

The McGraw-Hill Companies

Embedded Systems

Architecture, Programming and Design

Second Edition

Raj Kamal



Tata McGraw-Hill

Published by the Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008.

Copyright © 2008 by Tata McGraw-Hill Publishing Company Limited.

First reprint 2008

RQXLCRYXRXBHQ

No part of this publication may be reproduced or distributed in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise or stored in a database or retrieval system without the prior written permission of the publishers. The program listing (if any) may be entered, stored and executed in a computer system, but they may not be reproduced for publication.

This edition can be exported from India only by the publishers,
Tata McGraw-Hill Publishing Company Limited.

ISBN 13: 978-0-07-066764-8

ISBN 10: 0-07-066764-0

Managing Director: *Ajay Shukla*

General Manager: Publishing—SEM & Tech Ed: *Vibha Mahajan*

Sponsoring Editor: *Shalini Jha*

Jr. Sponsoring Editor: *Nilanjan Chakravarty*

Executive—Editorial Services: *Sohini Mukherjee*

Production Executive: *Suneeta S Bohra*

General Manager: Marketing—Higher Education & School: *Michael J Cruz*

Product Manager: SEM & Tech Ed: *Biju Ganesan*

Controller—Production: *Rajender P Ghansela*

Asst. General Manager—Production: *B L Dogra*

Information contained in this work has been obtained by Tata McGraw-Hill, from sources believed to be reliable. However, neither Tata McGraw-Hill nor its authors guarantee the accuracy or completeness of any information published herein, and neither Tata McGraw-Hill nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that Tata McGraw-Hill and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

Published by the Tata McGraw-Hill Publishing Company Limited, 7 West Patel Nagar, New Delhi 110 008, typeset at
Bukprint India, B-180A, Guru Nanak Pura, Laxmi Nagar-110 092 and printed at SDR Printers, A-28, West Jyoti Nagar,
Loni Road, Shahdara, Delhi 110 094

Cover: SDR Printers

Contents

<i>Preface to the Second Edition</i>	vii
<i>Preface to the First Edition</i>	ix
1. Introduction to Embedded Systems	1
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	61
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	128
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	187
4.1 Programmed-I/O Busy-wait Approach without Interrupt Service Mechanism	189

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 Rt os 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: Microc/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-1	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 Sofware 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
Appendix 1: Roadmap for Various Course Studies	662
Appendix 2: Select Bibliography	663
Index	668

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 Wilmhurst 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

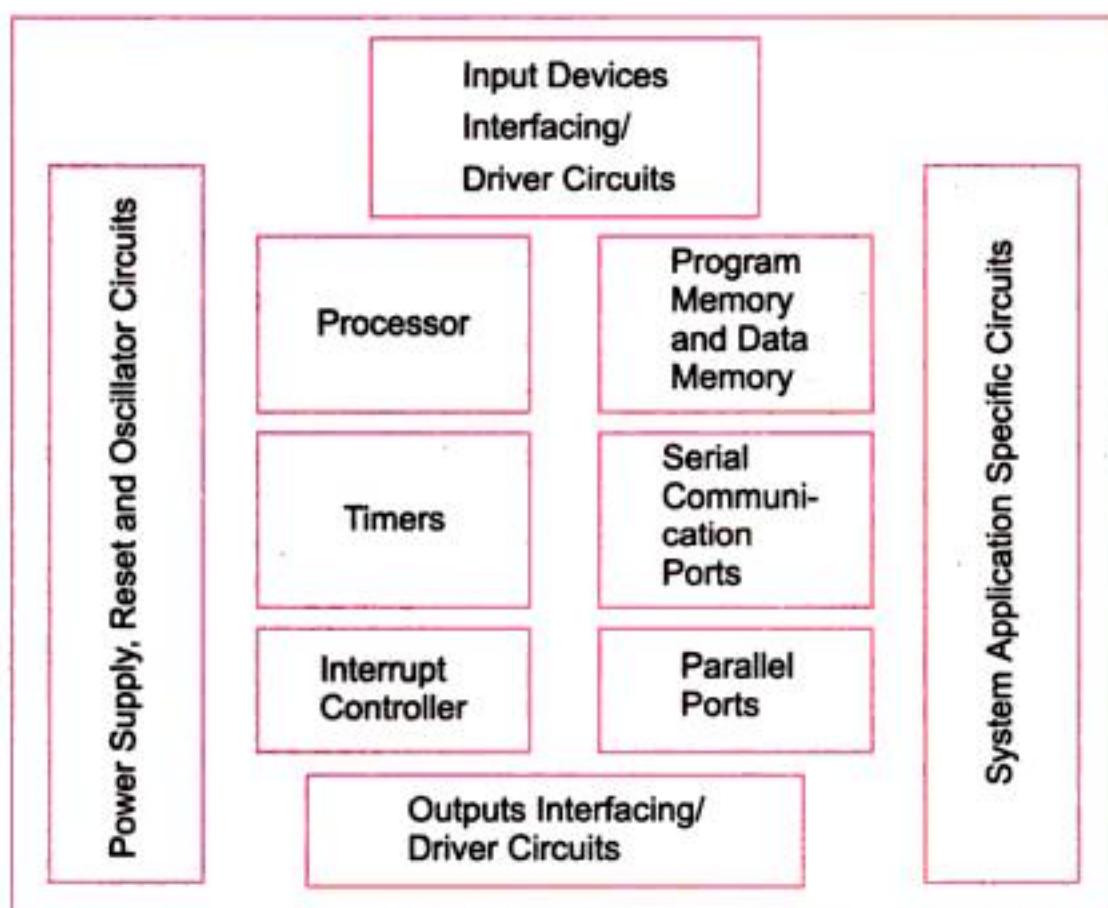


Fig. 1.1 The components of embedded system hardware

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 (FPUs). [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.

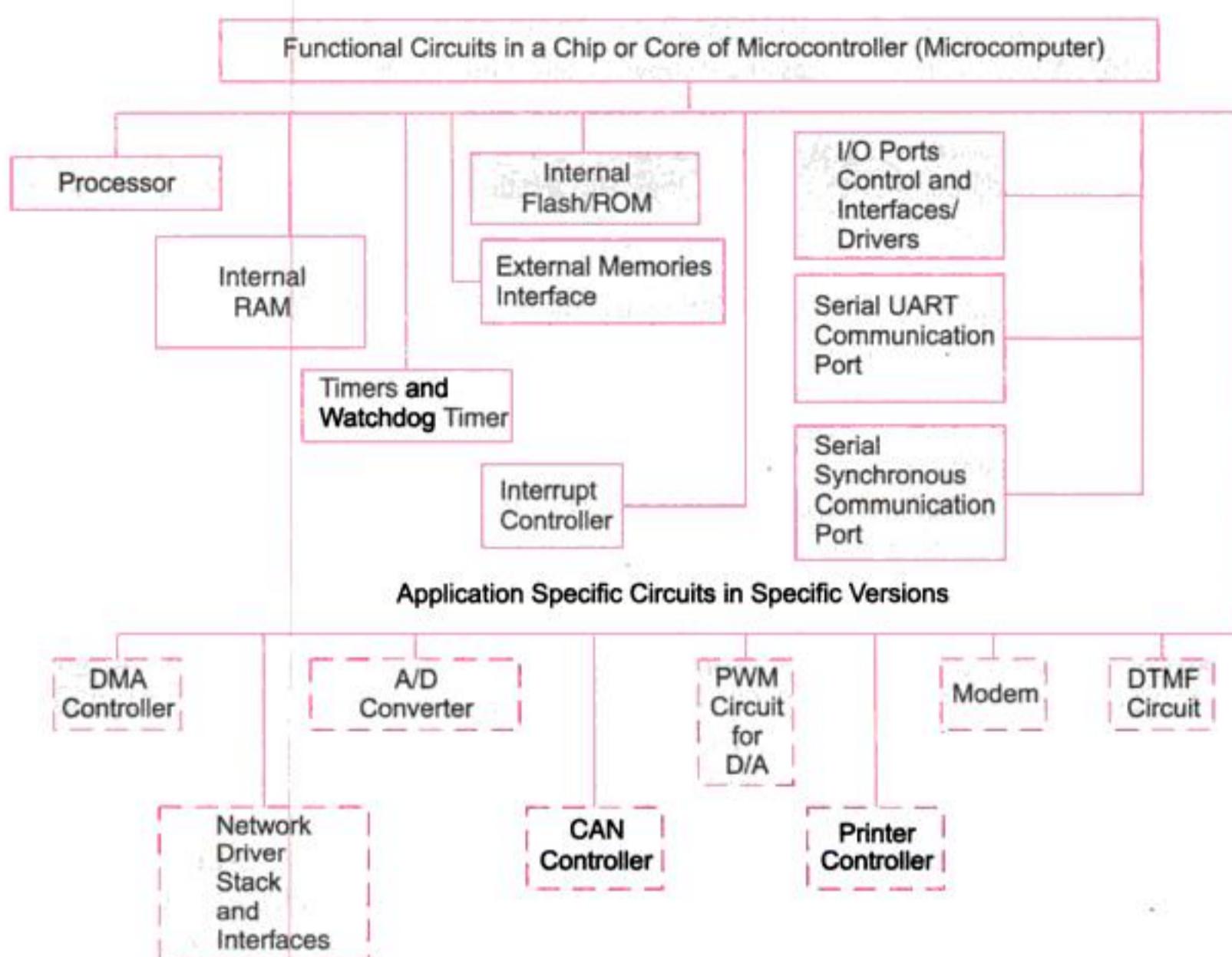


Fig. 1.2 Various functional circuits (solid boundary boxes) in a microcontroller chip or core in an embedded system. Also shown are the application-specific units (dashed boundary boxes) in a specific version of a microcontroller

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).

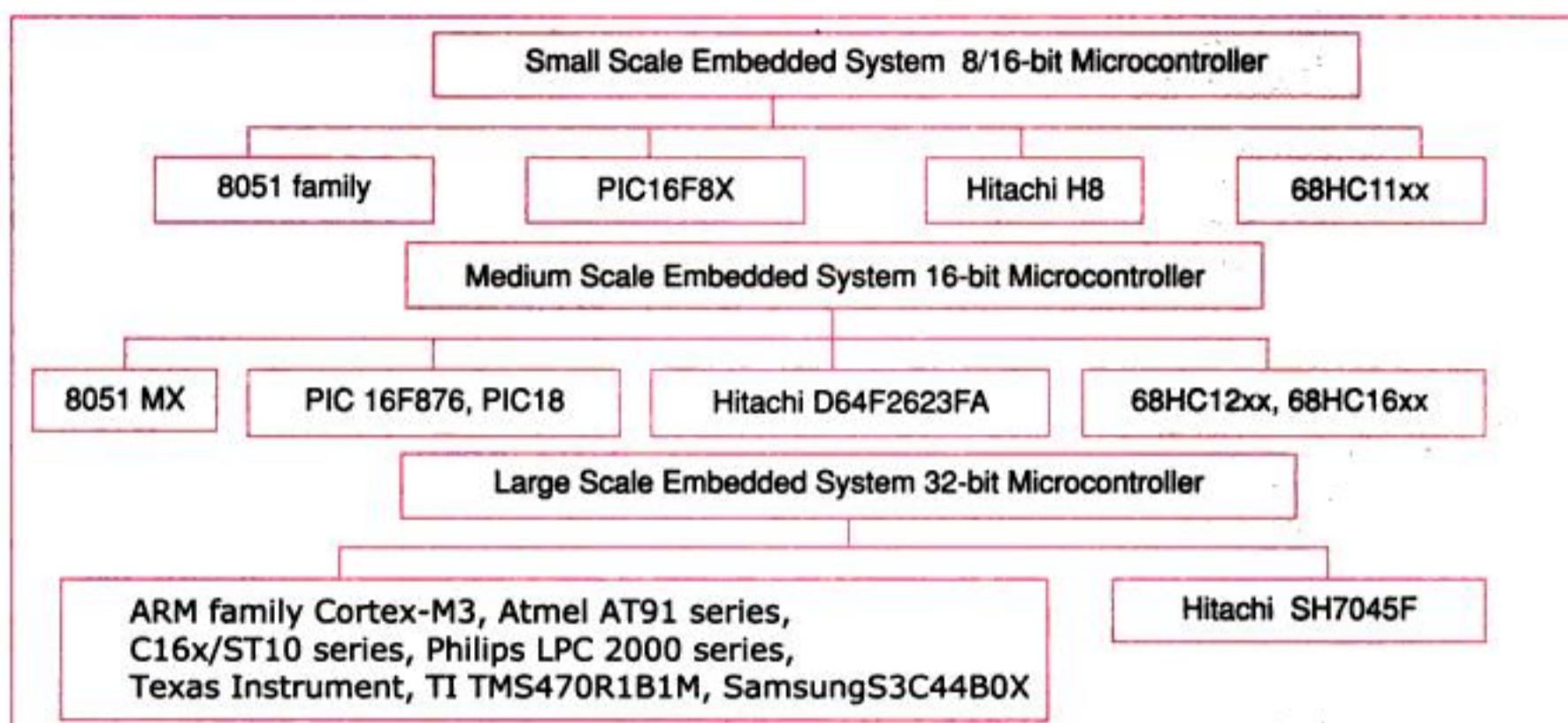


Fig. 1.3 Commonly used microcontrollers in small-, medium- and large-scale embedded systems

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 compilations 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, $\sim (2\text{ V}/5\text{ V})^2$. 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).

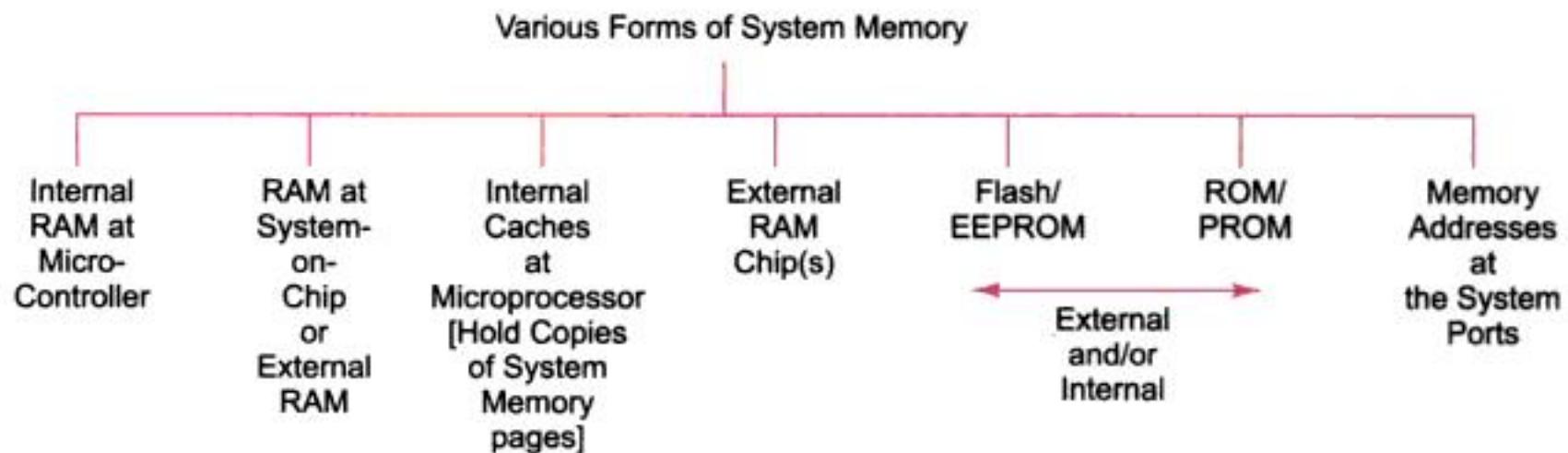


Fig. 1.4 The various forms of memories in the system

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 $15 \times 12.5 \times 1.2$ 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

<i>Memory Needed</i>	<i>Functions</i>
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. Let the lower limit reference voltage be 0.255 V. The difference in the limits is 2 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 $N = v \cdot (V_{ref+} - V_{ref-})/2^n$. Here, V_{ref+} is the reference voltage that gives all the bits that are equal to 1 and V_{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 digitize 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.

