LECTURE NOTES

ON

EMBEDDED SYSTEMS DESIGN

IV B. Tech I semester (JNTUH-R15)

Faculty Members

Mr. N Paparao
Assistant Professor
Mr. S Lakshmanachari
Assistant Professor
Mr. MD Khadir
Assistant Professor

ELECTRONICS AND COMMUNICATION ENGINEERING

INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
DUNDIGAL, HYDERABAD - 500 043

Embedded Systems Design
LECTURE NOTES

SYLLABUS:

Unit-I
Introduction to Embedded Systems:
Definition of Embedded System, Embedded Systems Vs General Computing Systems, History of Embedded Systems, Classification, Major Application Areas, Purpose of Embedded Systems, Characteristics and Quality Attributes of Embedded Systems.

UNIT-II
Typical Embedded System:
Core of the Embedded System: General Purpose and Domain Specific Processors, ASICs, PLDs, Commercial Off-The-Shelf Components (COTS), Memory: ROM, RAM, Memory according to the type of Interface, Memory Shadowing, Memory selection for Embedded Systems, Sensors and Actuators, Communication Interface: Onboard and External Communication Interfaces.

UNIT-III
Embedded Firmware:
Reset Circuit, Brown-out Protection Circuit, Oscillator Unit, Real Time Clock, Watchdog Timer, Embedded Firmware Design Approaches and Development Languages.

UNIT-IV
RTOS Based Embedded System Design:
Operating System Basics, Types of Operating Systems, Tasks, Process and Threads, Multiprocessing and Multitasking, Task Scheduling.

UNIT- V
Task Communication:
Shared Memory, Message Passing, Remote Procedure Call and Sockets, Task Synchronization: Task Communication Synchronization Issues, Task Synchronization Techniques, Device Drivers, How to Choose an RTOS.

TEXT BOOKS:
1. Introduction to Embedded Systems - Shibu K.V, Mc Graw Hill.

REFERENCE BOOKS:
Embedded Systems - Raj Kamal, TMH.
Embedded System Design - Frank Vahid, Tony Givargis, John Wiley.
Embedded Systems – Lyla, Pearson, 2013
An Embedded Software Primer - David E. Simon, Pearson Education.

UNIT -I

Introduction to Embedded systems

INTRODUCTION:

- An embedded system is an electronic system, which includes a single chip microcomputers (Microcontrollers) like the ARM or Cortex or Stellaris LM3S1968.
- It is configured to perform a specific dedicated application.
- An embedded system is some combination of computer hardware and software, either fixed in capability or programmable, that is designed for a specific function or for specific functions within a larger system.
- Here the microcomputer is embedded or hidden inside the system. Every **embedded microcomputer** system accepts inputs, performs computations, and generates outputs and runs in "real time."

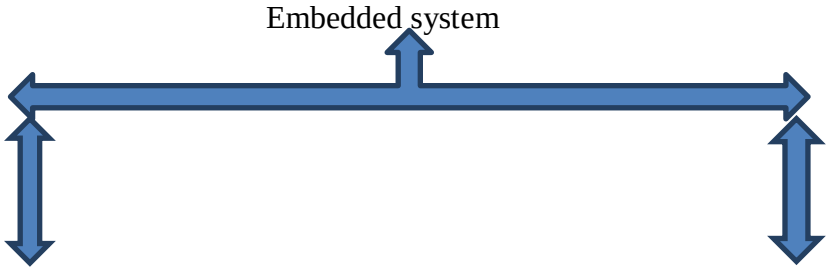
Ex: Cell phone, Digital camera, Microwave Oven, MP3 player, Portable digital assistant & automobile antilock brake system Industrial machines, agricultural and process industry devices, automobiles, medical equipment, household appliances, airplanes, vending machines and toys as well as mobile devices are all possible locations for an embedded system.etc.

Characteristics of an Embedded System: The important characteristics of an embedded system are

- Speed (bytes/sec) : Should be high speed
- Power (watts) : Low power dissipation
- Size and weight : As far as possible small in size and low weight
- Accuracy (% error) : Must be very accurate
- Adaptability: High adaptability and accessibility.
- Reliability: Must be reliable over a long period of time.

So, 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. The system must be easily adaptable for different situations.

CATEGORIES OF EMBEDDED SYSTEMS: Embedded systems can be classified into the following 4 categories based on their functional and performance requirements.



