_RESUME_EN 1 /* Enable inclusion of OSTaskResume ( ) function */

/* Specify all child prototype of the first task function that is called by the main function and is to be
scheduled by MUCOS at the start. In Step 11, we will be creating all other tasks within the first task. */
/* Remember: Static means permanent memory allocation */
static void FirstTask (void *taskPointer);
static OS_STK FirstTaskStack [FirstTask_StackSize];
/* Define public variables of the task service and timing functions */
#define OS_TASK_IDLE_STK_SIZE 100 /* Let memory allocation be for an idle state task stack size be
100*/
#define OS_TICKS_PER_SEC 1000 /* Let the number of ticks be 1000 per second. An RTC tick will
interrupt and thus tick every 1 ms to update counts. */
#define FirstTask_Priority 4 /* Define first task in main priority */
#define FirstTask_StackSize 100 /* Define first task in main stack size */
#define cost 8 /* Define cost of chocolate as Rs 8 */
2. /* Preprocessor definitions for maximum number of inter-process events to let the MUCOS allocate
memory for the Event Control Blocks */
#define OS_MAX_EVENTS 24 /* Let maximum IPC events be 24 */
#define OS_SEM_EN 1 /* Enable inclusion of semaphore functions. */
#define OS_MBOX_EN 1 /* Enable inclusion of mailbox functions for the mailing of the
message pointers to Task_Display. */
#define OS_Q_EN 1 /* Enable inclusion of queue functions for sending the string pointers
to LCD matrix Port_Display */
/* End of preprocessor commands */
3. /* Prototype definitions for six tasks */
static void Task_ReadPorts (void *taskPointer);
static void Task_ExcessRefund (void *taskPointer);
static void Task_Deliver (void *taskPointer);
static void Task_Refund (void *taskPointer);
static void Task_Collect (*taskPointer);
static void Task_Display (void *taskPointer);
/* Definitions for six task stacks. */
static OS_STK Task_ReadPortsStack [Task_ReadPortsStackSize];
static OS_STK Task_ExcessRefundStack [Task_ExcessRefundStackSize];

```

```

static OS_STK Task_DeliverStack [Task_DeliverStackSize];
static OS_STK Task_RefundStack [Task_RefundStackSize];
static OS_STK Task_CollectStack [Task_CollectStackSize];
static OS_STK Task_DisplayStack [Task_DisplayStackSize];
/* Definitions for six task-stack sizes. */
#define Task_ReadPortsStackSize 100 /* Define task 1 stack size*/
#define Task_ExcessRefundStackSize 100 /* Define task 2 stack size*/
#define Task_DeliverStackSize 100 /* Define task 3 stack size*/
#define Task_RefundStackSize 100 /* Define task 4 stack size*/
#define Task_CollectStackSize 100 /* Define task 5 stack size*/
#define Task_DisplayStackSize 100 /* Define task 6 stack size*/
4. /* Definitions for six task-priorities. */
#define Task_ReadPortsPriority 9 /* Define task 1 priority */
#define Task_ExcessRefundPriority 13 /* Define task 5 priority */
#define Task_DeliverPriority 12 /* Define task 3 priority */
#define Task_RefundPriority 17 /* Define task 4 priority */
#define Task_CollectPriority 11 /* Define task 2 priority */
#define Task_DisplayPriority 15 /* Define task 6 priority */
5. /* Prototype definitions for the semaphores. */
OS_EVENT *SemFlag1; /* On interrupt signals from Port_1 to Port_5, Task_ReadPorts starts running.
This flag needed when using semaphore for inter-process communication between tasks reading amount
message from Port_1, Port_2 and Port_5. */
OS_EVENT *SemFlag2; /* Needed when using semaphore as flag for inter-process communication for
task_delivery over event notification and between Task_Collect and amount refunding task, Task_Delivery. */
OS_EVENT *SemFlag3; /* Needed when using semaphore as flag for inter-process communication
between Task_Collect and Task_ExcessRefund. */
OS_EVENT *Stimeout; /* Semaphore posted by a timer ISR after 30s wait */
OS_EVENT *SemVal; /* Needed when using semaphore for passing the 16-bit message between steps b
and c. */
6. /* Prototype definitions for the mailboxes */
/* For using mailbox messages. These ISRS are for posting bits from Port_1, 2 and 5. */
OS_EVENT *MboxPtr1Msg;
OS_EVENT *MboxPtr2Msg; /*
OS_EVENT *MboxPtr3Msg;
OS_EVENT *MboxStr1Msg, *MboxStr2Msg, *MboxAmount, *MboxTimeDateStrMsg; /* Needed when
using mailbox message for sending display, amount and timedate pointers */
OS_EVENT *MboxStr3Msg, *MboxStr4Msg;
/*
7. /* Define four Semaphores as event flags
SemFlag1 = OSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as an event flag*/
SemFlag2 = OSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as an event flag*/
SemFlag3 = OSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as an event flag*/
SemStimeOut = OSSemCreate (0); /* declare initial value = 0 for timeout flag */
8. /* Create eight Mailboxes for the tasks. */
MboxAmount = OSMboxCreate (NULL);

```

```

MboxStr1Msg = OSMboxCreate (NULL);
MboxStr2Msg = OSMboxCreate (NULL);
MboxStr3Msg = OSMboxCreate (NULL);
MboxStr4Msg = OSMboxCreate (NULL);
MboxTimeDateStrMsg = OSMboxCreate (NULL); /* For message from Task_Display. */
9. /* Any other OS Events for the IPCs. */
OSMboxPtr1Msg = OSMboxCreate (NULL);
OSMboxPtr2Msg = OSMboxCreate (NULL);
OSMboxPtr5Msg = OSMboxCreate (NULL);
10. /* The codes are for reading from Port A and storing a character. Here, we have three ports, Port_1,
Port_2 and Port_5 for Rs 1, 2 and 5 denomination coins. These are basically device driver codes for
port_1, port_2 and port_5 and three status flags for resetting to the beginning. */
STAF_1 = 0;
STAF_2 = 0;
STAF_3 = 0;

```

11. /* Start of the codes of the application from Main.

Note: Code steps are similar to Steps 9 to 17 in Example 9.16 */

```

int port amount (int *amt); /* declare function to convert string from port to an integer value for amount */
void main (void) {
12. /* Initiate MUCOS RTOS to let us use the OS kernel functions */
OSInit ( );
13. /* Create first task, FirstTask that must execute once before any other. Task creates by defining its
identity as FirstTask. stack size and other TCB parameters. */
OSTaskCreate (FirstTask, void (*) 0, (void *) &FirstTaskStack [FirstTask_StackSize], FirstTask_Priority);
/* Create other main tasks and inter-process communication variables if these must also execute at least
once after the FirstTask. */
14. /* Start MUCOS RTOS to let us RTOS control and run the created tasks */
OSStart ( );
/* Infinite while-loop exits in each task. So there is no return from the RTOS function OSStart ( ). RTOS
takes the control forever. */
} /* End of the Main function */
15. /* The codes of ISR_Keypad Input, ISR_Port 1, ISR_Port 2, ISR_Port 5 for ACVM_System_ISRs */
16. static void FirstTask (void *taskPointer){
17. /* Start Timer Ticks for using timer ticks later. */
OSTickInit ( ); /* Function for initiating RTCSWT that starts ticks at the configured time in the
MUCOS configuration preprocessor commands in Step 1 */.
18. /* Create six Tasks defining by six task identities, Task_Display, Task_ReadPorts,
Task_ExcessRefund, Task_Deliver and Task_Refund and the stack sizes, other TCB parameters.
*/
OSTaskCreate (Task_Display, void (*) 0, (void *) & Task_DisplayStack
[Task_DisplayStackSize], Task_DisplayPriority);
OSTaskCreate (Task_ReadPorts, void (*) 0, (void *) & Task_ReadPortsStack
[Task_ReadPortsStackSize], Task_ReadPortsPriority);

```

```

OSTaskCreate (Task_ExcessRefund, void (*) 0, (void *) & Task_ExcessRefundStack
[Task_ExcessRefundStackSize], Task_ExcessRefundPriority);
OSTaskCreate (Task_Deliver, void (*) 0, (void *) & Task_DeliverStack [Task_DeliverStackSize],
Task_DeliverPriority);
OSTaskCreate (Task_Refund, void (*) 0, (void *) & Task_RefundStack [Task_RefundStackSize],
Task_RefundPriority);
OSTaskCreate (Task_Collect, void (*) 0, (void *) & Task_CollectStack [Task_CollectStackSize],
Task_CollectPriority);
19. while (1) { /* Start of the while loop*/
20. /* Suspend with no resumption later the First task, as it must run once only for initiation of timer ticks
and for creating the tasks that the scheduler controls by preemption. */
OSTaskSuspend (FirstTask_Priority); /*Suspend First Task and control of the RTOS passes forever to
other tasks, waiting their execution*/
21. } /* End of while loop */
22. } /* End of FirstTask Codes */
/*****
23. /* Function for finding amount value of the coins collected at ports */
Static int portamount (amt) {
int amount;
Switch (amt) {
case 0 : amount = 0; exit ( )
case 1 : amount = 1; exit ( )
case 2 : amount = 2; exit ( )
case 4 : amount = 3; exit ( )
case 8 : amount = 4; exit ( )
case 16 : amount = 5; exit ( )
case 32 : amount = 6; exit ( )
case 64 : amount = 7; exit ( )
case 128 : amount = 8; exit ( )
}
return amount;
}
24. /* The codes for Task_ReadPorts */
/*****
Static void Task_Read Ports (void *taskPointer)
{
25. /* initial declarations */
int amtport = 0; /* value of amount collected at ports */
int *amount1; /* Pointer to Port 1 message for amount */
int *amount2; /* Pointer to Port 2 message for amount */
int *amount5; /* Pointer to Port 5 message for amount */
26. while (1) { /* Start of while loop */
27. OSSemPend (SemFlag1, 0, *SemErrPointer); /* wait for semaphore from port ISRs */
28. while (Stimeout == false || amtport < cost){ OSSemAccept (Stimeout, 0, *SemErrPointer);
29. *amount1 = OSMboxAccept (MboxPtr1Msg, 0, ErrPointer M1);

```

```

*amount2 = OSMboxAccept (MboxPtr2Msg, 0, ErrPointer M2);
*amount5 = OSMboxAccept (MboxPtr5Msg, 0, ErrPointer M5)
30. amtport = portamount (&amount1) + 2 * portamount (& amount2) + 5 * _portamount (& amount5)
} /* wait till amount at ports > = cost */
OSMboxPost (MboxAmount, &amtport); /*post the among value */
amtport = 0; /*set the amount at port = 0 */
OSTaskSuspend (Task_ReadPortsPriority);
/* The lower priority task_collect now runs */
31. /* End of while loop*/
} /* End of the Task_ReadPorts function */
/*****
32. /* Start of Task_Collect codes */
static void Task_Collect (void *taskPointer) {
33. /* Initial Assignments of the variables and pre-infinite loop statements that execute once only*/
int *amount = 0;

34. while (1) /* Start an infinite while-loop /
35. /* Take Mbox amount value */
*amount = OSMboxPend (MboxAmount, 0, MboxErrPointer);
36. if (*amount > = cost) {OSMboxPost (MboxAmount, *amount);
OSMboxPost (MboxStr1Msg, "wait for a moment collect a nice chocolate soon"); /* Send display
message */
OSSemPost (semFlag2, 0 * ErrPointer); /* Event notify to task_deliver */
37. /* codes for mechanical system to collect the coins from Port_1, Port_2 and Port_5 so that ports are
empty for new coins input */

}
else {OSSemPost (SemFlag3, 0, *ErrPointer)}
OSTaskSuspend (Task_CollectPriority); /* Task_Refund */
OSTaskResume (Task_ReadPorts);
}; End of while loop
38. } /* End of the Task_Collect function */
/*****
39. /* The codes for the Task_Deliver */
static void Task_Deliver (void *taskPointer) {
40. /* The initial assignments of the variables and pre-infinite loop statements that execute once only*/

41. while (1) /* Start an infinite while-loop. */
42. /* Wait for flag SemFlag2 from Task_Collect */
OSSemPend (SemFlag2, 0, SemErrPointer);
43. /* Codes for device driver for Port_Deliver for delivering a chocolate into a bowl. */

44. /* Post two mails to waiting Task_Display through two message pointers for first line and second
line of LCD matrix at Port_Display. */

```

```

OSMboxPost (MboxStr2Msg, "Collect the nice chocolate. Thank you, Insert coins for more"); /*
45. /* Let delayed higher priority task err resume. */
46. /* Post semaphore to flag that the chocolate delivery is over. */
OSSemPost (SemFlag2); /* SemFlag3 to task excess refund if any */
OSTaskSuspend (Task_DeliverPriority);
/* This enables low priority take to start */
OSTaskResume (Task_CollectPriority);
}; /* End of while loop*/
47. } /* End of the Task_Deliver function */
/*****
48. /* Start of Task_Refund codes */
static void Task_Refund (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that execute once only*/

49. while (1) /* Start the infinite loop */
OSSemPend (SemFlag3, 0, SemErrPointer);
50. /* Code for the device driver to let Port_Exit release the coins as refund as within twenty cycles of
clock ticks, if child fails to insert the coins of the required amount. */

51. /* Post mail to waiting Task_Display through message pointer for first line and second line of LCD
matrix at Port_Display. */
OSMboxPost (MboxStr3Msg, "For chocolate." "Insert coins again collect Refund"); /*
OSTimeDlyResume (Task_Display);
OSTaskResume (Task_CollectPriority); /* Return to Task_Collect_Priority. */
}; /* End of while loop*/
52. } /* End of the Task_Refund function */
/*****
53. /* Start of Task_ExcessRefund codes */
static void Task_ExcessRefund (void *taskPointer) {
54. /* Initial assignments of the variables assignments and pre-infinite loop statements that execute once
only*/

55. while (1) /* Start the infinite loop */
/* Wait for SemFlag2 from chocolate deliver task */
OSSemPend (SemFlag2, 0, SemErrPointer);
/* Wait for amount pointer in order to refund the excess amount */
*amount = OSMboxPend (MboxAmount, 0, *MboxErrPointer);
*amount = *amount_cost;
/* Reduce the amount by cost of chocolate */
56. /* Reset Amt */
57. /* Code for the device driver to let Port_ExcessRefund release the coins for balance amount from a
channel as per the Amt value now. AVCS thus refund the coins, if child fails to insert the coins of the
required amount. */

```



```

58. /* Post mail to waiting Task_Display through message-pointer for the first line and second line of LCD
matrix at Port_Display. */
OSMboxPost (MboxStr4Msg, "Pl collect the excess Amount inserted Thank you, Visit Again" )/*
59. OSTaskSuspend (Task_Excess Refund Priority);
OSTask_Resume (Task_Deliver_Priority);
    }; /* End of while loop*/
60. }/ * End of the Task_ExcessRefund function */
/*****
61. /* The codes for the Task_Display */
static void Task_CharCheck (void *taskPointer) {
62. /* Declare string variables for the three lines and other initial assignments and pre-infinite loop statements
that execute once only. */
unsigned char [ ] Str1, Str2, Str3, Str4, currentTimeDate; /* A variable to display at the right corner of line
3 of LCD Matrix */

63. /* Start an infinite while-loop. */
while (1) {
    /* Wait for Messages for line 1, line 2 and line 3. */
Str1= OSMboxAccept (MboxStr1Msg, 0, MboxErrPointer);
Str2 = OSMboxAccept (MboxStr1Msg, 0, MboxErrPointer);
Str3 = OSMboxAccept (MboxStr3Msg, 0, MboxErrPointer);
Str4 = OSMboxAccept (MboxStr4Msg, 0, MboxErrPointer);
64. currentTimeDate = OSMboxAccept (MboxTimeDateStrMsg, 0, MboxErrPointer);
displayTimeDate (currentTimeDate );
/* TimeDate display */
if (Str1 != NULL) displayStr1 ( );
if (Str2 != NULL) displayStr2 ( );
if (Str3 != NULL) displayStr3 ( );
if (Str4 != NULL ) displayStr4 ( );
65. /* Device driver Codes for sending the four strings to a byte stream from line 0 character first to last
character line through Port_Display. */

OSTimeDly (200) (SemMKey2);
OSTaskResume (Task_ExcessRefundPriority);
66. }/ * End of While loop
}; /* End codes for the Task_Display */
/*****
67. /* Codes for function displayTimeDate */
static void displayTimeDate (unsigned char [ ] currentTimeDate ) {
68. /* Initial assignments of the variables */
unsigned char *timeDate;

```

```

    } /* End of Codes for displayTimeDate function */
69. /* ISR codes for posting mailbox message to Task_Display */
ISR_TimeDate ( ) {
    /* Codes for creating a message for time and date after each 1000th interrupt from the system
RTC tick. */

70. OSMboxPost (MboxTimeDateStrMsg, timeDate);
71. } /* End of ISR_TimeDate code*/

```

11.2 CASE STUDY OF DIGITAL CAMERA HARDWARE AND SOFTWARE ARCHITECTURE

The digital camera was introduced earlier in Section 1.10.4. A digital camera is an example of SoC. [Section 1.6.] Section 1.10.4 listed the camera functions, hardware and software units. Figures 1.14 showed the hardware and software components in a simple digital camera.

Sections 11.2.1 and 11.2.2 give the design steps of a digital camera. Sections 11.2.3 and 11.2.4 describe hardware and software architecture.

11.2.1 Requirements

Requirements of the digital camera can be understood through a requirement table given in Table 11.3.

The detailed functions inside the camera are as follows:

1. A set of controllers control shutter, flash, (for example, for peripherals, direct memory access, and buses), auto focus and eye-ball image control. GUI consists of the LCD display for graphics, and switches and buttons for inputs at camera. A touchscreen is another alternative for LCD and keypad. The user gives commands for switching on the camera, flash, shutter, adjust brightness, contrast, color, save and transfer. The user commands are in the form of interrupt signals. Each signal generates from a user input from an operated switch or button. When a button for opening the shutter is pressed, a flash lamp glows and a self-timer circuit switches off the lamp automatically.
2. The picture generates light, which falls on the CCD array, which through an ADC transmits the bits for each pixel in each row in the frame, and also for the dark area pixels in each row in a vertical strip for offset correction in CCD signaled light intensities for each row. The strip is in adjacent frame.
3. A picture consists of a number of pixels. The number of pixels used for a picture determines resolution. Each picture consists of a number of horizontal and vertical pixels. For 2592×1944 pixels, there are $2592 \times 1944 = 5038848$ sets of cells. Each set of pixel has three cells, for the red, green and blue components in a pixel. Each cell gets exposed to a picture when the shutter of camera opens on a user command. The camera records the pictures using a charge-coupled devices (CCD) array. The array consists of a large number of CCD cells, three at each pixel.

Table 11.3 Requirements of a Digital Camera

Requirement	Description
Purpose	<ol style="list-style-type: none"> 1. Digital recording and display of pictures. 2. Processing to get the pictures of required brightness, contrast and color. 3. Permanent saving of a picture in a file in a standard format at a flash-memory stick (or card). 4. Transfer files to a computer through a port.
Inputs	<ol style="list-style-type: none"> 1. Intensity and color values for each picture in horizontal and vertical rows of pixels in a picture frame. 2. Intensity and color values for unexposed (dark) areas in each horizontal row of pixels for offset correction in the row. 3. User control inputs.
Signals, events and notifications	User commands given as signals from switches/buttons.
Outputs	<ol style="list-style-type: none"> 1. Encoded file for a picture. 2. Permanent store of the picture at a file on a flash-memory stick. 3. Screen display of picture from the file after decoding. 4. File output to an interfaced computer.
Functions of the system	<ol style="list-style-type: none"> 1. A color LCD dot matrix displays the picture before shooting. This enables manual adjustment of the view of the picture. 2. For shooting, a shutter button is pressed. Then a charge-coupled device (CCD) array placed at the focus generates a byte stream in the output after operations by ADC on analog output of each CCD cell. 3. A file creates after encoding (compression) and pixel co-processing as follows: The byte stream is preprocessed and then encoded in a standard format using a CODEC. 4. The encoded picture file can be saved for permanent record. A memory stick saves the file. 5. The file is used for display of recorded picture using a display processor and can be copied or transferred to a memory stick and to a computer connected through a USB port. 6. The LCD displays a picture file after it is decoded (decompressed) using the CODEC. Texts such as picture-title, shooting date and time, and serial number are also displayed. 7. A USB port is used for transferring and storing pictures on a computer. Alternatively, Bluetooth or IR port can be used for interfacing the computer.
Design metrics	<ol style="list-style-type: none"> 1. <i>Power Dissipation:</i> Battery operation. Battery recharging after 400 pictures (assumed) 2. <i>Resolution:</i> High-resolution pictures with options of 2592×1944 pixels = 5038848 pixels, $2592 \times 1728 = 3.2$ M, $2048 \times 1536 = 3$ M and $1280 \times 960 = 1$ M. 3. <i>Performance:</i> Shooting a 4M pixel still picture in 0.5s. 25 pictures per m (assumed) 4. <i>Process Deadlines:</i> Exposing camera process in a maximum of 0.1s. Flash synchronous with shutter opening and closing. Picture display latency, maximum of 0.5s. 5. <i>User Interfaces:</i> Graphic at LCD or touchscreen display on LCD and commands by the camera user through fingers on touchscreen, switches and buttons. 6. <i>Engineering Cost:</i> US\$ 50000 (assumed). 7. <i>Manufacturing Cost:</i> US\$ 50 (assumed).
Test and validation conditions	<ol style="list-style-type: none"> 1. All user commands must function correctly. 2. All graphic displays and menus should appear as per the program. 3. Each task should be tested with test inputs. 4. Tested for 30 pictures per m.

4. The CCD cells charge up on exposure to light. The charging of each red or green or blue cell of a pixel is according to light intensity and color at that point in the picture. Three analog outputs to the system are generated for each pixel. Each analog output has to be first converted into bits for processing at the next stage [Section 1.3.7]. ADC operations thus take place for recording the picture that focuses on the cells. Each cell analog output is an input to the ADC which creates bytes for processing further. Each ADC byte corresponds to each value of the analog current which is as per the exposed intensity and color of a cell. This operation is called preprocessing.
5. ADC operations also take place for unexposed additional cells for the pixels. These pixels are adjacent to the focused frame in a vertical strip consisting of horizontal rows. These enable measuring the average dark (zero intensity) current value of output for each row of the picture frame. This average is subtracted as the offset dark current for each row of bytes obtained in Step 4. An average is taken after ADC conversions for each cell in the strip and averaging the values of bytes for the row. This subtraction operation for each row of bytes of the picture frame is also a part of preprocessing.
6. Bytes from preprocessing operations in Steps 3 and 4 for each cell for the picture pixels are stored in a file after compression. The processed signals are compressed using a JPEG CODEC and saved in one jpg file for each picture frame. [JPEG stands for Joint Photographic Experts Group]
7. The jpg file is created after JPEG encoding (compression) of bytes from the picture processor. The preprocessed bytestream is compressed using discrete cosine transformations (DCTs). Usually, integer arithmetic is used for saving the processing time at the expense of some inaccuracy in computations, with an advantage of a reduced number of VLSI gates in the preprocessor circuit of CCD bytestream, fast processing (about 6 times) and reduced battery energy (about 6 times). [Frank Vahid and Tony Givargis, *Embedded System—A unified Hardware/Software Introduction*. John Wiley and Sons, Inc. 2002] However, the inaccuracy is not perceptible to human eyes. After the DCTs, quantization is done and then Huffman coding does the compression in JPEG format. Quantization means, for example, division by 2 or 4 or 8 at the expense of reducing stored image resolution in order to create a smaller size file. The jpg file is stored from a memory address in a memory block. A block(s) is reassigned to each file.
8. Pictures and GUIs are shown on a colored LCD dot matrix screen. For display from the file obtained in Step 5 for the picture, the original bytes before encoding are retrieved back for decoding (decompression) operation using CODEC and display co-processor. The decompression operation on the file is by inverse DCTs.
9. The file is used for display through a display processor. A pixel display co-processor is used for displaying the pictures *directly* or after rotate right, rotate-left, move up, move-down, shift left, shift-right, zoom, stretch, change picture to next file and reprocess for display, and change to previous file and reprocess for display.
10. The JPEG file can be copied or transferred to a memory stick using a controller, and to a computer connected through USB port controller. Sony memory stick Micro (M2) has a size of $15 \times 12.5 \times 1.2$ mm and has a flash memory of 2 GB, 160 Mbps data transfer rate [Section 1.3.5].
11. A USB port is used for interfacing a computer for transferring and storing pictures on a computer. A USB controller is used for transfer. Alternatively, a Bluetooth or IR port can be used for interfacing the computer.

11.2.2 Class Diagrams

Digital camera file creation, display and transferring to printer, memory stick and USB port can be modeled by the class diagrams of abstract class `Picture_FileCreation`, `Picture_FileDisplay`, and `Picture_FileTransfer`. Figure 11.8 shows three class diagrams of `Picture_FileCreation`, `Picture_FileDisplay` and `Picture_FileTransfer` for creating a JPEG file, displaying the picture and transferring the file to the memory stick, printer and USB port.

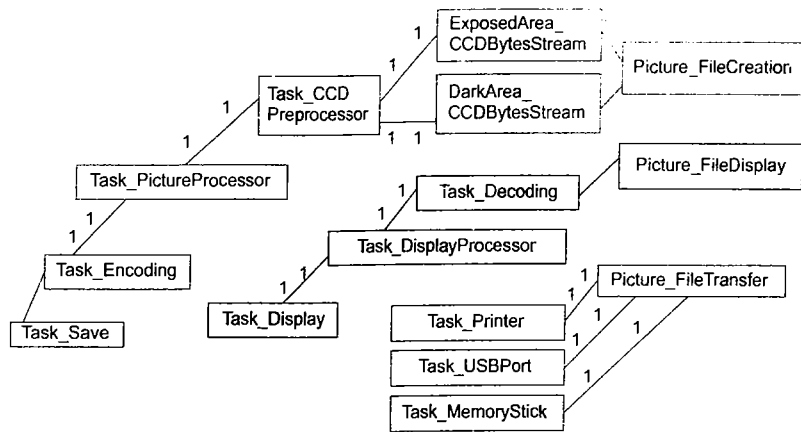


Fig. 11.8 Three class diagrams of Picture_FileCreation, Picture_FileDisplay, Picture_FileTransfer

1. Picture_FileCreation is an abstract class from which an extended class(es) is derived to create a JPEG encoded picture. The task objects are instances of the classes ExposedArea_CCDBytesStream, DarkArea_CCDBytesStream, Task_CCD Preprocessor, Task_PictureProcessor and Task_Encoding.
2. ExposedArea_CCDBytesStream is to create a bytestream from the ADC outputs from the exposed cells in each row of the picture frame.
3. DarkArea_CCDBytesStream is to create a bytestream from the ADC outputs from the unexposed (dark area) cells in each row of the picture frame.
4. Task_CCD Preprocessor creates a stream after the subtraction of the average of bytes for each row of bytes in output from DarkArea_CCDBytesStream from the stream for rows in output of ExposedArea_CCDBytesStream.
5. Task_PictureProcessor creates a stream after processing the Task_CCD Preprocessor for picture brightness, contrast and color adjustment.
6. Task_Encoding creates Huffman encoding for JPEG encoding and creates a filestream for saving onto internal flash or memory stick.
7. Picture_FileDisplay is an abstract class which extends three classes Task_Decoding, Task_DisplayProcessor, Task_Display.
8. Picture_FileTransfer is an abstract class which extends three classes, Task_Printer, Task_USBPort, Task_MemoryStick.
9. Controller_Tasks which extends to the following tasks: (i) Tasks_Initialization for initialization of tasks, (ii) Tasks_Shoot for shooting tasks, (iii) Initialize_Picture_FileCreation to initialize CCD processor (CCDP), (iv) Initialize_Picture_FileDisplay tasks, which initiates display processor (DisplP), (v) initiates processor (MemP), (vi) initiates processor PrintP, (vi) initiates USB port processor (USB_P), (vi) Task_LightLevel for control level control, (vi) Task_flash.

Drawing the class diagram for Controller_Tasks is an exercise for the reader.

11.2.3 Digital Camera Hardware Architecture

The digital camera hardware was described in Section 1.10.4. The camera embeds the following hardware units. It has keys, shutter, lens and charge-coupled device (CCD) array sensors, LCD display unit, a self-timer

lamp for flash, internal memory flash to store OS, embedded software and memory for limited number of picture files, flash memory stick of 2 GB or more for a large number of picture files, a Universal Serial Bus (USB) port (Section 3.10.3) for connecting it to computer and printer. Frank Vahid and Tony Givargis in *Embedded System—A unified Hardware/ Software Introduction*, John Wiley and Sons, Inc. 2002, described the VLSI and implementations of controller, CCD preprocessor with arithmetic implementation of floating point DCT, CCD preprocessor with fixed point DCT and CODEC. The book also described design metrics—performance, energy and cost considerations for each. Figure 11.9 shows a digital camera hardware architecture.

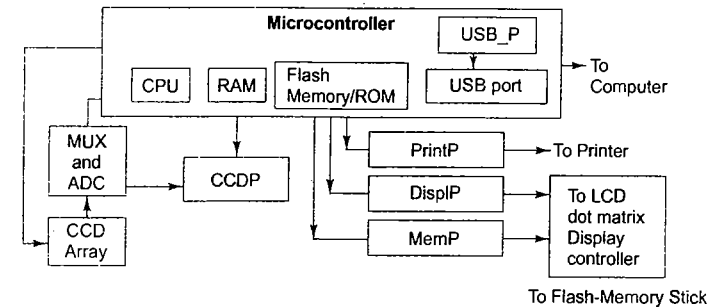


Fig. 11.9 Digital camera hardware architecture

1. A microcontroller executes the Controller_Tasks. The controller tasks are the following: (i) Task_LightLevel control (ii) Task_flash (ii) initialization of tasks, (iii) shooting task, (iv) initiates Picture_FileCreation tasks, which execute on a single purpose CCD processor (CCDP) for the dark current corrections, DCT compression, Huffman encoding, DCT decompression, Huffman decoding and file save. (v) initiates Picture_FileDisplay tasks, which execute on a single purpose display processor (DisplP) for decoded and compressed file image display after the required file bytestream processing for shift or rotate or stretching or zooming or contrast or color and resolution, (v) initiates memory stick save on a notification from Picture_FileTransfer file system object using a single purpose transfer processor (MemP), (vi) initiates printing on a notification from Picture_FileTransfer using a single purpose print processor (PrintP), and (v) initiates USB port controller on a notification from Picture_FileTransfer using a single purpose USB process (USB_P).
2. Multiple processors (CCDP, DSP, Pixel Processor and others)—a DSP does compression using the discrete cosine transformations (DCTs) and decompression by inverse DCTs. After DCTs, it also does the Huffman coding for the JPEG compression. This operation is done using an ASIP (single purpose processor) and is called CODEC (Section 1.2.4).
3. RAM for storing temporary variables and stack
4. ROM/Flash for application codes and RTOS codes for scheduling the tasks
5. Flash memory for storing user preferences, contact data, user address, user date of birth, user identification code, ADC, and Interrupt controller
6. LCD dot matrix display
7. Memory stick
8. Battery

11.2.4 Digital Camera Software Architecture

The camera embeds the following software components. Figure 11.10 shows the software architecture. There are the following layers:

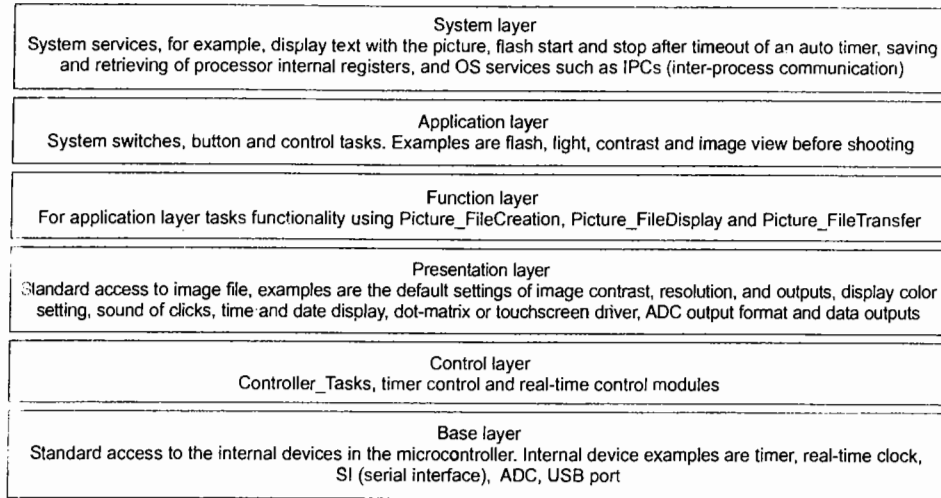


Fig. 11.10 Software layers in software architecture of a camera system

System layer System layer provides system services, for example, display text with the picture, flash start and stop after timeout of an auto timer, saving and retrieving of processor internal registers, and OS services such as IPCs (inter-process communication).

Application layer Application layer is for system switches, button and control tasks. Examples are flash, light, contrast and image view before shooting.

Function layer Function layer is for application layer tasks functionality using Picture_FileCreation, Picture_FileDisplay and Picture_FileTransfer.

Presentation layer Presentation layer is for providing standard access to an image file. Examples are the default settings of image contrast, resolution, outputs, display color setting, sound of clicks, time and date display, dot-matrix or touchscreen driver, ADC output format and data outputs.

Control layer Control layer is for Controller_Tasks, timer control and real time control modules.

Base layer Base layer provides a standard access to the internal devices in the microcontroller. Internal device examples are timer, real-time clock, SI (serial interface), ADC, and USB port.

Figure 11.11 shows a synchronization model for camera tasks using bytestreams as message queues and semaphores.

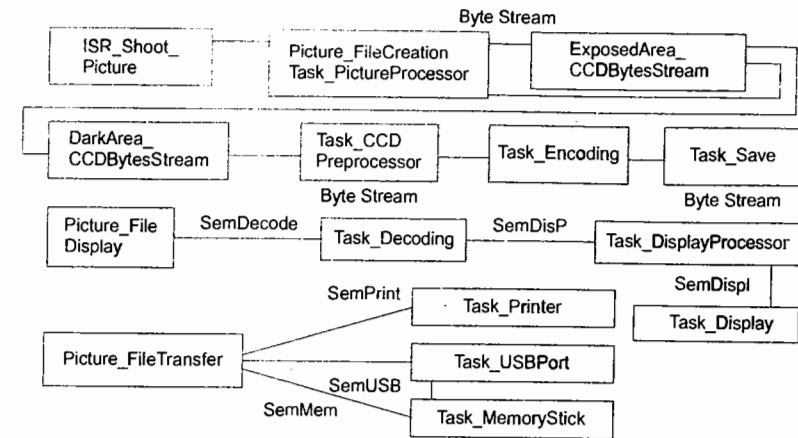


Fig. 11.11 Synchronization model for camera tasks

11.3 CASE STUDY OF CODING FOR SENDING APPLICATION LAYER BYTE STREAMS ON A TCP/IP NETWORK USING RTOS VxWorks

Embedded systems find a large number of applications in data communication and networks. When a system embeds the TCP/IP protocol APIs, it becomes Internet enabled. Many systems are Internet enabled in order that the system communicates to other systems using the Internet.

A communication may be within the same system or between remote systems connected through the network drivers. A communication may be from peer-to-peer or between client and server. Figure 7.11 showed the functions of two sockets communicating through a stack. Socket A is usually a client socket and Socket B is a server socket on a network. Each network socket is identified by a distinct pair of parameters, IP address and port number. Inter-socket communication between two applications is exactly the same, whatever may be the application and OS used. Many RTOSes including VxWorks provides inbuilt APIs for TCP/IP stack and several other network protocols.

The coding given here in Example code 11.2 is already a part of VxWorks network APIs. VxWorks provides the APIs (Application Interfaces) for networking. It has a library, *sockLib* for socket programming. [Section 7.15]. [The reader may refer to VxWorks Network Programmer's Guide when using these APIs and *sockLib*. The guide's source is Wind River Systems (www.windriver.com)].

Since our objective is to learn the use of IPCs in multitasking RTOS through a case study, these APIs and *sockLib* are not used here for creating a TCP stack. The present case study is given in order to learn from the example of TCP and UDP as to how a VxWorks RTOS programming environment can be used to develop APIs for an embedded system for a new networking protocol.

Assume a case of an embedded system in which the RTOS tasks communicate the TCP stacks from an application. An example of the application is HTTP or FTP. The latter communicates to a web server or transfers a file, respectively. A TCP/IP stack is a stack containing the frames communicated on the network using a TCP/IP suite of protocols. A reader may refer to the book *Internet and Web Technologies* by this

author from Tata McGraw-Hill, 7th Reprint, 2007. It describes bit-wise formats of TCP segment, UDP datagram, IP packet, and SLIP and Ethernet frame. The application-layer bytestreams are formatted at the successive layers to obtain the final stack for the network. A stream format is as per protocol specifications for in-between layers. A TCP stack typically has many frames into which a network driver writes the bytes.

Example 11.2 shows the application of RTOS for writing into the stack. We use VxWorks functions described in Section 9.3 for the present case. At another node (end point of the network), the priorities of the tasks that receive the stack to retrieve the bytes put in by the application will obviously be in reverse order. Coding for that is left as an exercise for the reader.

The advantages of multitasking should become obvious by this example. An application may not output the strings continuously. The tasks at the other layers can therefore also process concurrently during the intermediate periods.

The semaphores provide an efficient way of synchronizing the tasks of various priorities. The semaphores are used as a mutex guard for the shared data problems when using the global variables for the I/O streams. The semaphores are also used as event flags. The use of VxWorks or any other tested RTOS simplifies the coding as scheduling and IPC functions are now readily available at the RTOS.

11.3.1 Requirements

Figure 11.12(a) shows the requirements of a sub-system for application, which is transmitting a TCP/IP stack. Figure 11.12(b) shows the scheduling sequences of the tasks during a TCP/IP stack transmission.

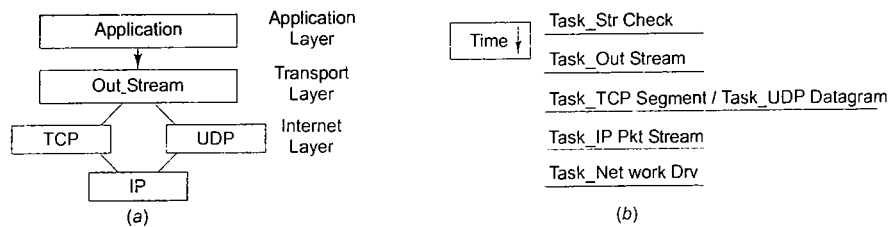


Fig. 11.12 (a) Sub-system for transmitting a TCP/IP stack from an application (b) The tasks and their scheduling sequence during a TCP/IP stack transmission

Requirements for creating a TCP/IP stack can be understood through a requirement table given in Table 11.4.

11.3.2 Class Diagram, Classes and Objects

TCP stack transmission can be modeled by a class diagram of abstract classes TCP_Stack. Figure 11.13 shows class diagram of a TCP_Stack for sending a TCP or UDP packet to a socket. The diagram shows how the classes and objects of a class relate and also the hierarchical associations during the creation of TCP or UDP data stream and packets transmission. The task objects are the processes or threads that are scheduled by the RTOS.

1. Task_TCP is an abstract class from which extended class(es) is derived to create TCP or UDP packet to a socket. The task objects are instances of the classes (i) Task_StrCheck, (ii) Task_OutStream, (iii) Task_TCPSegment or Task_UDPDatagram, (iv) Task_IPPktStream (v) Task_NetworkDrv.
2. Task_StrCheck is to check and get the string.
3. Task_OutStream extends from the two classes Task_StrCheck and Task_TCP.

Table 11.4 Requirements for creating a TCP/IP stack

Requirement	Description
Purpose	To generate a bytestream for sending on network using TCP or UDP protocol at transport layer and IP protocol at network layer
Inputs	1. Bytes from application layer 2. Notification SemTCPFlag and SemUDPFlag for selecting UDP socket or TCP socket, respectively
Signals, events and notifications	After forming the packets at IP layer, SemPktFlag for network driver task
Outputs	1. TCP or UDP bytestream to destination socket
Functions of the system	An HTTP application data is to be sent after encoding headers at transport and network layers. Tasks are scheduled in five sequences (i) Task_StrCheck, (ii) Task_OutStream, (iii) Task_TCPSegment or Task_UDPDatagram, (iv) Task_IPPktStream (v) Task_NetworkDrv
Test and validation conditions	1. A loop back from the destination socket should enable retrieval of application data stream as originally sent 2. Buffer Memory overflow tests

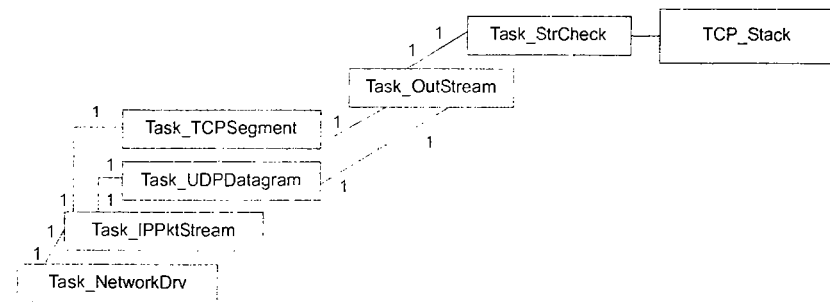


Fig. 11.13 Class diagram of TCP_Stack for sending TCP or UDP packet to a socket

4. Task_TCPSegment creates a stream from TCP segment for IP layer. When datagram is to be sent then Task_UDPDatagram creates a stream using from Task_OutStream output bytes.
5. Task_IPPktStream creates a packet using the stream Task_TCPSegment or Task_UDPDatagram. It depends whether the application layer object posts SemTCPFlag or SemUDPFlag.
6. Task_NetworkDrv outputs the packets on the network.

Class The left-hand side of Figure 11.14 shows an example of class Task_OutStream. It demonstrates how these two classes are shown using UML.

Task_OutStream has the following fields: (i) semFlag and SemMKey are two semaphores for IPC functions, (ii) three integers for the numBytes, (for number of bytes to be transmitted from application layer), task ID and task priority, respectively, and text lines 1, 2 and 3, (iii) OutStreamInputID and applStr static strings for the out stream ID and application data stream.

Task_OutStream has three methods (functions in C)—msgQSend () posts message string into a message queue, TaskDelay () delays the task and enables a low priority task to start, and TaskResume () resumes the delayed task.

Object The right-hand side of Figure 11.14 is an example of an object based on classes Task_OutStream of which that object is an instance and functional entity. An object is shown by a rectangular box with the object identity followed by semicolon and then the class identity. Object Task_OutStreamAppI is an instance of Task_OutStream.

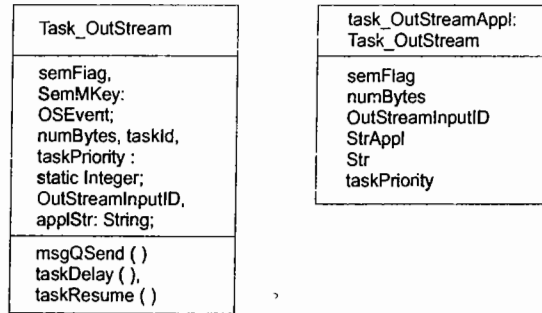


Fig. 11.14 Class Task_OutStream and Object Task_OutStream

State Diagram Figure 11.15 shows a state diagram for Task_Stack. It demonstrates how UML state diagram is shown. A state diagram shows a model of a structure for its start, end, in-between associations through the transitions and shows events labels (or condition) with associated transitions. The state transitions take place between the tasks, Task_GUI, Task_Display, TaskUser_KeypadInput, Task_Communication, Task_ReadPorts, Task_Refund, Task_ExcessRefund, Task_Collect, Task_Deliver and Task_Display. The steps which are used for drawing a state diagram are given in Section 11.3.3.

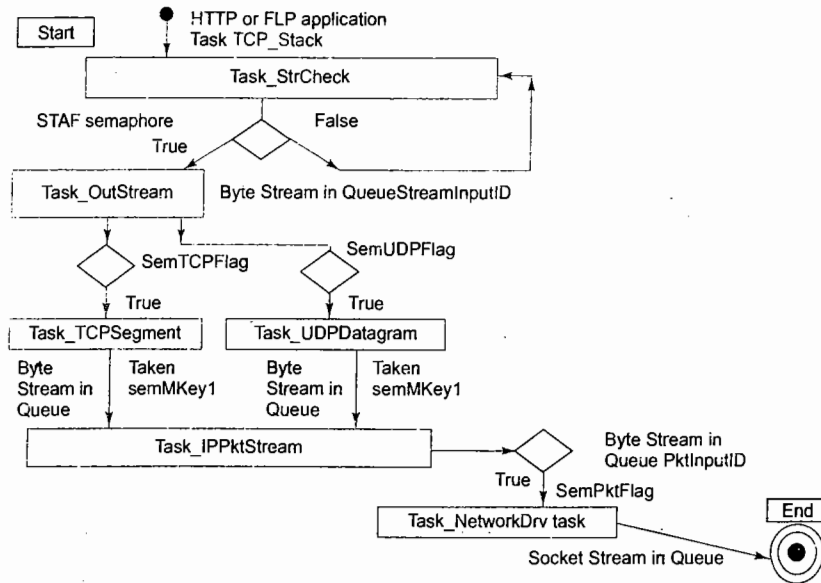


Fig. 11.15 State diagram for TCP_Stack

11.3.3 TCP Stack Hardware and Software Architecture

A TCP stack will run on the same hardware as for the system which is networked with another system. The only additional hardware needed is memory for codes, data and queue for data streams, packets and sockets. A single TCP packet or UDP datagram is of maximum 2^{16} bytes. 2 MB (512×2^{16} bytes) RAM can be taken as additional memory requirement.

Software Architecture The left-hand side of Figure 11.16 shows software architecture for a TCP_Stack. The right-hand side shows software model for a TCP/IP stack. Tasks of TCP_Stack and their priorities are defined as follows:

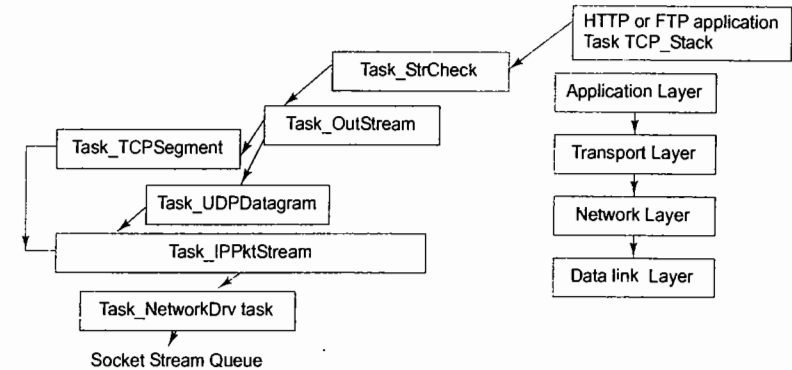


Fig. 11.16 TCP_Stack software architecture and TCP/IP model

Creating a List of Tasks, Functions and IPCs

1. VxWorks user-task priorities are defined above 100 and in layer-wise sequence.
2. Let us set an IPC naming convention. An IPC starting with initial characters as 'Sem' means binary semaphore and 'SemM' means mutex. An IPC for flagging an event will have four characters, 'Flag'. An IPC for resource locking has three characters, 'Key'. A message queue identifier begins with the character 'Q'. A pipe identifier begins with four characters 'pipe'.
3. Either SemTCPFlag or SemUDPFlag is given by the application task. It is as per the protocol to be employed at the transmission control layer. The IPC SemFinishFlag is to be taken by the application or any other task. It flags the intimation that the task of the message strings put into the pipe by the network driver is finished. It is an acknowledgement. Another IPC SemFlag2 is to be taken by OutStream as an acknowledgement that the stream sent to the driver has been successfully put into the pipe and the new stream can be sent in the output.
4. TCP header differs from UDP header. The former facilitates the connection establishment, network flow control and management, and connection termination by proper exchanges of parameters through the network. TCP is thus called a connection-oriented protocol. The latter is a simple datagram message to the receiver. UDP is thus a connection-less protocol. The task priorities in both cases are assigned equal. Any of the flags, which specifies protocol, can be given (posted) by the application.
5. A datagram is an independent (unconnected to the previous or next) message stream of maximum size 2^{16} bytes. UDP conveys only the source and destination port numbers, stream length and checksum to

the 'internet' layer. The socket stream (output of this layer) has the source and destination port numbers as well as IP addresses. A packet from a socket is a message stream with a of maximum size of 2^{16} bytes. Each packet has an inserted IP header. The socket is identified by its IP address and port number.

- The network driver forms the frame as per network-driver configuration. Table 11.5 gives a list of tasks.

Synchronization Model for Multiple Tasks and Their Functions

Figure 11.17 gives a synchronization model to show how the tasks can be synchronized. The top layer is an *application layer* in a TCP/IP network. Steps from the top layer are as follows:

Step A: Let a task, **Task_StrCheck**, check for availability of a string at output from an application. This is the highest priority task during sending to the net. If *check* shows string availability through a status flag semaphore, *STAF*, the task gives (posts) a semaphore to a waiting task, **Task_OutStream**. Let the maximum size of the stream be *maxSizeOutStream*.

Step B: The waiting task unblocks on taking the semaphore. It then reads a string and sends a byte stream into a message queue. Let the *SemFlag2* be the semaphore taken before sending the string from application into the queue for the buffer, *OutStream*. Let semaphore given by *Task_StrCheck* and taken by *Task_OutStream* be *SemFlag1*. Let the second task use the message queue, *OutStream*, for sending the strings from the application to the next waiting task. Let the outstream give bytes to the message queue identified by *QStreamInputID*.

Step C: Next to the application layer, there is a transmission control layer in the network. It is the equivalent of *transport layer* in the OSI model. The task is next in priority. The application task posts two other semaphore flags, besides *STAF*. One flag is to unblock a waiting task, **Task_TCPSegment**. It takes the semaphore and unblocks if this layer protocol is TCP. The other semaphore flag is to unblock another waiting task, **Task_UDPDatagram**. It unblocks when this layer protocol is UDP.

Step D: Appropriate headers must be inserted at the front of the stream at the *transport layer*, and the stream formats into either a TCP segment or UDP datagram, depending on which task unblocks (undergoes to the running state). *Task_TCPSegment* or *Task_UDPDatagram*. The task puts the *blocks*, each of 256 bytes, *blkSize* = 256 bytes, into a queue. This is necessary because if bytes are put into the queue, a time called context-switching time for inter-task through RTOS will be added to the overheads. The overall execution time will increase. Thus interrupt latencies also increase. [Refer to Section 4.6.]

Step E: A TCP segment or UDP datagram is too long compared to a *block*. The block size has to be optimised later, during a simulation run of the codes. A bigger block size lets another task wait till a block is ready for sending. A smaller block size, on the other hand, increases the task-switching overheads and the interrupt latencies. This task has a critical section. Header bytes are inserted into this section at the queue front and no bytes should be inserted by any other task at this stage. A mutex, *SemMKey1*, protects the section by resource

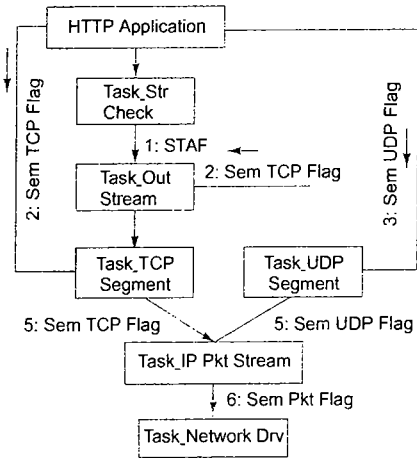


Fig. 11.17 Model for how the tasks are being synchronized

locking. Let semaphores taken by TCP task and taken by UDP task be *SemTCPFlag* and *SemUDPPFlag*, respectively. Let the task give the blocks to message queue, identified by *QStreamInputID*.

Table 11.5 List of Tasks, Functions and IPCs used in Example 11.2

Task Function	Priority	Layer in TCP/IP ⁹⁹	Action	Taken IPC	Posted IPC	Output Byte Stream
Task_Str-Check	120	Application	Get a string from the application	-	SemFlag1	None
Task_Out-Stream	121	Application	Read string and put bytes into an output stream	SemFlag1, SemFlag2	QStream-InputID	Out-Stream
Task_TCP-Segment	122	Transmission Control (Transport)	Insert TCP header to the stream	SemTCPFlag, SemMKey1, QStreamInputID	QStream SemMKey1,	Out-Stream
Task_UDP-Datagram	122	Transmission control (Transport)	Insert UDP header to the stream sent as a datagram	SemUDPPFlag, SemMKey1, QStreamInputID	QStream SemMKey1,	Out-Stream
Task_IPPkt-Stream	123	internet (Network)	Form the packets of 2^{16} bytes	SemMKey1, QStreamInputID	SemMKey1, SemPktFlag, QPktInputID	socket-Stream
Task_Net-workDrv	124	Network Interface (Data-link)	Send the packets as the frames	SemPktFlag, QPktInputID, SemMKey1	SemFinish-Flag, SemFlag2	pipeNet-Stream

⁹⁹ Corresponding layer name in OSI model is given in the bracket.

Step F: Next to the transport layer is the 'internet' layer. [It is a small 'i', not a capital, in internet. We are referring to inter-networking, not the Internet.] It is the equivalent of the network layer in the OSI model. A task, **Task_IPPktStream**, waits and unblocks on availability of a block from *Task_TCPSegment* or *Task_UDPDatagram*. Each packet has a maximum size of 2^{16} bytes. This task is next in priority. **Task_IPPktStream** is for forming an IP packet for transmission on the network through the network driver. This task inserts (writes) an IP header into a socket stream, *SocketStream*, to a network socket. The task inserts the header into the stream after the blocks received from the upper layer stack together in the packet. This task also sends a semaphore flag when the packet is ready for delivery to the network driver. Let the semaphore given by the task and taken later by *Task_NetworkDrv* be *SemPktFlag*. Let the task send the packets to a message queue, identified by *QPktInputID*.

Step G: Next to the internet layer is the 'network interface' layer. It is the equivalent of the data-link layer in the OSI model. Network driver task, **Task_NetworkDrv**, waits for the packet ready flag, *SemPktFlag*. When the task is unblocking, it reads *SocketStream* and writes the frame, *frame* into a pipe, *pipeNetStream*. The pipe stores and transmits the bytes at the frame header, *frameHeader*, at the *SocketStream* data or its fragment and at the trailing bytes, *trailBytes*. The latter are usually for error control functions or frame terminal (end) functions. Let the task give the *frame*, one by one, to a pipe identified by *pipeNetStream*. Let the semaphore given by the task and taken later by the application task be *SemFinishFlag*, when no more bytes are available

for transmission. A semaphore *SemFlag2* is given by the task when a block of byte is ready. This lets the next layer task unblock outStream and initiate the formation of packets whenever the RTOS schedules it next.

SLIP, PPP, Ethernet, and Token ring are examples of interface layer protocols. The *frameHeader* and *trailBytes* are as per the protocol used by the driver. Certain protocols do not write any *frameHeader*, for example SLIP. Certain protocols do not write *trailBytes*.

Task_NetworkDrv opens a configuration file. The configuration file is a virtual file device in the embedded system. [Virtual file device means a file not on the disk but in the memory and using similar open, read, write and close functions as for the files on the disk]. The file points to the configuration information. The function creat ()

[Note: missing 'e' in creat function] and remove () are used to create and remove a file device in VxWorks. VxWorks open, read and write functions are used as shown earlier in Example 9.26. The use of the select () and close () functions for a file is analogous to that for a pipe [Section 9.3.4.12].

Exemplary configuration information in the file can be as follows: We can define, in the case of a serial link, the following parameters:

- (i) Protocol for the link (SLIP or PPP)
- (ii) Host
- (iii) Port
- (iv) Baud rate
- (v) Number of bits per character, for Example, 8
- (vi) Number of stop bits, for Example, 1

A file can describe configuration as follows, in the case of an Ethernet card used as a network driver. The first line format may begin with *net*. This specifies that the line be for specifying the network card addresses. The next three words are then '*Ethernet 3COM 0xXYZ*'. This specifies that the network is Ethernet. The card is made by 3COM. The card address at the system is hexadecimal XYZ. The next line format may begin with IP. This specifies that the IP address should be defined at the line as the next word. The address is of the node that connects to the network. If another connection exists with the same network driver, there may be another line to assign another IP address. [An IP address is of 4 bytes. It is 0xFFFFFFFF for a broadcasting connection to all nodes. It can also be conventionally written as 255.255.255.255].

A programmer designing the codes first prepares a list of tasks, the task priorities, the layers at which the tasks function, actions by the tasks, IPCs for which each task or section waits (takes) before unblocking, IPC which it gives (posts) to let another waiting section or task unblock and the output stream, which the task sends as output. Table 11.5 lists these. Then the coding is done. The following points are to be noted from the table.

11.3.4 Exemplary Coding Steps

The task objects in Figure 11.16 can be implemented as C functions in a VxWorks environment with C as the programming language. VxWorks IPC functions can implement the synchronization model shown in Figure 11.15. Following are the VxWorks coding steps for the task objects listed in Table 11.5.

Example 11.2