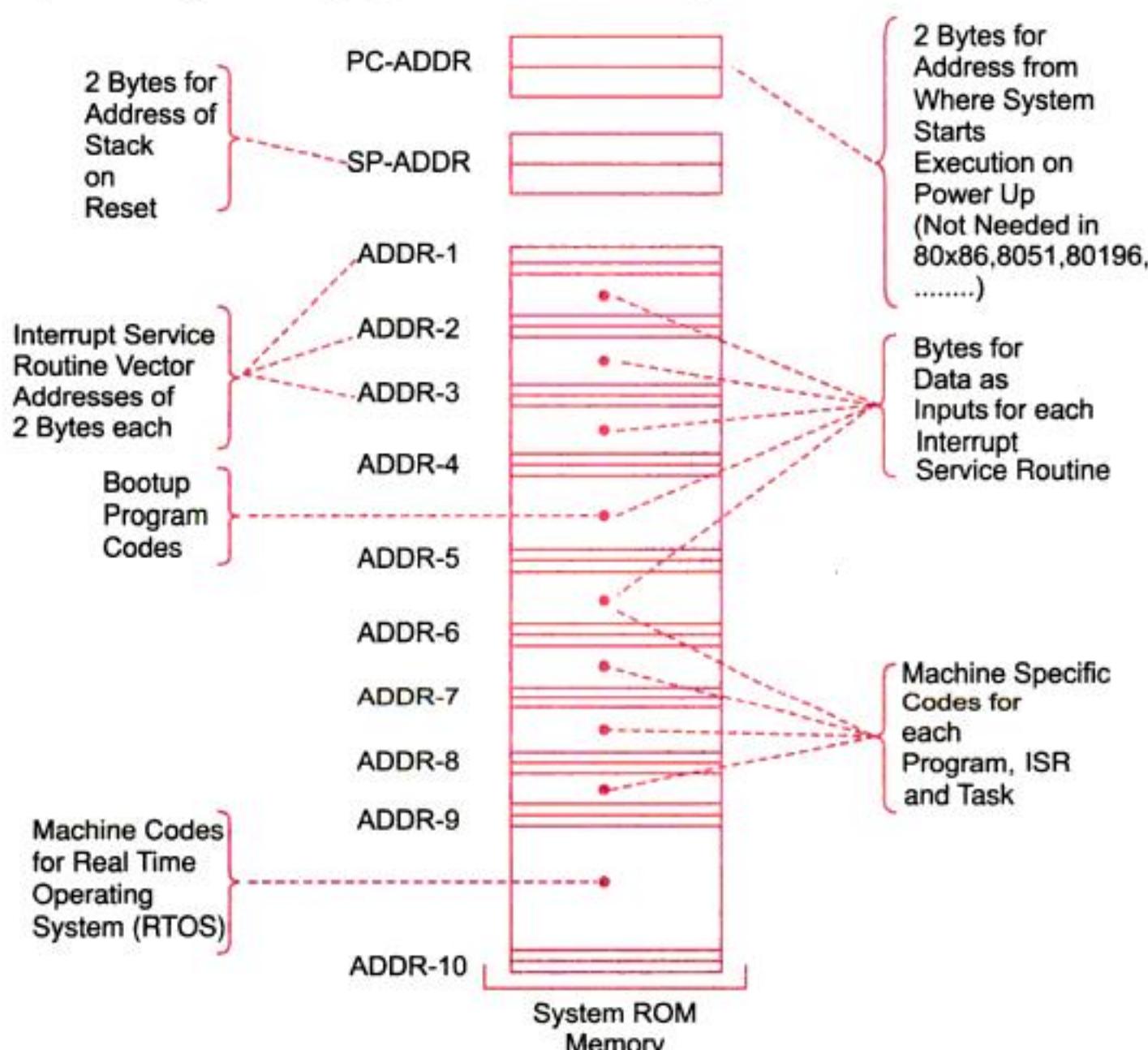


Fig. 1.5 System ROM memory embedding the software, RTOS, data and vector addresses

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.

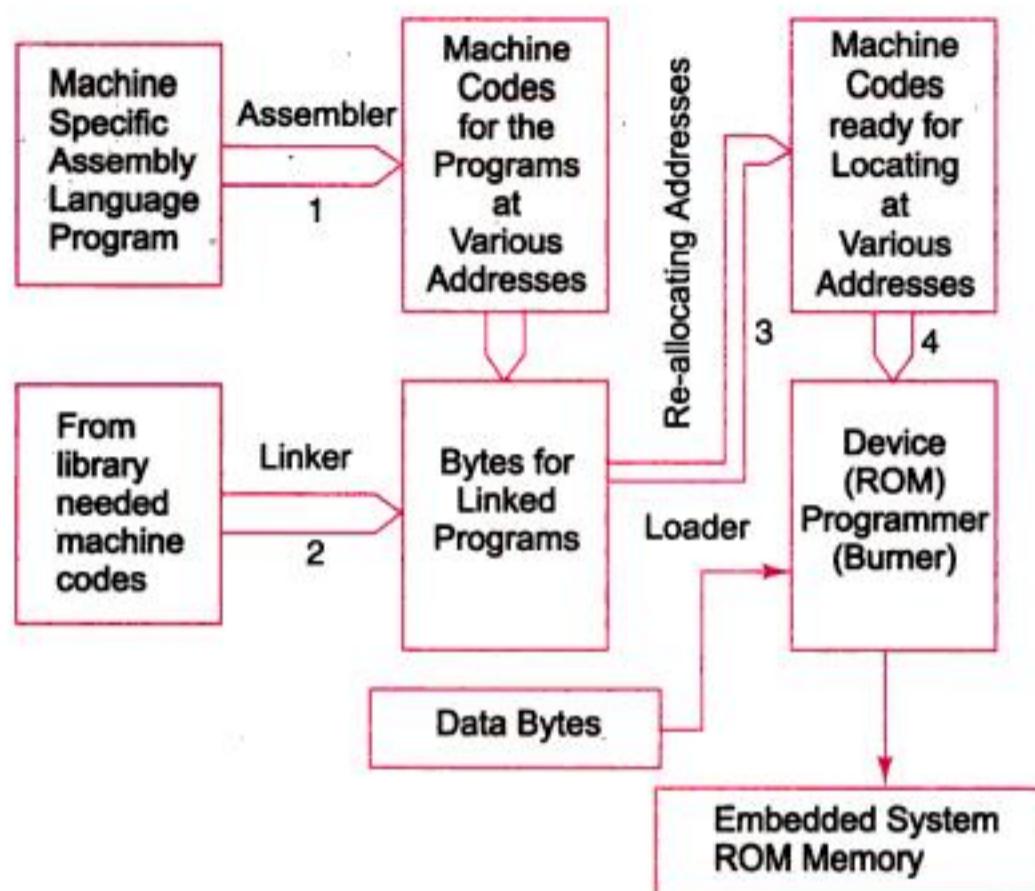


Fig. 1.6 The process of converting an assembly language program into the machine codes and finally obtaining the ROM image

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.

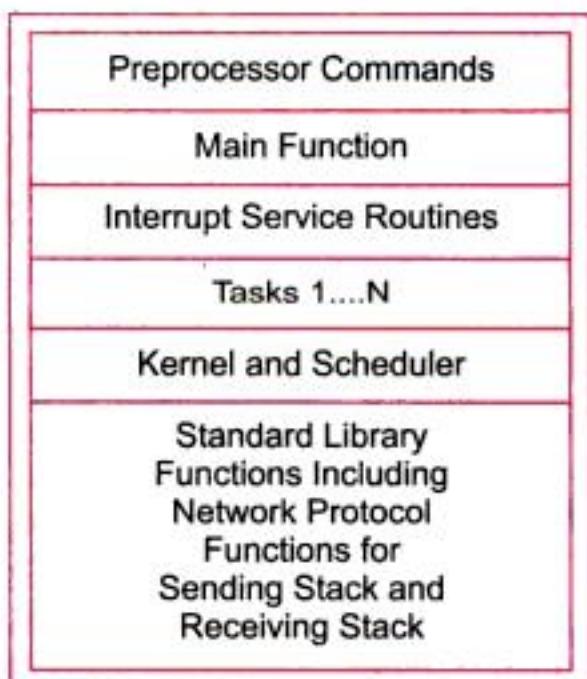


Fig. 1.7 The different program layers in the embedded software in C

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.

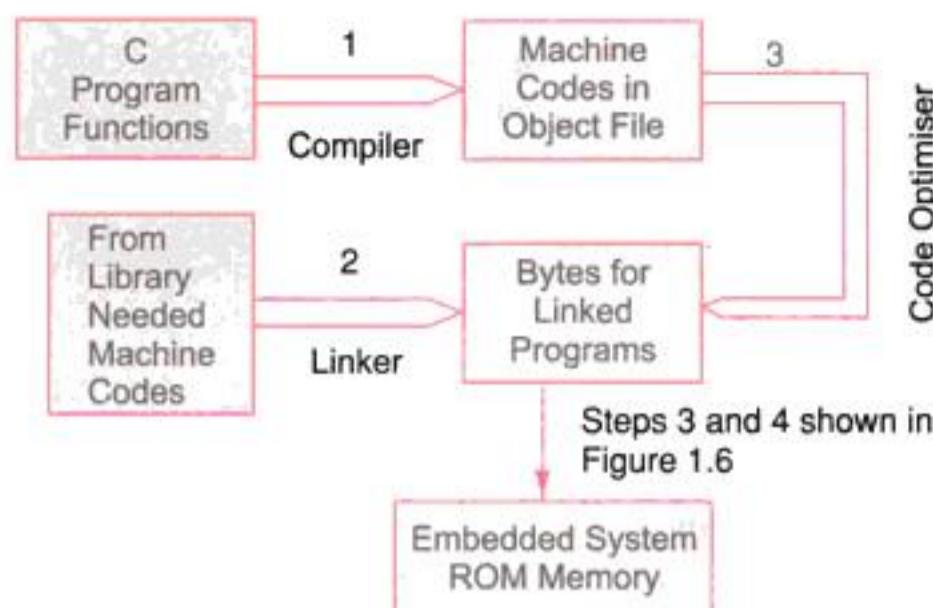


Fig. 1.8 The process of converting a C program into the file for ROM image

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

<i>Software Tools</i>	<i>Application</i>
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)

<i>Software Tools</i>	<i>Application</i>
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

<i>Software Tools</i>	<i>Automatic Chocolate Vending Machine^{&}</i>	<i>Data Acquisition System</i>	<i>Robot</i>	<i>Mobile Phone</i>	<i>Adaptive Cruise Control System with String Stability[#]</i>	<i>Voice Processor</i>
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

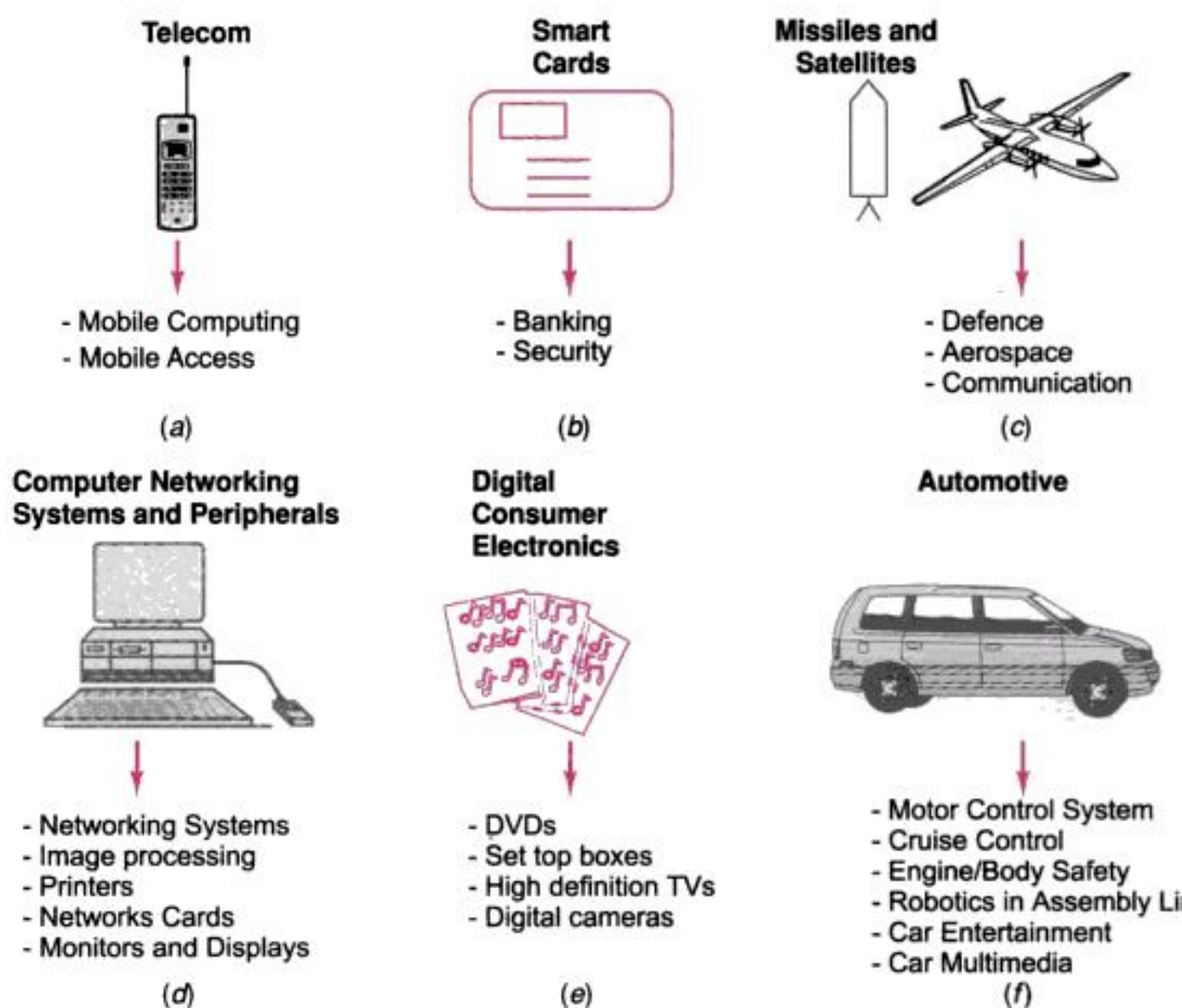


Fig. 1.9 Applications of the embedded systems in various areas

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.

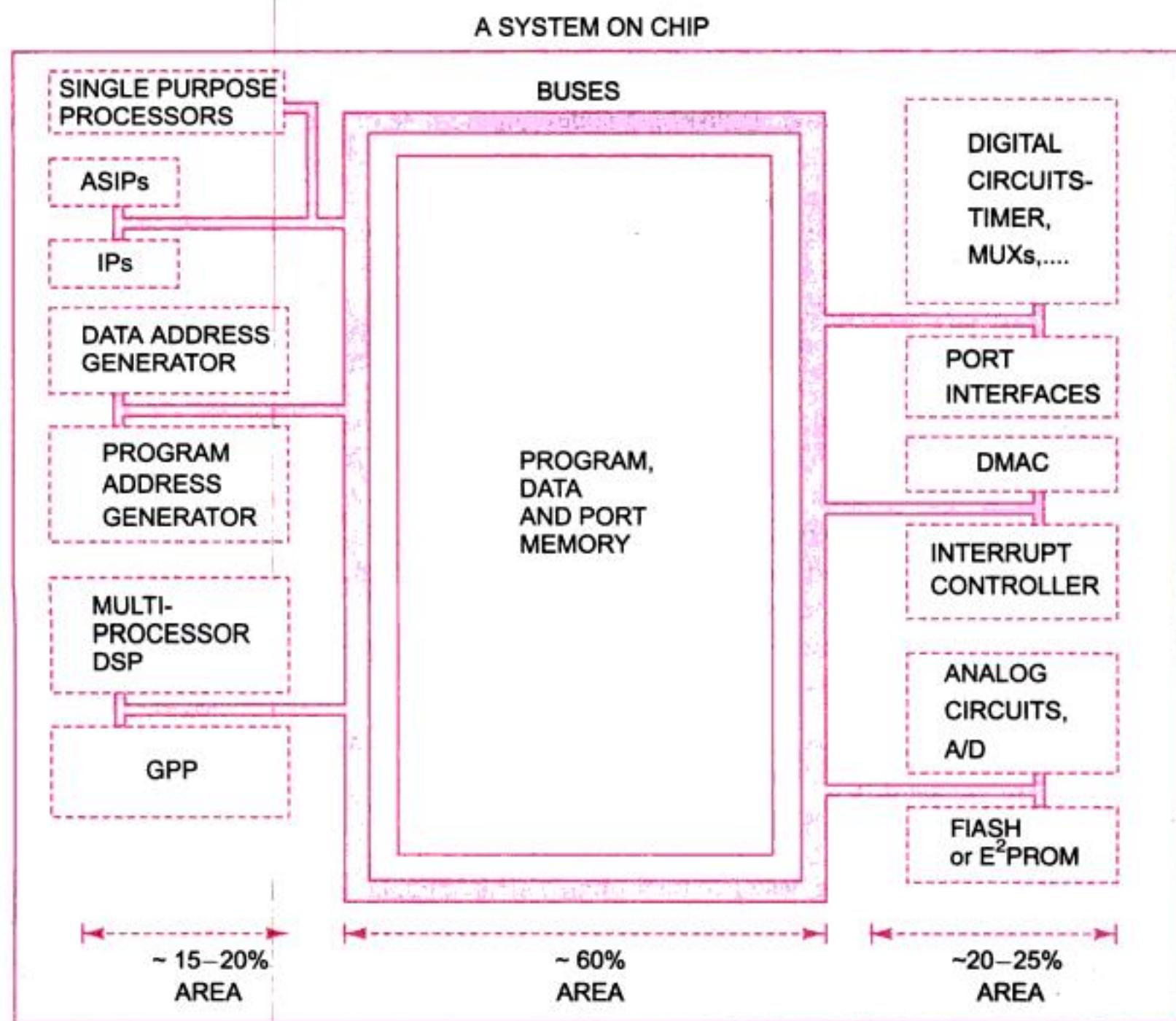


Fig. 1.10 A SoC embedded system and its common bus with internal ASIPs, internal processors, IPs, shared memories and peripheral interfaces

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 VSLI 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 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
General Purpose Microprocessor	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
Microcontroller	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.
DSP	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.
Single purpose processors and application specific system processor	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.
Dual core processor	To significantly enhance the performance of the system.	Reduced engineering cost.	Manufacturing cost, as dual core processors are costly.
Accelerator	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 $0.15 \mu\text{m}$ 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 $0.18 \mu\text{m}$ 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.