- Stand alone embedded systems — small scale embedded system
- Real time embedded system — medium scale embedded s/m
    a) Hard real time E.S
    b) Soft Real time E.S
- Networked embedded system — large scale embedded system
- Mobile embedded system .

Stand alone Embedded systems:

A stand-alone embedded system works by itself. It is a self-contained device which does not require any host system like a computer. It takes either digital or analog inputs from its input ports, calibrates, converts, and processes the data, and outputs the resulting data to its attached output device, which either displays data, or controls and drives the attached devices.

EX: Temperature measurement systems, Video game consoles, MP3 players, digital cameras, and microwave ovens are the examples for this category.

Real-time embedded systems:

An embedded system which gives the required output in a specified time or which strictly follows the time deadlines for completion of a task is known as a Real time system. i.e. a Real Time system , in addition to functional correctness, also satisfies the time constraints .

There are two types of Real time systems. (i) Soft real time system and (ii) Hard real time system.

- **Soft Real-Time system:** A Real time system in which, the violation of time constraints will cause only the degraded quality, but the system can continue to operate is known as a Soft real time system. In soft real-time systems, the design focus is to offer a guaranteed bandwidth to each real-time task and to distribute the resources to the tasks.

  Ex: A Microwave Oven, washing machine, TV remote etc.

- **Hard Real-Time system:** A Real time system in which, the violation of time constraints will cause critical failure and loss of life or property damage or catastrophe is known as a Hard Real time system.

These systems usually interact directly with physical hardware instead of through a human being .The hardware and software of hard real-time systems must allow a worst case execution (WCET) analysis that guarantees the execution be completed within a strict deadline. The chip selection and RTOS selection become important factors for hard real-time system design.

Ex: Deadline in a missile control embedded system , Delayed alarm during a Gas leakage , car airbag control system , A delayed response in pacemakers ,Failure in RADAR functioning etc.

Networked embedded systems:

The networked embedded systems are related to a network with network interfaces to access the resources. The connected network can be a Local Area Network (LAN) or a Wide Area Network (WAN), or the Internet. The connection can be either wired or wireless.

The networked embedded system is the fastest growing area in embedded systems applications. The embedded web server is such a system where all embedded devices are connected to a web server and can be accessed and controlled by any web browser.

Ex: A home security system is an example of a LAN networked embedded system where all sensors (e.g. motion detectors, light sensors, or smoke sensors) are wired and running on the TCP/IP protocol.

Mobile Embedded systems:

The portable embedded devices like mobile and cellular phones, digital cameras, MP3 players, PDA (Personal Digital Assistants) are the example for mobile embedded systems. The basic limitation of these devices is the limitation of memory and other resources.

Based on the performance of the Microcontroller they are also classified into (i) Small scaled embedded system (ii) Medium scaled embedded system and (iii) Large scaled embedded system.

Small scaled embedded system:

An embedded system supported by a single 8–16 bit Microcontroller with on-chip RAM and ROM designed to perform simple tasks is a Small scale embedded system.

Medium scaled embedded system:

An embedded system supported by 16–32 bit Microcontroller /Microprocessor with external RAM

and ROM that can perform more complex operations is a Medium scale embedded system.

Large scaled embedded system:

An embedded system supported by 32-64 bit multiple chips which can perform distributed jobs is considered as a Large scale embedded system.

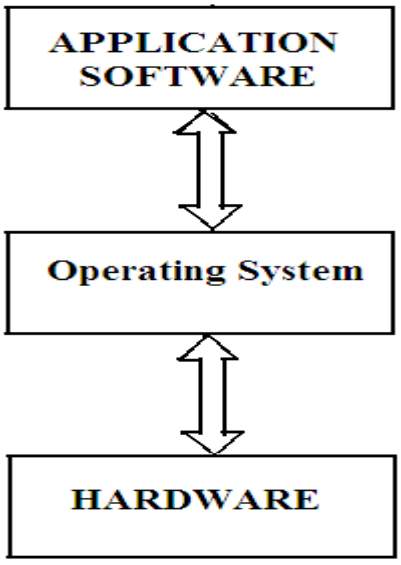
Application Areas of Embedded Systems:

The embedded systems have a huge variety of application domains which varies from very low cost to very high cost and from daily life consumer electronics to industry automation equipments, from entertainment devices to academic equipments, and from medical instruments to aerospace and weapon control systems. So, the embedded systems span all aspects of our modern life. The following table gives the various applications of embedded systems.