```
1. # include "vxWorks.h" /* Include VxWorks functions. */
# include "semLib.h" /* Include semaphore functions library. */
# include "taskLib.h" /* Include multitasking functions library. */
# include "msgQLib.h" /* Include message queue functions library. */
```

```
# include "fioLib.h" /* Include file-device input-output functions library. */
# include "sysLib.c" /* Include system library for system functions. */
pipeDrv ( ); /* Install a pipe driver. */
# include "netDrvConfig.txt" /* Include network driver configuration file for frame formatting protocol
(SLIP, PPP, Ethernet) description, card description/make, address at the system, IP address of the nodes
that drive the card for transmitting or receiving from the network. */
# include "prctlHndlers.c" /* Include file for the codes for handling and actions as per the protocols used
for driving streams to the network. */
2. sysClkRateSet (1000); /* Set system clock rate 1000 ticks per second. */
3. /* Initialise the socket parameters and other network parameters initial values*/
/* SourcePort means source port number of the application used. DestnPort means destination
port number. Define a variable string, Str. */
unsigned short SourcePort; unsigned short DestnPort; unsigned char [ ] Str;
unsigned short SourcePort = ;
unsigned short DestnPort = ;
unsigned short SourceIPAddr = ;
unsigned short DestnIPAddr = ;
4. /* Declare data types of Output Byte Streams for arguments in the tasks. */
unsigned char [ ] applStr, OutStream, socketStream, pipeNetStream;
5. /* Declare data types of the maximum sizes of streams from and to the tasks. Declare data type of block
size, blkSize. It is the number of bytes that must be available first before sending an IPC to a buffering
stream. It avoids repeated switching from one task to the next after each byte. */
unsigned int blkSize, strSize, strSize, maxSizeOutStream, maxSizeSocketStream, maxSizepipeNetStream;
6. /* Allocate Default Values to various sizes*/
maxSizeOutStream = 1024 * 1024; /* Let application put 1 MB on the net. */
unsigned int strSize = 1; /* Let default string size from an application be 1 byte. */
blkSize = 256; /* Let default block be 256 bytes. */
maxSizeSocketStream = 64 * 1024; /* Let Socket put 64 kB packet on the net. */
maxSizepipeNetStream = 16 * maxSizeSocketStream; /* Let network driver put 16 packets = 1 MB
maximum number of bytes. */
7. /* Declare all Table 11.5 Task function prototypes. */
void Task_StrCheck (SemID SemFlag1); /*Task check for the string Availability. */
void Task_OutStream (SEM_ID SemFlag1, SEM_ID SemFlag2, MSG_Q_ID QStreamInputID);
void Task_TCPSegment (SEM_ID SemTCPFlag, SEM_ID SemMKey1, MSG_Q_ID QStreamInputID);
void Task_UDPDatagram (SEM_ID SemUDPFlag, SEM_ID SemMKey1, MSG_Q_ID QStreamInputID,
OutStream, SourcePort, DestnPort, maxSizeOutStream, blkSize);
void Task_IPPktStream (SEM_ID SemMKey1, SEM_ID SemPktFlag, MSG_Q_ID QPktInputID);
void Task_NetworkDrv (SEM_ID SemMKey1, SEM_ID SemPktFlag, SEM_ID SemFinishFlag, SEM_ID
SemFlag2, MSG_Q_ID QPktInputID socketStream, pipeNetStream, blkSize, maxSizeSocketStream,
maxSizepipeNetStream, MSG_FRAME aFrame);
8. /* Declare all Table 11.5 Task IDs, Priorities, Options and Stacksize. Let initial ID till spawned be
none. No options and stacksize = 4096 for each of six tasks. */
int Task_StrCheckID = ERROR; int Task_StrCheckPriority = 120; int Task_StrCheckOptions = 0; int
```



```

Task_StrCheckStackSize = 4096;
int Task_OutStreamID = ERROR; int Task_OutStreamPriority = 121; int Task_OutStreamOptions = 0;
int Task_OutStreamStackSize = 4096;
int Task_TCPSegmentID = ERROR; int Task_TCPSegmentPriority = 122;
int Task_TCPSegmentOptions = 0; int Task_TCPSegmentStackSize = 4096;
int Task_UDPDatagramID = ERROR; int Task_UDPDatagramPriority = 122;
int Task_UDPDatagramOptions = 0; int Task_UDPDatagramStackSize = 4096;
int Task_IPPktStreamID = ERROR; int Task_IPPktStreamPriority = 123;
int Task_IPPktStreamOptions = 0; int Task_IPPktStreamStackSize = 4096;
int Task_NetworkDrvID = ERROR; int Task_NetworkDrvPriority = 124;
int Task_NetworkDrvOptions = 0; int Task_NetworkDrvStackSize = 4096;
9. /* Create and Initiate (Spawn) all the six tasks of Table 11.5. */
Task_StrCheckID = taskSpawn (" tTask_StrCheck", Task_StrCheckPriority,
Task_StrCheckOptions,
Task_StrCheckStackSize, void (*Task_StrCheck) (SEM_ID STAF, SEM_ID SemFlag1), 0, 0, 0, 0, 0, 0,
0, 0, 0, 0);
Task_OutStreamID = taskSpawn (" tTask_OutStream", Task_OutStreamPriority, Task_OutStreamOptions,
Task_OutStreamStackSize, void (*Task_OutStream) (SEM_ID SemFlag1, MSG_Q_ID QStreamInputID,
applStr, OutStream, maxSizeOutStream, blkSize), 0, 0, 0, 0, 0, 0, 0, 0, 0);
Task_TCPSegmentID = taskSpawn (" tTask_TCPSegment", Task_TCPSegmentPriority,
Task_TCPSegmentOptions, Task_TCPSegmentStackSize, void (*Task_TCPSegment) (SEM_ID
SemTCPFlag, SEM_ID SemMKey1, MSG_Q_ID QStreamInputID, unsigned char [ ] OutStream, int
maxSizeOutStream, int blkSize, unsigned char [16] txtcpState, unsigned char [16] txtcpOpPdFormat)
unsigned short SourcePort, unsigned DestnPort, unsigned int SequenNum, unsigned int AckNum, unsigned
char * TCPHdrLen, unsigned *TCPHdrFlags, unsigned short *TCPChecksum16, unsigned short window,
unsigned short *UrgPtr, unsigned char [optPdLen] extras)
Task_UDPDatagramID = taskSpawn (" tTask_UDPDatagram", Task_UDPDatagramPriority,
Task_UDPDatagramOptions, Task_UDPDatagramStackSize, void (*Task_UDPDatagram) (SEM_ID
SemUDPFlag, SEM_ID SemMKey1, MSG_Q_ID QStreamInputID, unsigned char [ ] OutStream, int
maxSizeOutStream, int blkSize, SourcePort, DestnPort, 0, 0, 0, 0, 0, 0, 0, 0);
Task_IPPktStreamID = taskSpawn ("tTask_IPPktStream", Task_IPPktStreamPriority,
Task_IPPktStreamOptions, Task_IPPktStreamStackSize, void (*Task_IPPktStream) (SEM_ID SemPktFlag,
MSG_Q_ID QPktInputID, unsigned char [ ] OutStream, int maxSizeOutStream, blkSize, unsigned char
[ ] SocketStream, maxSizeSocketStream), 0, 0, 0, 0, 0, 0, 0, 0, 0);
Task_NetworkDrvID = taskSpawn ("tTask_NetworkDrv", Task_NetworkDrvPriority, Task_
NetworkDrvOptions, Task_NetworkDrvStackSize, void (* Task_NetworkDrv) (SEM_ID SemMKey1,
SEM_ID SemPktFlag, SEM_ID SemFinishFlag, SEM_ID SemFlag2, MSG_Q_ID QPktInputID socketStream,
unsigned char [ ] pipeNetStream, int blkSize, int maxSizeSocketStream, int maxSizePipeNetStream, unsigned
char [ ] frameHeader, unsigned char [ ] trailBytes), 0, 0, 0, 0, 0, 0, 0, 0, 0);
10. /* Declare IDs and create the binary semaphore flags, keys and message queues. */
SEM_ID SemTCPFlag, SemUDPFlag; /* Declared at the application */
SemTCPFlag = semBCreate (SEM_Q_FIFO, SEM_EMPTY); /* Declared at the application */
SemUDPFlag = semBCreate (SEM_Q_FIFO, SEM_EMPTY); /* Declared at the application */
/* Note: The application posts wither SemUDPFlag or SemTCPFlag to let one of the two tasks run.

```

```

[Task_TCPSegment or Task_UDPDatagram. */
SEM_ID SemFlag1, SemFlag2, SemPktFlag, SemFinishFlag, SemMKey1; /* Declared for the six tasks
listed in Table 11.5. */
11. /* Create the binary semaphores, message queue for a stream of bytes from an application and pass the
options selected to it. */
SemFlag1 = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY); /*Task higher in priority takes it first. */
SemFlag2 = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY); /*Task higher in priority takes it first. */
SemPktFlag = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY); /*Task higher in priority takes it first. */
SemMKey1 = semBCreate (SEM_Q_FIFO, SEM_EMPTY); /*Taken in FIFO */
SemFinishFlag = semBCreate (SEM_Q_FIFO, SEM_EMPTY); /*Taken in FIFO */
MSG_Q_ID QStreamInputID;
void char [ ] msgStream; /* Pointer for the message buffer */
12. /* Create the message queue identity and pass the parameters and chosen options to it. Let maximum
number of messages be 256 kB and the message be of 1 byte each. An IP packet has a maximum 216 bytes.
Assume that a TCP segment stream for transmitting has maximum of 256 kB. Let 64 bytes be additionally
assigned for the headers. Header bytes add at the lower layers (refer column 3 in Table 11.5). */
QStreamInputID = msgQCreate (maxSizeOutStream, strSize, MSG_Q_FIFO | MSG_PRI_NORMAL);
QPktInputID = msgQCreate (maxSizeSocketStream, strSize, MSG_Q_FIFO | MSG_PRI_NORMAL);
13. /* Steps 1 to 3 as per Example 7.21 for creating a pipe. */
.
.
14. /* Create pipe named as pipeNetStream for including overheads and frame messages, each if 1 byte.
Overheads mean header bytes as well as trailing bytes. */
STATUS pipestatus;
pipestatus = pipeDevCreate ("/pipe/pipeNetStream", maxSizePipeNetStream, 1);
mode = 0 = 0x0;
15. /* Other declarations that are needed. */
.
.
16. /* Declare common functions needed in the networking tasks. */
/* Declare the functions to get 32-bit, 16-bit and 8-bit lengths from a stream or string. */
unsigned int getLength32 (unsigned char [ ] Str) {
/* Codes for finding as an unsigned integer the length of a string or stream up to 232 bytes */
.
.
};
unsigned short getLength16 (unsigned char [ ] Str) {
/* Codes for finding, as an unsigned short, the length of a string or stream up to 216 bytes. */
.
.
};
unsigned byte getLength8 (unsigned char [ ] Str) {
/* Codes for finding, as an unsigned byte, the length of a string up to 256 bytes. */
.
.
}

```

```

};
17. /* Declare Codes for adding extra padding in shorter size message or string or stream to fill
0s so that string size now equals block size, blkSize. */
unsigned char [ ] StrAddPadding (unsigned char [ ] Str, unsigned int blkSize) {
.
.
};
18. /* Exemplary codes for finding a function to get 16-bit checksum from a byte
stream or string. */
unsigned short checksum16 (unsigned char [ ] Str) {
/* Codes for finding, as an unsigned integer, the length of a byte stream or string* /
/* StrPtr is pointer to the string or stream. For the while loop, 'i' is a 16-bit integer. Number of carries
generated = numCarry. */
static unsigned short *strPtr, i, numCarry;
static unsigned int sum = 0;
unsigned short num = (unsigned short) getLength32 (unsigned char [ ] Str); /* num is 16-bit number for
total number of bytes in the string. We are typecasting the length data type to 16-bit */
/* A 16-bit Checksum method is as follows: Sum the 16-bit words and first count the carries, which
generate on successive additions. Then add these carries to get the 16-bit checksum. However, we are
having the byte stream. So method is as follows: */
strPtr = (unsigned short) Str; / Type cast address to 16-bit value pointer. */
i = num/2; /* Let us split the calculation into two parts because we are adding the bytes instead of 16-bit
words to get the checksum*/
while (i --) {
sum += *strPtr++; /* add the byte into the sum and then pointer to next byte. */
if (i & 1){sum += *(unsigned char *) strPtr;}; /* Sum the odd byte from next address. */
while ((numCarry = (unsigned short) (sum >>16)) != 0) {sum = (sum & 0xFFFF) + numCarry;}
return ((unsigned short) sum); / Type cast sum to 16-bit. */
};
/* Reader to write 32-bit checksum codes for function checksum32 by himself (or herself). */

.
.
/* Declare a Function for Cycle Redundancy Check */
unsigned int CRC32 (unsigned int crc, unsigned char aByte) /* Codes for finding 32-bit cycle redundancy
check bits as an unsigned integer for the given message frame. */
.
.
};
19. /* On a network, while we are sending the bytes a protocol uses the different data types, such as
unsigned int, unsigned short, unsigned char, etc. Hence, the definitions of the following six functions to get
the lower byte and higher byte from a 16-bit short, data16 and to get four bytes byte0, byte1, byte2 and
byte3 from a 32-bit int, data32 is esse;ntial. */
unsigned char LByte (unsigned short data16) { ... }; unsigned char HByte (unsigned short data16)
{ ... };

```

```

unsigned char byte0 (unsigned int data32) { ... }; unsigned char byte1 (unsigned int data32)
{ ... };
unsigned char byte2 (unsigned int data32) { ... }; unsigned char byte3 (unsigned int data32)
{ ... };
/* ----- */
20. to 27. /* Code for other declaration steps specific to the various networks */
.
.
/* End of the codes for creation of the tasks, semaphores, message queue, pipe tasks, and variables and all
needed function declarations */
/***** */
28. /* Start of Codes for Task_StrCheck*/
void Task_StrCheck (SEM_ID STAF, SEM_ID SemFlag1) {
.
.
29.
while (1) /* Start of while loop. */
/* When character output is generated by the application, the semaphore is given. */
SemTake (STAF, WAIT_FOREVER); /* Wait for Status flag from Application. */
30. /* Codes for the task. *
.
semGive (SemFlag1);
semGive (SemMKey1); /* Release the mutex if any taken before. */
taskDelay (10); /* Delay 10 ms to let lower priority task run. */
}; /* End of while loop. */

.
.
/* End of Task_StrCheck. */
31. /* Start of Codes for Task_OutStream. */
void Task_OutStream (SEM_ID SemFlag1, MSG_Q_ID QStreamInputID, applStr, OutStream,
maxSizeOutStream, blkSize) {
32. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
int numBytes;
.
.
33. while (1) { /* Start an infinite while-loop. We can also use FOREVER in place of while (1). */
34. /* Wait for SemFlag1 state change to SEM_FULL by semGive function for string availability check
task */
semTake (SemFlag1, WAIT_FOREVER);
35. /* Take the key to not let any application other than the present task put the message into the
queue port decipher task that needs SemMKeyID run */
semTake (SemMKey1, WAIT_FOREVER); /* SemMKeyID is now not available and the critical
region starts */

```

```

36. /* Codes for Encrypting applStr if any or illegal character or message check, if any */
.
.
37. /* At the end, send the byte input at OutStream as message to queue, QStreamInputID. Refer to
Section 9.3.4. for understanding the msgQSend function. NO_WAIT if string is more than the block size
256 bytes. Otherwise string is too short to deserve sending on the stream without padding it with extra 0s.
Call a function to add padding bytes. */
numBytes = getLength32 (applStr);
if (numBytes < blkSize) {StrAddPadding (applStr, blkSize); numBytes = blkSize};
/* Now wait if any of the previous bytes of applStr have not been put into the socket streams.
Task_IPPktStream does that. It posts SemFlag2 on successfully sending the applStr into the stream with
two transport and internet layer headers. */
semTake (SemFlag2, WAIT_FOREVER);
msgQSend (QStreamInputID, applStr, numBytes, NO_WAIT, MSG_PRI_NORMAL);;
semGive (SemMKey1); /* Critical Region ends here. */}
38. /* Resume the delayed task, Task_StrCheck as message has been put into the message queue. */
taskDelay (20)
taskResume (Task_StrCheckID);
. /* Other remaining codes for the task. */
.
.
};
39. } /* End of the codes for Task_OutStream. */
40. /* Start of Codes for Task_TCPSegment. */
void Task_TCPSegment (SEM_ID SemTCPFlag, SEM_ID SemMKey1, SEM_ID SemFlag2, MSG_Q_ID
QStreamInputID, OutStream, maxSizeOutStream, blkSize, unsigned char [16] txtcpState, unsigned char
[16] txtcpOpPdFormat) {
41. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
static int numBytes; /* Number of bytes successfully read. Equals error on timeout or message queue ID
not identified. */
static int timeout = 20; /* Let after 20 system clock-ticks 20 ms*/
unsigned char [16] txtcpState;
unsigned char [16] txtcpOpPdFormat;
unsigned char [ ] OptionAndPds (unsigned char [16] txtcpState, unsigned char [16] txtcpOpPdFormat);
unsigned char [optPdLen] extras;
unsigned char [ ] Str; unsigned int SequenNum, unsigned int AckNum, unsigned char * TCPHdrLen, unsigned
* TCPHdrFlags, unsigned short window, unsigned short *TCPChecksum16, unsigned short *UrgPtr,
unsigned char [optPdLen] extras);
static unsigned char [ ] header;
unsigned char [ ] TCPHeader (unsigned short SourcePort, unsigned short DestnPort, unsigned char
[16] txtcpState, unsigned char [16] txtcpOpPdFormat, unsigned char [ ] Str, unsigned int
SequenNum, unsigned int AckNum, unsigned char * TCPHdrLen, unsigned *TCPHdrFlags,
unsigned short window, unsigned short *TCPChecksum16, unsigned short *UrgPtr, unsigned
char [optPdLen] extras);
.
.

```

```

42. while (1) { /* Start task infinite loop. */
/* Take mutex so that till header is inserted into the stream, it is not released to
Task_OutStream. */
semTake (SemTCPFlag, WAIT_FOREVER);
semTake (SemMKey1, WAIT_FOREVER); /* SemMKeyID is now not available.
Wait for entering critical region*/
43. /* Receive the message sent by Task_OutStream */
numBytes = msgQReceive (QStreamInputID, OutStream, maxSizeOutStream, 20);
if (numBytes != ERROR) {
44. /* Code for defining the txtcpState as per the connection status. */
.
.
/* Code for defining the SequenNum as per txtcpState */
.
.
/* Code for defining the AckNum as per txtcpState */
.
.
/* Code for defining the window as per txtcpState */
.
.
/* Code for defining the txtcpOpPdFormat as per txtcpState */
.
.
45. /* Codes for finding the additional integers as options and padding to be put as the header. */
extras = OptionAndPds (txtcpState, txtcpOpPdFormat);
46. /* Codes for retrieving the TCP header bytes */
header = TCPHeader (SourcePort, DestnPort, txtcpState, txtcpOpPdFormat, OutStream, SequenNum,
AckNum, *TCPHdrLen, *TCPHdrFlags, window, *TCPChecksum16, *UrgPtr, extras);
47. /* Send header into the front of the queue */
msgQSend (QStreamInputID, header, getLength32 (header), NO_WAIT, MSG_PRI_URGENT);
48. /* Send data to the back of the queue */
msgQSend (QStreamInputID, applStr, numBytes, NO_WAIT, MSG_PRI_NORMAL);
}; /* End of the message handling codes. */
semGive (SemMKey1); /* Critical region ends here. */}
semGive (SemTCPFlag); /* SemTCPFlag. */
taskDelay (20);
taskResume (Task_OutStream);
}; /* End of while loop. */
49. } /* End of codes for Task_TCPSegment */
50. /* Start of codes for Task_UDPDatagram */
void Task_UDPDatagram (SEM_ID SemUDPFlag, SEM_ID SemMKey1, SEM_ID SemFlag2,
MSG_Q_ID QStreamInputID, OutStream, maxSizeOutStream, blkSize) {
51. /* Initial assignments of the variables and pre-infinite loop statements that execute once
only*/

```

```

static int numBytes; /* Number of bytes successfully read. Equals error on timeout or if
message queue ID not identified. */
static int timeout = 20; /* Let after 20 system clock-ticks 20 ms*/
static unsigned char [8] header;
unsigned char [8] UDPHeader (unsigned short SourcePort, unsigned short
DestnPort, unsigned char [ ] OutStream);
.
52. while (1) { /* Start task infinite loop. */
/* Take mutex so that till header is inserted into the stream; it is not released to Task_OutStream. */
semTake (SemUDPFlag, WAIT_FOREVER);
semTake (SemMKey1, WAIT_FOREVER); /* SemMKeyID is now not available. Wait for entering the
critical region*/
53. /* Receive the message sent by Task_OutStream */
numBytes = msgQReceive (QStreamInputID, OutStream, maxSizeOutStream, 20);
if (numBytes != ERROR) {
54. /* Codes for retrieving the UDP header bytes */
header = UDPHeader (SourcePort, DestnPort, OutStream); 55. /* Send Header to the front of the queue */
msgQSend (QStreamInputID, header, 8, NO_WAIT, MSG_PRI_URGENT);
56. /* Send Data to the back of the queue */
msgQSend (QStreamInputID, applStr, numBytes, NO_WAIT, MSG_PRI_NORMAL);
}; /* End of the message handling codes. */
/* Critical Region ends here. */
57. semGive (SemMKey1);
semGive (SemUDPFlag); /* SemUDPFlag release for use in new application string, applStr. */
58. /* Let lower priority task, Task_IPPktStream start Higher priority one resume. */
taskDelay (20);
taskResume (Task_OutStream);
}; /* End of while loop. */
59. } /* End of codes for Task_UDPDatagram */
60. /* Start of Codes for Task_IPPktStream */
void Task_IPPktStream (SEM_ID SemFlag2, SEM_ID SemPktFlag, MSG_Q_ID QPktInputID, int
maxSizeOutStream, int blkSize, unsigned short SourceAddr, unsigned short DestnAddr, unsigned short
IPverHdrPrioSer, unsigned char [16] txipState, unsigned char [16] txipOpPdFormat, unsigned char
*timeToLive, unsigned char *PrctlField) {
61. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
static int numBytes; /* Number of bytes successfully read. Equals error on timeout or if message queue
ID not identified. */
static int timeout = 20; /* Let after 20 system clock-ticks 20 ms*/
static unsigned char [8] header; unsigned char [ ] Str;
unsigned char [ ] IPHeaderSelPkt (unsigned short SourceAddr, unsigned short DestnAddr,
unsigned short IPverHdrPrioSer, unsigned short *IPPkLen, char * IPVerHdrLen, unsigned
char * IPHdrLen, unsigned char *IPHdrFlags, unsigned short *IPHdrFrag, unsigned short
*IPChecksum16, unsigned short UniqueID, unsigned char *timeToLive, unsigned char

```

```

*PrctlField, unsigned char [ ] Str, unsigned char [ ] OutStream, char [optPdLen] IPExtras,
unsigned char [16] txipState, unsigned char [16] txipOpPdFormat);
62. while (1) { /* Start task infinite loop. */
/* Take mutex so that till header is inserted into the stream; it is not released to
Task_OutStream. */
semTake (SemMKey1, WAIT_FOREVER); /* SemMKeyID is now not available. Wait for entering critical
region*/
63. /* Receive the message sent by Task_OutStream */
numBytes = msgQReceive (QStreamInputID, OutStream, maxSizeOutStream, 20);
if (numBytes != ERROR) {
64. /* Codes for retrieving the IP Packet header bytes */
header = IPHeaderSelPkt (SourceAddr, DestnAddr, IPverHdrPrioSer, *IPPkLen, * IPVerHdrLen, *
IPHdrLen, *IPHdrFlags, *IPHdrFrag, *IPChecksum16, UniqueID, *timeToLive, *PrctlField, Str,
OutStream, IPExtras, txipState, txipOpPdFormat);
65. /* Send Header into the front of the queue */
msgQSend (QPktInputID, header, * IPHdrLen, NO_WAIT, MSG_PRI_URGENT);};
66. /* Send data to the back of the queue */
numBytes = *IPPkLen - * IPHdrLen;
msgQSend (QPktInputID, Str, numBytes, NO_WAIT, MSG_PRI_NORMAL);};
/* Further send header and Str bytes into the queue if more IP packets are to be sent */
.
}; /* End of the message handling codes. */
/* Critical section ends here. */
67. semGive (SemMKey1);
semGive (SemPktFlag); /* SemPktFlag release for use in Network Driver Task. */
68. /* Let lower priority task, Task_IPPktStream start Higher priority one resume. */
taskDelay (20);
taskResume (Task_UDPDatagramID);
taskResume (Task_TCPSegmentID);
}; /* End of while loop. */
69. } /* End of codes for Task_IPPktStream. */
70. /* Start of Codes for Task_NetworkDrv. */
void Task_NetworkDrv (SEM_ID SemMKey1, SEM_ID SemPktFlag, SEM_ID SemFinishFlag, SEM_ID
SemFlag2, MSG_Q_ID QPktInputID socketStream, unsigned char [ ] pipeNetStream, int blkSize, int
maxSizeSocketStream, int maxSizePipeNetStream, unsigned char [ ] frameHeader, unsigned char [ ]
trailBytes) {
71. /* Declare data type for specifying the headers. Let header length of frame be length; type of
network driver be netDrvType; and fragment offset at the stream be frameOffset, which specifies the
index in the byte array from which a frame starts in case the stream is sent in fragments. */
unsigned char [ ] frame, frameHeader, trailBytes, fragment, trailBytes;
unsigned short fragOffset;
unsigned short *Frlength, * HdrLen, * endLen, *FrameHdrFlags, *FrameHdrFragOffset,
*PrctlField;

```

```

unsigned int *FrameCRC32;
unsigned char [ ] Str, unsigned char [ ] SocketStream, unsigned char [ ] FRExtras,
unsigned char [16] txFRState, unsigned char [16] txFROPdFormat, unsigned char
[12] netDrvType
72. /* Declare Network Driver Function. */
void NetHdrFrTr (unsigned char [ ] frame, unsigned char [ ] frameHeader, unsigned char [ ] trailBytes,
unsigned short fragOffset, unsigned char [ ] fragment, unsigned char [ ] trailBytes, unsigned short *FRLength,
unsigned short * HdrLen, unsigned short * endLen, unsigned short *FrameHdrFlags, unsigned short
*FrameHdrFragOffset, unsigned short *FrameCRC32, unsigned short *PrctlField, unsigned char [ ] Str,
unsigned char [ ] SocketStream, unsigned char [ ] FRExtras, unsigned char [16] txFRState, unsigned char
[16] txFROPdFormat, unsigned char [12] netDrvType);
73. /* Integers for number of bytes successfully read, message size and file device, respectively. */
int numBytes, IBytes, fd;
74. /* Let a pointer netDrvConfig define a pointer to a configuration file for a network driver. */
int fd; /* Define an integer for a file device. */
fd = open (netDrvConfig.txt, O_RDWR, 0); /* Refer Step 5 Example 8.26. */
75. /* Code for reading netDrvType, protocol for the link (SLIP or PPP), data link layer format (say
Ethernet), host IP, port, baudrate, card specifications, etc. Use IBytes and function, read as follows. */
IBytes = 12; /* Let the first message bytes read = 12 */
numBytes = read (fd, Str, IBytes);

close (fd);
75. /* Open the pipe for the Network Stream. Refer Step 5 Example 8.26. */
fd = open ("pipe/pipeNetStream", O_RDWR, 0);
76. while (1) {
/* Task reads SocketStream on unblocking and writes the frame, frame into a pipe, pipeNetStream. The
pipe stores and transmits the bytes at the frame header, frameHeader, and frame fragment from the
SocketStream in case frame is of smaller size than the SocketStream and at the end trailing bytes, trailBytes.
The latter are usually for error control functions or frame terminal (end) functions. Let the task give the
frame one after another to a pipe identified by pipeNetStream. This lets the next layer task unblock outStream
and initiate the formation of packets whenever the RTOS schedules it. */
semTake (SemPktFlag, WAIT_FOREVER);
semTake (SemMKey1, WAIT_FOREVER);
77. /* Receive the message sent by Task_IPPktStream */
numBytes = msgQReceive (QStreamInputID, SocketStream, maxSizeOutStream, 20);
if (numBytes != ERROR) {
78. /* Codes for retrieving the frame bytes */
NetHdrFrTr (frame, frameHeader, trailBytes, fragOffset, fragment, trailBytes, *FRLength, * HdrLen,
*endLen, *FrameHdrFlags, *FrameHdrFragOffset, *FrameCRC32, *PrctlField, Str, SocketStream,
FRExtras, txFRState, txFROPdFormat, netDrvType);
/* Write a message, info of IBytes. */
unsigned char [ ] info; /* Let the message be a string of characters. */
int IBytes;
IBytes = 12;

```

```

write (fd, frame, *FRLength);
79. /* Further write into the pipe if more frames are to be sent */

};
semGive (SemMKey1);
semGive (SemFinishFlag); /* Post the semaphore for next waiting task at the application layer. */
semGive (SemFlag2); /* Post the semaphore for waiting task for sending an Out Stream. */
taskResume (Task_IPPktStreamID); /* Resume the previously delayed higher-priority task. */
}; /* End of While-loop */
80. } /* End of codes for Task_NetworkDrv */
/* Start of codes for Task_NetworkDrv. */
/* **** */
A file prctlHandlers.c has the codes for UDP, TCP, IP and network protocols for including and handling
the headers functions, selecting the fragments bytes when forming IP packets and frame bytes when forming
the frames. The codes are designed as follows:
1. /* Declare a Function for returning a UDP header string. DatagramLen means length of UDP datagram,
which transmits to the next layer. Str means the stream or string from the application layer to be sent with
UDP protocol. */
unsigned char [8] UDPHeader (unsigned short SourcePort, unsigned short DestmPort, unsigned char [ ]
Str) {
2. /* Codes for returning UDP Header String. Remember that UDP protocol transmits the integers with big
endian. Refer to paragraph (1) in the instructions for a hardware designer in Section 2.1. Big endian means
the most significant bytes transmit first. */
unsigned char [8] UDPHStr;
unsigned short DatagramLen = getLength16 (Str) + 8; /* Add 8 byte header length also. */
unsigned short UDPChecksum16 = checksum16 (Str);
UDPHStr [0] = HByte (SourcePort); UDPHStr [1] = LByte (SourcePort);
UDPHStr [2] = HByte (DestmPort); UDPHStr [3] = LByte (DestmPort);
UDPHStr [4] = HByte (DatagramLen); UDPHStr [5] = LByte (DatagramLen);
UDPHStr [6] = HByte (UDPChecksum16); UDPHStr [7] = LByte (UDPChecksum16);
return (UDPHStr);
};
/* **** */
3. /* Declarations and codes for the function. TCPHeader for returning a TCP header string. Str means the
stream from the application layer at a node, node1 end to be sent with TCP protocol to other end, node2. */
/* Define the byte position (index) of the present OutStream bytes from the application. Initial value = 0 */
unsigned int SequenNum = 0;
/* Define the total number of bytes already received at an input stream from node2 to this node1. Input
stream is the one that was sent as TCP stacks and received at the node1 from the receiver node2. This is to
let an outstream simultaneously convey to the other end an acknowledgement along with a new sequence
of bytes. OutStream and input stream synchronize and there is controlled flow of bytes between the two
ends, node1 and node2. Initial value = 0. */
unsigned int AckNum = 0;
/* Declare 4-bit and 4-bit unused, reserved for future expansion. TCP headers vary between 5 to 15 unsigned
integers of 32-bit each. */

```

```

unsigned char *TCPHdrLen;
/* Let 16-character string txtcpState specify the state of transmission of the current TCP segment and its
action required for the controlled transmission. Action is to be as desired. It may be to establish a connection,
for termination or management or flow control. Refer to TCP protocol in any standard text for definitions.
*/
unsigned char [16] txtcpState;
/* Declare TCPHdrFlags for 6 bits for flags and 2 bits unused and reserved for future modifications in
protocol. Bit 0 is FIN, bit 1 is SYN, bit 2 is RST, bit 3 is PUSH, bit 4 is ACK and bit 5 is URG. */
unsigned char *TCPHdrFlags;
/* Let another 16-character string txtcpOpPdFormat specify format, which gives the number and meaning
of optionally added integers and padded integers. For example, an optional integer may be to specify an
alternative window of 32 bits in place of 16 given in the 16-bit window field at fourth integer in the TCP
header. Another option integer may specify the TCP maxSizeOutputStream. */
unsigned char [16] txtcpOpPdFormat;
4. /* Start of the codes for returning a short. It specifies the 4-bit TCP Header length field as well as six-
flag fields. These are as per transmission state and transmitting TCP options and padding format, txtcpState
and txtcpOpPdFormat, respectively */
unsigned short TCPHdrLenFlagBits (unsigned char [16] txtcpState, unsigned char [16] txtcpOpPdFormat,
&TCPHdrFlags, &TCPHdrLen) {
Boolean bit 15, bit 14, bit 13, bit 12, bit 11, bit 10, bit 9, bit 8;
5. /* Codes to have four bits, bit 15, bit 14, bit 13, bit 12 in the returned integer as per the TCP header size,
which specifies the total number of unsigned integers in the TCP header. The total is 5 plus the unsigned
integers for options and padding. These are as per txtcpState and txtcpOpPdFormat used by TCP segment
that is transmitting. The bits, bit 11, bit 10, bit 9, bit 8 are reserved. */
/* Bits 0 to 5 are as per six flags. Bits 6 to 11 are reserved. Bits 0 to 7 are at unsigned character pointer
TCPHdrFlags. */
*TCPHdrFlags = *TCPFlags (txtcpState); /* Using the function find bits 0 to 7, */

6. /* Code to find TCPHdrLen from bit 15, bit 14, bit 13, bit 12. */

}; /* End of the codes for returning an integer that specifies TCP Header length and flag
fields. */
7. /* Declare a function TCPFlags to return TCPHdrFlags as per txtcpState. Start of the codes
for finding the byte to represent the TCP flag fields. */
unsigned char *TCPFlags (unsigned char txtcpState) {
Boolean FIN, SYN, RST, PUSH, ACK, URG, bit 6, bit 7;
/* Codes to create as per txtcpState in which TCP segment is transmitting. */

}; /* End of the codes for returning the pointer for a byte for the TCP flags field. */
8. /* Declare other TCP Header fields. */
unsigned short window; /* node1 specifies by window as an advertisement to node2 how many more bytes
at node1 can be buffered in its buffer beyond the ones already buffered and acknowledged by node1 to

```

```

node2. Buffering is an intermediate state in which bytes are still to be sent to an upper application layer by
a TCP stack receiving entity. Node 2 if finds window = 0 or too small a number, then it should not send any
thing to this node1. This 16-bit field then is a request to the other end node not to flood data bytes on the
network unnecessarily. It is then indirectly a request to wait. */
unsigned short *TCPChecksum16; /* Define 16-bit Checksum. */
/* Define a 16-bit Urgent pointer. */
unsigned short Urgent;
/* A 16-bit value that will indicate an urgency to node2 whenever the URG flag is set. It indicates the first
byte of the start of the urgent data from node1. It is a request to the node2 that it should consider the pointed
bytes first in place of attending to the bytes in the buffer and the bytes after this segment header. The
buffered data and beginning data may be ignored and bytes beginning from UrgPtr are requested to be
given urgency. Given or not depends on node2 program task at TCP segment that sends the bytes to its
application layer. */
unsigned short *urgPtr = Urgent;
unsigned short *TCPURGENT (unsigned char txtcpState, &TCPHdrFlags, *urgPtr) {
Boolean URG;
URG = getFlag (&TCPHdrFlags) {
/* Codes to extract the URG bit. It is bit 5 of byte at address of TCPHdrFlags.

}
/* .If URG is true, then set the urgent field and return a 16-bit short. Codes to return a 16-bit short as urgent
pointer as per txtcpState in which TCP segment is transmitting. */
if (URG) {

};
};
unsigned char [ ] extras; unsigned char OptPdLen;
extras = OptionAndPds ( txtcpState, txtcpOpPdFormat);
optPdLen = getLength8 (extras);
*TCPHdrLen = (optPdLen + 20) / 4 ;
unsigned char [ ] OptionAndPds (unsigned char [16] txtcpState, unsigned char [16]
txtcpOpPdFormat) {
9. /* Codes for returning an array of bytes for the 32-bit integers for the options and padding.
These are as per the State and thus TCP header length parameters. */

};
10. /* Codes for returning TCP Header String. Remember that TCP protocol transmits
the integers with big endian. It means that the most significant byte should transmit first. */
/* Note: Instead of using many arguments, a typedef could have been used for the
header data structure. However, this will consume more memory. */
unsigned char [ ] TCPHeader (unsigned short SourcePort, unsigned short DestnPort, unsigned char [16]
txtcpState, unsigned char [16] txtcpOpPdFormat, unsigned char [ ] Str, unsigned int SequenNum, unsigned
int AckNum, unsigned char * TCPHdrLen, unsigned *TCPHdrFlags, unsigned short window, unsigned
short *TCPChecksum16, unsigned short *UrgPtr, unsigned char [optPdLen] extras) {

```

```

int i; unsigned char [ ] TCPHStr; unsigned short lenflag; unsigned char optPdLen;
unsigned short TCPChecksum16 = checksum16 (Str); unsigned short *urg; unsigned char [ ] extras;
*urg = *TCPURGENT (unsigned char txtcpState, &TCPHdrFlags, *urgPtr)
TCPHStr [0] = HByte (SourcePort); TCPHStr [1] = LByte (SourcePort); TCPHStr [2] = HByte (DestnPort);
TCPHStr [3] = LByte (DestnPort); TCPHStr [4] = Byte0 (SequenNum); TCPHStr [5] = Byte1 (SequenNum);
TCPHStr [6] = Byte2 (SequenNum); TCPHStr [7] = Byte3 (SequenNum);
TCPHStr [8] = Byte0 (AckNum); TCPHStr [9] = Byte1 (AckNum); TCPHStr [10] = Byte2 (AckNum);
TCPHStr [11] = Byte3 (AckNum);
*lenflag = * TCPHdlenFlagBits (unsigned char [16] txtcpState, unsigned char [16] txtcpOpPdFormat
&TCPHdrFlags, &TCPHdrLen);
TCPHStr [12] = HByte (lenflag); TCPHStr [13] = LByte (lenflag);
TCPHStr [14] = HByte (window); TCPHStr [15] = Byte3 (window);
TCPHStr [16] = HByte (TCPChecksum16); TCPHStr [17] = LByte (TCPChecksum16);
TCPHStr [18] = *Urg++; TCPHStr [19] = *(unsigned char *) urg);
extras = OptionAndPds (unsigned char [16] txtcpState, unsigned char [16] txtcpOpPdFormat);
optPdLen = getLength8 ( extras);
*TCPHdrLen = (optPdLen + 20) /4;
while (optPdLen > ++i) {TCPHStr [i] = extras [i - 20];}; /* Fill the options and padding bytes. */
unsigned short TCPSEgLen = getLength16 (Str) + (optPdLen + 20); /* Add byte of header length also. */
return (TCPHStr);
};
/* _____ */
/* Declarations and codes for the function. IPHeaderSelPkt for returning an IP header string and the
packet data in socketStream from OutStream. Str means the stream from transmission control layer at a
node1, end to be sent with header as per IP protocol to the other end, node2, through the network
driver, switches, bridges and routers. */
11. /* First let us declare IPVerHdrLen for 4 bits (bit 7, bit 6, bit 5 and bit 4) for the version number.
Presently, IP version 4 is mostly used. So these bits are 0100. Another four bits (bit 3, bit 2, bit 1 and bit 0)
are the numbers of unsigned integers in the header. Header length includes 5 unsigned standard integers
and 0 to 10 integers for options and padding. The latter integers are used for controlling the path and flow
through the routers. Some of these integers may also be used for adding network security. Options may be
recording of the route of packet, time stamp on the packet, source router IP address to be used before
routing through a common source, any flexible source option, security option, etc. Details can be found in
the classic work of D. E. Comer and D. Stevens, Internetworking with TCP/IP Vol.1, Principles, Protocols
and Architecture from Prentice Hall, NJ, 1995. Usually, the options and padding are not present. Then
the four lower bits are 0101. */
unsigned char * IPVerHdrLen;
/* Header Length as per four upper bits in IPVerHdrLen. */
unsigned char *IPHdrLen;
12. /* Let us assume that the options and padding are as per the format specified by a 16-character string,
txtcpOpPdFormat. It represents the number and meanings of optionally added integers and padded integers.
*/
unsigned char [16] txtcpOpPdFormat;
13. /* Let a 16-character string txtcpState specify the protocol of the current IP segment and its action

```

required for the controlled transmission. Protocol can be ICMP, IGMP, OSPF, EGP and BGP. Refer to this author's book *Internet and Web Technologies from Tata McGraw-Hill* for definitions and any standard text for details of these protocols. */

unsigned char [16] txtcpState;

14. /* Start of the codes for returning a short. It specifies 4-bit version field plus 4-bit IP Header length field in * IPVerHdrLen followed by 3-bit specifying precedence of the IP packet on the network plus 5 bits for the type of service. Let the latter 8 bits be at the address pointed by IPPrioServ. Precedence bits, '000' means usual precedence on the Internet. '111' means highest precedence, the one needed for streaming the audio and video on the net service bits for quality of service to be provided in terms of security, speed, delayed or cost to be charged from the sender. The 16-bit integer returned by function, IPVerHdrPrioServBits is thus as per version and IP packet transmission state and transmitting IP options and padding format, txtcpState and txtcpOpPdFormat, respectively. */

unsigned char * IPPrioServ;

unsigned short IPVerHdrPrioServBits (unsigned char [16] txtcpState, unsigned char [16] txtcpOpPdFormat, &IPPrioServ, &IPHdrLen) /

15. /* Returned integer bit 15, bit 14, bit 13, bit 12, bit 11, bit 10, bit 9 and bit 8 are as per 8 bits at IPHdrLen. Three precedence bits, bit 7, bit 6 and bit 5 in the returned integer are as per the IP precedence and service bits bit 4, bit 3, bit 2, bit 1 and bit 0 are as per QOS (quality of service specified in txtcpState. */ Boolean bit 15, bit 14, bit 13, bit 12, bit 11, bit 10, bit 9, bit 8, bit 7, bit 6, bit 5, bit 4, bit 3, bit 2, bit 1, bit 0; /* The codes to find character at IPPrioServ from txtcpState. */

16. /* Code to find IPHdrLen from txtcpOpPdFormat */

); /* End of the codes for returning the integer that specifies IP version, Header length, precedence and service. */

17. /* Two 16-bit short integers for packet length and packet identification by receiver. */

unsigned short IPPktLen; /* Specify the length of IP Packet */

unsigned short UniqueID; /* Specify the UniqueID of the present IP Packet This is put by the router or the transmitter, node1. It uniquely identification for the packet routed. This will help the receiver to reassemble the fragments of this IP Packet at node2 for upward transmission to transport and application layer there. Note: A fragment may be lost on the net in between routers due to some error popping in. This helps in recovering the lost fragment. */

18. /* A stream from OutStream may be bigger than 6536 bytes minus the header bytes in the IP packet. Therefore, it is to be fragmented. Fragmentation is also necessitated when the network driver and other units in-between do not permit even an IP packet of 6536 bytes and need shorter frames. Function, IPFlagFragBits Codes is for returning 3 flag bits and 13 fragment-offset bits. The offset specifies fragment number. Besides precedence and QOS, txtcpState specifies what 3 flag bits and 13 fragment-field bits should be defined by this function. */

unsigned char *IPHdrFlags; unsigned short *IPHdrFrag;

/* Function for finding a stream of bytes actually been put with the packet. */

void selectPktData (unsigned char [] Str, unsigned char OutStream, unsigned char [16] txtcpState, unsigned char [16] txtcpOpPdFormat, unsigned char [] Str) {

```

};
unsigned short IPFlagFragBits (unsigned char [16] txipState, unsigned char [16] txipOpPdFormat,
&IPHdrFlags, &IPHdrFrag). char [ ] OutStream) /
Boolean bit 15, bit 14, bit 13, bit 12, bit 11, bit 10, bit 9, bit 8, bit 7, bit 6, bit 5, bit 4, bit 3, bit 2, bit 1, bit 0;
19. /* Codes to have three bits, bit 15, bit 14 and bit 13 in the returned integer as per the flags. A flag is mbf.
mbf = 0 means that the receiver should wait as there are more fragments to follow this fragment. mbf = 0
for last fragment. It refers to the IP header size, which specifies the total number of unsigned integers in the
IP header. The total is 5 plus the unsigned integers for options and padding. These are as per txipState and
txipOpPdFormat used by IP segment that is transmitting. The bits, bit 11, bit 10, bit 9, bit 8 are reserved. */
/* Bits, bit 12 down to bit 0 are as fragment offset bits and as per already transmitted bytes from network
driver task at node1. */
/* Code to find character IPHdrFlags from bit 15, bit 14, and bit 13. The bits are as per txipState.

```

```

20. /* Code to find 16-bit short integer IPHdrFrag from bit 13 down to bit 0. These bits are as per txipState
specification, OutStreamSize and the bits already transmitted by the network driver. IPHdrFrag = i means
OutStream byte numbering (8)i is being sent with this packet. */

```

```

); /* End of the codes for returning an integer that specifies IP Header flags and fragment-offset
bits. */

```

```

21. /* Declare other header field, 16-bit Checksum, 8-bit time to live and 8-bit, protocol field, source
node1 IP address and destination node 2 IP address. */

```

```

unsigned short *IPChecksum16;
unsigned char *timeToLive; /* Note: It decrements at each router on the way to node2. */
unsigned char *PrctlField; /* Note: PrctlField = 17 for UDP, = 6 for TCP, = 1 for ICMP.

```

```

unsigned short SourceIPAddr;

```

```

unsigned short DestnIPAddr;

```

```

unsigned char [ ] extras; unsigned char OptPdLen;

```

```

IPExtras = OptionAndPds (txipState, txipOpPdFormat);

```

```

optPdLen = getLength8 (IPExtras);

```

```

*IPHdrLen = (optPdLen + 20) / 4;

```

```

unsigned char [ ] IPOptionAndPds (unsigned char [16] txipState, unsigned char [16]
txipOpPdFormat) /

```

```

22. /* Codes for returning an array of bytes for the 32-bit integers for the options and padding. These are
as per the State and thus IP header length parameters.

```

```

23. /* Codes for returning IP Header String. Remember that IP protocol transmits the integers with big
endian. */

```

```

/* Note: Instead of using many arguments, a typedef could have been used for the header data structure.
However, this will consume more memory. */

```

```

unsigned char [ ] IPHeaderSelPkt (unsigned short SourceAddr, unsigned short
short IPVerHdrPrioSer, unsigned short *IPPkLen, char *IPVerHdrLen, unsigned char
char *IPHdrFlags, unsigned short *IPHdrFrag, unsigned short *IPChecksum16, unsigned
unsigned char *timeToLive, unsigned char *PrctlField, unsigned char [ ] Str, unsigned
char [optPdLen] IPExtras, unsigned char [16] txipState, unsigned char [16] txipOpPdFormat);
int i; unsigned char [ ] IPHStr; unsigned short lenflag; unsigned char optPdLen; unsigned
24. /* Code for finding a stream of bytes actually been put with the packet. */
new = IPVerHdrPrioServBits (txipState, txipOpPdFormat, &IPPrrioServ, &IPHdrLen);
IPHStr [0] = HByte (new);
IPHStr [1] = LByte (new);
25. /* Codes for estimating assigning UniqueID to the packet. */

```

```

IPHStr [4] = HByte (UniqueID);

```

```

IPHStr [5] = LByte (UniqueID);

```

```

26. /* Codes for assigning Time to live and Protocol field from txipState. */

```

```

IPHStr [8] = *timeToLive; IPHStr [9] = *PrctlField;

```

```

IPHStr [12] = byte0 (SourceAddr); IPHStr [13] = byte1 (SourceAddr);

```

```

IPHStr [14] = byte2 (SourceAddr); IPHStr [15] = byte3 (SourceAddr);

```

```

IPHStr [16] = byte0 (DestnAddr); IPHStr [17] = byte1 (DestnAddr);

```

```

IPHStr [18] = byte0 (DestnAddr); IPHStr [19] = byte0 (DestnAddr);

```

```

IPExtras = IPOptionAndPds (txipState, txipOpPdFormat); /* Find Options and Padding. */

```

```

optPdLen = getLength8 (IPExtras);

```

```

*IPHdrLen = (optPdLen + 20) / 4;

```

```

while (optPdLen >= ++i) { IPHStr [i] = IPExtras [i - 20]; }; /* Fill the options and padding bytes.
*/

```

```

27. /* Codes for selecting the socket stream data to be sent and then estimating IPPkLen /
selectPktData (Str, OutStream, txipState, txipOpPdFormat, IPExtras);

```

```

unsigned short *IPPkLen = getLength16 (Str) + (optPdLen + 20); /* Add byte of header
length also. */

```

```

IPHStr [2] = HByte (IPPkLen); IPHStr [3] = LByte (IPPkLen);

```

```

new = IPFlagFragBits (txipState, txipOpPdFormat, &IPHdrFlags, &IPHdrFrag, Str)

```

```

IPHStr [6] = HByte (new); IPHStr [7] = LByte (new);

```

```

unsigned short *IPChecksum16 = checksum16 (IPHStr); /* Find checksum of IP header part only. */

```

```

IPHStr [10] = HByte (IPChecksum16); IPHStr [11] = LByte (IPChecksum16);

```

```

return (IPHStr);

```

```

};

```

```

/* _____ */

```

```

/* Declarations and codes for the function, NetHdrFrTr for returning a Frame header string, frameHeader
and the fragment data in frame [ ] array from SocketStream. Str means the stream from node1 at a
node, node1, end to be sent with header as per data link protocol and card protocol to other end, node2,
through the pipe having byte streams from the network driver. */

```



```

void NetHdrFrTr (unsigned char [ ] frame, unsigned char [ ] frameHeader, unsigned char [ ] trailBytes,
unsigned short fragOffset, unsigned char [ ] fragment, unsigned char [ ] trailBytes, unsigned short *FRlength,
unsigned short * HdrLen, unsigned short * endLen, unsigned short *FrameHdrFlags, unsigned short
*FrameHdrFragOffset unsigned short *FrameCRC32, unsigned short *PrctlField, unsigned char [ ] Str,
unsigned char [ ] SocketStream, unsigned char [ ] FRextras, unsigned char [16] txFRState, unsigned char
[16] txFRopPdFormat, unsigned char [12] netDrvType) {
unsigned char [ ] frame, frameHeader, trailBytes, fragment, trailBytes; unsigned short fragOffset;
unsigned short *FRlength, * HdrLen, * endLen, *FrameHdrFlags, *FrameHdrFragOffset
*FrameCRC32;
unsigned int CRC32 (unsigned int crc, unsigned char aByte);
/* Codes as per netDrvType, transmitting frame state txFRState, transmitting frame option
and padding format, txFRopPdFormat create an array of characters for the frameHeader,
for the fragment and for trailBytes. Each is of length, HdrLen, fraglen, endLen, FRlength.
Other fields calculated are short integer for fragOffset and unsigned integer for frame
CRC bits, FRCRC32, etc. */

```