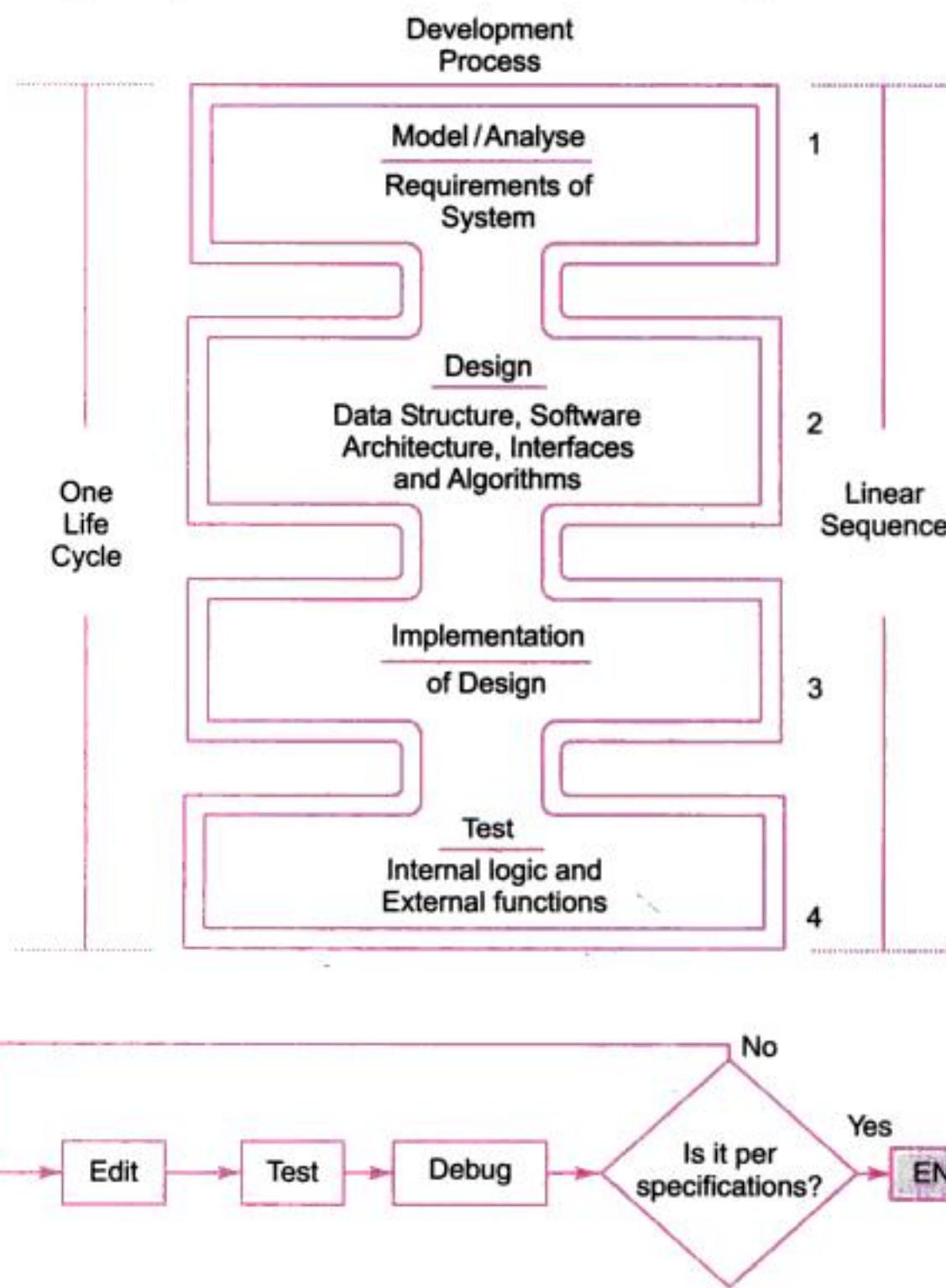


Fig. 1.11 Activities for software design during an embedded software-development process

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

<i>Design Metrics</i>	<i>Description</i>
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	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.
development time	Time taken in days or months after prototype development to put a product for users and consumers.
Time-to-market	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).
System and user safety	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.
Maintenance	

- (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 fulfil 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

<i>Effort</i>	<i>Activities</i>	<i>Model Deficiency</i>
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	
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	Need for robust interfaces and slow development in case the reusable components are not available in required numbers

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

<i>Design Metrics</i>	<i>Description</i>
<i>Analysis</i>	Design is analyzed
<i>Steps for improvement</i>	The result of analysis is used to improve design to meet specifications and metrics
<i>Verification</i>	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 $2.5 \mu\text{W}$ per 100 kHz of reduced clock rate. So reduction from 8000 kHz to 100 kHz reduces power dissipation by about $200 \mu\text{W}$, 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 $25 \mu\text{W}$ 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, $\sim(2V/5V)^2$, 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.

3. When first coin is inserted, a timer also starts. The child is expected to insert all required coins in 2 minutes.
4. 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.
5. 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.

1. Microcontroller or ASIP (Application Specific Instruction Set Processor)
2. RAM for storing temporary variables and stack
3. ROM for application codes and RTOS codes for scheduling the tasks
4. Flash memory for storing user preferences, contact data, user address, user date of birth, user identification code, answers of FAQs
5. Timer and interrupt controller
6. A TCP/IP port (Internet broadband connection) to the ACVM for remote control and for the owner to get ACVM status reports
7. 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
8. Power supply

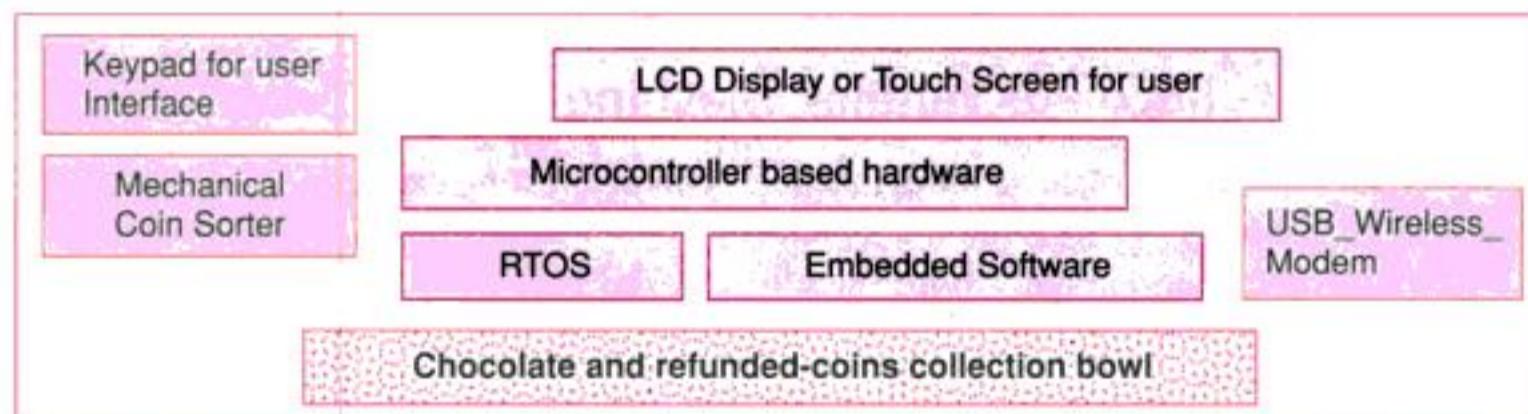


Fig. 1.12 Diagrammatic representation of the ACVM

Software components ACVM embeds the following software components:

1. Keypad input read task
2. Display task
3. Read coins task for finding coins sorted
4. Deliver chocolate task
5. TCP/IP stack processing task
6. 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.

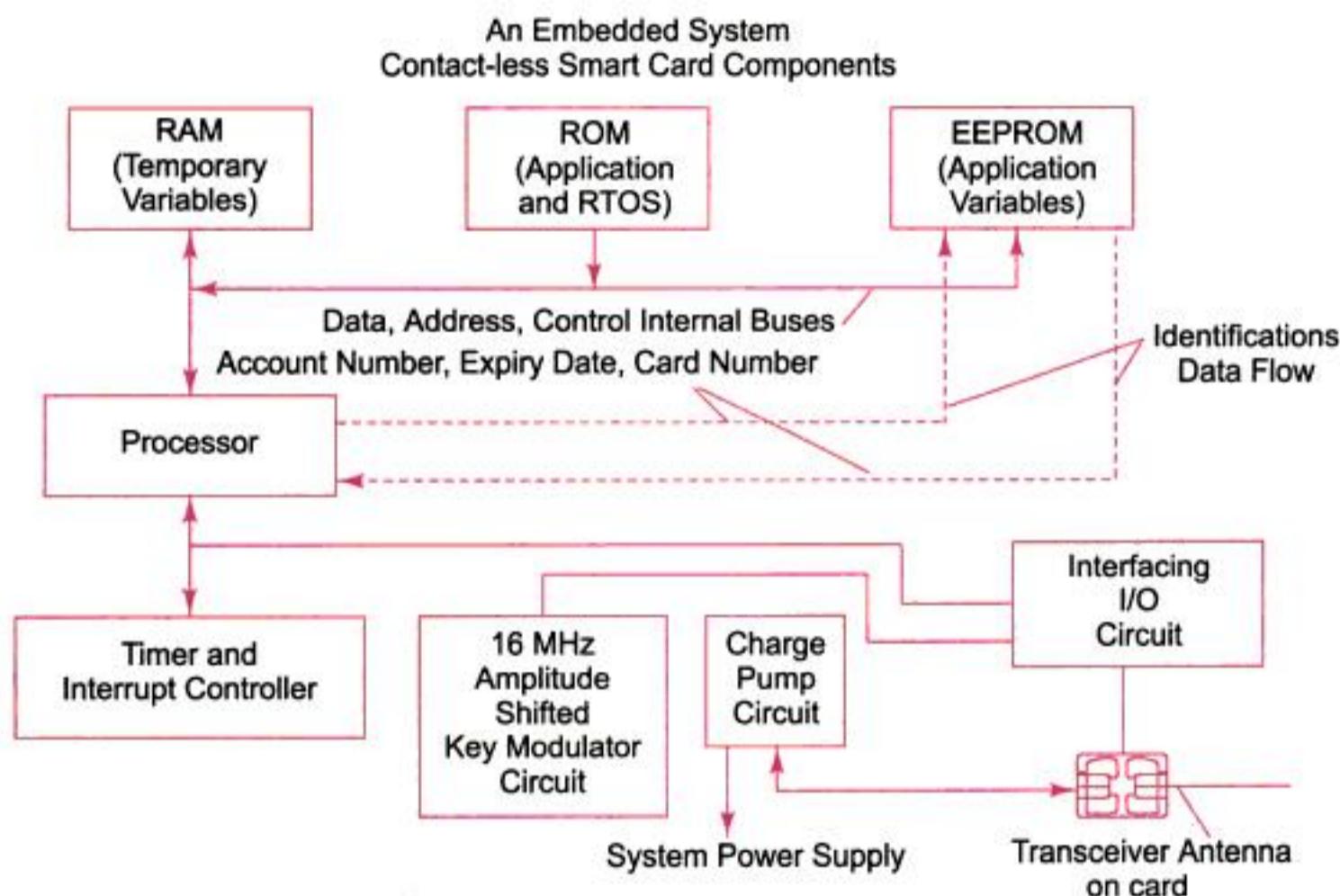


Fig. 1.13 Embedded hardware components in a contact less smart card

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:

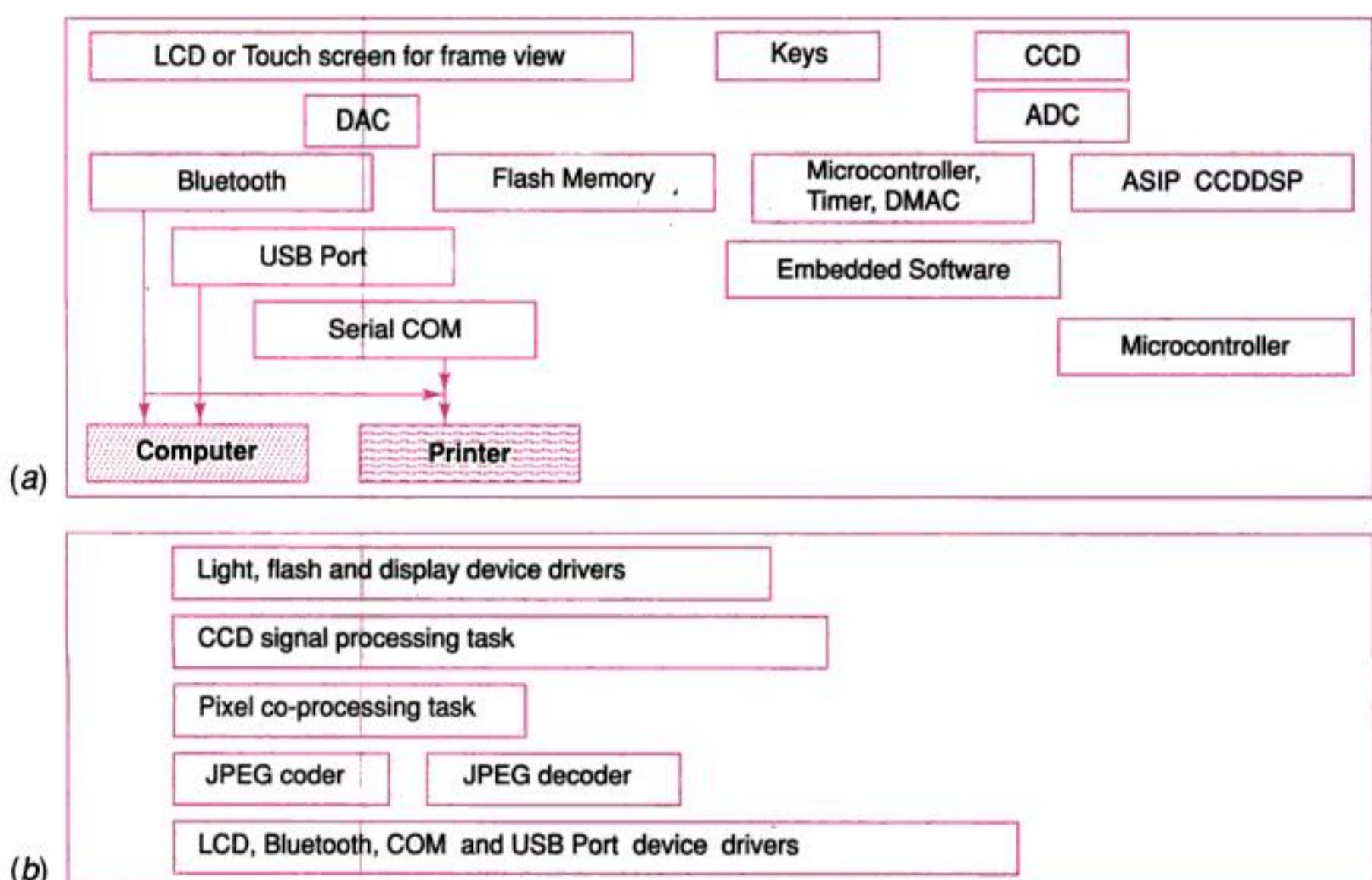


Fig. 1.14 (a) Digital camera hardware components (b) Digital camera software components

- 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.
- Shutter, lens and charge coupled device (CCD) array sensors for images in sizes 2592×1944 pixels = 5038848 pixels, VGA (E-mail) 640×480 = 307200 pixels, 2592×1728 = 3.2 M pixels, 2048×1536 pixels = 3 M pixels, or 1280×960 pixels = 1 M pixels.
- 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.
- It has a self-timer lamp for flash.
- Internal memory flash to store OS and embedded software, and limited number of image files.
- 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