Embedded System	Application
Home Appliances	Dishwasher, washing machine, microwave, Top-set box, security system , HVAC system, DVD, answering machine, garden sprinkler systems etc..
Office Automation	Fax, copy machine, smart phone system, modern, scanner, printers.
Security	Face recognition, finger recognition, eye recognition, building security system , airport security system, alarm system.
Academia	Smart board, smart room, OCR, calculator, smart cord.
Instrumentation	Signal generator, signal processor, power supplier,Process instrumentation,
Telecommunication	uter, hub, cellular phone, IP phone, web camera
Automobile	Fuel injection controller, anti-locking brake system, air-bag system, GPS, cruise control.
Entertainment	P3, video game, Mind Storm, smart toy.
Aerospace	Navigation system, automatic landing system, flight attitude controller, space explorer, space robotics.
Industrial automation	Assembly line, data collection system, monitoring systems on pressure, voltage, current, temperature, hazard detecting system, industrial robot.
Personal	PDA, iPhone, palmtop, data organizer.
Medical	CT scanner, ECG , EEG , EMG ,MRI, Glucose monitor, blood pressure monitor, medical diagnostic device.
Banking & Finance	ATM, smart vendor machine, cash register ,Share market
Miscellaneous:	Elevators, tread mill, smart card, security door etc.

Overview of embedded systems architecture:

- Every embedded system consists of customer-built hardware components supported by a Central Processing Unit (CPU), which is the heart of a microprocessor (µP) or microcontroller (µC).
- A microcontroller is an integrated chip which comes with built-in memory, I/O ports, timers, and other components.
- Most embedded systems are built on microcontrollers, which run faster than a custom-built system with a microprocessor, because all components are integrated within a single chip.
- Operating system plays an important role in most of the embedded systems. But all the embedded systems do not use the operating system.
- The systems with high end applications only use operating system. To use the operating system the embedded system should have large memory capability.
- So, this is not possible in low end applications like remote systems, digital cameras, MP3 players, robot toys etc.
- The architecture of an embedded system with OS can be denoted by layered structure as shown below.
- The OS will provide an interface between the hardware and application software.



In the case of embedded systems with OS, once the application software is loaded into memory it will run the application without any host system.

Coming to the hardware details of the embedded system, it consists of the following important blocks.

- CPU(Central Processing Unit)

- RAM and ROM
- I/O Devices
- Communication Interfaces
- Sensors etc. (Application specific circuitry)

This hardware architecture can be shown by the following block diagram.

