SRI VENKATESHWARAA
COLLEGE OF ENGINEERING & TECHNOLOGY
(Approved by AICTE, New Delhi & Affliated to Pondicherry University)
13-A Villupuram- Puducherry main road, Ariyur, Puducherry - 605 102
Phone: 0413-2644426, Fax – 2644424/ Website : www.svcetpondy.com

SYLLABUS

CS T62 – EMBEDDED SYSTEMS

UNIT I: Introduction to Embedded Systems - Processor in Embedded System – Other Hardware Units in the Embedded System - Software Embedded into a System - ARM Architecture: ARM Design Philosophy - Registers - Program Status Register - Instruction Pipeline - Interrupts and Vector Table - Architecture Revision - ARM Processor Families.

UNIT II: ARM Programming - Instruction Set - Data Processing Instructions - Addressing Modes - Branch, Load, Store Instructions - PSR Instructions - Conditional Instructions.

UNIT III: Thumb Instruction Set - Register Usage - Other Branch Instructions - Data Processing Instructions - Single-Register and Multi Register Load-Store Instructions - Stack - Software Interrupt Instructions

UNIT IV: ARM Programming using C: Simple C Programs using Function Calls – Pointers – Structures - Integer and Floating Point Arithmetic - Assembly Code is using Instruction Scheduling – Register Allocation - Conditional Execution and Loops.

UNIT V: Real Time Operating Systems: Brief History of OS - Defining RTOS - The Scheduler - Objects – Services - Characteristics of RTOS - Defining a Task - Tasks States and Scheduling - Task Operations – Structure – Synchronization - Communication and Concurrency. Defining Semaphores - Operations and Use - Defining Message Queue - States – Content – Storage - Operations and Use.

Textbooks:

1. Shibu K.V, Introduction to Embedded Systems, First Edition, McGraw Hill, 2009.
2. Andrew N. Sloss, Dominic Symes, Chris Wright, ARM Systems Developer's Guides- Designing & Optimizing System Software, Elsevier, 2008.
3. Qing Li, Real Time Concepts for Embedded Systems, Elsevier, 2011.

References:

1. Santanu Chattopadhyay, "Embedded System Design", Second Edition, PHI, 2013.
2. Andrew N Sloss, D. Symes and C. Wright, "ARM System Developers Guide", Morgan Kaufmann / Elsevier, 2006.
3. Wayne Wolf, "Computer as Components: Principles of Embedded Computer System Design", Elsevier, 2006.

TOTAL PERIODS: 60

UNIT – I

[VI SEM – CSE]

1.1 INTRODUCTION TO EMBEDDED SYSTEMS:

"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."

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.)

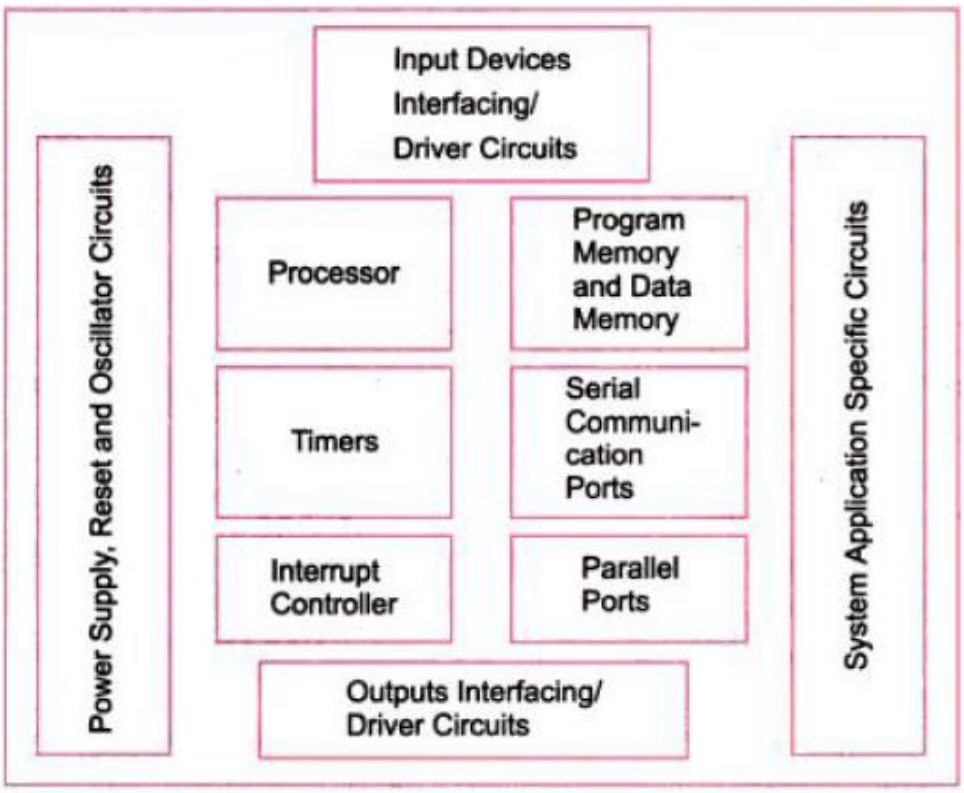


Fig. 1.1 The components of embedded system hardware

1.2 Challenges in Embedded Computing System Design:

External constraints are one important source of difficulty in embedded system design.

- *Hardware*
- *Deadlines*
- *Minimize power consumption*
- *Design for upgradability*
- *Reliability*

Hardware:

To select the type of microprocessor used, but also select the amount of memory, the peripheral devices, and more. Since often must meet both performance deadlines and manufacturing cost constraints, the choice of hardware is important—too little hardware and the system fails to meet its deadlines, too much hardware and it becomes too expensive.

Deadlines:

A deadline is to speed up the hardware so that the program runs faster. Of course, that makes the system more expensive. It is also entirely possible that increasing the CPU clock rate may not make enough difference to execution time, since the program's speed may be limited by the memory system.

Minimize power consumption:

In battery-powered applications, power consumption is extremely important. Even in no battery applications, excessive power consumption can increase heat dissipation. One way to make a digital system consume less power is to make it run more slowly, but naively slowing down the system can obviously lead to missed deadlines.

Upgradability:

The hardware platform may be used over several product generations or for several different versions of a product in the same generation, with few or no changes.

Reliability:

It is especially important in some applications, such as safety-critical systems. Another set of challenges comes from the characteristics of the components and systems them- selves. If workstation programming is like assembling a machine on a bench, then embedded system design is often more like working on a car—cramped, delicate, and difficult.

1.3 Performance in Embedded Computing:

Embedded system designers, in contrast, have a very clear performance goal in mind—their program must meet its ***deadline***. At the heart of embedded computing is ***real-time computing,*** the program receives its input data; the deadline is the time at which a computation must be finished.

CPU: The CPU clearly influences the behavior of the program, particularly when the CPU is a pipelined processor with a cache.

Platform: The platform includes the bus and I/O devices. The platform components that surround the CPU are responsible for feeding the CPU and can dramatically affect its performance.

Program: Programs are very large and the CPU sees only a small window of the program at a time.

Task: Generally run several programs simultaneously on a CPU, creating a ***multitasking system***. The tasks interact with each other in ways that have profound implications for performance.

Multiprocessor: Many embedded systems have more than one processor— they may include multiple programmable CPUs as well as accelerators. Once again, the interaction between these processors adds yet more complexity to the analysis of overall system performance.

1.3.1 EMBEDDED SYSTEM DESIGN PROCESS:

The design processes are only one axis along which we can view embedded system design. We also need to consider the **major goals of the design**:

- Manufacturing cost
- Performance (both overall speed and deadlines); and
- Power Consumption.

1.3.2 REQUIREMENTS OF EMBEDDED SYSTEM DESIGN:

Performance: The speed of the system is often a major consideration both for the usability of the system and for its ultimate cost. As we have noted, performance may be a combination of soft performance metrics such as approximate time to perform a user-level function and hard deadlines by which a particular operation must be completed.

Cost: The target cost or purchase price for the system is almost always a consideration. Cost typically has two major components: ***manufacturing cost*** includes the cost of components and assembly; ***nonrecurring engineering*** **(NRE)** costs include the personnel and other costs of designing the system.

Physical size and weight: The physical aspects of the final system can vary greatly depending upon the application. An industrial control system for an assembly line may be designed to fit into a standard-size rack with no strict limitations on weight. A handheld device typically has tight requirements on both size and weight that can ripple through the entire system design.

Power consumption: Power, of course, is important in battery-powered systems and is often important in other applications as well. Power can be specified in the requirements stage in terms of battery life the customer is unlikely to be able to describe the allowable wattage.

1.3.3 CHARACTERISTICS OF EMBEDDED SYSTEMS:

- Speed (bytes/sec)
- Power (watts)
- Size (cm3) and weight (g)
- Accuracy (% error)
- Adaptability.

An embedded system must perform the operations at a high speed so that it can be readily used for real time applications and its power consumption must be very low and the size of the system should be as for as possible small and the readings must be accurate with minimum error.

Software Issues: The important software issues related to the embedded system are mentioned below.

- Software maintenance is extremely important.
- Verification of proper operation,
- Updates for the software in periodic intervals are very important.
- Fixing the bugs in the software improves its efficiency and also a very important factor.
- Adding features, New features must be added to the software when ever they are available
- Extending to new applications, the software must be upgraded such that its applicability increases for new application areas.
- Change user configurations .This is an important factor to improve the popularity of the software.

1.3.4 APPLICATIONS: Embedded systems find wide variety of applications in various fields.

- Automobile
- Aeronautics
- Space
- Rail Transport
- Mobile communications
- Industrial processing
- Remote sensing , Radio and Networking
- Robotics
- Consumer electronics, music players, Computer applications
- Security (e-commerce, smart cards)
- Medical electronics (hospital equipment, and mobile monitoring) and
- Defense application

1.3.5 Design Metrics:

- *Unit cost:* the monetary cost of manufacturing each copy of the system, excluding
- NRE cost.

- *NRE cost (Non-Recurring Engineering cost):* The monetary cost of designing the system. Once the system is designed, any number of units can be manufactured without incurring any additional design cost (hence the term "non-recurring").
- *Size:* the physical space required by the system, often measured in bytes for software, and gates or transistors for hardware.
- *Performance:* the execution time or throughput of the system.
- *Power:* the amount of power consumed by the system, which determines the lifetime of a battery, or the cooling requirements of the IC, since more power means more heat.
- *Flexibility:* the ability to change the functionality of the system without incurring heavy NRE cost. Software is typically considered very flexible.
- *Time-to-market:* The amount of time required to design and manufacture the system to the point the system can be sold to customers.
- *Time-to-prototype:* The amount of time to build a working version of the system, which may be bigger or more expensive than the final system implementation, but can be used to verify the system's usefulness and correctness and to refine the system's functionality.
- *Correctness:* our confidence that we have implemented the system's functionality correctly. We can check the functionality throughout the process of designing the system, and we can insert test circuitry to check that manufacturing was correct.
- *Safety:* the probability that the system will not cause harm.

1.4 CLASSIFICATION OF EMBEDDED SYSTEMS:

- Based on generation
- Complexity and performance requirements
- Based on deterministic behavior
- Based on triggering

Based on generation:

1.4.1.1 First Generation The early embedded systems were built around 8bit microprocessors like 8085 and Z80, and 4bit microcontrollers. Simple in hardware circuits with firmware developed in Assembly code. Digital telephone keypads, stepper motor control units etc. are examples of this.

1.4.1.2 Second Generation These are embedded systems built around 16bit microprocessors and 8 or 16 bit microcontrollers, following the first generation embedded systems. The instruction set for the second generation processors/controllers were much more complex and powerful than the first generation processors/controllers. Some of the second generation embedded systems contained embedded operating systems for their operation. Data Acquisition Systems, SCADA systems, etc. are examples of second generation embedded systems.

1.4.1.3 Third Generation With advances in processor technology, embedded system developers started making use of powerful 32bit processors and 16bit microcontrollers for their design. A new concept of application and domain specific processors/controllers like Digital Signal Processors (DSP) and Application Specific Integrated Circuits (ASICs) came into the picture. The instruction set of processors became more complex and powerful and the concept of instruction pipelining also evolved. The processor market was flooded with different types of processors from different vendors. Processors like Intel Pentium, Motorola 68K, etc. gained attention in high performance embedded requirements. Dedicated embedded real time and general purpose operating systems entered into the embedded market. Embedded systems spread its ground to areas like robotics, media, industrial process control, networking, etc.

1.4.1.4 Fourth Generation The advent of System on Chips (SoC), reconfigurable processors and multicore processors are bringing high performance, tight integration and miniaturisation into the embedded device market. The SoC technique implements a total system on a chip by integrating different functionalities with a processor core on an integrated circuit. We will discuss about SoCs in a later chapter. The fourth generation embedded systems are making use of high performance real time embedded operating systems for their functioning. Smart phone devices, mobile internet devices (MIDs), etc. are examples of fourth generation embedded systems.

Complexity and performance requirements:

1.4.2.1 Small-Scale Embedded Systems Embedded systems which are simple in application needs and where the performance requirements are not time critical fall under this category. An electronic toy is a typical example of a small-scale embedded system. Small-scale embedded systems are usually built around low performance and low cost 8 or 16 bit microprocessors/microcontrollers. A small-scale embedded system may or may not contain an operating system for its functioning.

1.4.2.2 Medium-Scale Embedded Systems Embedded systems which are slightly complex in hardware and firmware (software) requirements fall under this category. Medium-scale embedded systems are usually built around medium performance, low cost 16 or 32 bit microprocessors/microcontrollers or digital signal processors. They usually contain an embedded operating system (either general purpose or real time operating system) for functioning.

1.4.2.3 Large-Scale Embedded Systems/Complex Systems Embedded systems which involve highly complex hardware and firmware requirements fall under this category. They are employed in mission critical applications demanding high performance. Such systems are commonly built around high performance 32 or 64 bit RISC processors/controllers or Reconfigurable System on Chip (RSoC) or multi-core processors and programmable logic devices. They may contain multiple processors/controllers and co-units/hardware accelerators for offloading the processing requirements from the main processor of the system. Decoding/encoding of media, cryptographic function implementation, etc. are examples for processing requirements which can be implemented using a co-processor/hardware accelerator. Complex embedded systems usually contain a high performance Real Time Operating System (RTOS) for task scheduling, prioritization and management.

Embedded processor in a system:

Processor technology involves the architecture of the computation engine used to implement a system's desired functionality. While the term "processor" is usually associated with programmable software processors. Each such processor differs in its specialization towards a particular application (like a digital camera application), thus manifesting different design metrics.

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.

Embedded systems are domain and application specific and are built around a central core. The core of the embedded system falls into any one of the following categories:

1. General Purpose and Domain Specific Processors
   - 1.1 Microprocessors
   - 1.2 Microcontrollers
   - 1.3 Digital Signal Processors
2. Application Specific Integrated Circuits (ASICs)
3. Programmable Logic Devices (PLDs)
4. Commercial off-the-shelf Components (COTS)

Microprocessor

The CPU is a unit that centrally fetches and processes a set of general-purpose instructions. The CPU instruction set includes instructions for data transfer operations, ALU operations, stack operations, IO operations and program control, sequencing and supervising operations. The general-purpose instruction set is always specific to a specific CPU. Any CPU must possess the following basic functional units:

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

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

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

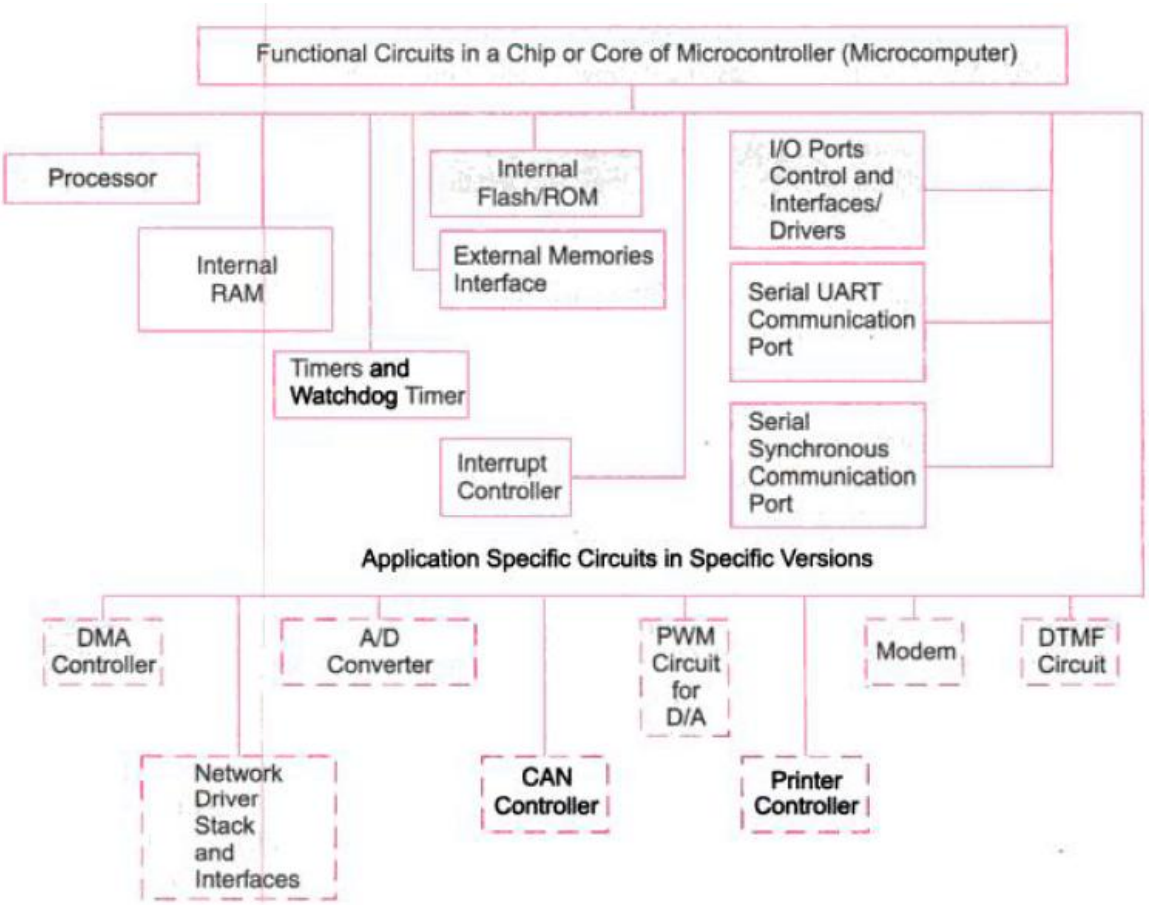
Intel 80x86 (also referred as x86) processors are the 32-bit successors of 8086. [The *x* here refers to an 8086 extended for 32 bits.] Examples of 32-bit processors in 80x86 series are Intel 80386, 80486 and Pentiums (a new generation of 32- and 64-bit microprocessors is the classic Pentium series).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

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

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.

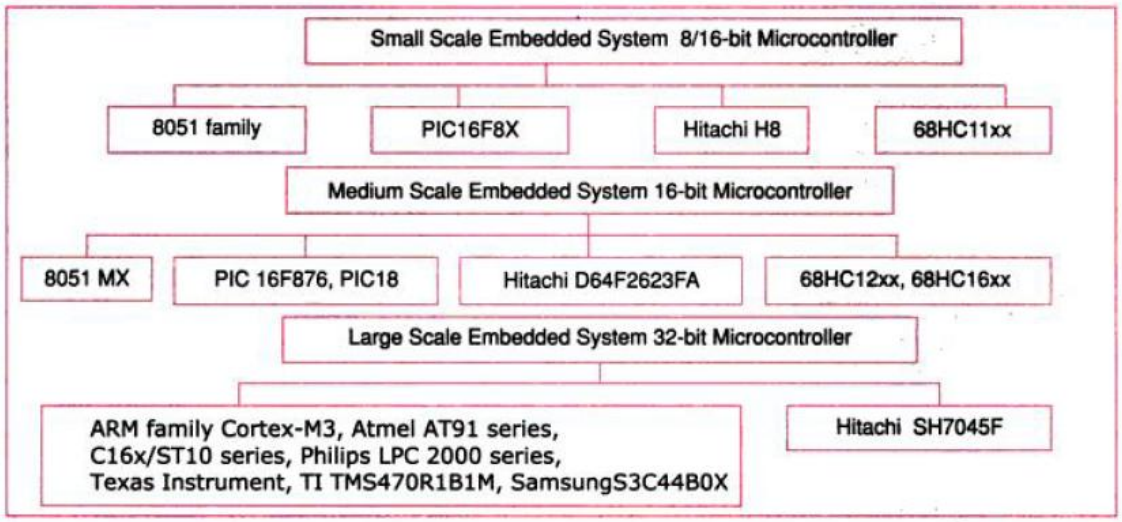
A few of the latest microcontrollers also have dual core and high computational and superscalar processing capabilities. Important microcontroller chips for embedded systems are 8051, 8051MX, 68HC11xx, HC12xx, HC16xx, PIC 16F84 or 16C76, 16F876 and PIC18, microcontroller enhancements of ARM9/ARM7 from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



Single Purpose Processors

Single purpose processors used in embedded systems include:

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



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 × vertical y-axis) pixels. Video frames have 525 × 625 pixels. The video graphic adapter (VGA) format of e-mailing and web pages has 640 × 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 × 1728 = 3.2 M; 2048 × 1536 = 3 M and 1280 × 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
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,
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.