```

/*****

```



Summary

The following is a summary of what we learnt in this chapter.

- Three case studies were explained. These are for automatic chocolate-vending machine, digital camera, and TCP/IP stack systems.
- Software engineering approach is first studying the requirements, then specifications, and finally the UML modeling.
- Drawing hardware architecture and software architecture simplifies design implementation.
- Use of MUCOS and VxWorks RTOSes and use of the IPCs for task synchronization and concurrent processing.
- ACVM system is modeled by three class diagrams of abstract classes ACVM_Devices, ACVM_Output_Ports and ACVM_Tasks.
- MUCOS RTOS functions have been used in an embedded system for an *automatic chocolate vending machine*. After initialization by the first task, seven tasks have been shown to control the various machine functions: (a) System user coins are of three denominations, Rs 1, 2 and 5 (b) Collection of coins at a chest (c) Refund in case of short change (d) Excess refund from a channel in case excess amount inserted into the machine (e) Delivering the chocolate through a port (f) Displaying messages as per the machine status (g) Displaying time and date. It shows how the semaphore as the MUCOS can be used as event flag. It also shows how to use the MUCOS mailboxes and the system RTC.
- Camera tasks are modeled by four class diagrams are divided Picture_FileCreation, Picture_FileDisplay, Picture_FileTransfer and Controller_tasks. We learnt that besides microcontroller and the several ASIPs are required for performance. A microcontroller executes the Controller_Tasks. The controller tasks are the following: (i) Task_LightLevel control (ii) Task_flash (ii) initialization of tasks, (iii) shooting task, (iv) initiates Picture_FileCreation tasks, which execute on a single purpose CCD processor (CCDP) for the dark current corrections, DCT compression, Huffman encoding, DCT decompression, Huffman decoding and file save.

- (v) initiates Picture_FileDisplay tasks, which execute on a single purpose display processor (DispIP) for decoded and compressed file image display after the required file bytestream processing for shift or rotate or stretching or zooming or contrast or color and resolution, (v) initiates memory stick save on notification from Picture_FileTransfer file system object using a single purpose transfer processor (MemP), (vi) initiates printing on a notification from Picture_FileTransfer using a single purpose print processor (PrintP), and (v)) initiates USB port controller on notification from Picture_FileTransfer using a single purpose USB processor (USB_P).
- TCP stack generation is modeled by class Task_TCP, which is an abstract class from which the extended class (es) is derived to create TCP or UDP packet to a socket. The task objects are instances of the classes (i) Task_StrCheck, (ii)Task_OutStream, (iii) Task_TCPSegment or Task_UDPDatagram, (iv) Task_IPPktStream (v) Task_NetworkDrv.
- VxWorks RTOS has been used for embedded system codes for driving a network card after generating the TCP/IP stack. The exemplary codes show a method of code designing for sending the bytestreams on a network. A bytestream first passes through the multiple layers with TCP/IP protocols. A network driver finally sends the stream on a TCP/IP network. How VxWorks schedules the six tasks is shown. (a) A task is for checking the insertion of application strings by the application. (b) A task for creating the application stream for the transmission control layer. (c) Two task codes are given for inserting the header fields using either of two protocols, TCP or UDP, at the transport layer. (d) A task creates IP packets for the network by fragmenting the TCP segment stream. (e) A task drives the network driver and places the bytestream into a pipe (a virtual device). How the tasks handle the multiple data types at the header fields was also shown. This case study gives the reader a thorough understanding of the application and the various functions of the system RTC in VxWorks RTOS. The reader must be able to develop solutions for an embedded networking system using the VxWorks.



Keywords and their Definitions

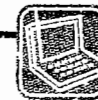
- Application Layer** : A layer consisting of fields attached before placing the message on the network so that the application at the other end understands the request and service.
- Checksum** : A sum representing the number of carries generated by adding the 8-bit numbers or 16-bit numbers or 32-bit numbers.
- CCD** : Charge couple device consisting of a large number of horizontal and vertical cells. Red, green and blue (RGB) cells generate three outputs for a pixel.
- CODEC** : A processing unit for encoding and decoding the bytes or bytestreams, for example, JPEG CODEC for encoding and decoding the bytes or bytestreams from a CCD preprocessor.
- Compression** : To convert the bytes in a file by suitable encoding into a reduced number so that they can be retrieved back by a reverse process of suitable encoding for decompression.
- CRC (Cyclic Redundancy Check)** : A 32-bit or 16-bit integer calculated so that if there is an error in the transmission, then it can be detected by comparing the CRC of the received message. It takes a longer time to calculate but it is better than the checksum.
- Connection-Oriented protocol** : A protocol in which inter-network communication first takes place for connection establishment, then the message flows under a flow control mechanism and then the connection termination occurs after suitable inter-network communication. Transmission Control Protocol, TCP, is an example.

- Connection-less protocol** : A protocol in which inter-network communication takes place without first connection establishment and without a flow control mechanism and without the connection termination by inter-network communication. Usually, in broadcast mode, this protocol is observed. User Datagram Protocol, UDP, is its example.
- Cryptographic Software** : Software for encrypting and deciphering a message or a set of bytestreams. It uses an algorithm for encrypting and another algorithm for decrypting.
- Datagram** : A stream of bytes that is independent of the previous stream. The UDP datagram has a maximum size of 2^{16} bytes.
- DES** : Data Encryption Standard.
- DES3** : A version of DES.
- Electromechanical Device** : A device to operate a mechanical system using electronic signals.
- Fabrication Key** : A key embedded in ROM at the time of card fabrication so that the card gets a unique identity.
- internet Layer** : A layer for inter-networking by TCP/IP protocol. The layer consists of fields attached before placing the message on the network through routers so that the routers send the message as per the source and destination IP address fields in the layer. The fields at this layer are for network-flow controlling mechanism. For example, IP header fields in TCP/IP.
- IP header** : Bytes as per IP protocol placed over the bytes of a packet of TCP segment stream.
- IP packet** : An IP protocol based packet of size upto 2^{16} bytes that transmits after packetization of TCP segment stream-data and after placing IP header bytes.
- LCD dot Matrix** : A set of Liquid Crystal Display consisting of rows and columns, like a matrix. A suitable controller shows the characters or pictures at the matrix.
- Network Driver** : A card that sends and receives the messages physically from the network and facilitates the physical connection for the stream of bytes between the different layers. A driver may use SLIP or PPP protocol for driving on the net after placing the header and trailing bytes as per Ethernet or another protocol. Each driver has a unique?
- Packet** : A set of bytes which includes application, transport and internet layer headers that routes through a set of routers along a path presently available towards a destination.
- Pixel** : Smallest unit of an image, the area of which depends inversely on image resolution. An image consists of horizontal and vertical pixels. At each pixel, in coloured image there are three cells (RGB).
- RGB** : red, green and blue colors. Any colour in the visible range is a combination of these three colors. There can be 256 or 2^{16} colours. It depends on the colour resolution.
- SLIP** : A Serial Line Interface Protocol.
- TCP Header** : A header placed at the transmission control layer as per TCP protocol.
- TCP/IP stack** : A stack of the bytes that transmit from network drive after placing appropriate TCP/IP suite protocol headers and option bytes over that.
- TCP Segment** : A segment of the bytes along with the headers of TCP and application layer protocols that transmits in a single instance through lower layers.
- Transmission Control Layer** : A layer for placing TCP header fields or UDP header fields on a TCP/IP network.
- UDP** : A protocol at the transmission control layer for sending a TCP/IP network datagram.
- UDP Header** : A header consisting of four fields, source and destination port addresses and for length and checksum.



Review Questions

1. Why must a designer first understand the system requirements before designing the codes? Why are the list of tasks and synchronization models required before using RTOS functions.
2. The while loop of the first task contains only task suspend functions. Explain why it should be so?
3. Explain how the task for reading ports synchronizes with the port device driver.
4. How do you use MboxAmount in Example 11.1.
5. What is the role of Task_Display? How do you code for the multiple line messages at an LCD matrix? (MUCOS)
6. Explain use of scmFlags in the ACVM tasks in Example 11.1. (MUCOS)
7. We can use the number of mailbox IPC messages from a task. Explain how this has been effectively used. Why is a message queue not used in place of multiple mailboxes? (MUCOS)
8. Explain the role of each semaphore used. How will the semaphore be used and consequently the codes in Example 11.1 be modified in the following modification to the system? ACVM using a random number generator C function delivers one chocolate free out of an average of eight coin insertions and refunds all the coins to the lucky ones. (MUCOS)
9. Why do we need multiple single purpose processors along with a microcontroller in a digital camera?
10. Why are the DCTs and inverse DCTs done in a digital camera? Why is fixed point arithmetic used instead of floating point arithmetic in digital camera?
11. Describe classes, the objects of which are used in writing codes for saving an image file in a digital camera.
12. How does the range of priority assignments differ from MUCOS assignments in VxWorks?
13. Explain how the semaphore is used in Example 11.2 to direct the use of UDP in place of TCP at the transport layer. (VxWorks)
14. How are size streams bigger than permitted in a packet, and permitted by MTUs (Maximum Transferable Units) at a time handled?



Practice Exercises

15. Explain how the task for reading ports synchronizes with the port device driver.
16. Draw the class diagram of Controller_Tasks for a digital camera.
17. Explain the use of the statement: "fd = open (netDrvConfig.txt, O_RDWR, 0);". (VxWorks)
18. Draw the state diagram of ACVM functions.
19. Draw the state diagram of digital camera functions.
20. Draw the state diagram of TCP stack generation.

Design Examples and Case Studies of Program Modeling and Programming with RTOS-2

12

R

In the previous chapter, ACVM was an interesting example designed to teach embedded system design concepts and RTOS applications. We have seen three case studies in that chapter. These were for ACVM, digital camera and TCP stack systems. We learnt that first we study the requirements, get clarity of the required purpose, inputs, signals, events, notifications, outputs, functions of the system, design metrics, and test and validation conditions, specifications, in terms of users, objects, sequences, activity, state and class diagrams for abstracting the component, behaviour and events and create a description in terms of signals, states and state machine transitions for each task that help us greatly in defining the system architecture for hardware and software. The coding for implementation is done as per the architecture. The system is then finely tested.

e

c

a

l

l

L

E

A

R

N

I

N

G

Like in the previous chapter, we will first start with an interesting example of robot orchestra. We will learn aspects of embedded system design from the four new case studies described in this chapter.

1. Inter-robot communication in a robot orchestra.
2. Embedded systems required in an automobile. Understand design of control system and code implementation using the example of an automotive cruise control (ACC) system in a car. Also learn how the code design can be done using VxWorks after emulating OSEK (Section 10.2) features.
3. Learn card-host communication tasks and smart card OS features. Also learn system design and code implementation for communication of smart card with the host machine (for example, bank ATM machine or payment machine when using a credit card at a shopping mall).
4. Learn interrupts, tasks, and state machine concepts by example of SMS create application in a mobile phone.

12.1 CASE STUDY OF COMMUNICATION BETWEEN ORCHESTRA ROBOTS

O

B

J

E

C

T

I

V

E

S

Qrio was an invention of Sony. *Times* magazine described it in 2003. Qrio means Quest for cuRIOsity. Qrio was a smoother and faster humanoid robot than ever designed before. Qrio weighed 7.3 kg and was 58 cm, and had a one-hour battery. Four Qrios performed a complicated dance routine in 2003.

Qrio was a bipedal robot which could wave hello and recognize voice, its two CCD cameras in the eyes recognized up to 10 different faces and objects, and determined the location of objects in view. Three CCD cameras were used in all, one in each eye and one at the centre. Qrio could converse, sing, walk uphill, dance, kick and play using its fingers. Qrio had seven microphones. Qrios could sing in unison and interact with humans with movements and speech with more than 1000 words. It could learn new words also. Emotions were shown by flashing lights.

In 2006, ten Qrios of the fourth generation performed a dance number. They also conducted an entire orchestra. They gave a unique rendition of Beethoven's 5th symphony (1808), one of the most well known and popular compositions of western classical music, which is often played in the orchestra using several musical instruments and conducted by a conductor. [[http://en.wikipedia.org/wiki/Symphony_No._5_\(Beethoven\)](http://en.wikipedia.org/wiki/Symphony_No._5_(Beethoven))]. The Qrio robot orchestra had novel instruments played by robotic actuators.

Each Qrio had 38 fluid motion flexible joints controlled by separate motors for each with an ASIP in each. Three central system microcontrollers controlled motion, recognized speech and visual images, respectively. Memory was 64 MB with each microcontroller.

Qrio embedded complex algorithms for complex mechanical manoeuvres for balancing during dancing or in the orchestra. Qrios embedded complex orchestral and/or choreographic programs.

Figure 12.1 shows a robot orchestra.

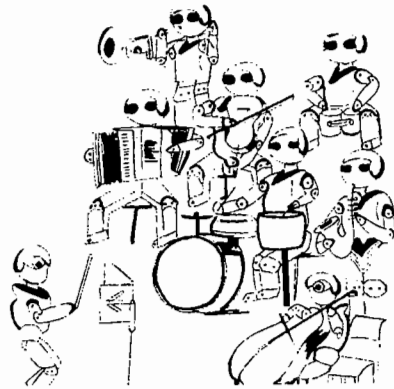


Fig. 12.1 Robot-orchestra

Commands and messages communicated in each Qrio between the microcontrollers, sensors and actuators. A robot was programmed using an *Orchestrator*.

Assume that there are k sensor inputs to the module and q outputs generate to actuators and p outputs to message boxes (also called mailboxes in certain OSes or notifications in certain OSes) in a sequence. *Orchestrator* is software which sequences, synchronizes the inputs from the 1st to the k^{th} sensor and generates messages and outputs for the actuators, display and message boxes at the specified instances and time intervals. Message boxes store the notifications, which initiate the tasks as per the notifications.

Figure 12.2(a) shows embedded software module Orchestrator-1 at a microcontroller 1. Figure 12.2(b) shows commands and messages communication between Orchestrator-x, Orchestrator-y and Orchestrator-z software modules at same or different microcontrollers.

A musical device communicates data to another using a protocol called MIDI (Musical Instrument Digital Interface). Reader is advised to study a tutorial on MIDI. Popular websites are <http://www.borg.com/~jglatt/tutr/miditutr.htm> and <http://www.xtec.es/rtee/eng/tutorial/midi.htm> for this purpose.

Most musical instruments are MIDI compatible and have MIDI IN and MIDI OUT connections, which are optically isolated with the musical-instrument hardware. Three MIDI specifications define (i) what a physical connector is, (ii) what message format is used by connecting devices, controlling them in *real time* and standard for MIDI (musical instruments digital interface) files. Each message consists of a command and corresponding data for that command. Data are sent in byte formats and are always between 0 and 127 and corresponding command bytes in a channel message are from 128 to 255.

A channel (command) message 128–255 (between 0x80 to 0xEF) in MIDI file has musical *note*, *pitch-bend*, *control change*, *program change* and *after-touch* (poly-pressure) messages. There are a maximum of 16 channels in a system. MIDI specifies system messages, manufacturer's system exclusive messages and real time system exclusive messages (between 0xF0 to 0xFF). A real time system message example is as follows: A MIDI Start from conductor is 0xFA command and always begins playback at the very beginning of the song (called MIDI Beat 0). So when a slave player receives a MIDI Start (0xFA), it automatically resets its song position = 0.

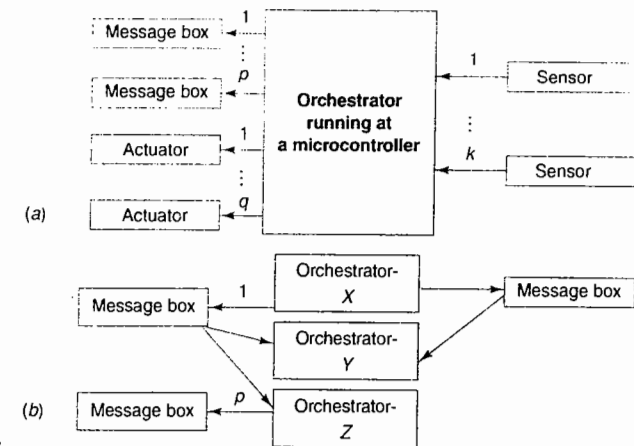


Fig. 12.2 (a) Embedded software module of Orchestrator-1 at a microcontroller 1, with k sensor inputs, q outputs to actuators and p outputs to message boxes (b) Commands and messages communication between Orchestrator-x, Orchestrator-y and Orchestrator-z software modules at same or different microcontrollers

MIDI messages define an event as what musical note is pressed and with what speed it was pressed. This event is input to another MIDI receiver. MIDI receiver plays that note back with the specified speed. Therefore, an entire ensemble of a robot-orchestra can be controlled using MIDI protocol. Also the actuators are synchronized as per the notes and speeds.

Transport of a MIDI file can be over Bluetooth, high Speed Serial, USB or FireWire. For the robot orchestra, communication protocol for MIDI files can be Bluetooth or WLAN 802.11.

Figure 12.3 shows communication between the robot conductor C and four slave-robots (players $P1$ to $P4$) in the robot-orchestra. Each slave robot plays distinct musical instruments and thus plays the notes from distinct track (channel) of MIDI file from the master (conductor).

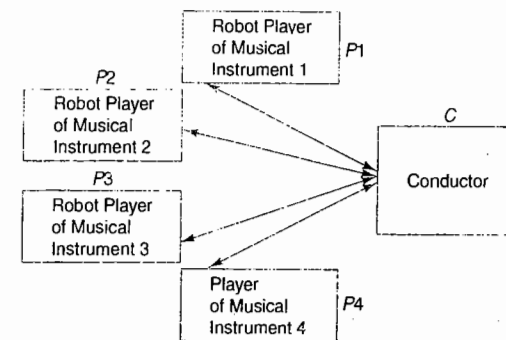


Fig. 12.3 Communication model of a robot- orchestra

12.1.1 Requirements

Requirements of the communicating robots can be understood through a requirement table given in Table 12.1.

Table 12.1 Requirements of Communication system for MIDI files in Robot orchestra

Requirement	Description
Purpose	To communicate selected MIDI file's data over Bluetooth personal-area-network from Robot Conductor communication task to music playing robots
Inputs	1. Orchestra_Choice for selected MIDI file 2. MIDI File
Signals, Events and Notifications	1. Commands (channel-messages) in the file's data
Outputs	1. MIDI File over the communication network 2. Messages to actuators for movements as per specific track messages and data
Functions of the system	A user signals an Orchestra_Choice, say Beethoven's 5 th symphony to start. The conductor C (Figure 12.3) task_MIDI selects the chosen MIDI file and posts the bytes from the files in message queue for task_Piconet_Master. Piconet is a network of Bluetooth devices. Bluetooth devices can form a network known as <i>piconet</i> with the devices within a distance of about 10 m. Bluetooth piconet consists of network of C, P1, P2, P3 and P4. task_PICONET_SlaveP1, task_PICONET_SlaveP2, task_PICONET_SlaveP3 and task_PICONET_SlaveP4 accept the messages from task_Piconet_Master. A task_Piconet_Slave posts the MIDI messages to a task_MIDI_Slave. task_MIDI_Slave posts the messages to a task_Orchestrator, which controls the motor movements of the slave robot motor actuators and musical-note actuators.
Design metrics	1. <i>Power Dissipation</i> : As required by mechanical units, music units and actuators 2. <i>Performance</i> : Near human equivalent of Orchestra play 3. <i>Engineering Cost</i> : US\$ 150000 (assumed) including mechanical actuators 4. <i>Manufacturing Cost</i> : US\$ 50000 (assumed)
Test and validation conditions	1. All MIDI commands must enable all orchestral functions correctly

12.1.2 Class Diagram and Classes

A class diagram [Table 6.3] can be modeled by a class diagram of Task_Conductor. Figure 12.4 shows class diagrams of Task_Conductor and other classes, which are used for posting a MIDI file messages over the piconet to the slaves.

1. Task_Conductor interfaces ISR_Orchestra_Choice and extends to Task_MIDI. Task_Conductor extends to Task_Piconet_Master.
2. Task_SlaveP1, Task_SlaveP2, Task_SlaveP3, and Task_SlaveP4 are abstract classes and extend to classes Task_Piconet_SlaveP1, Task_Piconet_SlaveP2, Task_Piconet_SlaveP3 and Task_Piconet_SlaveP4, respectively.
3. Task_MIDI_SlaveP1, Task_MIDI_SlaveP2, Task_MIDI_SlaveP3 and Task_MIDI_SlaveP4 are classes in slaves P1, P2, P3 and P4, respectively.

4. Task_Orchestrator_SlaveP1, Task_Orchestrator_SlaveP2, Task_Orchestrator_SlaveP3 and Task_Orchestrator_SlaveP4 are classes in slaves P1, P2, P3 and P4, respectively.

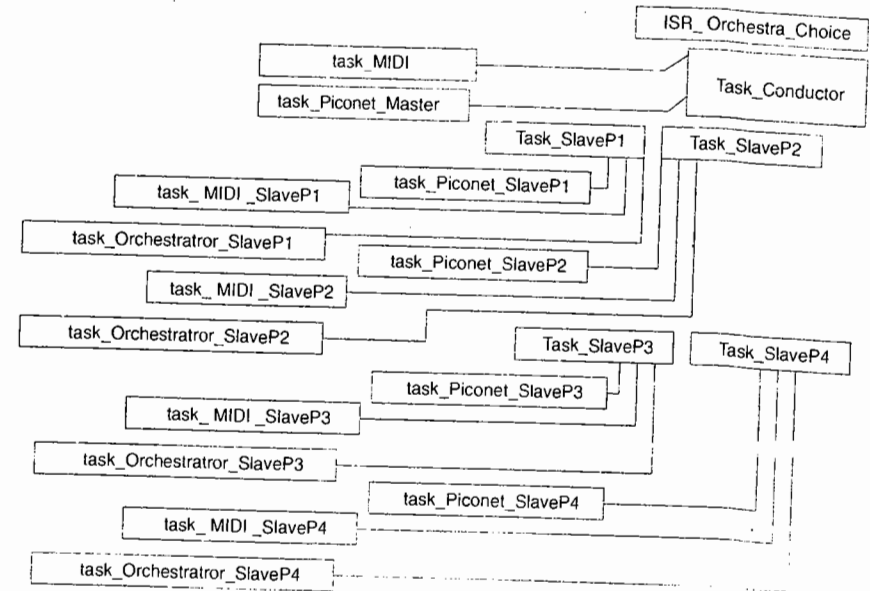


Fig. 12.4 Task_Conductor class diagram of for posting a MIDI file data over the Bluetooth piconet to the slaves

Class A class is a logical group and has fields (attributes) and methods (operations). Figure 12.5 shows an example of fields in class Task_MIDI. The class consists of *fileType* field. It specifies the type of file. It defines Type 0 file if the file contains only one track, which has all of the MIDI messages, which means the entire orchestra performance for that one track. A track can have many musical parts

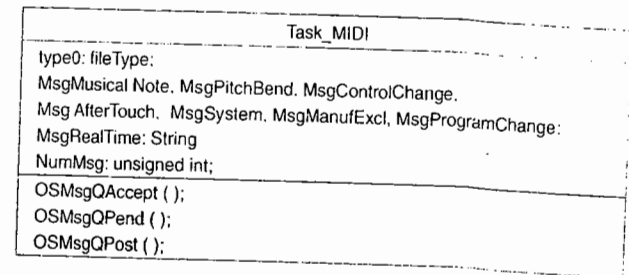


Fig. 12.5 Class MIDI

in different MIDI channels, a channel for each slave robot. Type 1 file specifies each musical part in a separate track for each slave robot. Both Type 0 and 1 store one orchestral performance, for example, Beethoven's 5th symphony. The class other fields are MsgMusicalNote, MsgPitchBend, MsgControlChange, MsgProgramChange, MsgAfterTouch, MsgSystem, MsgManufExclMsg and MsgRealTime. These are string objects for musical note, pitch-bend, control change, program change, after-touch, system messages,

manufacturer's system exclusive messages and real-time system exclusive messages. NumMsg is the number of messages which transfer in one cycle from masters to slave. OSMsgQAccept(), OSMsgQPend() and OSMsgQPost() are the OS methods to accept, wait and post the messages of orchestra choice and MIDI messages.

Objects An object is an instance of a class. Assume a notation representation in which we represent an object ID for process or task by characters starting with lower case and for the corresponding class by characters starting with first character in capital case. For example, the task_MIDI, task_Piconet_Master, task_Piconet_SlaveP1, task_Piconet_SlaveP2, task_Piconet_SlaveP3, task_Piconet_SlaveP4, task_MIDI_SlaveP1, task_MIDI_SlaveP2, task_MIDI_SlaveP3 and task_MIDI_SlaveP4, task_Orchestrator_SlaveP1, task_Orchestrator_SlaveP2, task_Orchestrator_SlaveP3 and task_Orchestrator_SlaveP4 are the objects (processes) of the classes Task_MIDI, Task_Piconet_Master, Task_Piconet_SlaveP1, Task_Piconet_SlaveP2, Task_Piconet_SlaveP3, Task_Piconet_SlaveP4, Task_MIDI_SlaveP1, Task_MIDI_SlaveP2, Task_MIDI_SlaveP3 and Task_MIDI_SlaveP4, Task_Orchestrator_SlaveP1, Task_Orchestrator_SlaveP2, Task_Orchestrator_SlaveP3 and Task_Orchestrator_SlaveP4.

Message queue objects MsgQMidi, MsgQBluetooth, MsgQMidiP1, MsgQBluetoothP2, MsgQMidiP2, MsgQBluetoothP3, MsgQMidiP3, MsgQBluetoothP4, MsgQMidiP4, SigPort1, SigPort2, SigPort3 and SigPort4 are used for posting and accepting messages. MsgQMidi posts NumMsg messages from MIDI file in one cycle to the object task_Piconet_Master. MsgQBluetooth posts Bluetooth stack data in one cycle to Port_Bluetooth. Signal objects SigPorts to ports initiates transfer of Bluetooth stack. Signal object SigPort1, SigPort2, SigPort3 and SigPort4 initiates reception of Bluetooth stack.

12.1.3 State Diagram

Figure 12.6 shows a state diagram for objects from classes of Task_Conductor. A state diagram shows a model of a structure for its start, end, in-between associations through the transitions and shows events-labels (or condition) with associated transitions. A dark rectangular mark within a circle shows the end. [Table 6.3] The state transitions takes place between the tasks, task_MIDI, task_Piconet_Master, task_Piconet_SlaveP1, task_Piconet_SlaveP2, task_Piconet_SlaveP3, task_Piconet_SlaveP4, task_MIDI_SlaveP1, task_MIDI_SlaveP2, task_MIDI_SlaveP3 and task_MIDI_SlaveP4, task_Orchestrator_SlaveP1, task_Orchestrator_SlaveP2, task_Orchestrator_SlaveP3 and task_Orchestrator_SlaveP4.

12.1.4 Robot Orchestra MIDI Communication Hardware and Software Architecture

Hardware architecture specifies the appropriate decomposition of hardware into processors, ASIPs, memory, ports, devices and mechanical and electromechanical units. It also specifies interfacing and mapping of these components. Figure 12.7 shows a block diagram of communication hardware architecture. Following are the specifications:

1. A microcontroller at master and each slave to control Orchestrator for movements. An ASIP for each motor movement.
2. An ASIP at master and each slave for Bluetooth piconet communication between master and slaves.

Software Architecture Software architecture specifies the appropriate decomposition of software into modules, components, appropriate protection strategies, and mapping of software. Software consists of the following at master and each slave.

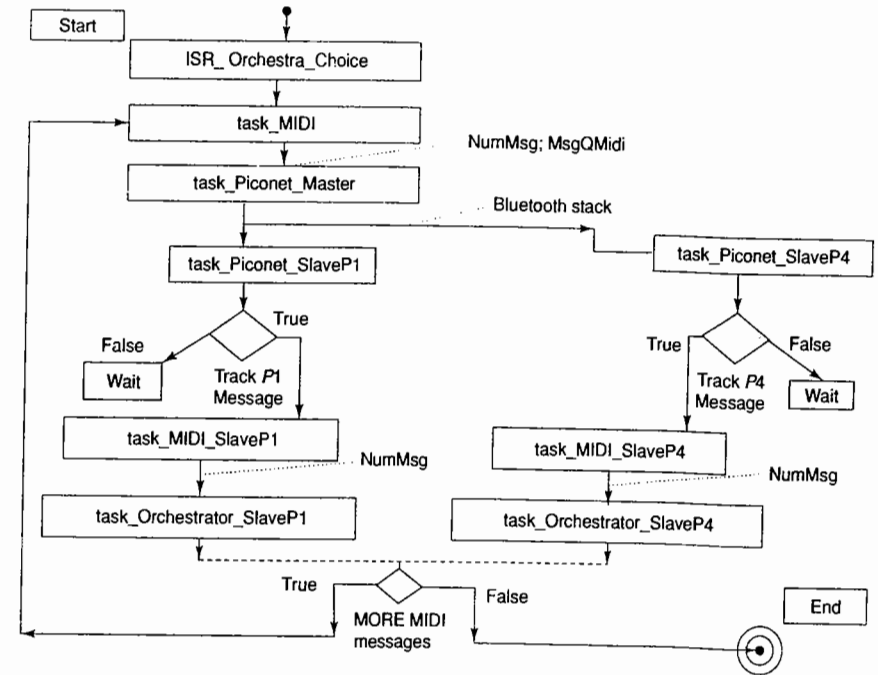


Fig. 12.6 State diagram for task_MIDI

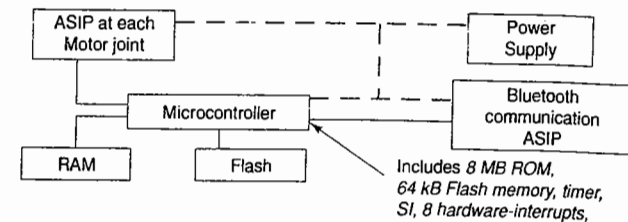


Fig. 12.7 Block diagram of communication-system Microcontroller and ASIPs at master and each slave

1. OS
2. ISRs for initiating action on user inputs and GUI notifications, for example for Orchestra_Choice.
3. The tasks are the objects of the classes shown in Figure 12.4.
4. Orchestrator tasks for master and slaves.
5. OSMsgQAccept, OSMsgQPost and OSMsgQPend are the message queue methods (functions) for system call to the OS. Three methods synchronize the tasks and their concurrent processing such that first

task_MIDI waits for message for Orchestra_Choice from the ISR_Orchestra_Choice, then the task_MIDI posts the bytes in message queue for task_Piconet_Master.

6. task_MIDI waits for NumMsg MIDI messages from MIDI file for chosen orchestra and posts the NumMsg MsgQMidi messages to task_Piconet_Master.
7. task_Piconet_Master discovers the slaves and sets up piconet Bluetooth device network on first initiation. On accepting MsgQMidi messages, it sends protocol stack outputs through the communication port.
8. A task_Piconet_Slave receives Bluetooth stack for the NumMsg messages of MIDI file and sets up network with master as server. The MIDI messages are sorted for the messages to be directed to track P1 or P2 or P3 or P4. For example, trackP1 messages post to task_MIDI_SlaveP1 from task_Piconet_SlaveP1. Then task_MIDI_SlaveP1 posts the messages to actuators and Orchestrator task_OrchetratorP1.
9. Steps similar to Step 8 above repeat for all slaves tracks in robot-orchestra.

12.1.5 Communication Tasks Synchronization Model

Figure 12.6 showed state diagram of synchronizing cycle of different tasks. The communication system has a cycle of actions in a task synchronizing model.

1. A cycle starts from task_MIDI, which receives the *events* (Orchestra_Choice). It posts MIDI messages into message queue MsgQMidi. The MIDI messages are in the MIDI file of Orchestra_Choice stored at master.
2. A task task_Piconet_Master is for discovering the piconet slaves, establishing Bluetooth communication with the slaves P1, P2, P3 and P4 and then accepting MIDI messages from the MsgQMidi. task_Piconet_Master posts Bluetooth stack into message queue MsgQBluetooth and then signals sigPort to Port_Bluetooth.
3. task_Piconet_SlaveP1 executes on a signal SigPortP1 from Bluetooth port Port_BluetoothP1 at the slave. Similarly, actions are at Port_BluetoothP2, Port_BluetoothP3 and PortBluetoothP4, which signal three signals SigPortP2, SigPortP3 and SigPortP4 to the task_Piconet_SlaveP2, task_Piconet_SlaveP3 and task_Piconet_SlaveP4 respectively.
4. task_Piconet_SlaveP1, task_Piconet_SlaveP2, task_Piconet_SlaveP3 and task_Piconet_SlaveP4 (i) accept Bluetooth stack of the master, (ii) select the track messages for the four tracks of four slaves, and post into four message queues the four messages in MIDI format task_MIDI_SlaveP1, task_MIDI_SlaveP2, task_MIDI_SlaveP3 and task_MIDI_SlaveP4 for track P1, P2, P3 and P4, respectively.
5. task_MIDI_SlaveP1, task_MIDI_SlaveP2, task_MIDI_SlaveP3 and task_MIDI_SlaveP4 post to track P1, P2, P3 and P4 Orchestrators the MIDI messages. Orchestrators direct the messages to the actuators of the music track and robot movements.

Figure 12.8 shows a synchronization model for master-slave robots communication tasks using message queues and signals. (Tasks waiting for IPCs 6b, 6c, 6d, 5b, 5c and 5d are not shown in the figure.)

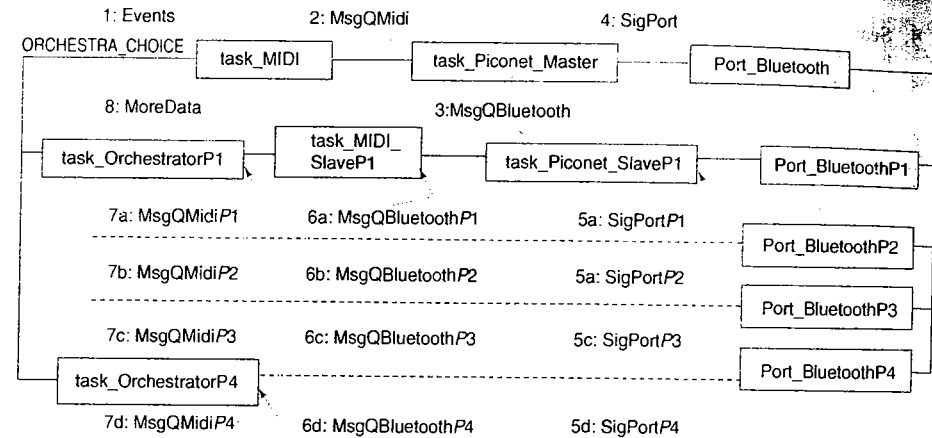


Fig. 12.8 Synchronization model for master-slave robots communication tasks

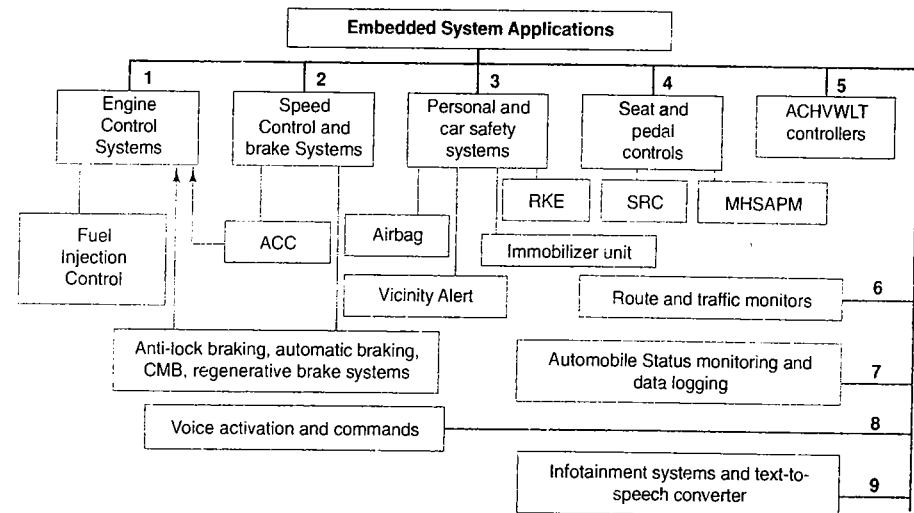


Fig. 12.9 Type of applications of the embedded systems in the car

12.2 EMBEDDED SYSTEMS IN AUTOMOBILE

Present-day automobiles have many embedded systems. Figure 12.9 shows the type of systems in a car. Each system has at least one microprocessor or microcontroller and software. A car may contain the following nine types of systems embedded in it.

1. **Engine control:** Engine control system is by automatic control of fuel injection.
2. **Speed control and brake:** Speed Control and brake systems are automatic cruise control (ACC), antilock braking, automatic braking and regenerative braking.

3. **Safety systems:** Safety systems are for the personal safety and car safety. Personal safety is provided by multiple and variable speed airbags. Airbags are used for cushioning, in particular for automatic rapid inflation of the bag in the case of a collision followed by deflation. Car safety is provided by RKE (remote key entry) and immobilizer unit. RKE system has embedded CPU and software to control door locks. An immobilizer unit immobilizes the car if an unauthorized user attempts to drive the car away. Security systems include alarms, vicinity alert and lane departure alert. A CMB (collision mitigation brake) system is also deployed for safety.

4. **Seat and pedal controls:** Seat and pedal controllers are the seat restraint controller (SRC) and memory heated seat adjustable pedal module (MHSAPM).
 5. **Car environment controls:** ACHVWLT controllers to control the air-conditioning, heater, ventilation, windows, light and temperature.
 6. **Route and traffic monitors:** Route and traffic controls and monitoring are done by GPS (global positioning system) mapping, GPS guided route programming and route planners.
 7. **Automobile status monitoring:** Automobile status monitoring, data loggers and driving assisting device management, steerable head lights, windshield IR camera, windscreen heads up display, night vision assistance.
 8. **System interfaces for commands, voice activation, and interfacing:** System interfaces are soft programmable buttons, voice commands, Bluetooth interfacing for handsfree telephony and wireless personal area connectivity in the automobile, WiFi Internet connectivity, and GSM/CDMA for mobile connectivity.
 9. **Infotainment systems:** Infotainment systems are as following: displayed text-to-speech converters, GPS based car location and surrounding area maps, cached traffic reports for real time traffic monitoring, cassette player, car phones, audio CD player, LCD screen, touch screen panel, satellite or Internet based Radio, VCD/DVD players and hands free telephony using Bluetooth and GSM or CDMA.
- The following functions are the discrete components which are later integrated to a central serving system.
1. **RTC and watchdog timers** for the tasks. [Refer to Sections 1.3.4 and Section 3.7]
 2. **Real-time control** for all electronic, electromechanical and mechanical systems.
 3. **Adaptive Cruise Control (ACC)** to maintain constant speed in cruise mode and to decelerate when a vehicle comes in front at a distance less than safe and to accelerate again to cruise mode in adaptive mode.
 4. **Data Acquisition System (DAS)** for the following:
 - (a) Current time and date display, updating, recoding of instances of malfunction and time intervals of normal functioning. Updating can also be by periodically synchronizing time with time signals from GPS system
 - (b) External temperature
 - (c) Internal temperature
 - (d) Total distance covered for odometer and for ACC
 - (e) Road speed of vehicle in km/hr for the ACC, speedometer and speed warning
 - (f) Engine speed in r.p.m. [revolution per minute]
 - (g) Coolant temperature, instances of its excess above 115° C and constancy within 80 to 90° C
 - (h) Illumination levels at display panels and inside the vehicle
 - (i) Fuel level (Empty, R1, R2, R3, 1/4, 3/8, 1/2, 5/8, 3/4, 7/8 or Full)
 - (j) Oil pressure for starting oil pressure warning system activation for engine speed above 5000 r.p.m. and alarm system activation above 15000 r.p.m.
 - (k) Presently engaged gear information
 - (l) Front-end car distance
 5. **Front panel Switches and display controls**
 6. Port for **alarm** signals. These are sent to display panel, beep and buzzer-sound systems. A warning is issued by displaying pictograms and raising sound-alarms and their appropriate records is saved for diagnostic analysis. [A pictogram is a pre-recorded picture in an image data file. It displays on an LCD matrix. Pictogram displayed is as per the required message]
 7. **Diagnostics computations** for driving time malfunctions statistics and analysis results for fast fault diagnosis by mechanic
 8. **Multi Media Interfaces**
 9. **Control Area Network** Interfaces
 10. **Serial Communication Interface (SCI)**, transmitter and receiver.

Hardware of embedded systems of a car can be designed using ASICs and microcontrollers and DSPs, for example, 80x51, 68HC11/12, PIC, C167, ADSP2106x, 68HC0x, MCOE, Star12, TMS470, Hitachi H8S2xxx series, and ARM 9 based ST9 series. Section 12.3 describes a case study of software implementation aspects of an ACC system in a car.

12.3 CASE STUDY OF AN EMBEDDED SYSTEM FOR AN ADAPTIVE CRUISE CONTROL (ACC) SYSTEM IN A CAR

The choice of case study of an ACC is taken up to understand a control system design and also to understand use of RTOS for code implementation. The system has a number of ports for data input and output. The system uses a control algorithm. Sections 12.3.1 and 12.3.2 give the design steps, for requirements and class diagram of ACC system tasks. Sections 12.3.3 and 12.3.4 describe hardware and software architecture. Sections 12.3.5 and 12.3.6 give the ACC tasks synchronization model and software implementation, respectively.

12.3.1 Requirements

Requirements of the ACC system can be understood through a requirement table given in Table 12.2.

Table 12.2 Requirements of an Adaptive Cruise Control

Requirement	Description
Purpose	1. Controlled cruising of car using adaptive control for the car speed and inter-car distance.
Inputs	<ol style="list-style-type: none"> 1. Present alignment of radar (or laser) beam emitter. 2. Delay interval in reflected pulse with respect to transmitted pulse from emitter. 3. Throttle position from a stepper motor position sensor. 4. Speed from a speedometer. 5. Brake status for brake activities from brake switch and pedal.
Signals, Events and Notifications	<ol style="list-style-type: none"> 1. User commands given as signals from switches/buttons. User control inputs for ACC ON, OFF, Coast, resume, set/accelerate buttons. 2. Brake event. (Brake taping to disable the ACC system, as alternative to "cancel" button at front panel). 3. Safe/Unsafe distance notification.
Outputs	<ol style="list-style-type: none"> 1. Transmitted pulses at regular intervals. 2. Alarms 3. Flashed Messages 4. Range and speed messages for other cars (in case of string stability mode). 5. Brake control 6. Output to pedal system for applying emergency brakes and driver non-intervention for taking charge of cruising from the ACC system.
Control Panel	Control front-end panel has the following: (a) Switch cum display for 'ON', for 'OFF', 'COAST', 'RESUME', and SET/ACCELERATE. The driver activates or deactivates, the ACC system by pressing ON or OFF, respectively. S/he hands over or resumes the ACC system

(Contd)

Requirement	Description
Functions of the system	<p>charge by pressing COAST or RESUME, respectively. S/he sets the cruise speed by SET/ACCELERATE switch. A switch glows either green or red as per the status when the ACC activates. (b) Alarms and message flashing unit issues appropriate alarms and message flashing pictograms.</p> <ol style="list-style-type: none"> 1. Cruise control system takes <i>charge</i> of controlling the throttle position from the driver and enables the cruising of the vehicle at the preset constant speed. A radar system maintains inter-car distance and warns of emergency situations. 2. An alignment circuit aligns the radar emitter. When driving in a hilly area, the emitter alignment is a must. A stepper motor aligns the attachment so that transmitter beam of radar emits with the required beam alignment for the given driving lane and divergence to maintain the in-lane line of sight of the front-end car. task_Align does this function. 3. Transmit pulses emit at periodic intervals and the delay period in receiving its reflection from front-end vehicle is computed. Each pulse can be suitably modulated so that there is noise immunity in the system and beams of multiple sources don't interfere. task_Signal does this function. 4. The measured delay at periodic intervals is multiplied by 1.5×10^8 m/s (half of light velocity) gives the computed distance d (= RangeNow) of front end car at that instance. task_ReadRange does this function. 5. The differences of d with respect to safe d_{safe} and preset distances d_{set} (in case of maintaining string stability) are cyclically estimated. task_RangeRate does this function. 6. The speedometer measures the speed and error in preset speed and measured speed is also periodically estimated. task_Speed does this function. 7. All estimated differences are cyclically sent as input to an adaptive algorithm, which adapts the control parameters and sends the computed output to vacuum actuator of throttle valve. task_Algorithm does computations and task_Throttle initiates the control output functions for this action. Interrupt service routine ISR_ThrottleControl does the critical functions of throttle control. The car decelerates and accelerates as per setting of throttle valve orifice. 8. The brake is controlled when the safe distance is not maintained and warning message is flashed on the screen. task_Brake initiates the critical functions of brake control. Interrupt service routine ISR_BrakeControl performs these functions. 9. When battery power becomes low, the ACC system deactivates after issuing alarm and flashing messages (notifications).
Design metrics	<ol style="list-style-type: none"> 1. <i>Power Source and Dissipation</i>: Car Battery operation. 2. <i>Resolution</i>: 2 m inter-car distance. 3. <i>Performance</i>: Safe distance setting 75 to 200 m. No overshooting of controlled output for the throttle from adaptive algorithm. 4. <i>Process Deadlines</i>: Less than 1 s response on observation of unsafe distance of front-end car. 5. <i>User Interfaces</i>: Graphic at LCD or touch screen display on LCD and commands for ACC cruise mode ON or OFF, resumption of driver control from ACC, setting the preset cruising speed, alarms of different tones for ACC messages to driver. 6. <i>Extensibility</i>: The system is extendable to maintain string stability of multiple cars in a row. 7. <i>Engineering Cost</i>: US\$ 50000 (assumed). 8. <i>Manufacturing Cost</i>: US\$ 600 (assumed).
Test and validation conditions	<ol style="list-style-type: none"> 1. Tested in dense as well light traffic conditions. 2. Tested on plains, hills and valley roads. 3. All user commands must function correctly.

Following are the detailed functions of an ACC system. Cruise control (CC) is also called auto-cruise control or speed control. It automatically controls the car-speed. The driver presets a speed and the system will take over the control of the throttle of the car. In systems a current control in a solenoid controls the throttle position. A stepper motor with worn drive attachment can be used. An adaptive algorithm calculates and sends the control signals to the stepper motor at the vacuum valve actuator. The orifice opening of the vacuum valve controls the throttle valve. The valve is electro-pneumatic. Vacuum creation provides the force via bellows. [A d.c. or stepper motor with a worn drive attachment to the throttle can also be directly used instead of vacuum actuator and bellows].

Generally, the driver holds the vehicle steady during the drive using the accelerator pedal. Cruise control relieves the driver from that duty and the driver hands over the charge to the CC when the road conditions are suitable (not wet or icy, or there are no strong winds or fog) and, if the car is cruising at high speed, when there is no heavy traffic. The driver resumes the charge in these conditions.

Adaptive cruise control (ACC) or autonomous cruise control (ACC) or active cruise control (ACC) or intelligent cruise control (ICC) system is commonly used in aviation electronics and defense aircrafts for cruising since long. Use in the automotives is of recent origin. [Refer to <http://www.ee.surrey.ac.uk/Personal/R.Young/java/html/cruise.html>]. An ACC-system car moves in cruise mode at a preset speed. A radar or laser or UV (ultraviolet) emits signals at regular intervals. These signals are reflected from vehicle in the front. When the reflected signal is received earlier than expected from a minimum safe distance, it notifies the presence of another vehicle to the system. The ACC system then decelerates and the car slows down. It accelerates again to the preset speed when conditions permit. A common method for ACC is only to control speed through throttle position downshifting. New systems also apply the brakes and deploy a CMB system. CMB alerts the driver if front object is less than 100 m distance. For a close object, CMB applies brakes softly and alerts by tugs at the seatbelt. If driver still doesn't react, the system retracts, locks the seatbelt and brakes hard.

A sophisticated ACC system can also maintain *string stability* to control the multiple cars streaming through highways and in case of VIP convoys. [Refer to a paper Chi-Ying Liang and Huei Peng, "Optimal Adaptive Cruise Control with Guaranteed String Stability" Journal of Vehicle System Dynamics, 31, pp. 313-330, 1999].

An adaptive control refers to an algorithm parameters in which *adapt* to the present status of the control inputs in place of a constant set of mathematical parameters in the algorithmic equations. Parameters adapt dynamically. Exemplary parameters, which are adapted continuously, are *proportionality, integration and differentiation constants*.

Figure 12.10 shows *how an adaptive control algorithm can adapt and function*. It calculates the output values for the control signals. For an ACC system, an adjustable-system subunit generates output control signal for throttle valve. The desired preset cruise velocity v_r , desired preset distance d_{set} and safe preset distance d_{safe} are the inputs to index-of-performance measurement-subunit. The measured cruise-velocity v and distance d are also the inputs to index-of-performance computing-subunit. The comparison and decision subunit has inputs of set performance parameters and observed performance parameters. It sends outputs which are inputs to adaptive mechanism subunit. The adaptive mechanism subunit sends outputs, which are inputs to adjustable system. [For details of the control system algorithms, the reader may refer to the standard texts in Control Engineering, namely *Continuous and Discrete Control Systems* by John F. Dorsey, McGraw-Hill International Edition, 2002. *Digital Control and State Variable Method* by Madan Gopal, Tata McGraw-Hill, New Delhi, 1997 and *Modern Control Systems - Analysis and Design* by Walter J. Grantham and Thomas L. Vincent, John Wiley & Sons, New York, 1993].

The port devices and their functions are as follows:

1. Port_Align: It is a stepper motor port. Motor steps up clockwise or anticlockwise on a signal from task_Align. The motor aligns the radar or UVHF transmitting device in the lane of front-end car.

2. Port_ReadRange: It is a front-end car range-measuring port. Time difference ΔT is read on a signal from task_Signal to port device. The port radar emits the signals through the antenna and sensor antenna receives the reflected signal from the front-end car. task_ReadRange reads the Port device circuit for the computations of delay period between the transmission and reception instances. Delay period in s multiplied by 1.5×10^5 m measures the range *rangeNow* (= distance d) in km of the front-end car. task_ReadRange sends message for d to task_RangeRate and all other streaming cars.

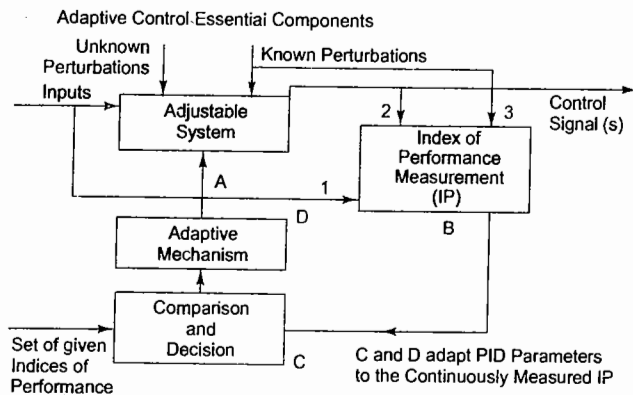


Fig. 12.10 Model for an adaptive control algorithm adaptation and functions

3. Port_Speed: The port control function routine enables free running counter overflow interrupt. On receiving a signal from task_Speed, the port device reads a free running counter *count0* on first one-bit input from the wheel and also sets a parameter $N_{rotation} = 0$. The counts are saved in a memory buffer allotted for the port data. Each successive wheel rotation causes port to note the *countN* at this instance, where *countN* is the count on N^{th} rotation. Also $N_{rotation}$ increments by 1 each time *countN* saves. The port control function routine disables when free running counter overflows. After the counter overflows, it finds the difference of the last value of *countN* in the buffer and *count0*. Current speed $v = speedNow = (N_{rotation} \times \text{wheel circumference in m}) / [T_{clock} \times (countN - count0) \times 1000]$ km/s. task_Speed sends message to task_RangeRate and the for display controller at the speedometer, for *speedNow*. task_RangeRate then sends messages for the $(rangeNow - d_{set})$ and $(speedNow - v_{set})$ to task_Algorithm.
4. Port_Brake: Port device applies the brakes or emergency brakes on an interrupt signal. The service routine ISR_BrakeControl disables the interrupts at the beginning and enables on exiting the critical section. It applies the brakes and signals this information to all the other streaming cars also.

Figure 12.11 shows block diagram of units of ACC system. Figure 12.12 shows ACC system cycle of actions and synchronizing cycle of different units.

12.3.2 Class Diagrams

Table 12.2 lists the required functions and different tasks. A cycle of actions and synchronization of units showed in Figure 12.12 leads us to a model for the system tasks. ACC system measurements of front end car range, distance and error estimations and adaptive control can be modeled by two class diagrams of abstract classes, Task_ACC and Task_Control. Figure 12.13 shows two class diagrams of Task_ACC and Task_Control and also other extended classes from these classes.

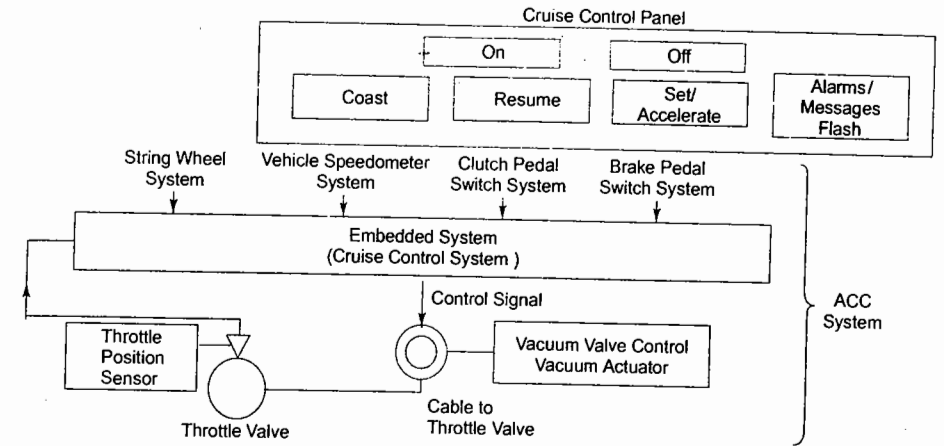


Fig. 12.11 Block diagram of units of ACC system

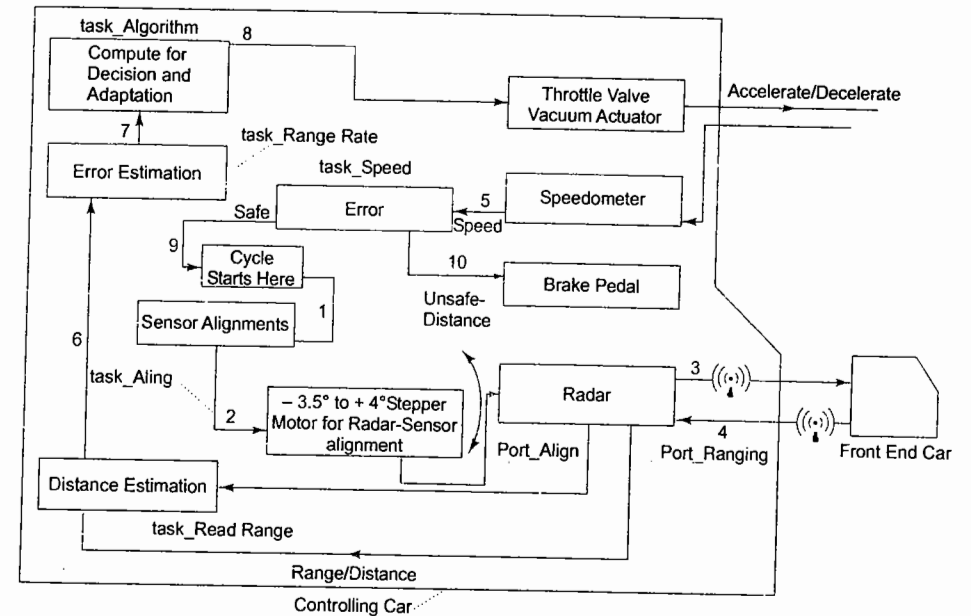


Fig. 12.12 ACC system cycle of actions and synchronizing cycle of different units

1. Task_ACC is an abstract class from which extended class(es) is derived to measure the range and errors. A task objects is instances of the extended class, which Task_ACC extends. Task_ACC extends to Task_Align, Task_Signal, Task_ReadRange and Task_Algorithm.

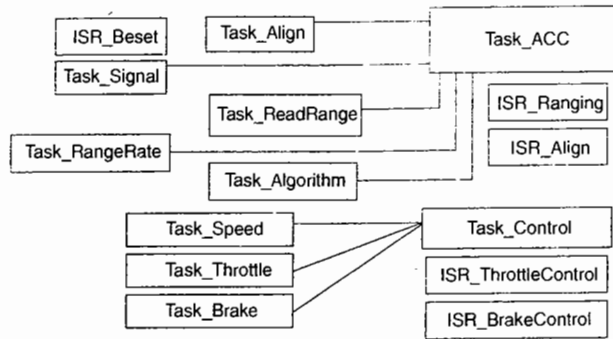


Fig. 12.13 Two class diagrams of Task_ACC and Task_Control

2. Task_Control is an abstract class from which an extended class is derived to measure the range and errors. The task objects are instances of the classes (i) Task_Brake, (ii) Task_Throttle and (iii) Task_Speed, which extends from task_Control. Task_Algorithm interfaces Task_Brake, Task_Throttle, Task_Speed and Task_ReadRange.
3. There are two ISR objects, ISR_ThrottleControl and ISR_BrakeControl.

12.3.3 ACC Hardware Architecture

A hardware system in automotive electronics has to provide functional safety. Important hardware standards and guidance at present are the following:

- (a) TTP (Time Triggered Protocol)
- (b) CAN (Controller Area Network) [Section 3.10.2]
- (c) MOST (Media Oriented System Transport)
- (d) IEE (Institute of Electrical Engineers) guidance standard exists for EMC (Electromagnetic Magnetic Control) and functional safety guidance.

Figure 12.14 shows hardware subunits in an ACC system. An automotive embedded system-based control unit uses microcontroller and separate microprocessor or DSP. ACC embeds the following hardware units:

1. A microcontroller runs the service routines and tasks (Figure 12.12) except task_Algorithm. Microcontroller has the internal RAM/ROM. RAM stores temporary variables and stack. ROM/Flash saves the application codes and RTOS codes for scheduling the tasks. CAN port interfaces with the CAN bus (Section 3.10.2) at the car. The CAN interfaces ACC system with the other embedded systems of the car. Interrupt controller at microcontroller control the interrupts.
2. A separate processor with RAM and ROM for the task_Algorithm executes the adaptive control algorithm (Figure 12.10).
3. Speedometer
4. Stepper motor-based alignment unit.
5. Stepper motor-based throttle control unit.
6. Transceiver for transmitting pulses through an antenna hidden under the plastic plates.
7. LCD dot matrix display controller, display panel with buttons.
8. Port devices are Port_Align, Port_Speed, Port_ReadRange, Port_Throttle and Port_Brake. These five port devices are used for five actions as follows: aligning transmitted beam toward the lane, measuring speed v , range d at the ACC, throttle positioning (as per the control messages from task_Algorithm) and braking action (as per messages from ISR_BrakeControl).

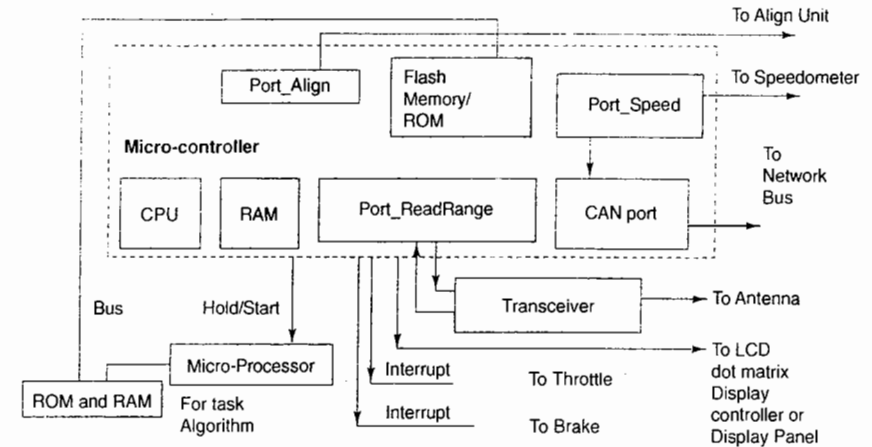


Fig. 12.14 ACC hardware

12.3.4 ACC Software Architecture

OSEK (Section 10.2) mentioned that tasks can be classified into four types and a programmer must have a clear-cut distinction: which class to use for what modules in the system. For the ACC system, Basic Conformance Class 1 (BCCI) is used. Table 12.3 lists the BCCI tasks, functions and IPCs, which are needed for the ACC with string stability algorithm. OSEK tasks can consist of three types of objects, *event* (semaphore), *resource* (statements and functions) and devices including port devices. The columns 5 and 4 give of signal and semaphore IPCs posted and taken. The semaphores are used for defining the sequence of running of tasks and cyclically running the ACC tasks and executing the string stability controller algorithm at task_Algorithm.

12.3.5 ACC Software tasks Synchronization Model

Figure 12.12 showed the units, tasks of ACC system, cycle of actions and synchronizing cycle of different units. ACC system cycle of actions and task scheduling model. Note the marks 1 to 10 of cycle starting and finishing in the figure.

1. Cycle starts from a task, task_Align on an event (ISR call). It sends the signal to a stepper motor port Port_Align and Port_Ranging. The stepper motor moves by one step clockwise or anticlockwise as directed.
2. A task task_ReadRange is for measuring front-end car range. The task disables all interrupts as it is entering a critical section. We need real-time measurement. Port_ReadRange finds d .
3. task_Speed gets the port reading at a port Port_Speed. Task sends v , using the countN and count0 interval between the initial and N^{th} rotation.
4. task_RangeRate sends the *rangeNow*. It estimates the final error in maintaining the string stability from task_ReadRange output. It estimates of the error for maintaining the car speed from the task_Speed output. It outputs both error values for use by the control system adaptive algorithm. Port_Speed connects to the speedometer system of the DAS, which displays speedNow after appropriate filtering function. Port_RangeRate transmits the *speedNow* also to other streaming cars also. Now Task calculates range and rate errors, and transmits both *rangeNow* and *speedNow*.

- task_Algorithm runs the main adaptive algorithm. Its inputs are as follows. It gets inputs from task_RangeRate. The task outputs are events to Port_Throttle and brake. Port_Throttle attaches to the vacuum actuator stepper motor. After a delay, the cycle again starts from the task_Align. It reads the statuses of Port_Brake of this and other streaming cars.

Table 12.3 List of BCC 1 task Functions and IPCs objects of the Classes shown in Figure 12.5

BCC 1 task Function	BCC 1 Priority	Action	IPCs pending	IPCs posted	Input	Output
task_Align	101	Starts on Event and Send signal to Port_Align and Port_Ranging	Reset	Align	deltaStep, step	step
task_Read-Range	103	Disable interrupts, gets signal from Port. Port activates a radar flashing, records activation time, gets time of sensing the reflected radar signal and finds time difference, deltaT. Enable interrupts.	Align	Range	-	deltaT
task_Speed	105	Event Port_Speed to start a timer, counter start message and wait for the 10 counts for the number of wheel rotations. Event outputs deltaT from port.	-	Speed	-	speedNow
task_Range-Rate	107	It calculates rangeNow. Get preset front car range and stringRange from memory and compare. Get preset cruising speed, v _{set} ; compare it with current speed speedNow.	Speed	ACC	avgTire Circum, time-Diff, deltaT, string Range, Speed, Cruise N_rotation	range-Error, speed-Error, range-Now, speed-Now
task_Algorithm	109	(i) Get errors of speed and range and execute adaptive control algorithm. (ii) Get errors of other vehicles through Port_RangeRate. (iii) Get other vehicles Port_Brake status. (iv) Get present throttle position. (v) Send output, throttleAdjst to Port_Throttle. (vi) Send signal to Port_Brake in case of emergency braking action needed. (vii) Port_Brake transmits the action needed to other vehicles also.	ACC	-	range-Error, speed-Error, All Port_RangeRate values and Port_Brake statuses, VehicleID,	throttle-adjust, emergency

¹Basic task with one task of each priority and single activation. It is called BCC 1 (Basic Conformance Class 1)

task_Algorithm generates output for ACC action. Output port for control signals generates event for a port with a throttle valve. The signals at this port, Port_Throttle, are calculated as follows: An adjustable algorithm, A, gets inputs of the speed 'speed' and front-object range 'range' as well as the unknown and unknown perturbations, P_Unknown and P_Known. It adjusts the output signal to Port_Throttle. An algorithm, B, estimates the index of performance IP. An algorithm, C, compares IP with a set of given IP values. Algorithm D is as per the adaptive control that adapts the output of C. C sends the new parameters that are to be adapted by A. Figure 12.15 shows a synchronization model for ACC tasks and semaphores.

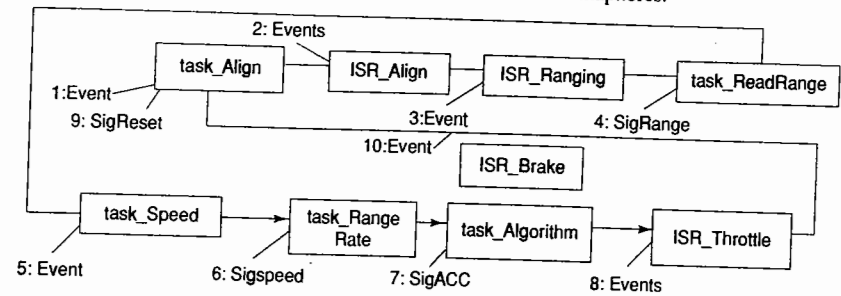


Fig. 12.15 Synchronization Model for ACC tasks

12.3.6 ACC Software Implementation

ACC software for use in automobiles must first be certified by an organization authorized to issue that certification. We have seen that OSEK OS standard is required [Section 10.2]. Only those VxWorks or MUCOS functions that adhere to OSEK must be used. Software coding IEC 61508 part 3 and MISRA C version 2 (2004) specifications of safety standards and coding language *must be used*. [MISRA stands for Motor Industry Reliability Association.] MISRA C specifies a collection of rules to be used while coding in C.

MISRA-C is a standard for C language software and defines the guidelines for automotive systems. MISRA-C version 2 (2004) specified 141 rules for coding and gave a new structure for C. Details can be found at <http://www.misra.org.uk>. Figure 12.16 shows important rules and coding standards in MISRA-C.

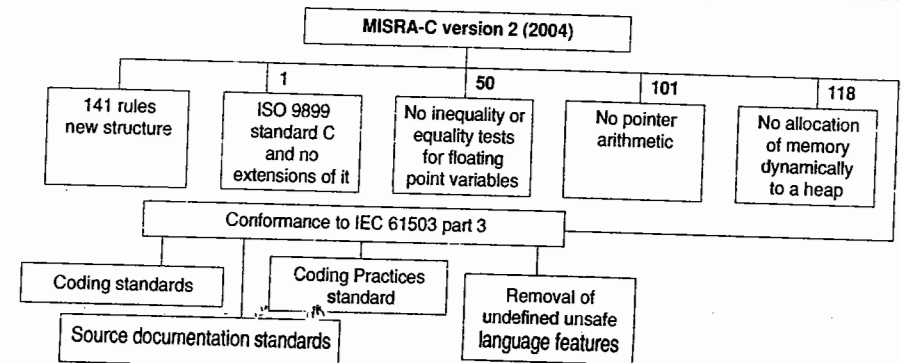


Fig. 12.16 Important rules and coding standards in MISRA-C

A few rules are discussed next. Its *first rule* is that all C codes used in automobiles must conform to ISO 9899 standard C and no extensions of it should be permitted. Rule 43 does not permit use of implicit cast that may result in a loss of information. Rule number 50 does not permit inequality or equality tests for floating point variables. Floating-point calculations undergo rounding off errors. The logic for introducing this rule is as follows: Consider 'If $(1/3) * 3 = 1$ then ...'. $1/3 = 0.33333$ with 3 or 4 in the last digit and the result is always uncertain. Rule number 65 does not permit use of floating-point numbers as a loop counter. Rule 101 does not permit use of pointer arithmetic. It is similar to the rule in Java. Rule 118 does not permit allocation of memory dynamically to a heap. Dynamic allocation has risk of using additional memory allocation than available in the system, which may cause memory leaks.

We have already learnt VxWorks (Section 9.3). Let us use VxWorks for understanding code implementation for the ACC system. To demonstrate ACC application, let us see the application of VxWorks and how we adapt OSEK in the following exemplary codes (Example 12.2).

1. Use BCC 1 type of tasks, as shown in VxWorks code application in Example 12.2. We define each task of different priority and activate it only once in the codes.
2. Use no message queues.
3. Use no creation and deletion of task.
4. Use semaphores as event flag only with no task having run-time deletion or creation of these.
5. Use MISRA C rules in coding.
6. Use disable interrupts when a task or function enters critical section and enable interrupts when leaving critical section.

Example 12.1