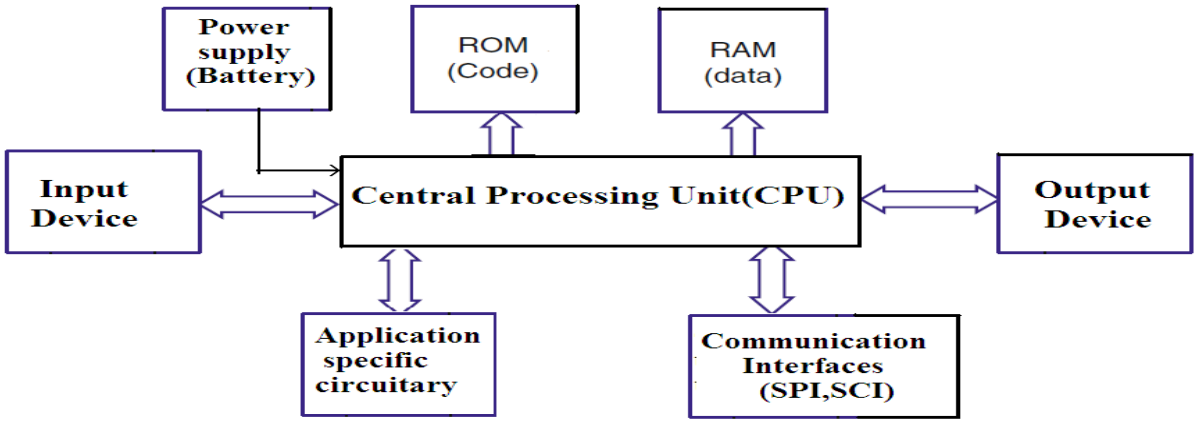
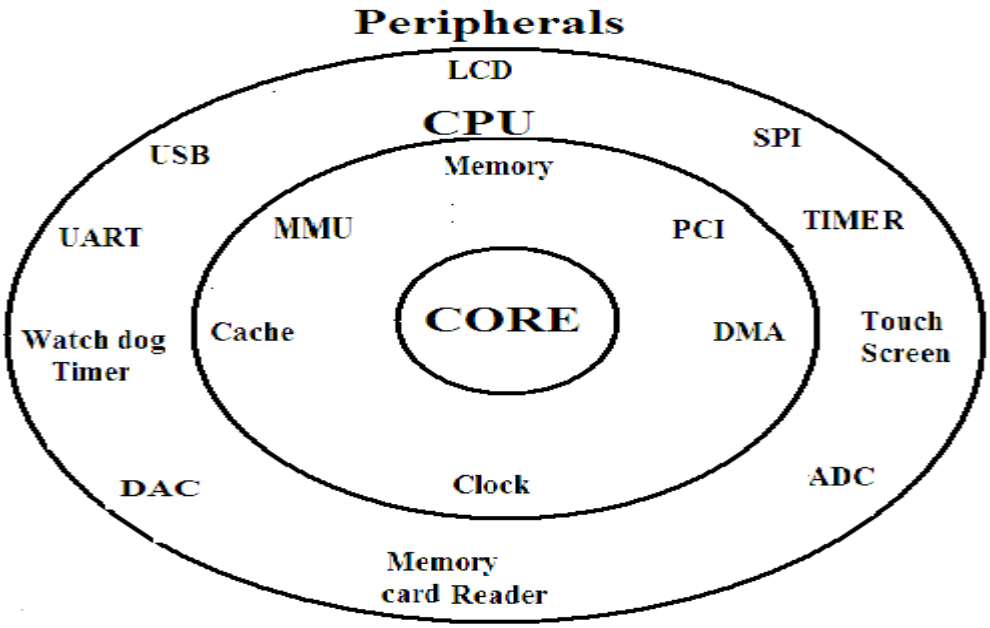


Fig: hardware architecture of embedded system

Central Processing Unit:

- A CPU is composed of an Arithmetic Logic Unit (ALU), a Control Unit (CU), and many internal registers that are connected by buses.
- The ALU performs all the mathematical operations (Add, Sub, Mul, Div), logical operations (AND, OR), and shifting operations within CPU.
- The timing and sequencing of all CPU operations are controlled by the CU, which is actually built of many selection circuits including latches and decoders. The CU is responsible for directing the flow of instruction and data within the CPU and continuously running program instructions step by step.
- The CPU works in a cycle of fetching an instruction, decoding it, and executing it, known as the fetch-decode-execute cycle.
- For embedded system design, many factors impact the CPU selection, e.g., the maximum size (number of bits) in a single operand for ALU (8, 16, 32, 64 bits), and CPU clock frequency for timing tick control, i.e. the number of ticks (clock cycles) per second in measures of MHz
- CPU contains the core and the other components which support the core to execute programs. Peripherals are the components which communicate with other systems or physical world (Like ports, ADC,DAC, Watch dog Timers etc.). The core is separated from other components by the system bus.

- The CPU in the embedded system may be a general purpose processor like a microcontroller or a special purpose processor like a DSP (Digital signal processor). But any CPU consists of of an Arithmetic Logic Unit (ALU), a Control Unit (CU), and many internal registers that are connected by buses. The ALU performs all the mathematical operations (Add, Sub, Mul, Div), logical operations (AND, OR), and shifting operations within CPU.

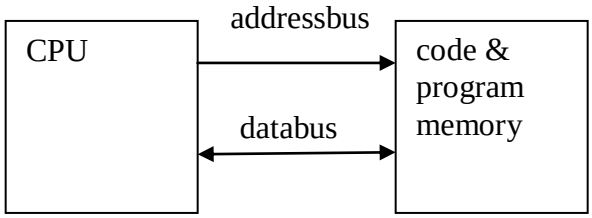


- There are many internal registers in the CPU.
- The accumulator (A) is a special data register that stores the result of ALU operations. It can also be used as an operand. The Program Counter (PC) stores the memory location of the next instruction to be executed. The Instruction Register (IR) stores the current machine instruction to be decoded and executed.
- The Data Buffer Registers store the data received from the memory or the data to be sent to memory. The Data Buffer Registers are connected to the data bus.
- The Address Register stores the memory location of the data to be accessed (get or set). The Address Register is connected to the address bus.
- In an embedded system, the CPU may never stop and run forever .The CPU works in a cycle of fetching an instruction, decoding it, and executing it, known as the fetch-decode-execute cycle. The cycle begins when an instruction is fetched from a memory location pointed to by the PC to the IR via the data bus.