Single-purpose processors – hardware:

A *single-purpose* processor is a digital circuit designed to execute exactly one program. For example, consider the digital camera example of Figure 1.1. All of the components other than the microcontroller are single-purpose processors. The JPEG codec, for example, executes a single program that compresses and decompresses video frames. An

embedded system designer creates a single-purpose processor by designing a custom digital circuit, as discussed in later chapters. Many people refer to this portion of the implementation simply as the "hardware" portion (although even software requires a hardware processor on which to run). Other common terms include coprocessor and accelerator.

Using a single-purpose processor in an embedded system results in several design metric benefits and drawbacks, which are essentially the inverse of those for general-purpose processors. Performance may be fast, size and power may be small, and unit-cost may be low for large quantities, while design time and NRE costs may be high, flexibility is low, unit cost may be high for small quantities, and performance may not match general-purpose processors for some applications. For example, Figure 1.5(d) illustrates the use of a single-purpose processor in our embedded system example, representing an exact fit of the desired functionality, nothing more, nothing less. Figure 1.6(c) illustrates the architecture of such a single-purpose processor for the example. Since the example counts from one to N, we add an *index* register. The index register will be loaded with N, and will then count down to zero, at which time it will assert a status line read by the controller. Since the example has only one other value, we add only one register labeled *total* to the data path. Since the example's only arithmetic operation is addition, we add a single adder to the data path. Since the processor only executes this one program, we hardwire the program directly into the control logic.

Application-specific processors:

Application Specific Integrated Circuit (ASIC) is a microchip designed to perform a specific or unique application. It is used as replacement to conventional general purpose logic chips. It integrates several functions into a single chip and there by reduces the system development cost. Most of the ASICs are proprietary products. As a single chip, ASIC consumes a very small area in the total system and thereby helps in the design of smaller systems with high capabilities/functionalities.

ASICs can be pre-fabricated for a special application or it can be custom fabricated by using the components from a re-usable '*building block*' library of components for a particular customer application. ASIC based systems are profitable only for large volume commercial productions. Fabrication of ASICs requires a non-refundable initial investment for the process technology and configuration expenses. This investment is known as Non-Recurring Engineering Charge (NRE) and it is a one time investment.

If the Non-Recurring Engineering Charges (NRE) is borne by a third party and the Application Specific Integrated Circuit (ASIC) is made openly available in the market, the ASIC is referred as Application Specific Standard Product (ASSP). The ASSP is marketed to multiple customers just as a general-purpose product is, but to a smaller number of customers since it is for a specific application. "The ADE7760 Energy Metre ASIC developed by Analog Devices for Energy metreing applications is a typical example for ASSP".

Advantages of PLD Programmable logic devices offer a number of important advantages over fixed logic devices, including:

- PLDs offer customers much more flexibility during the design cycle because design iterations are simply a matter of changing the programming file, and the results of design changes can be seen immediately in working parts.
- PLDs do not require long lead times for prototypes or production parts–the PLDs are already on a distributor's shelf and ready for shipment.
- PLDs do not require customers to pay for large NRE costs and purchase expensive mask sets–PLD suppliers incur those costs when they design their programmable devices and are able to amortize those costs over the multi-year lifespan of a given line of PLDs.
- PLDs allow customers to order just the number of parts they need, when they need them, allowing them to control inventory. Customers who use fixed logic devices often end up with excess inventory which must be scrapped, or if demand for their product surges, they may be caught short of parts and face production delays.
- PLDs can be reprogrammed even after a piece of equipment is shipped to a customer. In fact, thanks to programmable logic devices, a number of equipment manufacturers now tout the ability to add new features or upgrade products that already are in the field. To do this, they simply upload a new programming file to the PLD, via the Internet, creating new hardware logic in the system.

Commercial Off-the-Shelf Components (COTS)

A Commercial Off-the-Shelf (**COTS**) product is one which is used '*as-is*'. COTS products are designed in such a way to provide easy integration and interoperability with existing system components. The COTS component itself may be developed around a general purpose or domain specific processor or an Application Specific Integrated circuit or a programmable logic device. Typical examples of COTS hardware unit are remote controlled toy car control units including the RF circuitry part, high performance, high frequency microwave electronics (2–200 GHz), high bandwidth analog-to-digital converters, devices and components for operation at very high temperatures, electro-optic IR imaging arrays, UV/IR detectors, etc. The major advantage of using COTS is that they are readily available in the market, are cheap and a developer can cut down his/her development time to a great extent. This in turn reduces the time to market your embedded systems.

EMBEDDED HARDWARE UNITS:

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 V ± 0.25 V, 3.3 V ± 0.3 V, 2.0 V ± 0.2 V and 1.5 V ± 0.2 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, ~ $(2\ V/5\ V)^2$. 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.

Clock Oscillator Circuits:

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.

Real Time Clock (RTC):

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.

Reset Circuit:

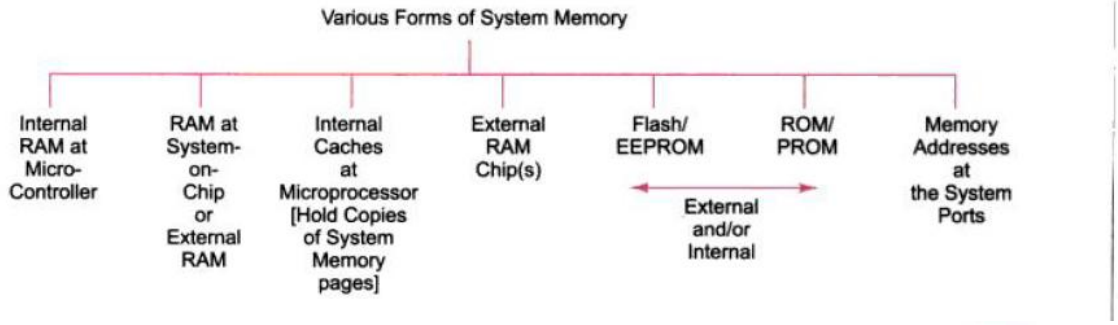
Reset means that the processor begins the processing of instructions from a starting address. That address is one that is set by default in the processor PC (or IP and CS in x86 processors) on a power-up. From that address in memory, program-instructions are fetched following the reset of the processor. 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.

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.

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

Memory:

1. Internal RAM of 256 or 512 bytes in a microcontroller for registers, temporary data and stack.
2. Internal ROM/PROM / E^2PROM 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).



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

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.

Certain sources are not maskable and cannot be disabled. Some are assigned the highest priority during processing.

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.

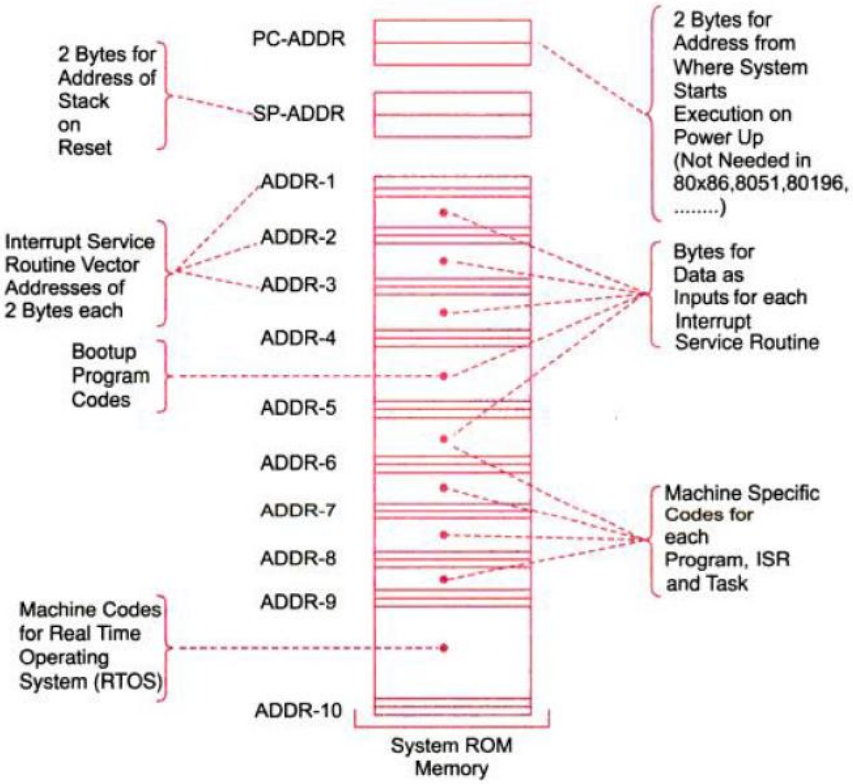
There is a programmable unit on-chip for the interrupt handling mechanism in a microcontroller.

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.

EMBEDDED SOFTWARE UNITS:

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.



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.

Software in Processor Specific Assembly Language

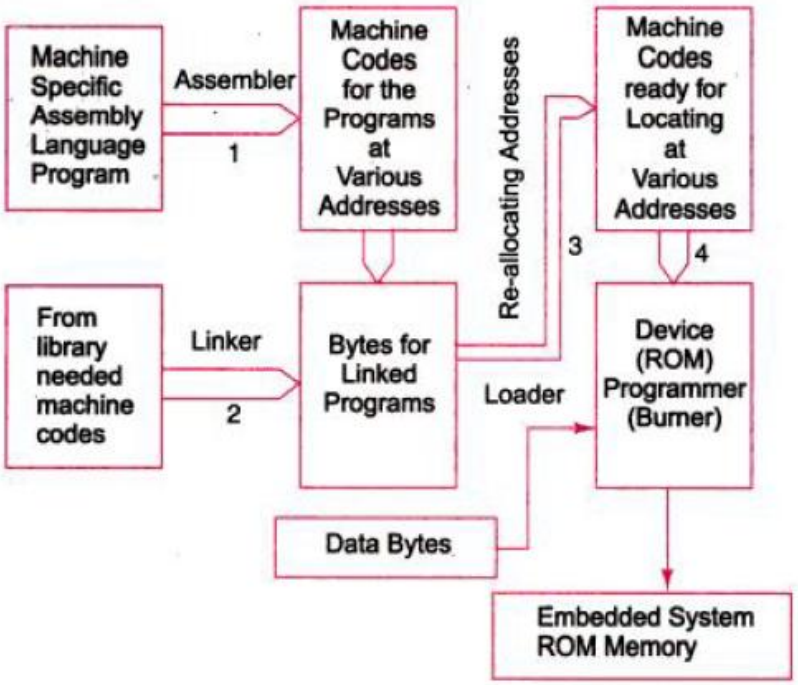
A program or a small specific part can be coded in *assembly language* using an assembler after understanding the processor and its instruction set. Assembler is software used for developing codes in *assembly*.

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.

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.

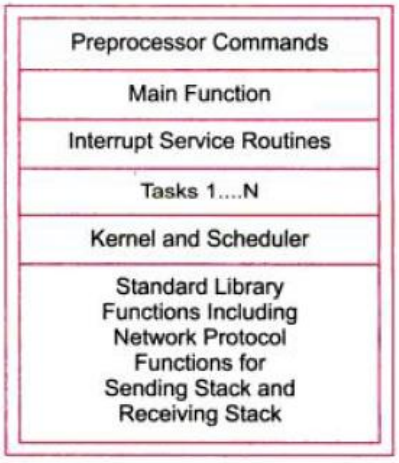


3. In the next step, the *loader* program performs the task of *reallocating* the codes after finding the physical memory addresses available at a given instant. The loader is a part of the operating system and places codes into the memory after reading the '.exe' file. This step is necessary because 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.

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.

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.



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.

UNIT II

ARM Programming - Instruction Set - Data Processing Instructions Addressing Modes - Branch, Load, Store Instructions - PSR Instructions - Conditional Instructions.

Data Processing Instructions:

The data processing instructions manipulate data within registers. They are move instructions, arithmetic instructions, logical instructions, comparison instructions, and multiply instructions. Most data processing instructions can process one of their operands using the barrel shifter.

A data processing instruction, then it updates the flags in the cpsr. Move and logical operations update the carry flag *C*, negative flag *N*, and zero flag *Z*. The carry flag is set from the result of the barrel shift as the last bit shifted out.

Move Instructions:

Move is the simplest ARM instruction. It copies *N* into a destination register *Rd*, where *N* is a register or immediate value. This instruction is useful for setting initial values and transferring data between registers.

Syntax: <instruction>{<cond>}{S} Rd, N

MO	Move a 32-bit value into a register	$Rd = N$
MV	move the NOT of the 32-bit value into a	$Rd = \sim N$

Example:

The MOV instruction takes the contents of register *r5* and copies them into register *r7*, in this case, taking the value 5, and overwriting the value 8 in register *r7*.

```
PRE     r5 = 5
        r7 = 8
        MOV r7, r5      ; let r7 = r5
POST    r5 = 5
        r7 = 5
```

Barrel Shifter:

Data processing instructions are processed within the arithmetic logic unit (ALU).

- A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU.
- This shift increases the power and flexibility of many data processing operations.

Arithmetic Instructions:

The arithmetic instructions implement addition and subtraction of 32-bit signed and unsigned values.

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-	$Rd = N - Rn - !$(carry
SBC	subtract with carry of two 32-bit	$Rd = Rn - N - !$(carry
SUB	subtract two 32-bit values	$Rd = Rn - N$

Example:

Simple subtract instruction subtracts a value stored in register *r2* from a value stored in register *r1*. The result is stored in register *r0*.

PRE r0 = 0x00000000
r1 = 0x00000002 r2 =
0x00000001

SUB r0, r1, r2

POST r0 = 0x00000001

Using the Barrel Shifter with Arithmetic Instructions:

The wide range of second operand shifts available on arithmetic and logical instructions is a very powerful feature of the ARM instruction set.

Example:

Register *r1* is first shifted one location to the left to give the value of twice *r1*. The ADD instruction then adds the result of the barrel shift operation to register *r1*. The final result transferred into register *r0* is equal to three times the value stored in register *r1*.

PRE r0 = 0x00000000
r1 = 0x00000005

ADD r0, r1, r1, LSL #1

POST r0 = **0x0000000f**
r1 = 0x00000005

Logical Instructions:

Logical instructions perform bitwise logical operations on the two source registers.

Syntax: <instruction>{<cond>}{S} Rd, Rn, N

AND	logical bitwise AND of two 32-bit	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \&$

Example:

A logical OR operation between registers *r1* and *r2*. *r0 holds* the result.

PRE r0 = 0x00000000
r1 = 0x02040608
r2 = 0x10305070
ORR r0, r1, r2

POST r0 = **0x12345678**

Comparison Instructions:

The comparison instructions are used to compare or test a register with a 32-bit value. To update the *cpsr* flag bits according to the result, but do not affect other registers. After the bits have been set, the information can then be used to change program flow by using conditional execution.

Syntax: <instruction>{<cond>} Rn, N

CMN	compare negated	flags set as a result of $Rn +$
CMP	compare	flags set as a result of $Rn -$
TEQ	test for equality of two 32-bit	flags set as a result of $Rn \wedge$
TST	test bits of a 32-bit value	flags set as a result of $Rn \&$

Example:

A CMP comparison instruction both registers, *r0* and *r9*, are equal before executing the instruction. The value of the *z* flag prior to execution is 0 and is represented by a lowercase *z*. After execution the *z* flag changes to 1 or an uppercase *Z*.

PRE cpsr = nzcvqiFt_USER r0 = 4
r9 = 4

CMP r0, r9

POST cpsr = nZcvqiFt_USER

The CMP is effectively a subtract instruction with the result discarded; similarly the TST instruction is a logical AND operation, and TEQ is a logical exclusive OR operation. It is important to understand that comparison instructions only modify the condition flags of the *cpsr* and do not affect the registers being compared.

Multiply Instructions:

The multiply instructions multiply the contents of a pair of registers and, depending upon the instruction, accumulate the results in with another register. The long multiplies accumulate onto a pair of registers representing a 64-bit value. The final result is placed in a destination register or a pair of registers.

Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn MUL{<cond>}{S} Rd, Rm,

ML	multiply and	$Rd=(Rm*Rs)+$
MU	multiply	$Rd=Rm*Rs$

The number of cycles taken to execute a multiply instruction depends on the processor implementation.

Example:

A simple multiply instruction that multiplies registers *r1* and *r2* together and places the result into register *r0*. In this example, register *r1* is equal to the value 2, and *r2* is equal to 2. The result, 4, is then placed into register *r0*.

```
PRE   r0          =
      0x00000000
      r1          =
      MUL r0, r1, r2   ; r0 =

POST  r0          =
      0x00000004
      r1          =
      0x00000002
      r2          =
      0x00000002
```

Branch Instructions:

A branch instruction changes the flow of execution or is used to call a routine. The change of execution flow forces the program counter *pc* to point to a new address. The ARMv5E instruction set includes four different branch instructions.

Syntax: B{<cond>} label
BL{<cond>} label
BX{<cond>} Rm
BLX{<cond>} label |
Rm

B	branch	*pc =label*
BL	branch with link	*pc =label* *lr* =address of the next instruction after the BL
BX	branch exchange	*pc=Rm* & 0xfffffffe, *T=Rm* & 1
BLX	branch exchange with link	*pc =label, T* =1 *pc=Rm* & 0xfffffffe, *T=Rm* & 1 *lr* =address of the next instruction after the instructions.

- The address *label* is stored in the instruction as a signed *pc*-relative offset and must be within approximately 32 MB of the branch instruction. *T* refers to the Thumb bit in the *cpsr*. When instructions set *T*, the ARM switches to Thumb state.

Example:

A simple fragment of code that branches to a subroutine using the BL instruction. To return from a subroutine, copy the link register to the *pc*.

```
        BL    subroutine    ; branch to
        CMP   r1, #5        ; compare r1 with 5
        MO    r1, #0        ; if (r1==5) then r1 =
        MO                  5
        :
subroutine
        <subroutine code>
        MOV   pc, lr        ; return by moving pc = lr
```

The branch exchange (BX) and branch exchange with link (BLX) are the third type of branch instruction. The BX instruction uses an absolute address stored in register *Rm*.

Load-Store Instructions:

Load-store instructions transfer data between memory and processor registers. There are three types of load-store instructions: single-register transfer, multiple-register transfer, and swap.

Single-Register Transfer:

These instructions are used for moving a single data item in and out of a register. The data types supported are signed and unsigned words (32-bit), half words (16-bit), and bytes.

Syntax: <LDR|STR>{<cond>}{B}
Rd,addressing1
LDR{<cond>}SB|H|SH Rd,
addressing2 STR{<cond>}H Rd,
addressing2

LD	load word into a register	*Rd* <-
ST	save byte or word from a	*Rd* ->
LD	load byte into a register	*Rd* <-
ST	save byte from a register	*Rd* ->

LDR	load halfword into a register	*Rd* <-
STR	save halfword into a register	*Rd* ->
LDR SB	load signed byte into a register	*Rd* <- *SignExtend*
LDR SH	load signed halfword into a register	*Rd* <- *SignExtend*

Example:

LDR and STR instructions can load and store data on a boundary alignment that is the same as the datatype size being loaded or stored. For example, LDR can only load 32-bit words on a memory address that is a multiple of four bytes—0, 4, 8, and so on. Load from a memory address contained in register *r1*, followed by a store back to the same address in memory.

```
load register r0 with the contents of
the memory address pointed to by register

 r1.
LDR r0, [r1]   ; = LDR r0, [r1, #0]
store the contents of register r0 to
the memory address pointed to by
register r1.
STR  r0, [r1]    ; = STR r0, [r1, #0]
```

- The first instruction loads a word from the address stored in register *r1* and places it into register *r0*. The second instruction goes the other way by storing the contents of register *r0* to the address contained in register *r1*. The offset from register *r1* is zero. Register *r1* is called the *base address register*.

Single-Register Load-Store Addressing Modes:

The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the indexing methods: preindex with write back, preindex, and post index.

Single-register load-store addressing, word or unsigned byte.