```
1. # include "vxWorks.h" /* Include VxWorks functions. */
# include "semLib.h" /* Include semaphore functions library. */
# include "taskLib.h" /* Include multitasking functions library. */
# include "sysLib.c" /* Include system library for system functions. */
#include "sigLib.h" /* Initialize kernel component for signal functions. */
2. /* Set system clock rate 10000 ticks per second. Every 100 µs per tick*/
sysClkRateSet (10000);
3. /* Declare and Initialize Global parameters. */
unsigned byte VehicleID; /* Declare this car ID. */
static numCars = ...; /* Numbers of cars in the string that should move at cruise speed. */
static unsigned byte N_rotation = ...; /* Initialize number of rotations needed when finding the speed.*/
static int avgTireCircum = ...; /* Initialize average tire Circumference in mm. */
4. /* Initialize the string Range = Separation to be maintained between the two cars in the string
in mm. Initialize cruise speed. */
/* All distances are in mm, speeds in km/hr and time in nanosecond unless otherwise
specified. */
static int stringRange = ...; /* In unit of mm */
static int CruiseSpeed = ...; /* In unit of km/hr */
static float unitChange = 3600000.0; /* Km in one mm divided by hours in one nanosecond. */
float permittedSpeedError = ...; /* In units of mm/tis error in speed permitted. It prevents oscillations in
the control system. */
static boolean alignment = false; /* Declare alignment = false to initiate radar transmitter alignment. */
```

```
5. /* Other Variables. */
static byte *step; /* Stepper motor step angle in degrees. */
static byte *deltaStep = 0; /* Stepper motor step angle change in degrees. */
/* Time difference between emitted signal and reflected signal from front-end car. rangeNow is present
range in mm. It is timeDiff multiplied by speedNow after a filter function application. */
static unsigned long *timeDiff; static int *rangeNow;
static unsigned long *deltaT; /* Time interval for N rotation in ns. */
6. /* Declare pointers for variables range error, range now, speed error, speed now. */
int rangeError, speedError, rangeNow = 0. speedNow = 0; /* Speeds are in km/hr and range in mm. */
7. /* Declare arrays of size number of cars, numCars. These many cars are running as a string. Declare
brakeStatus, RangeErrors, SpeedErrors, Ranges, Speeds for all the cars. */
boolean brakeStatus [numCars];
int RangeErrors [numCars], Ranges [numCars], SpeedErrors [numCars], Speeds [numCars];
boolean emergency [numCars]; /* Declare variable for emergency message sent to Port_Brake of Nth car.
*/
int *throttleAdjst; /* Declare variable for throttle adjusting parameter */
8. /* Other Variables. */

9. /* Declare all Table 12.3 task function prototypes. */
void task_Alignment (SemID SigReset, SemID SigAlign, byte *step, byte *deltaStep); /*task for aligning
stepper motor in front-end car view. */
void task_ReadRange (SIGID SigAlign, SIGID SigRange, unsigned long *timeDiff); /*task for receiving
the timeDiff using the radar for calculating rangeNow. */
void task_Speed (SIGID SigRange, SIGID SigSpeed, unsigned long *deltaT); /*task for receiving the
deltaT using the wheel counter and timer for calculating speedNow. */
void task_Range_Rate (SIGID SigSpeed, SIGID SigReset, SIGID SigACC, int avgTireCircum, unsigned
byte N_rotation, int CruiseSpeed, int stringRange, unsigned long *time-Diff, unsigned long *deltaT, int
*range-Error, int *speedError, int *range-Now, int *speed-Now); /*task for calculating rangeNow, speedNow,
rangeError, speedError. */
void task_Algorithm (SIGID SigACC, SIGID SigReset, boolean brakeStatus [numCars],
int RangeErrors [numCars], int Ranges [numCars], int SpeedErrors [numCars], int Speeds [numCars],
boolean emergency [numCars], unsigned byte VehicleID); /* Declare array for emergency message sent to
Port_Brake of Nth car. */
int *throttleAdjst; /* Declare variable for throttle adjusting parameter */
10. /* Declare all Table 12.3 task IDs, Priorities, Options and Stacksize. Let initial ID, till spawned
(initiated) be none. No options and Stacksize = 4096 for each of six tasks. */
int task_AlignID = ERROR; int task_AlignPriority = 101; int task_AlignOptions = 0; int
task_AlignStackSize = 4096;
int task_ReadRangeID = ERROR; int task_ReadRangePriority = 103; int task_ReadRangeOptions = 0; int
task_ReadRangeStackSize = 4096;
int task_SpeedID = ERROR; int task_SpeedPriority = 105; int task_SpeedOptions = 0; int
task_SpeedStackSize = 4096;
int task_RangeRateID = ERROR; int task_RangeRatePriority = 107; int task_RangeRateOptions = 0; int
```

```

task_RangeRateStackSize = 4096;
int task_AlgorithmID = ERROR; int task_AlgorithmPriority = 109; int task_AlgorithmOptions = 0; int
task_AlgorithmStackSize = 4096;
11. /* Create and Initiate (Spawn) all the six tasks of Table 12.3. */
task_AlignID = taskSpawn (" task_Align", task_AlignPriority, task_AlignOptions,
task_AlignStackSize, void (*task_Alignment) (SemID SigReset, SemID SigAlign, byte *step, byte
*deltaStep), 0, 0, 0, 0, 0, 0, 0, 0, 0);
task_ReadRangeID = taskSpawn (" ttask_ReadRange", task_ReadRangePriority, task_ReadRangeOptions,
task_ReadRangeStackSize, void (*task_ReadRange) (SIGID SigAlign, SIGID SigRange, unsigned long
*timeDiff), 0, 0, 0, 0, 0, 0, 0, 0, 0);
task_SpeedID = taskSpawn (" ttask_Speed", task_SpeedPriority, task_SpeedOptions, task_SpeedStackSize,
void (*task_Speed) (SIGID SigRange, SIGID SigSpeed, unsigned long *deltaT), 0, 0, 0, 0, 0, 0, 0, 0, 0);
task_RangeRateID = taskSpawn (" ttask_Range_Rate", task_RangeRatePriority, task_RangeRateOptions,
task_RangeRateStackSize, void (*task_Range_Rate) (SIGID SigSpeed, SIGID SigReset, SIGID SigACC,
int a;gTireCircum, unsigned byte N_rotation, int CruiseSpeed, int stringRange, unsigned long *time-Diff,
unsigned long *deltaT, int *range-Error, int *speedError, int *rangeNow, int *speedNow), 0, 0, 0, 0, 0, 0,
0, 0, 0, 0);
task_AlgorithmID = taskSpawn ("ttask_Algorithm", task_AlgorithmPriority, task_AlgorithmOptions,
task_AlgorithmStackSize, void (*task_Algorithm) (SIGID SigACC, Sig_ID SigReset, boolean brakeStatus
[numCars], int RangeErrors [numCars], int Ranges [numCars], int SpeedErrors [numCars], int Speeds
[numCars], boolean emergency [numCars], unsigned byte VehicleID), 0, 0, 0, 0, 0, 0, 0, 0, 0);
12. /* Declare IDs and create binary semaphore event flags. */
SIGID SigAlign, SigRange, SigSpeed, SigACC, SigReset; /* Declared for Table 12.3 five tasks. */
13. /* Create the binary semaphores taken in FIFO as empty to start with. */
SigAlign = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
SigRange = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
SigSpeed = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
SigACC = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
SigReset = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
14. /* Declare function for starting an RTC timer at the Port_Ranging. */
void RTCTimer_Port_Ranging_Start ( ) {
};
15. /* Declare a function for starting an RTC timer at the Port_Speed. */
void RTCTimer_Port_Speed_Start ( ) {
};
16. /* Declare a function to read 64-bit time from an RTC. */
unsigned long timer_gettime (&RTC) {
};

```

```

17. /* Define a macro for calculating time between instance when control bit = true and Status flag = true.
*/
boolean * CB; /* Control bit */
boolean * SF; /* Status flag. */
*CB = 0; /* Control bit = 0; */
*SF = 0; /* Status flag = 0; */
unsigned short * RTC; /* Pointer to a real-time clock. */
#define unsigned long calculate_TimeInterval (unsigned short * RTC, boolean * CB, boolean *SF) (
unsigned long timeInstance0, timerInstance1;
while (*CB != 1 && *SF == 0) { }; /* Wait for read instruction to timer. */
timeInstance0 = timer_gettime (&RTC); /* Find initial time at start in the timer */
while (*SF != 1) { }; /* Wait for sensor status flag to be true */
timeInstance1 = timer_gettime (&RTC); /* Find initial time at start in the timer */
*SF = 0; *CB = 0;
return (timeInstance1 - timeInstance0);
) /* End of Macro to calculating time interval between two instances specified by CB and SF becoming
true. */
18. /* Define a macro for calculating time interval deltaT between instance when control run bit = true and
instance of Nth count input. */
boolean CR; /* Control Run bit. */
boolean countInput; /* Status flag. */
*CR = 0; /* Control run bit = 0; */
*countInput = 0; /* countInput initial value. */
#define unsigned long DelT (unsigned short &RTC, boolean *CR, N_rotation, boolean * countInput)(
unsigned long timeInstance = 0; byte N = 0;
for (byte N = 0; N < N_rotation; N++) {
while (*CR != 1 && *countInput != 0) { }; /* Wait for count input true. */
timeInstance += timer_gettime (&RTC); /* Find initial time at start in the timer */
*countInput = 0; /*Reset countInput. It will set on start of next rotation. */
}; *countInput = 0;
return (timeInstance);
) /* End of Macro to calculate time interval for N count inputs. */
19. /* Declare Macro for sending a byte for step angle setting to Port_Align. */
unsigned short * Port_Align; /* Declare a pointer for Port_Align. */
#define Out_Alignment (&Port_Align, Step) /* Codes in Assembly Language for stepper motor routine.
*/
); /* End of Macro to sending byte for step angle to Port_Align. */
20. /* Declare Macro to find timeDiff from Port_Ranging. */
unsigned short RTC_Port_Ranging = ...; /* Declare Address of RTC at Port_Ranging. */
RTCTimer_Port_Ranging_Start ( );
#define RANGE (unsigned short * Port_Ranging, unsigned long * timeDiff) (
unsigned short * RTC_Port_Ranging = ...; /* Declare address of RTC of Port_Ranging. */

```

```

intLock ( ); /*disable interrupts. */
/* Codes in Assembly Language for Ranging routine for Start Radar transmission by making control bit
CB = 1*/
*CB = 1;
*timeDiff = calculate_TimeInterval (&RTC_Port_Ranging, &CB, &SF);
intUnlock ( ); /* Enable Interrupts. */
)/* End of Macro to find time interval for reflected radar signals. */
21. /* Declare Macro to find deltaT from Port_Speed. */
unsigned short RTC_Port_Speed = ...; /* Declare Address of RTC at Port_Speed. */
RTCtimer_Port_Speed_Start ( );
#define SPEED (unsigned short * Port_Speed, unsigned long * deltaT) (
unsigned short * RTC_Port_Ranging = .....; /* Declare address of RTC of Port_Ranging.*/
intLock ( ); /*disable interrupts. */
/* Codes in Assembly Language for Speed routine for Start counting the tire rotation count inputs by
making control bit CR = 1*/
*CR = 1;
*deltaT = DelT (&RTC_Port_Speed, &CR, N_rotation, &countInput);
intUnlock ( ); /* Enable Interrupts. */
)/* End of Macro to find time interval for N count-inputs. */
22.
# define float filter_speed (float calculatedSpeed, int *speedNow, float permittedSpeedError) /* Codes
for filtering the calculated speed. If the calculated value in mm/ns is within plus minus limit of
permittedSpeedError, do not change it; else modify it with new value. */
float a;
a = (float) (speedNow/ unitChange);
if (calculatedSpeed > a + permittedSpeedError || calculatedSpeed < a - permittedSpeedError) {return
(calculatedSpeed);} else return (a);
)/* End of macro for filtering the speed calculated to prevent vibrations and oscillation in controlling
vehicle. */
23.
#define RangeRate (unsigned short * Port_RangeRate, int *speedNow, int *rangeNow, int *speedError,
int *rangeError, unsigned byte VehicleID) (
/*Assembly codes for sending to Port_RangeRate and transmitting the range and rate parameters and
vehicleID. Port_RangeRate sends also *speedNow for display on speedometer at Port_Speed. /
.
.
); /* End of Macro to transmit range and rate parameters through Port_RangeRate. */
24. to 26. /* Code for Other Declaration Steps specific to various functions */
.
.
/* End of the codes for creation of tasks, semaphores, message queue, pipe tasks, and variables and all
needed function declarations */
/*****
27. /* Start of Codes for task_Alignment. */

```

```

void task_Alignment (SemID SigReset, SemID SigAlign, byte *step, byte *deltaStep) {
.
28.
while (1) /* Start of while loop. */
/* When cycle starts the semaphore is given. Wait for it. */
semTake (SigReset, WAIT_FOREVER); /* Wait for Cycle Reset Event flag. */
29. /* Codes for sending the step to the address of Port_Ranging. *
*step = *step + *deltaStep;
Out_Alignment (&Port_Align, *Step);
semGive (SigAlign);
}; /* End of while loop. */
} /* End of task_Alignment. */
/*****
30. /* Start of codes for task_ReadRange. */
void task_ReadRange (SIGID SigAlign, SIGID SigRange, unsigned long *timeDiff) {
31. /* Initial assignments of the variables and pre-infinite loop statements that execute only once */
static unsigned short Port_Ranging = .....; /* Declare pointer to Port_Ranging. */
.
.
32. while (1) { /* Start an infinite while-loop. We can also use FOREVER in place of while (1). */
33. /* Wait for SigAlign state change to SEM_FULL by semGive function. */
semTake (SigAlign, WAIT_FOREVER);
/*Send signal to Port_Ranging. Port activates a radar flashing, records activation time, gets time of
sensing the reflected radar signal and finds time difference, timeDiff. Enable interrupts. */
RANGE (& Port_Ranging, &timeDiff);
);
semGive (SigRange);
};
34. } /* End of the codes for task_ReadRange. */
/*****
35. /* Start of codes for task_Speed. */
void task_Speed (SIGID SigRange, SIGID SigSpeed, unsigned long *deltaT) {
36. /* Initial assignments of the variables and pre-infinite loop statements that execute only once */
static unsigned short Port_Speed = ...; /* Declare pointer to Port_Speed. */
.
.
37. while (1) /* Start task infinite loop. */
semTake (SigRange, WAIT_FOREVER);
/* Codes for receiving the deltaT using the wheel counter and timer for later on calculating speedNow. */
SPEED (& Port_Speed, &deltaT);
semGive (SigSpeed);
}; /* End of while loop. */
38. } /* End of codes for task_Speed */
/*****

```

```

/* Start of codes for task_Range_Rate. */
void task_Range_Rate (SIGID SigSpeed, SIGID SigReset, SIGID SigACC, int avgTireCircum, unsigned
byte N_rotation, int CruiseSpeed, int stringRange, unsigned long *time-Diff, unsigned long *deltaT, int *
range-Error, int *speedError, int *range-Now, int *speed-Now) {
static unsigned short Port_RangeRate = ...; /* Declare pointer to Port_Ranging. */
.
.
39. while (1) /* Start task infinite loop. */
semTake (SigRange, WAIT_FOREVER);
40. /* .Codes for calculating rangeNow, speedNow, rangeError, speedError. */
*speedNow = (int) (unitChange * filter_speed ((float) (avgTireCircum * N_rotation) / (float) (*deltaT),
int *speedNow, float permittedSpeedError));
*rangeNow = (*speedNow/unitChange) * (*timeDiff)/2.0; /* Divide by 2 because reflected signal travels
twice the distance in mm/ns. */
*speedError = cruiseSpeed - *SpeedNow;
RangeRate (& Port_RangeRate, *speedNow, *rangeNow, *speedError, *rangeError, VehicleID); /* Send
the parameters for transmission to other vehicles and Port_Speed for displays. */
if (alignment != true) {semGive (SigReset);
41. /* Code for loop of tasks of priorities 101, 103, 105 in which the values of rangeNow are calculated at
different step values by changing deltaStep and, finally, at that instance, alignment is declared as true, for
rangeNow is minimum. Front-end vehicle is now in line of sight. */
.
.
} else semGive (SigACC); /* After alignment is perfect the control algorithm is sent the event flag. */
/* End of while loop. */
42. } /* End of codes for task_Range_Rate. */
/*****
/* Start of Codes for task_Algorithm. */
void task_Algorithm (SIGID SigACC, SIGID SigReset, boolean brakeStatus [numCars],
int RangeErrors [numCars], int Ranges [numCars], int SpeedErrors [numCars], int Speeds [numCars],
boolean emergency [numCars], unsigned byte VehicleID) {
.
.
43. while (1) /* Start task infinite loop. */
semTake (SigACC, WAIT_FOREVER);
44. /* Assembly codes for getting errors of other vehicles through Port_RangeRate and other vehicles
Port_Brake statuses through Port_Brake. */
.
.
45. /* Assembly codes to read throttle position from Port_Throttle. */
.
.
46. /* Codes for cruise speed adaptive algorithm and code for string stability maintaining adaptive algorithm
to generate appropriate throttleAdjsut signal to Port_Throttie. */

```

```

47. /* Codes for Port_Brake action, if emergency = true. Port_Brake transmits the action needed
to other vehicles also. */
if (emergency [numCars] == 1) {
.
.
} else semGive (SigACC); /* After alignment is perfect the control algorithm is sent the
event flag. */
48. /* End of while loop. */
49. } /* End of codes for task_Algorithm. */
/*****

```

12.4 CASE STUDY OF AN EMBEDDED SYSTEM FOR A SMART CARD

Section 1.10.3 introduced the smart card system hardware and software. Section 12.4.1 gives the requirements and functioning of smart card communication system. Section 12.4.2 gives the class diagram. Figure 1.13 showed smart card-system hardware components for a contact-less smart. Sections 12.4.3 and 12.4.4 give the hardware and software architecture and synchronization model. Section 12.4.5 gives the exemplary codes.

12.4.1 Requirements

Assume a contact-less smart card for bank transactions. Let it not be magnetic. [The earlier card used a magnetic strip to hold the nonvolatile memory. Nowadays, it is EEPROM or flash that is used to hold nonvolatile application data.] Requirements of smart card communication system with a host can be understood through a requirement-table given in Table 12.4.

12.4.2 Class Diagram

Table 12.4 listed the functions and the different tasks. An abstract class is Task_CardCommunication. Figure 12.17 shows the class diagram of Task_CardCommunication. A cycle of actions and card-host synchronization in the card leads us to the model Task_CardCommunication for system tasks. Card system communicates to host for identifying host and authenticating itself to the host. ISR1_Port_IO, ISR2_Port_IO and ISR3_Port_IO are interfaces to the tasks. [A class gives the implementation methods of the interfacing routines.] The task_Appl, task_PW, task_ReadPort, and resetTask are the objects of Task_Appl, Task_PW, Task_ReadPort and Task_Reset, respectively. These classes are extended classes of abstract class Task_CardCommunication.

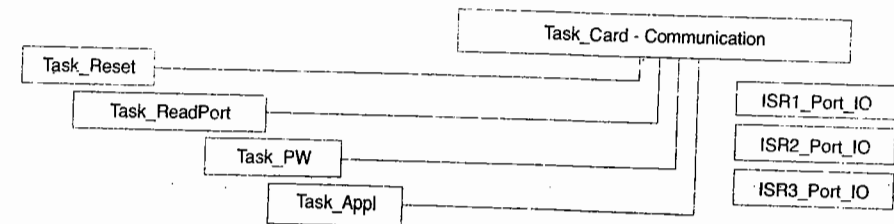


Fig. 12.17 Class diagrams of Task_CardCommunication

Table 12.4 Requirements of smart card communication system with a host

Requirement	Description
Purpose	1. Enabling authentication and verification of card and card holder by a host and enabling GUI at host machine to interact with the card holder/user for the required transactions; for example, financial transactions with a bank or credit card transactions.
System Functioning	1. The card inserts at host machine. The radiations from the host activate a charge pump at card. The charge pump powers the SoC circuit, which consists of card processor, memory, timer, interrupt handler and Port_IO. 2. On power up, system-reset signals resetTask to start. The resetTask sends the messages— <i>requestHeader</i> and <i>requestStart</i> for waiting task task_ReadPort. 3. task_ReadPort sends requests for host identification and reads through the Port_IO the host-identification message and request from host for card identification. 4. The task_PW sends through Port_IO the requested card identification after system receives the host identity through Port_IO. 5. The task_Appl then runs required APL. The <i>requestApplClose</i> message closes the application. 6. The card can now be withdrawn and all transactions between card-holder/user now takes place through GUIs using at the host control panel (screen or touch screen or LCD display panel).
Inputs	1. Received header and messages at IO port <i>Port_IO</i> from host through the antenna.
Signals, Events and Notifications	1. On power up by radiation-powered charge-pump supply of the card, a <i>signal</i> to start the system boot program at resetTask. 2. Card start <i>requestHeader</i> message to task_ReadPort from resetTask. 3. Host authentication request <i>requestStart</i> message to task_ReadPort from resetTask to enable requests for Port_IO. 4. <i>UserPW</i> verification message (notification) through Port_IO from host. 5. Card application close request <i>requestApplClose</i> message to Port_IO.
Outputs	Transmitted headers and messages at Port_IO through antenna.
Control Panel	No control panel is at the card. The control panel and GUIs activate at the host machine (for example, at ATM or credit card reader).
Design metrics	1. <i>Power Source and Dissipation</i> : Radiation powered contact-less operation. 2. <i>Code size</i> : Code-size generated should be optimum. The card system memory needs should not exceed 64 kB memory. Limited use of data types: multidimensional arrays, long 64-bit integer and floating points and very limited use of the error handlers, exceptions, signals, serialization, debugging and profiling. 3. <i>File system(s)</i> : Three-layered file system for the data. One file for the <i>master file</i> to store all file headers. A header has strings for file status, access conditions and file-lock. The second file is a <i>dedicated file</i> to hold a file grouping and headers of the immediate successor elementary files of the group. The third file is the <i>elementary file</i> to hold the file header and file data. 4. <i>File management</i> : There is either a fixed length file management or a variable file length management with each file with a predefined offset. 5. <i>Microcontroller hardware</i> : Generates distinct coded physical addresses for the program and data logical addresses. Protected once writable memory space. 6. <i>Validity</i> : System is embedded with expiry date, after which the card authorization through the hosts disables. 7. <i>Extendibility</i> : The system expiry date is extendable by transactions and authorization of master control unit (for example, bank servce). 8. <i>Performance</i> : Less than 1 s for transferring control from the card to host machine.

(Contd)

Requirement	Description
	9. <i>Process Deadlines</i> : None. 10. <i>User Interfaces</i> : At host machine, graphic at LCD or touchscreen display on LCD and commands for card holder (card user) transactions. 11. <i>Engineering Cost</i> : US\$ 50000 (assumed). 12. <i>Manufacturing Cost</i> : US\$ 1 (assumed).
Test and validation conditions	1. Tested on different host machine versions for fail proof card-host communication.

1. Task_CardCommunication is an abstract class from which extended to class (es) derive to read port and authenticate. The tasks (objects) are the instances of the classes Task_Appl, Task_Reset, Task_ReadPort and Task_ReadRange.
2. Task_ReadPort interfaces ISR1_Port_IO.
3. The task_PW is object of Task_PW and interfaces ISR2_Port_IO. Task_Appl interfaces ISR3_Port_IO.

12.4.3 Hardware and Software Architecture

Smart card hardware was introduced in Section 1.10.3. Figure 12.18 shows hardware inside the card.

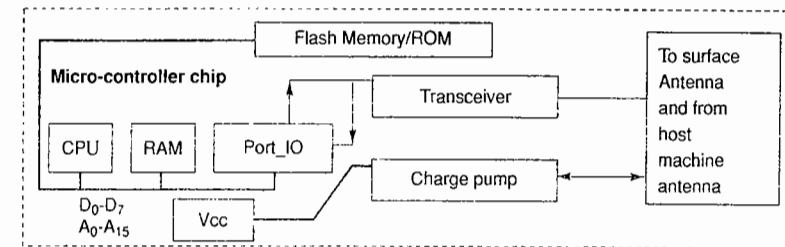


Fig. 12.18 Smart card hardware

Software using Java Card™ provides one solution. [Refer 5.7.5] JVM has thread scheduler built in. No separate multitasking OS is thus needed when using Java because all Java byte codes run in JVM environment. Java provides the features to support (i) security using class `java.lang.SecurityManager`, (ii) cryptographic needs (package `java.security`). Java provides support to connections, datagrams, IO streams and network sockets. Java mix is a new technology in which the native applications of the card run in C or C++ and downloadable applications run in Java or Java Card™. The system has OS and JVM both.

SmartOS is an assumed hypothetical OS in this example, as RTOS in the card. Remember that a similar OS function name is used for understanding purposes identical to MUCOS but actual SmartOS has to be different from MUCOS. Its file structure is different. It has two functions as follows: The function `unsigned char [] SmartOSEncrypt (unsigned char *applStr, EnType type)` encrypts as per encryption method, `EnType = "RSA"` or `"DES"` algorithm chosen and returns the encrypted string. The function `unsigned char [] SmartOSDecrypt (unsigned char *Str, DeType type)` encrypts as per deciphering method, `DeType = "RSA"` or `"DES"` algorithm chosen and returns the deciphered string. `SmartOSEncrypt` and `SmartOSDecrypt` execute after verifying the access conditions from the data files that store the keys, PINs and password.

Table 12.5 gives the tasks for the card OS in this case study.

12.4.4 Synchronization Model

Following are the actions on the card placed near the host machine antenna in a machine slot.

Step 1: Receive from the host, on card insertion, the radiation of carrier frequency or clock signals in case of contact with the card. Extract charge for the system power supply for the modem, processor, memories and Port_IO (card's UART port) device.

Table 12.5 List of tasks, Functions and IPCs

Task Function	Priority	Action	IPCs pending	IPCs posted	String or System or Host input	String or System or Host Output
resetTask	1	Initiates system timer ticks, creates tasks, sends initial messages and suspends itself.	None	SigReset, MsgQStart	SmartOS call to the main	request-Header, requestStart
task_ReadPort	2	Wait for resetTask Suspension, sends the queue messages and receives the messages. Starts the application and seeks closure permission for closing the application.	SigReset, Messages from MsgQStart, MsgQPW, MsgQAppl, MsgQAppl-Close	SePW	Functions Smart OS-Encrypt, SmartOS-decrypt, ApplStr, Str, close-Permitted	request-password, request-Appl, request-ApplClose
task_PW	3	Sends request for password on verification of host when SemPW = 1.	SemPW	MsgQPW, SemAppl	request-Password	-
task_Appl	8	when SemPW = 1. runs the application program.	SemAppl	MsgQAppl	-	-

Step 2: Execute codes for a boot up task on reset resetTask. Let us code in a similar way as the codes in Example 11.1 for Firsttask. The codes begin to execute from the main and the main creates and initiates this task and starts the SmartOS. There it is the resetTask, which executes first.

The steps taken by the task synchronization model are as follows:

1. **resetTask** is the task that executes like Firsttask in Example 11.1. It suspends permanently after the following: (a) It initiates the timer for the system ticks, which are reset at 1 ms. (b) It creates three tasks, task_ReadPort, task_PW and task_Appl of the system that are described below. (c) For a waiting task, task_ReadPort, it sends into a message queue MsgQStart, the request header string *requestHeader*. The latter specifies the bank-allotted PIN to the user. (d) It also sends another string *requestStart* a request for host PIN at the I/O Port Port_IO in order to identify the host. (e) It posts a semaphore flag, SigReset, and suspends itself so that system control does not return to it till another reset.

2. A macro function *ReceiveStr* (&Str) uses function `void portIO_ISR_Input (*portIOdata)` to return an input string Str. The `portIO_ISR_Input` receives the characters one by one from the port on successive calls. Similarly, a macro function *SendStr* (&ApplStr) is used by the function `void Port_OutStr (unsigned char [] *applStr)` to send an output string. `portIO_ISR_Output` sends a character to the port.
3. **task_ReadPort** begins only when a semaphore *SigReset* is posted by the *resetTask*. (a) *task_ReadPort* takes the message from the queue *MsgQStart* and gets the pending queue messages, *requestHeader* and *requestStart*. It encrypts these two strings and sends to the Port_IO, which transmits it to the host through the transceiver or modem. It receives host message *hostStr* through the transceiver. Host specifies, by the *hostStr*, the host PIN. This PIN is the one used for the bank authorization PIN of the card. (b) It posts a semaphore *SemPW* flag to the waiting task *task_PW* if the presently running task verifies the hostPIN. It waits for a message from the queue *MsgQPW* and receives *userPW* after deciphering the port input data string. (c) It posts semaphore *SemAppl*, in case the user password stored in a file at the EEPROM is verified. It posts a different semaphore *SemAppl*, if it verifies the user password. It sends in the end a close request message into a queue *MsgQApplClose*. Message is *requestApplClose* to the Port_IO and receives encrypted string "Closure Permitted". The tasks delete on deciphering.
4. *task_PW* after encryption on taking the pending *SemPW* is to send the string *requestPW*. When it takes *SemPW*, it sends the *requestPW* into the *MsgQPW*. *task_ReadPort* will send it to the host through the IO Port, Port_IO, in order to identify the user at the host.
5. *task_Appl* runs on taking the semaphore *SemAppl* and executes the operations. The operation may (i) modify user password. (ii) print mini statement of bank account of the user. (iii) eject requisite cash from the host, (iv) request for accepting the envelope with cash, (v) request for a print of this transaction and (vi) request for a transfer to another party. It interacts through *task_ReadPort* by sending the messages through the queue *MsgQAppl*.

The task synchronization model is also shown in Figure 12.19.

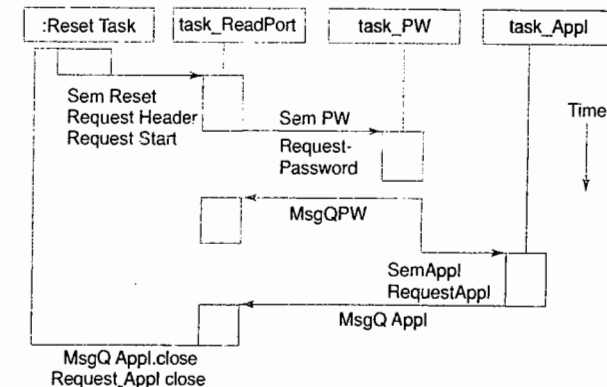


Fig. 12.19 Tasks and the synchronization model

12.4.5 Exemplary Codes

Example 12.2 gives the exemplary coding procedure for an application of this card.

Example 12.2

```

1. /* Preprocessor definitions for maximum number of interprocess events to let the SmartOS allocate
memory for the Event Control Blocks */
#define SmartOS_MAX_EVENTS 24/* Let maximum IPC events be 24 */
#define SmartOS_SEM_EN 1/* Enable inclusion of semaphore functions in application. */
#define SmartOS_Q_EN 1/* Enable inclusion of queue functions for sending the string pointers to
task_ReadPort */
#define SmartOS_task_De_En = 0/* Disable task deletion by SmartOS at the start. */
/* End of preprocessor commands for enabling IPC functions of the SmartOS*/
2. /* Specify all user prototype of the reset task function that is called by the main function and is to be
scheduled by SmartOS first at the start. In Step 11, we will be creating all other tasks within the reset task.
Remember: Static means permanent memory allocation. */
static void resetTask (void *taskPointer);
static SmartOS_STK_resetTaskStack [resetTask_StackSize];
3. /* Define public variable of the task service and timing functions */
#define SmartOS_TASK_IDLE_STK_SIZE 100 /* Let memory allocation for an idle state task stack size
be 100*/
#define SmartOS_TICKS_PER_SEC 1000 /* Let the number of ticks be 1000 per second. An RTCSWT
will interrupt and thus tick every 1 ms to update counts. */
#define resetTask_Priority 1 /* Define reset task in main priority */
#define resetTask_StackSize 100 /* Define reset task in main stack size */
#define TAF_In = 0; /* Define flag for signaling modem interrupt for receiving a character. */
#define TAF_Out = 0; /* Define flag for signal from a modem interrupt after sending a character. */
/*-----*/
; /* Prototype definitions for three tasks for the car application codes after reset. */
static void task_ReadPort (void *taskPointer);
static void task_PW (void *taskPointer);
static void task_Appl (void *taskPointer);
. /* Definitions for three task stacks. */
static SmartOS_STK task_ReadPortStack [task_ReadPortStackSize];
static SmartOS_STK task_PWStack [task_PWStackSize];
static SmartOS_STK task_ApplStack [task_ApplStackSize];
. /* Definitions for three task stack size. */
#define task_ReadPortStackSize 100 /* Define task 2 stack size*/
#define task_PWStackSize 100 /* Define task 3 stack size*/
#define task_ApplStackSize 100 /* Define task 4 stack size*/
. /* Definitions for three task priorities. */
#define task_ReadPortPriority 2 /* Define task 2 priority */
#define task_PWPriority 3 /* Define task 3 priority */
. /* Prototype definitions for the semaphores. */
SmartOS_EVENT *SigReset; /* First task that resets the card posts it. */
SmartOS_EVENT *SemPW; /* task_PW posts it to send request for getting user password through the
port. */

```

```

SmartOS_EVENT *SemAppl; /* Needed when using Semaphore as flag for interprocess communication
between task_ReadPort and task_PW. */
7. /* Prototype definitions for the queues. */
SmartOS_EVENT *MsgQStart; /* Needed for IPC between resetTask and task_ReadPort. */
void *MsgQStart [QStartMessagesSize]; /* Let the maximum number of message pointers at the queue be
QStartMessagesSize. */
SmartOS_EVENT *MsgQPW; /* Needed for IPC between task_PW and task_ReadPort. */
void *MsgQPW [QPWMessagesSize]; /* Let the maximum number of message pointers at the queue be
QPWMessagesSize. */
SmartOS_EVENT *MsgQAppl; /* Needed for IPC between task_Appl and task_ReadPort. */
void *MsgQAppl [QApplMessagesSize]; /* Let the maximum number of message pointers at the queue be
QApplMessagesSize. */
SmartOS_EVENT *MsgQApplClose; /* Needed for IPC between task_Appl and task_ReadPort. */
void *MsgQApplClose [QApplCloseMessagesSize]; /* Let the maximum number of message pointers at
the queue be QApplCloseMessagesSize. */
8. /* Define both queue array sizes. Assume a maximum of 16 strings can be sent in a queue. */
#define QStartMessagesSize 16; /* Define size of start message pointer queue when full */
#define QPWMessagesSize 16; /* Define size of password message pointer queue when full */
#define QApplMessagesSize 16; /* Define size of application message pointer queue when full */
#define QApplCloseMessagesSize 16; /* Define size of application message pointer queue when full */
9. /* Define Semaphore initial values, 0 when used as an event flag and 1 when resource key. */
SigReset = SmartOSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as an event flag
from resetTask. */
SemPW = SmartOSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as an event flag
from task_ReadPort */
SemAppl = SmartOSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as an event flag
from task_ReadPort */
/* Define a top of the message pointer array. QMsgPointer points to top of the Messages to start with. */
MsgQStart = SmartOSQCreate (&QStart [0], QStartMessagesSize);
MsgQPW = SmartOSQCreate (&QPW [0], QPWMessagesSize);
MsgQAppl = SmartOSQCreate (&QAppl [0], QApplMessagesSize);
MsgQApplClose = SmartOSQCreate (&QApplClose [0], QApplCloseMessagesSize);
.
.
10. /* Any other SmartOS Events for the IPCs. */
.
.
11. /* Code similar to steps for ISR_CharInt and Task_ReadPortA in Example 9.16. These were for reading
from Port A and storing a character. Here, we have Port_IO. */
/* Prototype Declarations for modem Port_IO input and output strings. */
char [ ] Str; /* Port_IO input string to hold the data from the host through the demodulator circuit of
modem. */
char [ ] ApplStr; /* Port_IO string, which the modem transfers to host after modulation. */

```

```

unsigned char *portIndata; unsigned char *portOutdata;
void portIO_ISR_Input (*portIndata); /* Prototype declaration for receiving an input character. */
void portIO_ISR_Output (*portOutdata); /* Prototype declaration for sending an output character. */
/* Start of Port_IO Input Interrupt Service Routine */
void portIO_ISR_Input (*portIodata) {
    disable_PortIO_InIntr (); /* Function for disabling another interrupt from port IO input. */
    /* Insert Code for reading Port I/O bits
portIodata = &Str;
*/
/* Start of Port_IO Output Interrupt Service Routine */
void portIO_ISR_Output (*portIodata) {
    disable_PortIO_OutIntr (); /* Function for disabling another interrupt from port I/O output */
    /* Define a macro for sending a String */
    unsigned byte i;
    #define SendStr (&ApplStr) (
        portIodata = &ApplStr; i = 0; STAF_Out = 1;
        while (STAF_Out == 1 && ApplStr [i] != NULL) {
            portIodata = ApplStr [i]; /*Pick a character from the queue message. */
            portIO_ISR_Output (&portIodata); /*Send it to the Port_IO for the modem output. */
            i++; /* Be Ready for next Character */
            STAF_Out = 0; /*Port interrupt when one character sent by setting STAF_Out again. */
        }
        ApplStr = ""; /* Clear the Queue message for the new one*/
        ApplStr = ApplStr [0];
    } /* End of the macro function SendStr. */
    /* Define a macro function for string comparison. Note: 'C' function strcmp is available at a C library. In
order to optimize codes, we are not using strcmp library function, but our own macro here. */
    #define boolean strcmp (ApplStr, Str) (
        /* Define a macro for receiving a string */
        #define ReceiveStr (&Str) (
            while (STAF_In != 1) { }; i=0;
            /* Execute interrupt service routines for each character received at modem Port_IO*/
            while (STAF_In == 1 && Str [i] != NULL) {
                portIO_ISR_Input (&portIodata);};
            STAF_In = 0; /* Remember: as soon as Port A is read STAF will reset itself to reflect
next interrupt status. */
            Str [i] = portIodata; /* Write port I/O input array element from the returned data*/
            ++;};
        } /* End of the macro function ReceiveStr. */
} /* Start of the codes of the application from Main.
Note: Code steps are similar to in Example 9.16 */

```

```

void main (void) {
    /* Initiate SmartOS RTOS to use OS kernel functions */
    SmartOSInit ();
    13. /* Create Reset task, resetTask that must execute once before any other. task creates by defining its
Identity as resetTask, stack size and other TCB parameters. */
    SmartOSStaskCreate (resetTask, void (*) 0, (void *) &resetTaskStack [resetTask_StackSize],
resetTask_Priority);
    14. /* Create other main tasks and interprocess communication variables if these must also execute at least
once after the resetTask. */
    15. /* Start SmartOS RTOS to let us RTOS control and run the created tasks */
    SmartOSStart ();
    /* Infinite while-loop exists in each task. So there is never a return from the RTOS function SmartOSStart
(.). RTOS takes the control forever. */
    16. } /* *** End of the Main function *** */
    /* _____ */
    /* The codes of the application reset task that main created. */
    17. static void resetTask (void *taskPointer){
    18. /* Start Timer Ticks for using timer ticks later. */
    SmartOSTickInit (); /* Function for initiating RTCSWT that starts ticks at the configured time in the
SmartOS configuration preprocessor commands in Step 1 */.
    19. /* Create three tasks defined by three task identities, task_ReadPort, task_PW, task_Appl and the stack
sizes, other TCB parameters. */
    SmartOSStaskCreate (task_ReadPort, void (*) 0, (void *) & task_ReadPortStack [task_ReadPortStackSize],
task_ReadPortPriority);
    SmartOSStaskCreate (task_PW, void (*) 0, (void *) & task_PWStack [task_PWStackSize], task_PWPriority);
    SmartOSStaskCreate (task_Appl, void (*) 0, (void *) & task_ApplStack [task_ApplStackSize],
task_ApplPriority);
    /* Declare requestHeader */
    unsigned char [ ] requestHeader;
    unsigned char [ ] requestStart;
    20. while (1) /* Start of the while loop*/
    /* Code for retrieving two strings requestHeader for user PIN and request for host PIN string
from the protected file structure. */
    .
    /* Write the array elements after encryption. */
    ApplStr = SmartOSEncrypt (requestHeader, DES); /* Using an RTOS function encrypt
requestHeader and post it in message queue. */
    SmartOSQPost (MsgQStart, ApplStr);
    *ApplStr = SmartOSEncrypt (requestStart, DES); /* Using an RTOS function encrypt requestStart and
post it in message queue. */
    SmartOSQPost (MsgQStart, ApplStr);
    SmartOSSemPost (SigReset); /* Post Semaphore event flag. */
    21. /* Suspend the Reset task with no resumption later, as it must run once only for initiation of timer ticks

```

```

and for creating the tasks that the scheduler controls by preemption. */
SmartOSTaskSuspend (resetTask_Priority); /*Suspend Reset task and control of the RTOS passes to other
tasks of waiting execution*/
22. /* End of while loop */
23. } /* End of resetTask Codes */
/*****/
24. static void task_ReadPort (void *taskPointer) {
while (1) {
25. /* Wait for IPC from resetTask. */
SmartOSSemPend (SigReset, 0, SemErrPointer);
26. /* Wait for a message for requestHeader from queue MsgQStart. */
&QStart = SmartOSQPend (MsgQStart, 0, QErrPointer);
SendStr (&QStart); /* Send it to transceiver Port_IO. Note: after sending the message in string QStart
becomes null, "". */
27. /* Wait for a message for requestStart from queue MsgQStart. */
&QStart = SmartOSQPend (MsgQStart, 0, QErrPointer);
SendStr (&QStart); /* Send it to modem Port_IO. */
/* Receive and decipher a string from the transceiver Port IO. */
ReceiveStr (&Str);
ApplStr = SmartOSDecrypt (Str, DES); /* Using an RTOS function, decrypt the input string from the host
*/
28. /* Code for saving the Host PIN and, if verified, then application commands from the host is also
saved. The savings are at the protected file structure. */
.
SmartOSSemPost (SemPW); /* Post event flag for requesting a password at MsgQPW. */
SmartOSTimeDly (100); /* Delay for 100 ms to allow lower priority task task_PW run. */
29. /* Wait for a message for requestPassword from queue MsgQPW. If available, send request and
wait for the password. */
&QPW = SmartOSQPend (MsgQPW, 0, QErrPointer);
SendStr (&QPW); /* Send password request to modem Port_IO. */
ReceiveStr (&Str); /* Receive a String from the modem Port IO. */
ApplStr = SmartOSDecrypt (Str, DES); /* Using an RTOS function, decrypt the input string
for password from the host */
30. /* Code for verifying the deciphered user password at the protected memory or file
data. If verified, then application commands from the host by posing event flag
SemAppl. */
.
31. /* Wait for a message for requestAppl from queue MsgQAppl. If available, send request and wait for
the application command and user data. */
&QAppl = SmartOSQPend (MsgQAppl, 0, QErrPointer);
SendStr (&QAppl); /* Send password request to transceiver Port_IO. */
ReceiveStr (&Str); /* Receive a String from the transceiver Port IO. */

```