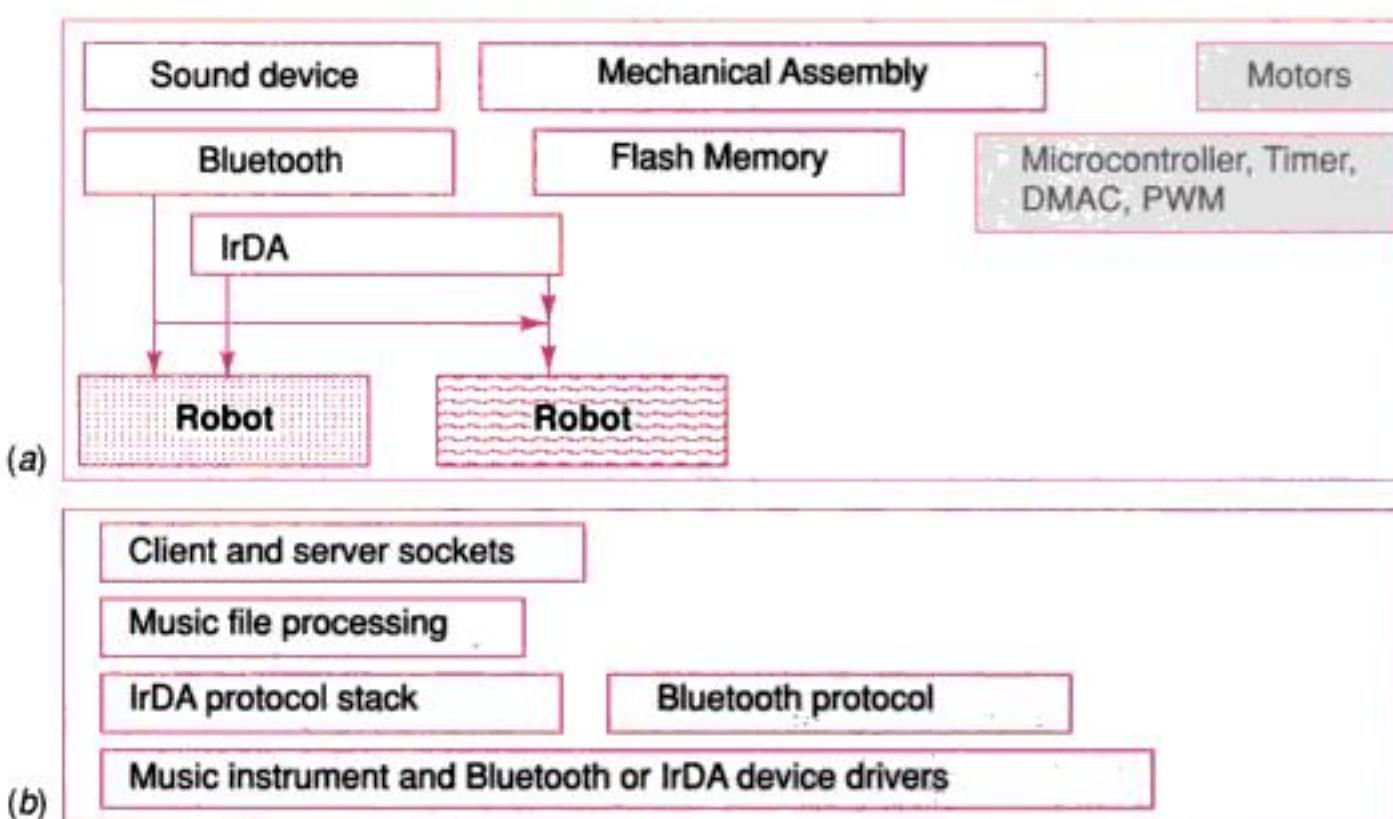


Fig. 1.15 (a) Hardware components in the set of robots (b) software components in the set of robots in which a master robots signals the commands and slave robots play according to the signals from the master

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 on 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 handing 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.

- The design process is abstracted by (i) definitions and analysis of system requirements, design metrics (Table 1.8) and validation requirements for finally developed systems specifications. There has to be consistency in the requirements, (ii) specifications (iii) architecture (iv) components (v) system integration.
- Design metrics are power dissipation, performance, process deadlines, user interfaces, size, engineering cost, manufacturing cost, flexibility, prototype development time, time-to-market, system and user safety and maintenance.
- The challenge in the process designing the system is to optimize competing design metrics.



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 15×12.5×1.2 mm³ 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 ~1M 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) Transceiver.
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 (ii) $V_{ref-} = -1.024$ 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

R

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

In this chapter, we will learn the following

E

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

A

R

N

I

N

G

O

2.1 8051 ARCHITECTURE

B

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

J

2.1.1 8051 Microcontroller Architecture

E

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:

C

T

I

V

E

S

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.

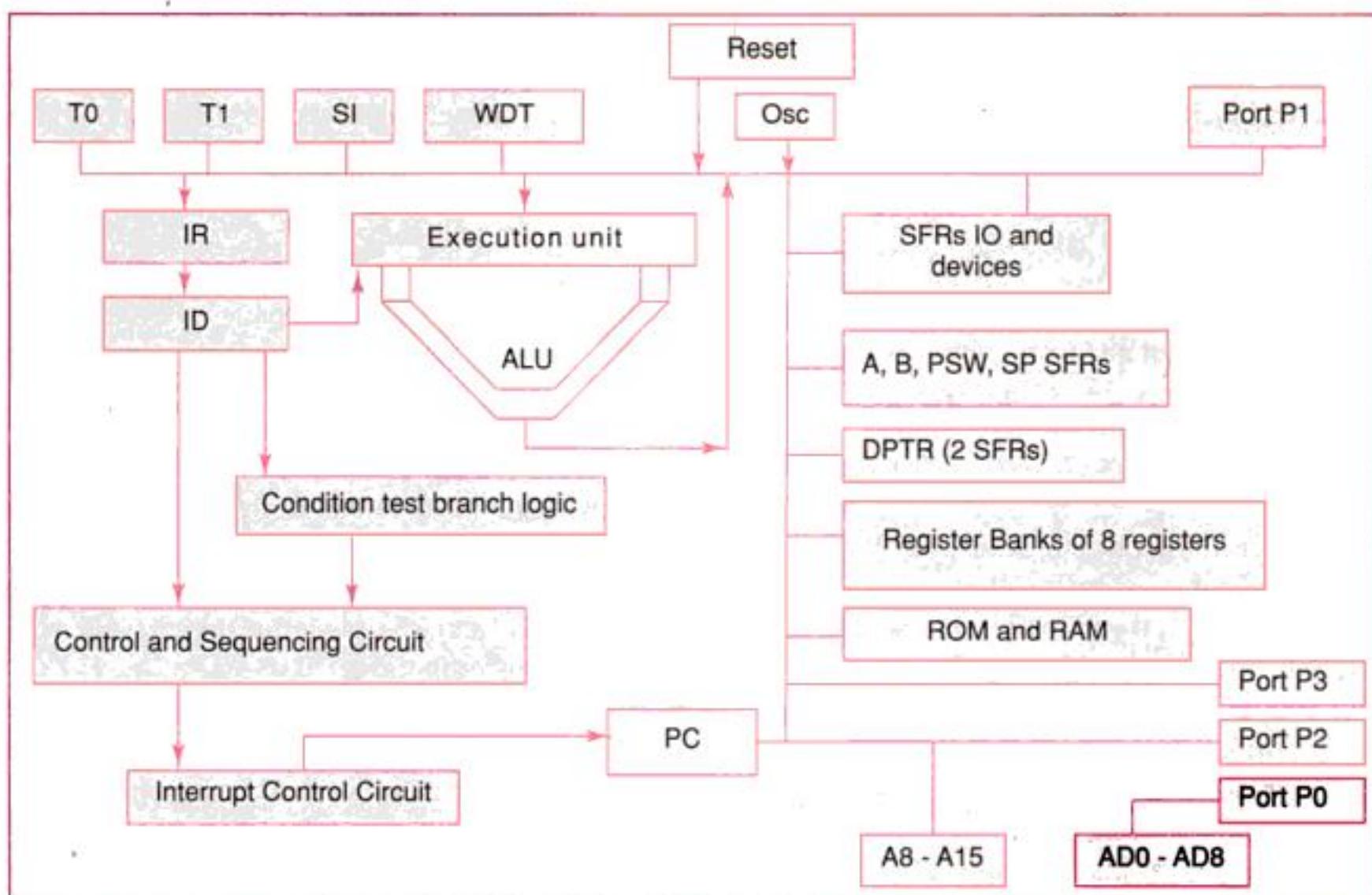


Fig. 2.1 8051 Architecture

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 no 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 $MOV \text{ direct}, \#\text{data}$, $MOV A, \#\text{data}$, $MOV R_n, \#\text{data}$ and $MOV @R_i, \#\text{data}$ 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 $MOV \text{ direct}, \text{direct}$, $MOV A, \text{direct}$, $MOV \text{ direct}, A$, $MOV \text{ direct}, R_n$, $MOV R_n, \text{direct}$, $MOV \text{ direct}, @R_i$, and $MOV @R_i, \text{direct}$ 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 $MOV \text{ DPTR}, \text{data16}$ 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 $MOVC A, @A + \text{DPTR}$ and $MOVC A, @A + \text{PC}$.

For stack operations, there are $PUSH \text{ direct}$ and $POP \text{ direct}$ 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.

Data Transfer Instructions

- Move byte between accumulator (an SFR) and register at a register bank
- Move byte from an SFR/Internal RAM to another *direct*
- Move *indirect*
- Move *immediate*, MOV immediate DPTR
- MOVC and MOVX *indirect*
- PUSH *direct*, Pop *direct*

Bit Manipulation (Boolean processing Instructions)

- Clear, Set, Complement, AND or OR or MOV the bit

Byte Manipulation Instruction

- Clear, Complement, Swap and Rotate Instructions

Logic Instructions

- AND, XOR, OR Operation Instructions

Arithmetic Instructions

- Arithmetic Instructions
- Increment-Decrement Instructions

Program Flow Control Instructions

- Branch instructions
- Conditional jumps
- Decrement and Jump conditional
- Compare and then conditional jump
- Subroutine Call Instructions
- NOP
- Delay

Interrupt Flow Control Instructions

- Interrupt flow control- mask bits, priority bits
- RETI

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 $A - B \leftarrow A + B$. 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. ($A \leftarrow A . R_n$). 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 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.

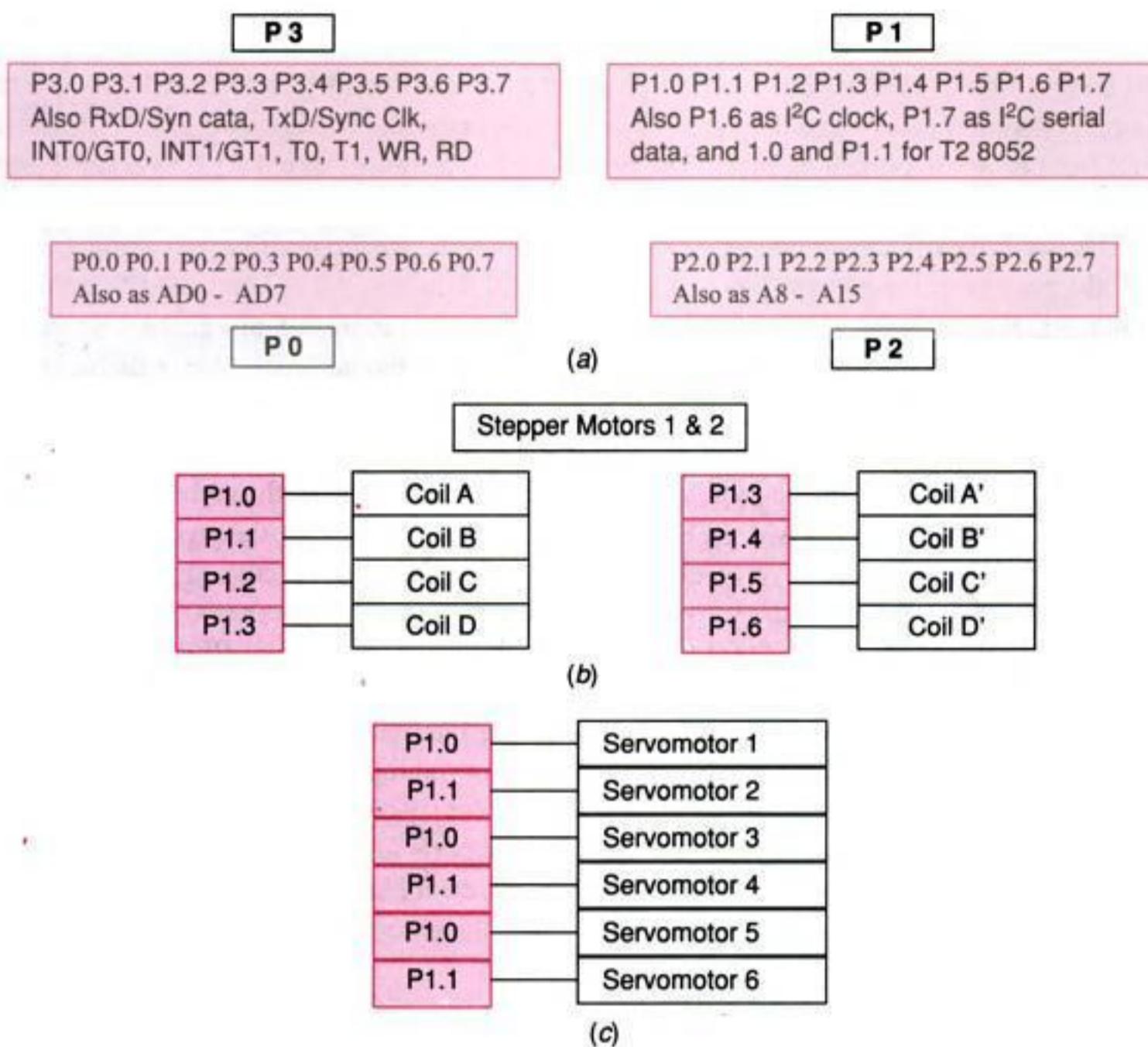


Fig. 2.3 (a) IO ports in 8051 (b) IO port P1 circuit for two stepper motors in a printer
(c) IO port circuit for six servo motors in a robot

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 CLRB 0x80 will make gain P0.0 as 0. A pulse of time-period T2 and duty cycle $100 \cdot T1/(T1 + T2)$ 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, 0xB2 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 RxD and TxD 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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 an 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

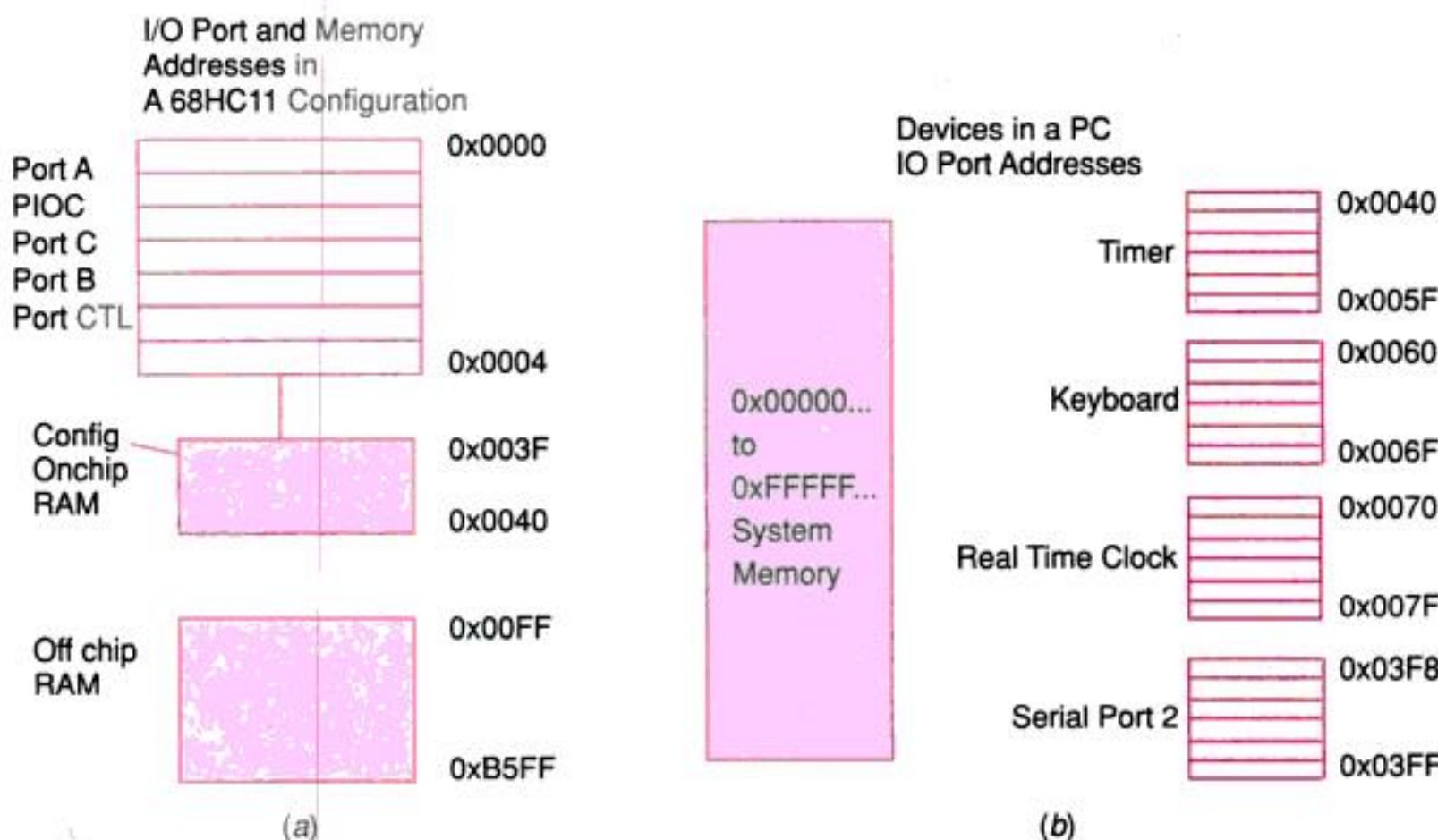


Fig. 2.12 (a) Processor and memory organization with I/O devices memory assignments in 68HC11 (with memory mapped IO architecture) (b) Device Addresses in the 80x86-based IBM PC