Addressing[1] mode and index method	Addressing[1] syntax
Preindex with immediate offset	[Rn, #+/-offset_12]
Preindex with register offset	[Rn, +/-Rm]
Preindex with scaled register offset	[Rn, +/-Rm, shift #shift_imm]
Preindex writeback with immediate offset	[Rn, #+/-offset_12]!
Preindex writeback with register offset	[Rn, +/-Rm]!
Preindex writeback with scaled register offset	[Rn, +/-Rm, shift #shift_imm]!
Immediate postindexed	[Rn], #+/-offset_12
Register postindex	[Rn], +/-Rm
Scaled register postindex	[Rn], +/-Rm, shift #shift_imm

- A signed offset or register is denoted by "+/−", identifying that it is either a positive or negative offset from the base address register *Rn*. The base address register is a pointer to a byte in memory, and the offset specifies a number of bytes.
- *Immediate* means the address is calculated using the base address register and a 12-bit offset encoded in the instruction. *Register* means the address is calculated using the base address register and a specific register's contents. *Scaled* means the address is calculated using the base address register and a barrel shift operation.

Multiple-Register Transfer:

- Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction.
- The transfer occurs from a base address register *Rn* pointing into memory.
- Multiple-register transfer instructions are more efficient from single-register transfers for moving blocks of data around memory and saving and restoring context and stacks.
- Load-store multiple instructions can increase interrupts latency.
- An ARM7 a load multiple instruction takes $2 + Nt$ cycles, where N is the number of registers to load and t is the number of cycles required for each sequential access to memory.
- If an interrupt has been raised, then it has no effect until the load-store multiple instruction is complete.

Syntax: <LDM|STM>{<cond>}<addressing mode> Rn{!},<registers>{^}

LD	load multiple	{Rd}*N <- mem32[start address + 4*N] optional
ST	save multiple	{Rd}*N -> mem32[start address + 4*N] optional

Any subset of the current bank of registers can be transferred to memory or fetched from memory. The base register *Rn* determines the source or destination address for a load- store multiple instruction.

Stack Operations:

- The ARM architecture uses the load-store multiple instructions to carry out stack operations.
- The *pop* operation (removing data from a stack) uses a load multiple instruction; similarly, the *push* operation (placing data onto the stack) uses a store multiple instruction.
- A stack is either *ascending* (A) or *descending* (D). Ascending stacks grow towards higher memory addresses; in contrast, descending stacks grow towards lower memory addresses.
- A *full stack* (F), the stack pointer *sp* points to an address that is the last used or full location (i.e., *sp* points to the last item on the stack).
- An *empty stack* (E) the *sp* points to an address that is the first unused or empty location (i.e., it points after the last item on the stack).

A numbers of load-store multiple addressing mode aliases available to support stack operations:

ARM has specified an ARM-Thumb Procedure Call Standard (ATPCS) that defines how routines are called and how registers are allocated. In the ATPCS, stacks are defined as being full descending stacks. LDMFD and STMFD instructions provide the pop and push functions, respectively.

Addressing methods for stack operations.

Addressing mode	Description	Pop	=	Push	=
FA	full ascending	LDMF	LDM	STMF	STM
FD	full descending	LDMF	LDMI	STMF	STM
EA	empty ascending	LDM	LDM	STME	STM
ED	empty descending	LDM	LDMI	STME	STM

Example:

A push operation on an empty stack using the STMED instruction. The STMED instruction pushes the registers onto the stack but updates register *sp* to point to the next empty location.

PRE r1 = 0x00000002
r4 = 0x00000003 sp = 0x00080010
STMED sp!, {r1,r4}

POST r1 = 0x00000002
r4 = 0x00000003
sp = 0x00080008

When handling a checked stack there are three attributes that need to be preserved:

- *Stack base*
- *Stack pointer*
- *Stack limit.*

a) The stack base is the starting address of the stack in memory.
b) The stack pointer initially points to the stack base; as data is pushed onto the stack
c) The stack pointer descends memory and continuously points to the top of stack.

Swap Instruction:

The swap instruction is a special case of a load-store instruction. It swaps the contents of memory with the contents of a register. This instruction is an *atomic operation* it reads and writes a location in the same bus operation, preventing any other instruction from reading or writing to that location until it completes.

Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]

SWP	swap a word between memory and a register	*tmp = mem32[Rn]* *mem32[Rn]*
SWPB	swap a byte between memory and a register	*tmp = mem8[Rn]* *mem8[Rn] =*

The swap instruction loads a word from memory into register *r0* and overwrites the memory with register *r1*.

Example:

A simple data guard that can be used to protect data from being written by another task. The SWP instruction "holds the bus" until the transaction is complete.

```
 spin
MOV r1, =semaphore MOV       r2, #1
SWP  r3, r2, [r1] ; hold the bus until complete CMP   r3, #1
BEQ  spin
```

Software Interrupt Instruction:

A software interrupt instruction (SWI) causes a software interrupt exception, which provides a mechanism for applications to call operating system routines.

Syntax: SWI{<cond>} SWI_number

SWI	software interrupt	*lr_svc* = address of instruction following the SWI *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC*

- When the processor executes an SWI instruction, it sets the program counter *pc* to the offset 0x8 in the vector table.
- The instruction also forces the processor mode to *SVC*, which allows an operating system routine to be called in a privileged mode.
- Each SWI instruction has an associated SWI number, which is used to represent a particular function call or feature.

Example:

A simple example of an SWI call with SWI number 0x123456, used by ARM toolkits as a debugging SWI. The SWI instruction is executed in user mode.

```
PRE  cpsr = nzcVqift_USER pc =
       0x00008000
       lr = 0x003fffff; lr = r14 r0 = 0x12

     0x00008000        SWI        0x123456

POST cpsr = nzcVqIft_SVC spsr =
       nzcVqift_USER pc =
       0x00000008
       lr = 0x00008004
       r0 = 0x12
```

Program Status Register Instructions:

- The ARM instruction set provides two instructions to directly control a program status register (*psr*). The MRS instruction transfers the contents of either the *cpsr* or *spsr* into a register; in the reverse direction, the MSR instruction transfers the contents of a register into the *cpsr* or *spsr*.
- These instructions are used to read and write the *cpsr* and *spsr*.
- In the syntax a label called *fields*. This can be any combination of control (*c*), extension (*x*), status (*s*), and flags (*f*). These fields relate to particular byte regions in a *psr*,

Syntax: MRS{<cond>} Rd,<cpsr|spsr>

MSR{<cond>} <cpsr|spsr>_<fields>,Rm
MSR{<cond>} <cpsr|spsr>_<fields>,#immediate

Conditional Execution:

- ARM instructions are *conditionally executed and* the instruction only executes if the condition code flags pass a given condition or test.
- The condition field is a two-letter mnemonic appended to the instruction mnemonic.
- The default mnemonic is AL, or *always execute*.
- Conditional execution reduces the number of branches, which also reduces the number of pipeline flushes and thus improves the performance of the executed code.
- Conditional execution depends upon two components: the condition field and condition flags.

Example::

An ADD instruction with the EQ condition appended. This instruction will only be executed when the zero flag in the *cpsr* is set to 1.

```
; r0 = r1 + r2 if zero flag is set ADDEQ r0, r1, r2
```

Only comparison instructions and data processing instructions with the S suffix appended to the mnemonic update the condition flags in the *cpsr*.

UNIT – III

Thumb Instruction Set - Register Usage - Other Branch Instructions - Data Processing Instructions - Single-Register and Multi Register Load-Store Instructions - Stack - Software Interrupt Instructions

Thumb Instruction Set:

Thumb encodes a subset of the 32-bit ARM instructions into a 16-bit instruction set space. Since Thumb has higher performance than ARM on a processor with a 16-bit data bus, but lower performance than ARM on a 32-bit data bus, use Thumb for memory-constrained systems.

Thumb has higher *code density*—the space taken up in memory by an executable program than ARM. The Thumb implementation uses more instructions; the overall memory footprint is reduced.

Mnemonic	THUMB	Description
ADC	v1	add two 32-bit values and carry
ADD	v1	add two 32-bit values
AND	v1	logical bitwise AND of two 32-bit values
ASR	v1	arithmetic shift right
B	v1	branch relative
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v2	breakpoint instructions
BL	v1	relative branch with link
BLX	v2	branch with link and exchange
BX	v1	branch with exchange
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit integers
EOR	v1	logical exclusive OR of two 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM
LDR	v1	load a single value from a virtual address in memory
LSL	v1	logical shift left
LSR	v1	logical shift right
MOV	v1	move a 32-bit value into a register
MUL	v1	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register
NEG	v1	negate a 32-bit value
ORR	v1	logical bitwise OR of two 32-bit values
POP	v1	pops multiple registers from the stack
PUSH	v1	pushes multiple registers to the stack
ROR	v1	rotate right a 32-bit value
SBC	v1	subtract with carry a 32-bit value
STM	v1	store multiple 32-bit registers to memory
STR	v1	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
TST	v1	test bits of a 32-bit value

Thumb Register Usage:

- In Thumb state, do not have direct access to all registers. Only the low registers *r0* to *r7* are fully accessible, is shown in Table. The higher registers *r8* to *r12* are only accessible with MOV, ADD, or CMP instructions.
- CMP and all the data processing instructions that operate on low registers update the condition flags in the *cpsr*.

Thumb register usage:

Registers	Access
r0–r7	fully accessible
r8–r12	only accessible by MOV, ADD, and CMP
r13 sp	limited accessibility
r14 lr	limited accessibility
r15 pc	limited accessibility
cpsr	only indirect access
spsr	no access

- Thumb instruction set list and from the Thumb register usage table that there is no direct access to the *cpsr* or *spsr*. In other words, there are no MSR- and MRS-equivalent Thumb instructions.
- To alter the *cpsr* or *spsr*, you must switch into ARM state to use MSR and MRS.
- There are no coprocessor instructions in Thumb state.
- In ARM state to access the coprocessor for configuring cache and memory management.

ARM-Thumb Interworking:

- *ARM-Thumb interworking* is the name given to the method of linking ARM and Thumb code together for both assembly and C/C++.
- It handles the transition between the two states. Extra code, called a *veneer*, is sometimes needed to carry out the transition. ATPCS defines the ARM and Thumb procedure call standards.
- To call a Thumb routine from an ARM routine, the core has to change state. This state change in the *T* bit of the *cpsr*.
- The BX and BLX branch instructions cause a switch between ARM and Thumb state while branching to a routine.
- The BX*lr* instruction returns from a routine, also with a state switch if necessary.

There are two versions of the BX or BLX instructions: an ARM instruction and a Thumb

equivalent. The ARM BX instruction enters Thumb state only if bit 0 of the address in *Rn* is set to binary 1; otherwise it enters ARM state. The Thumb BX instruction does the same.

Syntax: BX Rm
BLX Rm | label

BX	Thumb version branch exchange	*pc* = *Rn* & 0xfffffffe *T* = *Rn*[0]
BLX	Thumb version of the branch exchange with link	*lr* = (instruction address after the BLX)+1 *pc* = *label*, *T* = 0 *pc* = *Rm* & 0xfffffffe, *T* = *Rm*[0]

Example:

Replacing the BX instruction with BLX simplifies the calling of a Thumb routine since it sets the return address in the link register *lr*:

```
CODE32
LDR   r0, =thumbRoutine+1      ; enter Thumb state BLX
      r0                       ; jump to Thumb code
; continue here
```

Other Branch Instructions:

- There are two variations of the standard branch instruction, or B.
- The first is similar to the ARM version and is conditionally executed; the branch range is limited to a signed 8-bit immediate, or −256 to +254 bytes.
- The second version removes the conditional part of the instruction and expands the effective branch range to a signed 11-bit immediate, or −2048 to +2046 bytes.

The conditional branch instruction is the only conditionally executed instruction in Thumb state.

Syntax: B<cond>
label B label
BL label

B	branch	*pc* = *label*
BL	branch with link	*pc* = *label* *lr* = (instruction address after the BL)+1

- The BL instruction is not conditionally executed and has an approximate range of +/−4 MB.

- This range is possible because BL (and BLX) instructions are translated into a pair of 16-bit Thumb instructions.
- The first instruction in the pair holds the high part of the branch offset, and the second the low part. These instructions must be used as a pair.

The various instructions used to return from a BL subroutine call:

```
MOV   pc, lr
-----------------
BX    lr
-----------------
POP   {pc}
```

Data Processing Instructions:

- The data processing instructions manipulate data within registers.
- They include move instructions, arithmetic instructions, shifts, logical instructions, comparison instructions, and multiply instructions.
- The Thumb data processing instructions are a subset of the ARM data processing instructions.

Syntax:

<ADC|ADD|AND|BIC|EOR|MOV|MUL|MVN|NEG|ORR|SBC|SUB> Rd, Rm

ADC	add two 32-bit values and	*Rd = Rd + Rm + C* flag
ADD	add two 32-bit values	*Rd = Rn + immediate* *Rd = Rd + immediate* *Rd = Rd + Rm* *Rd = Rd + Rm* *Rd = (pc & 0xfffffffc) + (immediate « 2)*
AND	logical bitwise AND of two 32-bit values	*Rd= Rd & Rm*
ASR	arithmetic shift right	*Rd= Rm » immediate,* *C flag = Rm[immediate − 1]* *Rd= Rd » Rs,* *C flag =*

BIC	logical bit clear (AND NOT) of two 32-bit values	*Rd= Rd AND NOT(Rm)*
CMN	compare negative two 32-bit values	*Rn + Rm sets*
CMP	compare two 32-bit integers	*Rn – immediate sets flags Rn − Rm*
EOR	logical exclusive OR of two 32-bit values	*Rd= Rd EOR Rm*
LSL	logical shift left	*Rd= Rm « immediate,* *C flag = Rm[32 – immediate]* *Rd= Rd « Rs,* *C flag =*
LSR	logical shift right	*Rd= Rm » immediate,* *C flag = Rd[immediate − 1]* *Rd= Rd » Rs,* *C flag =*
MOV	move a 32-bit value into a register	*Rd = immediate* *Rd= Rn*
MUL	multiply two 32-bit values	*Rd = (Rm * Rd)[31:0]*
MVN	move the logical NOT of a 32-bit value into a	*Rd = NOT(Rm)*
NEG	negate a 32-bit value	*Rd = 0 − Rm*
ORR	logical bitwise OR of two 32-bit values	*Rd= Rd OR Rm*
ROR	rotate right a 32-bit value	*Rd=Rd RIGHT_ROTATE Rs,* *C flag = Rd[Rs−1]*

SBC	subtract with carry a 32-bit value	*Rd= Rd − Rm − NOT(C flag)*
SUB	subtract two 32-bit values	*Rd= Rn − immediate* *Rd= Rd − immediate* *Rd= Rn − Rm* *sp= sp −*
TST	test bits of a 32-bit value	*Rn AND Rm sets*

Thumb data processing instructions operate on low registers and update the *cpsr*. The exceptions are

```
MOV Rd,Rn
ADD  Rd,Rm
CMP  Rn,Rm
ADD  sp, #immediate
SUB  sp, #immediate
ADD  Rd,sp,#immediate
ADD  Rd,pc,#immediate
```

which can operate on the higher registers *r8–r14* and the *pc*. These instructions, except for CMP, do not update the condition flags in the *cpsr* when using the higher registers. The CMP instruction, however, always updates the *cpsr*.

Example:

A simple Thumb ADD instruction. It takes two low registers *r1* and *r2* and adds them together. The result is then placed into register *r0*, overwriting the original contents. The *cpsr* is also updated.

```
PRE  cpsr = nzcvIFT_SVC r1 =
     0x80000000
     r2 = 0x10000000
     ADD  r0, r1, r2

POST     r0 = 0x90000000
cpsr = NzcvIFT_SVC
```

Single-Register Load-Store Instructions:

The Thumb instruction set supports load and storing registers, or LDR and STR. These instructions use two preindexed addressing modes: offset by register and offset by immediate.

```
Syntax: <LDR|STR>{<B|H>} Rd,
        [Rn,#immediate] LDR{<H|SB|SH>}
        Rd,[Rn,Rm]
```

STR{<B|H>} Rd,[Rn,Rm]
LDR Rd,[pc,#immediate]
<LDR|STR> Rd,[sp,#immediate]

LDR	load word into a register	*Rd <- mem32[address]*
STR	save word from a register	*Rd -> mem32[address]*
LDR	load byte into a register	*Rd <- mem8[address]*
STR	save byte from a register	*Rd -> mem8[address]*
LDR	load halfword into a register	*Rd <- mem16[address]*
STR	save halfword into a register	*Rd -> mem16[address]*
LDR	load signed byte into a register	*Rd <-*
LDR	load signed halfword into a	*Rd <-*

Different Addressing Modes:

Load/store register	[Rn, Rm]
Base register + offset	[Rn, #immediate]
Relative	[pc|sp, #immediate]

- The offset by register uses a base register *Rn* plus the register offset *Rm*.
- The second uses the same base register *Rn* plus a 5-bit immediate or a value dependent on the data size.
- The 5-bit offset encoded in the instruction is multiplied by one for byte accesses, two for 16-bit accesses, and four for 32-bit accesses.

Multiple-Register Load-Store Instructions:

The Thumb versions of the load-store multiple instructions are reduced forms of the ARM load-store multiple instructions. They only support the increment after (IA) addressing mode.

Syntax : <LDM|STM>IA Rn!, {low Register list}

LDMIA	load multiple registers	{Rd}^{*N} <- mem32[Rn + 4*N], Rn =Rn + 4*N
STMIA	save multiple registers	{Rd}^{*N} -> mem32[Rn + 4*N], Rn =Rn + 4*N

N is the number of registers in the list of registers. These instructions always update the base register *Rn* after execution. The base register and list of registers are limited to the low registers *r0* to *r7*.

Stack Instructions:

The Thumb stack operations are different from the equivalent ARM instructions because they use the more traditional POP and PUSH concept.

Syntax: POP {low_register_list{, pc}}
PUSH {low_register_list{, lr}}

POP	pop registers from the stacks	Rd*N <- mem32[sp+4*N], sp= sp+4*N
PUSH	push registers on to the stack	Rd*N -> mem32[sp+4*N], sp= sp−4*N

- The stack pointer is fixed as register *r13* in Thumb operations and *sp* is automatically updated. The list of registers is limited to the low registers *r0* to *r7*.

- The PUSH register list also can include the link register *lr*; similarly the POP register list can include the *pc*.

Example:

The POP and PUSH instructions. The subroutine Thumb Routine is called using a branch with link (BL) instruction.

```
Call subroutine BL   ThumbRo
        ThumbRou
              PUS   {r1, lr}      ; enter subroutine
              H
              MO    r0, #2
              V
              POP   {r1, pc}      ; return from
                                    subroutine
```

- The link register *lr* is pushed onto the stack with register *r1*. Register *r1* is popped off the stack, as well as the return address being loaded into the *pc*. This returns from the subroutine.

Software Interrupt Instruction:

- The Thumb software interrupt (SWI) instruction causes a software interrupt exception.
- If any interrupt or exception flag is raised in Thumb state, the processor automatically reverts back to ARM state to handle the exception.

Syntax: SWI immediate

SWI	software interrupt	*lr_svc* = address of instruction following the SWI *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC*

The Thumb SWI instruction has the same effect and nearly the same syntax as the ARM equivalent. It differs in that the SWI number is limited to the range 0 to 255 and it is not conditionally executed.

Example:

Execution of a Thumb SWI instruction. The processor goes from Thumb state to ARM state after execution.

```
PR    cpsr =
      pc = 0x00008000
      lr = 0x003fffff          ; lr = r14
      r0 = 0x12

0x00008000      SWI      0x45

POST  cpsr = nzcVqIft_SVC
      spsr =
      nzcVqifT_USER pc
      = 0x00000008
      lr = 0x00008002
      r0 = 0x12
```

UNIT – IV

ARM Programming using C:

The compiler translates C source to ARM assembler. The techniques apply equally to C++, an overview of C compilers and optimization write source code that will compile more efficiently in terms of increased speed and reduced code size.

4.1 Overview of C Compilers and Optimization

The C language and have some knowledge of assembly programming. Optimizing code takes time and reduces source code readability. C compilers have to translate C function literally into assembler so that it works for all possible

inputs. The memclr function clears N bytes of memory at address data.

```
void memclr(char *data, int N)
{
for (; N>0; N--)
{
*data=0; data++;
}
}
```

To write efficient C code, compiler has to be conservative, the limits of the processor architecture the C compiler is mapping to, and the limits of a specific C compiler.

To keep our examples concrete, we have tested them using the following specific C compilers:

- *armcc* from ARM Developer Suite version 1.1 (ADS1.1). You can license this compiler, or a later version, directly from ARM.
- *arm-elf-gcc* version 2.95.2. This is the ARM target for the GNU C compiler, *gcc*, and is freely available.

4.2 Basic C Data Types

ARM compilers handle the basic C data types. It is more efficient to use for local variables than others. There are also differences between the addressing modes available when loading and storing data of each type. ARM processors have 32-bit registers and 32-bit data processing operations. The ARM architecture is RISC load/store architecture. There is no arithmetic or logical instructions that manipulate values in memory directly.

Load and store instructions by ARM architecture.