```

ApplStr = SmartOSDecrypt (Str, DES); /* Using an RTOS function, decrypt the input string for application
command and user data from the host */
32. /* Code for using the user data and executing received application command. */
.
.
33. /* Using an RTOS function, encrypt request closing request and post it in message queue. The closing
request is from a message queue MsgQApplClose. Then retrieve it from queue in QApplClose after
encryption. */
.
.
&QApplClose = SmartOSEncrypt (requestApplClose, DES);
Send (*QApplClose);
ReceiveStr (&Str);
ApplStr = SmartOSDecrypt (Str, DES); /* Using an RTOS function, decrypt the input string for password
from the host. */
/* Compare deciphered string with message "Closure Permitted". If found equal, then closure permitted,
then delete this task and other low priority tasks. */
If (strcmp (ApplStr, "Closure Permitted") {
SmartOS_task_DelEn = 1 /* Enable task deletion by SmartOS. */
OSTaskDel (task_ReadPortPriority); OSTaskDel (task_PWPriority); OSTaskDel (task_ApplPriority);};
} /* End of While loop */
/* End of task_ReadPorts Codes. */
/*****/
40. static void task_PW (void *taskPointer) {
while (1) {
41. /* Wait for IPC from task_ReadPort. */
SmartOSSemPend (SemPW, 0, SemErrPointer);
42. /* Code for retrieving one string from the protected file structure. It is used for requesting the
password from the user at the host of the card. */
.
.
43. /* Write the array elements after encryption. */
ApplStr = SmartOSEncrypt (requestPassword, DES);
SmartOSQPost (MsgQPW, ApplStr);
SmartOSSemPost (SemAppl) SmartSTimeDly (100);
SmartOSTimeDlyResume (task_ReadPortPriority); /* Resume Delayed task task_
ReadPort. */
44. /* End of While loop */
45. /* End of task_PW Codes. */
/*****/
46. static void task_Appl (void *taskPointer) {
while (1) {
SmartOSSemPend (SemAppl, 0, SemErrPointer); /* Wait for IPC from task_ReadPort. */
47. /* Code for retrieving one string for requesting the application commands requestAppl from the
protected file structure. It is for requesting the password from the user at the host of the card. */

```