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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

BG signals, BG0, BG1, ..., BG n for each controller. An i^{th} controller sends BR i (i^{th} bus request signal) and when it receives BG i (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 p 2 , p 1 , p 0 , 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 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 FLPU 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)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

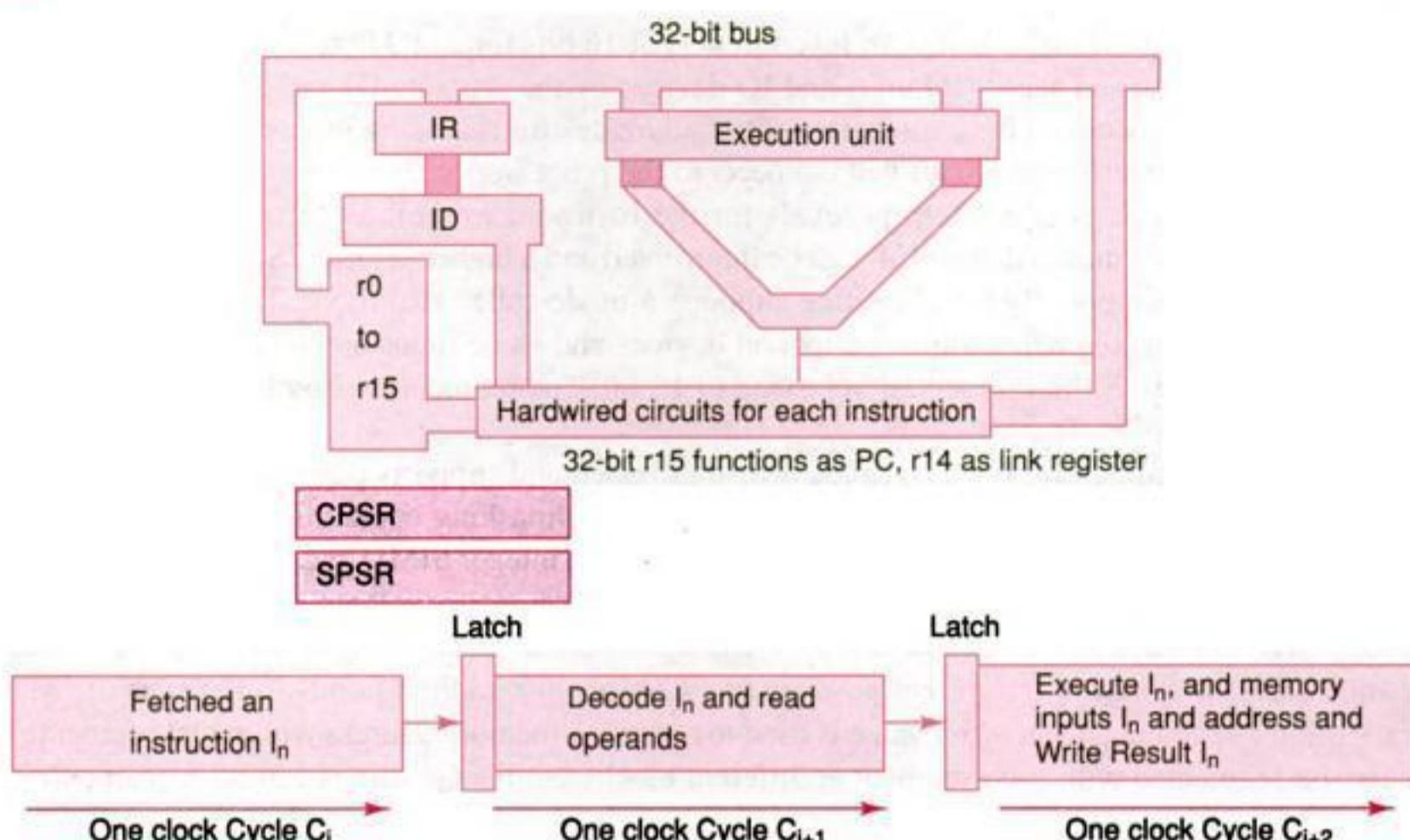


Fig. 2.19 ARM7 registers and three stage pipeline architecture

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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Feature	ARM7™ Thumb® Family	ARM9™ Thumb® Family	ARM11
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 µ 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 µ 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

5. 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.
6. 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-110, TI OMAP and 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

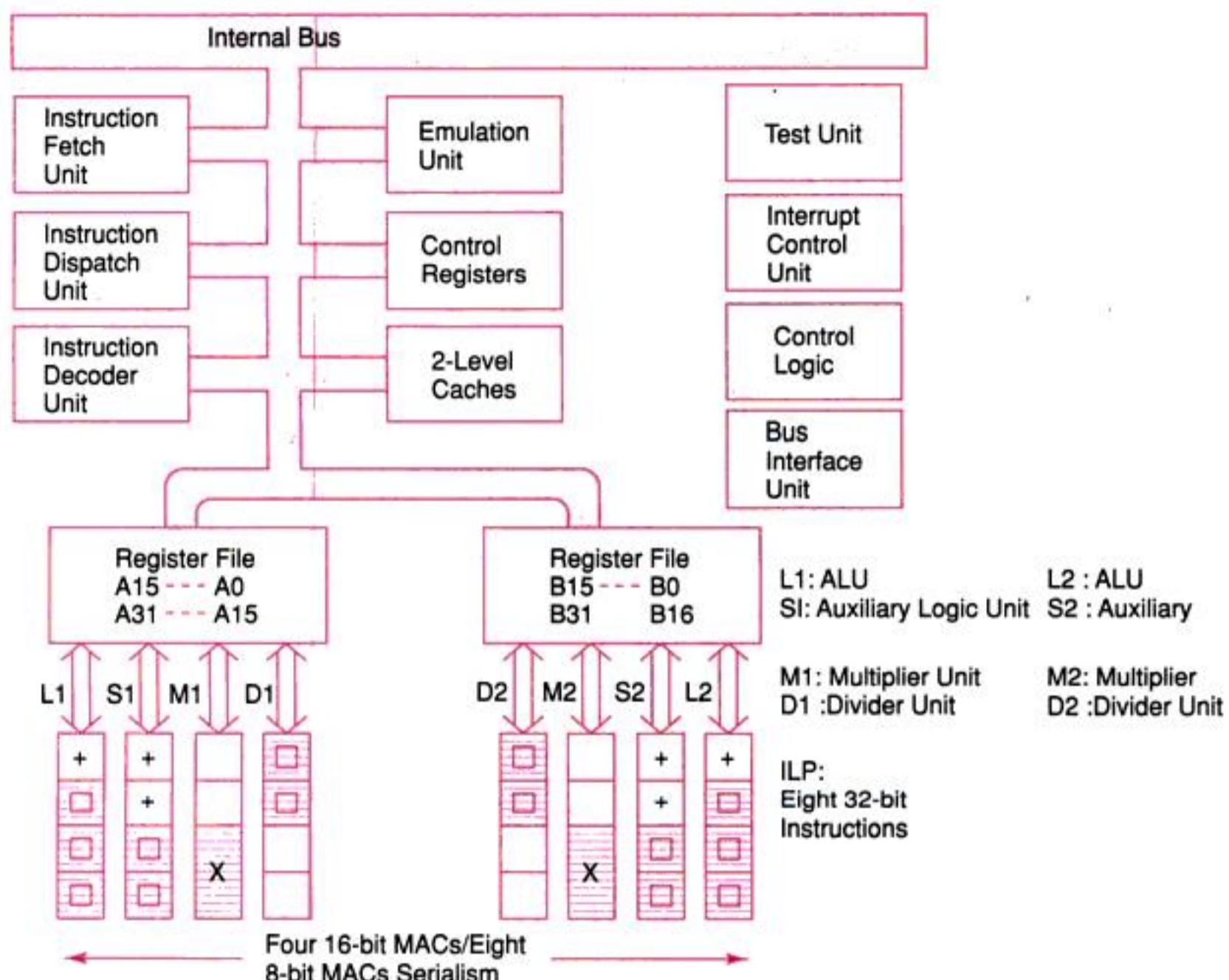


Fig. 2.21 Core and special structure units in DSP, TMS320C64x DSP
Note: Floating Point Units present in C67x

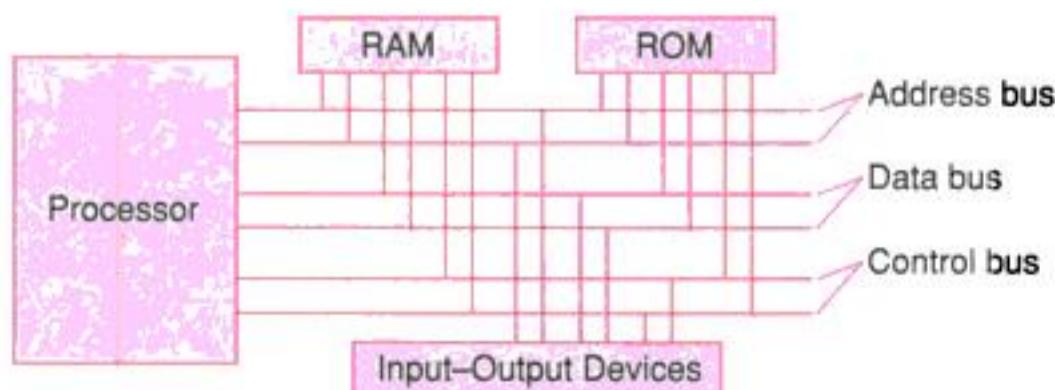


Fig. 2.22 A simple view of organization of processor, buses and memory in a system

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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- (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 modern 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
LittleEndian	EF	CD	AB	90
BigEndian	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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 IO or OS calls. Dhystones 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.](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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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-*



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- 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, voicedata 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

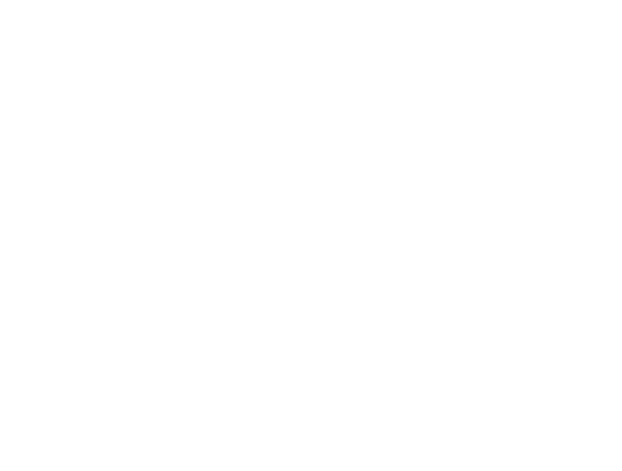
Consider another data acquisition system, which is used for recording the ECG waveforms. Let each waveform be recorded at 256 points. A 64 kB flash will be required for 256 patient records.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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?



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 retrieves 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 8 T. The bytes are received at constant rates. Each byte at the input port separates by 8 T and data transfer rate for the serial line bits is $1/T$ bps [$1 \text{ bps} = 1\text{-bit per second}$]. 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

IO Device Type	Examples
<i>Serial asynchronous output</i>	Output from modem, output for printer, the output on a serial line [also called UART output when according to UART mode]
<i>Parallel port single bit input</i>	(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
<i>Parallel port single bit output</i>	(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
<i>Parallel port input</i>	(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
<i>Parallel port output</i>	(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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(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 $1 \rightarrow 0$ 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 TxD and RxD 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Table 3.4 Processor with internal serial ports in microcontrollers

Features	<i>Intel 8051 and Intel 8751</i>	<i>Motorola M68MC11E2</i>	<i>Intel 80196</i>
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 up to 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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).



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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>Precedence</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)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 era 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 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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



Practice Exercises

15. How do the following device features help in embedded systems? (a) Schmitt trigger input (b) low voltage 3.3 V IOs (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 802.3oe draft 4.1 [10 Gbps Ethernet MAC] for 156.25 MHz dual direction performance (c) IEEE 802.3oe 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) XSBI (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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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*.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

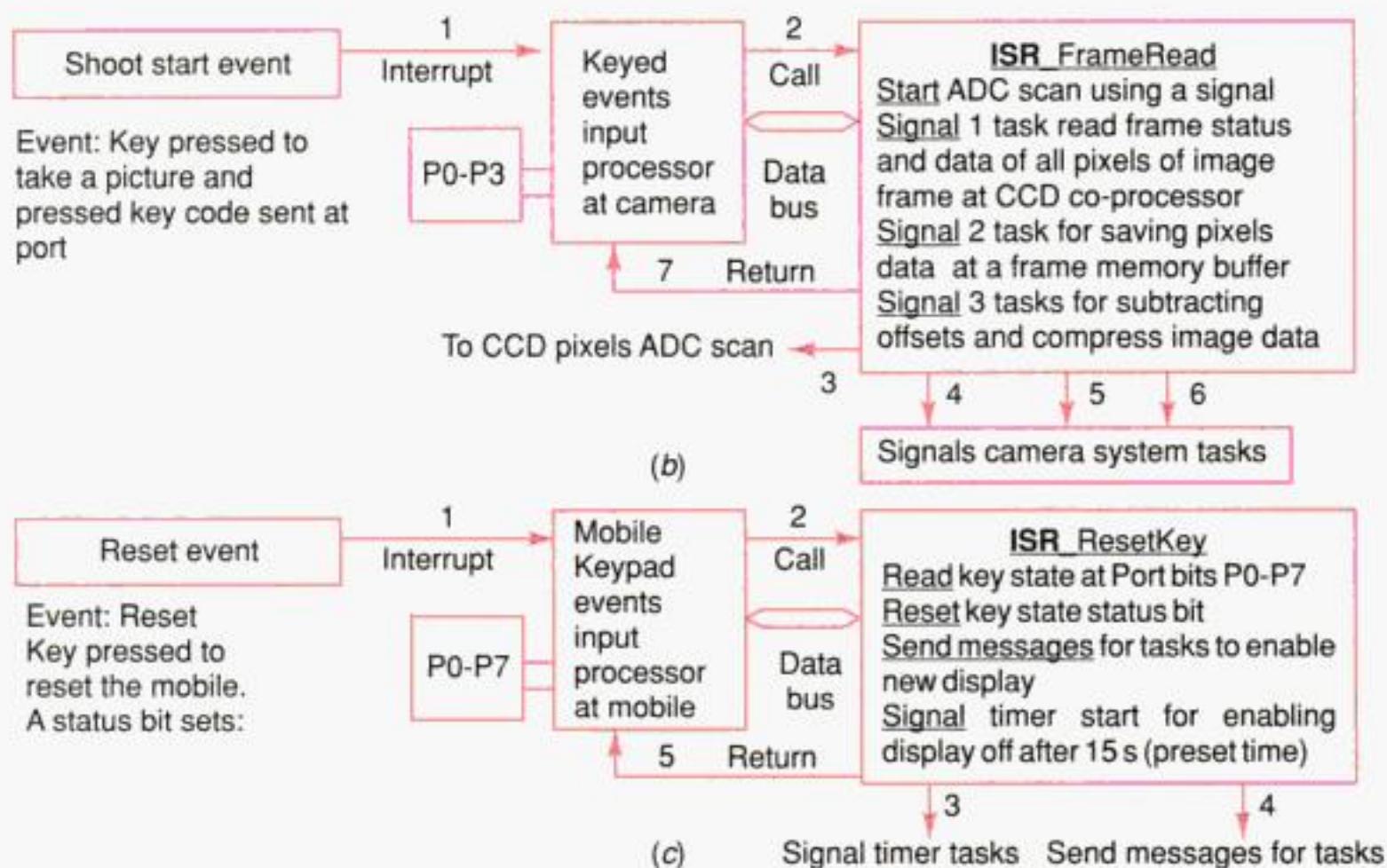


Fig. 4.3 (a) Use of ISR in the automatic chocolate vending machine (b) Use of ISR and three signals 1, 2 and 3 for three tasks in the digital camera example (c) Use of ISR in the mobile phone reset-key interrupt example

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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 interrupt 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.