Architecture	Instruction	Action
Pre-ARMv4	LDRB	load an unsigned 8-bit value
	STRB	store a signed or unsigned 8-bit value
	LDR	load a signed or unsigned 32-bit value
	STR	store a signed or unsigned 32-bit value
ARMv4	LDRSB	load a signed 8-bit value
	LDRH	load an unsigned 16-bit value
	LDRSH	load a signed 16-bit value
	STRH	store a signed or unsigned 16-bit value
ARMv5	LDRD	load a signed or unsigned 64-bit value
	STRD	store a signed or unsigned 64-bit value

These architectures were used on processors prior to the ARM7TDMI. 8- or 16-bit values extend the value to 32 bits before writing to an ARM register. Unsigned values are zero-extended, and signed values sign-extended. This means that the cast of a loaded value to an int type does not cost extra instructions. Similarly, a store of an 8- or 16-bit value selects the lowest 8 or 16 bits of the register. The cast of an int to smaller type does not cost extra instructions on a store. ARMv5 adds instruction support for 64-bit load and stores.

C Data Type	Implementation
char	unsigned 8-bit byte
short	signed 16-bit halfword
int	signed 32-bit word
long	signed 32-bit word
long long	signed 64-bit double word

4.3. Local Variable Types:

ARMv4-based processors can efficiently load and store 8-, 16-, and 32-bit data. However, most ARM data processing operations are 32-bit only. For this reason, you should use a 32-bit data type, int or long, for local variables wherever possible. Avoid using char and short as local variable types, even if you are manipulating an 8- or 16-bit value.

The one exception is when you want wrap-around to occur. If you require modulo arithmetic of the form 255 + 1 = 0, then use the char type. A checksum function that sums the values in a data packet. most communication protocols (such as TCP/IP) have a checksum or cyclic redundancy check (CRC) routine to check for errors in a data packet.

The following code checksums a data packet containing 64 words.

```
int checksum_v1(int *data)
{
  char i; int sum=
  0;

  for (i=0; i<64; i++)
  {
    sum += data[i];
  }
  return sum;
}
```

All ARM registers are 32-bit and all stack entries are at least 32-bit. Furthermore, to implement the i++ exactly, the compiler must account for the case when i = 255.

Function Argument Types:

Converting local variables from types char or short to type int increases performance and reduces code size. The same holds for function arguments. Consider the following simple function, which adds two 16-bit values, halving the second, and returns a 16-bit sum:

```
short add_v1(short a, short b)
{
return a + (b>>1);
}
```

The input values a, b, and the return value will be passed in 32-bit ARM registers. Should the compiler assume that these 32-bit values are in the range of a short type, that is, −32,768 to +32,767.

Function arguments are passed *wide* if they are not reduced to the range of the type and *narrow* the compiler has made by looking at the assembly output for add_v1. If the compiler passes arguments wide, then the callee must reduce function arguments to the correct range.

If the compiler passes arguments narrow, then the caller must reduce the range. If the compiler returns values wide, then the caller must reduce the return value to the correct range. If the compiler returns values narrow, then the callee must reduce the range before returning the value.

The *armcc* output for add_v1 shows that the compiler casts the return value to a short type, but does not cast the input values. It assumes that the caller has already ensured that the 32-bit values r0 and r1 are in the range of the short type. This shows narrow passing of arguments and return value.

Signed versus Unsigned Types:

This section compares the efficiencies of signed int and unsigned int. If code uses addition, subtraction, and multiplication,

then there is no performance difference between signed and unsigned operations. However, there is a difference when it comes to division. Consider the following short example that averages two integers:

```
int average_v1(int a, int b)
{
  return (a+b)/2;
}
```

This compiles to

```
average_v1
        ADD     r0,r0,r1            ; r0 = a + b
        ADD     r0,r0,r0,LSR #31    ; if (r0<0) r0++
        MOV     r0,r0,ASR #1        ; r0 = r0>>1
        MOV     pc,r14              ; return r0
```

The compiler adds one to the sum before shifting by right if the sum is negative. In other words it replaces $x/2$ by the statement:

```
(x<0) ? ((x+1)>>1): (x>>1)
```

It must do this because x is signed. In C on an ARM target, a divide by two is not a right shift if x is negative. For example, $-3 \gg 1 = -2$ but $-3/2 = -1$. Division rounds towards zero, but arithmetic right shift rounds towards $-\infty$.

It is more efficient to use unsigned types for divisions. The compiler converts unsigned power of two divisions directly to right shifts. For general divisions, the divide routine in the C library is faster for unsigned types.

Function Calls:

The ARM Procedure Call Standard (APCS) defines how to pass function arguments and return values in ARM registers. The more recent ARM-Thumb Procedure Call Standard (ATPCS) covers ARM and Thumb interworking as well.

The first four integer arguments are passed in the first four ARM registers: *r0, r1, r2*, and *r3*. Subsequent integer arguments are placed on the full descending stack, ascending in memory. Function return integer values are passed in *r0*.

This description covers only integer or pointer arguments. Two-word arguments such as long long or double are passed in a pair of consecutive argument registers and returned in *r0, r1*. The compiler may pass structures in registers or by reference according to command line compiler options.

The first point to note about the procedure call standard is the *four-register rule*. Functions with four or fewer arguments are far more efficient to call than functions with five or more arguments. For functions with four or fewer arguments, the compiler can pass all the arguments in registers. For functions with more arguments, both the caller and callee must access the stack for some arguments.

C function needs more than four arguments, or your C++ method more than three explicit arguments, then it is almost always more efficient to use structures. Group related arguments into structures, and pass a structure pointer rather than multiple arguments.

Example: The following code creates a Queue structure and passes this to the function to reduce the number of function arguments.

The queue_bytes_v2 is one instruction longer than queue_bytes_v1, but it is in fact more efficient overall. The second version has only three function arguments rather than five.

Each call to the function requires only three register setups. This compares with four register setups, a stack push,

and a stack pull for the first version. There is a net saving of two instructions in function call overhead. There are likely further savings in the callee function, as it only needs to assign a single register to the Queue structure pointer, rather than three registers in the nonstructured case.

There are other ways of reducing function call overhead if your function is very small and corrupts few registers (uses few local variables). Put the C function in the same C file as the functions that will call it. The C compiler then knows the code generated for the callee function and can make optimizations in the caller function:

- The caller function need not preserve registers that it can see the callee doesn't corrupt. Therefore the caller function need not save all the ATPCS corruptible registers.
- If the callee function is very small, then the compilers can inline the code in the caller function. This removes the function call overhead completely.

Example: The function uint_to_hex converts a 32-bit unsigned integer into an array of eight hexa- decimal digits. It uses a helper function nybble_to_hex, which converts a digit din the range 0 to 15 to a hexadecimal digit.

```
 uint_to_hex
        MOV   r3,#8              ; i = 8
uint_to_hex_loop
    MOV     r1,r1,ROR #28       ; in = (in<<4)|(in>>28)
    AND     r2,r1,#0xf          ; r2 = in & 15
    CMP     r2,#0xa             ; if (r2>=10)
    ADDCS   r2,r2,#0x37         ;    r2 +=' A' -10
    ADDCC   r2,r2,#0x30         ; else r2 +=' 0'
    STRB    r2,[r0],#1          ; *(out++) = r2
    SUBS    r3,r3,#1            ; i-- and set flags
    BNE     uint_to_hex_loop    ; if (i!=0) goto loop
    MOV     pc,r14              ; return
```

Pointers:

Two pointers are said to *alias* when they point to the same address. If write to one pointer, it will affect the value to read from the other pointer. In a function, the compiler often doesn't know which pointers can alias and which pointers can't. The compiler must be very pessimistic and assume that any write to a pointer may affect the value read from any other pointer, which can significantly reduce code efficiency.

A very simple example. The following function increments two timer values by a step amount:

```
void timers_v1(int *timer1, int *timer2, int *step)
{
  *timer1 += *step;
  *timer2 += *step;
}
```

This compiles to

```
timers_v1
        LDR     r3,[r0,#0]          ; r3 = *timer1
        LDR     r12,[r2,#0]         ; r12 = *step
        ADD     r3,r3,r12           ; r3 += r12
        STR     r3,[r0,#0]          ; *timer1 = r3
        LDR     r0,[r1,#0]          ; r0 = *timer2
        LDR     r2,[r2,#0]          ; r2 = *step
        ADD     r0,r0,r2            ; r0 += r2
```

```
STR     r0,[r1,#0]          ; *timer2 = t0
MOV     pc,r14              ; return
```

Structure:

A frequently used structure can have a significant impact on its performance and code density. There are two issues concerning structures on the ARM: alignment of the structure entries and the overall size of the structure.

For architectures up to and including ARMv5TE, load and store instructions are only guaranteed to load and store values with address aligned to the size of the access width.

ARM compilers will automatically align the start address of a structure to a multiple of the largest access width used within the structure (usually four or eight bytes) and align entries within structures to their access width by inserting padding.

For example, consider the structure

```
struct { char a;
  int b; char c;
  short d;
}
```

To improve the memory usage, you should reorder the elements

```
struct { char a;
  char c; short d;
  int b;
}
```

This reduces the structure size from 12 bytes to 8 bytes, with the following new layout:

The exact layout of a structure in memory may depend on the compiler vendor and a good idea to insert any padding that you cannot get rid of into the structure manually. This way the structure layout is not ambiguous. It is easier to link code between compiler versions and compiler vendors if you stick to unambiguous structures.

Another point of ambiguity is enum. Different compilers use different sizes for an enumerated type, depending on the range of the enumeration. For example, consider the type

```
typedef enum { FALSE,
    TRUE
} Bool;
```

The *armcc* in ADS1.1 will treat Bool as a one-byte type as it only uses the values 0 and 1. Bool will only take up 8 bits of space in a structure. However, *gcc* will treat Bool as a word and take up 32 bits of space in a structure. To avoid ambiguity it is best to avoid using enum types in structures used in the API to your code.

Another consideration is the size of the structure and the offsets of elements within the structure. This problem is most acute when you are compiling for the Thumb instruction set. Thumb instructions are only 16 bits wide and so only allow for small element offsets from a structure base pointer.

The compiler can only access an 8-bit structure element with a single instruction if it appears within the first 32 bytes of the structure. Similarly, single instructions can only access 16-bit values in the first 64 bytes and 32-bit values in the first 128 bytes. Once you exceed these limits, structure accesses become inefficient.

The following rules generate a structure with the elements packed for maximum efficiency:

- Place all 8-bit elements at the start of the structure.

- Place all 16-bit elements next, then 32-bit, then 64-bit.
- Place all arrays and larger elements at the end of the structure.
- If the structure is too big for a single instruction to access all the elements, then group the elements into substructures. The compiler can maintain pointers to the individual substructures.

Floating Point:

The majority of ARM processor implementations do not provide hardware floating-point support, which saves on power and area when using ARM in a price-sensitive, embedded application. With the exceptions of the Floating Point Accelerator (FPA) used on the ARM7500FE and the Vector Floating Point accelerator (VFP) hardware, the C compiler must provide support for floating point in software.

The C compiler converts every floating-point operation into a subroutine call. The C library contains subroutines to simulate floating-point behavior using integer arithmetic. This code is written in highly optimized assembly. Even so, floating-point algorithms will execute far more slowly than corresponding integer algorithms.

Inline Functions and Inline Assembly:

Using inline functions that contain assembly you can get the compiler to support ARM instructions and optimizations that aren't usually available. For the examples of this section we will use the inline assembler in *armcc*.

The inline assembler with the main assembler *armasm*. The inline assembler is part of the C compiler. The C compiler still performs register allocation, function entry, and exit. The compiler also attempts to optimize the inline assembly write, or deoptimize it for debug mode. Although the compiler output will be functionally equivalent to your inline assembly, it may not be identical.

The main benefit of inline functions and inline assembly is to make accessible in C operations that are not usually available as part of the C language. It is better to use inline functions rather than #define macros because the latter doesn't check the types of the function arguments and return value.

Example: To calculate a saturating correlation. In other words, we calculate $a = 2x0y0 + \cdots 2xN-1yN-1$ with saturation.

```
int sat_correlate(short *x, short *y, unsigned int N)
{
  int a=0;

  do
  {
    a = qmac(a, *(x++), *(y++));
  } while (--N); return a;
}
```

Portability Issues:

- *The* **char** *type*. On the ARM, char is unsigned rather than signed as for many other processors. A common problem concerns loops that use a char loop counter i and the continuation condition i > 0, they become infinite loop.
- *The* **int** *type*. When moving to ARM's 32-bit int type although this is rare nowadays. Note that expressions are promoted to an int type before evaluation. Therefore if i = -0x1000, the expression i==0xF000 is true on a 16-bit machine but false on a 32-bit machine.

- ***Unaligned data pointers***. Processors support the loading of shortand inttyped values from unaligned addresses. A C program may manipulate pointers directly so that they become unaligned.
- ***Endian assumptions***. C code may make assumptions about the endianness of a memory system, for example, by casting a char *to an int *. If you configure the ARM for the same endianness the code is expecting, then there is no issue.
- ***Function prototyping***. The *armcc* compiler passes arguments narrow, that is, reduced to the range of the argument type. If functions are not prototyped correctly, then the function may return the wrong answer. Other compilers that pass arguments wide may give the correct answer even if the function prototype is incorrect. Always use ANSI prototypes.
- ***Use of bit-fields***. The layout of bits within a bit-field is implementation and endian dependent. If C code assumes that bits are laid out in a certain order, then the code is not portable.
- ***Inline assembly.*** Using inline assembly in C code reduces portability between architectures. You should separate any inline assembly into small in lined functions that can easily be replaced. It is also useful to supply reference, plain C implementations of these functions that can be used on other architectures, where this is possible.

Writing and Optimizing ARM Assembly Code:

By optimizing these routines can reduce the system power consumption and reduce the clock speed needed for real-time operation. Optimization can turn an infeasible system into a feasible one, or an uncompetitive system into a competitive one.

Writing assembly code by direct control of three optimization tools that cannot explicitly use by writing C source:

- ***Instruction scheduling***: Reordering the instructions in a code sequence to avoid processor stalls. Since ARM implementations are pipelined, the timing of an instruction can be affected by neighboring instructions.
- ***Register allocation***: Deciding how variables should be allocated to ARM registers or stack locations for maximum performance. To minimize the number of memory accesses.
- ***Conditional execution***: Accessing the full range of ARM condition codes and conditional instructions.

Writing Assembly Code:

This section gives examples showing how to write basic assembly code and familiar with the ARM instructions covered. This chapter uses the ARM macro assembler *armasm.*

Example: To convert a C function to an assembly function.

Consider the simple C program main.cfollowing that prints the squares of the integers from 0 to 9:

```
#include <stdio.h>
int square(int i); int main(void)
{
Int i;
For (i=0; i<10; i++)
```

```
  {
printf("Square of %d is %d\n", i, square(i));
}
}
int square(int i)
{
Return i*i;
}
```

Instruction Scheduling:

Instructions that are conditional on the value of the ARM condition codes in the *cpsr* take one cycle if the condition is not met. If the condition is met, then the following rules apply:

- ALU operations such as addition, subtraction, and logical operations take one cycle. This includes a shift by an immediate value. If you use a register-specified shift, then add one cycle. If the instruction writes to the *pc*, then add two cycles.
- Load instructions that load *N* 32-bit words of memory such as LDR and LDM take *N* cycles to issue, but the result of the last word loaded is not available on the following cycle. The updated load address is available on the next cycle. This assumes zero-wait-state memory for an uncached system, or a cache hit for a cached system. An LDM of a single value is exceptional, taking two cycles. If the instruction loads *pc*, then add two cycles.
- Load instructions that load 16-bit or 8-bit data such as LDRB, LDRSB, LDRH, and LDRSH take one cycle to issue. The load result is not available on the following two cycles. The updated load address is available on the next cycle. This assumes zero-wait-state memory for an uncached system, or a cache hit for a cached system.
- Branch instructions take three cycles.
- Store instructions that store *N* values take *N* cycles. This assumes zero-wait-state memory for an uncached system, or a cache hit or a write buffer with *N* free entries for a cached system. An STM of a single value is exceptional, taking two cycles.
- Multiply instructions take a varying number of cycles depending on the value of the second operand in the product.

How to schedule code efficiently on the ARM, The ARM9TDMI processor performs five operations in parallel:

- ***Fetch***: Fetch from memory the instruction at address *pc*. The instruction is loaded into the core and then processes down the core pipeline.
- ***Decode***: Decode the instruction that was fetched in the previous cycle. The processor also reads the input operands from the register bank if they are not available via one of the forwarding paths.

ALU: Executes the instruction that was decoded in the previous cycle. Note this instruction was originally fetched from address *pc* − 8 (ARM state) or *pc* − 4 (Thumb state). Normally this involves calculating the answer for a data processing operation, or the address for a load, store, or branch operation.

- ***LS1***: Load or store the data specified by a load or store instruction. If the instruction is not a load or store, then this stage has no effect.
- ***LS2***: Extract and zero- or sign-extend the data loaded by a byte or half word load instruction. If the instruction is not a load of an 8-bit byte or 16-bit half word item, then this stage has no effect.

Five-stage ARM9TDMI pipeline. After an instruction has completed the five stages of the pipeline, the core writes the result to the register file. *pc* points to the address of the instruction being fetched. The ALU is executing the instruction that was originally fetched from address *pc* − 8 in parallel with fetching the instruction at address *pc*.

If an instruction requires the result of a previous instruction that is not available, then the processor stalls. This is called a pipeline *hazard* or pipeline *interlock*.

Why a branch instruction takes three cycles. The processor must flush the pipeline when jumping to a new address.

```
MOV    r1, #1
B      case1
AND    r0, r0, r1 EOR   r2, r2, r3
...

SUB    r0, r0, r1
```

The three executed instructions take a total of five cycles. The MOV instruction executes on the first cycle. On the second cycle, the branch instruction calculates the destination address.

This causes the core to flush the pipeline and refill it using this new *pc* value.

The refill takes two cycles. Finally, the SUB instruction executes normally. The pipeline drops the two instructions following the branch when the branch takes place.

Register Allocation:

We can use 14 of the 16 visible ARM registers to hold general-purpose data. The other two registers are the stack pointer *r13* and the program counter *r15*. For a function to be ATPCS compliant it must preserve the callee values of registers *r4* to *r11*. ATPCS also specifies that the stack should be eight-byte aligned; therefore you must preserve this alignment if calling subroutines. Use the following template for optimized assembly routines requiring many registers:

Allocating Variables to Register Numbers:

An assembly routine, it is best to start by using names for the variables, rather than explicit register numbers. This allows to change the allocation of variables to register numbers easily. Register names increase the clarity and readability of optimized code.

If swap all occurrences of two registers *Ra* and *Rb* in a routine, the function of the routine does not change. However, there are several cases where the physical number of the register is important:

- ***Argument registers***. The ATPCS convention defines that the first four arguments to a function are placed in registers *r0* to *r3*. Further arguments are placed on the stack. The return value must be placed in *r0*.
- ***Registers used in a load or store multiple***. Load and store multiple instructions LDM and STM operate on a list of registers in order of ascending register number. If *r0* and *r1* appear in the register list, then the processor will always load or store *r0* using a lower address than *r1* and so on.
- ***Load and store double word.*** The LDRD and STRD instructions introduced in ARMv5E operate on a pair of registers with sequential register numbers, *Rd* and *Rd* + 1. Furthermore, *Rd* must be an even register number.

For an example of how to allocate registers when writing assembly, suppose we want to shift an array of *N* bits upwards in memory by *k* bits. For simplicity assume that *N* is large and a multiple of 256.

Assume that $0 \leq k < 32$ and that the input and output pointers are word aligned. This type of operation is common in dealing with the arithmetic of multiple precision numbers where we want to multiply by 2^k. It is also useful to block copy from one bit or byte alignment to a different bit or byte alignment.

The C routine shift bits implements the simple *k*-bit shift of *N* bits of data. It returns the *k* bits remaining following the shift.

```
unsigned int shift_bits(unsigned int *out, unsigned int *in,
                        unsigned int N, unsigned int k)
{
  unsigned int carry=0, x;

  do
  {
    x = *in++;
    *out++ = (x<<k) | carry;
    carry = x>>(32-k); N
    -= 32;
  } while (N);

  return carry;
}
```

The obvious way to improve efficiency is to unroll the loop to process eight words of 256 bits at a time so that can use load and store multiple operations to load and store eight words at a time for maximum efficiency.

There are two remaining variables *carry* and *kr*, but only one remaining free register *lr*. There are several possible ways proceeding when we run out of registers:

- Reduce the number of registers we require by performing fewer operations in each loop. In this case we could load four words in each load multiple rather than eight.
- Use the stack to store the least-used values to free up more registers. In this case we could store the loop counter *N* on the stack. Alter the code implementation to free up more registers.

Making the Most of Available Registers:

On load-store architecture such as the ARM, it is more efficient to access values held in registers than values held in memory. There are several tricks you can use to fit several sub-32-bit length variables into a single 32-bit register and thus can reduce code size and increase performance.

Single issue Multiple Data (SIMD):

Arrays of 8-bit or 16-bit values, it is sometimes possible to manipulate multiple values at a time by packing several values into a single 32-bit register. This is called ***single issue multiple data*** (SIMD) processing.