```

48. /* Write the array elements after encryption. */
   ApplStr = SmartOSEncrypt (requestAppl, DES);
   SmartOSQPost (MsgQStart, ApplStr);
49. /* Resume Delayed task task_PW. */
   SmartOSTimeDlyResume (task_PWPriority);
   /* End of While loop */
50. /* End of task_Appl Codes. */

```

12.5 CASE STUDY OF A MOBILE PHONE SOFTWARE FOR KEY INPUTS

Mobile phones are smart. Each phone has many APIs. Example of APIs are phone, SMS (short message service), MMS (multimedia messaging service), e-mail, address book, web browsing, calendar, task-to-do list, WordPad, Pocket-Word, Pocket-Excel, note-pad for memos, Pocket-PPTs, slide show and camera.

Mobile phone with a large touchscreen uses a virtual keypad. Mobile phone with a small screen uses T9 keypad. The present case study relates to 'SMS create application' in a mobile phone with T9 keypad for inputs.

Section 12.5.1 gives the requirements of 'SMS create and send application'. Section 12.5.2 gives the classes and class diagrams. Section 12.5.3 gives the state diagram and Section 12.5.4 gives communication hardware. Section 12.5.5 describes software architecture. Section 12.5.6 describes the software tasks and Synchronization Model for the application.

12.5.1 Requirements

A processor, keypad, screen, scratch pad memory, persistence memory and communication units are required for SMS create and send application. Scratch pad memory addresses are used for temporary saving of characters (bytes) during the application. Persistence memory addresses are used such that as soon a change is made in the byte, it persists even after the power switches off. Further, when there is a change there, an identical change is reflected in other correlated objects. For example, a name is edited in a file for the Contacts, the same change takes in the file for Address book for sending the e-mails.

Figure 12.20 shows specific units, which are used for the SMS text create application. The screen is used for displaying the menu. Figure 12.20 shows that there are four cursor keys (up, down, left and right) denoted by C1, C2, C3, and C4. In computer keyboard, four different cursor keys are used. The mobile cursor key in the keypad is such that it functions as four keys. When the key is pressed towards the left the cursor moves left (←), when it is pressed towards the right the cursor moves right (→), and so on for up (↑) or down (↓).

In addition there are four command keys (right-corner second-row, left-corner second-row, right-corner first-row and left-corner first-row) denoted by key2Row2, key1Row2, key2Row1 and key1Row1. Also, there are nine T9 keys for numbers 1 to 9 as well as for alphabets a to z (or A to Z). There are two mode-keys (keyM1 and keyM2) and one key for keying in a text character number 0 or space. Alphanumeric text in small case or capital case is controlled by a mode-key's state. Text character entered on keying depends on state of the T9 key. [Recall Examples 3.6 and 6.8 and Section 6.3].

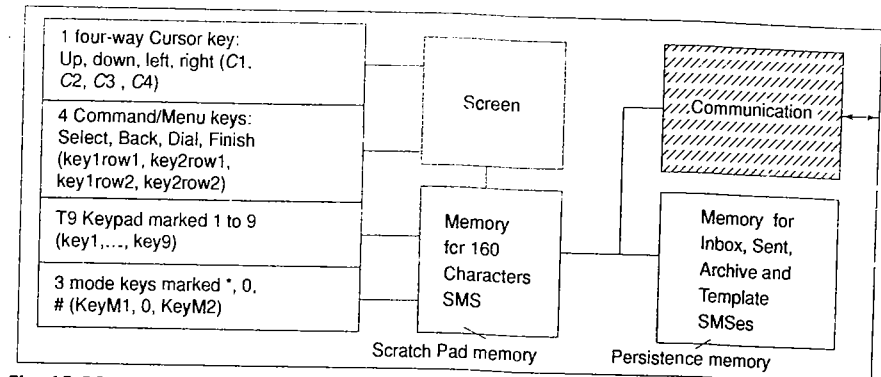


Fig. 12.20 Mobile phone Keypad, screen, memory and communication units for SMSes

Requirements of SMS create and send application module of SMS creation system is tabulated in Table 12.6.

Table 12.6 Requirements of 'SMS create and send application' module in a SMS creation system

Requirement	Description
Purpose	To create an SMS message and communicate using a T9 format keypad and using the tasks for inserting a desired mobile number into a list, editing a message and then sending the message.
Functions of the system	<ol style="list-style-type: none"> An option is selected by using cursor key for moving the cursor displayed on screen and then pressing command key Key1Row1. When a set of options consisting of 4 notifications, one for a command Messages, next for command-option Text Messages, next for application Create Message and last for Text, are selected generates an event E_SMS. [Section 8.4] that signals a task for creation of text messages in alphanumeric format. The characters are entered and edited with T9 keypad of nine keys from numbers 1 to 9, one four-way cursor keys, four command keys, two mode keys marked * and #, and one key marked 0. When any key is clicked by the user, its state is computed based upon the key's earlier state, cursor's present position, timer-status and count, and then a notification E_NewState is posted into a message box for an ISR, task or application that initiates the required action. For choosing the application SMS creating text, the cursor and command-cum-options-select (Key1Row1) keys are used as follows: (a) When command key (Key1Row1) is used to select command Messages, a command-option is then selected using a GUI. (b) User selects Text Messages option by clicking for the displayed option 'Text messages'. (c) A menu then shows up for selecting one of the following using the cursor down and up keys: Create message, Inbox, Sent items, Archive, Templates, My folders, Distribution lists, Delete messages and Message settings. User selects 'Create message' for selecting SMS creation application. (d) A menu then shows up for selecting type of message to be created. Cursor can select one of the following menu items on display: Text and Numeric page. User selects Text for creating SMS text. When option Text is selected then event E_SMS posts (signals) to start execution of 'SMS create and send application' when the user finishes selection of four options Messages, Text Messages, Create Message and Text in sequence.

(Contd)

Requirement	Description
	<p>4. On E_SMS event the tasks <i>Add Number</i>, <i>Edit Message</i> and <i>Send Message</i> execute in order to specify (i) a new or additional mobile number for transmitting the SMS, (ii) editing the SMS-message by keying in from T9 keypad and (iii) transmitting it over the mobile, respectively.</p> <p>5. When mobile is inactive, display screen startup display shows up on key2row2 interrupt, which causes an ISR_key2row2 routine execution and state of mobile becomes wake up state <i>S_Wake</i>. When mobile is in active state <i>S_Active</i> and is running an application, it is brought to idle state <i>S_Idle</i> by at another key2row2 interrupt. Display screen switches off (shuts down) after a preset interval, say, 15 s and the state of mobile becomes sleeping state <i>S_Sleep</i> (only on incoming call there will be port interrupt, and mobile wakes to state <i>S_Wake</i> and screen shows 'start-up display').</p>
GUIs	A GUI uses display of screen menu or text and cursor, command-select and cursor-position-change keys. The cursor position can be changed up, down, left and right by using C1, C2, C3 and C4. The cursor when it points on the screen to a line of menu for a command, it shows that menu item with blue or other background. When the cursor points to a character in a line of text or phone number, it shows a vertical line at the right of the character position. At the cursor position, a text can be selected or entered by a click of command-select key (Key1Row1). At the cursor position, the shown character can be cleared by a click of clear or back key (Key2Row1).
Inputs	<ol style="list-style-type: none"> 1. State of a T9 key (key1, ..., key9). 2. State of mode key (keyM1 and keyM2) pair-marked * and # and that defines the functioning mode. 3. State of key0 (0, 0), (1, space), (1, 0), (1, new-line) or (1, space), when editing an SMS. [State of key0 (0, 0) or (1, 0) when dialling a number.] 4. State for command from one of the four command keys (key1row1, key2row1, key1row2 and key2row2). 5. State of cursor-input on the cursor clicks on move up, down, left or right using C1key, C2key, C3key or C4key.
Signals, events and notifications	<ol style="list-style-type: none"> 1. The E_NewState and message for state of key are posted on the interrupts from command key or any other key. 2. Event E_SMS, which starts SMS_Create_Text application, happens on posting of a set of four GUI notifications - <i>MsgMessages</i>, <i>MsgTextMessages</i>, <i>MsgCreate</i> and <i>MsgText</i>. [<i>MsgMessages</i> notification on selecting option Messages, <i>MsgTextMessages</i> on selecting option Text_messages, <i>MsgCreate</i> on selecting option Create_Message, and <i>MsgText</i> message on selecting option Text.] 3. Notifications for display on completion of tasks. For example, after completing the sending of SMS, display-notification 'Message sent' and before completing the transmission of SMS 'Sending message'. 4. A software timer's time-out interrupt ISR_T_Deactivate switches off the phone at display of idle state (start up menu display in idle state) to the display off and sleep state. [Interrupt is after there is no action for a period longer than a programmed period = 15s.] 5. Another software timer's time-out interrupt ISR_T_Out_Help_Option when a cursor or marked menu is displayed then a pop-up help displays after a period longer than the programmed period 2s.
Outputs	<ol style="list-style-type: none"> 1. SMS_Create_Text string, which is displayed on the screen and is also saved in scratchpad memory during the editing and also saved in sent folder after sending the SMS. 2. Screen menu text lines for displaying option(s), text of menu, marked text or character to enable its selection by clicks. 3. Help menu text display to display action, which will take place on selecting an option after a T_Out_Help_Option interrupt. Option is assumed as one at which cursor points.

(Contd)

Requirement	Description
Scratchpad Memory	The memory is used as scratchpad memory for 160 characters maximum in an SMS.
Persistence memory	The memory is allotted in the system at persistence memory addresses (in flash memory) for SMSes in Inbox, Sent, Archive and Template.
Design metrics	<ol style="list-style-type: none"> 1. <i>Power Dissipation</i>: Battery operation 2. <i>Performance</i>: 3 minute for 1 SMS message creation and send 3. <i>Engineering Cost</i>: US\$ 20000 (assumed) for software 4. <i>Manufacturing Cost</i>: None once the codes are ready and tested
Test and validation conditions	<ol style="list-style-type: none"> 1. All commands and options functioning are tested.

Functioning can be explained in detail as follows:

Command: For *Messages* command, a key (for example) left-hand first-row command key (Key1Row1) is used and user selects the command Messages.

Options: On selection by clicking Key1Row1 when cursor is at a displayed command *Messages*, the command-options display. These options are used for selecting a command-option using the cursor at one of the following command-options: *Text messages*, *Voice Message* and *Minibrowser Messages*.

On selection by clicking Key1Row1 when cursor is at the displayed command-option *Text messages* Text messages option selects. Then one of the following application-options are displayed: *Create message*, *Inbox*, *Sent items*, *Archive*, *Templates*, *My folders*, *Distribution lists*, *Delete messages* and *Message settings*.

SMS Create Application: For selecting SMS create application option. Create message application-option is selected using cursor and click. Then one of the following two type messages can be created, *Text* and *Numeric page* as follows.

Message types: On selection by clicking at the displayed application option *Create message*, the type-options are displayed. These are used for selecting a message type-option using the cursor at one of the following command-options: *Text*, and *Numeric page*. Text is selected as the type-option.

Tasks: On selection by clicking at the displayed type option Text, the task-options are displayed. These are used for selecting task-option using cursor at one of the following task-options: *Add number*, *Add e-mail*, *Add list*, *Edit message*, *List recipient* and *Send*. Add number is selected as the task-option. On return from the task *Add number*, a new task option *Edit message* is selected. On return from the task *Edit message*, next task option *Send* is selected. If message is sent to two numbers, then before the *Send*, a number is added using task *Add number*. [A state diagram will be given in Section 12.5.3 to explain the concurrent processing of the tasks *Add number*, *Add e-mail*, *Add list*, *Edit message*, *List recipient* and *Send*.]

Editing and creating the alphanumeric text SMS: To enable the alphanumeric character entries using just 9 keys, the state of key is changed and notification is issued for the new state on a transition, which causes a new input. A state-transition causing input is applied on when there is a repeat pressing of either the same T9 key again or another key within a prefixed time interval.

This can be explained as follows: Example 3.6 showed how a key marked 5 (5 in first line in large font and jkl in small font in second line marking on key5 surface) on pressing can produce the different states after transition from idle state (0, 5) in sequences represented by (1, 5), (1, j), (1, k), (1, l), (1, 5), (1, j), (1, k), (1, l), (1, 5), till user does not repeatedly press key5 within an interval Δt or presses another key in state (0, n) where n is for any other T9 key.

First character inside bracket in a key-state representation shows whether key is inactive (0) or active (1). There is transition to 1 when a key is pressed and to 0 when its key-state is accepted as the input undergoes transition to idle state. A state is accepted as input if within a time interval Δt either the user presses another key or there is no repeat press of the same key. The second character after comma in a state representation shows the number or text that will be sent in ASCII code format if the active state remains unchanged in a preset time interval Δt and mode M2 key has not modified the mode to text in another language.

Other keys also have similar characteristics. The transition of a key state occurs if it is pressed again within Δt interval. Let us recapitulate Section 6.3, which described a model of state machine. Example 6.8 gave the state table and Example 6.9 gave C codes for the state table.

12.5.2 Class Diagram and Classes

An SMS application program uses the principle of Orchestrator software (Figure 12.2). The inputs from the keys for commands, GUIs and data inputs are taken as equivalent to sensor inputs. A notification or signal issued after each input is taken as equivalent to actuator output. Let us therefore define an Orchestrator class for the application.

Display screen is used during every input and every output in the mobile device. There is a start up display screen. A typical example is screen display, which shows *time* on right hand corner, service company name in the centre, *date* in next line, left side bar for *antenna signal strength*, right side bar for *battery power* and also shows a *start up display graphic*. The *options* are according to provisioning by the service provider and an image for wall-display selected by user. Let us design a Task_ScreenDispl class. It is interfaced to the graphic method. The graphic is chosen from the options to user such as Beach, Car, Coral, Daisy, Dance, Disco, Dragon and others. Task_ScreenDispl extends the other classes, the objects (instances) of which are used in menu item display. GUIs or displaying current action or help text and other display.

SMS creation and send application has a number of concurrent processing tasks. Let us define Task_SMS_CreateTextSend.

Assume that SMS creation application consists of three class diagrams for Orchestrator and two for Task_SMS_CreateTextSend and Task_ScreenDispl. Figure 12.21 shows class diagrams of ORCHESTRATOR, Task_ScreenDispl and Task_SMS_CreateTextSend for creating and communicating SMS message.

1. ORCHESTRATOR class extends to Orchestrator_CommandsGUIs and to Orchestrator_SMSCreateSend.
2. Task_Messages, Task_TextMessages, Task_CreateMessage and Task_Text and interface keypad interrupt ISR_KINT.
3. Task_SMS_CreateTextSend extends to Task_AddNumber, Task_AddEmail, Task_AddList, Task_EditMessage, Task_ListRecipient and Task_Send.
4. Four types of screen displays are used during SMS create and send application. Start up screen display, menu items display, SMS display during editing task and action display during sending the SMS. Task_ScreenDispl thus extends to four classes Task_StartUpDispl, Task_SMSDispl, Task_ActionDispl and Task_MenuTextLinesDispl.
5. The ISRs are ISR_WirelessPort, ISR_T_Out_Help_Option, ISR_T_Deactivate and ISR_KINT.
6. ISR_KINT runs the service functions for any of the state transitions of twenty key and senates notifications for the state of a key Key1Row1 or Key1Row2, Key2Row1, Key2Row2, C1 to C4, M1 or M2 or keys 0 to 9.

Class Figure 12.22 shows Task_MenuTextLinesDispl. It has pixels field of Unsigned byte []. A string is an array of characters. StrLine1, StrLine2, StrLine3 and StrLine4 are the strings in the object of MenuItems. The colour fields are textLineColor, cursorTextLineColor, screenBackgroundColor. The cursor has two fields line and a coloured bar. The methods are OSMBBoxAccept (); OSMBBoxPend (); OSMBBoxPost (); and mouseClicked ().

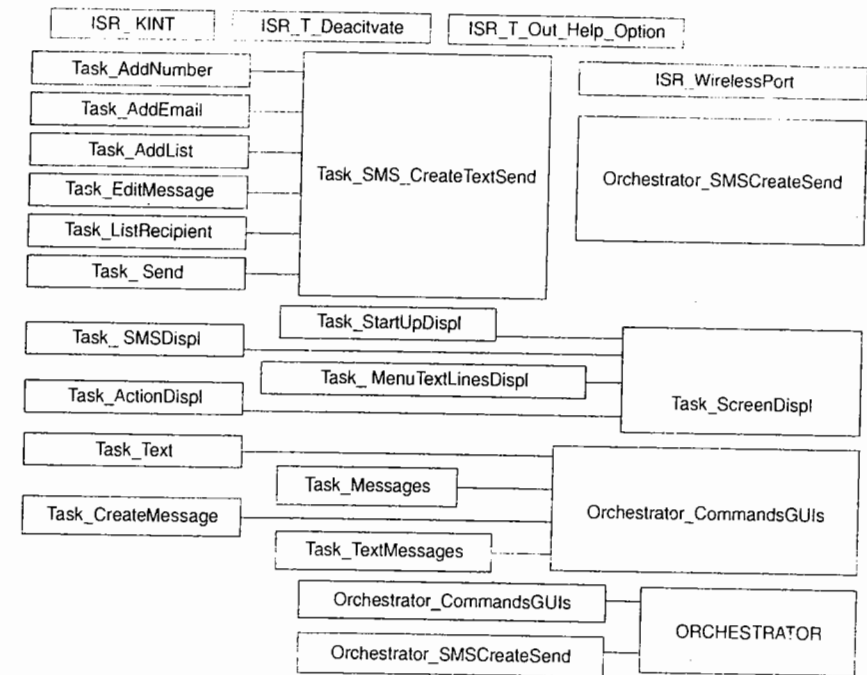


Fig. 12.21 ORCHESTRATOR, Task_SMS_CreateTextSend and Task_ScreenDispl class diagrams of SMS create and send application

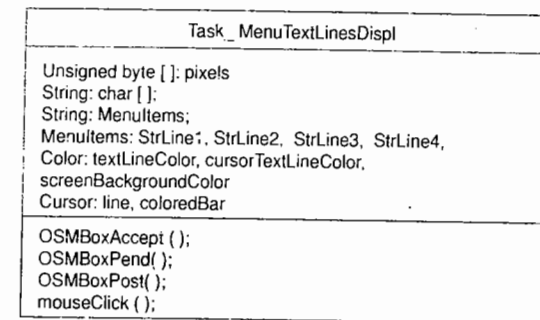


Fig. 12.22 Class Task_MenuTextLinesDispl

Objects Message queue objects are used for posting to the ISRs and tasks. Message queue objects are accepted or waited for at the tasks. For example, *MsgMessages*, *MsgTextMessages*, *MsgCreate* and *MsgText* are posted on Key1Row1 interrupts. Event objects are posted on a set of notifications. For example, *E_SMS* is posted on *MsgMessages*, *MsgTextMessages*, *MsgCreate* and *MsgText*. *E_NewState* is posted on any new state generation on interrupt from any key in the mobile. [Figure 12.20]

12.5.3 State Diagram

Figure 12.23 shows a state diagram for task_SMS_CreateTextSend. A state diagram shows a model of a structure for its start, end, in-between associations through the transitions and shows events-labels (or conditions) with associated transitions. A dark rectangular mark within a circle shows the end. The state transitions take place between the tasks, task_SMS_CreateTextSend and task_AddNumber, task_AddEmail, task_AddList, task_EditMessage, task_ListRecipient and task_Send. A state transition occurs after notification of MsgAddNumber, MsgAddEmail, MsgAddList, MsgEditMessage, MsgListRecipient and MsgSend on selection of menuitems Add Number, Add Email, Add List, Edit Message, List Recipient and Send, respectively. Task_Send posts the event to initiate ISR_WirelessPorts through Orchestrator to send SMS and end the application.

12.5.4 SMS Keying Hardware

Hardware architecture specifies the appropriate decomposition of hardware into processors, ASIPs, keys, memory, ports and devices. It also specifies interfacing and mapping of these components. Specifications for SMS keying-in hardware are as follows:

Cursor key One four-way cursor key, which is pressed to move the cursor up, down or left or right of character when editing the SMS when it is being created. The actions are similar to ↑, ↓, ← and → keys in a keyboard. On cursor-key interrupt on click, the notifications are sent for the states C1key, C2key, C3key and C4key and current cursor display-position.

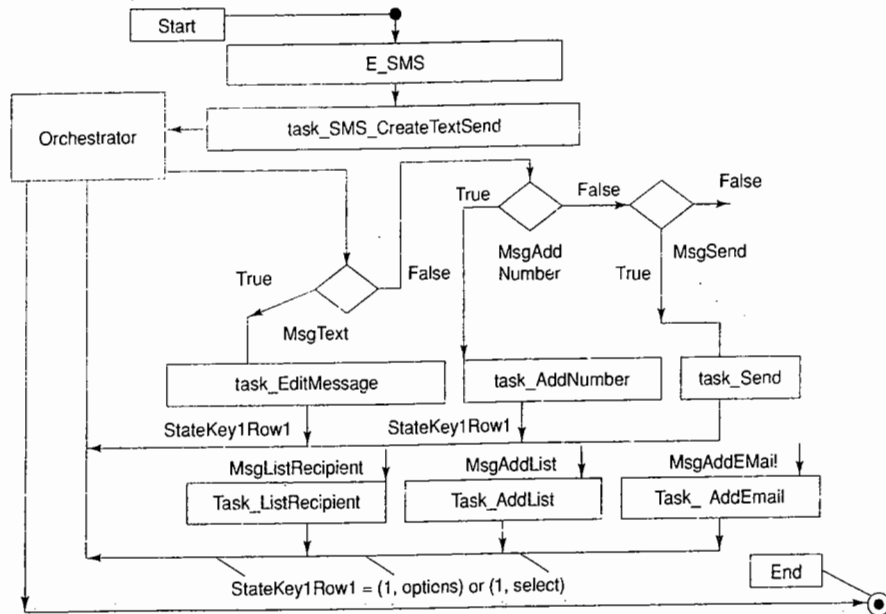


Fig. 12.23 State diagram for task_SMS_CreateTextSend

Command keys Four number command keys, key1row1, key2row2, key1row2 and key2row2 are present. Use of Key1Row1 and Key2Row1 is similar to the left and right clicks in a computer-mouse for GUIs. Use of

Key1Row2 and Key2Row2 clicks is similar to click to start menu item and turn-off or restart menu item, respectively, in a computer start up window.

On clicking a command-key, command-key interrupt routine sends notifications for the commands and their options. The keys are used as follows:

1. Right corner-second row command key (for key2row2 interrupt) is marked red with phone head down sign. It is used to activate the idle device and show start-up display. The same key is also used to switch off an active operation (including a phone call at any instant).
2. Left-corner-second row key (for key1row2 interrupt) is marked green with phone head lifted sign. It is used to dial the number, which is in view on screen or selected using cursor from viewed number (s) in the screen-menu. The same key is also used to receive an incoming call, after a ring tone is played and the number flashes on the screen.
3. Left-corner-first row key (for key1row1 interrupt) is marked green with dash sign and is used to activate all the available options, menus and submenus for starting an application. This key action is similar to left click mice button when the cursor is at a Window for selecting a command from the set of options, buttons and menus.
4. Right-corner-first row key (for key2row1 interrupt) is marked green with dash sign and is used to activate to select the menu commands, and is additionally used or to clear the keyed character during editing process or to go back to previous menu options. For example, the key is used for Contacts, Calculator, Create message and Game or the programmed options for their start. It opens for selecting a command from a limited set of displayed menu options. This key action is similar to action on right click of computer-mouse button when cursor is over a button or at a screen position.

T9 Keys T9 keypad is used for keying in of SMS. T9 keypad has nine keys 1 to 9 plus a key0. T9 key (key1.... key9 key) inputs are used for dialling numbers as well as editing the text inputs during SMS create application.

Mode Keys Two keys marked * and # key modify the states of keyM1 and keyM2 when they are pressed. Mode Key use can be understood by the following example. For example, to protect accidental use of the keys when phone is kept in the pocket, the key1row1 and M1 are simultaneously pressed, the pad undergoes transition to lock state if previously it was in the unlock state and to unlock state if it was in lock state. Another example is bilingual or multilingual text SMS editing. M2 is used to convert the English text mode to other language text mode.

Display Screen Screen displays the GUIs for start up display, menus, options, text being edited or actions currently taking place.

12.5.5 SMS Create and Send Application Software Architecture

Software architecture specifies the appropriate decomposition of software into modules, components, appropriate protection strategies and mapping of software. Software architecture consists of the following in the SMS create and send application.

1. OS. IOS controls the OSMsgQAccept, OSMsgQPost and OSMsgQPend message queue functions at message boxes and event functions at the even boxes. OS synchronizes the tasks and facilitates concurrent processing of SMS application on the mobile.
2. Key-system layer using Orchestrator and ISR_KINTs (interrupt service routines on interrupts from the keys)
3. Application layer

Application layer has

1. Tasks as shown in Figure 12.21
2. ISRs for initiating action on user inputs and GUI notifications

(1) **Key-system layer:** A layer in software architecture is used for the key-system. Figure 12.24 shows a key-system layer. A key click generates an interrupt and a service routine ISR_KINT, which then executes an Orchestrator. The ISR_KINT reads the port status bits to find which key has been clicked, and also to read the timer status and timer counts and cursor position and that position menu or text message. It signals the Orchestrator to initiate and generates the notifications and events and posts these into the message boxes and event boxes for waiting tasks. For example, ISR_KINT initiated Orchestrator posts the following messages:

1. *MsgMessages* when cursor on display screen points to a Command_Msg Messages.
2. *MsgTextMessages* when cursor on display screen points to a command_option_Msg TextMessages.
3. *MsgCreate* when cursor on display screen points to an application-option_Msg point to Create, [Screen displays the application options *Create message, Inbox, Sent items, Archive, Templates, My folders, Distribution lists, Delete messages* and *Message settings* options].

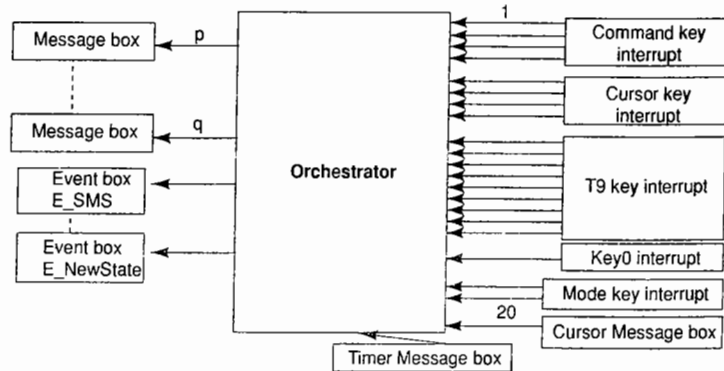


Fig. 12.24 Key-system layer with ISR_KINT and Orchestrator in software architecture of mobile

4. *MsgTextType* when cursor on display screen points to a type_option_Msg Text.
 5. *S_Key0, S_Key1, S_Key2,* or *S_Key9* state as per the timer status and counts if there is new state of key0, key1, key2, ..., key9.
 6. *S_C1, S_C2, S_C3, or S_C4* state if C1 or C2 or C3 or C4 is clicked.
 7. *S_M1* or *S_M2* if M1 or M2 is clicked.
 8. *S_key1Row1, S_key1Row2, S_key1Row2* or *S_key2Row2 S_key1Row1* if a command key is pressed is clicked.
 9. *E_SMS* if command, command option, application option and type options *MsgMessages, MsgTextmessages, MsgCreate* and *MsgTextType* were posted in steps 1, 2, 3 and 4. [*MsgMessages, MsgTextMessages, MsgCreate* and *MsgTextType* initiates event object *E_SMS*.]
 10. *E_NewState* if any state change step 5 or 6 or 7 or 8.
- (2) **Application layer** is a layer in software architecture.

1. An application task is object of class *Task_MenuTextLinesDispl*. It executes on posting of a message-object *MsgTextMessages* or *MsgMessages* or *MsgTextMessages, MsgCreate* or *MsgTextType* into a message box by an ISR_KINT (Fig. 12.24) on *E_NewState*.
2. Another application task is *Task_SMS_CreateTextSend* for SMS create and send application and it executes on event *E_SMS*. State diagram of it was shown in Figure 12.23 Section 12.5.3.

12.5.6 Software Tasks and Synchronization Model

Figure 12.23 showed the state diagram of synchronizing cycle of different tasks. The SMS communication system has a cycle of actions and tasks synchronizing model. Orchestrator posts notifications to the *task_Messages, task_TextMessages, task_CreateMessage* and *task_Text*.

1. A cycle starts in Orchestrator. A task, *task_Messages*, which receives a notification by message *S_Key1Row1* choice of command. It posts *MsgMessages*.
2. A task *task_TextMessages* is for command option. It posts message *MsgTextMessages* on user selection. Another task *task_CreateMessage* accepts *MsgTextMessages* and posts *MsgCreate* on user selection. Another task *task_Text* accepts *MsgCreate* and posts *MsgText* on user selection.
3. Orchestrator then posts *E_SMS*. On receiving *E_SMS* the *Task_SMS_CreateTextSend* signals the displaying of menu items for initiating *task_AddNumber, task_EditMessage* and *task_Send, task_AddEmail* and *task_ListRecipient*.
4. Task *task_AddNumber* adds the message mobile number for sending the SMS in a list. It posts message *MsgAddNumber* on user selection.
5. On user selection, Orchestrator posts message, which signals another task *task_EditMessage* to start. It is used for creating and editing the message by keying-in the characters. Orchestrator accepts state of key on each key click and posts state in the message box of the key clicked during the editing. *task_EditMessage* accepts state message of the keys and creates the *SMS_Create_Text* string.
6. On user selection, Orchestrator posts message, which signals another task *task_Send*.
7. *Task_Send* posts *MsgSend*. Orchestrator accepts *MsgSend* and posts *MsgCommunication* for Communication Port Interface. It accepts *MsgCommunication* and posts *SMS_Create_Text* string through wireless.

Figure 12.25 shows a Synchronization model for SMS create and application tasks. ISRs and Orchestrator tasks (*task_Addlist, Task_ListRecipient* and *task_AddEmail* synchronization not shown) using the message and event boxes.

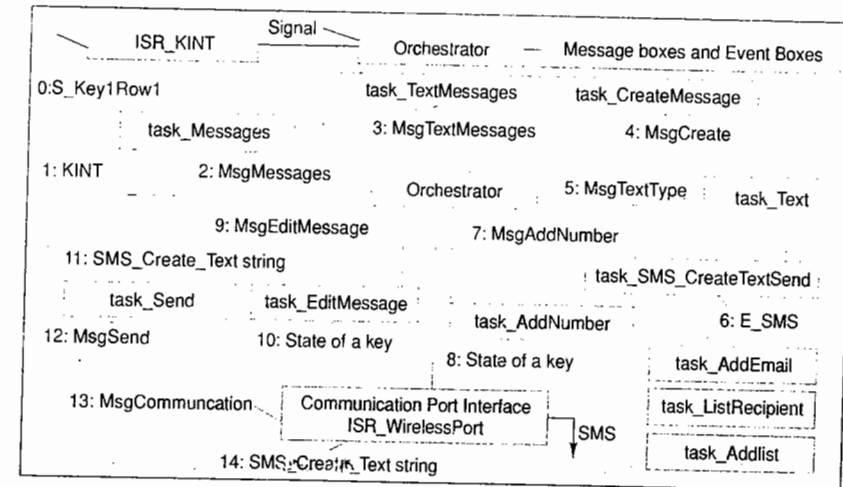


Fig. 12.25 Synchronization model for SMS create and application tasks, ISRs and Orchestrator tasks (*task_Addlist, Task_ListRecipient* and *task_AddEmail* synchronization not shown)



Summary

The following is a summary of this chapter

- Four case studies are explained: system design for the robot orchestra, automatic cruise control system, smart card and mobile phone SMS create and send application. The system design by software engineering and UML modeling approaches are explained in these case studies.
- Class diagrams, classes, state diagram, synchronization model, hardware architecture and software architecture are described using examples.
- Robot orchestra example explains the synchronization of MIDI messages between conductor and players.
- Orchestrator is software which sequences, synchronizes the inputs from 1st to kth sensors or other sources and generates messages, notifications, signals and outputs for the actuators, display and message boxes at specified instances and time intervals after an input change. Message boxes store the notifications, that initiate the tasks. Application of Orchestrator is explained by taking examples of the robot orchestra and mobile phone SMS create and send application.
- There are many automobile electronics applications of embedded systems and the important ones were summarized here. A feature in modern cars, automobile cruise control system, was described. The adaptive control system is defined. Due to the greater need of reliability from the point of view of human safety, the need for special features in OS for automobiles was explained. MISRA-C standard for C language software defines the guidelines for automotive systems for using C. MISRA-C version 2 (2004) specifies 141 rules for coding and gave a new structure for C. A standard RTOS is OSEK-OS for automotive electronics. *Automotive cruise control* system code design is given using the VxWorks after retaining and incorporating the OSEK-OS features with it during the exemplary coding.
- Embedded system design for card-host communication in a smart card is given to explain special embedded hardware and special RTOS functions needs. The case study showed that for developing codes for an embedded system, the RTOS MUCOS or VxWorks functions might not suffice. A hypothetical RTOS, SmartOS, which has MUCOS features plus the embedded system special OS functions when it requires cryptographic features and file security, access conditions and restricted access permissions. After an initialization task that executes on system booting, three tasks are scheduled by the SmartOS. (a) A task reads the application strings from the card data files. It sends, after encryption, the messages to UART. It receives the encrypted strings from the UART of the host. This example also shows how multiple functions can be handled by the same task to reduce the memory needs. It is a desired feature in the smart card case. There is only 8 kB in most cases and 64 kB in extreme cases. The task for the password as well applications interacts through the IPCs. In the end, after seeking host authorization, the tasks are deleted. (b) A task sends the password request from the user interacting with the host. (c) One task gets the commands for executing the desired application routines and user data from the host.
- Embedded system design for SMS message creation and communication is described. The example explained the uses of command keys, cursor keys and GUIs in mobile phone SMS create and send applications. It is shown that the concept of Orchestrator, used in robot applications, is also useful for mobile phone applications.



Keywords and their Definitions

- Adaptive Algorithm** : An algorithm that adjusts and adapts to the parameters and limits the changing perturbations in a control system.
- Adaptive Control** : A control system that uses an adaptive algorithm to generate output control signals.
- Adaptive Cruise Control** : An automobile throttle control system to maintain constant preset cruising speed.

Design Examples and Case Studies of Program Modeling and Programming

- Bluetooth** : A protocol used in mobile devices with features like self-organizing network, self-establishing and self-maintaining. It uses COM port of PC and other features and is used for communication between the devices within a picometer.
- Charge Pump** : A combination of diode and capacitor to extract and store a supply to the system using an appropriate voltage.
- COM port** : A port in PC, which is used for connecting to mice or keyboards or 25 pin RS232C standard serial or parallel connectors. It uses handshaking signals standard specified by UART. Emulation of the RxD (received data) and TxD (transmitted) data in serial protocol as in 11-bit UART (one start bit, 8 data bits, parity bit and stop bit). (Section 3.2)
- Command key** : A key used to select menu item options to start a new task or application.
- Cryptographic Software** : Software for encrypting and deciphering a message or a set of bytes. It uses an algorithm for encrypting and another algorithm for deciphering.
- Cursor** : A line or symbol or icon displayed on the screen to guide a user to select a button or text shown at that position.
- Cursor key** : A key, when pressed towards left moves a cursor to the left, right moves the cursor to the right, and so also up and down.
- Data Acquisition System** : A system to acquire data from multiple ports and channels.
- Event** : An instance of presence of a notification or message or set of messages or action(s) that initiates an ISR. There is reset of the notification on start of ISR to enable response to next event.
- Fabrication Key** : A key embedded in ROM at the time of card fabrication so that the card gets an unique identity.
- GPS (Global positioning system)** : A system for determining *locations, speed, direction* and *time* by a receiver. A set of 24 or more medium earth orbit satellites beams the signals to enable a GPS receiver. The receiver is positioned at any place on globe to receive signals for determining these four parameters.
- Invalidation Lock** : A lock, which, if placed in the application data files in the card, makes the card invalid for further use.
- Java Card** : A Java language format for smart card applications.
- JVM (Java Virtual Machine)** : The supervisory codes that execute the Java classes compiled as byte codes by the Java compiler. The codes run with the help of JVM in a computer system.
- Logical Address** : A memory address used for an instruction or data byte of the RTOS or application.
- Message box** : A mailbox in which notification(s) are placed by a task and another task expects the message or waits for the message.
- MIDI (musical instruments digital interface)** : A protocol to define the specification required for hardware interfaces, message formats and for sending the program change, system and channel messages. The channel messages define the musical *note, pitch-bend, control change, program change* and *after-touch* (poly-pressure) messages.
- Notification** : A message generated on *listening* to clicking a key or on a change of *state* of a key or clicking of a button or selection of a menu item. Notification is for another task.
- Orchestra** : A musical event played using several musical instruments, each with a player and conducted by a conductor.
- Orchestrator** : Software, which sequences and synchronizes the inputs from the 1st to the kth source and generates the messages and outputs for the actuators, display and message boxes at the specified instances and time intervals. Message boxes store the notifications that initiate the events and tasks as per the notifications.

- Personalisation Key** : A key placed after testing the smart card circuit. The card is personalized for its own protected area of memory and own translation scheme for conversion between physical and logical addresses during actual running of the tasks at the card. After insertion of this key, the RTOS and application use only the logical addresses, and the processor uses this key during the translation between two addresses.
- Physical address** : The address used by the processor to fetch the data or send the data to memory and ports.
- Piconet** : A Bluetooth network with the devices within a distance of about 10 m.
- PIN** : Personal Identification number. Bank or hosting service allocates this PIN. The allocation unit has its own PIN, called host PIN.
- Protection Bit** : A bit at the ROM that the processor uses for not letting the transfer of instructions and data in the protected part to the system external buses. The processor externally blocks the write cycles for accessing these protected addresses.
- Qrio** : An invention of Sony meaning **Q**uest for **c**uriosity, which gave a smoother and faster humanoid robot than ever before in 2003, weighed 7.3 kg and was 58 cm, and had one hour battery. A set of the Qrios also played the orchestra and dance numbers.
- Radar (Radio Detection and Ranging)** : A system that uses radio waves of below 1 m to enable ranging of short distant objects by measuring time delay between transmitted signal and reflected signal.
- RSA** : An algorithm that uses the prime numbers. RSA stands for the first letters of the last names of its three inventors: Ron Rivest, Adi Shamir, Leonard Adleman.
- Scatternet** : A network within 100 m between various piconets connected through a Bluetooth-enabled bridging device and formed by self discovery and self organizing features of the Bluetooth protocol.
- SHA** : A Security Hash Algorithm based on a hashing function.
- State** : A state of parameters that results after an input and that depends upon the previous sequences of states which occurred since starting from initial state. For example, a counter has a different state after a count input and its state depends on the previous number of inputs. State can also depend on time if time-out interrupt is used as state-transition input. A key has on-off states or if clicked one, two, three times then multiple states. For example the states of the T9 keys.
- String Stability of Vehicles** : Stability by maintaining constant distance between multiple streaming vehicles in a convoy on highway or VIP duty.
- T9 Keypad** : A keypad that includes nine T9 keys for nine numbers 1 to 9 as well as for alphabets a to z (or A to Z). Entered alphanumeric text in small case or capital case is per the as per a mode-key state. Text character depends on state of the T9 key.
- Throttle Valve** : A valve to control the engine thrust and hence acceleration.
- Unblock PIN** : A host PIN used for unblocking a certain part of the card memory for using. For example, for permitting a modification of the user password after unblocking (permitting access by modifying the access condition fields) password file in the memory.



Review Questions

- How are the MIDI messages used for conducting an orchestra? Give specific examples.
- List the tasks required for MIDI file communication using Bluetooth piconet between the players and conductor robots. Explain task priority assignments in robot orchestra MIDI messages communication.
- List the embedded devices in a high-end car.
- What is adaptive control? How does adaptive control algorithm differ from feedback proportional control.
- List the tasks and ISRs required for ACC system. Explain the actions of each. Explain task priority assignments in an ACC system.
- What should be features in OS for automobile applications? Why should a MISRA-C version of C be used in ACC tasks?
- What are the Bluetooth device piconet and scatternet ranges? Discuss the advantages and disadvantages of Bluetooth based inter car communication in place of radar- or laser-based communication?
- Why is Java popular for smart card applications?
- How does a contactless smart card hardware derive power?
- Why is the use of a processor with memory protection bit essential?
- What is the advantage of encryption when using a fabrication key, personalization key, utilization lock and PIN?
- Tabulate the features needed in the OS for a smart card.
- Explain how the task of reading ports in smart card synchronizes with the port device driver.
- List the tasks and ISRs required for a smart card system. Explain the actions of each. Explain task priority assignments in a smart system.
- List the tasks and ISRs required for mobile phone SMS create and send application. Explain the actions of each. Explain task priority assignments in an SMS create and send application of mobile phone.
- We can use a number of mailbox IPC messages from a task in a mobile phone. Explain how this has been effectively used. Why is the use of mailbox option followed in place of message queue in a mobile phone SMS create and send application?



Practice Exercises

- What are the list of tasks for dancing robots.
- Give the list of tasks for the ACC with string stability among 4 cars.
- List classes for host of smart card used in the Bank ATM.
- Draw Class diagrams of SMS Inbox messages read application.
- Write the sequences and state diagrams of key pressing event in T9 keys. Assume that key5 has state sequences s on transitions from idle state (0, 5) (1, 5), (1, j), (1, k), (1, l), (1, 5), (1, j), (1, k), (1, l), (1, 5), till within an interval Δt user does not repeat pressing of key5 or presses another key in state (0, n) where n is any other T9 key. What will be the state sequences for key1, key2, key3, key4, key6, key7, key8 and key9.
- List the classes that extend the Task_ScreenDispl in a mobile phone.
- Give a synchronization model for editing and string creation using Task_EditMessage during creating an SMS message.
- Design the codes for SMS text editing using Task_EditMessage.
- List classes that will be used for start up display, display off after 15s and pop-ups a help-display if cursor stays at a menu item for more than 2s.

Embedded Software Development Process and Tools

13

R

Recall chapter 1 afresh. We defined an embedded system as one that has software embedded into a computer hardware. Embedded system has three main components.

- Hardware
- Main application software. Application software perform multiple tasks
- RTOS

In the previous chapters, we have learnt all the three main components of embedded systems and have covered following topics.

- Embedded system hardware consisting of processor, memory, devices and basic hardware units – power supply, clock circuit and reset circuit.
- Devices consisting of I/O ports to access the peripheral and other on-chip or off-chip physical-devices. Physical-device examples are UART, modem, transceiver, timer-counter, keypad, keyboard, LED display, LCD display, DAC, ADC and pulse-dialer.

e

c

a

l

l

L

E

A

R

N

I

N

G

O

B

J

E

C

T

I

V

E

S

- Real-time clock-driven software timers.
- Virtual devices (pipe, socket, file, etc).
- Device drivers and interrupt-handling mechanism in an embedded system.
- Need for power dissipation management by the processor instructions during high-speed computations.
- Selection of appropriate processor, memory and devices for optimum system performance.
- Interfacing of system buses with the memory and I/O devices, and use of DMA controller to improve system performance by enabling the I/O units to have direct access to system memory.
- High-level language programming concepts, program models and software-engineering approaches.
- RTOS, use of IPCs, exemplary uses of RTOS MUCOS (μ C/OS-II) and VxWorks functions, and study of MUCOS, VxWorks, Windows CE, OSEK and Real Time Linux programming environments and their applications.

Chapters between 5 and 12 focussed on software aspects. In this chapter, we focus on hardware and software integration aspect. We will learn the following for understanding embedded development process and tools.

1. Software, source code engineering and integrated development environment (IDE) tools.
2. Software is developed on a host machine, say, a personal computer for a target system, which in most cases uses distinct processor and OS. There are two development platforms: host and target machines.
3. Linking and locator to create file for binary image for the final software.
4. Device programmer to burn the codes in the PROM or flash burning of system monitor codes in ROM.
5. Issues in embedded system development that need to be addressed by any development team. These are independent software-hardware design, hardware-software co-design, choosing right processor, allocation of memory addresses, devices and bus and porting issues of OS/RTOS.

Chapter 14 will describe testing and debugging.

13.1 INTRODUCTION TO EMBEDDED SOFTWARE DEVELOPMENT PROCESS AND TOOLS

13.1.1 Development Process and Hardware-Software

Figure 13.1(a) shows the development process of an embedded system and Figure 13.1(b) edit-test-debug cycle during implementation phase of the development process. There are cycles of editing-testing-debugging during the development phases. Whereas the processor part once chosen remains fixed, the application software codes have to be perfected by a number of runs and tests. Whereas the cost of the processor is quite small, the cost of developing a final targeted system is quite high and needs a larger time frame than the hardware circuit design.

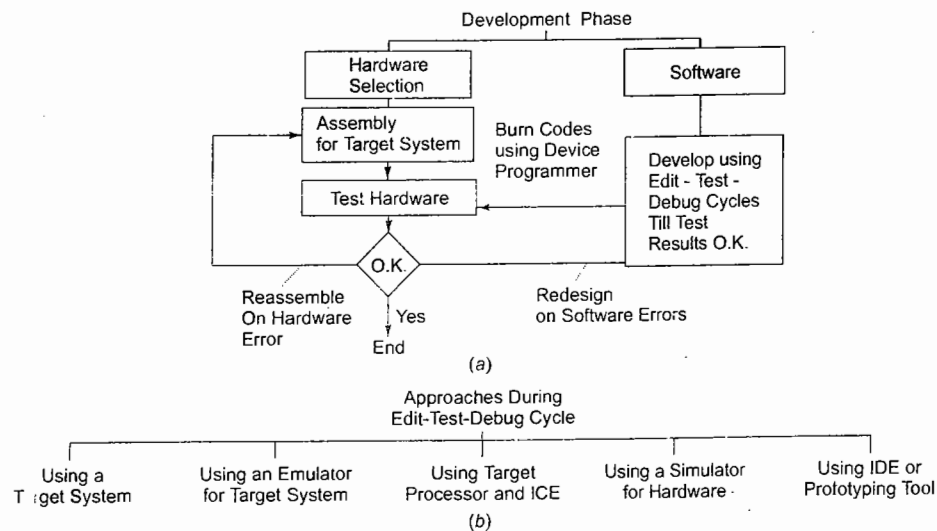


Fig. 13.1 (a) Development process of an embedded system (b) Edit-test-debug cycle during the implementation phase of the development process

The developer uses four main approaches to the edit-test-debug cycles.

1. An IDE or prototype tool (Refer to Section 13.1.4).
2. A simulator without any hardware (Refer to Section 14.2).
3. Processor only at the target system and uses an in-between ICE (in-circuit-emulator) (Refer to Section 14.3.6).
4. Target system at the last stage.

13.1.2 Software Tools

The tools are required for the application software high-level language programming. Also required are the RTOS, testing debugging, assembly language programming (for implementing the device-driver functions)

and system integration tools. Table 13.1 lists the software tools in software and hardware implementation for embedded system.

Table 13.1 Software Modules and Tools for implementation of an Embedded System

Software Tools	Application
Development kit	Development kit is used for editing, configuring (disabling and enabling the C++ features), GUIs development and compiling.
Source-code engineering software	Source code engineering tool is used for editing, configuring (e.g., disabling and enabling the C++ features), GUIs development and compiling as well as for source code comprehension, navigation and browsing, and debugging (Section 13.1.3).
RTOS	An operating system (OS) for multitasking, process, memory, IO, network, devices, file system and for real-time control of processes.
Integrated development environment	Software and hardware environment that consists of simulators, editors, compilers, assemblers, RTOS, debuggers, stethoscope, tracer, emulators, logic analysers, application codes' burners for the integrated development of a system (Section 13.1.1).
Prototyper	For simulating, source code engineering including compiling, debugging and navigating through the codes using a browser, summarizing complete status of final target system during the development phase. Tornado prototyper from WindRiver® for integrated cross-development environment with a set of tools (Section 14.2.4).
Compiler	For using the complete set of the codes, functions, expressions and library routines and creating a file called object file.
Assembler	For translating the assembly mnemonics into binary opcodes (instructions), that is, into an executable file, called binary file. It also creates a list file that can be printed. The list file has address, source code (assembly language mnemonic) and hexadecimal object codes. The file has addresses which are allocated again during the actual run of assembly language program.
Cross-assembler	For converting object codes or executable codes for a processor at development system to other codes for another processor for embedded system and vice versa.
Cross-compiler	For compiling source codes for a another processor and vice versa.
Testing and debugging tools	Simulator for simulating most functions of a target embedded system circuit including additional memory, peripherals and buses on the host system itself (Section 14.2.3); stethoscope for dynamically tracking the changes in any program variable; trace scope for tracing the changes in the modules and tasks as a function of time on the X-axis; memoscope for memory usage which is a critical aspect of an embedded system; ScopeProfile to find in which task the CPU spends how many of its cycles in order to understand performance bottlenecks; in-circuit emulator (Section 14.3.6); monitor (Section 14.3.7).
Locator	Uses cross-assembler output and a memory allocation map and provides the locator-program's output (Section 13.3).
Editor	For writing C codes or assembly mnemonics using the keyboard of host system (PC) for entering the program. Allows the entry, addition, deletion, insertion appending previously written lines or files, merging record and files at the specific positions. Creates a source file that stores the edited file. That has an appropriate name.
Interpreter	For expression-by-expression (line-by-line) translation to the machine executable codes.

Software tools are used to develop software for designing an embedded system. Sophisticated tools—Integrated development environment and prototype development tools—are needed for integrated development of system software and hardware. The testing and debugging tools are needed for testing and debugging.

13.1.3 Source Code Engineering Tool

A source code engineering tool is of great help for source code development, compiling and cross-compiling. The tools are commercially available for embedded C/C++ code engineering, testing and debugging.

The features of a typical tool are comprehension, navigation and browsing, editing, debugging, configuring (disabling and enabling the C++ features) and compiling. A tool for C and C++ is SNIFF+. It is from WindRiver® Systems. A version, SNIFF+ PRO has full SNIFF+ code as well as debug module. Main features of the tool are as follows:

1. It searches and lists the definitions, symbols, hierarchy of the classes and class inheritance trees. [The symbols include the class members. A tree is a data structure. A data structure tree has a root. From the roots, the branches emerge and from the branches more branches emerge. On the branches, finally there are the leaves (terminating nodes).]
2. It searches and lists the dependencies of symbols and defined symbols, variables, functions (methods) and other symbols.
3. It monitors, enables and disables the implementation virtual functions. Use of virtual functions is for dynamic run-time binding.
4. It finds the complete effect of any code change on the source code.
5. It searches and lists the dependencies and hierarchy of the included header files.
6. It navigates to and fro between the implementation and symbol declaration.
7. It navigates to and fro between the over-ridden and over-riding methods. (Overriding method is a method in a daughter class with the same name and number and types of arguments as in the parent class. Over-ridden method is the method of the parent class, which has been redefined at the daughter class.)
8. It browses through information regarding instantiation (object creation) of a class.
9. It browses through the encapsulation of variables among the members and browses through the public, private and protected visibility of the members.
10. It browses through object component relationships.
11. It automatically removes error-prone and unused tasks.
12. It provides easy and automated search and replacement.

The embedded software programmer for sophisticated applications uses a source code engineering tool for program coding, profiling, testing and debugging of embedded system software.

13.1.4 Integrated Development Environment (IDE)

IDE consists of simulators with editors, compilers, assemblers, etc., emulators, logic analysers and EPROM/EEPROM application codes burner. An IDE must have the following features.

1. It has a facility for defining a processor family as well as defining its version. It has source code engineering tools (Section 13.1.3) which incorporate the editor, compiler for C, embedded C++ assembler, linker, locator, logic analyser, stethoscope and 'Help'.
2. It has the facility of a user-definable assembler to support a new version or type of processor. It provides a multiuser environment.

3. The design process divides into number of subparts. Each programmer is assigned independent but linked tasks.
4. It simulates hardware unit-like emulator, peripherals and I/O devices on a host system (PC). It supports conditional and unconditional breakpoints. It provides test-vectors. A test-vector is program-path for the controlled flow of the program used during testing phase and later removed or disabled on completing that phase.
5. It debugs by single stepping. It has the facility for synchronizing the internal peripherals.
6. It provides Windows on the screen. These provide the detailed information of the source code part with labels and symbolic arguments, the registers as the execution continues, the detailed information of the status of peripheral devices, status of RAM and ports, and the status of stack and program flow as it continues.
7. It verifies the performance of a target system. It has an emulator built into the development system that remains independent of a particular targeted system, plus a logic analyser for up to 256 or 512 transactions on the address and data buses after triggering.

An IDE tool is from WindRiver® Systems and that employs VxWorks RTOS (Section 9.3). An architectural feature is 'dynamic linking and incrementally loading the object modules' into the target system. Exemplary target processor families that are supported are PowerPC, Intel, Motorola, Pentiums, MIPS and ARM/Strong ARM. It helps in prototype development and tests the prototype applications. There is a text editor with GNU C/C++ compilers. Debugging is performed at three levels, source code-level, task-level (scheduling, IPCs and interrupts study) and domain-level. It includes VxSim, stethoscope and tracescope. Figure 13.2(a) and (b) show simple and sophisticated IDE, respectively.

An IDE (μ Vision_2) is from keil software Inc. with RTX51 RTOS for 8051 target processor families. Another IDE keil μ Vision_3 is for ARM family of processors and microcontrollers. It has cross-compiler, source-level debugger, object browser, monitor for run-time behaviour, event-to-event viewing. The object browser browses the applications behaviour overtime. It graphically displays the RTOS tasks, queues, semaphores and IPC objects. A real-time analysis (RTA) suite profiles the code coverage and locates run-time errors. It optimizes the use of the memory.

13.2 HOST AND TARGET MACHINES

During the development process, a host system is used before locating and burning the codes in the target board. The target board hardware and software is later copied to get the final embedded system, which will function exactly as the one tested and debugged and finalized during the development process.

13.2.1 Using a Host System

Host system is a PC or workstation or laptop. It has the following hardwares.

1. High-performance processor with caches
2. Large RAM memory
3. ROMBIOS (read only memory basic input-output system)
4. Very large memory on disk
5. Keyboard
6. Display monitor
7. Mice
8. Network connection

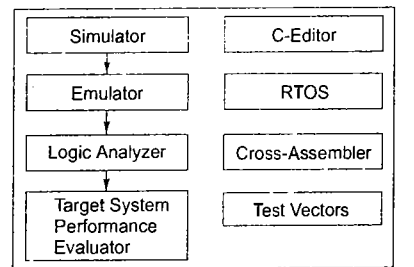
It a full-fledged computer. It has software tools (Table 13.1) and must include the following:

1. Program development kit for a high-level language program or IDE.
2. Host processor compiler and cross-compiler.
3. Cross-assembler.

Program Development Tool Kit Program development tool kit or IDE has an editor. The editor is used for writing C codes or assembly mnemonics or C++ or Java or Visual C++ using the keyboard of the host system (PC) for entering the program. Using GUIs, it allows the entry, addition, deletion, insert, appending previously written lines or files, merging record and files at the specific positions. It creates a source file that stores the edited file. It also has an appropriate name (given by the programmer). It can use previously created files and can also integrate the various source files. It can save different versions of the source files. Program development kit or IDE has the code generation tools (assembler, compiler, loader and linker).

A high-level language is machine-independent. It will have an expression like $X = X + 23$, or $X = 2 * Y + V * Z + 19$ and so on. When we use a high-level language C, a tool is needed for obtaining the machine codes for a target system. The programmer writes the mnemonics or C program, using the editor. The mice and keyboard combinations of the host system (PC) or host system are for entering the program codes. Each language needs a compiler. The codes may not be executable using an interpreter.

1. An *interpreter* does expression-by-expression (line-by-line) translation to the machine-executable codes.
2. A *compiler* uses the complete set of the expressions. It may also include the expressions from the library routines; that is, standard tailor-made programs. Whereas an interpreter helps in on-line execution of the codes, a compiler helps in the off-line programming for obtaining the executable machine codes later. The C programs are used with an interpreter as well as with a compiler. A cross-compiler is a compiler that created binary executable files for the target system processor.
3. An assembly language program has the mnemonics that are machine-dependent. Example of a mnemonic is SBC A, 0x0B. It means an instruction, which subtracts, along with the previous 'carry', the A register of the processor with the hexadecimal number 0x0B. An assembly mnemonic is specific to a processor or microcontroller. It is according to the instructions provided in the instruction set. The assembly mnemonics needs an interpreter to translate into the machine codes that are executed on a specific processing device.
4. A *disassembler* translates the object codes into the mnemonics form of assembly language. It helps in understanding the previously made object codes.
5. An *assembler* is a program that translates the assembly mnemonics into the binary opcodes and instructions, that is, into an executable file, called object file. It also creates a list file that can be printed. The list file has address, source code (assembly language mnemonic) and object codes in hexadecimal. The object file has addresses that are to be allocated again during actual run of the assembly language program. A loader is a program that helps in this task by reallocating addresses before loading the opcode and operands in the computer memory.



IDE for Various types and Versions of Microcontroller with Upgradability of IDE for future Versions.

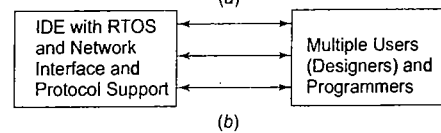


Fig. 13.2 (a) Simple integrated development environment (IDE) (b) Sophisticated IDE

6. A *linker* links the needed object code files and library code files. This is before the *loader* reallocates the addresses, and puts the codes at the physical addresses in the memory, and the program runs. Loader performs the analogous functions on host machine as the locator does on a target system in conjunction with a device programmer.

Cross-Compiler C or C++ or visual C++ source files compile according to the native platform (system including OS on which their binary image runs). Java classes compile as byte codes and are therefore platform-independent. A cross-compiler is a compiler that creates binary executable files for the target system processor.

Cross-Assembler It converts object codes or executable codes for a processor to other codes for another processor and vice versa. The cross-assembler assembles the assembly codes of the target processor as the assembly codes of the lets us use a processor of the host system (PC) used in the system development. Later, it provides the object codes for the target processor. These codes will be the ones actually needed in the finally developed system.

Code generation tools are used for creating and compiling at the host system. Then codes are tested at the host system using simulators and number of latest software tools like profiler, memory scope, stethoscope and memory and code coverage scope.

13.2.2 Target System

A target system has a processor, ROM memory for ROM image of the embedded software, RAM for stack, temporary variables and memory buffers, peripherals and interfaces. Figure 13.3(a) and (b) show simple and sophisticated target systems, respectively. Some target systems have 8 or 16 MB flash memory and 64 MB SDRAM. A target system may possess the RS232 as well as 10/100-base Ethernet connectivity or USB port.

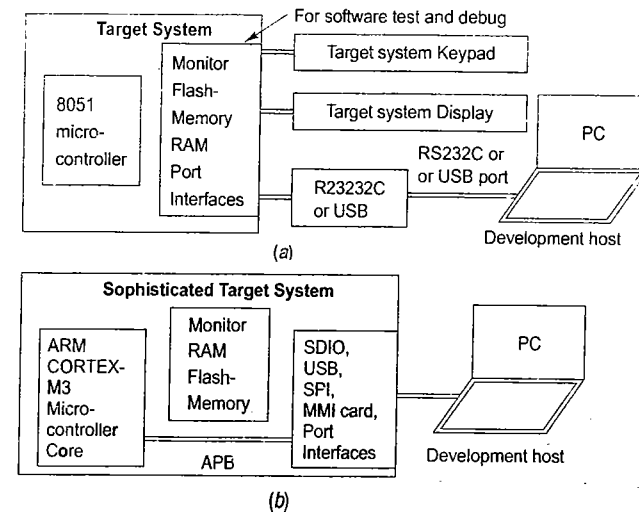


Fig. 13.3 (a) Simple target system (b) Sophisticated target system

A target system differs from a final system. It interfaces with the computer as well works as a standalone system. There might be repeated downloading of the codes into it during the development phase. The target system or its copies simply work later as the embedded system.

Consider that a targeted system is under development. In the target system development phase, say of a *router*, the codes of application software have to be written. These have to be embedded in flash. These have to be repeatedly written or modified and tested using diagnostic, simulation and debugging tools, and embedded in a final testing in an edit-test-debug cycle shows it working according to specifications. The programmer iterates on simply copies it into the final system or product. Also a final system may use a ROM in place of flash in the target system.

An exemplary target system is a board that has an Philips LPC21xx processor (ARM microcontroller). It is an IC2100 evaluation board from keil.

Let us consider an exemplary sophisticated target system, VxWorks 5.4. It provides run-time support by a real-time OS support, Internet protocols support, POSIX library support, file system and graphic supports. It has a debugging agent. It has a back end support package for a specific processor or microcontroller. The target system connects the simulator in parallel the host computer through a target server tool with ICE (Section 14.3.5) using Ethernet or serial lines from the host computer.

13.3 LINKING AND LOCATING SOFTWARE

A *linker* links the compiled codes of application software, object codes from library and OS kernel. Linking is necessary because there are number of codes to be linked for the final binary file. For example, there are standard codes to program a delay task for which there is a reference in the assembly language program. The codes for the delay must link with the assembled codes. The delay code is sequential from a certain beginning address. The assembly software code is also sequential from a certain beginning address. Both the codes are present at the distinct and the available addresses in the system. A linker links these. The linked file in binary format on a computer is commonly known as executable file or simply '.exe' file. After linking, there has to be allocation of the sequences of placing the codes before the actual placement of the codes in the memory.

A program is loaded in a computer RAM. The *loader* program performs the task of *reallocating* the codes by finding the physical memory addresses available at a given instant. The loader is a part of the OS and places codes into the memory after reading the '.exe' file. This step is necessary because the available memory addresses may not start from 0x0000, and binary codes have to be loaded at the different addresses during the run. The loader finds the appropriate start address. In a computer, after the loader loads into a section of RAM, the program is ready to run.

When the code embeds into ROM or flash, a system design process *locates* these codes as a ROM image. The codes are permanently placed at the actually available addresses in flash-ROM. In embedded systems, there is no separate program to keep track of the available addresses at different times during the run as in a computer. In embedded systems, therefore next step after linking is the use of a *locator* for the program-codes and data in place of the loader. The locator features are as follows.

1. The locator is specified by the programmer the available addresses at the RAM and ROM in target. The programmer has to define the available addresses to load and create files for permanently locating the codes using a device programmer.
2. It uses cross-assembler output, a memory allocation map and provides the locator program output file. It is the final step of software design process for the embedded system. Locator program output is in the Intel hex file or Motorola S-record format. The locator uses the cross-compiled codes in different

- cross-compiled segments for: (i) instructions, (ii) initialized values and addresses, (iii) constant strings and (iv) un-initialized data.
3. The locator locates the I/O tasks and hardware device-driver codes at the addresses without reallocation. This is because the port and device addresses for these are fixed for a given system. These are as per the interfacing circuit between the system buses and ports or devices.
 4. The *locator* program reallocates the linked file and creates a file for permanent location of codes in a standard format.
 5. The file format may be Motorola S-record format or Intel hex file or any other format (Section 13.3.2).

Figure 13.4 shows various software tools and chain of actions of linker at host and locator in an embedded system.

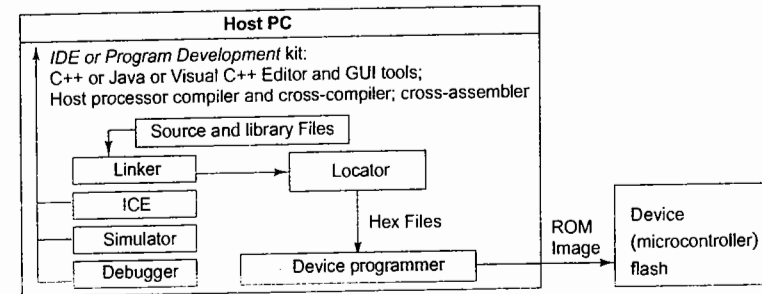


Fig. 13.4 Various software tools and chain of actions of linker at host and locator in an embedded system

13.3.1 Differences in Files, Addressing and Address Resolution Method

Table 13.2 gives the differences in addressing in linker and locator.

Table 13.2 Differences in Files, Addressing in Linker and Locator

Action	Difference
File creation	Linker creates linked file for the disk for use by the host system OS and the loader and then does the memory address allocation. Locator creates linked file for the use of a device-programmer, which copies the file at the device EEROM or flash and copied file runs at the located program directly.
File format	Linker file formats are as per the file system used for the disk. Locator file is as per Motorola-S or Intel hex or any other format (Section 13.3.2).
Addresses	Linker addresses are for the host system processor and are relative addresses, which the loader reallocates. Locator addresses are for the target system processor and are addresses, which are not reallocated later.
Address resolution method	An instruction may have specified address in object file, while actual host or target may be allocated different address-spaces for calling that object file. Addresses are properly resolved in the linker as well as locator. Linker uses relative addresses and actual addresses are allocated at run time when the OS does the memory allocations and the loader loads the program. Locator uses addresses, which once allocated remains permanent as the created file records of locator burns (embeds) into the system.

13.3.2 Locator Output File in Binary Image Motorola-S and Intel Hex Formats

Binary bit mapped (binary image) means bytes are sent in a sequence as per starting address to the end address.

Motorola S-Record Format Motorola S-record format is an industry standard for storing the locator file, before its use by the device programmer or ROM-mask programmer. It is called S-record because it has first character as 'S' in each line. A line is as follows: first character is S, second character is 2 (for specifying the record type), third and fourth characters are for a hexadecimal number, say 14 (to specify that there are 20 bytes in that line), the remaining 40 characters (nibbles) divide as the address (3 bytes) and data (16 bytes) and checksum (1 byte). Table 13.3 shows a typical S-record as a locator output and device programmer input. It is left as an exercise to the reader to show that *Addr* for line 6 in the record of Table 13.3 will be 0x000037.

Intel Hex File Format Intel hex file format is another industry standard for storing the locator file output, before its use by the device programmer or ROM-mask programmer. A line is as follows: first character is ':' (colon), second and third characters for data counts (assume = 10 in hexadecimal in case $N_d = 16$) in the line (address bytes, checksum byte and data type byte excluded, only actual data bytes at the line, which are to be burned in ROM are counted), fourth to seventh address (2 bytes), sixth and seventh as 0 and 0 to specify data as ROM data and the remaining 32 characters as the data (16 bytes) and 2 characters for the checksum (1 byte). Table 13.4 shows an Intel hex file, which corresponds to the same data as at the Motorola S-record in Table 13.4 as a locator output and device programmer input. It is left as an exercise to the reader to show that *Addr* for line 6 in the record of Table 13.4 will be 0x0037.

Table 13.3 An Exemplary Motorola S-Record format

Line Number ¹	First Character	Second Character ²	Third and Fourth Characters for N ³	Address Addr ⁴	N _d ⁵ Bytes for Storage in ROM from Addr (Maximum value of N _d can be 253 decimal)	Checksum ⁵
0	S	2	1 0	000000	aa bb cc dd ee ff xx yy zz bb cc dd	cs0
1	S	2	0 C	00000C	cc aa cc dd ee ff xx yy	cs1
2	S	2	1 2	000014	dd bb cc dd ee ff xx yy zz bb cc dd aa xx	cs2
3	S	2	0 5	000022	0A	cs3
4	S	2	0 8	000023	dd bb cc dd	cs4
5	S	2	1 4	000027	dd bb cc dd ee ff xx yy zz bb cc dd aa ff 01 c0	cs5

¹ Line number is not in the record.

² 2 means the availability of data record in this line. A byte from the data sequentially burns at the ROM.

³ N = 10 means that there are 16 hexadecimal bytes in this line including the 3 bytes for the address and 1 byte for checksum at the end of the line. Number of data bytes for storing are specified in this line, N_d = 12 decimal. 0C means N = 12 and N_d = 8 decimal.

⁴ Starting address of 3 bytes, 000000 means the next 12 bytes store between address 0x000000 to 0x00000B. Therefore, in the next line starting address Addr = 0x00000C.

⁵ Bytes for burning in ROM are in this line and numbering = N_d. Each character in this column represents a nibble. cs0, cs1, ... are the checksums of 1 byte each of all the bits in line number 0, 1, ..., respectively.

Table 13.4 An Exemplary Intel Hex File format

Line Number ¹	First Character	Second and Third Characters for C ²	Address Addr ³	Sixth and Seventh Characters ⁴	N _d ⁵ Bytes for Storage in ROM from Addr (Maximum value of N _d can be 253 decimal)	Checksum ⁵
0	:	0 C	0000	0 0	aa bb cc dd ee ff xx yy zz bb cc dd	cs0
1	:	0 8	000C	0 0	cc aa cc dd ee ff xx yy	cs1
2	:	0 E	0014	0 0	dd bb cc dd ee ff xx yy zz bb cc dd aa xx	cs2
3	:	0 1	0022	0 0	0A	cs3
4	:	0 4	0023	0 0	dd bb cc dd	cs4
5	:	1 0	0027	0 0	dd bb cc dd ee ff xx yy zz bb cc dd aa ff 01 c0	cs5

¹ Line number is not in the record.

² Number of data bytes for storing specified in this line. C_d = 12 decimal. 0C means C_d = 12.

³ Starting address of 2 bytes, 0000 means the next 12 bytes store between address 0x0000 and 0x000B. Therefore in the next line starting address Addr = 0x000C.

⁴ 0 and 0 means the availability of data record in this line is for the ROM. A byte from the data sequentially burns at the ROM.

⁵ Bytes for burning in ROM are in this line and numbering = N_d. Each character in this column represents a nibble. cs0, cs1, ... are the checksums of 1 byte each of all the bits in line number 0, 1, ..., respectively.

13.3.3 Memory Map for coding a locator

Figure 13.5(a) shows memory addresses needed in the case of Princeton architecture in the system. Figure 13.5(b) shows memory addresses needed in the case of Harvard architecture. These differ in following respect.

1. Vectors and pointers, variables, program segments and memory blocks for data and stacks have different addresses in the program in Princeton memory-architecture.
2. Program segments and memory blocks for data and stacks have separate sets of addresses in Harvard architecture. Control signals and read-write instructions are also separate.

The *system memory allocation map* is not only a reflection of addresses available to the memory blocks, and the program segments and addresses available to the IO devices, but also reflects a description of the memory and IO devices in the system hardware. It maps guides to the actual presence of the various memories at the various units, EPROM, PROM, ROM, EEPROM, Flash memory, SRAM (static RAM), DRAM (dynamic RAM) and IO devices. It reflects memory allocation for the programs, and data and IO operations by the locator program. It shows the memory blocks and ports (devices) at these addresses. Figure 13.6(a) and (b) show memory and I/O devices memory allocation map for the 68HC11 (having memory-mapped IO architecture), and for an IBM 80x86 PC (having IO-mapped IO architecture), respectively.