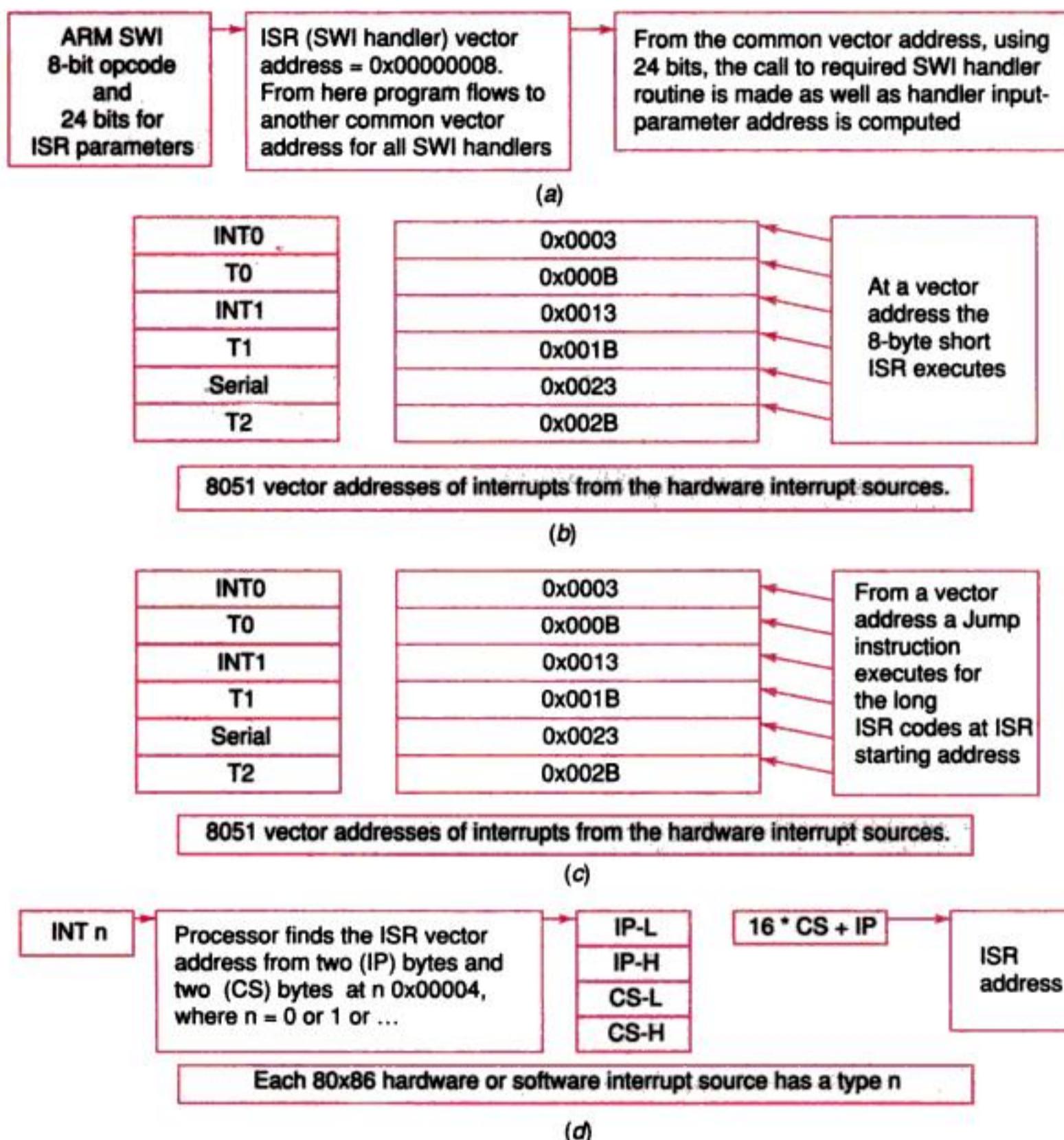


Fig. 4.7 (a) Use of ISR_VECTADDR in ARM for the jump to the routine for the interrupt servicing
 (b) Use of ISR_VECTADDR in 8051 in case of short-code interrupt service routine (ISR)
 (c) Use of ISR_VECTADDR in 8051 in case of long-code ISR (d) Use of ISR_VECTADDR address in 80x86 processors



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.

- 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 $T_{switch} + T_{disabled}$ because it is a top priority ISR.
- 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 $T_{switch} + T_{exec}$ (TCAPTURE) + T_{exec} (TF) + $T_{disabled}$, because it has the lowest priority than the timer interrupts.
- 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 $T_{switch} + T_{exec}$ (TCAPTURE) + $T_{disabled}$, 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_Full interrupt. If ISR for RI interrupt does not execute fast, it does not matter, because eventually there is a cushion of SI raising 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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).



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- (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 = $length_1$ from addresses memTxAddress to memTxAddress + $length_1 - 1$.
- (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 $2 \mu s$. Assume that an SWT is to be programmed to tick every $31 \times 32,768 \mu s = 1.015808$ 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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 ' <i>read and write</i> ' 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</i> function 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.



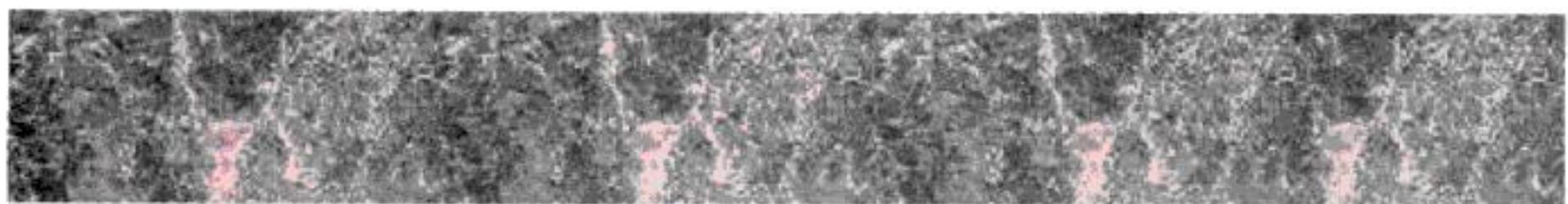
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- (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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 $= 4 \times 4 + 4 \times 4 + 4 \times 4 = 48\text{ B}$.
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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 $*Q_{\text{limit}}$, returns to its starting value $*Q_{\text{start}}$. (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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 RTCSTW_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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