Conditional Execution:

The processor core can conditionally execute most ARM instructions. This conditional assembler defaults to the execute always condition (AL). The other 14 conditions split into seven pairs of complements. The conditions depend on the four condition code flags *N*, *Z*, *C*, *V* stored in the *cpsr* register.

By default, ARM instructions do not update the *N*, *Z*, *C*, *V* flags in the ARM *cpsr*. For most instructions, to update these flags you append an S suffix to the instruction mnemonic.

Exceptions to this are comparison instructions that do not write to a destination register. Their sole purpose is to update the flags and so they don't require the S suffix.

By combining conditional execution and conditional setting of the flags, you can imple- ment simple if statements without any need for branches. This improves efficiency since branches can take many cycles and also reduces code size.

Example: The following C code converts an unsigned integer $0 \leq i \leq 15$ to a hexadecimal character c:

```
if (i<10)
{
   c = i + '0';
}
else
{
c = i +'A'-10;
}
```

Write in assembly using conditional execution rather than conditional branches:

```
CMP i, #10
ADDLO c,i,#'0'
ADDHS c,i,#'A'-10
```

The sequence works since the first ADD does not change the condition codes. The second ADD is still conditional on the result of the compare. Conditional execution is even more powerful for cascading conditions.

Example: Consider the following code that detects if c is a letter:

```
if((c>='A'&&c<='Z')||(c>='a'&&c<='z'))
{
letter++;
}
```

To implement this efficiently, we can use an addition or subtraction to move each range to the form $0 \leq c \leq limit$. Then we use unsigned comparisons to detect this range and conditional comparisons to chain together ranges. The following assembly implements this efficiently:

```
SUB temp, c, #'A'
CMP temp, #'Z'-'A'
SUBHI temp, c, #'a'
CMPHI temp, #'z'-'a'
ADDLS letter, letter,
#1
```

UNIT - V

Real Time Operating Systems: Brief History of OS - Defining RTOS – The Scheduler - Objects – Services - Characteristics of RTOS - Defining a Task - Tasks States and Scheduling - Task Operations – Structure – Synchronization - Communication and Concurrency. Defining Semaphores - Operations and Use - Defining Message Queue - States – Content – Storage - Operations and Use.

Introduction To Real-Time Operating Systems

Introduction

A real-time operating system (RTOS) is key to many embedded systems today and, provides a software platform upon which to build applications. Not all embedded systems, however, are designed with an RTOS. Some embedded systems with relatively simple hardware or a small amount of software application code might not require an RTOS. Many embedded systems, however, with moderate-to-large software applications require some form of scheduling, and these systems require an RTOS.

This chapter sets the stage for all subsequent chapters in this section. It describes the key concepts upon which most real-time operating systems are based. Specifically, this chapter provides

- A brief history of operating systems,
- A definition of an RTOS,
- A description of the scheduler,
- A discussion of objects,
- A discussion of services, and
- The key characteristics of an RTOS.

A Brief History of Operating Systems

- In the early days of computing, developers created software applications that included low-level machine code to initialize and interact with the system's hardware directly. This tight integration between the software and hardware resulted in non-portable applications. A small change in the hardware might result in rewriting much of the application itself. Obviously, these systems were difficult and costly to maintain.
- As the software industry progressed, operating systems that provided the basic software foundation for computing systems evolved and facilitated the abstraction of the underlying hardware from the application code. In addition, the evolution of operating systems helped shift the design of software applications from large, monolithic applications to more modular, interconnected applications that could run on top of the operating system environment.

Later in the decade, momentum started building for the next generation of computing: the post-PC, embedded-computing era. To meet the needs of embedded computing,

commercial RTOSes, such as VxWorks, were developed. Although some functional similarities exist between RTOSes and GPOSes, many important differences occur as well. These differences help explain why RTOSes are better suited for real-time embedded systems.

Some core functional similarities between a typical RTOS and GPOS include:

- Some level of multitasking,
- Software and hardware resource management,
- Provision of underlying OS services to applications, and
- Abstracting the hardware from the software application.

On the other hand, some key functional differences that set RTOSes apart from GPOSes include:

- Better reliability in embedded application contexts,
- The ability to scale up or down to meet application needs,
- Faster performance,
- Reduced memory requirements,
- Scheduling policies tailored for real-time embedded systems,
- Support for diskless embedded systems by allowing executables to boot and run from ROM or RAM, and
- Better portability to different hardware platforms.

RTOSes, on the other hand, can meet these requirements. They are reliable, compact, and scalable, and they perform well in real-time embedded systems. In addition, RTOSs can be easily tailored to use only those components required for a particular application.

Defining an RTOS

A real-time operating system (RTOS) is a program that schedules execution in a timely manner, manages system resources, and provides a consistent foundation for developing application code. Application code designed on an RTOS can be quite diverse, ranging from a simple application for a digital stopwatch to a much more complex application for aircraft navigation. Good RTOSes, therefore, are scalable in order to meet different sets of requirements for different applications.

For example, in some applications, an RTOS comprises only a kernel, which is the core supervisory software that provides minimal logic, scheduling, and resource-management algorithms. Every RTOS has a kernel. On the other hand, an RTOS can be a combination of

various modules, including the kernel, a file system, networking protocol stacks, and other components required for a particular application, as illustrated at a high level in Figure 1.

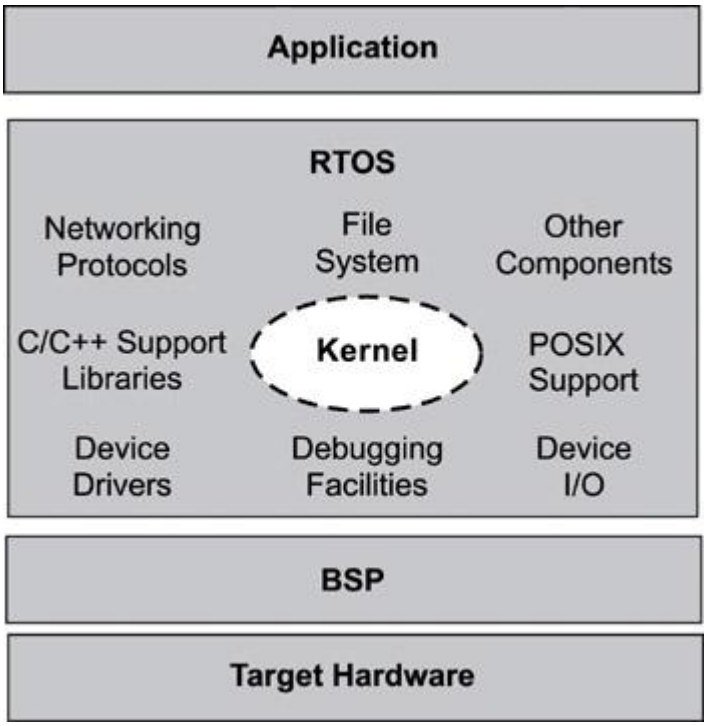


Figure 1: High-level view of an RTOS, its kernel, and other components found in embedded systems.

RTOSes can scale up or down to meet application requirements, this book focuses on the common element at the heart of all RTOSes-the kernel. Most RTOS kernels contain the following components:

- **Scheduler** - is contained within each kernel and follows a set of algorithms that determines which task executes when. Some common examples of scheduling algorithms include round-robin and preemptive scheduling.
- **Objects** - are special kernel constructs that help developers create applications for real-time embedded systems. Common kernel objects include tasks, semaphores, and message queues.
- **Services** - are operations that the kernel performs on an object or, generally operations such as timing, interrupt handling, and resource management.

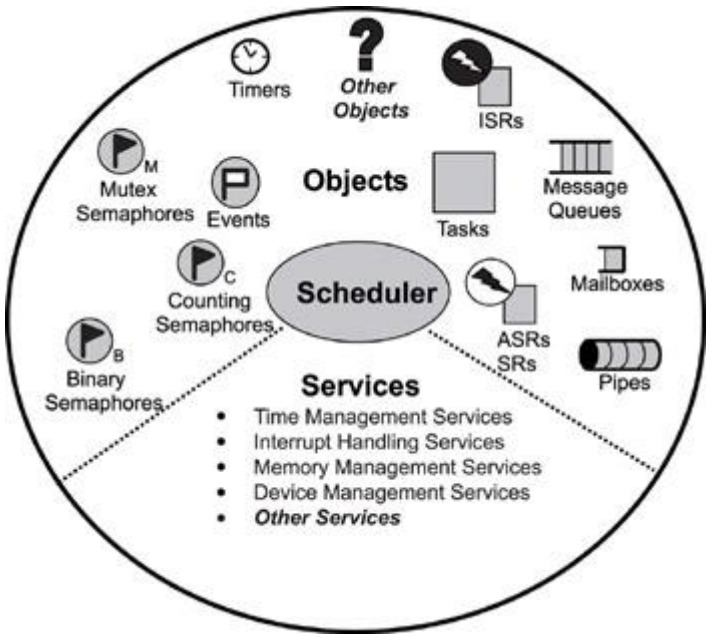


Figure 2: Common components in an RTOS kernel that including objects, the scheduler, and some services.

This diagram is highly simplified; remember that not all RTOS kernels conform to this exact set of objects, scheduling algorithms, and services.

The Scheduler

The scheduler is at the heart of every kernel. A scheduler provides the algorithms needed to determine which task executes when. To understand how scheduling works, this section describes the following topics:

- Schedulable entities,
- Multitasking,
- Context switching,
- Dispatcher, and
- Scheduling algorithms.

1. Schedulable Entities

A schedulable entity is a kernel object that can compete for execution time on a system, based on a predefined scheduling algorithm. Tasks and processes are all examples of schedulable entities found in most kernels.

A task is an independent thread of execution that contains a sequence of independently schedulable instructions. Some kernels provide another type of a schedulable object called a process. Processes are similar to tasks in that they can independently compete

for CPU execution time. Processes differ from tasks in that they provide better memory protection features, at the expense of performance and memory overhead. Despite these differences, for the sake of simplicity, this book uses task to mean either a task or a process.

Note that message queues and semaphores are not schedulable entities. These items are inter-task communication objects used for synchronization and communication.

2. Multitasking

Multitasking is the ability of the operating system to handle multiple activities within set deadlines. A real-time kernel might have multiple tasks that it has to schedule to run. One such multitasking scenario is illustrated in Figure 3.

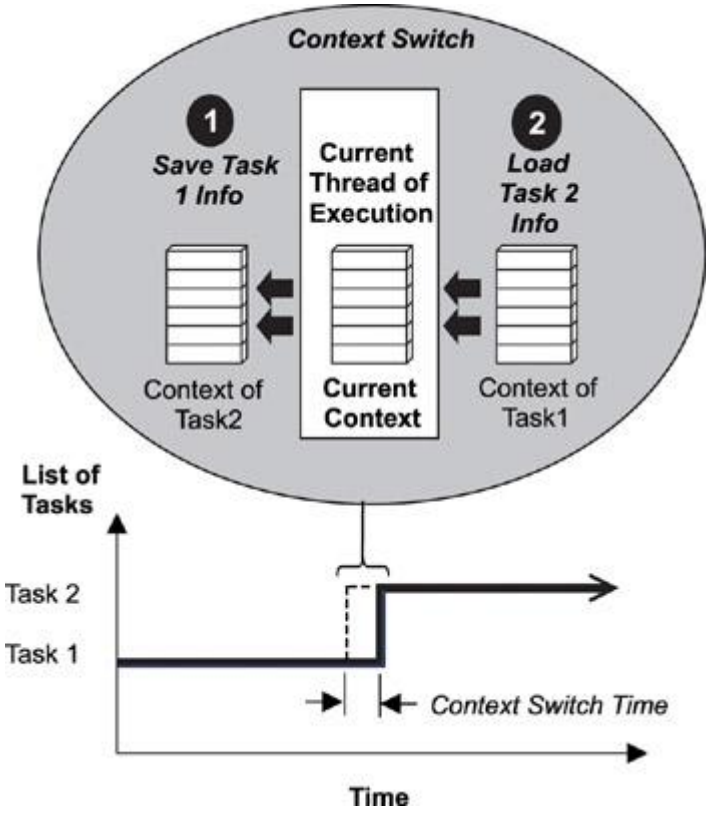


Figure 3: Multitasking using a context switch.

In this scenario, the kernel multitasks in such a way that many threads of execution appear to be running concurrently; however, the kernel is actually interleaving executions sequentially, based on a preset scheduling algorithm The scheduler must ensure that the appropriate task runs at the right time.

An important point to note here is that the tasks follow the kernel's scheduling algorithm, while interrupt service routines (ISR) are triggered to run because of hardware interrupts and their established priorities.

As the number of tasks to schedule increases, so do CPU performance requirements. This fact is due to increased switching between the contexts of the different threads of execution.

3. The Context Switch

Each task has its own context, which is the state of the CPU registers required each time it is scheduled to run. A context switch occurs when the scheduler switches from one task to another. To better understand what happens during a context switch, let's examine further what a typical kernel does in this scenario.

Every time a new task is created, the kernel also creates and maintains an associated task control block (TCB). TCBs are system data structures that the kernel uses to maintain task-specific information. TCBs contain everything a kernel needs to know about a particular task. When a task is running, its context is highly dynamic. This dynamic context is maintained in the TCB. When the task is not running, its context is frozen within the TCB, to be restored the next time the task runs. A typical context switch scenario is illustrated in Figure 3.

As shown in Figure 3, when the kernel's scheduler determines that it needs to stop running task 1 and start running task 2, it takes the following steps:

1. The kernel saves task 1's context information in its TCB.
2. It loads task 2's context information from its TCB, which becomes the current thread of execution.
3. The context of task 1 is frozen while task 2 executes, but if the scheduler needs to run task 1 again, task 1 continues from where it left off just before the context switch.

1. The Dispatcher

The dispatcher is the part of the scheduler that performs context switching and changes the flow of execution. At any time an RTOS is running, the flow of execution, also known as flow of control, is passing through one of three areas: through an application task, through an ISR, or through the kernel. When a task or ISR makes a system call, the flow of control passes to the kernel to execute one of the system routines provided by the kernel. When it is time to leave the kernel, the dispatcher is responsible for passing control to one of the tasks in the user's application. It will not necessarily be the same task that made the system call. It is the scheduling algorithms (to be discussed shortly) of the scheduler that

determines which task executes next. It is the dispatcher that does the actual work of context switching and passing execution control.

Depending on how the kernel is first entered, dispatching can happen differently. When a task makes system calls, the dispatcher is used to exit the kernel after every system call completes. In this case, the dispatcher is used on a call-by-call basis so that it can coordinate task-state transitions that any of the system calls might have caused. (One or more tasks may have become ready to run, for example.)

4. Scheduling Algorithms

As mentioned earlier, the scheduler determines which task runs by following a scheduling algorithm (also known as scheduling policy). Most kernels today support two common scheduling algorithms:

- Preemptive priority-based scheduling, and
- Round-Robin scheduling.

The RTOS manufacturer typically predefines these algorithms; however, in some cases, developers can create and define their own scheduling algorithms. Each algorithm is described next.

Preemptive Priority-Based Scheduling

Of the two scheduling algorithms introduced here, most real-time kernels use preemptive priority-based scheduling by default. As shown in Figure 4 with this type of scheduling, the task that gets to run at any point is the task with the highest priority among all other tasks ready to run in the system.

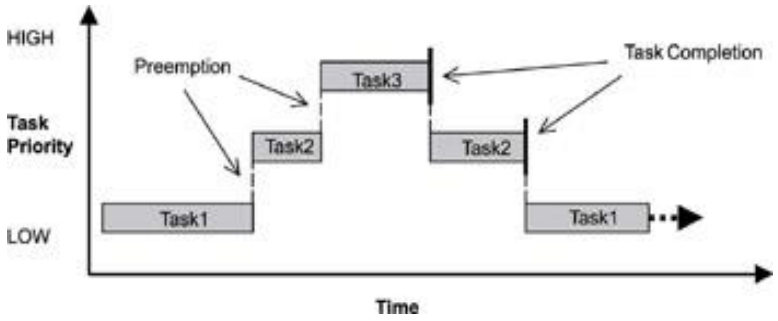


Figure 4: Preemptive priority-based scheduling.

Real-time kernels generally support 256 priority levels, in which 0 is the highest and 255 the lowest. Some kernels appoint the priorities in reverse order, where 255 is the highest

and 0 the lowest. Regardless, the concepts are basically the same. With a preemptive priority-based scheduler, each task has a priority, and the highest-priority task runs first. If a task with a priority higher than the current task becomes ready to run, the kernel immediately saves the current task's context in its TCB and switches to the higher-priority task. As shown in Figure 4 task 1 is preempted by higher-priority task 2, which is then preempted by task 3. When task 3 completes, task 2 resumes; likewise, when task 2 completes, task 1 resumes.

Although tasks are assigned a priority when they are created, a task's priority can be changed dynamically using kernel-provided calls. The ability to change task priorities dynamically allows an embedded application the flexibility to adjust to external events as they occur, creating a true real-time, responsive system. Note, however, that misuse of this capability can lead to priority inversions, deadlock, and eventual system failure.

Round-Robin Scheduling

Round-robin scheduling provides each task an equal share of the CPU execution time. Pure round-robin scheduling cannot satisfy real-time system requirements because in real-time systems, tasks perform work of varying degrees of importance. Instead, preemptive, priority-based scheduling can be augmented with round-robin scheduling which uses time slicing to achieve equal allocation of the CPU for tasks of the same priority as shown in Figure 5.

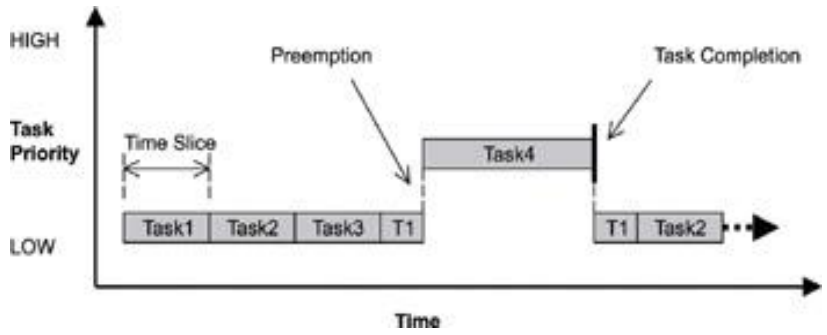


Figure 5: Round-robin and preemptive scheduling.

With time slicing, each task executes for a defined interval, or time slice, in an ongoing cycle, which is the round robin. A run-time counter tracks the time slice for each task, incrementing on every clock tick. When one task's time slice completes, the counter is cleared, and the task is placed at the end of the cycle. Newly added tasks of the same priority are placed at the end of the cycle, with their run-time counters initialized to 0.

If a task in a round-robin cycle is preempted by a higher-priority task, its run-time count is saved and then restored when the interrupted task is again eligible for execution. This

idea is illustrated in Figure 5, in which task 1 is preempted by a higher-priority task 4 but resumes where it left off when task 4 completes.

Objects

Kernel objects are special constructs that are the building blocks for application development for real-time embedded systems. The most common RTOS kernel objects are

- **Tasks**—are concurrent and independent threads of execution that can compete for CPU execution time.
- **Semaphores**—are token-like objects that can be incremented or decremented by tasks for synchronization or mutual exclusion.
- **Message Queues**—are buffer-like data structures that can be used for synchronization, mutual exclusion, and data exchange by passing messages between tasks. Developers creating real-time embedded applications can combine these basic kernel objects (as well as others not mentioned here) to solve common real-time design problems, such as concurrency, activity synchronization, and data communication. These design problems and the kernel objects used to solve them.

Services

Along with objects, most kernels provide services that help developers create applications for real-time embedded systems. These services comprise sets of API calls that can be used to perform operations on kernel objects or can be used in general to facilitate timer management, interrupt handling, device I/O, and memory management. Again, other services might be provided; these services are those most commonly found in RTOS kernels.

Key Characteristics of an RTOS

`An application's requirements define the requirements of its underlying RTOS. Some of the more common attributes are

- Reliability,
- Predictability,
- Performance,
- Compactness, and
- Scalability.

1. Reliability

Embedded systems must be reliable. Depending on the application, the system might need to operate for long periods without human intervention.

Different degrees of reliability may be required. For example, a digital solar-powered calculator might reset itself if it does not get enough light, yet the calculator might still be considered acceptable. On the other hand, a telecom switch cannot reset during operation without incurring high associated costs for down time. The RTOSes in these applications require different degrees of reliability.

Although different degrees of reliability might be acceptable, in general, a reliable system is one that is available (continues to provide service) and does not fail. While RTOSes must be reliable, note that the RTOS by itself is not what is measured to determine system reliability. It is the combination of all system elements-including the hardware, BSP, RTOS, and application-that determines the reliability of a system.

2. Predictability

Because many embedded systems are also real-time systems, meeting time requirements is key to ensuring proper operation. The RTOS used in this case needs to be predictable to a certain degree. The term deterministic describes RTOSes with predictable behavior, in which the completion of operating system calls occurs within known timeframes.