Four examples of memory allocation maps are given in Figure 13.7(a)–(d). *System I/O devices map* may be designed separately. An IO map not only reflects the actual presence of the I/O devices, but also guides the available addresses of the various device registers and port data. (An example of a device is a timer. I/O devices are the peripheral units of the system.)

Memory map is used for coding locator software. The memory map defined for a locator includes the device I/O addresses designed after appropriate address allocations of the pointers, vectors, data sets and data structures. From the map, the locator program input can be easily designed. When the main memory is of Harvard architecture, the program memory map will be separate, for example, 8051. The processor reads from the program memory by a separate set of instructions (input-output instructions) and control signals.

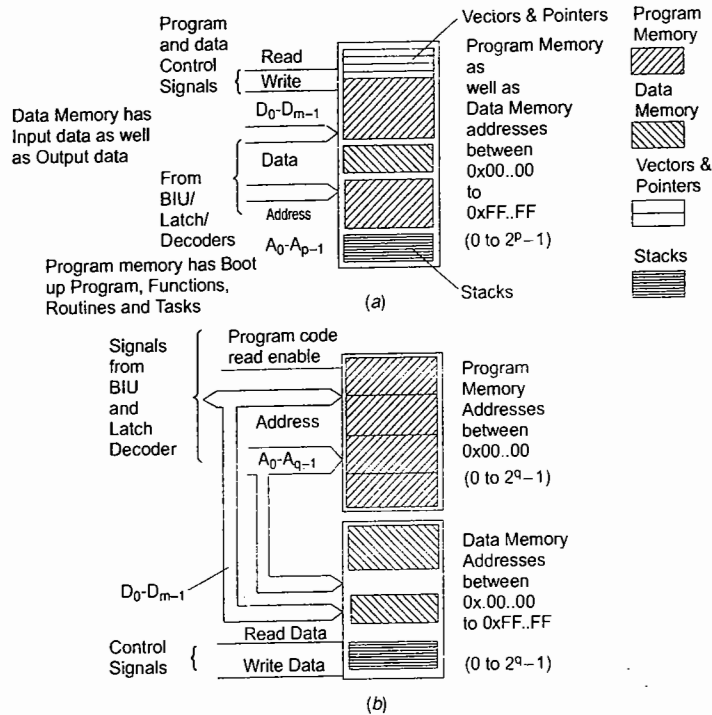


Fig. 13.5 (a) Memory map (Princeton architecture) (b) Memory map (Harvard memory architecture)

13.4 GETTING EMBEDDED SOFTWARE INTO THE TARGET SYSTEM

13.4.1 Device PROM or Flash Programmer

Device Programmer This is also called laboratory programmer which is a programming system for a device. The device is selectable and may be a PROM or EPROM chip or a flash or a unit in a microcontroller or PLA, GAL or PLC. The selected device inserts into a socket (at the device programmer circuit) and is programmed (burned the codes) by transfer of the bytes for each address using the software at the host.

The software of the device programmer runs at a host system (PC or workstation or laptop). The host system interconnects with the socket and the device programmer circuit usually through a serial port (UART or USB).

Device programmer software running at the host uses an input file from the locator software output records. The file reflects the final design and has a bootstrap program plus the compressed record, which the processor decompresses before the embedded system processor starts execution. (Bootstrap program is the program to start up a system. We start from home by strapping our boots.)

Note: An IDE incorporates the device programmer within it.

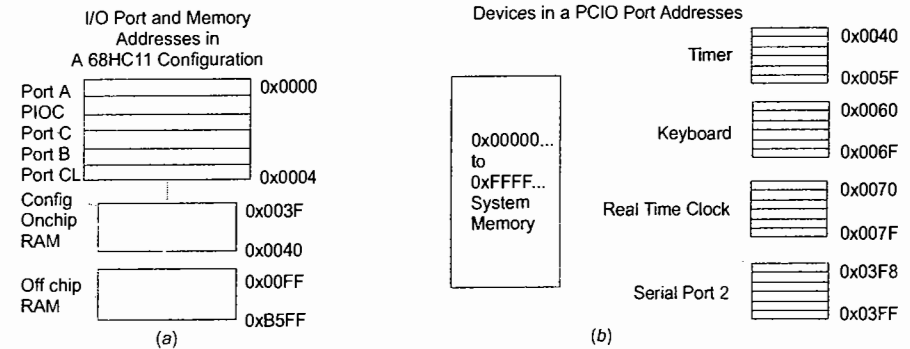


Fig. 13.6 (a) IO port, memory and device address spaces in 68HC11 (b) Device addresses in 80x86-based host system (PC)

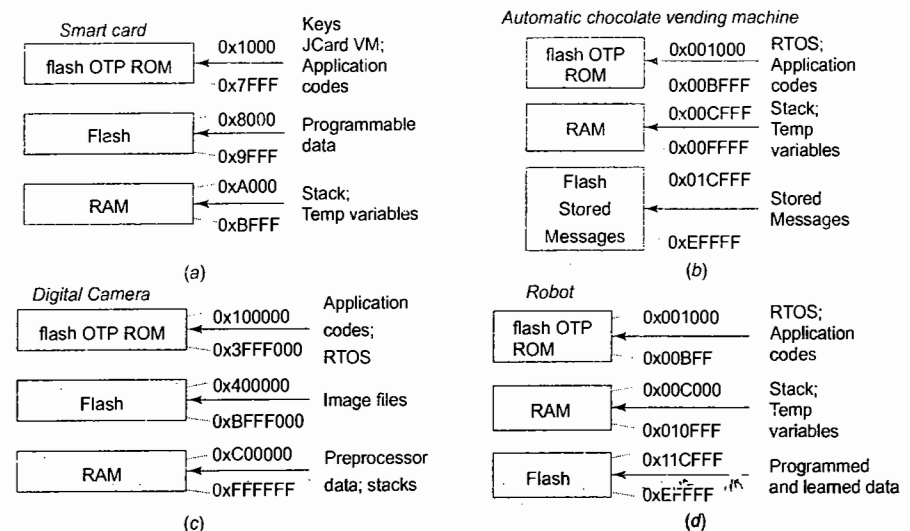


Fig. 13.7 (a-d) Four memory allocation maps in four exemplary systems for their locator programs

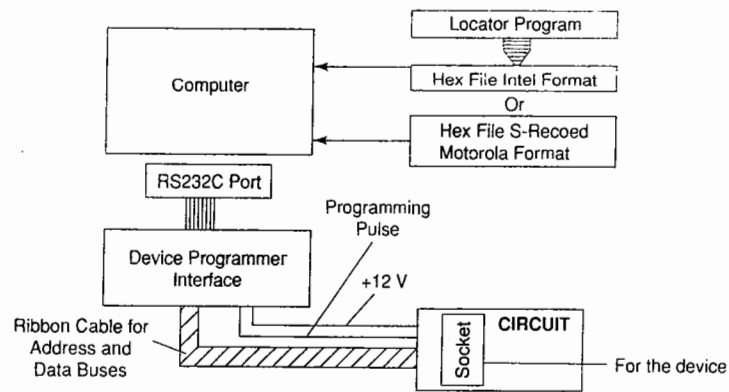


Fig. 13.8 Burning in of the application software codes, data and tables using a device programmer

Use of Device Programmer for Downloading the Finalized Codes into PROM or Flash A locator output is of the final design with a booting program plus the system program (with or without a compression) plus initial data and shadow RAM data. Assume that a system design phase up to the target system is over. Refer to a memory map of the target PROM or flash. Finalized ones are put in non-volatile memory at each memory address into the system by a process called *burning*.

Burning is a process that places the codes. Codes are the ones to be downloaded, according to ROM image locator output). Burning is done in the laboratory using a device programmer into an erased EPROM or EPROM or PROM or flash.

Figure 13.4 showed the method for burning-in the EPROM or in the EEPROM the S record or hex file generated by a locator (Section 13.3.2). EEPROM does no erasing and can be programmed directly by the device programmer. Flash uses a different file system.

Consider a device that has a 512 kB programmable memory. It means that it has eight signals (D_0 to D_7) and $9 [= \log_2(512 * 1024)]$ address (A_0 to A_{18}) signals. There are a total of $5,24,288 (= 512 * 1024)$ arrays of cells with each array having eight cells. Each cell has for output the D bit as logic 1 in the un-programmed (fresh) state. Programming the device (locating the desired bytes) means replacing 1s with 0s according to each byte needed at each cell-array address. Bytes saved are as generated by the locator program of the embedded system after the programming (after programming, the device memory part will hold the final bytes according to the need for the completion of software development, testing and debugging cycle of design).

A ROM device programmer is a programming system for the PROM or EPROM or flash in a chip or unit of a microcontroller or device. A device inserted into a socket (at the device programmer circuit) is programmed by transferring the bytes for each address using a software tool at the computer and interconnecting the computer with this circuit. The device program needs the locator output records in the input. This output must reflect the final design, only then can the device program put the final inputs into the ROM. Records that are input to a device programmer as per the device being programmed, are in three formats. The file formats for burning the binary image are described in Section 13.3.2.

Alternative to a device programmer is that the application software codes are sent in a tabular form to a specialized manufacturer. The manufacturer prepares the ROM. The ROM is needed especially when many thousands of pieces are needed. Integration of ROM with the processor, RAM and other hardware of the target system gives the final product.

13.4.2 Programming Method of Device Programmer

A 512 kB device cell-array (at the address defined by A_0 to A_{18} signals) stores the '0's as per 0s at D_0 to D_7 when a strobe pulse of a few microseconds duration is applied in the presence of a voltage V_p by the device programmer circuit. A device programmer that programs its memory unit performs the following eight steps in a sequence using the software tool, computer and the device programmer circuit. (i) Applies the A_0 to A_{18} bits as needed at a selected address input of the array of cells. (ii) Applies as inputs the D_0 to D_7 bits that are meant for that address. (iii) Applies V_p to make programming feasible for the needed duration in microseconds. (iv) Applies a programming pulse for a sufficient duration to cause fusing of the desired links in the array, to convert a '1' to '0'. (v) Switches off the V_p . (vi) Applies a next higher address than the previous one. (vii) Repeats the aforementioned steps (ii) to (iv) for writing (converting) the logic states of D_0 to D_7 bits at current instance at new address. (viii) Continues till a cell array at last desired address is programmed.

The process of writing into the device is by converting logic 1 to 0, and is done by fusing the *links* on applying programming voltage and programming pulse for a short duration.

The working of a device programmer is according to the processing and memory device.

Using EEPROM 68HC11 Example The working of a device programmer for programming the internal EEPROM of 68HC11 is as follows. The 68HC11 has a control register CONFIG. It is for system configuration control. It keeps the data bits like an internal EEPROM address. It is also called an EEPROM register whenever the programming unit is used within 68HC11. There is another register, the EEPROM register, at the address 0x003B. This register is kept to program both CONFIG as well as the EEPROM addresses in 68HC11 (the on-chip EEPROM addresses in 68HC11 are from 0xB600 to 0B7FF). If CONFIG.0-bit (EPROM) is '0', the erase or write (burn in) does not become feasible by any instruction. To burn in, first, the 0th bit of CONFIG (EEPROM) register is made '1'. Only then is the EEPROM programming voltage can be ON. If the first bit is also made '1' the EEPROM addresses and their data are latched with the help of programming units within 68HC11. If the register EEPROM.3 and .4 bits becomes 00 (0 and 0, respectively) bulk erase takes place at the EEPROM addresses (bulk erase refers to all the available EEPROM addresses). If 01 is written a byte is erased (byte erase means erase at one EEPROM address only). If 10, then a row of 16 bytes is erased. If the second bit in EEPROM is made '1' then only is the erase function enabled. An erase means all bits at an EEPROM address are made '1's. The erase time is a total of 10 ms in all these three modes. When the erase function is disabled (due to second bit = '0' but programming voltage becoming enabled because the 0th bit = '1'), the burn in of bytes takes place by the execution of the write instruction for the appropriate address.

A computer's RS232C UART port that sends and receives at 1200 baud and connects to Rx/D and Tx/D pins in 68HC11 through a line receiver and a line driver, respectively. VDD, VRH, IRQ, XIRQ pins are at +5V. VSS and VRL are at '0'. The reset circuit and 8 MHz crystal circuit connect as usual. Once 68HC11 is configured in the bootstrap mode, the bits at its EEPROM addresses are programmable using the software in the computer that is a part of the development system. An external computer transfers at 1200 baud in bootstrap mode when using an 8 MHz crystal with 68HC11. Further, the transfer is first of a byte with all '1's (0xFF). The computer then transfers 256 bytes to 68HC11. These bytes load between 0x0000 and 0x00FF internal RAM addresses. 68HC11 automatically sets its PC at the end of transmission after reading its vector from 0xFFFFE. It, therefore, starts executing a program called bootstrap program. This can be a test program that includes the write to CONFIG or EEPROM register and to EEPROM addresses.

Using EPROM When using EPROM of the processing device (e.g., microcontroller), PROM is first erased in ultraviolet (UV) light. Erasing makes all the 8 bits at each of the addresses into '1's. An erase facility provides reusability of the memory whenever the application software changes for another version

of the system. The software executed in the computer programs the EPROM as well as verifies the bytes burned into the EPROM with the help of an interfacing circuit between the EPROM and computer's RS232C serial port. The EPROM interface circuit receives a byte serially from the computer through the TxD line and later sends, along with its address, this byte for burning in the processing device's EPROM. Burning of codes is done as follows: During the period when the appropriate address and data are available from this circuit, it also switches ON a high-voltage $-V_p$ Volt and applies a program pulse for a needed period. This circuit is sequentially programmed at each address by increasing the address after every program pulse.

An EPROM interface circuit also receives another byte serially from the computer through a TxD line and sends this byte again for burning-in the processing device at the appropriate address. The bytes at the successive addresses are received by the computer in a verify mode through the interface and RxD line. Some processing devices have an auto program mode for its EPROM in which it can automatically copy the codes and data from an IC.

13.5 ISSUES IN HARDWARE–SOFTWARE DESIGN AND CO-DESIGN

There are two approaches for the embedded system design.

- (1) The software development life cycle ends and the life cycle for process of integrating the software into hardware begin at the time when a system is designed.
- (2) Both cycles concurrently proceed when co-designing a time critical sophisticated system.

The final design, when implemented, gives the targeted embedded system and thus the final product. Therefore, an understanding of the (i) software and hardware designs and integrating both into a system and (ii) hardware–software co-designing are important aspects of designing embedded systems.

Further there is hardware–software trade-off. Certain embedded components, for example, CCD coprocessor, CODEC execute fast when implemented by hardware but the design cost is high and the processor performance requirements are also high.

Let us refer to an interview of Jean-Louis Brelet in an article 'Exploring Hardware/ Software Co-design with Vertex-II Pro FPGAs' (*Xcell Journal*, pp. 24–29, Summer issue, 2002). A Brelet reply, quoted verbatim, when asked about the expertise required for successful implementation is as follows: 'Software people must understand the nature of hardware design and type of problems encountered by hardware team. They also must understand the possibilities and capabilities of hardware. Likewise hardware team must have a good understanding of software and how the applications operate. Both teams must have a good understanding of each other's language and a willingness to adapt'.

The selection of the right hardware during hardware design and an understanding of the possibilities and capabilities of hardware during software design is critical especially for a sophisticated embedded system development such as Apple iPhone.

3.5.1 Choosing Right Platform

Software Hardware Tradeoff There is a tradeoff between the hardware and software. Hardware implementations provide advantage of processing speed. It is possible that certain subsystems in hardware—controller, IO memory access circuit, real-time clock, system clock, pulse width modulation, timer and serial communication are also implemented by the software. A serial communication real-time clock and timers featuring microcontroller may cost more than the microprocessor with external memory and a software implementation.

Hardware implementation provides the following other advantages. (i) Reduced memory for the program. (ii) Reduced number of chips but at an increased cost. (iii) Simple coding for the device drivers. (iv) Internally embedded codes, which are more secure than at the external ROM.

Software implementation provides the following advantages. (i) Easier to change when new hardware versions become available. (ii) Programmability for complex operations. (iii) Faster development time. (iv) Modularity and portability. (v) Use of standard software engineering, modelling and RTOS tools. (vi) Faster speed of operation of complex functions with high-speed microprocessors. (vii) Less cost for simple systems.

It may be or may not be possible that certain subsystems in hardware (ASIP, microcontroller, DSP, single-purpose processors) are implemented by software to get desired performance with the least system cost.

Choosing a Right Platform System design of an embedded system also involves *choosing a right platform*. A platform consists of a number of units. Table 13.5 shows a list of these units and various corresponding sections, which a programmer can refer to while selecting the unit to finally obtain a right platform and right development tools.

Table 13.5 List of Units to Choose for Finally Obtaining a Right Platform and Right Development Tools

<i>Unit to be Chosen</i>	<i>Section Describing that in Detail to Enable the Right Choice</i>
Processor	Sections 1.2, 2.1 and 2.3
ASIP or ASSP	Sections 1.2.3 and 1.2.4
Multiple processors	Section 1.2
System-on-chip	Section 1.6
Memory	Section 2.7.1
Other hardware units of system	Section 1.3
Buses	Sections 2.1.4, 2.2.1, 3.2, 3.3, 3.10, 3.11 and 3.12
Software language	Sections 5.1, 5.5, 5.6 and 5.7
RTOS (real-time programming operating system)	Sections 8.8, 8.9, 9.2, 9.3, 10.1, 10.2 and 10.3
Code generation tools	Section 13.1
Tools for finally embedding the software into binary image	Section 13.3

Embedded System Processors Choice

(A) Processor-less System: We have an alternative to a microprocessor or microcontroller or DSP. Figure 13.9(a) shows the use of a PLC in place of processor. We can use a PLC for the clothes-in clothes-out type system (Section 1.1.1). A PLC fabricated by the programmable-gates, PALs, GALs, PLDs and CPLDs.

A PLC has very low operation speed. It also has a very low computational ability. It has very strong interfacing capability with its multiple inputs and outputs. It has system-specific programmability. It is simple in application. Its design implementation is also fast. Automatic chocolate-vending machine can be another exemplary application of PLC.

(B) System with Microprocessor or Microcontroller or DSP: Section 1.2 gave a detailed description of the processors described for the embedded systems in detail. Figure 13.9(b) shows the use of a microprocessor or microcontroller or a DSP.

(C) System with Single-Purpose Processors and IPs in VLSI or FPGA: A line of action in designing can be use of the IPs, synthesizing using VHDL-like tool and embedding the synthesis into the FPGA.

Figure 13.9(c) shows the processing of functions by using IPs embedded into VLSI or FPGA instead of processing by the ALU. The IPs implement the functions, which if implemented with the ALU then coding by programmer will take a long development time.

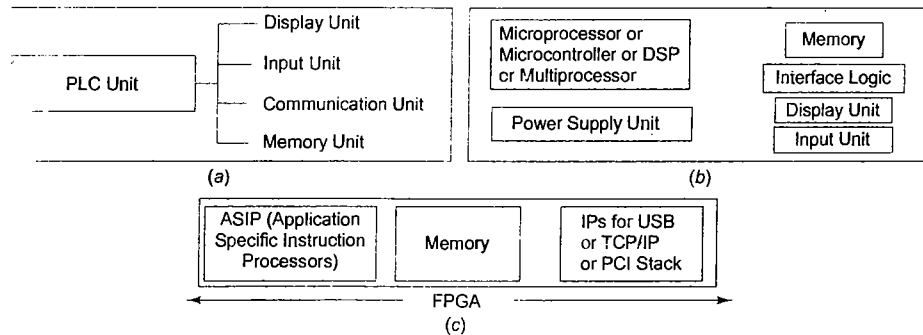


Figure 13.9 (a) Use of a PLC in place of a processor (b) Use of a microprocessor or microcontroller or a DSP (c) Processing of functions by using IP embedded into the FPGA instead of processing by the ALU

The sophisticated operation parts on a VLSI chip implement using copyrighted IPs. Each IP is synthesized at gate level using VHDL or Verilog. (VHDL (VLSI high-level description language) and Verilog are the languages for simulating and synthesizing the gate-level design. VHDL also implements concurrency and synchronization problems and a structural hierarchy strategy. In addition to these features, Verilog uses C-like functions. Therefore, the exception handling and timing problems are also programmable. There are two languages for programming and implementing FSM, state transitions, concurrency, synchronization and behavioural hierarchy. These are StateCharts and SpecCharts.)

(D) **Factors and Needed Features Taken into Consideration:** We consider a general purpose processor or choose an ASIP (microcontroller or DSP or network processor). When the 32-bit system, 16kB+ on-chip memory and need of cache, memory management unit or SIMD or MIMD or DSP instructions arise, we choose a microprocessor or DSP. For example, the video game, voice recognition and image-filtering systems need a DSP. Table 13.6 gives the factors that are considered by a system programmer before choosing a microprocessor or microcontroller as a processing unit.

Refer Section 2.1 for 8051. Microcontroller provides the advantage of on-chip memories and subsystems like the timers. Table 13.6 can help in selecting the microcontroller. A microcontroller available at a reasonable price may not support these on-chip needs. Then decide which one will suffice, 8-bit or 16-bit or 32-bit ALU. We will take a decision about which microcontroller and its version with what features is needed. First selection criterion is on-chip memories needed for the embedded system. A second criterion is the on-chip timers and external communication subsystems needed in each system. Third and fourth may or may not be needed in the system being designed. The third criterion is the need for input captures (interrupt and load time on an input) and output-compare (output and interrupt when timer contents equal a comparison register). Fourth is about I/O and/or ADC on-chip availability. Latest versions for 8-, 16- and 32-bit microcontrollers can be found on the websites of the giants, ARM, Intel, Motorola, Phillips and Microchip (for 8-bit systems).

Table 13.6 Factors and Needed Features in the Microprocessor or Microcontroller or DSP-Processing Unit of the System

Factors for On-Chip Feature	Needed or which One Needed	Available in Chosen Chip
8-bit or 16-bit or 32-bit ALU	8/16/32	8/16/32
Cache, memory management Unit or DSP calculations	Yes or no	Yes or no
Intensive computations at fast rate	Yes or no	Yes or no
Total external and internal memory up to or more than 64 kB	Yes or no	Yes or no
Internal RAM	256/512 B	256/512 B
Internal ROM/EPROM/EEPROM	4 kB/8 kB/16 kB	4 kB/8 kB/16 kB
Flash	16 kB/64 kB/1 MB/8 MB	16 kB/64 kB/1 MB/8 MB
Timer 1, 2 or 3	1/2/3	1/2/3
Watchdog timer	Yes or no	Yes or no
Serial peripheral interface full duplex or serial Synchronous communication interface (SI) half duplex	Full/half	Full/half
Serial UART	Yes or no	Yes or no
Input captures and Out-compare	Yes or no	Yes or no
PWM	Yes or no	Yes or no
Single- or multi-channel ADC with or without programmable voltage reference (single or dual reference)	S/M W/WO V_{ref} S/D	S/M W/WO V_{ref} S/D
DMA controller	Yes or no	Yes or no
Power dissipation	Very low/low or normal	Very low/low or normal

13.5.2 Memory- and Processor-Sensitive Software

Table 13.7 gives the examples of memory-, processor-sensitive programs.

13.5.3 Allocation of Addresses to Memory, Program Segments and Devices

Functions, Processes, Data and Stacks at the Various Segments of Memory Program routines and processes can have different segments. For example, a program code can be segmented and each segment stored at a different memory block. A pointer points to the start of the memory block storing a segment and an offset value is used to retrieve a memory address within that segment.

There can be different segments at the memory for the functions and processes (threads or tasks). These can comprise of different segments for data and different segments for the stacks. Each segment has a starting memory address and ending memory address. Each segment has a pointer address and an offset address. Using offset, a code or data word is retrieved from a segment.

There can be different sets and different structures of data at the memory (Sections 5.4.2 and 5.4.3). Following are the examples of the data structures and data sets that are commonly used during processing in a system and that are stored at the different memory blocks in a system.

Data structure, called *stack* is a special program element. A *stack* means an allotted memory block from which a data element is always read in a LIFO mode by the processor. Various stack structures may be created during processing. A call can be made for another routine during running of a routine. In order that on

completion of the called routine, the processor returns only to the one calling, the instruction address for return must be saved on the stack. There can also be nesting. It means one routine calling another, and that calling another and return from the called routine is always to the calling routine. Therefore, at the memory a block of memory address is allocated to the stack that saves the *return addresses* of the nested calls.

1. There may be at the beginning an input data saved as a stack at RAM in order to be retrieved later in the LIFO mode. An application may create the run-time stack structures. There can be *multiple data stacks* at the different memory blocks, each having a separate pointer address (Figure 5.1).

Table 13.7 Hardware-Sensitive Programming

Program	Examples
Processor-sensitive	Recall Sections 2.1, 2.3 and 2.4. A processor has different types of structural units. It can have memory-mapped IOs or IO-mapped IOs. IO instructions are processor-sensitive. A processor may be having fixed-point ALU only. Floating-point operations when needed are handled differently than in a processor with floating-point operations. A processor may not provide for execution of SIMD (single instruction multiple data) and VLIW (very large instruction word) instructions. Programming of the modules needing these instructions is handled differently in different processors. Assembly language may sometimes facilitate an optimal use of the processor's special features and instructions. Advanced processors usually provide the compiler or optimizing compiler subunit to obviate need for programming in assembly.
Memory-sensitive	<ol style="list-style-type: none"> (i) An example of a memory-sensitive program is video processing and real-time video processing. The picture resolution actually used for processing and number of frames processed per second will depend on the memory available as well as the processor performance. If a large memory is available, then higher resolution pictures can be processed. When higher resolution without acceptable missing frames is used, then for the given performance of processor in MIPS the less number of frames can be processed. Real-time programming model and algorithm used by a programmer will depend on memory available and processor performance. (ii) Memory address of IO device registers, buffers, control registers and vector addresses for the interrupt sources or source groups are prefixed in a microcontroller. Programming for these takes into account these addresses. The same addresses must be allotted for these by the RTOS. Memory-sensitive programs need to be optimized for the memory use by skillful programming. (iii) When using certain instruction sets like Thumb@ in ARM processor helps 16-bit instructions, which save less memory space than the use of 32-bit ARM instruction set.

2. Each task or thread in a multitasking or multithreading software (Sections 7.1 to 7.3) should have its own stack where its *context* is saved. The context is saved on the processor switching to another task or thread. The context includes the return address for PC for retrieval on switching back to the task. There are *multiple stacks* at the memory for the different contexts at the different memory blocks, each having a separate pointer address. Application programs and supervisory programs (OS) have separate stacks at separate memory blocks.

Each processor has at least one stack pointer so that the instruction stack can be pointed and calling of the routines can be facilitated. When a processor has only one stack pointer, the OS allocates the memory addresses that are used as the pointers for the multiple instruction and data stacks of the tasks (or processes or threads).

A stack is a special data structure at the memory. It has a pointer address that always points to the top of a stack. This pointer address is called a stack pointer.

The other data sets, which are also allotted memory, are as following.

1. A *string* is allotted memory for ASCII (8-bit) or Unicode (16-bit) characters, followed by a null character at the end. A string object is allotted memory for the fields for string characters and for the methods to manipulate the string (e.g., concatenation).
2. A *circular queue* is allotted addresses for a queue in which both pointers cannot increment beyond the memory block (buffer) and reset to starting value on insertion beyond the boundary (Figure 5.2).
3. A *one-dimensional array* is allotted addresses for a special data structure at the memory. It has a pointer address that always points to the first element of the array. From the first element pointer and index of that element, an address is constructed from which the processor can access one of the array elements. Index is an integer that starts from 0. Data word can be retrieved from any element address in the block that is allocated to the array.
4. A *table* is a two-dimensional array (matrix) data set that is allocated a memory block. Three pointers, table-base, column-index and destination-index pointers can retrieve an element of the table. There is always a base pointer for a table. It points to its first element at the first column first row. There are two indices, one for a column and the other for a row. Figure 5.3(a) shows a memory block with the pointers for a table.
5. A *hash table* is allocated a memory block for a data set that is a collection of pairs of a key and a corresponding value [Figure 5.3(b)]. A hash table has a key or name in one column. The corresponding value or object is at the second column. The keys may be at non-consecutive memory addresses. Just as an index identifies an array element, a hash key identifies a hash element.
6. Look-up tables have columns and store the pointers to the values. The first column of a table is used as a pointer to the value to get the set of values.
7. A list is allotted memory for a data structure in which each element also stores a pointer to the next element at list. It has one memory block allotted to each of its elements. The list-top pointer points to its first element and the last element points to null [Figure 5.3(c)]. A *list* is a data structure with a number of memory blocks, one for each element. A list has a top (head) pointer for the memory address from where it starts. Each list element at the memory also stores the pointer to the next element. The last element points to *null*. A *list* is for non-consecutively located objects at the memory.

Device, Internal Devices and I/O Device Addresses and Device Drivers All I/O ports and devices have addresses. These are allocated to the devices according to the system processor and the system hardware configuration.

I/O device addresses are considered as part of the memory addresses by certain processors. Certain processors provide for configuring the memory addresses. On the other hand, the 8051, 80196 and 80196 microcontrollers have pre-assigned device addresses for its internal devices and these are un-configurable addresses.

I/O device addresses are not the part of the memory addresses in 80x86 processors. (Figure 2.8(b)). Sections 3.2 and 3.3 described I/O serial and parallel devices in detail. Device addresses are used for processing by the *driver* (Section 4.9). A device has an address, which is usually according to the system hardware or may also be the processor-assigned ones. These addresses allocate to the following.

1. Device data register(s) or RAM buffer(s).
2. Device control register(s). It saves control bits and may save configuration bits also.
3. Device status register(s). It saves flag bits as device status. A flag may indicate the need for servicing and show occurrence of a device interrupt.

Each device, and thus each device register must be allocated addresses at the memory map. A very important point to remember is that in most cases, each set of IO device addresses is often fixed by the system hardware. A locator or loader cannot reallocate these to any other set of addresses. Another point to remember is that

depending on the device, at a device address there can be one or a number of device registers. A physical or virtual device can be configured to attach or detach from receiving input and sending output. A device address can also be just like a file, making it read only or write only or both read and write only.

The address of I/O device registers, buffers, control registers, vector addresses for the interrupt sources or source groups are prefixed. Similarly, the addresses of device control register bits and status register bits are prefixed. Programming of each bit is used in different functions of the device. Device-driver codes implementation is hardware-dependent. Open source drivers are available for ports, buses and physical media attachments in Linux. Device drivers in Linux let us use each module of a class of device register, de-register and schedule like a process. Programmers can port these directly as these are open sources also.

Appropriate interface functions are needed for porting into system the processor-sensitive memory-sensitive programs and ISRs. Appropriate drivers are needed for device-sensitive programs.

Example 13.1 gives the details of addresses of the registers of an I/O device, *serial-line UART device* (Section 3.2).

Example 13.1

A serial-line device has the addresses of device registers as follows: These addresses are fixed by its hardware configuration of UART port interface circuit in a system employing 80x86 processor. They are from 0x2F8 to 0x2FE at COM1 in a PC.

- (a) Two I/O data buffer registers (one for receiving and the other for transmitting) are at a common address, 0x2F8. Provided a control bit at address 0x2FBH is 0, during read from the address, the processor accesses from the RBR (receiver data buffer register) of the device and during write to the address, the processor accesses the TRH (transmitter-holding register) of the device at 0x2F8H.
- Provided a control bit at address 0x2FB is 1, data of two bytes of *divisor latch* are at the distinct addresses, 0x2F8 (LSB) and 0x2F9 (MSB). Divisor latch holds a 16-bit value for dividing the system clock. This then selects the rate of serial transmission of bits at the line. (While writing a device driver, remember that a bit in another register (control register) changes the 0x2F8 from an IO register to lower byte of the divisor latch register.
- Three control registers of the device are at three distinct addresses 0x2FA, 0x2FB and 0x2FC. These are as follows: (a) IER (interrupt-enabling register). It enables the device-interrupts. (b) LCR (line control register). It defines how and how many bits will be on the line. (c) MCR (modem control register). It defines how the modem handshakes and communicates.
- Three status registers of the device are at three distinct addresses 0x2FA, 0x2FD and 0x2FE. These are as follows: (a) IIR (interrupt identification register) at 0x2FA. It has the flags. A flag sets on a device interrupt and resets at system reset and at servicing of corresponding device interrupt. (b) LCR at 0x2FD. It defines how and how many bits will be on the line. (c) MCR at 0x2FE. It defines how the modem handshakes and communicates.

An I/O device is at a distinct addresses. The device has three sets of registers: data buffer register(s), control register(s) and status register(s). There can be one or more device registers at a device address. The addresses of a device are according to the system processor and the system hardware configuration. Most processors process the memory devices and other devices with the same instructions. 80x86 processors process the IOs with a different set of instructions (input-output instructions).

13.5.4 Porting Issues of OS in an Embedded Platform

The following portability issues may arise when OS is used in an embedded platform. Table 13.8 gives the platform-dependency issues and the need for appropriate OS-Hardware interface functions for each issue.

Table 13.8 Platform-Dependency Issues and Need for Appropriate OS-Hardware Interface Functions

<i>Platform Dependency</i>	<i>Need of Appropriate OS-Hardware Interface Functions</i>
I/O instructions	A port instruction data type may be different on the different platforms, as follows: <ol style="list-style-type: none"> unsigned char* (PowerPC, M68HC11/12, M68K, S390). unsigned int (ARM) unsigned long (Itanium, Alfa, SPARC) unsigned short (80x86).
Interrupt-servicing routines	Interrupt vectors are to be defined differently. (Section 4.4) OS supports these differently on different platforms.
Data types	OS should have appropriate APIs for data types. There may also be need as Linux declares all data types in <code><asm/types.h></code> and it includes in <code><linux/types.h></code> as the following: <ol style="list-style-type: none"> unsigned byte (means 8-bit character also) unsigned word (means unsigned 16-bit and also unsigned short) unsigned int (means unsigned 32-bit) unsigned long (means unsigned 32-bit).
Interface-specific data types	For example, a network interface card supports 32-bit unsigned integers and with a big endian.
Byte order	It may depend on the processor. Lower byte first in an integer (little endian) and the upper byte first in an integer (big-endian). Some processors support both (ARM).
Data alignment	<ol style="list-style-type: none"> Two or three bytes stored at an address from which the processor accesses 4 bytes in an access. Same data structure at 'C' source file may show differently on different platforms ('C' takes 16-bit integer on a 16-bit processor and 32-bit integer on a 32-bit processor). Compiler must force the alignment of data by the OS-hardware interface function.
Linked lists	An OS maintains the lists for different data structures. OS provides the standard implementation of doubly linked lists and circular linked lists. Platform-dependent device manager and drivers must include support to these. (Circularly linked means last element of the list linked not to the NULL pointer but to the first element of the list. Doubly linked list means that each element has two pointers, one for the next element and one for the previous element.)
Memory page size	PAGE_SIZE is 4 kB in Linux. A processor may support different page sizes than this.
Time intervals	Linux OS defines the system-clock ticks and interrupts at each 10 ms. The timer functions need to be verified for actual functioning on porting an OS into a platform.

When porting RTOS codes into the system, the porting of I/O instructions, ISRs, data types, interface-specific data types, byte order, data alignment, linked lists, memory page size and time intervals must be taken care of as these are platform-specific. OS-hardware interface functions are needed for these.

3.5.5 Performance and Performance Accelerators

Performance Modeling

(A) **System Performance Index:** The performance of the finally developed embedded system is a measure of success. Its performance at each life cycle of the development process is tested for the following: each required function must show after the test that its characteristics are in conformity with the required and agreed specifications.

The system performance index can be defined as the ability to meet required functions and specifications while using the minimum amount of resources of memory, power dissipation and devices and minimum design efforts and optimum utilization of each resource (e.g., high CPU load).

The best embedded software and hardware is the one that achieves the balance among different performance metrics.

(B) **Multiprocessor System Performance:** The multiprocessor system performance is measured by: (i) an optimized partition of the program into the tasks or set of instructions between the various processors, and (ii) an optimized scheduling of the instructions and data over the available processor times and resources. Performance cost is more if there is idle time left than the available time. Performance matrix is first obtained to calculate the total cost.

(C) **MIPs, MFLOPs and DMIPS as Performance Indices:** One performance design metric is how long system takes to execute the desired system functions. Processor clock frequency and MIPs (million instruction per second) and MFLOPs (million floating point instructions per second) are often quoted as design characteristics for expected system performance. It is not however correct metrics. A processor performance design metric is Dhystone/second. Processing performance is often measured in DMIPS Dhystone million instruction per second (1 MIPS = 1757 Dhystone/second) (Section 2.6). EDN Embedded Benchmark Consortium (EEMBC) proposed five-benchmark program suites for: (i) telecommunications, (ii) consumer electronics, (iii) automotive and industrial electronics, (iv) consumer electronics, (v) office automation. It is also used for measuring and comparing embedded system processor performances.

(D) **Performance Metrics: Buffer Requirement, IO Performance and Bandwidth Requirement:** The buffer helps in accelerating the performance of the system. Memory or I/O buffer requirement may be sometimes constraint. IO performance is measured by throughput and buffer utilization. Larger bandwidth requirement in client-server systems may be a constraint.

(E) **Real-Time Program Performance:** Recall Sections 8.10.8 to 8.10.10. Three performance metrics were described: (i) ratio of sum of interrupt latencies as a function of the execution times, (ii) CPU load, (iii) worst case execution time with respect to the mean execution time.

Data communication and multimedia communication have differing performance indices. Loss of any bit needs retransmission. Also no frame or packet miss is tolerable. On the other hand missing frames within acceptable limits are tolerable in video and multimedia systems.

The time of scheduling of a task can be measured by appropriate scope or analyser or by instruction counts or by instruction execution time profiler at the simulators.

Choice of appropriate real-time programming model, partitioning into tasks and scheduling algorithm affect in the following three metrics as follows:

1. **System throughput.** Comparative performance with respect to the previous life cycle in the development process or previous performance of the system. Relative performance equals relative increase in throughput.
2. **Latency or response time of each task or ISR** (Section 4.6). Both throughput and latency may be unrelated.
3. **Delay zitters** may be a performance metric instead of response times in some cases. The delays (latencies) between retrievals of the data frames or packets or video-frames can vary. This variation is random or statistically Gaussian distributed and is called delay zitter. The noticeable zitter in delay from the expected variation is undesired. It degrades the system performance. Image zitters may not be tolerable, but delayed retrieval within the acceptable threshold is tolerable.

Performance Accelerators There can be several ways to accelerate the performance. Examples of these are as follows.

1. Conversion of CDFGs into DFGs, for example, by using loop flattening (loops are converted to straight program flows) and using look-up tables instead of control condition tests to decide a program flow path.
2. Reusing the used arrays and memory and appropriate variable selection, appropriate memory allocation and de-allocation strategy.
3. Using stacks as data structure when feasible instead of queue and using queue instead of list, whenever feasible.
4. Computing slowest cycle first and examining the possibilities of its speed-up.
5. Code such that more words are fetched from ROM as a byte than the multibyte words.
6. Co-processors and IPs such as Java accelerator accelerate the performance.



Summary

- Host system and software development tools are used in developing, testing and debugging the embedded software in development phase.
- There are a number of software and hardware tools to implement the designed system easily with simple efforts. These are: simulators, editors, compilers, assemblers, source code engineering tool, profiler (for viewing time spent at each function or set of instructions), memory scope, stethoscope-like view of code execution, memory and code coverage scope, emulators, ICs, oscilloscopes, logic probes, logic analysers and EPROM/EEPROM application codes burner.
- Linker and locator are used for developing the codes for the target hardware. Locator files have Intel hex or Motorola S format. Device programmer is used to burn the binary image of the codes from the locator-created files.
- System implementation and integration is done using program development kit, source code engineering tool and IDE.
- Prototype development tools and IDE are used to develop the fully simulated, tested and debugged sophisticated embedded systems with simpler efforts.
- Selection of right hardware during hardware design and understanding of possibilities and capabilities of hardware during software design is critical especially for a sophisticated embedded system development.
- There are several ways of measuring system performance. It can be a system performance as per the required and agreed specifications, power dissipation, throughputs, IO throughputs, response time of tasks, deadline misses, response to sporadic tasks, memory buffers, bandwidth requirements and memory optimization. Latency intervals and deadline misses are measured to understand the performance of the real-time programming, scheduling models and algorithms.

- Performance index gives the desired performance with respect to the required specifications or parameters.
- Performance accelerators are used to improve the performance. Acceleration means using the same system by alternative ways such that it reduces execution times of a set of codes, reduces latencies of the tasks or increases throughput or minimizes memory usage or power dissipation or reduces missing deadlines. Some ways are loop flattening, look-up tables, reusing the used arrays and memory and appropriate variable selection, appropriate memory allocation and de-allocation strategy and using stacks as data structure when feasible instead of queue and using queue instead of list whenever feasible. We must look at computing slowest cycle first and examining possibilities of its speed-up.
- Choosing the right processor, memory, devices and bus and porting by OS/RTOS the processor-sensitive, memory-sensitive and device-sensitive instruction is a must. Byte order and data alignment must be according to the platform chosen.



Keywords and their Definitions

- Action plan** : A plan for action of the development process.
- Assembler** : A tool for assembling the edited codes in mnemonics.
- Big endian** : An ordering in which the highest byte of a number is taken as first.
- Burning** : An act of placing the ROM image for code and data in uncompressed or compressed format into an EPROM or EEPROM or flash or microcontroller or some other similar device.
- Circular linked list** : A data structure for a list in which the last element points to the first element instead of pointing to NULL in a usual list.
- Doubly linked list** : A data structure for a list in which each element points to the next as well as the previous element in the list in place of pointing to only the next element or NULL in a usual list.
- Co-designing** : Software team designs with complete knowledge of hardware capabilities and features and hardware team designs with complete knowledge of software CDFGs and functions to be achieved. Certain software functions are implemented by hardware and certain hardware functions are implemented by software with the aim of achieving desired system performance at lowest cost.
- Cross-assembler** : An assembler that assembles code for host machine for simulation and other purposes and later generates assembled codes for the targeted processor.
- Data alignment** : Data alignment means alignment in specific way, for example, when (i) two or three bytes stored at an address but the processor accesses 4 bytes during one access in a processor designed for 32-bit per instruction or word and (ii) the same data structure at 'C' source file showing differently on different platforms having different data alignments.
- Debugging tools** : Tools for debugging embedded system hardware and software functioning.
- Delay zitters** : The delay zitters mean the variations in the delays in retrieving or arrival of successive data sets. The noticeable variations are undesired.
- Device programmer** : A device for burning in the codes (Refer to Section 13.4).
- Disassembler** : A tool for obtaining higher-level codes from the machine codes, which were assembled earlier.

- Edit-test-debug cycle** : A cycle in implementation phase in which codes are edited, tested and debugged for reported error on test.
- Embedded system project management** : Organizing people, processes, product and project. People in embedded system development project means a team of software development, hardware development and system integration engineers.
- Host system** : A PC or workstation or laptop, which is a computer loaded with software tools and includes the program development kit for a high-level language program or IDE.
- Human-machine interactions** : Interactions of a user through tools like keypad, display unit and GUIs.
- I/O instructions** : Processor read, write, byte manipulation and other instructions for using a device at a port.
- Platform dependency** : A function or ISR or device driver or OS function or data type or data structure utilization, dependent on the processor or memory or devices in the system.
- Integrated development environment** : Refer to IDE.
- IDE** : A fully integrated tool consists of simulators with editors, compilers, assemblers, RTOS source code engineering tool, profiler (for viewing time spent at each function or set of instructions), memory scope, stethoscope-like view of code execution, memory and code coverage scope, emulators, logic analysers and EPROM/EEPROM application codes burner.
- Little endian** : An ordering in which the lowest byte of a number is taken as first.
- Networking stack** : A stack according to a protocol chosen, for example, protocol RFC-1323, CIDR, IP Multicast, IP, UDP, TCP, DNS client, DHCP server, SMTP server, RIPv1 support, RIPv2 support, ARP, proxy ARP, BOOTP or RLOGINN client and server (for Telnet). An embedded system socket can then connect to multiprotocol LANs, ATM network or SONET or wireless access and intelligent networks using the protocol stacks for the network.
- Interpreter** : Interpreter does at run-time expression-by-expression (line-by-line) translation to the machine-executable codes.
- Latency** : Time taken to activate code execution after an event or time taken in finishing certain codes before the next one starts.
- Performance index** : Index to measure the desired performance with respect to required specifications.
- Performance accelerators** : Using the same system, alternative ways to *improve* execution time for a set of codes and *reduce* latency or *increase* throughput or *minimize* memory usage or power dissipation.
- Performance metrics** : Indices for measuring the performance using different measures.
- Page** : A unit of memory in kilobytes, which can be, referred to as a single block from start address and a memory address in; it can be referred by start address plus offset.
- Page size** : Size of the page taken by memory manager.
- Prototyping tools** : Tools for developing by co-designing a prototype for embedded system.
- PLC** : A programmable unit to perform sequential logic control functions.
- Porting issues** : Issues when a software developed at one platform is embedded at another platform.
- Right platform** : An appropriate hardware platform with appropriate software to give best

performance at minimum efforts or costs.

- Software–hardware tradeoff** : To appropriately plan and optimize performance at the least cost and choosing which set of processing elements, functions and codes (e.g., VLIWs) are implemented by a hardware subunit and which by a software module.
- System cost** : Cost for hardware and software. It includes all the costs for the development team and management efforts.
- System integration** : Integration of embedded software into the hardware and getting a validated product with optimized performance.
- Target system** : A system for the targeted embedded system that is used during development phase and the final products of software and hardware are made from it.
- Test vector** : A set of statements in the program for controlled flow of programs—path during test phase.
- Throughput** : Number of processes or specified functions executed per unit time. For I/O systems, it is the number of bytes outputted or read per unit time.
- VHDL and VeriLog** : Languages for designing and synthesizing the VLSI implementation of a system or a part of the system.



Review Questions

- Describe functions of compiler, linker, locator, loader, interpreter, disassembler, cross-assembler and integrated development system.
 - Explain functions of device programmer.
 - Why do we use host system for most of the development? What are the software tools needed at the host?
 - What is a target system? How does the target system differ from the final embedded system? What do we mean by application software for a target system?
 - How do the readily-available networking stacks and device drivers at RTOS help in faster error-free design?
 - Why is *system performance index* defined as the ability to meet required functions and specifications while using the minimum amount of resources of memory, power dissipation and devices and minimum design efforts and optimum utilization of each resource (e.g., high CPU load)?
 - Why is the I/O instructions platform dependent? Define throughput of an I/O system.
 - How do the data align? Take the example of 32-bit integer stored as big endian as an example for aligning bytes from an input stream.
 - How do you solve the problem of interface-specific data types?
 - Why is the selection of the right platform essential during the embedded system development process?
 - Explain the software–hardware trade off? What are the advantages and disadvantages of software implementation instead of hardware implementation?
 - What are the advantages and disadvantages of hardware implementation instead of software implementation? What are the advantages of using FPGAs (field programmable system logic IC) in an embedded system?
 - Why are the device drivers of the programs memory- and processor-sensitive?
 - What are the factors for selecting a processor during the system design phase?
- Describe performance-accelerating methods.



Practice Exercises

- Take a commercial IDE, for example, from Kiel and study its functions, features and capabilities.
- Explain with one example the use of each of the following: application development tools, native development environment, APIs to RTOS, debugging capability device simulation, network simulation and user interface.
- Explain with one example the use of each of the following software tools: profiler scope, memory usage scope, stethoscope, scope for trace of program flow, scope for memory allocations and uses and scope for code coverage.
- Explain hardware–software tradeoff by taking the examples of digital camera and ACC.
- You can design an SoC by three routes: using gate arrays, using standard cell and using IPs and basic component layouts. List cases of embedded systems for each of these three routes.
- How does a buffer help in improving a system performance? What is the performance metric for a multiprocessor-based embedded system *router*? When is the minimum interrupt latency taken as embedded system performance metric? (Assume that router that has 10/100 Mbps bandwidth, ethernet interfaces for LANs, Gbps ethernet interface for connection to servers, WAN and internet interface of frame relay, ATM and packet over SONET/SDH.)

Read on-line topic: 'Software Engineering Approach in Embedded System Development Process' and 'Embedded Systems Project Management' at web material accompanying the book and answer the following.

- What do you mean by embedded system-independent design followed by system integration and by embedded system concurrent hardware–software co-design? Give five examples for each design strategy.
- What should be the goal during an embedded system development process? How does it vary from the software development process?
- What is the action plan to follow while designing an embedded system?
- Who are the *people* involved in an embedded system development project? How will you select them for the case studies of systems described in Chapters 11 and 12? How will the team change when real-time video-processing system is under development?
- Give system specifications for: (i) product functions and tasks, (ii) delivery time schedule, (iii) product life cycle, (iv) load on system, (v) human–machine interaction, (vi) operating environment, (vii) sensors, (viii) power requirement and environment, (ix) system cost for a digital camera. Camera should be capable of storing 4 minute video or 500 still images. The system should include the USB port, imaging cum video software, single shot timer standard as well as 10 second delay modes. Multiple resolutions are: 1024 × 768, 640 × 480, 320 × 240 and 160 × 120 pixels. Answer after web search.
- Explain product design life cycle.
- What do you mean by system project management?
- Explain the terms: (i) product functions and tasks, (ii) delivery time schedule, (iii) product life cycle, (iv) load on system, (v) human–machine interaction (e.g., by keypad and display subunits), (vi) operating environment (e.g., temperature and humidity), (vii) sensors, (viii) power requirement and environment, (ix) system cost.
- Explain meaning of conceptual design.
- List UML diagrams, which help in developing the conceptual design, structure and layout.
- Explain two design approaches: independent design and co-design.

Testing, Simulation and Debugging Techniques and Tools

14

R

Embedded system hardware and software architecture, programming and design have been learnt in previous chapters. At the stage of porting codes into hardware, there is edit-test-debug cycle, which is repeated till a bug-free code is obtained.

e

c

a

l

l

L

E

A

R

N

I

N

G

O

B

J

E

C

T

I

V

E

S

Testing and debugging ensures the system quality. A rule, which the developer must follow is that **wrong until confirmed right by testing and debugging**. Documentation in detail for each stage of testing and debugging is also a necessity. We will learn the following:

1. System codes are tested on the host system as host system has application development tools, large memory and windows or powerful GUIs.
2. Simulation by a simulator, which runs on host, helps in system development by simulating target processor or microcontroller, peripherals, devices and network interfaces.
3. Laboratory tools, in-circuit emulator and monitor help in target system hardware development and target system software testing and debugging in the target environment.

14.1 TESTING ON HOST MACHINE

We have two systems with different CPUs or microcontroller and hardware architecture. One system is host and the other is the target (Sections 13.2 and 13.4). The host is generally PC or laptop or workstation. Target is actual hardware to be used for embedded system under development.

Testing and debugging have to be there at each stage as well as at the final stage when the modules are put together. Test at initial stages is done on the host machine. Host machine is used to test hardware-independent codes. Host machine is also used to run simulator (Section 14.2). Figure 14.1 shows the test systems in a development process. It shows host and hardware systems, and host-dependent, target-independent and target-dependent code. The code has two parts: hardware-independent and hardware-dependent codes. For example, port and devices will have fixed addresses on hardware.

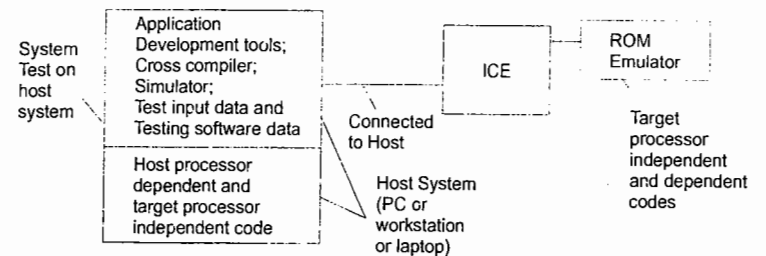


Fig. 14.1 Host and hardware systems and host-dependent, target-independent and target-dependent codes and test systems in a development process

Table 14.1 gives nine steps during testing.

Table 14.1 Testing Steps at Host Machine

Steps	Action
1. Initial tests	Test each module or segment at initial stage itself and on host itself.
2. Test data	All possible combinations of data are designed and taken as test data.
3. Exception condition tests	Consider all possible exceptions for the test.
4. Tests-1	Test hardware-independent code.
5. Tests-2	Test scaffold software (scaffold software is software running on the host of the target-dependent codes and which have the same start code and port and device addresses as at the hardware. Instructions are given from file or keyboard inputs. Outputs are at LCD display and saves a file).
6. Test interrupt service routines hardware-independent part	Those sections of interrupt service routines are called, which are hardware-independent and tested (e.g., deciphering the data routine).
7. Test interrupt service routines hardware-dependent part	Those sections of interrupt service routines are called, which are hardware-dependent and tested (e.g., receiving the port data into a buffer).
8. Timer tests	Hardware-dependent code has timing functions and uses a timing device. Timer-related routines such as <i>clock tick set</i> , <i>counts get</i> , <i>counts put</i> , <i>delay</i> are tested.
9. Assert macro tests	The use of an <i>assert</i> macro is an important test technique. For example, consider a command, <code>assert (pPointer != NULL);</code> . When the <i>pPointer</i> becomes NULL, the program will halt. We insert the codes in the program that check whether a condition or a parameter actually turns true or false. If it turns false, the program stops. We can use the assert macro at different critical places in the application program.

14.2 SIMULATORS

Before flying an aircraft or fighter plane, a pilot uses the flight simulator for training. (A flight simulator may cost hundreds of millions of dollars!)

Simulator uses knowledge of target processor or microcontroller, and target system architecture on the host processor. Simulator first does cross-compilation of the codes and places these into the host system RAM. The behaviour of the target system processor registers is also simulated in RAM. It uses linker and locator to port the cross-compiled codes in RAM and functions like the code that would have run at the actual target system. Host system is a PC or workstation or laptop and generally works in Windows.

Simulator software also simulates hardware units such as emulator, peripherals, network and input-output devices on a host (PC or workstation or laptop). A simulator remains independent of a particular targeted system. It is extremely useful during the development phase for application software for the system that is expected to employ a particular processor or microcontroller or device. The results expected from codes at target system RAM, peripherals, network and input-output devices are obtained at the host system RAM.

A simulator helps in the development of the system before the final target system is ready with only a PC as the tool for development. Simulators are readily available for different processors and processing devices employing embedded systems, and a system designer and/or developer need not code for the simulator for application software and hardware development in the design laboratory. Figure 14.2 shows the detailed design development process using the simulator.

Section 14.2.1 gives the simulator features. Section 14.2.2 gives the possible inabilities of the simulator. Section 14.2.3 describes features of a simulator software VxSim. Section 14.2.4 describes features in the prototype development, testing and debugger tools.

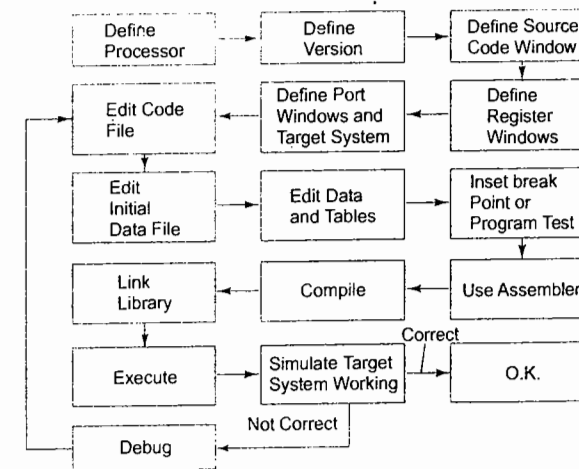


Fig. 14.2 The detailed design development process using the simulator

14.2.1 Simulator Features

A typical simulator is mostly run on a PC Windows environment. A typical simulator includes the following features.

- (1) It defines the processor or processing device family as well as its various versions for the target system.
- (2) It monitors the detailed information of a source code part with labels and symbolic arguments as the execution goes on for each single step.
- (3) It provides the detailed information of the status of RAM and ports (simulated) of the defined target system as the execution goes on for each single step.
- (4) It provides the detailed information of the status of peripheral devices (simulated, assumed to be attached) with the defined system.
- (5) It provides the detailed information of the registers as the execution goes on for each single step or for each single module. It also monitors system response and determines throughput.
- (6) The Windows on the screen provide the following.
 - (a) The detailed information of the status of stack, devices and ports (simulated) of the defined microcontroller system.
 - (b) Program flow trace as the execution continues. A trace means the output of contents of PC versus the processor registers. It is an important debugging tool of an assembly language program. Trace of application software means an output of chosen variables in a function of stepping sequence. Tracescope gives the time on X-axis and chosen parameter on Y-axis as the program continues further. (TraceScope is a tool module to obtain a trace of the changes in the modules and tasks with time on the X-axis. An action-list also produces with specifications of expected time scales.)

- (7) It provide help Windows on the screen. A help Window gives the detailed meaning of the present command pointed by the mice-cursor.
- (8) It monitors the detailed information of the simulator commands as these are entered from the keyboard or selected from the menu.
- (9) It incorporates the assembler, dissembler, user-defined keystroke or mouse-selected macros, and interpreters for C language expressions, as well as for assembly language mnemonics (expressions). It thus tests the assembly codes. The user-defined keystroke macro is a very useful facility. For example, we can define keystroke I, say, for providing a particular input byte at a port n and a particular RAM address byte.
- (10) It supports the conditions (up to 8 or 16 or 32 conditions) and unconditional breakpoints. There is a feature that halts a program after a definite number of times an instruction executes. *Breakpoints* and *trace* are used in the testing and debugging tool.
- (11) It facilitates synchronizing the internal peripherals and delays.
- (12) It employs preempting RTOS scheduler support for the high priority tasks.
- (13) It simulates the inputs from the interrupts, the timers, ports and peripherals. Hence it tests the codes for these.
- (14) It provides network driver and device driver support.

Simulator simulates most functions of a target-embedded system circuit including additional memory, peripherals and buses on the host system itself. It makes application development independent of prior availability of a particular target system. It also simulates the real time processes and shows the outputs on the host system that will be obtained when the codes will actually execute on the targeted particular processor.

14.2.2 Simulator Possible Inabilities

Simulator may not resolve timing issues and hardware-dependent problems. Processor speed at the target processor may not be adequately mapped with the processor speed at the host for calculating time responses and calculating output instances and throughputs at the target.

A simulator may fail to show a bug from the shared data (Section 7.8) as it arises from an interrupt in some peculiar situation only.

A simulator may not be able to simulate the ASICs and IP(s), which may be embedded at the target system. An IC or IP core manufacturer usually provides an alternative debugging tool in that case. For example, ICE for processor ARM7 or ARM9 (Section 2.3.3) emulates the ARM functions on the host processor and system.

A simulator may not be able to take into account of existence internal devices. For example, the target system may use a Java accelerator, whereas the host system may not have that.

A simulator may not be able to take into account portability problems. For example, target system may have its data bus between RAM and un-pipelined processor and host have pipelined processor and 32-bit bus.

14.2.3 Simulating tool Software

Sim. *VxSim* is a simulator tool, which provides a virtual target for developing and debugging the codes. It helps in avoiding the repeated code located in actual target board of the embedded system. Simulating the application with *VxSim* is of great help in the early development stage, as the *VxWorks* RTOS task scheduling can be thoroughly simulated before implementation into the target.

Table 14.2 gives the features.

Table 14.2 Features in an Exemplary Simulator *VxSim*

Supporting Features	Activities
Application development tools	It supports the UML and 'RougeWave'. It gives a short design cycle.
Native development environment	It supports several native development environments and debugging. Environment may be MS Visual C++ or GNU tools.
Simulation APIs of the RTOS	Simulates use of many APIs of the RTOS for a given hardware.
Debugging capability	Debugging capability enables fault finding a much easier task.
Device simulation	Simulates devices and device driver behaviour.
Network simulation	Network simulation capabilities make it a virtual test bed, which permits modelling of complex multinode networked systems. For example, a router or gateway. A network application can simulate internal subnet or a real network. When simulating a network, it generates stacks for various standard network protocols that include even the IP multicast and IP broadcast.
User interface simulation	Simulates, for example, a set-up box interface.

14.2.4 Prototype Development, Testing and Debugger Tools for Embedded System

A prototype development tool can be used in place of target system hardware. These tools simulate, compile and debug with a browser. The browser summarizes the final targeted embedded system's complete status during the development phase. Table 14.3 gives the features of a set of prototyping tools from WindRiver.

14.3 LABORATORY TOOLS

14.3.1 Simple Volt-Ohm Meter

Simple Volt-Ohm Meter can be used to test the target hardware. It has two leads marked red and black. One end of each is connected to the meter and the other to points between which the voltage or resistance is to be measured. The meter is set for *Volt* for checking the power supply voltage at source and voltage levels at chips power input pins, and port pins' initial at start and final voltage levels after the software runs. The meter is set for *Ohm* when checking broken connections, improper ground connections and burn out resistances and diodes.

14.3.2 Simple LED Tests and Logic Probe

Let us remember that a digital signal is simply a discrete signal between two voltage ranges. CMOS logic, '1' is the discrete range V_{DD} to $0.66 V_{DD}$ and '0' is $0.33V_{DD}$ to V_{DD} , where V_{DD} is usually 5 V with respect to V_{DD} (ground potential). An analog signal varies continuously. A *logic probe* is the simplest hardware test device. It is a handheld pen-like device with LEDs. Its LED glows green, when the probe tip touches at the test point (port or at hardware pin) and the bit there is '1'. It glows red if it is '0'. A logic probe LED blinks fast in a probe version, when a test point is at '1' and does not blink when at '0'. The device at the other end connects by a wire to the ground potential.

Table 14.3 Set of Prototyping Tools from WindRiver®

Tool	Features
ScopeProfile	This dynamic execution profiler lets us see, like an oscilloscope waveform, where the CPU is spending its cycles. Performance bottlenecks can then be understood. It shows how much time the processor spends in each function in the task or ISR.
MemScope	Memory usage is a critical aspect of an embedded system. Is there any wasteful use of memory? Is there any memory leak error? Memory leak means that a pointer is incrementing into the unassigned area for a task or stack overflow or writing at the end of an array. MemScope gives the memory block usage. It detects the leak due to system call or another ported module.
StethoScope	Just as a stethoscope helps a doctor in diagnosis, it dynamically tracks the changes in any program variable. It tracks the changes in a parameter. It lets us understand the sequences of multiple threads (tasks) that execute. It records the entire time history.
TraceScope	It helps in tracing the changes with time on the X-axis and an item from the list of actions on Y. TraceScope lets us find the RTOS scheduler behaviour during task switching and notes the times for various RTOS actions.
CodeTest memory, trace and coverage	These tools help in code testing by dynamic memory allocation analysis, controlled flow view trace and code coverage under various real-world situations. The code coverage study helps in removing the extra codes and functions not needed for a specific application. It facilitates development of a scalable system.
VxWorks networking stacks	Another power tool that enhances the code development process. VxWorks RTOS prepares the stack for sending data on the internet to test high-performance switching devices. The stack is according to the protocol chosen. The protocols are the following: RFC-1323, CIDR, IP Multicast, IP, UDP, TCP, DNS client, DHCP server, SMTP server, RIPv1 support, RIPv2 support, ARP, Proxy ARP, BOOTP, RLOGINN client and server (for Telnet). An embedded system socket can then connect to multiprotocol LANs, ATM network or SONET or wireless access and intelligent networks.
VxSim	A powerful simulator tool, which provides a virtual target for developing and debugging the codes. It helps in avoiding the repeated code located in the actual target board of the embedded system. Simulating the application with VxSim is of great help in the early development stage, as the RTOS task scheduling can be thoroughly simulated before implementation into the target.

A logic probe becomes an important tool when studying long delay effects (>1 second) at a port. Its application is as follows. A short program for a delay and then sending the results at the port using logic probe will test the OS timer ticks.

14.3.3 Oscilloscope

Finally, code-downloaded hardware needs testing after completing the edit-test and debug cycle, using a simulator or IDE. An oscilloscope is a scope with a screen to display two signal voltages as a function of time. It displays analog as well as digital signals as a function of time.