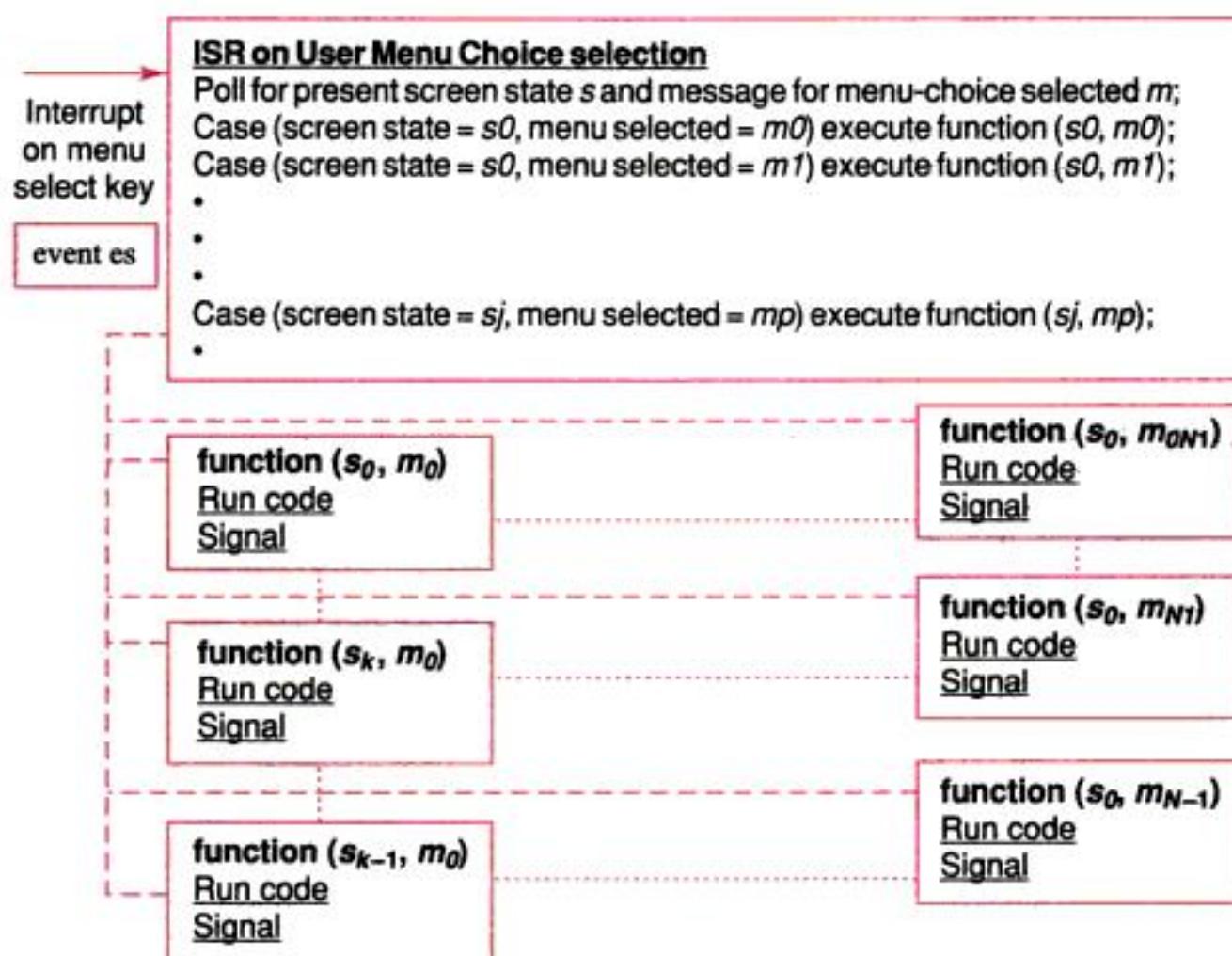


Fig. 5.4 Programming model use here, which facilitates execution of one of the multiple possible function calls and the function executes after polling for screen state j and for a message m from an interrupt service routine as per the user choice

```

}

/*****
void poll_Screen_State (j)
/* 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 ( );
}

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

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*/
    outPortB (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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

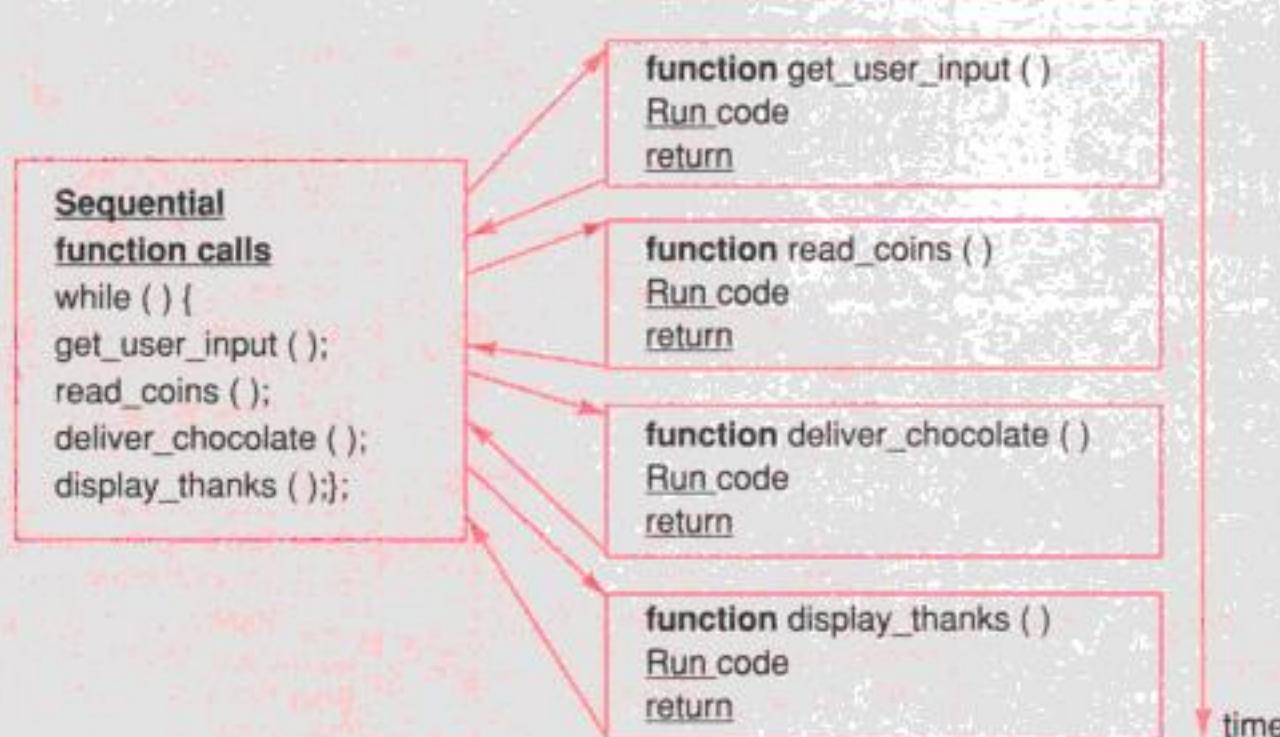


Fig. 6.1 Sequential programming model of an ACVM

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_thanks()` to start.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.

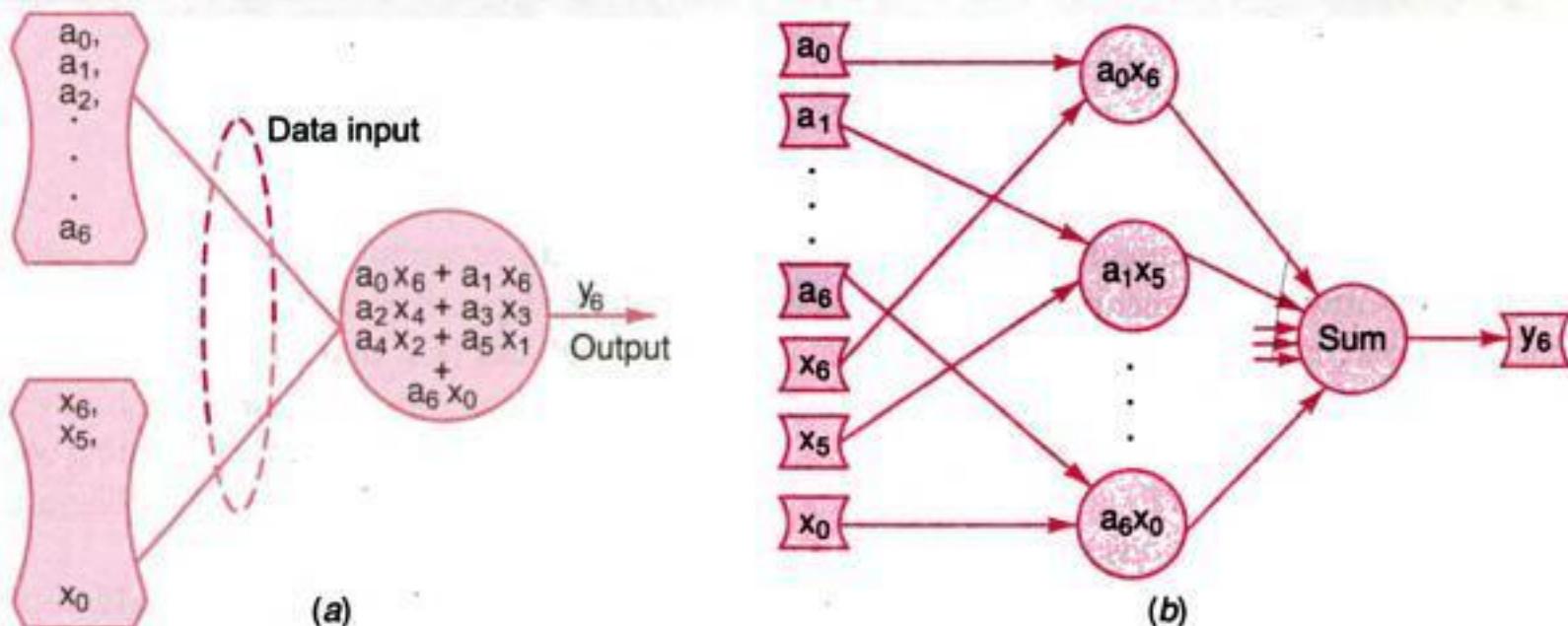


Fig. 6.4 (a) Data flow graph (DFG) for a process for the sixth finite impulse response (FIR) sequence
(b) DFG for a set of processes of the same sequence for an FIR filter with 6 inputs and 6 coefficients

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.

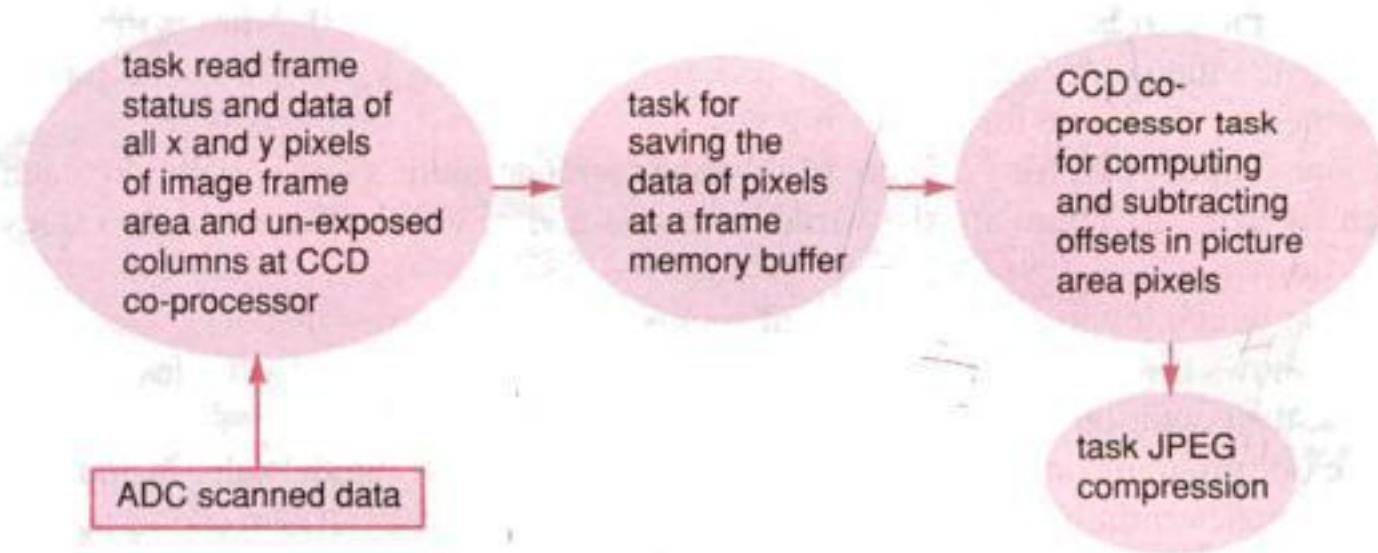


Fig. 6.5 DFG model for program for saving a picture in a digital camera



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

1. 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.
2. The '*running state*' has instructions being executed and the PC continuously changes as per the program flow.
3. 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.
4. Program flow to the '*running state*' again occurs on a token from the scheduler and after retrieving the values from the stack.
5. The flow to the '*finish state*' happens when the instruction reaches the end stage. The output is a message to the scheduler.
6. The flow to the '*ready state*' instead of '*finished state*' occurs when the tasks are in an infinite waiting loop.
7. 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.

1. The state one of the finite number of states.
2. Finite set of inputs (tokens or event flags or status flags) with their values for the state.

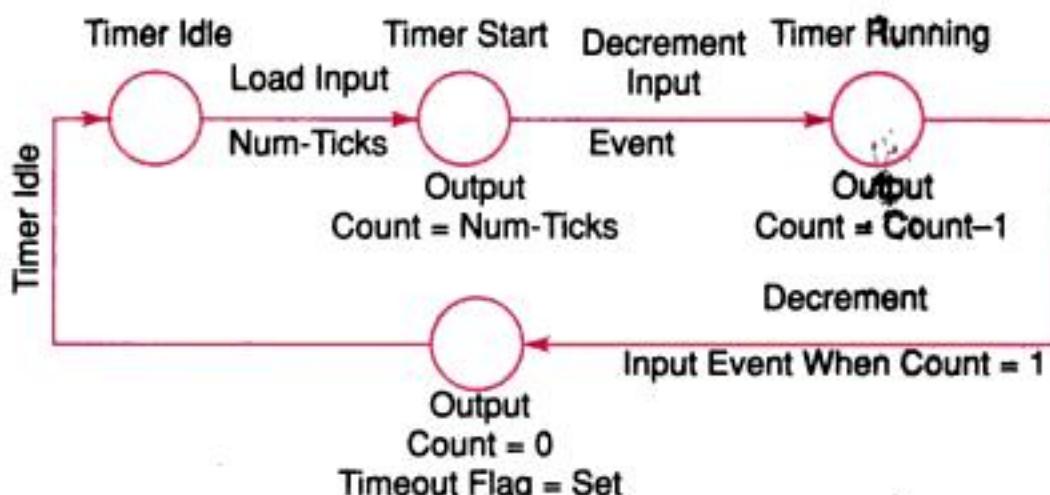


Fig. 6.8 States of a timer using finite states machine model



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
# 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 keying 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 ("k"); State = State4;
    }
    break;
    ****
State4: if ((KF == 0) && Count == x)) {
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



L

E

A

R

N

I

N

G

O

B

J

E

C

T

I

V

E

S

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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 μCOS-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) *GUI_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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 */
```

7.6.2 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.]



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
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 sTask2 is 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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.

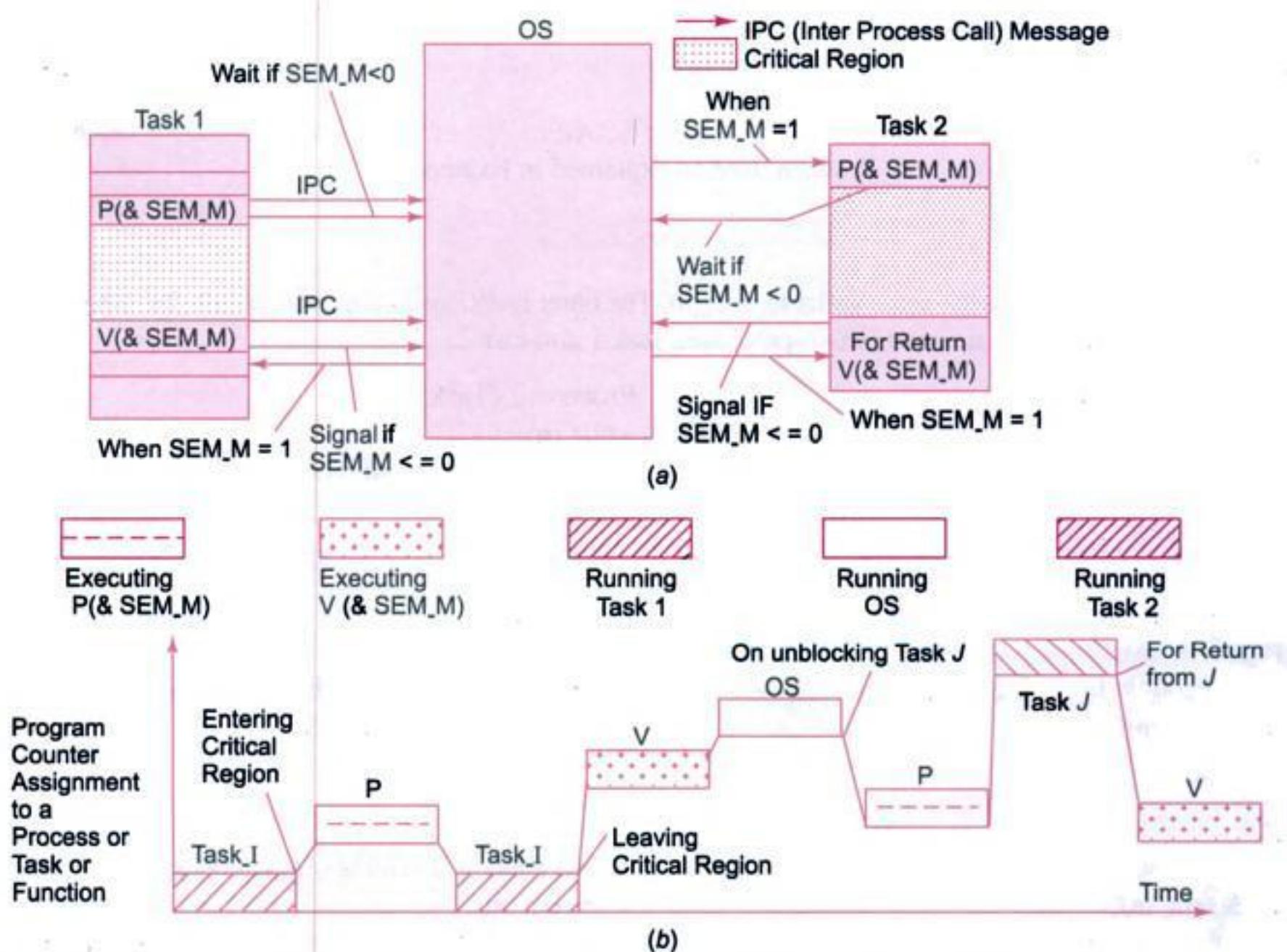


Fig. 7.7 (a) Use of P and V semaphores at a task 1, and at another task 2 and at a scheduler (b) The program counter assignments to a process or function when using P and V semaphores