Developers can write simple benchmark programs to validate the determinism of an RTOS. The result is based on timed responses to specific RTOS calls. In a good deterministic RTOS, the variance of the response times for each type of system call is very small.

3. Performance

This requirement dictates that an embedded system must perform fast enough to fulfill its timing requirements. Typically, the more deadlines to be met-and the shorter the time between them-the faster the system's CPU must be. Although underlying hardware can dictate a system's processing power, its software can also contribute to system performance. Typically, the processor's performance is expressed in million instructions per second (MIPS).

Throughput also measures the overall performance of a system, with hardware and software combined. One definition of throughput is the rate at which a system can generate output based on the inputs coming in. Throughput also means the amount of data transferred divided by the time taken to transfer it. Data transfer throughput is typically measured in multiples of bits per second (bps).

Sometimes developers measure RTOS performance on a call-by-call basis. Benchmarks are written by producing timestamps when a system call starts and when it completes. Although this step can be helpful in the analysis stages of design, true performance testing is achieved only when the system performance is measured as a whole.

Tasks

Introduction

Simple software applications are typically designed to run sequentially , one instruction at a time, in a pre-determined chain of instructions. However, this scheme is inappropriate for real-time embedded applications, which generally handle multiple inputs and outputs within tight time constraints. Real-time embedded software applications must be designed for concurrency.

Concurrent design requires developers to decompose an application into small, schedulable, and sequential program units. When done correctly, concurrent design allows system multitasking to meet performance and timing requirements for a real-time system. Most RTOS kernels provide task objects and task management services to facilitate designing concurrency within an application.

This chapter discusses the following topics:

- Task definition,
- Task states and scheduling,
- Typical task operations,
- Typical task structure, and
- Task coordination and concurrency.

Defining a Task

A *task* is an independent thread of execution that can compete with other concurrent tasks for processor execution time. As mentioned earlier, developers decompose applications into multiple concurrent tasks to optimize the handling of inputs and outputs within set time constraints.

A task is *schedulable*; the task is able to compete for execution time on a system, based on a predefined scheduling algorithm. A task is defined by its distinct set of parameters and supporting data structures. Specifically, upon creation, each task has an associated name, a unique ID, a priority (if part of a preemptive scheduling plan), a task control block (TCB), a

stack, and a task routine, as shown in Figure 1). Together, these components make up what is known as the *task object.*

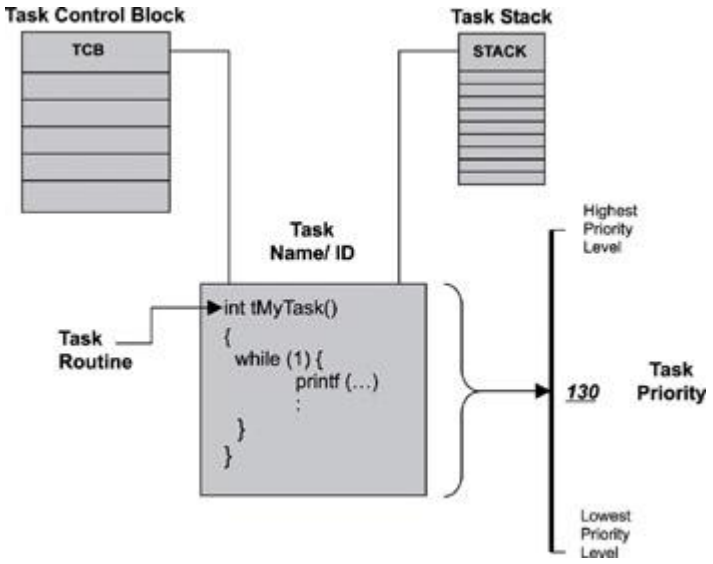


Figure 1: A task, its associated parameters, and supporting data structures.

When the kernel first starts, it creates its own set of *system tasks* and allocates the appropriate priority for each from a set of *reserved priority levels.* The reserved priority levels refer to the priorities used internally by the RTOS for its system tasks. An application should avoid using these priority levels for its tasks because running application tasks at such level may affect the overall system performance or behavior. For most RTOSes, these reserved priorities are not enforced. The kernel needs its system tasks and their reserved priority levels to operate. These priorities should not be modified. Examples of system tasks include:

- **Initialization or startup task**—initializes the system and creates and starts system tasks,
- **Idle task**—uses up processor idle cycles when no other activity is present,
- **Logging task**—logs system messages,
- **Exception-handling task**—handles exceptions, and
- **Debug agent task**—allows debugging with a host debugger. Note that other system tasks might be created during initialization, depending on what other components are included with the kernel.

The idle task, which is created at kernel startup, is one system task that bears mention and should not be ignored. The idle task is set to the lowest priority, typically executes in an endless loop, and runs when either no other task can run or when no other tasks exist, for the sole purpose of using idle processor cycles. The idle task is necessary because the processor

executes the instruction to which the program counter register points while it is running. Unless the processor can be suspended, the program counter must still point to valid instructions even when no tasks exist in the system or when no tasks can run. Therefore, the idle task ensures the processor program counter is always valid when no other tasks are running.

In some cases, however, the kernel might allow a user-configured routine to run instead of the idle task in order to implement special requirements for a particular application. One example of a special requirement is power conservation. When no other tasks can run, the kernel can switch control to the user-supplied routine instead of to the idle task. In this case, the user-supplied routine acts like the idle task but instead initiates power conservation code, such as system suspension, after a period of idle time.

After the kernel has initialized and created all of the required tasks, the kernel jumps to a predefined entry point (such as a predefined function) that serves, in effect, as the beginning of the application. From the entry point, the developer can initialize and create other application tasks , as well as other kernel objects, which the application design might require.

As the developer creates new tasks, the developer must assign each a task name, priority, stack size, and a task routine. The kernel does the rest by assigning each task a unique ID and creating an associated TCB and stack space in memory for it.

Task States and Scheduling

Whether it's a system task or an application task, at any time each task exists in one of a small number of states, including ready, running, or blocked. As the real-time embedded system runs, each task moves from one state to another, according to the logic of a simple finite state machine (FSM). Figure 2 illustrates a typical FSM for task execution states, with brief descriptions of state transitions.

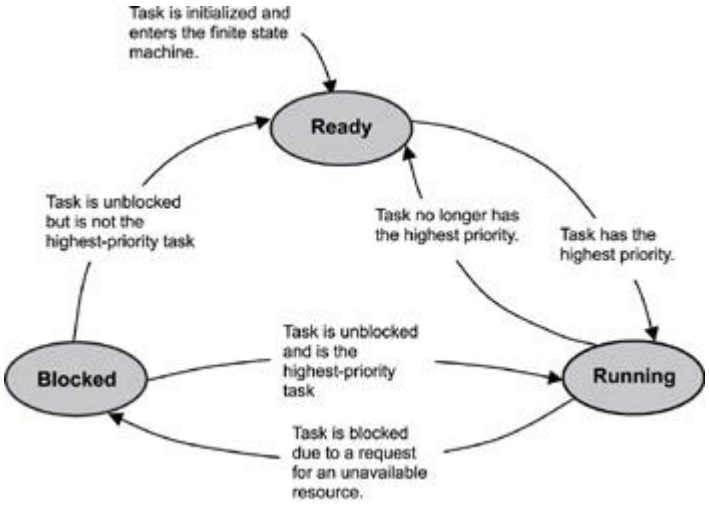


Figure 2: A typical finite state machine for task execution states.

Although kernels can define task-state groupings differently, generally three main states are used in most typical preemptive-scheduling kernels, including:

- **Ready state**-the task is ready to run but cannot because a higher priority task is executing.
- **Blocked state**-the task has requested a resource that is not available, has requested to wait until some event occurs, or has delayed itself for some duration.
- **Running state**-the task is the highest priority task and is running.

Note some commercial kernels, such as the VxWorks kernel, define other, more granular states, such as suspended, pended, and delayed. In this case, pended and delayed are actually sub-states of the blocked state. A pended task is waiting for a resource that it needs to be freed; a delayed task is waiting for a timing delay to end. The suspended state exists for debugging purposes. For more detailed information on the way a particular RTOS kernel implements its FSM for each task, refer to the kernel's user manual.

Regardless of how a kernel implements a task's FSM, it must maintain the current state of all tasks in a running system. As calls are made into the kernel by executing tasks, the kernel's scheduler first determines which tasks need to change states and then makes those changes.

In some cases, the kernel changes the states of some tasks, but no context switching occurs because the state of the highest priority task is unaffected. In other cases, however, these state changes result in a context switch because the former highest priority task either gets blocked or is no longer the highest priority task. When this process happens, the former running task is put into the blocked or ready state, and the new highest priority task starts to execute.

1. Ready State

When a task is first created and made ready to run, the kernel puts it into the ready state. In this state, the task actively competes with all other ready tasks for the processor's execution time. As Figure 2 shows, tasks in the ready state cannot move directly to the blocked state. A task first needs to run so it can make a *blocking call* , which is a call to a function that cannot immediately run to completion, thus putting the task in the blocked state. Ready tasks, therefore, can only move to the running state. Because many tasks might be in the ready state, the kernel's scheduler uses the priority of each task to determine which task to move to the running state.

Figure 3 illustrates, in a five-step scenario, how a kernel scheduler might use a task-ready list to move tasks from the ready state to the running state. This example assumes a single-processor system and a priority-based preemptive scheduling algorithm in which 255 is the lowest priority and 0 is the highest. Note that for simplicity this example does not show system tasks, such as the idle task.

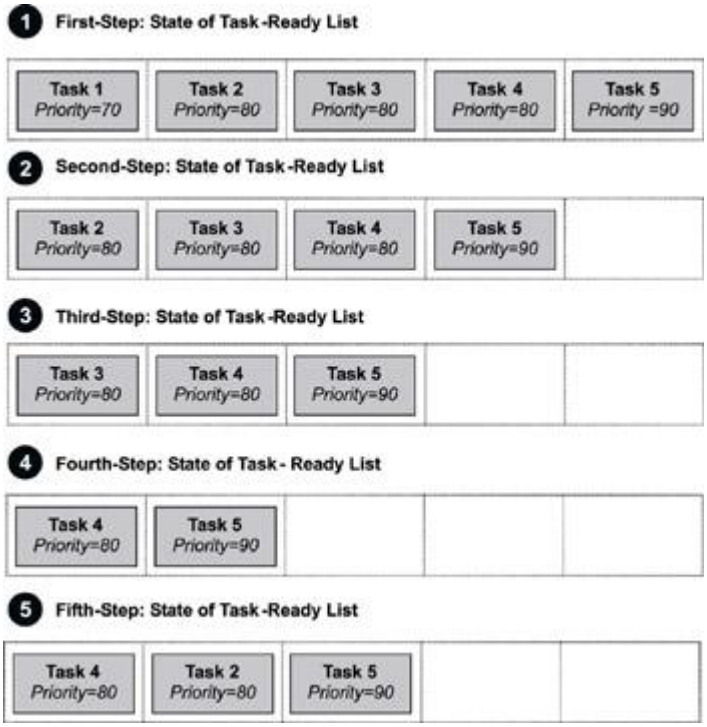


Figure 3: Five steps showing the way a task-ready list works.

In this example, tasks 1, 2, 3, 4, and 5 are ready to run, and the kernel queues them by priority in a task-ready list. Task 1 is the highest priority task (70); tasks 2, 3, and 4 are at the next-highest priority level (80); and task 5 is the lowest priority (90). The following steps explains how a kernel might use the task-ready list to move tasks to and from the ready state:

1. Tasks 1, 2, 3, 4, and 5 are ready to run and are waiting in the task-ready list.

2. Because task 1 has the highest priority (70), it is the first task ready to run. If nothing higher is running, the kernel removes task 1 from the ready list and moves it to the running state.
3. During execution, task 1 makes a blocking call. As a result, the kernel moves task 1 to the blocked state; takes task 2, which is first in the list of the next-highest priority tasks (80), off the ready list; and moves task 2 to the running state.
4. Next, task 2 makes a blocking call. The kernel moves task 2 to the blocked state; takes task 3, which is next in line of the priority 80 tasks, off the ready list; and moves task 3 to the running state.
5. As task 3 runs, frees the resource that task 2 requested. The kernel returns task 2 to the ready state and inserts it at the end of the list of tasks ready to run at priority level 80. Task 3 continues as the currently running task.

2. Running State

On a single-processor system, only one task can run at a time. In this case, when a task is moved to the running state, the processor loads its registers with this task's context. The processor can then execute the task's instructions and manipulate the associated stack.

A task can move back to the ready state while it is running. When a task moves from the running state to the ready state, it is preempted by a higher priority task. In this case, the preempted task is put in the appropriate, priority-based location in the task-ready list, and the higher priority task is moved from the ready state to the running state.

Unlike a ready task, a running task can move to the blocked state in any of the following ways:

- by making a call that requests an unavailable resource,
- by making a call that requests to wait for an event to occur, and
- by making a call to delay the task for some duration.

3. Blocked State

The possibility of blocked states is extremely important in real-time systems because without blocked states, lower priority tasks could not run. If higher priority tasks are not designed to block, CPU starvation can result.

CPU starvation occurs when higher priority tasks use all of the CPU execution time and lower priority tasks do not get to run.

A task can only move to the blocked state by making a blocking call, requesting that some blocking condition be met. A blocked task remains blocked until the blocking condition is met. (It probably ought to be called the *un* blocking condition, but blocking is the terminology in common use among real-time programmers.) Examples of how blocking conditions are met include the following:

- a semaphore token for which a task is waiting is released,
- a message, on which the task is waiting, arrives in a message queue, or a time delay imposed on the task expires.

Typical Task Operations

In addition to providing a task object, kernels also provide *task-management services* . Task-management services include the actions that a kernel performs behind the scenes to support tasks, for example, creating and maintaining the TCB and task stacks.

A kernel, however, also provides an API that allows developers to manipulate tasks. Some of the more common operations that developers can perform with a task object from within the application include:

- Creating and Deleting tasks,
- Controlling task scheduling, and
- Obtaining task information.

Developers should learn how to perform each of these operations for the kernel selected for the project. Each operation is briefly discussed next.

1. Task Creation and Deletion

The most fundamental operations that developers must learn are creating and deleting tasks, as shown in Table 1.

Table 1: Operations for task creation and deletion.

Operation	Description
Create	Creates a task
Delete	Deletes a task

Developers typically create a task using one or two operations, depending on the kernel's API. Some kernels allow developers first to create a task and then start it. In this case,

the task is first created and put into a suspended state; then, the task is moved to the ready state when it is started (made ready to run).

Creating tasks in this manner might be useful for debugging or when special initialization needs to occur between the times that a task is created and started. However, in most cases, it is sufficient to create and start a task using one kernel call.

Many kernels also provide *user-configurable hooks* , which are mechanisms that execute programmer-supplied functions, at the time of specific kernel events. The programmer *registers* the function with the kernel by passing a function pointer to a kernel-provided API . The kernel executes this function when the event of interest occurs. Such events can include:

- When a task is first created,
- When a task is suspended for any reason and a context switch occurs, and
- When a task is deleted.

2. Task Scheduling

From the time a task is created to the time it is deleted, the task can move through various states resulting from program execution and kernel scheduling. Although much of this state changing is automatic, many kernels provide a set of API calls that allow developers to control when a task moves to a different state, as shown in Table 2. This capability is called *manual scheduling* .

Table 2: Operations for task scheduling.

Operation	Description
Suspend	Suspends a task
Resume	Resumes a task
Delay	Delays a task
Restart	Restarts a task
Get Priority	Gets the current task's priority
Set Priority	Dynamically sets a task's priority
Preemption lock	Locks out higher priority tasks from preempting the current task

Table 2: Operations for task scheduling.

Operation	Description
Preemption unlock	Unlocks a preemption lock

Using manual scheduling, developers can suspend and resume tasks from within an application. Doing so might be important for debugging purposes or, as discussed earlier, for suspending a high-priority task so that lower priority tasks can execute.

A developer might want to delay (block) a task, for example, to allow manual scheduling or to wait for an external condition that does not have an associated interrupt. Delaying a task causes it to relinquish the CPU and allow another task to execute. After the delay expires, the task is returned to the task-ready list after all other ready tasks at its priority level. A delayed task waiting for an external condition can wake up after a set time to check whether a specified condition or event has occurred, which is called *polling*.

Typical Task Structure

When writing code for tasks, tasks are structured in one of two ways:

- Run to completion, or
- Endless loop.

Both task structures are relatively simple. Run-to-completion tasks are most useful for initialization and startup. They typically run once, when the system first powers on. Endless-loop tasks do the majority of the work in the application by handling inputs and outputs. Typically, they run many times while the system is powered on.

1 Run-to-Completion Tasks

An example of a run-to-completion task is the application-level initialization task, shown in Listing 1. The initialization task initializes the application and creates additional services, tasks, and needed kernel objects.

Listing 1: Pseudo code for a run-to-completion task.

```
RunToCompletionTask ()
{
    Initialize application
    Create ‘endless loop tasks'
```

Create kernel objects

Delete or suspend this task

}

2. Endless-Loop Tasks

As with the structure of the application initialization task, the structure of an endless loop task can also contain initialization code. The endless loop's initialization code, however, only needs to be executed when the task first runs, after which the task executes in an endless loop, as shown in Listing 2.

The critical part of the design of an endless-loop task is the one or more blocking calls within the body of the loop. These blocking calls can result in the blocking of this endless-loop task, allowing lower priority tasks to run.

Listing 5.2: Pseudo code for an endless-loop task.

```
EndlessLoopTask ()
{
    Initialization code
    Loop Forever
    {
        Body of loop
        Make one or more blocking calls
    }
}
```

Synchronization, Communication, and Concurrency

Tasks synchronize and communicate amongst themselves by using *intertask primitives* , which are kernel objects that facilitate synchronization and communication between two or more threads of execution. Examples of such objects include semaphores, message queues, signals, and pipes, as well as other types of objects.

The concept of concurrency and how an application is optimally decomposed into concurrent tasks. For now, remember that the task object is the fundamental construct of most kernels. Tasks, along with task-management services, allow developers to design applications

for concurrency to meet multiple time constraints and to address various design problems inherent to real-time embedded applications.

Modularizing An Application For Concurrency

Introduction

Many activities need to be completed when designing applications for real-time systems. One group of activities requires identifying certain elements. Some of the more important elements to identify include:

1. system requirements,

2. inputs and outputs,

3. real-time deadlines,

4. events and event response times,

5. event arrival patterns and frequencies,

6. required objects and other components,

7. tasks that need to be concurrent,

8. system schedulability, and

9. useful or needed synchronization protocols for inter-task communications.

Synchronization And Communication

Introduction

Software applications for real-time embedded systems use concurrency to maximize efficiency. As a result, an application's design typically involves multiple concurrent threads, tasks, or processes. Coordinating these activities requires inter-task synchronization and communication.

This chapter focuses on:

- resource synchronization,
- activity synchronization,
- inter-task communication, and
- ready-to-use embedded design patterns.

Synchronization

Synchronization is classified into two categories: *resource synchronization* and *activity synchronization* . Resource synchronization determines whether access to a shared resource is safe, and, if not, when it will be safe. Activity synchronization determines whether the execution of a multithreaded program has reached a certain state and, if it hasn't, how to wait for and be notified when this state is reached.

Resource Synchronization

Access by multiple tasks must be synchronized to maintain the integrity of a shared resource. This process is called *resource synchronization* , a term closely associated with critical sections and mutual exclusions.

Mutual exclusion is a provision by which only one task at a time can access a shared resource. A *critical section* is the section of code from which the shared resource is accessed.

As an example, consider two tasks trying to access shared memory. One task (the sensor task) periodically receives data from a sensor and writes the data to shared memory. Meanwhile, a second task (the display task) periodically reads from shared memory and sends the data to a display. The common design pattern of using shared memory is illustrated in Figure 1.

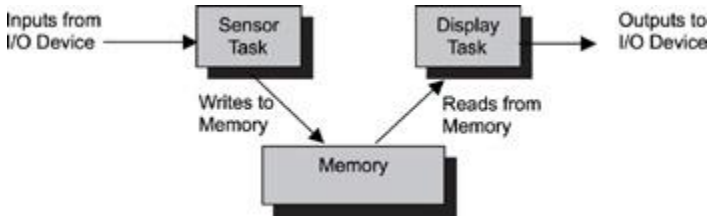


Figure 1: Multiple tasks accessing shared memory.

Problems arise if access to the shared memory is not exclusive, and multiple tasks can simultaneously access it. For example, if the sensor task has not completed writing data to the shared memory area before the display task tries to display the data, the display would contain a mixture of data extracted at different times, leading to erroneous data interpretation.

The section of code in the sensor task that writes input data to the shared memory is a critical section of the sensor task. The section of code in the display task that reads data from

the shared memory is a critical section of the display task. These two critical sections are called *competing critical sections* because they access the same shared resource.