When data and code lie in different memory blocks, then the architecture is referred as **Harvard architecture**. In case data and code lie in the same memory block, then the architecture is referred as **Von Neumann architecture**.

Von Neumann Architecture:

The Von Neumann architecture was first proposed by a computer scientist John von Neumann. In this architecture, one data path or bus exists for both instruction and data. As a result, the CPU does one operation at a time. It either fetches an instruction from memory, or performs read/write operation on data. So an instruction fetch and a data operation cannot occur simultaneously, sharing a common bus.



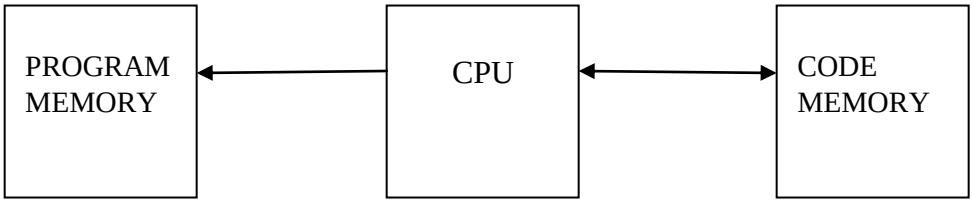
Von-Neumann architecture supports simple hardware. It allows the use of a single, sequential memory. Today's processing speeds vastly outpace memory access times, and we employ a very fast but small amount of memory (cache) local to the processor.

Harvard Architecture:

The Harvard architecture offers separate storage and signal buses for instructions and data. This architecture has data storage entirely contained within the CPU, and there is no access to the instruction storage as data. Computers have separate memory areas for program instructions and data using internal data buses, allowing simultaneous access to both instructions and data.

Programs needed to be loaded by an operator; the processor could not boot itself. In a Harvard architecture, there is no need to make the two memories share properties.

Von-Neumann Architecture vs Harvard Architecture:



The following points distinguish the Von Neumann Architecture from the Harvard Architecture.

Von-Neumann Architecture	Harvard Architecture
Single memory to be shared by both code and data.	Separate memories for code and data.
Processor needs to fetch code in a separate clock cycle and data in another clock cycle. So it requires two clock cycles.	Single clock cycle is sufficient, as separate buses are used to access code and data.
Higher speed, thus less time consuming.	Slower in speed, thus more time-consuming.
Simple in design.	Complex in design.

CISC and RISC:

- CISC is a Complex Instruction Set Computer. It is a computer that can address a large number of instructions.
- In the early 1980s, computer designers recommended that computers should use fewer instructions with simple constructs so that they can be executed much faster within the CPU without having to use memory. Such computers are classified as Reduced Instruction Set Computer or RISC.

CISC vs RISC:

The following points differentiate a CISC from a RISC −

CISC	RISC
Larger set of instructions. Easy to program	Smaller set of Instructions. Difficult to program.
Simpler design of compiler, considering larger set of instructions.	Complex design of compiler.
Many addressing modes causing complex instruction formats.	Few addressing modes, fix instruction format.
Instruction length is variable.	Instruction length varies.
Higher clock cycles per second.	Low clock cycle per second.
Emphasis is on hardware.	Emphasis is on software.
Control unit implements large instruction set using micro-program unit.	Each instruction is to be executed by hardware.
Slower execution, as instructions are to be read from memory and decoded by the decoder unit.	Faster execution, as each instruction is to be executed by hardware.

Pipelining is not possible.	Pipelining of instructions is possible, considering single clock cycle.
Mainly used in normal pc's, workstations & servers.	Mainly used for real time applications.