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*/
    .
    .
}
```



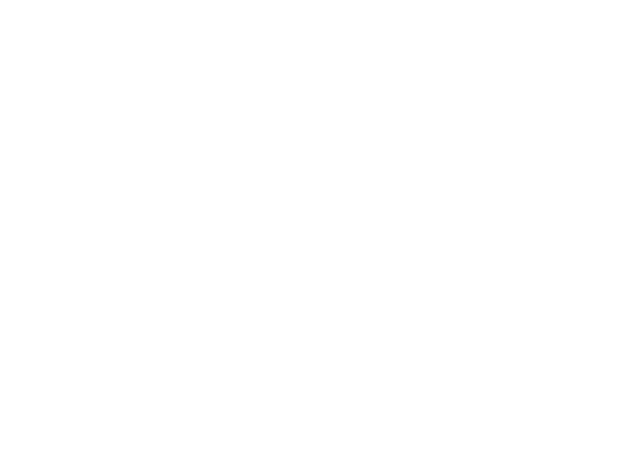
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

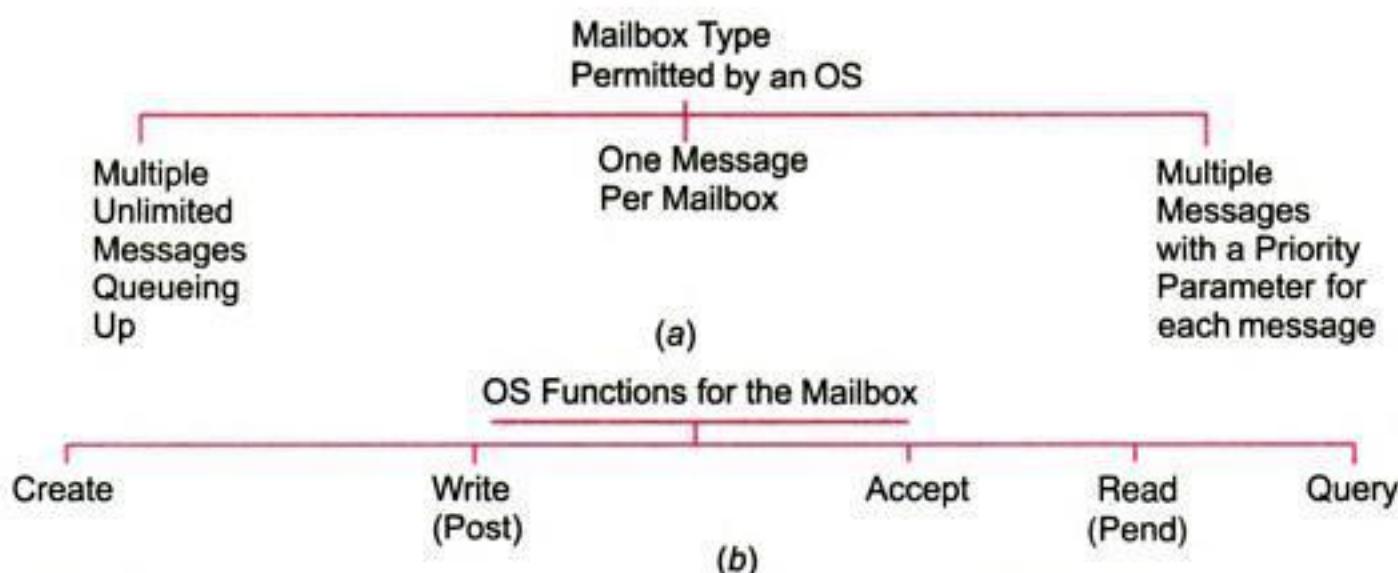


Fig. 7.9 (a) Mailbox types at the different operating systems (OSes) (b) Initialization and other functions for a mailbox at an OS

3. OSMBoxWait (pend) waits for a mailbox message, which is read when received.
4. OSMBoxAccept reads the current message pointer after checking the presence yes or no (no wait). Deletes the mailbox when read.
5. 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) {

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

We will learn the following in this chapter.

1. OS services.
2. Program structure, in-between layers of OS and interfaces between the top application software and down system-hardware layers.
3. OS functions for the process (task), timer, event, memory, device, file-system and I/O-subsystem management.
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.
5. ISR-handling in an RTOS environment.
6. Basic design principles when using an RTOS.
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.

8.1 OS SERVICES

8.1.1 Goal

The OS goals are *perfection and correctness* to achieve the following:

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.
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.
3. *Optimally scheduling* the processes on one (or more CPUs if available) and providing an appropriate context-switching mechanism.
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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 drive 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)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.

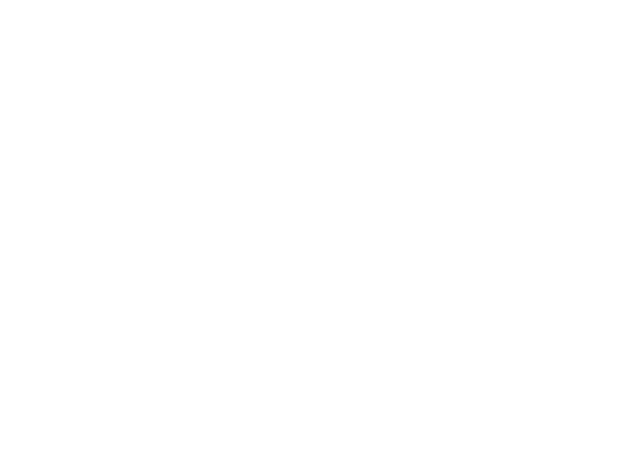
I/O subsystems are an important part of OS services. Examples are the UART access and the parallel port access. There are synchronous and asynchronous IOs. A task gets blocked during the synchronous IOs, for example, *fread()* or *write()* (Section 7.14). RTOSes support asynchronous IOs, for example, *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*.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Table 8.11 Real-Time Operating System (RTOS) Services

<i>Function</i>	<i>Activities</i>
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	Priorities 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)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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*/  
        OSSemPost (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 preempt the task chocolate delivery. Even if the user does not insert any coin, the task will wrongly post semChocolate and SemDisplayCollect.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 A_1 , task A_2 and task A_3 in a set of tasks A_1 to A_N . Figure 8.2(a) shows the first three tasks of the multiple process embedded software. The scheduler first starts the task A_1 waiting loop and waits for the message A_1 from task A_1 .

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 A_1 to schedule task A_2 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 A_1 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 A_2 to schedule task A_3 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 A_2 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 A_3 , 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 A_1 context has a pointer for task A_1 , `ADDR_A1`. Task A_2 context has a pointer for task A_2 , `ADDR_A2`. Task A_3 context has a pointer for task A_3 , `ADDR_A3`.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- The priorities when executing codes are first the ISRs, then ISTs and then threads of the processes.
- 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.
- 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.
- 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 *pmda, 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_PRIO – 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_PRIO 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
13. } /* End of FirstTask Codes */
/*****The codes for the ISR_CharIntr ****/
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 */
/*****The codes for the Task_Decrypt_Port_A ****/
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*/
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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_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_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_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_SendPortB*. If there is no error, the task sends the message of port B.
6. Step *f*: *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);
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 OSQuery (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 OSQuery 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 TIPCs 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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.

- (a) Declare initial value as SEM_EMPTY (not available)
- (b) Use semTake() in a task, which makes SEM_FULL to signal an event to another task
- (c) 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 after 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).



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 SemFlag1ID 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 SemFlag1ID 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 SemFlag1ID state change to SEM_FULL by semGive function of character availability
check task */

semTake (SemFlag1ID, 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 */



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



L
E
A
R
N
I
N
G

O
B
J
E
C
T
T
I
V
E
S

- 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 elegance 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.]



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

¹ CE_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.

² 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_SENDDMESSAGE (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_LBUTTONDOWNDBLCLK message is sent on the double tap of the stylus. For each message the parameters lParameter = two 16-bit screen tap horizontal and vertical position values x and y, and wParameter = 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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. struct sigaction signal_1; /* statement in C defines a structure signal_1. */ 2. signal_1.sa_flags =0; /* Sets flags = 0*/ 3. signal_1.sa_handler = handlefunction_1; /* Defines signal handler as a user defined function handlefunction_1. */ 4. sigemptyset (&signal_1.sa_mask); /* Defines signal as empty and sa_mask masks the execution of signal handler. */ 5. sigaction (SIGINIT, &signal_1, 0); /* Initiates signal function handlefunction_1() using the structure signal_1. */
3	Thread functions	<ol style="list-style-type: none"> 1. struct pthread_t clientThd_1, clientThd_2, ServingThd; /* statement in C defines three structures for POSIX threads clientThd_1, clientThd_2, ServingThd. */ 2. 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. 3. pthread_self () ; /* Returns (gets) ID of the function pthread_self calling thread (self) */ 4. 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. 5. pthread_exit ("text") /* exits the thread under execution and text displays on console on exit. */ 6. sleep (<i>sleeptime</i>); /* The thread sleeps for a period = <i>sleeptime</i> nanoseconds. Other thread gets the CPU in sleep interval*/ 7. pthread_kill () /* Sends 'kill' signal to the thread */
4	Semaphore functions	<ol style="list-style-type: none"> 1. struct sem_t sem1, sem2; /* statement in C defines three structures for semaphore structure (event control block), sem1 and sem2. */

(Contd)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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 taping 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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



L
E
A
R
N
I
N
G

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.

O
B
J
E
C
T
I
V
E

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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

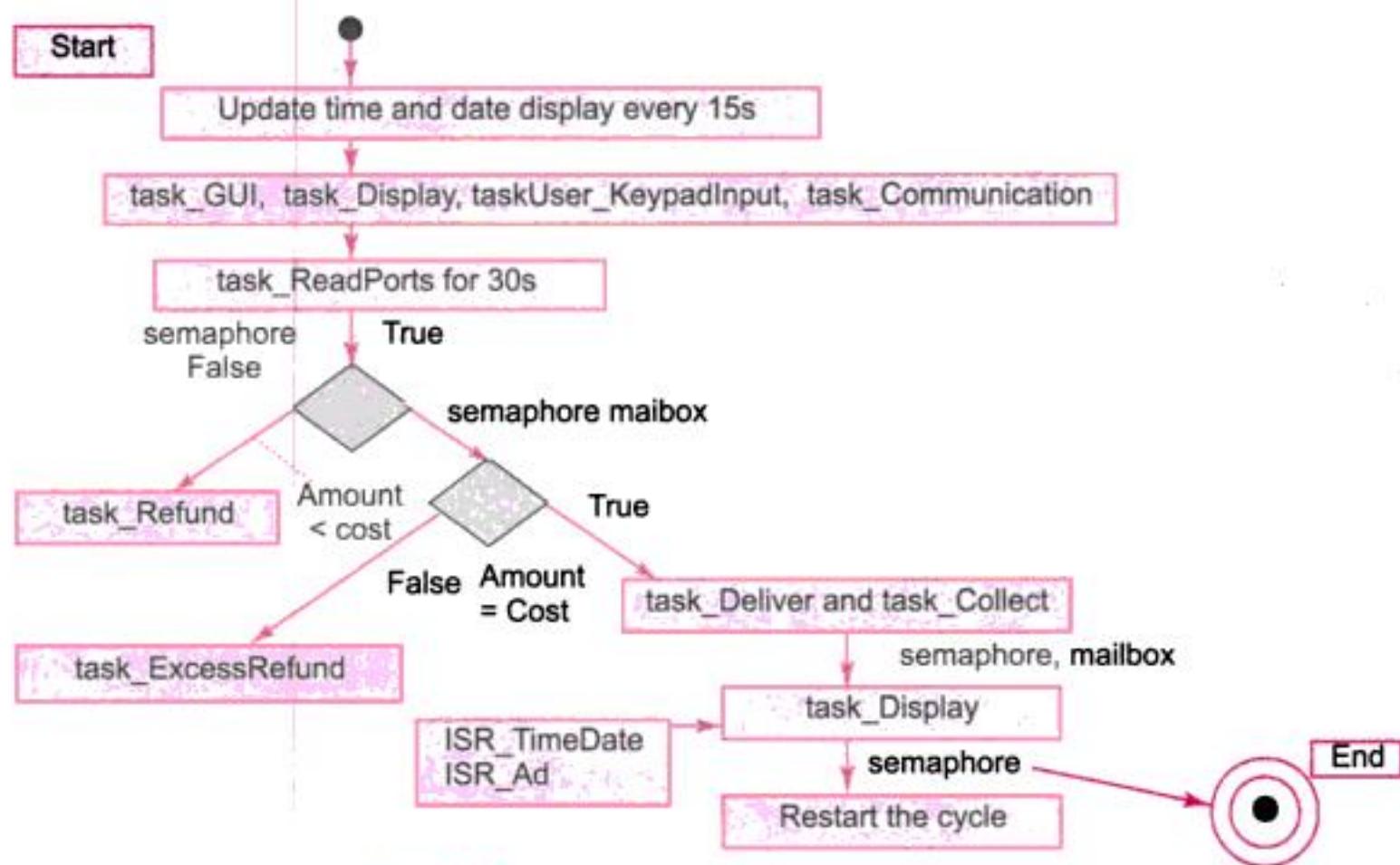


Fig. 11.4 State diagram for ACVM tasks

3. 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.
4. 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.
5. 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.

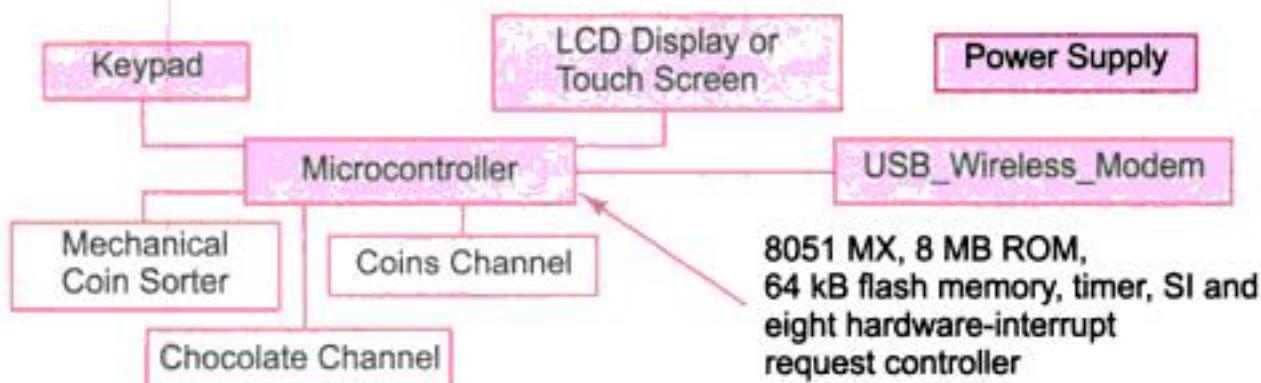


Fig. 11.5 Block diagram of ACVM hardware including Microcontroller

6. 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
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);
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Table 11.3 Requirements of a Digital Camera

<i>Requirement</i>	<i>Description</i>
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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

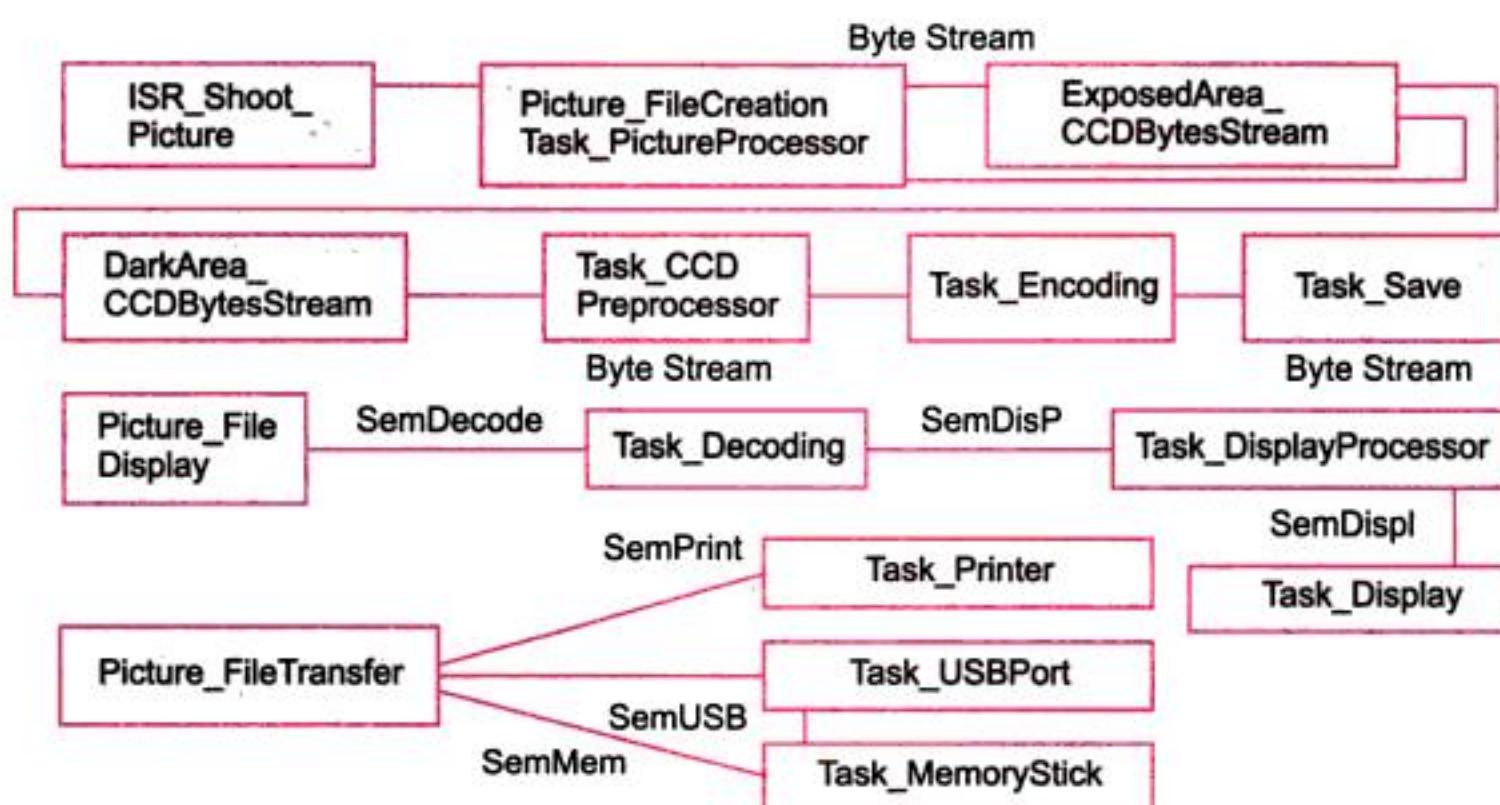


Fig. 11.11 Synchronization model for camera tasks

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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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>
Functions of the system	<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>Extendibility:</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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

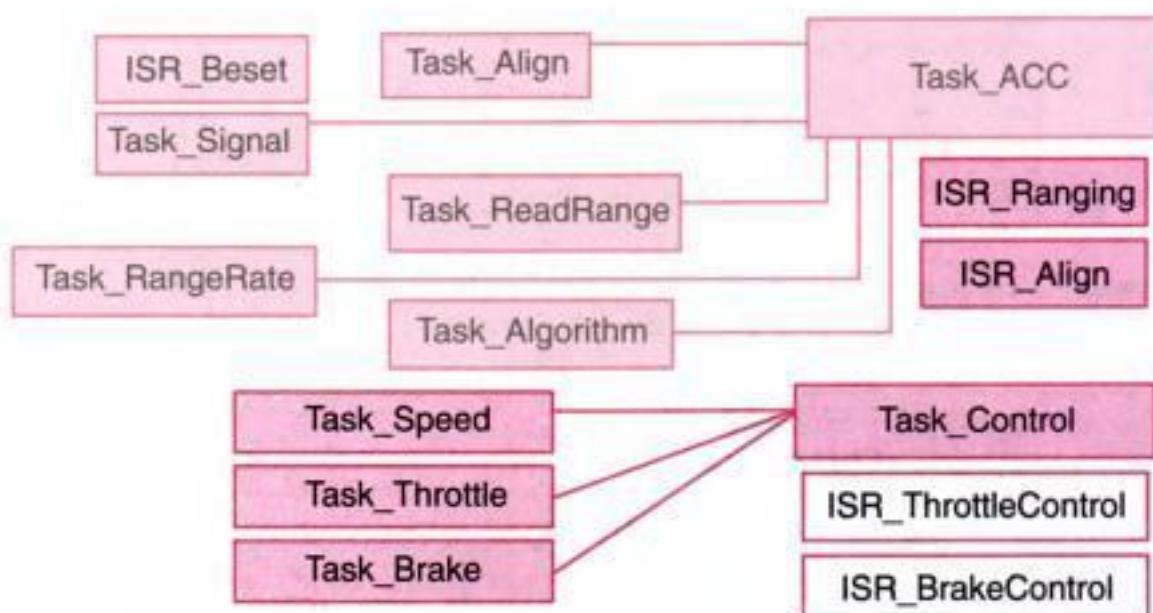


Fig. 12.13 Two class diagrams of Task_ACC and Task_Control

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).



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
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
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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_Del_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 */  
STAF_In = 0; /*Define flag for signaling modem interrupt for receiving a character. */  
STAF_Out = 0; /*Define flag for signal from a modem interrupt after sending a character. */  
/*-----*/  
3. /* 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);  
4. /* 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];  
5. /* 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*/  
6. /* Definitions for three task priorities. */  
#define task_ReadPortPriority 2 /* Define task 2 priority */  
#define task_PWPriority 3 /* Define task 3 priority */  
6. /* 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  
host. */
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Embedded Software Development Process and Tools

13

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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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. Overridden 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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)	Check-sum 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.)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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, ICEs, 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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. <i>Assert</i> macro tests	The use of an <i>assert</i> macro is an important test technique. For example, consider a command, ‘ <i>assert (pPointer != NULL);</i> ’. 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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 recognises 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Embedded Systems

Architecture, Programming and Design

Embedded Systems, 2e, comes with an extensive account of the subject ensuring balanced coverage of hardware and software concepts. The book now gives due weightage to architecture, programming and design aspects. Comprehensive treatment of popular Real Time Operating Systems along with case studies will help the students be in tune with the requirements of the industry.

Salient features

- Bottom-up approach**—hardware and software issues have been discussed followed by case studies
- Real Time Operating Systems discussed in detail
- Design process and examples** are covered throughout the book
- Practical orientation:** Two chapters dedicated to case studies

New to this edition

- Better balance between Hardware and Software aspects achieved by including 8051 and other advanced microprocessor architectures
- Discussion of two new popular RTOSes—**Windows CE** and **RTLinux**
- New topics on Design process in embedded systems, Formalism of system design, Wireless communication devices, Internet enabled systems, Wireless and Mobile system protocols
- Expanded coverage of Memory organization, Bus architecture and Threads
- Fresh new case studies on Digital camera, Robot orchestra and Mobile phone SMS creating and sending
- Embedded programming in Java included

www.tatamcgrawhill.com

<http://www.mhhe.com/kamal/emb2>

ISBN-13: 978-0-07-066764-8

ISBN-10: 0-07-066764-0



9 780070 667648



Tata McGraw-Hill

Copyrighted material