A mutual exclusion algorithm ensures that one task's execution of a critical section is not interrupted by the competing critical sections of other concurrently executing tasks.

One way to synchronize access to shared resources is to use a client-server model, in which a central entity called a *resource server* is responsible for synchronization. Access requests are made to the resource server, which must grant permission to the requestor before the requestor can access the shared resource. The resource server determines the eligibility of the requestor based on pre-assigned rules or run-time heuristics.

Barrier synchronization comprises three actions:

- a task posts its arrival at the barrier,
- the task waits for other participating tasks to reach the barrier, and
- the task receives notification to proceed beyond the barrier.

- Another representative of activity synchronization mechanisms is *rendezvous synchronization* , which, as its name implies, is an execution point where two tasks meet. The main difference between the barrier and the rendezvous is that the barrier allows activity synchronization among two or more tasks, while rendezvous synchronization is between two tasks.

- In rendezvous synchronization, a synchronization and communication point called an *entry* is constructed as a function call. One task defines its entry and makes it public. Any task with knowledge of this entry can call it as an ordinary function call. The task that defines the entry accepts the call, executes it, and returns the results to the caller. The issuer of the entry call establishes a rendezvous with the task that defined the entry.

Communication

Tasks communicate with one another so that they can pass information to each other and coordinate their activities in a multithreaded embedded application. Communication can be signal-centric, data-centric, or both. In *signal-centric communication* , all necessary information is conveyed within the event signal itself. In *data-centric communication* , information is carried within the transferred data. When the two are combined, data transfer accompanies event notification.

When communication involves data flow and is unidirectional, this communication model is called *loosely coupled communication.* In this model, the data producer does not require a response from the consumer. Figure 4 illustrates an example of loosely coupled communication.

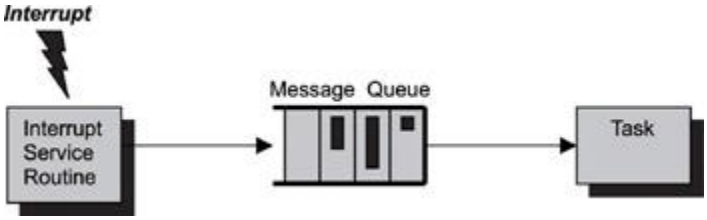


Figure 4: Loosely coupled ISR-to-task communication using message queues.

For example, an ISR for an I/O device retrieves data from a device and routes the data to a dedicated processing task. The ISR neither solicits nor requires feedback from the processing task. By contrast, in *tightly coupled communication* , the data movement is bidirectional. The data producer synchronously waits for a response to its data transfer before resuming execution, or the response is returned asynchronously while the data producer continues its function.

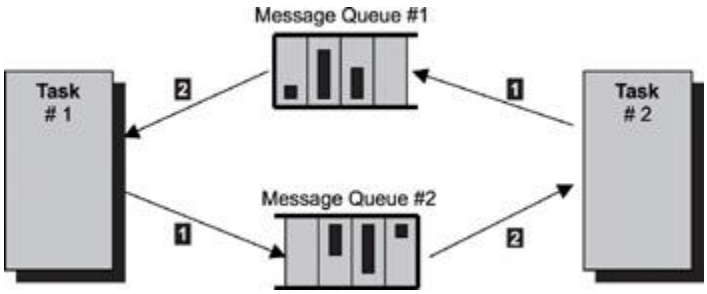


Figure 5: Tightly coupled task-to-task communication using message queues.

In tightly coupled communication, as shown in Figure 5, task #1 sends data to task #2 using message queue #2 and waits for confirmation to arrive at message queue #1. The data communication is bidirectional. It is necessary to use a message queue for confirmations because the confirmation should contain enough information in case task #1 needs to re-send the data. Task #1 can send multiple messages to task #2, i.e., task #1 can continue sending messages while waiting for confirmation to arrive on message queue #2.

Communication has several purposes, including the following:

- transferring data from one task to another,
- signaling the occurrences of events between tasks,
- allowing one task to control the execution of other tasks,
- synchronizing activities, and

- implementing custom synchronization protocols for resource sharing.

- The first purpose of communication is for one task to transfer data to another task. Between the tasks, there can exist data dependency, in which one task is the data producer and another task is the data consumer. For example, consider a specialized processing task that is waiting for data to arrive from message queues or pipes or from shared memory. In this case, the data producer can be either an ISR or another task. The consumer is the processing task. The data source can be an I/O device or another task.

- The second purpose of communication is for one task to signal the occurrences of events to another task. Either physical devices or other tasks can generate events. A task or an ISR that is responsible for an event, such as an I/O event, or a set of events can signal the occurrences of these events to other tasks. Data might or might not accompany event signals. Consider, for example, a timer chip ISR that notifies another task of the passing of a time tick.

- The third purpose of communication is for one task to control the execution of other tasks. Tasks can have a master/slave relationship, known as *process control* . For example, in a control system, a master task that has the full knowledge of the entire running system controls individual subordinate tasks. Each subtask is responsible for a component, such as various sensors of the control system. The master task sends commands to the subordinate tasks to enable or disable sensors. In this scenario, data flow can be either unidirectional or bidirectional if feedback is returned from the subordinate tasks.

- The fourth purpose of communication is to synchronize activities. The computation example given in 'Activity Synchronization' on page 233, section 15.2.2, shows that when multiple tasks are waiting at the execution barrier, each task waits for a signal from the last task that enters the barrier, so that each task can continue its own execution. In this example, it is insufficient to notify the tasks that the final computation has completed; additional information, such as the actual computation results, must also be conveyed.

- The fifth purpose of communication is to implement additional synchronization protocols for resource sharing. The tasks of a multithreaded program can implement

custom, more-complex resource synchronization protocols on top of the system-supplied synchronization primitives.

Semaphores

Introduction

Multiple concurrent threads of execution within an application must be able to synchronize their execution and coordinate mutually exclusive access to shared resources. To address these requirements, RTOS kernels provide a semaphore object and associated semaphore management services.

This chapter discusses the following:

- Defining a semaphore,
- Typical semaphore operations, and
- Common semaphore use.

Defining Semaphores

A *semaphore* (sometimes called a *semaphore token*) is a kernel object that one or more threads of execution can acquire or release for the purposes of synchronization or mutual exclusion.

When a semaphore is first created, the kernel assigns to it an associated semaphore control block (SCB), a unique ID, a value (binary or a count), and a task-waiting list, as shown in Figure 1.

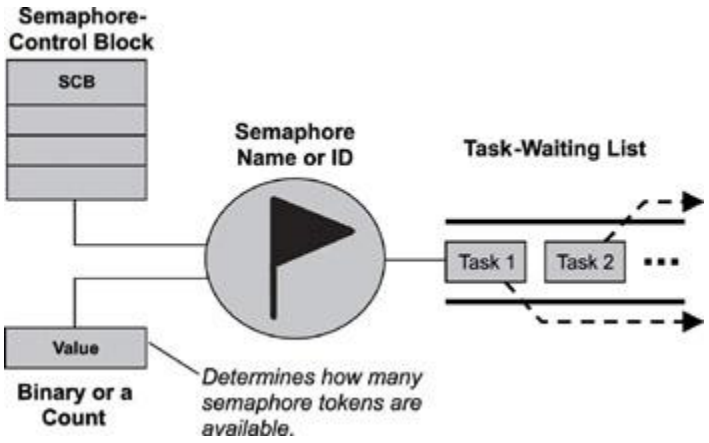


Figure 1: A semaphore, its associated parameters, and supporting data structures.

A semaphore is like a key that allows a task to carry out some operation or to access a resource. If the task can acquire the semaphore, it can carry out the intended operation or access the resource. A single semaphore can be acquired a finite number of times. In this sense, acquiring a semaphore is like acquiring the duplicate of a key from an apartment manager—when the apartment manager runs out of duplicates, the manager can give out no more keys. Likewise, when a semaphore's limit is reached, it can no longer be acquired until someone gives a key back or releases the semaphore.

The kernel tracks the number of times a semaphore has been acquired or released by maintaining a token count, which is initialized to a value when the semaphore is created. As a task acquires the semaphore, the token count is decremented; as a task releases the semaphore, the count is incremented.

If the token count reaches 0, the semaphore has no tokens left. A requesting task, therefore, cannot acquire the semaphore, and the task blocks if it chooses to wait for the semaphore to become available.

The task-waiting list tracks all tasks blocked while waiting on an unavailable semaphore. These blocked tasks are kept in the task-waiting list in either first in/first out (FIFO) order or highest priority first order.

When an unavailable semaphore becomes available, the kernel allows the first task in the task-waiting list to acquire it. The kernel moves this unblocked task either to the running state, if it is the highest priority task, or to the ready state, until it becomes the highest priority task and is able to run. Note that the exact implementation of a task-waiting list can vary from one kernel to another.

1. Binary Semaphores

A *binary semaphore* can have a value of either 0 or 1. When a binary semaphore's value is 0, the semaphore is considered *unavailable* (or *empty);* when the value is 1, the binary semaphore is considered *available* (or *full*). Note that when a binary semaphore is first created, it can be initialized to either available or unavailable (1 or 0, respectively). The state diagram of a binary semaphore is shown in Figure 2.

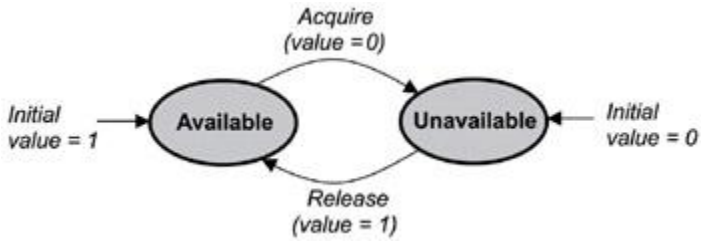


Figure 2: The state diagram of a binary semaphore.

Binary semaphores are treated as *global resources*, which means they are shared among all tasks that need them. Making the semaphore a global resource allows any task to release it, even if the task did not initially acquire it.

2. Counting Semaphores

A *counting semaphore* uses a count to allow it to be acquired or released multiple times. When creating a counting semaphore, assign the semaphore a count that denotes the number of semaphore tokens it has initially. If the initial count is 0, the counting semaphore is created in the unavailable state. If the count is greater than 0, the semaphore is created in the available state, and the number of tokens it has equals its count, as shown in Figure 3.

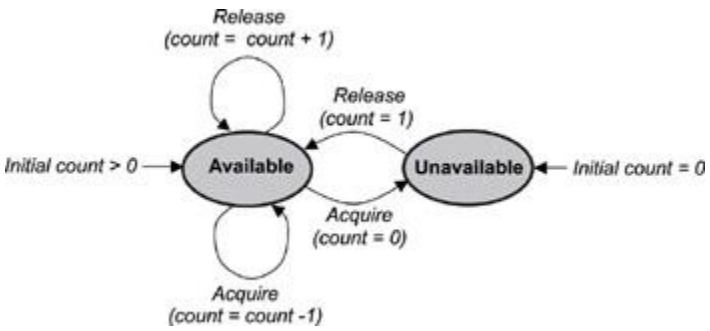


Figure 3: The state diagram of a counting semaphore.

One or more tasks can continue to acquire a token from the counting semaphore until no tokens are left. When all the tokens are gone, the count equals 0, and the counting semaphore moves from the available state to the unavailable state. To move from the unavailable state back to the available state, a semaphore token must be released by any task.

3. Mutual Exclusion (Mutex) Semaphores

A mutual exclusion (mutex) semaphore is a special binary semaphore that supports ownership, recursive access, task deletion safety, and one or more protocols for avoiding problems inherent to mutual exclusion. Figure 4 illustrates the state diagram of a mutex.

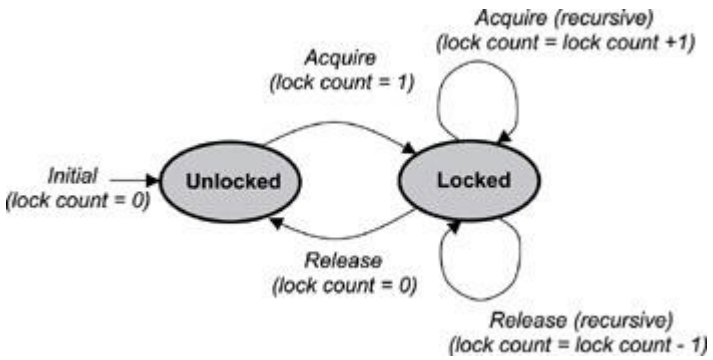


Figure 4: The state diagram of a mutual exclusion (mutex) semaphore.

A mutex is initially created in the unlocked state, in which it can be acquired by a task. After being acquired, the mutex moves to the locked state. Conversely, when the task releases the mutex, the mutex returns to the unlocked state. Note that some kernels might use the terms *lock* and *unlock* for a mutex instead of *acquire* and *release*.

Depending on the implementation, a mutex can support additional features not found in binary or counting semaphores. These key differentiating features include ownership, recursive locking, task deletion safety, and priority inversion avoidance protocols.

Typical Semaphore Operations

Typical operations that developers might want to perform with the semaphores in an application include:

- creating and deleting semaphores,
- acquiring and releasing semaphores,
- clearing a semaphore's task-waiting list, and
- getting semaphore information.

1. Creating and Deleting Semaphores

Table 1 identifies the operations used to create and delete semaphores.

Table 1: Semaphore creation and deletion operations.

Operation	Description
Create	Creates a semaphore
Delete	Deletes a semaphore

Several things must be considered, however, when creating and deleting semaphores. If a kernel supports different types of semaphores, different calls might be used for creating binary, counting, and mutex semaphores, as follows:

- **Binary—**specify the initial semaphore state and the task-waiting order.
- **Counting—**specify the initial semaphore count and the task-waiting order.
- **Mutex**—specify the task-waiting order and enable task deletion safety, recursion, and priority-inversion avoidance protocols, if supported.

Semaphores can be deleted from within any task by specifying their IDs and making semaphore-deletion calls. Deleting a semaphore is not the same as releasing it. When a semaphore is deleted, blocked tasks in its task-waiting list are unblocked and moved either to the ready state or to the running state (if the unblocked task has the highest priority). Any tasks, however, that try to acquire the deleted semaphore return with an error because the semaphore no longer exists.

2. Acquiring and Releasing Semaphores

Table 2 identifies the operations used to acquire or release semaphores.

Table 2: Semaphore acquire and release operations.

Operation	Description
Acquire	Acquire a semaphore token
Release	Release a semaphore token

The operations for acquiring and releasing a semaphore might have different names, depending on the kernel: for example, *take* and *give* , *sm_p* and *sm_v* , *pend* and *post* , and *lock* and *unlock* . Regardless of the name, they all effectively acquire and release semaphores.

Tasks typically make a request to acquire a semaphore in one of the following ways:

- **Wait forever**—task remains blocked until it is able to acquire a semaphore.
- **Wait with a timeout**—task remains blocked until it is able to acquire a semaphore or until a set interval of time, called the *timeout interval* , passes. At this point, the task is removed from the semaphore's task-waiting list and put in either the ready state or the running state.
- **Do not wait**—task makes a request to acquire a semaphore token, but, if one is not available, the task does not block.

Typical Semaphore Use

Semaphores are useful either for synchronizing execution of multiple tasks or for coordinating access to a shared resource. The following examples and general discussions illustrate using different types of semaphores to address common synchronization design requirements effectively, as listed:

- Wait-and-signal synchronization,
- Multiple-task wait-and-signal synchronization,
- Credit-tracking synchronization,
- Single shared-resource-access synchronization,
- Recursive shared-resource-access synchronization, and
- Multiple shared-resource-access synchronization.

Note that, for the sake of simplicity, not all uses of semaphores are listed here. Also, later chapters of this book contain more advanced discussions on the different ways that mutex semaphores can handle priority inversion.

1 Wait-and-Signal Synchronization

Two tasks can communicate for the purpose of synchronization without exchanging data. For example, a binary semaphore can be used between two tasks to coordinate the transfer of execution control, as shown in Figure 5.

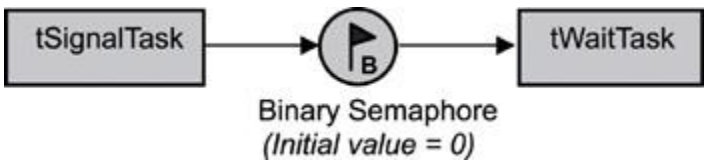


Figure 5: Wait-and-signal synchronization between two tasks.

In this situation, the binary semaphore is initially unavailable (value of 0). tWaitTask has higher priority and runs first. The task makes a request to acquire the semaphore but is blocked because the semaphore is unavailable. This step gives the lower priority tSignalTask a chance to run; at some point, tSignalTask releases the binary semaphore and unblocks tWaitTask. The pseudo code for this scenario is shown in Listing 1.

2. Multiple-Task Wait-and-Signal Synchronization

When coordinating the synchronization of more than two tasks, use the flush operation on the task-waiting list of a binary semaphore, as shown in Figure 6.

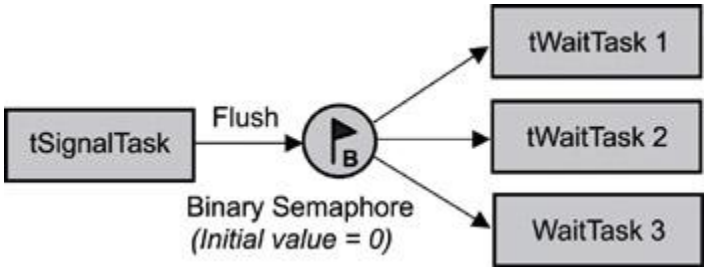


Figure 6: Wait-and-signal synchronization between multiple tasks.

As in the previous case, the binary semaphore is initially unavailable (value of 0). The higher priority tWaitTasks 1, 2, and 3 all do some processing; when they are done, they try to acquire the unavailable semaphore and, as a result, block. This action gives tSignalTask a chance to complete its processing and execute a flush command on the semaphore, effectively unblocking the three tWaitTasks, as shown in Listing 2.

3. Credit-Tracking Synchronization

Sometimes the rate at which the signaling task executes is higher than that of the signaled task. In this case, a mechanism is needed to count each signaling occurrence. The counting semaphore provides just this facility. With a counting semaphore, the signaling task can continue to execute and increment a count at its own pace, while the wait task, when unblocked, executes at its own pace, as shown in Figure 7.

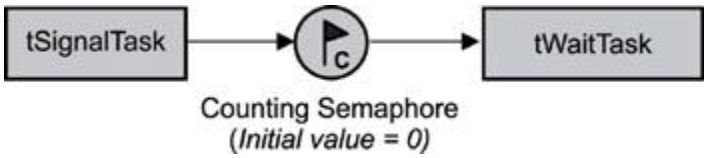


Figure 7: Credit-tracking synchronization between two tasks.

Again, the counting semaphore's count is initially 0, making it unavailable. The lower priority tWaitTask tries to acquire this semaphore but blocks until tSignalTask makes the semaphore available by performing a release on it. Even then, tWaitTask will waits in the ready state until the higher priority tSignalTask eventually relinquishes the CPU by making a blocking call or delaying itself, as shown in Listing 3.

4. Single Shared-Resource-Access Synchronization

One of the more common uses of semaphores is to provide for mutually exclusive access to a shared resource. A shared resource might be a memory location, a data structure, or an I/O device-essentially anything that might have to be shared between two or more concurrent threads of execution. A semaphore can be used to serialize access to a shared resource, as shown in Figure 8.

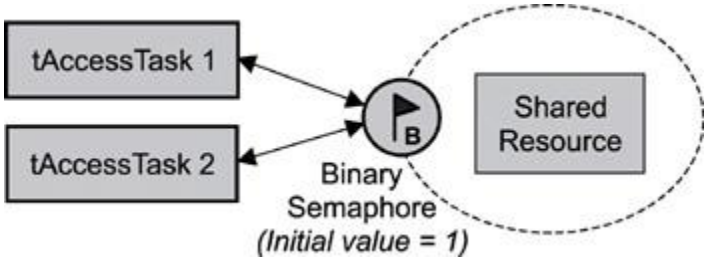


Figure 8: Single shared-resource-access synchronization.

In this scenario, a binary semaphore is initially created in the available state (value = 1) and is used to protect the shared resource.

5. Recursive Shared-Resource-Access Synchronization

Sometimes a developer might want a task to access a shared resource recursively. This situation might exist if tAccessTask calls Routine A that calls Routine B, and all three need access to the same shared resource, as shown in Figure 9.

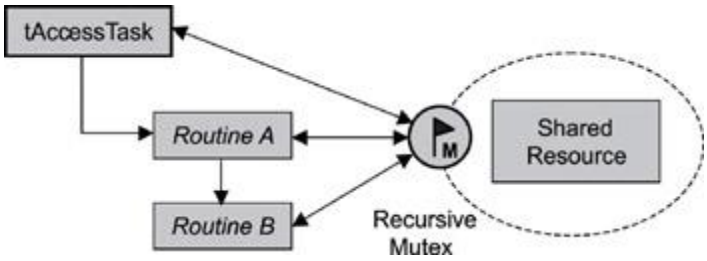


Figure 9: Recursive shared- resource-access synchronization.