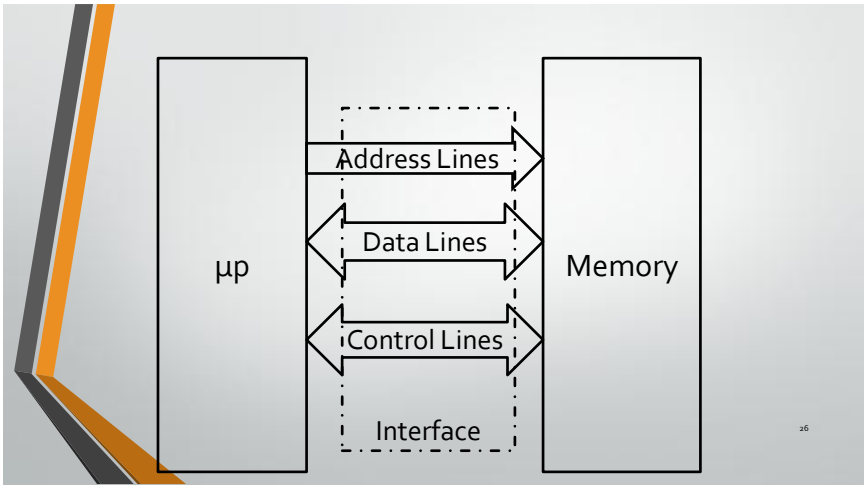
Memory:

- Embedded system memory can be either on-chip or off-chip.
- On chip memory access is much fast than off-chip memory, but the size of on-chip memory is much smaller than the size of off-chip memory.
- Usually, it takes at least two I/O ports as external address lines plus a few control lines such as R/W and ALE control lines to enable the extended memory. Generally the data is stored in RAM and the program is stored in ROM.
- The ROM, EPROM, and Flash memory are all read-only type memories often used to store code in an embedded system.
- The embedded system code does not change after the code is loaded into memory.
- The ROM is programmed at the factory and cannot be changed over time.
- The newer microcontrollers come with EPROM or Flash instead of ROM.
- Most microcontroller development kits come with EPROM as well.
- EPROM and Flash memory are easier to rewrite than ROM. EPROM is an Erasable Programmable ROM in which the contents can be field programmed by a special burner and can be erased by a UV light bulb.
- The size of EPROM ranges up to 32kb in most embedded systems.
- Flash memory is an Electrically EPROM which can be programmed from software so that the developers don't need to physically remove the EPROM from the circuit to re-program it.
- It is much quicker and easier to re-write Flash than other types of EPROM.
- When the power is on, the first instruction in ROM is loaded into the PC and then the CPU fetches the instruction from the location in the ROM pointed to by the PC and stores it in the IR to start the continuous CPU fetch and execution cycle. The PC is advanced to the address of the next instruction depending on the length of the current instruction or the destination of the Jump instruction.
- The memory is divided into Data Memory and Code Memory.

- Most of data is stored in Random Access Memory (RAM) and code is stored in Read Only Memory (ROM).
- This is due to the RAM constraint of the embedded system and the memory organization.
- The RAM is readable and writable, faster access and more expensive volatile storage, which can be used to store either data or code.
- Once the power is turned off, all information stored in the RAM will be lost.
- The RAM chip can be SRAM (static) or DRAM (dynamic) depending on the manufacturer. SRAM is faster than DRAM, but is more expensive.