We must use it with DC (directly coupled) inputs. Another terminal of the input is always properly kept at ground potential. The term DC should not be confused with the direct current supply. Oscilloscope has two selections for an input, DC and AC. DC means directly coupling the input to the scope input amplifier. AC means input to amplifier (vertical section amplifier) after a capacitor. When using at AC, the signal shape can distort. Its use is only when viewing the signal in a form, in which the signal amplitude = 0 when averaged over time (alternating component of the given signal). (Averaging means a simple average, not root mean

square average.) If the bus signals are viewed with AC selection, a false overshoot or undershoot may show up. Therefore, most of the times, we connect to DC input for observing the waveforms.

A clock, if running will show the states 0 and 1 on the scope. The horizontal gap between the successive rising edges gives us the clock time period. For example, an 8051 using a 12 MHz crystal, there will be states, each of period 0.0825 μ s. The check for this and ALE (address latch enable) simultaneously at two input amplifiers will test the processor activity. Another use of scope is in checking the real-time clock routines and pulse width output routine. Real-time software test and debugging are easy using the scopes. The output signal on a serial port or the output bit on a parallel port test will provide significant information. Scope is usable for testing the delay time routines. We can set three delay parameters in three registers and note the interval taken for the port bit to change with each run. From these three measured intervals, we estimate the actual setting in the register for requisite delay. We then run with this delay parameter setting and test, using the scope and fine-tune to the exact delay setting.

An advantage of scope is its use as a noise detection tool and as a voltmeter. Another use is detection of a sudden in-between transition between '0' and '1' states during a clock period. This debugs a bus malfunction. A *storage scope* is another version of oscilloscope. It stores the signals versus time. Later we analyse the stored activity.

14.3.4 Bit Rate Meter

A bit rate meter is a measuring device that finds the numbers of '1's and '0's in the preselected time spans. How to measure the throughput, the number of bytes per second on a network? Assume that 0xA55A (binary 10100101011010) is sent repeatedly as output bits. The number of 1s multiplied by 16 is the throughput in byte/second. Similarly we can find another bit pattern, and find the expected number of bits in the given time. We can estimate the bits, '1's and '0's in a test message and then use bit rate meter to find whether that matches with the message.

14.3.5 Logic Analyser

After using the simulator, ICE and debug codes in ROM, in the last stage of debugging, we may use a troubleshooting hardware diagnostic tool that records the state (i) as a function of time and (ii) as a function of other states. Logic analyzer can be used in any of these two modes.

Logic analyser is a power tool to collect through multiple input lines (say, 24 or 48) from the buses, ports and records many bus transactions (about 128 or more). It displays these on the monitor (screen) to debug real-time triggering conditions. It helps in sequentially finding the signals as the instructions execute with respect to a reference. One of the bus signal or clock signal is taken as the reference.

A logic analyser can easily debug small-level embedded system. It is a more powerful tool than the scope. Scope views and checks only two signal lines. A logic analyser is a powerful software tool for checking multiple lines carrying the address data and control bits and the clock. The analyser recognizes only discrete voltage conditions, '1' and '0'.

In the *first* mode, the analyser collects the logic states as a function of time and stores these in memory and displays on screen. It tracks the multiple signals simultaneously and successively. There are multiple input lines (24 or 48 or more). We connect the lines from the system and IO buses, ports and peripherals. It collects simultaneously for the duration of the many bus transactions (about 128 or more). It later displays, using this tool, each transaction on each of these on the computer monitor (screen). It also prints the displayed results. The phase differences in each input line also give important clues. It debugs the real-time triggering conditions. It helps in finding the bus signals and port signal status sequentially as the instructions are executed. A variant of the logic analyser also provides the analog measurement when needed.

In the *second* mode, buses are connected to logic analyzer probe pins and the analyser gives the captured states of all the signals at the clock edge. The triggering point for capturing the states can be defined by the user. The triggering point can be defined as *observation* of an illegal op-code or processor at particular startup address or a certain port byte at output.

For example, the analyzer is set to measure at first, second, third, fourth and so on clock edges, up to 64 or 128 or any number (say 2^{20}) clock edges from a start address 0x10000. The analyser gives the address and data bus states in hexadecimal and it gives each control signal state. An advanced version of logic analyzer can also trace the instruction sequences from the observed address and data bus states at the clock edges from the given start address. A software engineer can trace illegal instruction or protected address accesses, when running the codes.

Certain bugs that intermittently arise can also be recorded with a logic analyser by continuous and repeated runs of the system.

Logic Analyser Inabilities A logic analyser does not help on a *program halt due to a bug*. It does not show the processor register and memory contents. If the processor uses the caches, bus examination alone may not help. We cannot modify the memory contents or input parameters during trace and display as we do on a simulator. The effects of these changes are invisible.

With SOC use in embedded systems design, the inner-connections are just not visible to the logic analyzer.

14.3.6 In-Circuit Emulator (ICE)

Instead of the target system that is copied to obtain an embedded system, can we have a separate unit that remains independent of a particular targeted system processor or microcontroller? Yes.

We use a *target* or ICE. Instead of a target circuit, an ICE provides a greater flexibility and ease for developing various applications on a single system instead of testing multiple targeted systems. Figure 14.3(a) and (b) shows emulator and ICE, respectively.

ICE is a circuit for emulating the target system that remains independent of a particular targeted system processor, usable during the development phase for most of the target systems that will incorporate a particular microcontroller chip. It works independently as well as by connecting to the PC through a serial link. It is a target circuit minus target microprocessor or microcontroller.

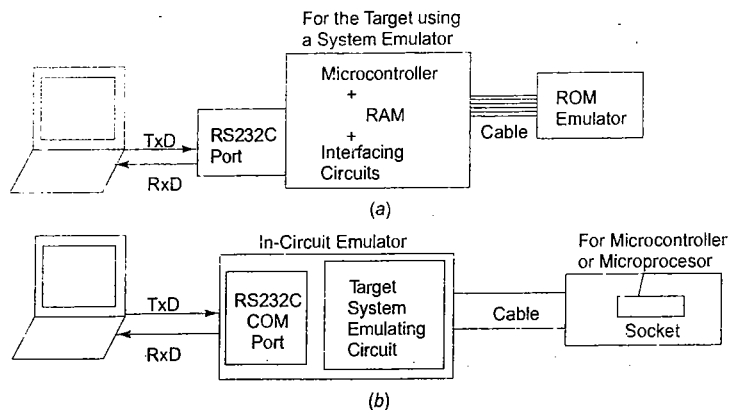


Fig. 14.3 (a) An emulator (b) An in-circuit emulator

ICE is an emulator of the microprocessor of target circuit, such that a host system connects to the ICE through a serial link for debugging purposes. ICE emulates various versions of a microcontroller family during development phase using the remaining part of the target circuit.

ICE is an emulator of microprocessor or microcontroller of target circuit in a target emulating circuit.

How does an ICE differ from target? The target uses the circuit consisting of the microcontroller or processor itself. The emulator emulates the target system with extended memory and with codes-downloading ability during the edit-test-debug cycles. ICE emulates the processor or microcontroller. It uses another circuit with a card that connects to the target processor (or circuit) through a socket.

The back support hardware package and ICE have the subunits listed and explained in Table 14.4.

Table 14.4 Back Support Hardware Package and In-Circuit Emulator (ICE) Subunits

Emulator subunits ¹	Action(s)
Interface circuit	It is for downloading ROM images into EPROM and RAM bytes from the host system into the emulator. It uses a serial (COM RS232C) port of PC (Figure 14.3). It helps in embedding in the program memory part the large application codes directly from the PC. Codes may be developed on the host using a high-level language. For example, the development of the application codes' designer finds it much more easy to write the large application programs instead of keying them in machine codes using 20 keys pad at the emulator.
Socket	A multipin male-female socket to insert a general-purpose processor or DSP or embedded processor or microcontroller, which connects to the ICE through a cable (usually a ribbon cable) and connectors. (See Figure 14.3(b), right corner socket).
External memory	<i>Additional RAM and EPROM or EEPROM</i> , enough for use by most possible targeted systems and their applications.
Emulator-board display unit	<i>A single-line 8- or 12-character display</i> . It is to show the content of memory addresses one by one. Also, it is to show the contents of registers at the various program steps.
Twenty-keys pad	It is to <i>enter data and codes directly</i> by the user locally at the memory addresses. These codes have to be machine codes.
Registers	<i>Additional system registers</i> for the single step as well as full speed test runs during testing of the system.
Connectors	To plug-in this emulator to the interface circuits and other devices and peripherals that are typical to the system. A connector for the target system display module is an example. Another example is for the PC interface circuit.
Target system keyboard	Keyboard user input board equivalent to the target system expected keyboard.
Target system driver circuit	For example, <i>driver hardware</i> for network or motor or solenoid valve or furnace or printer.
Monitor codes	These are in the emulator EPROM or EEROM or at the target system ROM to test and debug with actual target processor or microcontroller and target circuit.

¹Emulators from Orion Instruments, USA embed the logic analyser-like facility (Section 14.3.4). Intel provides the emulators and back support packages for its different processors and microcontrollers.

ICE consists of the following: (i) An emulator pod with a ribbon cable, which extends to a processor or microcontroller socket of the target system [Figure 14.3(b)]. We later on insert the processor IC in that socket.

ter on, we can test the target system also. (Remember that this developed and debugged system is a one-to-one copy for the final, embedded system.) To avoid coupling capacitive effect due to a long cable, we must use as short cable as possible. The pod circuit emulates the target system microcontroller or processor. (ii) The pod links rially to COM RS232C port of a computer. Through this port, the pod gets the downloaded codes from the mputer. The computer program for the emulator monitors completely the bytes at the registers and memory cations. The pod may have some card between its basic circuit and ribbon cable jumper. The replacement of this rd makes it feasible to use the ICE for another version of a processor or microcontroller family.

What about processor core itself accompanying the ICE core? A feature in ARM7 and 9 processors (Section 2.3.3) is that these processors have accompanying ICE subunit. It helps in debugging the targeted hardware.

ICE or emulator disables after the development phase is complete. An actual circuit forms just by copying the codes developed using the ICE. This circuit after the interconnection to the target processor consists of the ed processor, required memory chips and keys and display units or other peripherals. This should work exactly the same and as perfectly as at the end of the development phase that we completed using the emulator ICE. An emulator helps in the development of the system before the final target system is ready.

Motorola provides M68HC11EVM and M68HCEVB as the emulators for 68HC11 microcontroller-based target system. These emulators have the following external connections.

When using an ICE or emulator, software required for implementation phase are the editors, assemblers, assembler, simulators and so on (Section 12.4). The host system is just for downloading the codes to the emulator and for echoing back the codes and data at the various addresses in the emulator memory. A designer needs the host system to save machine-level programming time that can be too much for the sophisticated applications. We can have the additional socket connectors for the different versions of the microcontroller: for example, for emulating a 48-pin version as well as a 52-pin version of the 68HC11.

An ICE 'visionICE 1' is an ICE that has the networking capabilities. The latter imbibes by 10/100 Mbps ethernet connectivity. This lets the ICE accessible to a LAN. Remote debugging is another advantage. It also connects to the serial port of the target system.

A ROM emulator [Figure 14.3(a), right side] emulates only a ROM. The target connects through a ROM socket and also connects to the computer. There is a need during the edit-test-debug cycle for downloading the codes into the target system flash or EEPROM cyclically. The ROM emulator obviates this need. Monitor (Section 14.3.7) codes can be downloaded in ICE ROM. It may run a 'Power On Self Test' (POST) program for bootstrapping. The embedded system when coupled to the RS232C COM port or network port of a computer can use *gdb*, a GNU debugger (it is a downloadable freeware).

4.3.7 Monitor

Monitor is a debugging tool for actual target microprocessor or microcontroller in ICE ROM emulator or in target development board. It also lets host system debugging interface just like an ICE. Monitors from different sources differ in their functioning. One typical monitor does the following.

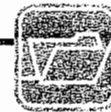
1. Monitor loads the application codes, is also used for corrections in codes and then to test the system. A command for download can download a new application code into the monitor. A command for resetting the program restarts the program. Monitor loads the application (in hex file) from the developing system (at host) that can also be modified later to correct the codes.
2. A part of the monitor runs on host system. Debug monitor codes are downloaded along with the locator binary image. A write and a read command is used to correct or examine the codes at the memory addresses of the system. Monitor controls (as per command from debugger) the execution of application at full speed, as well as by single stepping during debug phase.

3. Monitor controls (inserts, removes, modifies) breakpoints as per command from the debugger. A breakpoint partitions the program into separate segments. When a program segment runs, there is a pause at breakpoint and then test the result is observed after the run and is examined; then, the segment is run. Breakpoints enable program test running between the different program segments.
4. Monitor can be run in single step mode also.
5. Monitor facilitates controlled execution of application and controlled display of executing program status. Table 14.5 lists the target board units with monitor and monitor segments.

Table 14.5 Target Board Subunits Including Monitor

Target Board Subunits	Action(s)
Socket	It is for downloading monitor and ROM images into EPROM and RAM bytes from the host system into the target. It uses a serial (COM RS232C) port of the host system. Codes may be developed on the host using a high-level language.
RAM and interfaces	RAM and interfaces.
Display	Display subunit displays the application codes (as per command from debugger) in full or in segments, the registers and internal RAM or memory addresses data during debugging phase running through the single stepping or breakpoints.
Monitor segments in ROM	Interface commands for interfacing with the host system; command interpreter; loaded application codes; and data.
Twenty-keys pad	It is to enter data and commands of monitor and corrects the codes directly by the user locally at the memory addresses. These codes have to be machine codes.
Connectors	Connectors for display subunit and printer.

Monitor means a ROM resident program at the target board or ROM emulator connected to ICE. It monitors the device applications, the runs for different hardware architecture and is used for debugging.



Summary

The following is a summary of what we had discussed in this chapter.

- System codes are tested on the host system as host system has application development tools, large memory and windows or powerful GUIs. Each module must be tested at the initial stage of its development as well as by integrating all modules. Software can be tested on host machine. It is divided into two parts: hardware (target)-independent code and hardware-dependent code. Hardware-dependent code has fixed start addresses, fixed port and device register and other addresses.
- Simulation by a simulator, which runs on host, helps in system development by simulating target processor or microcontroller, peripherals, devices and network interfaces. Instruction set of target processor or microcontroller simulates on the host in a simulator.
- Volt Ohm meter is useful for checking the power supply voltage at source and voltage-level, port chips power input pins, and port pins initial at start and final voltage levels after the software runs, checking broken connections, improper ground connections and burnout resistances and diodes.

- Oscilloscope is used to test the fast changing signals, their wave forms, overshoots and undershoots at transitions.
- Logic analyser measures logic states on many connections simultaneously. It has two modes of functioning. One mode is to show time on X-axis, and logic states of the clock signal, bus signals and other signals on Y-axis. Second mode is to give address, data bus and other signal states from a trigger point to examine illegal op-codes, access in protected address space and other states as a function of a reference state.
- ICE is used for debugging a target system without using the target processor/microcontroller.
- ASIC and SoC system hardware cannot be tested by laboratory tools, such as logic analyser and ICE.
- Monitor is used to debug software and hardware for the given target processor or microcontroller.



Keywords and their Definitions

Host system	: PC or workstation or laptop on which an application development is done for a target system.
ICE	: An emulator of microprocessor of target circuit, such that a host system connects to the ICE through a serial link for debugging purposes and emulating various versions of a microcontroller family during development phase using the remaining part of the target circuit.
Logic analyser	: A power tool to collect through its multiple input lines (say, 24 or 48) from the buses, ports and many bus transactions (about 128 or more) to display these on the monitor (screen) to debug real-time triggering conditions.
Monitor	: Codes placed in the emulator EPROM or EEROM or at the target system ROM to test and debug with actual target processor or microcontroller-based circuit.
Oscilloscope	: A scope with a screen to display two signal voltages as a function of time. It displays analog as well as digital signals as a function of time.
Simulator	: Software, which runs on host in powerful GUIs environment, helps in system development by simulating target processor or microcontroller, peripherals, devices and network interfaces.
Target system	: A system which has hardware similar to that of the final product and on which the embedded software has to run.
Volt-Ohm meter	: A meter to measure voltage and resistance between two points to test voltage levels at supply rails, broken connection, resistances and diodes.



Review Questions

1. Why is host system used for most stages of development and test and simulation?
2. Give examples of hardware-dependent and hardware-independent codes.
3. What is a target system? How does the target system differ from the final embedded system?
4. What do we mean by application software for a target system?
5. What is back support package? What are the various components of a target emulator? What are the advantages of using an ICE?

6. Explain the use of the following hardware tools: target emulator and ICE.
7. What is the use of a simulator in a development phase?
8. How does a calling of interrupt routine help in testing a design?
9. What is a cross-assembler?
10. What is time mode of a logic analyzer? What is *state mode* of a logic analyzer?
11. What do we mean by a logic analyzer? What is the use of a logic analyser during the development phase?
12. A LED circuit is also a powerful analysis tool. How is it so?
13. What are the uses of an oscilloscope?
14. How will you use a bit rate meter to measure throughput from a real-time system?
15. Why is the I/O instructions platform dependent? Define throughput of an I/O system.

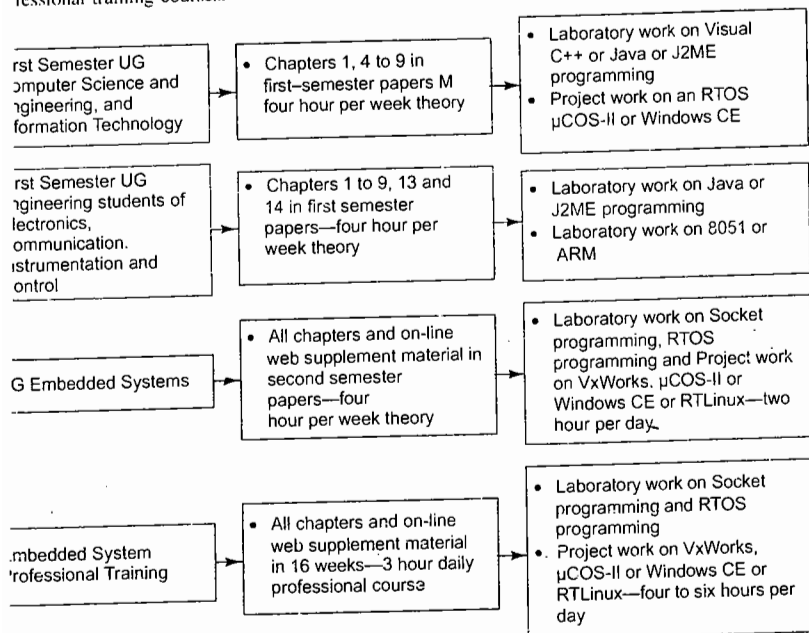


Practice Exercises

16. Which are the popular simulators used?
17. Prepare a list of emulator systems available for various microprocessors, microcontrollers and DSPs.
18. Explain with one example the use of each of the following: debugging capability, device simulation, network simulation and user interface.
19. Explain with one example the use of each of the following software tools: profiler scope, memory usage scope, stethoscope and scope for trace of program flow, scope for memory allocations and uses and scope for code coverage.
20. List prototyping tools with a popular RTOS.

Appendix 1: Roadmap for Various Course Studies

Learned professors and syllabi designers are the best judges. From the author's experience, the roadmap shown in the following figure can be adapted by various disciplines of UG, PG and professional training courses.



Suggested Roadmap for Various Disciplines of UG, PG and Professional Training Courses

Appendix 2: Select Bibliography

A. PRINTED BOOKS

- Graham Phillips, Bill Pierce and John Hardin, "Linux Appliance Design: A Hands-On Guide to Building Linux Appliances", BS Starch Press, 2007.
- Grzegorz Rozenberg, and Frits Vaandrager (Eds.) "Lectures on Embedded Systems: European Educational Forum School on Embedded Systems, Veldhoven", Springer, Nov. 2006.
- Michael Barr and Anthony Massa, "Programming Embedded Systems: With C and GNU Development Tools", 2nd Edition, O'Reilly, Oct. 2006.
- Nicolas Carter, and Raj Kamal (adoption author), "Computer Architecture", Schaum Series TMH Edition, May, 2006.
- Peter Marwedel, "Embedded System Design" -- Springer Verlag, New York 2006.
- Raghavan P, Anil Lad, and Sriram Neelakandan, "Embedded Linux System Design and Development", Auerbach Publications, Taylor and Francis, Dec. 2005.
- Bruno Buoysounouse and Joseph Sifakis, "Embedded Systems Design: The Artist Roadmap for Research and Development", Springer, 2005.
- Tammy Noergaard, "Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers", Newnes, Butter-worth Heinemann, Newton, Mass. USA, 2005.
- Raj Kamal, "Microcontrollers- Architecture, Programming, Interfacing and System Design", Pearson Education, Singapore, 2005.
- Jack Ganssle (Ed.), "The Firmware Handbook", Newnes, Butter-worth Heinemann, Newton, Mass. USA, 2004.
- Jack Ganssle and Michael Barr, "The Embedded Systems Dictionary", CMP Books, 2003.
- Prasad K. V. K. K., "Embedded Real Time Systems: Concepts, Design and Programming -- The Ultimate Reference" Dreamtech, 2003.
- Douglas Boling "Programming Microsoft WINDOWS CE.NET", Microsoft, USA, 2003.
- John Catsoulis, "Designing Embedded Hardware", 2nd Edition, O'Reilly, 2003.
- Prasad K. V. K. K., Vikas Gupta, Avinash Dass, Ankur Verma, "Programming for Embedded Systems—Cracking the Code", Wiley, New Delhi, 2002.
- Jonathan W. Valvano, "Embedded Microcomputer Systems- Real Time Interfacing", Thomson, Brooks/Cole, 2002.
- Stephen Palmer and John Felsing, "A Practical Guide to Feature-Driven Development", Prentice Hall, 2002.
- Stuart R. Ball, "Embedded Microprocessor Systems: Real World Design", Butter-worth Heinemann, Newton, Mass. USA, 1996. (2nd Edition, May 2002).
- Phillip A. Laplante, "Real-Time Systems Design and Analysis – An Engineer's Handbook, 2nd Edition, IEE Press, USA, 1997 (Prentice Hall of India, Third Indian Reprint, April, 2002).

20. Raj Kamal, *Internet and Web Technologies*, Tata McGraw-Hill, 2002.
21. Bob Zeidman, *Designing with FPGAs and CPLDs*, CMP Books, Sept. 2002.
22. Demuth B. and D. Eisenreich, *Designing Embedded Internet Devices*, Butterworth Heinemann, July 2002.
23. Al Williams, *Embedded Internet Design*, McGraw Hill, July 2002.
24. Miro Samek, *Practical StateCharts in C/C++—Quantum Programming for Embedded Systems*, CMP Books, July, 2002.
25. Tim Jones M., *TCP/IP Applications Layer Protocols for Embedded Systems*, Charles River Media, June 2002.
26. Steve Heath, *Embedded System Design: Real World Design*, Butterworth Heinemann, Newton, Mass. USA, May 2002.
27. Michael J. Pont, *Embedded C*, Addison Wesley, April 2002.
28. Lewis D., *Fundamentals of Embedded Software: Where C and Assembly Meet*, Prentice Hall, Feb. 2002.
29. Dreamtech Software Team, *Programming for Embedded Systems—Cracking the Code*, Hungry Minds, April 2002.
30. Craig Hollabaugh, *Embedded Linux Hardware and Software*, Addison Wesley, March 2002.
31. Macii, Benini and Poncino, *Modern Design Technologies for Low Energy Embedded Systems*, Kluwer Academic Publishers, March 2002.
32. George Pajuri, *Unix Device Drivers*, Pearson Education, Indian Reprint, 2002.
33. Ed Sutter, *Embedded System Firmware Demystified* (with CD), CMP Books, Feb. 2002.
34. Frank Vahid and Tony Givargis, *Embedded System—A unified Hardware/ Software Introduction*, John Wiley and Sons, Inc. 2002.
35. Steve B. Farber, *ARM System-on-Chip Architecture*, 2nd Edition, Addison Wesley & Benjamin Cummings, 2002.
36. Wayne Wolf, *Modern VLSI: System on Chip Design* Pearson, Jan. 2002.
37. Jim Ledin, *Simulation Engineering- Build Better Embedded Systems faster*, CMP Books, Aug. 2001.
38. Todd D. Morton, *Embedded Microcontrollers*, Prentice Hall, New Jersey USA 2001.
39. Adam Drozdek, *Data Structures and Algorithms in C++*, Brooks/Cole Thomson Learning, 2001.
40. Joseph Lemieux, *Programming in the OSEK/VDX Environment*, CMP Books, Oct. 2001.
41. Thomas D. Burd and Robert W. Brodersen, *Energy Efficient Microprocessor Design* Kluwer Academic Publishers, Oct. 2001.
42. Eric Giguere, *Java 2 Micro Edition-The ultimate Guide to Programming Handheld and Embedded Devices*, John Wiley, USA, Canada 2000.
43. John Uffenbeck, *The 80x86 Family*, 3rd Ed., Pearson Education India, 2002.
44. Ali Mazidi M. and J.G. Mazidi, *The 8051 Microcontroller and Embedded Systems*, Pearson Education, 2000, First Indian Reprint, 2002.
45. Jeremy Bentham, *TCP/IP Lean Web Servers for Embedded Systems*, CMP Books, USA 2000. (Also 2nd Edition, 2002).
46. Sundrajan Sriram, and Survra S. Bhattacharya, *Embedded Multiprocessors- Scheduling and Synchronization*, Marcel Dekker, Inc., New York, USA 2000.
47. Raj Kamal, *The Concepts and Features of Microcontrollers (68HC11, 8051 and 8096) -Includes Programmable Logic Controllers*, S. Chand & Co. (Originally Wheeler Pubs.), New Delhi, 2000.
48. Gary Nutt, *Operating Systems—A Modern Perspective*, Addison Wesley Longman, Inc., USA, 2000 (Pearson Education Asia Singapore, India Reprint 2000).
49. Steve White, *Digital Signal Processing*, Thomson Learning – Delmar, 2000 (First Indian Reprint, Vikas Publishing House, 2002).
50. Filip Thoen and Francky Catthoor, *Modeling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems*, Kluwer Academic Publishers 2000.
51. Sommerville, *Software Engineering*, Addison Wesley, Reading, MA, USA, 2000.
52. Rainer Laeupers, *Code Optimization Techniques for Embedded Processors: Methods, Algorithms and Tools*, Kluwer Academic Publishers, Oct. 2000.
53. William A. Shay, *Understanding Data Communications and Networks*, 2nd Edition, Thomson Learning – Brooks/Cole, 1999 (First Indian Reprint, Vikas Publishing House, 2001).

54. Randall S. Janka, *Specification and Design Methodology for Real-Time Embedded Systems*, CMP Books, Nov. 2001.
55. Scott Rixner, *Stream Processor Architecture* Kluwer Academic Publishers, Nov. 2001.
56. Tim Wilmshurst, *An Introduction to the Design of Small Scale Embedded Systems - with examples from PIC, 8051, and 68HC05/08 Microcontrollers*, Palgrave, Great Britain, 2001.
57. Pflieger S. L., *Software Engineering Theory and Practices*, Pearson Education, USA Singapore, India Reprint 2001.
58. Rogers S. Pressman, *Software Engineering*, 20th Edition, McGraw-Hill, 2001.
59. Arnold S. Berger, *Embedded Systems Design—An Introduction to Processes, Tools and Techniques*, CMP Books, Nov. 2001.
60. Kirk Zurell, *C Programming for Embedded Systems*, CMP Books, Feb. 2002.
61. Wayne Wolf, *Computers as Components—Principles of Embedded Computing System Design*, Academic Press (A Harcourt Science and Technology Company), USA, 2001.
62. Jack Ganssle, *"The Art of Designing Embedded Systems"* (Edn Series for Design Engineers), Newnes, Butterworth Heinemann, Newton, Mass. USA, 2000.
63. Jane W.S. Liu, *Real Time Systems*, Pearson Education, 2000 (First Indian Reprint 2001).
64. Joseph L. Weber, *Using Java™ 2 Platform*, Que Corporation, Reprint by Prentice Hall of India, New Delhi, May 2000.
65. Jack W. Crenshaw, *Math Toolkit for Real-Time Programming*, CMP Books, Aug. 2000.
66. David E. Simon, *An Embedded Software Primer*, Addison Wesley Longman, Inc., USA, (Pearson Education Asia) Singapore, USA 1999 (India Reprint 2000).
67. Barry Kauler, *Flow Design for Embedded Systems—A Simple Unified Object Oriented Methodology*, CMP Books, Feb. 1999.
68. Franz J. Rammig (Ed.), *Distributed and Parallel Embedded Systems*, Kluwer Academic Publishers, Netherlands, 1999.
69. Alessandro Rubini, *Linux Device Drivers*, O'Reilly, USA, June 1999.
70. Luis Miguel Silveira, Srinivas Devadas, Ricardo A. Reis, *VLSI: Systems on a Chip*, Kluwer Academic Publishers, Dec. 1999.
71. John Hyde, *USB Design by Example*, John Wiley & Sons, Inc., New York, 1999.
72. Jean J. Labrosse, *Embedded Systems Building Blocks*, 2nd Edition, CMP Books, Dec. 1999.
73. Jack G. Ganssle, *Art of Programming Embedded Systems*, Butterworth Heinemann, Newton, Mass., USA, 1999.
74. Michael Barr, *Programming Embedded Systems in C and C++*, O'Reilly, USA Aug. 1999 Reprinted Shroff Pubs, India Reprint August 1999.
75. Myke Predko, *Programming and Customizing the 8051 Microcontroller*, McGraw-Hill, 1999, Third Reprint Tata McGraw-Hill, 2002.
76. Jean J. Labrosse, *MicroC/OS-II The Real Time Kernel*, R&D Books, an Imprint of Miller Freeman, Inc. Lawrence, KS 66046, USA, 1999. (Also 2nd Edition in 2002 from CMP Books).
77. Bruce Powel Douglass, *Real-Time UML—Developing Efficient Objects for the Embedded Systems*, Addison Wesley Object Technology Series, 1998.
78. Calcutt M.C., F.J. Cowan, and G.H.Parchizadeh, *8051 Microcontrollers—Hardware, Software and Applications*, Arnold (and also by John Wiley), 1998.
79. Rick Grehan, Robert Moote and Ingo Cylix, *Real-Time Programming—A guide to 32-bit Embedded Development*, Addison Wesley, 1998.
80. John A. Stankovic, Marco Spuri, Krithi Ramamritham and Giorgio C. Buttazzo, *Deadline Scheduling for Real-Time Systems—EDF and Related Algorithms*, Kluwer Academic Publishers, Netherlands, Oct. 1998.
81. Stuart R. Ball, *Debugging Embedded Microprocessor Systems*, Butterworth Heinemann, Newton, Mass. USA, 1998.
82. Niall Murphy, *Front Panel—Designing Software for Embedded User Interface*, CMP Books, June 1998.
83. M. Costanzo, *Programmable Logic Controllers—The Industrial Computers*, Arnold (and also John Wiley) 1997.
84. Cady F. M., *Software and Hardware Engineering—Motorola M68HC11*, Oxford University Press, 1997.

85. Cady F. M., *Microcontrollers and Microcomputers—Principles of Software and Hardware Engineering*, Oxford University Press, New York, 1997.
86. Balarin F., M. Cludio, A. Jurecska, H. Hsieh, A. L. Lavagno, C. Paasseroni, A. E. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-Software Co-Design of Embedded Systems: A Polis Approach*, Norwell, MA, Kluwer Academic Publishers, June 1997.
87. John Forrest Brown, *Embedded System Programming in C and Assembly*, Van Nostrand Reinhold, New York, USA, 1996.
88. Peter Spasov, *Microcontroller Technology- The 68HC11*, 2nd Edition, Prentice Hall, Englewood Cliffs, NJ, 1996.
89. Fred Halsall, *Data Communication, Computer Networks and Open Systems*, 4th Edition, Pearson Education, 1996 (Fourth Indian Reprint, 2001).
90. Silberschatz and P.B. Galvin, *Operating Systems*, Addison Wesley, Reading, MA, USA, 1996.
91. Peter Marwedel, and Gerl Gossens, *Code Generation for Embedded Processors*, Kluwer Academic Publishers, June, 1995.
92. Daniel Tabak, *Advanced Microprocessors*, McGraw-Hill, USA 1995.
93. Gajski, Daniel D., Frank Vahid, Sanjiv Narayan and Jie Gong, *Specification and Design of Embedded Systems*, Englewood Cliffs, NJ, Prentice Hall, 1994.
94. Franklin G. F., J. D. Powell and A. Emami-Naeini, *Feedback Control of Dynamic Systems*, 3rd Ed., Addison Wesley, Reading, MA, USA, 1994.
95. Stewart J. W., *The 8051 Microcontroller—Hardware, Software and Interfacing*, Prentice Hall, 1993.
96. Walter J. Grantham and Thomas L. Vincent, *Modern Control Systems—Analysis and Design*, John Wiley, 1993.
97. Hintz K. J. and Daniel Tabak, *Microcontrollers—Architecture, Implementation and Programming*, McGraw-Hill, 1992.
98. Jack G. Ganssle, *Art of Programming Embedded Systems* Academic USA, 1992.
99. Greenfield G. D., *The 68HC11 Microcontroller*, Saunders College Publishing, 1991.
100. Peatman J. B., *Design with Microcontrollers and Microcomputers*, McGraw-Hill, 1988.

B. WEBSITE REFERENCES

1. <http://www.dspvillage.ti.com> [For Texas Instruments DSP Processor, Section 1.2.4, 2.3.6].
2. <http://www.mentor.com/seamless> [For Section 1.6].
3. <http://www.ti.com/sc/docs/asic/modules/arm7.htm> and [arm9.htm](http://www.ti.com/sc/docs/asic/modules/arm9.htm) [For Section 2.3.3].
4. <http://www.arm.com> [Section 2.3.3, For ARM Processors].
5. <http://www.ti.com/sc/docs/psheets/abstract/apps/spra638a.htm> [For Section 2.3.6].
6. <http://www.eembc.org> [For benchmarking of performances of embedded Systems, Sections 2.6 and 13.5.3].
7. <http://www.java.sun.com/products/javacard> [For Section 5.7.5].
8. http://www.webopedia.com/TERM/N/operating_system.htm [For Section 8.1].
9. <http://www.wrs.com> [For Section 9.3].
10. <http://www.osek-vdx.org> [For Section 10.2].
11. <http://www.linuxdoc.org> [For Section 10.3].
12. <http://www.cs.ucr.edu/esd> [For Computer Sciences Embedded System Design website of University of California, Riverside, Section 11.2].
13. <http://www.ee.surrey.ac.uk/Personal/R.Young/java/html/cruise.html> [For Section 11.3].
14. <http://www.borg.com/~jglatt/tut/miditutr.htm> [For tutorial on MIDI, Section 12.1].
15. <http://www.xtec.es/rtee/eng/tutorial/midi.htm> [For MIDI interface, Section 12.1].
16. <http://www.misra.org.uk> [For Section 12.2 and for Guidelines for the Use of the C Language in Vehicle Based Software of MISRA (Motor Industry Software Reliability Association)].
17. <http://www.research.ibm.com/secureystems/scard.htm> [For Section 12.4].
18. <http://www.home.hkstar.com/~alanchan/papers/smartCardSecurity/> [For Section 12.4].

19. <http://www.e-insite.net/edmag/> [For subscribing to a popular embedded system magazine].
20. <http://www.EETAsia.com> [For subscribing to a popular embedded system magazine].
21. <http://www.eet.com/embedsub> [For subscribing to a popular embedded system magazine].
22. <http://www.embedded-computing.com/letter> [For subscribing to a popular embedded computing system magazine].
23. <http://www.goembedded.com> [A popular embedded system site].
24. <http://www.webopedia.com> [A popular world wide used encyclopedia on website to clarify the key-terms in various topics].

C. PRINTED JOURNAL PAPER REFERENCES

1. Huan Li, Krithi Ramamritham, Prashant Shenoy, Roderic A. Grupen and John D. Sweeney, "Resource Management for Real Time Tasks in Mobile Robotics", *Journal of Systems and Software*, 80(7), 962–971, 2006.
2. G. R. Gaud, N. Sharma, K. Ramamritham, S. Malewar, "Efficient Real Time Support for Automotive Applications: A case Study", *Proc. of 12th IEEE International Conference on Embedded Real Time Computing Systems and Applications* 2006.
3. Lars-Berno, Fredriksson and Kvaser, "CAN for Critical Embedded Automotive Networks" *IEEE Micro*, 22(4), 28–35, 2002.
4. Wayne Wolf, Burak Ozer and Tichan Lu, "Smart Cameras as Embedded Systems", *IEEE Micro*, 22(5), 48–55, 2002.
5. Jean-Louis Brelet, "Exploring Hardware/ Software Co-design with Vertex-II Pro FPGAs" [*Xcell Journal*, pp. 24–29, Summer issue, 2002].
6. Chi-Ying Liang and Huei Peng, "Optimal Adaptive Cruise Control with Guaranteed String Stability" *Journal Vehicle System Dynamics*, 31, pp. 313–330, 1999.
7. Tadao Murata, "Petri Nets, Properties, Analysis and Applications", *Proc. IEEE*, 77(4), 541–580, 1989.

11, 1394b) is a 165
 Port 140
 63
 Port 140
 is a bus 163
 Peeksum16 552
 MboxAccept 438
 MboxCreate 437
 MboxFend 438
 MemCreate 424, 425
 MemGet 424, 425
 QFlush 446, 447
 QPend 446, 447
 MsgPointer 447
 22
 138, 226
 8
 16F876 32
 8
 0508 4
 11 12, 145, 157, 201,
 205, 207, 217, 223, 224
 11 74, 75, 77, 117
 11 RTC 227
 11xx 8
 11xx, HC12xx,
 16xx 32
 12 12, 117
 16 117
 Cxxx 32
 0.3 455
 196 117, 145
 2.11 WLAN 151
 2.11a to 802 178
 2.11b 178
 51 63, 64, 65, 66, 67, 69,
 70, 71, 72, 77, 117, 204,
 205, 206, 209, 217
 51 two 145
 51, 8051MX 32
 8051MX 517
 80C51 4
 80x86 7, 32, 89, 200, 201,
 202, 205, 209, 211, 226,
 639
 8250 138
 8255 145
 8255 ports 69
A
 a circular queue 328
 CMOS 10
 A CODEC 9
 a critical section 214
 design process 37
 free running counter 227
 function 311
 global variable 258
 A hash table 247
 a keyboard 79
 A linker 625
 a list file that 624
 A list is a data structure 249
 a locator program 111
 mobile phone 355, 375, 380
 mobile phone device 377
 A PCI driver 168
 a pipeline of 87
 Queue for 446
 real-time clock 18
 serial 160
 serial bus 160
 serial IO bus 159
 software timer 18
 A task 309
 a task 310
 A timer cum counting
 device 153
 a touch screen 380

A watchdog timer 158
 a wireless protocol 160
 A/D 217
 A0-A7 75
 abstract class 593
 class 297
 Abstraction 37, 38
 ACC 574, 575, 576, 577,
 578, 579, 580, 581, 582,
 583, 584, 585, 586
 tasks 585
 Accelerator 9, 35
 Accept 338
 function 491
 Active class 297
 object 297
 ActiveSync 176
 ACVM 43, 193, 274, 275,
 276, 277, 278, 283, 284,
 308, 312, 315, 321,
 373, 374, 388, 512, 513,
 514, 515, 516, 518, 519,
 519, 520
 ACVM Hardware
 Architecture 517
 AD0, AD1 75
 AD0-AD7 68
 adaptive control 580, 582
 cruise control 579
 ADC 15, 16, 48, 82, 83
 531, 532, 533, 536, 637
 ADC, DAC 49
 Add number 607
 Address Bus 72, 73, 84
 resolution 627
 ADFG 278
 ADV 73
 AHB 90
 Airbag 575
 alarm signals 576

ALE 68, 73
 Alignment 101
 Allocation 378, 424
 ALU 7, 18, 34, 84, 85
 AMBA 90, 169
 AHB 169
 APB 166, 169, 170
 analog to digital conversions 305
 anonymous object 296, 298
 AOU 84, 87
 APEG 292, 293
 aperiodic 401
 tasks 385
 APIs 267, 353, 480
 applet 268
 application 28, 29
 Specific Instruction-Set Processor
 (AS) 6
 Specific System Processor
 (ASSP) 6
 architecture 40, 63
 arguments 256
 Arithmetic and Logical Unit
 (ALU) 5
 instructions 65
 Arithmetical 91
 ARM 8, 32, 36, 89, 90, 93, 101,
 102, 114, 205, 410, 623, 636, 641
 ARM family Cortex-M3, Aumel 9
 ARM, 68HCxxx, 80x86 7
 ARM11 93
 ARM7 91, 202, 205, 209, 212, 213,
 217
 ARM9TM 92
 array 245, 246
 (one dimensional vector) 250
 ARS 84, 85
 AS 73
 as mutex 426
 object-oriented 40
 per user requirements 37
 ASCII 373, 608, 639
 ASCII code 146
 ASCII codes 149
 ASIC 31
 ASICs 30, 31, 652
 ASIP 33, 49, 51, 573, 635, 636
 Assembler 20, 21, 25, 27, 621,
 622, 624
 assembly language 21, 25
 Language Programming 235
 language using 20
 Assert 650
 ASSP 36, 37
 asynchronous 365, 366, 381
 IO 366
 serial 130
 Serial input 132
 Serial Output 132
 atomic 327
 operation 257, 326
 attributes 276
 Automatic 119
 Chocolate Vending Machine
 (ACVM) 43, 299
 Automobile 574
B
 Banking 28
 Baud or Bit Rate Control 154
 per second 132
 rate 137, 138, 140
 rates 141
 BCC 494, 583, 584, 586
 bedded 266
 before the actual 108
 behaviour 276
 BG 80
 BG0 82
 BG1 81
 Big Endian 102
 binary 315
 semaphore1 463
 Bind function 491
 Bit Manipulation 65
 rate meter 655
 Transfer 91
 BIU 84, 85
 Blind Counting
 Synchronization 152
 block 229
 driver 223
 file system 365
 blocking 420
 Bluetooth 48,
 143, 175, 177, 178, 569,
 570, 572, 573, 574, 576
 device 49
 Bootps 171

bottom-to-top design if it 38
 -up-design approach 236
 bounded buffer 325, 326
 BR 81, 82
 Branch target cache 86
 BSD 363
 BT Cache 84
 buffers 321, 331
 Burning 632
 bus 14, 15
 Arbitration 80, 81, 82
 controller 81
 master 80, 82
 request 81
 Requests 80
 width 84
 busy wait loop 396
 Button 488, 493
 byte codes 266
 Manipulation 65
 order 641
C
 C 237, 238, 239, 242, 250, 252,
 256, 264
 language program 22
 or C++ 236
 program 23
 C/C++ 264
 C++ 263, 264, 383
 Cache 13
 Disable Instructions 42
 caches 12, 103
 Call 190, 191, 193
 called wireless 178
 camera 49, 537
 CAN 161, 164
 Bus 162
 is a 163
 capture register 153
 card 596, 597
 catch 195
 CCD 48, 49, 194, 531, 532, 533,
 535, 567
 Preprocessor 534
 CCDDSP 48
 CCDP 535
 CCDSP 49
 CDC 267

Index

Index

- FG 280
 editor 624
 ular 479
 challenges 41
 r 229
 driver 223
 aracteristics 4
 urge 47
 pump 10
 Pump Circuit 45
 ecksum 32, 174, 548
 uit glitches 384
 ular 639
 queue 242, 245
 SC 62, 88, 98
 ss 263, 265, 266, 268,
 276, 295, 296, 297, 516, 539,
 570, 571, 572, 608
 diagram 298, 299, 515, 516,
 533, 534, 538, 539, 570, 580,
 582, 593, 608
 inheritance 622
 Task 540, 609
 sses 262, 267, 517, 570
 assification 52
 ussify embedded 52
 _DC 267, 268
 ck 10, 84
 frequency 6, 7
 oscillator circuit 10
 Rate Reduction 41
 ose 340, 364, 365, 554
 ose () 343
 lose socket 492
 MB 579
 MOSs 36
 O 640
 ode 19
 segment register (CS) 11, 85
 ODEC 10, 305, 532, 535, 634
 o-designing 634
 odeTest 654
 ollaboration diagram 298
 ollision mitigation brake 575
 OM 48, 49, 136, 138, 221
 OM port 137
 OM1 78
 OM2 77, 226
 ommand 605, 610
 ommand key 604, 611, 612
- compare register 152
 compares 637
 Compile 651
 compiler 27, 621, 622
 componentization 480
 Computer Networking 28
 computers and mobile 479
 Concurrent processes 275
 processing 23
 processing program 276
 concurrently processing 299
 condition 252
 Conditional statements 254
 Conditions 251
 Conductor class diagram 571
 Configuration files 239
 Conformance 494
 connect 340
 function 492
 connectionless 342
 connection-oriented protocol 342
 Constraints 5
 contact-less 593
 CONTEXT 211
 context 212,
 305, 306, 310, 311, 312, 313, 314,
 315, 355, 372, 388, 390, 391,
 394, 395, 386, 638
 at 213
 switch 244, 305, 310, 311, 327,
 390, 396
 switching 211, 212, 213, 309,
 355, 381
 Saving 217
 Control 489
 Area Network 576
 Bus 72, 73, 84
 register 78, 640
 controller 80, 82, 83, 143
 and on 169
 Controls 492
 Cooperative schedulers 392
 scheduling 385, 386
 Core2 88
 CORTEX-M3 32
 COS 357
 counting device 152
 semaphore 321, 325, 331, 426,
 434, 463, 467
 coverage 654
- cPCI 167
 CPSR 212, 213
 CPU 7, 34, 307, 310
 load 642
 machine cycles 11
 register 305, 306, 311, 313
 CRC 163, 164, 175, 179
 CRC32 548, 562
 Create 338
 critical 330, 396
 section 214, 318, 328, 329, 395,
 398, 397, 465, 467, 487, 495
 cross 650
 compiler 649
 Assembler 27, 621, 624, 625
 Compiler 621, 625
 CS 88
 CSWT 248, 249
 Cursor key 604, 605, 610, 612
 cyclic order in an 258
 scheduling 386
- D**
- D0-D7 68, 75
 DAC 15, 16, 48, 82, 83
 daisy chaining 80, 81
 DAS 576, 583
 Data 241
 Acquisition Systems 118, 119
 being 101
 buffer 640
 Bus 72, 73, 84
 cache 84, 86
 Direction Register 145
 encapsulation 262, 375
 flow 275
 Flow Graph (SDFG) 281
 memory 68, 74
 Register 78
 structure 243, 244, 246, 247, 248
 Transfer 65, 91
 Transfer Instructions 64
 type declarations 236
 types 266, 641
 Datagram 342, 541
 DCE 137
 DCT 535
 (IDCT) 33
 DCTs 49, 533, 535
- DDRO 141
 deadline 215, 262, 392, 216
 Deadlock 329, 330
 de-allocation 372, 424
 Debug 651
 debugger 627, 650, 659
 debugging 595, 622, 652
 tools 26, 40
 Declare 525
 Decoder 69
 Decryption engine 9
 Definition 3
 Delay zitters 643
 deleting 245
 deletion 246
 deletions 247
 demultiplexers 14, 53, 74
 De-Registering 498
 DES 595
 design 5
 Metrics 38, 39, 513, 532, 570,
 578, 594, 607
 Process 37, 38, 41
 stages 40
 device 370, 408, 489
 address 78
 addresses 77, 631
 buffer management 362
 control 639
 data 639
 driver 24, 25, 199, 200, 223,
 224, 235, 256, 409, 498, 652
 driver ISR 221, 363
 drivers 211, 222, 220, 223, 227,
 228, 354, 372, 408, 480, 497
 Management 24, 354, 361, 363
 manager 25, 364
 programmer 21, 627, 630, 632,
 633, 631
 software optimization 410
 status 639
 buffer addresses 306
 driver 195, 382
 driver modules 496
 drivers 195
 sensitive 640
 DFG 277, 278, 279, 282
 DFGs 275, 280
 DHCP 171
- Dhystone for 6, 642
 DI 207
 digital 194
 camera 9, 48, 94, 120, 279, 531,
 532, 533, 534, 536, 631
 camera hardware
 architecture 535
 Camera Software Architecture
 Signal Processor (DSP) 32, 96
 Direct memory 50
 memory access 73
 memory access controller 51
 director robot 389
 Disable the interrupts 328
 disabling 329
 discrete cosine transformation
 (DCT) 9
 dispatch table 306
 display co-processor 533
 DISR 370
 dissembler 624
 distributed embedded systems 160
 division by zero 210
 DLL 484
 DLLs 480
 DMA 99, 218, 327
 chip 8237 167
 controller 218
 DMAC 219
 DPCs 260
 DRAM 109
 DRAM, EDO RAM, SDRAM 109
 driver 639
 DSP 32, 33, 35, 50, 93, 96, 104,
 535, 635, 636, 637,
 DSPs 36, 95, 577
 DSSS 152, 179
 Dual Core 33, 88
 core processor 35, 289
 dynamic binding 360
 data memory 360
 memory 501
 programming model 400
 scheduling 290
- E**
- E_SMS 609, 612
 E2PROM 12
- each frame 175
 ECB 427, 428, 429, 437, 438, 439,
 446, 447, 448, 463, 467, 469, 470
 ECG 118
 EDF 385, 398
 Edit 651
 editing 622
 Editor 25, 621, 622
 Edit-test-debug 620
 EDN 642
 EDO RAM 108
 EEPROM 13, 110, 597
 EI 207
 EISA 168
 Embedded hardware 45
 Java 267
 processor 5, 34
 Software 19
 system 3, 4, 5
 system hardware 5
 systems 28
 emption 381, 394
 emptive scheduler 375, 496
 emulator 620, 623, 650, 656, 657,
 658
 encoder 147
 Encryption engine 9
 Engineering cost 39
 software 26
 EPROM 106, 107, 633, 634
 Ethernet 175, 190
 Ethernet adapter, GPS 143
 event 284, 369, 612
 control 356
 Control Blocks */ 440
 flag-group 412
 Functions 358, 487
 OSSemCreate 427
 register 359
 controlled Program Flow 282
 Flag 359
 events 368, 494, 513, 539, 570, 606
 Examples of 27
 exception 192, 195, 197, 202, 211,
 266
 handler 203
 handling signals 486
 Exceptional handler 195
 handling 264

option-handling functions 460
 options 201, 204, 333
 option time 401
 output 455
 output interrupt 72
 output pins 63
 Memory 68

operation key 46
 369
 554
 SET 473
 152, 178
 276
 214, 215, 244, 245, 246, 247,
 321, 445, 456, 467, 470,
 501, 502, 503, 505
 method 412
 504
 24, 242, 223, 483
 descriptor id 306, 365, 473
 management 354
 system 47, 364, 365, 496, 595
 impulse response filter 281
 209
 278, 279, 280
 are 165
 level interrupt service
 routine 199
 priority scheduler 502
 scheduling 399
 12, 13, 20, 120, 121, 627,
 632
 45
 memory 19, 44, 51, 107, 194, 625
 R 198, 369
 Multipoint Processing Units 7
 84, 86, 99
 135
 237, 252
 MPEG compression 49
 RAM and ROM 101
 SOC 90
 super harvard architecture
 single 95
 e priority inversion 330
 497
 realization of system design 42

foundation classes are GUIs 264
 FPGA 31, 74, 75
 FPGAs 634
 free running counter 153, 159
 frequencies 151
 FRS 84, 86, 99
 FSM 283, 285, 286, 288, 296
 State Table 285
 FTP 171
 Full Duplex 131, 133, 140, 142
 function 239, 240, 256, 257, 258,
 259, 260, 261, 262, 294, 312,
 313, 314
 function argument 257
 Calls 258
 Pointers 259
 queues 262
 Queues 260, 261
 functions 238, 241, 263, 531
 call 257

G
 GAL 74, 75
 game 479
 General Purpose
 Microprocessor 35
 Purpose Processor 6, 86
 global variables 237
 glue-circuit 74
 Goal 351
 of testing 42
 GPP 7
 GPS 576
 Graphic Accelerator 10
 Graphics processor 9
 GUI 254, 408, 497, 513, 531
 GUI notification2 573
 GUIs 305, 410, 606, 624

H
 half 142
 Duplex 131, 133, 140
 Handle 482, 483, 484, 485, 486, 493,
 496
 handshaking 145
 Hand-shaking signal 144, 225
 Hard Real-Time 380
 real-time systems 381

hardware 5, 313, 518, 541
 Architecture 534
 components 52
 internal timer 156
 interrupt 192, 200, 201
 scheduling 290
 software trade-off 634
 Harvard 629
 Architecture 89, 103
 memory 630
 hash table 242, 248
 HDLC 138
 header 237
 = UDP 552
 files 238
 heap 482
 heavyweight 307
 Heuristic 400
 hex 627
 File Intel 632
 File S-Recoded 632
 Hierarchical RTOS 378
 High definition 33
 level language 22
 performance 106
 performance processors 84
 processor performance 104
 Speed Serial 569
 level Language
 Programming 236, 237
 level program 237
 host 623, 624, 625, 649, 650
 HotSync 176
 HSDFGs 293
 HSDPA 96
 HSTL 150
 HTTP 171, 172, 173
 Huffman 534
 Huffman coding 533, 535

I
 I/O 639
 Device 629, 639
 instructions 641
 management 354
 stream 264
 Subsystems 365
 I2C 162
 I2C bus 66, 161, 162

IBM 136, 137, 138, 144
 ICE 620, 626, 627, 649, 652, 656
 IDE 408, 620, 622, 623, 624
 identity 276
 IE 216
 IEE 582
 IEM 90
 if, if-else, else-if 237
 image pixels 34
 in DSP, TMS320C64x DSP 98
 linker 627
 Unix 352
 Circuit-Emulators 53
 include 238
 "preHandler.c" 238
 directive 237
 Infinite 252, 254, 394
 Loop 251, 373
 inheritance 276, 277, 278
 Input captures 637
 Input/Output (I/O) Subsystem 366
 inserting 245
 insertion 246, 247
 Inst. Cache 84
 Instance field 265
 method 265
 Instruction cache 86
 cache Data cache 99
 level parallelism unit, 86
 pointer 85
 pointer (IP) 11
 queue 86
 Level Parallelism 104, 105
 instructions 91, 101, 385
 INT0 69, 72
 INT1 69, 72
 Integrated development 26, 27
 Intel 636
 Hex 628, 629
 XScale 94
 Inter Processor Communication
 (IPC) 291
 interface 277
 interfacing 68, 72
 circuit 68
 Internal RAM 12
 Internet Enabled Systems 170
 layer 543
 interoperability 352
 Interpreter 25, 27, 621, 624

interprocess 354
 Communication 330, 463
 communication model 254, 275
 interrupt 18, 207
 flags 208
 Handler 18
 latencies 215, 381, 400, 401
 latency 213, 214
 mask 212
 mechanism 198
 Pending Register 208
 Request 145
 service 496
 Service Deadline 215
 Service Functions 461
 service routine 18, 240, 261, 289
 Service Threads 373
 Servicing (Handling)
 Mechanism 203
 signal 145
 source 200, 208, 209
 vector 203
 Vector Table 205
 handling functions 461
 interrupts 4
 in 8051 71
 handling 18
 intLock 590
 intUnlock 590
 inversion problem 318
 IO Addresses Mapped IO 77
 bus 76
 Byte Programming 66
 device 79, 130, 151
 devices map 109
 Interfaces 13
 Port Bit Programming 66
 Ports 66
 Ports, IO Buses 13
 stream 268
 System 47
 mapped 629
 IP 31, 174, 216, 539, 555, 560,
 643, 652
 address 2, 170, 171
 Instruction 88
 Pkt 542
 IPC 293, 312, 316, 317, 319, 331,
 332, 333, 334, 335, 337, 340,
 341, 342, 345, 354, 373, 408,

428, 437, 439, 445, 449, 455,
 459, 462, 470, 486, 494, 504,
 522, 636
 functions 339
 IPCs 23, 30, 290, 326, 328, 356,
 371, 372, 440, 469, 472, 484,
 495, 541, 543, 544, 584, 596
 IPHeaderSelPkt 561
 IPPkt 539, 543, 546
 IPPktStream 545
 IPv4 174
 IPVerHdr 559
 IQ 84
 IR source 151
 IrCOMM 177
 IrDA 51, 175, 176, 177
 adapter 143
 protocol 152
 IRQ 209
 is 361
 ISA 168
 ISA Bus 167
 Iso-synchronous 134
 ISR 71, 192, 193, 194, 195, 197,
 198, 199, 200, 201, 202, 204,
 205, 206, 207, 208, 210, 213,
 214, 215, 216, 217, 218, 220,
 222, 224, 225, 227, 255, 260,
 312, 313, 314, 315, 316, 317,
 327, 329, 366, 367, 368, 370,
 372, 373, 394, 414, 439, 448,
 454, 461, 462, 498, 516, 518,
 519, 520, 531, 578, 600, 643
 call 212
 latency 262
 Queues 259
 T1 208
 ISRs 18, 196, 326, 328, 335, 353,
 354, 358, 359, 360, 362, 377,
 388, 395, 411, 494, 515, 640, 642
 ISRs and device drivers 23
 IST 198, 369
 ISTs 260, 353, 354, 377

J
 J2ME 383
 Java 236, 263, 264, 265, 267, 268,
 307, 595
 2 267

accelerator 643
 Card 267, 268, 595
 embedded card 110, 111
 objects 268
 PipeInputOutputStreams 339
 the Java execution
 accelerator 94
 I 267
 G 10, 533, 534
 CODEC 49
 A 94, 266, 267, 268, 595

 nel 307, 312, 313, 330,
 353, 354, 408
 mode 352
 space 370
 stack 311
 TimeSlice 458
 /board 82
 is 17
 pad 17, 285, 286
 -state 146, 608
 M 267

 N 175, 176
 guage 299
 ency 214, 643
 D 14, 17, 48, 50, 143, 195, 199,
 383, 497, 513, 521, 532
 Controller 50, 148, 149
 display 82, 83
 Display or Touch Screen 518
 LED and touchscreen 82
 D 14, 17
 Tests 653
 : than 286
 ary 22, 23, 236
 ary functions 264, 381
 cycle 634
 O 243, 445, 456, 637
 itweight 307
 ked lists 641
 er 20, 21, 622, 627
 e-register 203, 211, 217
 ux 312, 332, 641
 2.6.x 496, 499

 Device Drivers 229
 internals 228
 kernel 497, 498
 Modules 498
 LISR 370
 list 242, 250
 Listen function 492
 Little Endian 102
 Load 101
 loader 21, 625, 626
 locator 21, 26, 27, 109, 621, 622,
 627, 628, 629, 631, 632
 Lock 334, 396, 455
 locking 335
 Logic 65
 Analyzer 622, 624, 655
 Instructions 65
 Probe 653
 Logical Instructions 91
 Look-up 639
 lookup table 242
 loop 252
 back 223
 control instructions 66
 Loosely coupled 291
 Low-voltage operation 384
 LSTTL 145
 LVC MOS 150
 LVTTTL 150

M
 MAC 96, 176, 178, 179
 Mac OS X 370
 MAC unit 33
 MAC/Octal 97
 machine 19
 code 19, 20
 or Real time 119
 Macro 240, 589, 590, 600
 MACROS 239
 mailbox 240, 337, 338, 339, 368,
 409, 437, 438, 445, 439, 440,
 441, 442, 443, 444, 449, 523, 525
 Main 240, 261
 function 237
 Mapping 37
 maps 631
 mask bits 207
 maskable 18, 207

 master 132
 master device 135
 mCOS-II 308, 409
 mCOS-II RTOS 337
 Mealy model 283
 Media processor 33
 Median scale embedded 52, 53
 memory 12, 68, 372, 459, 496,
 631, 639
 Address-Mapped IO
 Operations 75
 allocation 359, 378, 371, 412
 map 109
 blocks 424
 circuits 69
 management 354, 359, 371, 482
 manager 361
 Managing 360
 map 110, 111, 112, 113, 629,
 630
 Organization 78, 101
 protection 360
 stick 13, 50, 120
 buffer 14
 management Unit 86
 mapped 629
 MemScope 654
 Message 357, 609
 box 493, 568, 569, 612
 pointer 599
 queue 487, 500, 572
 method overloading 263
 over-riding 263
 MFLOPs 642
 MFM 135
 microarchitecture 93
 Microchip 636
 Microcontroller 5, 6, 7, 8, 9,
 35, 44, 45, 77, 115, 117, 517, 518,
 535, 573, 574, 577, 582, 583,
 595, 635, 636, 637
 Core 625
 Selection 114
 microkernel 354
 microprocessor 4, 5, 6, 7, 32
 MIDI 568, 569, 570, 571, 572, 573,
 574
 MIDP 268
 Million Instructions Per Second
 (MIPS) 6
 MIMD 290, 636

MIME 171
 MIPS 642
 MIPS R5000 are other 94
 MISO 130
 MISRA C 495, 585, 586
 MISRA C version 585
 MMU 84, 98, 100, 360, 361
 mobile 286
 computer 50
 Phone 50, 119, 195, 285, 383, 440
 phone device 331
 phone keypad 146, 147
 phone LCD 337
 mode key 604, 605
 models 23
 Modem 18, 17
 Modifier 250, 251
 Modular Design 37
 programming approach 236
 module initialization 498
 Monitor 621, 625, 658, 659
 codes 657
 MOSI 130, 132
 Motorola 627, 628, 636
 MPEG 10
 MSG 470
 msgQCreate 469, 471, 547, 554
 msgQSend 470, 471, 550, 551,
 552, 553
 MUCOS 408, 411, 412, 413, 414,
 415, 416, 417, 422, 437, 440,
 446, 449, 456, 459, 495, 498, 522
 Multibyte Store 101
 Multi-dimensional array) 250
 Multilevel Buses 75
 Multiple inheritance 264
 multiplexer is 14
 multiprocessor 33, 292, 642
 MULTIPROCESSOR
 SYSTEMS 288, 293
 multirate operations 4
 multitasking 319, 328, 330, 354,
 408, 479, 485
 program 284
 multithreaded 307, 479, 485
 Music file 51
 Musical Instrument Digital
 Interface 568
 Mutex 318, 323, 329, 331, 334,
 396, 397, 500

 priority inversion 455
 semaphore 463

N
 nanosleep 500
 nested function 244
 Nesting 368
 network 652
 driver 237
 Interface Card 10
 Networked Embedded Systems 159
 NMI 210
 Non-maskable 207
 notification 486, 513, 532, 539,
 570, 577, 606
 NTP 171
 null 248, 440, 463
 pointer 241, 523

O
 object 262, 276, 277, 278, 296,
 297, 487, 517, 540, 572
 diagram 298, 299
 file 624
 store 482
 oriented design 295
 of 80x86 88
 general purpose registers 88
 interrupt sources 18
 RISC 87
 tokens 281
 OMAP of 36
 one master 161
 dimension array 244
 dimensional array 639
 OOP 262, 263, 264, 265, 276
 open 340, 364, 365, 473, 554
 drain port 145
 operating system 18, 340
 System Interface 364
 operator overloading 263
 optimize 385
 Optimizing 41
 the Power 383
 or critical section 316
 PROM 20
 ORCHESTRA 567
 orchestra 569

 Playing Robots 336, 343, 389
 Orchestrator 568, 569, 572, 573,
 574, 608, 609, 610, 613
 ordered list 387, 388
 organization 84
 OS 23, 24, 307, 308, 309, 310, 311,
 312, 313, 316, 318, 320, 321,
 322, 329, 331, 332, 333, 334,
 335, 336, 337, 338, 341, 342,
 345, 351, 352, 353, 354, 355,
 356, 357, 358, 359, 360, 361,
 362, 363, 365, 366, 367, 368,
 382, 389, 390, 395, 396, 397,
 398, 399, 400, 401, 402,
 494, 496, 585, 595, 598, 642
 (kernel) 305
 functions 340
 pipe 339
 scheduler 388
 ENTER_CRITICAL 396, 413,
 415
 EXIT_CRITICAL 413, 415
 NO_ERR 439
 TASK_CREATE 418
 TASK_RESUME 418
 TASK_SUSPEND 418
 TICK_PER 357
 OS's 427
 Oscilloscope 654
 OSFK 495, 496, 583, 585, 586
 OSes 330
 OSEventDelete 359
 Query 359
 OS-Hardware interface 641
 OSInit 414, 418, 526
 OSIntEnter 368, 414
 OSIntEnter():/P 431
 OSIntExit 368, 415
 OSISRSemPost() 367
 OSMboxAccept 437, 444, 530
 OSMboxCreate 441, 526
 OSMboxPend 39, 358, 439, 443,
 444, 456
 OSMBoxPost 337, 338,
 438, 443, 444, 528
 OSMboxQuery 438, 439
 OSMBoxWait 338
 OSMemPut 424, 426
 OSMemQuery 424, 426
 OSMsgPend 573

OSMsgQAccept 336, 571, 611
 OSMsgQPend 571, 611
 OSMsgQPost 373, 571, 573, 611
 OSQ 380
 OSQAccept 335
 OSQCreate 335, 336, 380, 446, 450
 OSQFlush 335, 336
 OSQPend 335, 336, 337, 452, 456, 602
 OSQPost 335, 336, 446, 447, 451, 601, 603, 604
 OSQPostFront 335, 446, 447
 OSQQuery 335, 336, 380, 446, 448
 OSS 374, 379
 OSSchedLock 396, 416
 OSSchedUnlock 4, 413, 416
 OSSemAccept 428, 444, 527
 OSSemCreate 430, 432, 441, 450, 525
 OSSemPend 314, 316, 317, 318, 319, 320, 321, 331, 358, 374, 379, 427, 431, 432, 433, 436, 442, 451, 456, 527, 528, 529, 603
 OSSemPost 316, 317, 318, 320, 374, 428, 429, 431, 432, 433, 434, 436, 442, 528, 529, 602, 603
 OSSemQuery 428, 429, 443, 444
 OSStart 413, 414, 419, 526
 OSStart () 430
 OSTask_Resume 530
 OSTaskCreate 417, 419, 435, 442, 450, 459, 526, 527
 OSTaskDelete () 377
 OSTaskResume 417, 420, 421, 529
 OSTaskSuspend 417, 420, 456, 527, 528, 529, 530
 OSTickinit 413, 419, 421
 OSTimeDly 422, 423, 431, 436, 443, 445, 452, 456, 530, 602
 OSTimeDlyResume 432
 OSTimeDlyHMSM 422, 424
 OSTimeDlyResume 422, 436, 437, 444, 445, 452, 529, 603, 604
 OSTimeSet 417, 421
 OTP 106
 ROM 107
 OutputStream 545, 553
 Overriding 622

P
 P and V 323, 324, 326
 Semaphore 325, 322
 P83C538 66
 package 296, 298
 packet is 31
 PAGE_SIZE 641
 Palm OS 50
 PAN 489
 parallel bus 160, 166, 167, 169
 device 147
 IO bus 160
 port 130, 133, 143, 146, 223, 225
 Port Interfacing 149
 partition 424
 Passing the references 258
 the Values 257
 PC 211, 243, 244, 305, 306, 307, 308, 310, 311, 323, 355, 631, 640, 649, 650
 PCB 306, 312, 355, 356, 365
 or resource 402
 PCI 166, 167, 168, 169
 PCL 217
 PCM 16
 PCS 151
 PDAs 268
 Performance 39, 401, 532
 Accelerators 643
 Index 642
 Metric 400
 periodic 385
 Peripheral Transactions Server 100
 persistence memory 604, 605, 607
 Personalization key 46
 Petri net 397
 PFCU 84
 Phillips 636
 physical 24, 25
 Device 24, 221
 PIC 16F84 8, 32
 PIC18 8, 32
 Piconet 570, 572, 573, 574
 PIM 46, 482
 pipe 24, 223, 242, 247, 340, 341, 472, 473
 pipeDevCreate 473

pipeline 86, 90, 104, 289
 pipelining 105
 Pipes 331
 PISO 135
 Pixel coprocessor 9, 535
 Platform independence 266
 dependency 641
 PLC 635
 Unit 636
 PLD 75
 PMA 151, 228
 PocketPC 50, 149, 479, 482, 489
 pointer 241, 242, 243, 244, 245, 246, 247, 248, 249, 637
 polled bus 164
 Polling 80, 81, 82, 190, 210, 255, 274, 361
 pop 243
 port 518, 521, 533
 by 14
 devices 582
 Deliver 515
 ISR 223
 portability 352
 ports 14, 514, 515, 516, 519
 POSIX 226, 364, 366, 453, 469, 499, 626
 100.3b 326
 1003.1.b 322
 FIFO 454
 queues 472
 IPCs 455
 posting the mailbox IPC 338
 power 384, 385
 Dissipation 39, 41
 manager 480
 supply 10, 47
 PPI 145
 preempt 392
 preemption 368, 394, 395
 preemptive 313, 385, 401, 412, 461
 scheduler 396
 Scheduling 392, 395
 Prefetch control unit 86
 preprocessor 237
 constants 239
 declarations 237
 directive 238
 Directives 239

global variables 239
 Pre-Scalar 155
 Primary 4
 Princeton 629
 Architecture 103, 630
 memory 89
 printer. Assume 79
 priorities 330, 393, 499, 525
 priority 209, 210, 211, 307, 308, 310, 318, 321, 334, 417, 418, 456
 ceiling 408
 inheritance 330
 Inversion 329, 396
 scheduling 461
 based 387
 based scheduling 388
 procedure-based language 262
 Process 305, 307, 313, 323, 341, 345, 355, 479, 486, 496
 1 322
 1 306
 3 326
 c 324
 control block 305
 Creation 355
 deadlines 39, 42, 513
 ID 497
 management 354, 355, 371
 manager creates 356
 or task or thread 240
 state 305
 structure 305, 307
 processes 23, 289, 312, 325, 342, 351, 352, 353, 485, 497
 processing 18
 processor 5, 6, 72
 organization 86
 PROCESSOR SELECTION 113
 less 635
 sensitive 638
 sensitive memory-sensitive 640
 Process-state signal 306
 producer-consumer 325
 program 23, 65, 74
 counter (PC) 11, 62, 84, 85, 313, 324, 387, 388, 391, 395
 Flow Control Instructions 66
 layers 22
 memory 68, 482
 Model 23
 counter 84
 programmed-I/O busy-wait 189
 Programming model 255, 258
 PROM 12, 13, 21, 106, 107, 108, 632
 property 276
 Protection 402
 protocol 15, 342, 344
 stack processor 9
 Prototype 39, 650, 653
 Prototyper 26, 27
 PSEN 68
 PSW 63
 pthread_mutex 500
 wait 503
 pump 47
 push 243
 pushing 244
 PWM 15, 51

Q
 QAM 135
 QError 336
 QoS 174
 Qrio 567
 Quarter-CIF 9, 34
 quasi bi-directional 145
 Query 338
 queue 240, 242, 244, 246, 247, 249, 335, 336, 378, 380, 409, 445, 447, 448, 450, 451, 452, 643
 queue-related 449
 mailboxes 331
 Queuing 260

R
 RAM 12, 45, 46, 109, 110
 disk 223, 482
 memory 13
 parity error 210
 Rate Monotonic 398
 Monotonic Scheduler 399
 RD 68, 73
 RDRAM 109
 read 340, 364, 365, 473, 554
 () 343
 Ports 517, 520
 Range 583
 Real time clock 158, 159
 Time Linux 496
 time Robotic 115
 time 497
 time clock 11, 248, 356
 time FIFO 504
 Time Program 642
 time programming 385
 time task 502
 Time Video Processing 117
 Recursive function 240
 Recv 492
 reentrant 313, 328
 function 240
 Registering 498
 Registering, De-Registering 503
 Registry 483, 484
 Reliability 42, 585
 Remote procedure 331
 requery hopping 178
 Requirements 513, 531, 532, 539, 570, 593, 604, 605, 606
 of smart card 594
 Reset 11
 reset value 85
 Resource 318, 323
 Key 316, 317
 type 355
 resources 334
 Rest Vectors 624
 restricted runtime 268
 RETI 65
 return to 190
 return 191, 193
 RI 205
 Right Platform 635
 RISC 33, 88, 98
 formats 89
 RKE 575
 Robert Metcalfe 175
 Robin Time 389
 time-slice 461
 robot 51, 52, 117, 631
 Functions 51
 orchestra 568, 570, 572
 corchestra 569
 Robotic 116
 ROM 12, 19, 21, 45, 46, 106, 107, 108, 109, 110, 116, 117, 482
 ROM Emulator 649

- image 19, 20, 110, 626
 - image are 22
 - image file 118
 - ROS 314
 - Round Robin 388, 390
 - (time slicing) scheduling 385
 - time 390
 - routers 174
 - RS232C 136, 137, 634
 - RS232C Port 632
 - RS485 138
 - RSA 595
 - RTC 257, 263, 588, 589, 590
 - RTLinux 496, 501, 502, 503, 504
 - RTO 478
 - RTOS 24, 27, 333, 359, 363, 369, 370, 371, 372, 373, 375, 376, 377, 378, 381, 385, 392, 393, 394, 397, 409, 410, 411, 412, 413, 415, 453, 454, 478, 515, 522, 577, 621, 623, 642
 - RTOS kernel 415
 - mCOS-II 360, 368
 - provides 23
 - scheduler 375
 - timer functions 357
 - VxWorks 537
 - Windows CE 352
 - RTOS's 408
 - RTOSes 332
 - RxD 633
 - RxRDY 79, 80
- S**
- Scalable 378, 454
 - hierarchical RTOS 382
 - scheduler 313, 385, 386, 387, 389, 390, 391, 392, 394, 395, 397, 398, 401, 408, 486
 - Scheduling 385, 386, 393
 - functions 23
 - Schmitt trigger 150
 - SCI 142, 576
 - SCLK 130
 - ScopeProfile 654
 - Scratchpad Memory 607
 - Screen_State 254
 - SDF graph 291
 - SDFGs 291, 293
 - SDIO 142, 143, 165
 - card 133
 - SDK 480
 - Secondary memory 4
 - level 207
 - second-level interrupt service
 - thread 199
 - Section 314
 - Section 3.12.1 170
 - self-host 410
 - semaphore 240, 284, 314, 315, 316, 317, 318, 319, 320, 323, 325, 328, 329, 330, 331, 332, 375, 379, 409, 412, 426, 428, 448, 455, 463, 494, 499, 599
 - functions 487
 - SEMAPHORES 322, 378, 469, 525
 - sem_c1 326
 - semBCreate 463, 464, 467, 546
 - CCreate 468
 - Flush 463, 465, 467
 - Give 463, 464, 465, 466, 467, 469, 549, 551, 552, 553, 591, 592, 593
 - MCreate 465, 466, 467
 - Pend 602
 - Take 463, 464, 465, 466, 468, 470, 549
 - Take 550, 551, 552, 553, 554, 591, 592
 - Send function 492
 - Sequence diagram 298
 - sequential model 280
 - program model 274
 - programming model 275
 - SerDes 151
 - Serial asynchronous output 134
 - clock 161
 - Data Communication 139
 - line device 78
 - port 130
 - Port Drivers 226
 - ports 69, 70
 - SERVICES 351, 353
 - servo motors 67
 - SHARC 95, 105
 - Shared 327, 328
 - bus 290, 291
 - Data 329, 652
 - memory 455, 496, 500, 501, 502
 - Shooting 532
 - show 175
 - SI 69, 70, 71, 72, 215
 - SigACC 587
 - sigHandler 462
 - signal 199, 201, 204, 284, 297, 306, 331, 332, 333, 462, 499, 513, 539, 570, 606
 - handler 203
 - Signals, events 532, 577
 - Events and Notifications 594
 - SIMD 33, 89, 94, 636
 - SIMDs 290
 - Simulated annealing method 400
 - simulates 623
 - Simulator 26, 27, 621, 650, 653, 651, 652, 620, 625, 649
 - Single Purpose Processors 6, 8, 635
 - purpose processors and application specific 35
 - stepping 210
 - SIPO 135
 - Six Tasks 523
 - Skills 54
 - for Small Scale 53
 - slave device 135
 - slice 392
 - SLISR 198
 - SLISRs 370
 - slow-level ISR 369
 - Small scale embedded 52
 - smart 45, 602
 - card 44, 47, 110, 111, 267, 268, 309, 341, 593, 595, 631
 - card processor 46
 - mobile phone 254
 - OS 603
 - OSTickInit 601
 - SMS 50, 147, 604, 606, 607, 608, 610, 611, 613
 - SMS Create 607, 609
 - SoC 29, 30
 - socket 24, 342, 345, 489, 491, 492
 - descriptor 306, 343
 - sockets 51, 331, 343, 344
 - soft real time 375, 381
 - software 25, 211
 - Architecture 519, 541, 572
 - components 52

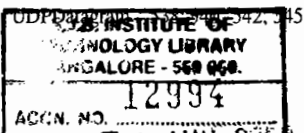
- development process 41
 - exceptions 202
 - instruction-related sources 201
 - interrupt 18, 192
 - interrupt (SWI) 196
 - interrupt instruction (SWI) 195
 - timer 155, 156, 157
 - timer 606
 - Tools 25, 26, 620, 622, 627
 - traps 202
 - Sophisticated embedded systems: 52
 - Source 27
 - Code Engineering 622
 - files 237, 239
 - SP 84, 244, 305, 306, 308, 310, 311, 312, 355
 - Specifications 40, 514, 515
 - SPI 141, 142
 - spin 381
 - Lock 334, 335, 371, 396
 - sporadic 385, 399, 401
 - Sporadic task 398
 - SPSR 212, 213
 - SRAM 108
 - S-record 628
 - SRS 84, 86
 - SSTL 150
 - stack 191, 244, 249, 637
 - pointer 62, 211, 243
 - pointer 85, 217
 - stacks 321, 643
 - standard file 482
 - Start Timer 526
 - state 276, 286, 287, 296, 298, 305, 309, 355
 - diagram 298, 299, 517, 540, 572, 573, 610
 - machine 282
 - machine model 275
 - of a key 613
 - table 285, 286, 288
 - transition 285
 - transition function 282, 284
 - transition-functions (Moore model) 283
 - states 282, 284, 285, 288, 308, 309, 356
 - machine (FSM) 284
 - of a timer 283
 - state-transition 607
 - TransitionFunction 285
 - static 251
 - scheduling 281, 290
 - scheduling issue 290
 - Status flag 155
 - register 78, 640
 - stepper motor 67, 147, 148, 578, 582
 - stereotype 296, 297
 - Stethoscope 26, 27, 654
 - STOP 117
 - storage 655
 - Store 101, 102
 - string 639
 - StrongARM SA-110 94, 114
 - structural 383
 - structure 352, 353
 - SuperH 410
 - Superscalar 105
 - processor 289
 - Units 104
 - Supervisory mode 352, 353
 - SWI 197, 202, 204, 254, 287, 288
 - switch 254
 - case 237
 - switching 391
 - SWT 155, 227, 228, 257, 263
 - SWTs 155
 - Symbian OS 50
 - Synchronization 537
 - Diagram 521
 - model 522, 536, 542, 574, 583, 585, 596, 575, 597, 613
 - synchronous 135
 - Communication 134
 - HDLC protocol 139
 - input 133
 - IO operations 365
 - serial 130
 - Serial Input-Output 132
 - Serial Output 131
 - serial port registers 142
 - SyncML 176
 - SysC 357
 - SysClkIntr 356
 - system 3, 343
 - Bus 72, 74, 76
 - buses 80, 81, 85
 - clock 155, 159, 414, 460
 - design 41
 - interrupts 159
 - Timers 11
 - call 353
- T**
- T 606
 - T0 68, 69, 70, 71
 - T1 68, 70, 71
 - T1 in 8051 68
 - T2 68
 - T9 keypad 611
 - T9-keys 604
 - table 242, 247, 248
 - Target 620, 623, 624, 625, 626, 649, 652, 656, 657, 659
 - targeted 652
 - task 282, 283, 289, 294, 308, 311, 312, 313, 314, 315, 316, 318, 323, 325, 385, 396, 494, 516, 517, 520, 521, 524, 525, 526, 529, 530, 571, 572, 574, 580, 581, 582, 587, 588, 590, 591, 592, 593, 595, 596, 598, 599, 601, 602, 603, 604
 - 1 299
 - 2 322
 - A 317
 - control 311
 - create 376
 - Delay 412, 549, 550, 551, 552, 553
 - delete 376, 459
 - Function 584
 - so 290
 - stacks 308, 317, 524
 - User 519
 - 3 395
 - Conductor 570
 - iconet 571
 - J 397, 398
 - SMS_Create 608
 - 3 320
 - Delete () 467
 - Lock () 458
 - Priority 417
 - PriorityGet 458
 - PriorityGet () 458
 - PriorityPut 458
 - PrioritySet () 458

Resume 457, 550, 551, 552, 553
 tasks 23, 295, 319, 320, 325, 326,
 372, 386, 515, 516, 543, 586,
 597, 613
 taskSafe 459
 scheduler 307
 Spawn 456, 468, 588
 Suspend 459
 Unlock () 458
 USB 308, 310, 312, 313, 315, 355,
 356, 454, 459
 UDP 173, 174, 342, 537, 539, 540,
 543, 555
 header 541, 556
 or UDP 538
 UDP/IP 44, 171, 173, 538, 539, 541
 UDP/IP network 170
 UDP/IP stack 539
 Checksum 16 557
 Flag 546
 Hd 556
 Header 555, 557
 TELNET 171
 Template 263
 Test 513, 539, 607, 651
 and validation 532, 578, 570, 595
 timing 622
 Tools 650
 Connectors 623
 Test Messages 605
 text-to-speech converters 576
 the internal RAM, SFR 68
 latency 361
 pixel processor 49
 process 308
 resource 316
 semaphore as 449
 thread 307, 356, 486, 499, 505
 period 501
 stack 308
 priorities 479
 scheduler 308
 stack 307
 threads 306, 485, 496, 502
 rattle 578, 579, 581, 584, 585
 throughput 643
 timing 195, 197
 thumb 93
 thumb@ 91, 93

TI 205
 TigerSHARC 96, 114
 tightly coupled processors 290, 291
 Time 388
 division multiplexing 154
 functions 500
 slices 499
 slicing 154, 371
 Triggered Protocol 582
 Timer 153, 154, 156, 157, 158,
 215, 221, 227, 494
 Functions 356
 interrupt 208
 time-slice 461
 to-live 174
 to-market 39
 TIPC's 455
 TMS320C62XX 33
 to queue 356
 tokens 284
 tool 650
 Top-down design 236
 top-to-down 38
 touch screen 14, 17, 50, 209, 268,
 369, 480, 488, 532, 576
 Trace scope 26, 27
 TraceScope 654
 track 571
 transceiver 597
 transition 608
 trap 192, 202
 traps 204
 Tree 250
 triggering 623
 Tworst 390
 TxD 633, 634
 TxDE 79, 80
 type checking 237
 typedef 241

U

UART 14, 130, 132, 133, 136, 137,
 138, 140, 141, 189, 201, 226,
 236, 257, 366, 637
 16550 226
 8250 226
 UDP 342, 539, 541, 543, 546, 555



UML 23, 43, 295, 296, 297, 298,
 515, 517, 540, 653
 unblocking 420
 PIN 46
 Unicode 639
 Unified 299
 Unix 307, 312, 332, 364, 365
 device driver 362
 operating 459
 unlock 334, 455
 unlocks 335
 USB 49, 50, 536
 bus 194
 bus cable 164
 is a serial 165
 or FireWire 569
 port 532, 533, 448
 User mode 352
 UWB 165

V

Validation 42, 513, 539
 Vector 206
 Address 204, 71
 addresses 19
 table 207
 Verification 41
 video graphic adapter 9
 virtual 223, 25
 base classes 264
 device 223, 24
 machine 266
 memory 480
 VLIW 33, 97
 in TMS320C6 290
 VLIWs 295
 VLSI design 31
 Voice Data Compression 117
 recorder 120
 void main 252
 OSInit 413
 OSIntEnter 413
 OSIntExit 413
 resetTask 601
 VoIP 392
 volatile 251
 Volt-Ohm Meter 653
 VPN 34

VxSim 650, 654
 VxWorks 308, 332, 453, 454, 455,
 456, 458, 459, 460, 467, 469,
 472, 495, 499, 538, 586,
 623, 626, 654
 VxWorks scheduler 494
 VxWorks 409
 vxWorks.h 464, 466

W

Waiting 384
 state 383
 watchdog timer 12, 154, 157, 329,
 461
 timer-related 460
 timers 576
 WCE 483, 484, 486, 488, 489, 491,
 492, 493, 494
 WCE serial port 489
 When bus master 81
 while loop 252

o-while, break 237
 WiFi 151, 576
 Win32 484, 492, 493
 API 492
 graphics 493
 Wind 499
 Window NT 365
 Windows 409, 651
 CE 50, 478, 480, 481, 482, 485,
 487
 CE 6.0 485
 CE Features 479
 CE Serial Communication 490
 CE.NET 478
 Controls 488
 Management 482
 Menus 489
 Mobile 50
 WindRiver 453
 Winsock 489
 Wireless devices 151
 LAN 175, 178
 USB 165
 WLAN 179
 WLAN 802.11 569
 WNet API 489
 WNetDis 491
 Worst case latency 214
 execution time 400
 latencies 387
 latency 329, 386, 394
 WPAN 160, 177
 WR 73
 write 340, 364, 365, 473
 write () 343

X

XCITE 150
 Xilinx 150
 X-Windows 497

Z

ZigBee 151, 175, 177, 179