If a semaphore were used in this scenario, the task would end up blocking, causing a deadlock. When a routine is called from a task, the routine effectively becomes a part of the task. When Routine A runs, therefore, it is running as a part of tAccessTask. Routine A trying to acquire the semaphore is effectively the same as tAccessTask trying to acquire the same semaphore. In this case, tAccessTask would end up blocking while waiting for the unavailable semaphore that it already has.

One solution to this situation is to use a recursive mutex. After tAccessTask locks the mutex, the task owns it. Additional attempts from the task itself or from routines that it calls to lock the mutex succeed. As a result, when Routines A and B attempt to lock the mutex, they succeed without blocking.

6. Multiple Shared-Resource-Access Synchronization

For cases in which multiple equivalent shared resources are used, a counting semaphore comes in handy, as shown in Figure 10.

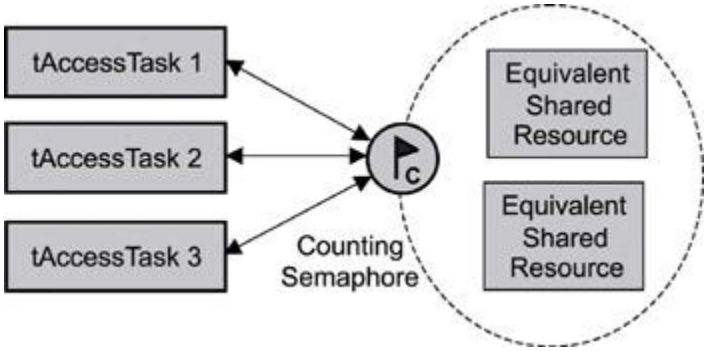


Figure 10: Single shared-resource-access synchronization.

Note that this scenario does not work if the shared resources are not equivalent. The counting semaphore's count is initially set to the number of equivalent shared resources: in this example, 2. As a result, the first two tasks requesting a semaphore token are successful. However, the third task ends up blocking until one of the previous two tasks releases a semaphore token, as shown in Listing 6. Note that similar code is used for tAccessTask 1, 2, and 3.

Message Queues

Introduction

Activity synchronization of two or more threads of execution. Such synchronization helps tasks cooperate in order to produce an efficient real-time system. In many cases, however, task activity synchronization alone does not yield a sufficiently responsive application. Tasks must also be able to exchange messages. To facilitate inter-task data communication, kernels provide a message queue object and message queue management services.

This chapter discusses the following:

- Defining message queues,
- Message queue states,
- Message queue content,
- Typical message queue operations, and
- Typical message queue use.

Defining Message Queues

- A message queue is a buffer-like object through which tasks and ISRs send and receive messages to communicate and synchornize with data. A message queue is like

a pipeline. It temporarily holds messages from a sender until the intended receiver is ready to read them. This temporary buffering decouples a sending and receiving task; that is, it frees the tasks from having to send and receive messages simultaneously.

- A message queue has several associated components that the kernel uses to manage the queue. When a message queue is first created, it is assigned an associated queue control block (QCB), a message queue name, a unique ID, memory buffers, a queue length, a maximum message length, and one or more task-waiting lists, as illustrated in Figure 1.

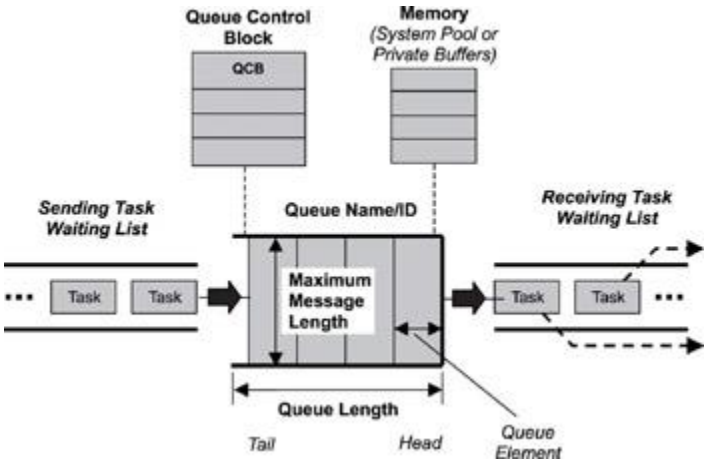


Figure 1: A message queue, its associated parameters, and supporting data structures.

- It is the kernel's job to assign a unique ID to a message queue and to create its QCB and task-waiting list. The kernel also takes developer-supplied parameters—such as the length of the queue and the maximum message length—to determine how much memory is required for the message queue. After the kernel has this information, it allocates memory for the message queue from either a pool of system memory or some private memory space.

- The message queue itself consists of a number of elements, each of which can hold a single message. The elements holding the first and last messages are called the *head* and *tail* respectively. Some elements of the queue may be empty (not containing a message). The total number of elements (empty or not) in the queue is the *total length of the queue* . The developer specified the queue length when the queue was created.

- As Figure 1 shows, a message queue has two associated task-waiting lists. The receiving task-waiting list consists of tasks that wait on the queue when it is empty.

The sending list consists of tasks that wait on the queue when it is full. Empty and full message-queue states, as well as other key concepts, are discussed in more detail next.

Message Queue States

- As with other kernel objects, message queues follow the logic of a simple FSM, as shown in Figure 2 When a message queue is first created, the FSM is in the empty state. If a task attempts to receive messages from this message queue while the queue is empty, the task blocks and, if it chooses to, is held on the message queue's task-waiting list, in either a FIFO or priority-based order.

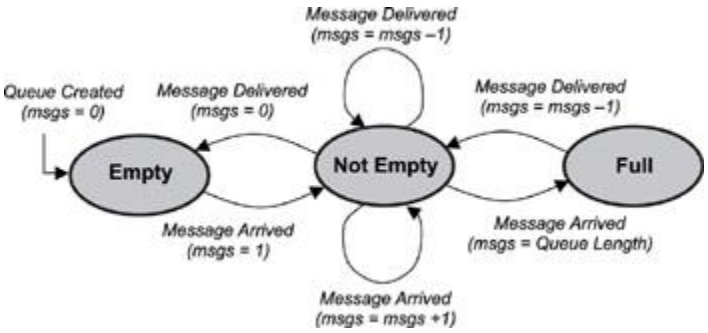


Figure 2: The state diagram for a message queue.

- In this scenario, if another task sends a message to the message queue, the message is delivered directly to the blocked task. The blocked task is then removed from the task-waiting list and moved to either the ready or the running state. The message queue in this case remains empty because it has successfully delivered the message.

- If another message is sent to the same message queue and no tasks are waiting in the message queue's task-waiting list, the message queue's state becomes not empty.

- As additional messages arrive at the queue, the queue eventually fills up until it has exhausted its free space. At this point, the number of messages in the queue is equal to the queue's length, and the message queue's state becomes full. While a message queue is in this state, any task sending messages to it will not be successful unless some other task first requests a message from that queue, thus freeing a queue element.

- In some kernel implementations when a task attempts to send a message to a full message queue, the sending function returns an error code to that task. Other kernel implementations allow such a task to block, moving the blocked task into the sending task-waiting list, which is separate from the receiving task-waiting list.

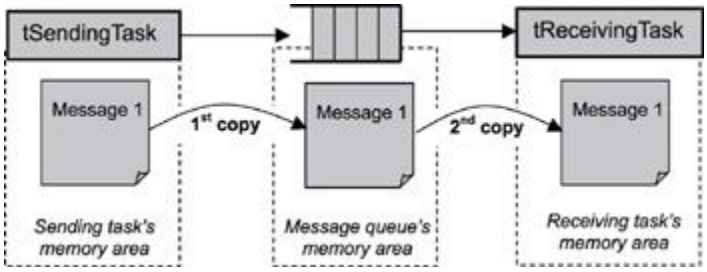


Figure 3: Message copying and memory use for sending and receiving messages.

Message Queue Content

Message queues can be used to send and receive a variety of data. Some examples include:

- a temperature value from a sensor,
- a bitmap to draw on a display,
- a text message to print to an LCD,
- a keyboard event, and
- a data packet to send over the network.

Some of these messages can be quite long and may exceed the maximum message length, which is determined when the queue is created. (Maximum message length should not be confused with total queue length, which is the total number of messages the queue can hold.) One way to overcome the limit on message length is to send a pointer to the data, rather than the data itself. Even if a long message might fit into the queue, it is sometimes better to send a pointer instead in order to improve both performance and memory utilization.

When a task sends a message to another task, the message normally is copied twice, as shown in Figure 3 The first time, the message is copied when the message is sent from the sending task's memory area to the message queue's memory area. The second copy occurs when the message is copied from the message queue's memory area to the receiving task's memory area.

An exception to this situation is if the receiving task is already blocked waiting at the message queue. Depending on a kernel's implementation, the message might be copied just once in this case—from the sending task's memory area to the receiving task's memory area, bypassing the copy to the message queue's memory area.

Because copying data can be expensive in terms of performance and memory requirements, keep copying to a minimum in a real-time embedded system by keeping messages small or, if that is not feasible, by using a pointer instead.

Message Queue Storage

Different kernels store message queues in different locations in memory. One kernel might use a system pool, in which the messages of all queues are stored in one large shared area of memory. Another kernel might use separate memory areas, called private buffers, for each message queue.

Typical Message Queue Operations

Typical message queue operations include the following:

- creating and deleting message queues,
- sending and receiving messages, and
- obtaining message queue information.

1 Creating and Deleting Message Queues

Message queues can be created and deleted by using two simple calls, as shown in Table 1.

Table 1: Message queue creation and deletion operations.

Operation	Description
Create	Creates a message queue
Delete	Deletes a message queue

When created, message queues are treated as global objects and are not owned by any particular task. Typically, the queue to be used by each group of tasks or ISRs is assigned in the design.

When creating a message queue, a developer needs to make some initial decisions about the length of the message queue, the maximum size of the messages it can handle, and the waiting order for tasks when they block on a message queue.

Deleting a message queue automatically unblocks waiting tasks. The blocking call in each of these tasks returns with an error. Messages that were queued are lost when the queue is deleted.

2 Sending and Receiving Messages

The most common uses for a message queue are sending and receiving messages. These operations are performed in different ways, some of which are listed in Table 2 .

Table 2: Sending and receiving messages.

Operation	Description
Send	Sends a message to a message queue
Receive	Receives a message from a message queue
Broadcast	Broadcasts messages

Sending Messages

When sending messages, a kernel typically fills a message queue from head to tail in FIFO order, as shown in Figure 4. Each new message is placed at the end of the queue.

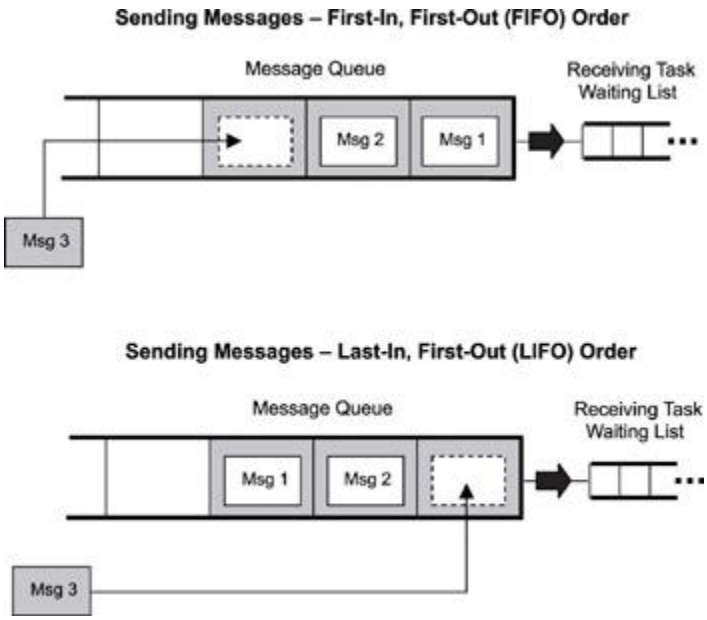


Figure 4: Sending messages in FIFO or LIFO order.

Many message-queue implementations allow urgent messages to go straight to the head of the queue. If all arriving messages are urgent, they all go to the head of the queue, and the queuing order effectively becomes last-in/first-out (LIFO). Many message-queue implementations also allow ISRs to send messages to a message queue. In any case, messages are sent to a message queue in the following ways:

- not block (ISRs and tasks),
- block with a timeout (tasks only), and

- block forever (tasks only).

At times, messages must be sent without blocking the sender. If a message queue is already full, the send call returns with an error, and the task or ISR making the call continues executing. This type of approach to sending messages is the only way to send messages from ISRs, because ISRs cannot block.

Most times, however, the system should be designed so that a task will block if it attempts to send a message to a queue that is full. Setting the task to block either forever or for a specified timeout accomplishes this step. (Figure 5). The blocked task is placed in the message queue's task-waiting list, which is set up in either FIFO or priority-based order.

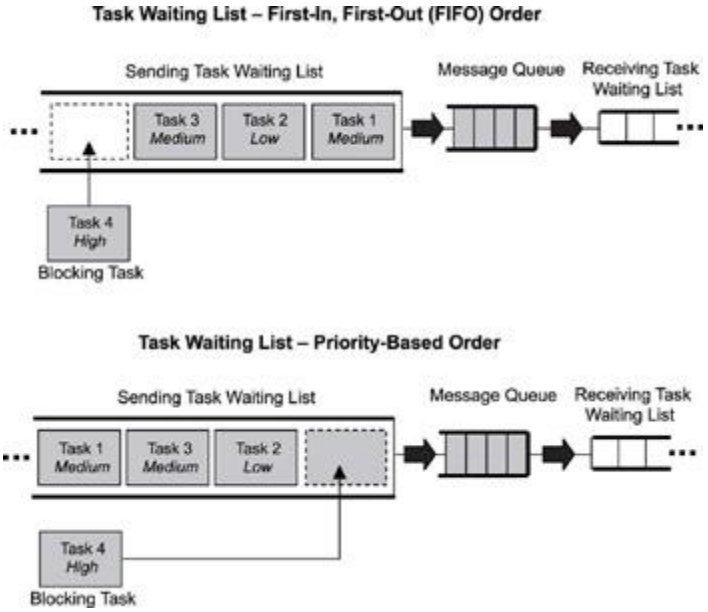


Figure 7.5: FIFO and priority-based task-waiting lists.

In the case of a task set to block forever when sending a message, the task blocks until a message queue element becomes free (e.g., a receiving task takes a message out of the queue). In the case of a task set to block for a specified time, the task is unblocked if either a queue element becomes free or the timeout expires, in which case an error is returned.

Receiving Messages

As with sending messages, tasks can receive messages with different blocking policies—the same way as they send them—with a policy of not blocking, blocking with a timeout, or blocking forever. Note, however, that in this case, the blocking occurs due to the message queue being empty, and the receiving tasks wait in either a FIFO or prioritybased order. The diagram for the receiving tasks is similar to Figure 5, except that the blocked receiving tasks are what fills the task list.

For the message queue to become full, either the receiving task list must be empty or the rate at which messages are posted in the message queue must be greater than the rate at which messages are removed. Only when the message queue is full does the task-waiting list for sending tasks start to fill. Conversely, for the task-waiting list for receiving tasks to start to fill, the message queue must be empty.

Messages can be read from the head of a message queue in two different ways:

- destructive read, and
- non-destructive read.

In a destructive read, when a task successfully receives a message from a queue, the task permanently removes the message from the message queue's storage buffer. In a non-destructive read, a receiving task peeks at the message at the head of the queue without removing it. Both ways of reading a message can be useful; however, not all kernel implementations support the non-destructive read.

Some kernels support additional ways of sending and receiving messages. One way is the example of peeking at a message. Other kernels allow broadcast messaging, explained later in this chapter.

3 Obtaining Message Queue Information

Obtaining message queue information can be done from an application by using the operations listed in Table 3.

Table 3: Obtaining message queue information operations.

Operation	Description
Show queue info	Gets information on a message queue
Show queue's task-waiting list	Gets a list of tasks in the queue's task-waiting list

Different kernels allow developers to obtain different types of information about a message queue, including the message queue ID, the queuing order used for blocked tasks (FIFO or priority-based), and the number of messages queued. Some calls might even allow developers to get a full list of messages that have been queued up.

As with other calls that get information about a particular kernel object, be careful when using these calls. The information is dynamic and might have changed by the time it's viewed. These types of calls should only be used for debugging purposes.

Typical Message Queue Use

The following are typical ways to use message queues within an application:

- non-interlocked, one-way data communication,
- interlocked, one-way data communication,
- interlocked, two-way data communication, and
- broadcast communication.

Note that this is not an exhaustive list of the data communication patterns involving message queues. The following sections discuss each of these simple cases.

1 Non-Interlocked, One-Way Data Communication

One of the simplest scenarios for message-based communications requires a sending task (also called the message source), a message queue, and a receiving task (also called a message sink), as illustrated in Figure 6.

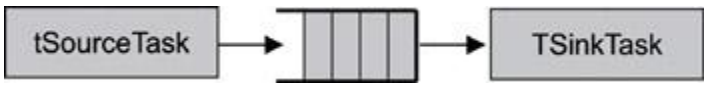


Figure 6: Non-interlocked, one-way data communication.

This type of communication is also called non-interlocked (or loosely coupled), one-way data communication. The activities of tSourceTask and tSinkTask are not synchronized. TSourceTask simply sends a message; it does not require acknowledgement from tSinkTask.

2. Interlocked, One-Way Data Communication

In some designs, a sending task might require a handshake (acknowledgement) that the receiving task has been successful in receiving the message. This process is called interlocked communication, in which the sending task sends a message and waits to see if the message is received.

This requirement can be useful for reliable communications or task synchronization. For example, if the message for some reason is not received correctly, the sending task can resend it. Using interlocked communication can close a synchronization loop. To do so, you can construct a continuous loop in which sending and receiving tasks operate in lockstep with each other. An example of one-way, interlocked data communication is illustrated in Figure 7.

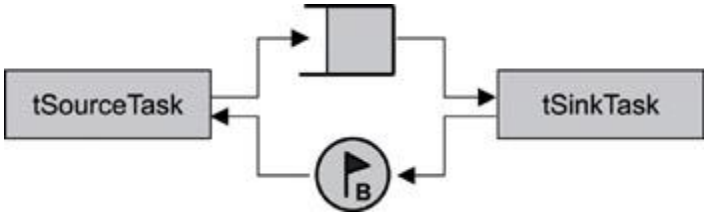


Figure 7: Interlocked, one-way data communication.

In this case, tSourceTask and tSinkTask use a binary semaphore initially set to 0 and a message queue with a length of 1 (also called a mailbox). tSourceTask sends the message to the message queue and blocks on the binary semaphore. tSinkTask receives the message and increments the binary semaphore. The semaphore that has just been made available wakes up tSourceTask. tSourceTask, which executes and posts another message into the message queue, blocking again afterward on the binary semaphore.

3. Interlocked, Two-Way Data Communication

Sometimes data must flow bidirectionally between tasks, which is called interlocked, two-way data communication (also called full-duplex or tightly coupled communication). This form of communication can be useful when designing a client/server-based system. A diagram is provided in Figure 8.

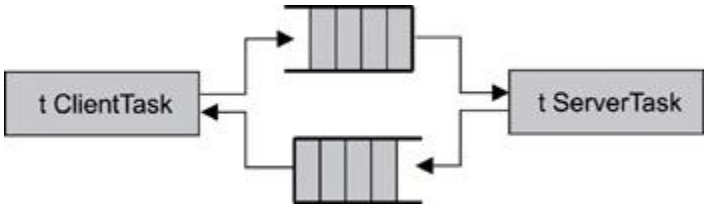


Figure 8: Interlocked, two-way data communication.

Two separate message queues are required for full-duplex communication. If any kind of data needs to be exchanged, message queues are required; otherwise, a simple semaphore can be used to synchronize acknowledgement.

4. Broadcast Communication

Some message-queue implementations allow developers to broadcast a copy of the same message to multiple tasks, as shown in Figure 9.

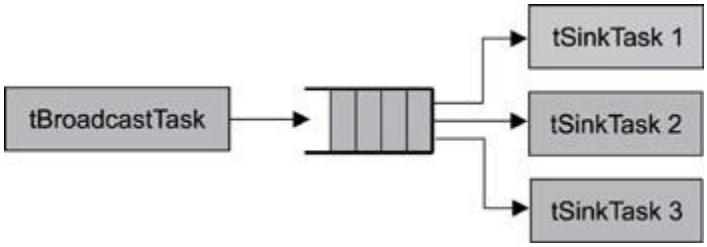


Figure 9: Broadcasting messages.

Message broadcasting is a one-to-many-task relationship. tBroadcastTask sends the message on which multiple tSink-Task are waiting.