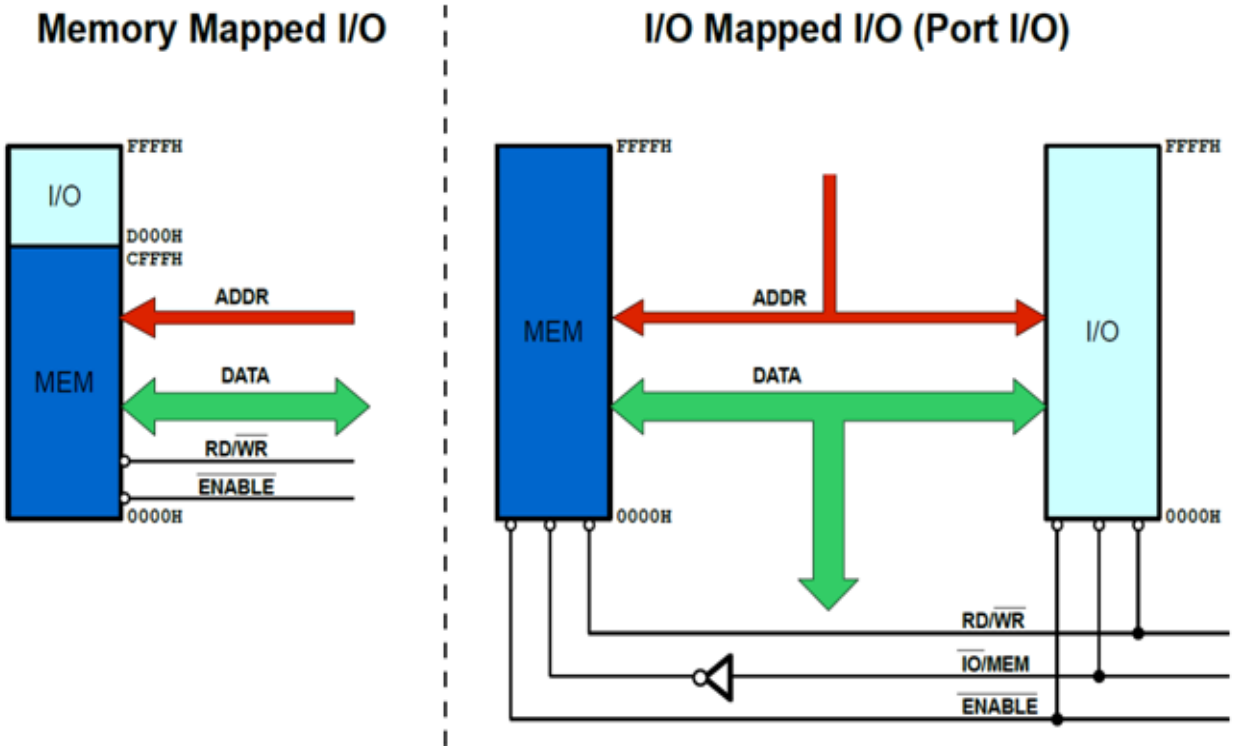
I/O Ports:

- The I/O ports are used to connect input and output devices. The common input devices for an embedded system include keypads, switches, buttons, knobs, and all kinds of sensors (light, temperature, pressure, etc).
- The output devices include Light Emitting Diodes (LED), Liquid Crystal Displays (LCD), printers, alarms, actuators, etc. Some devices support both input and output, such as communication interfaces including Network Interface Cards (NIC), modems, and mobile phones.



In parallel I/O have two types of interfacing

1. Memory mapped I/O
2. I/O mapped I/O



Memory Mapped IO

- IO is treated as memory.
- 16-bit addressing.
- More Decoder Hardware.
- Can address 2^{16}=64k locations.
- Less memory is available.

IO Mapped IO

- IO is treated IO.
- 8- bit addressing.
- Less Decoder Hardware.
- Can address 2^{8}=256 locations.
- Whole memory address space is available.

Memory Mapped IO	IO Mapped IO
• Memory Instructions are used. • Memory control signals are used. • Arithmetic and logic operations can be performed on data. • Data transfer b/w register and IO.	• Special Instructions are used like IN, OUT. • Special control signals are used. • Arithmetic and logic operations can not be performed on data. • Data transfer b/w accumulator and IO.

39

Communication Interfaces:

- To transfer the data or to interact with other devices, the embedded devices are provided the various communication interfaces like RS232, RS422, RS485 ,USB, SPI(Serial Peripheral Interface) ,SCI (Serial Communication Interface) ,Ethernet etc.

Application Specific Circuitry:

- The embedded system sometimes receives the input from a sensor or actuator. In such situations certain signal conditioning circuitry is needed. This hardware circuitry may contain ADC, Op-amps, DAC etc. Such circuitry will interact with the embedded system to give correct output.

ADC & DAC:

- Many embedded system application need to deal with non-digital external signals such as electronic voltage, music or voice, temperature, pressures, and many other signals in the analog form.
- The digital computer does not understand these data unless they are converted to digital formats. The ADC is responsible for converting analog values to binary digits.
- The DAC is responsible for outputting analog signals for automation controls such as DC motor or HVDC furnace control.

In addition to these peripherals, an embedded system may also have sensors, Display modules like LCD or Touch screen panels, Debug ports certain communication peripherals like I^2C, SPI, Ethernet, CAN, USB for high speed data transmission. Now a days various sensors are also becoming an important part in the design of real time embedded systems. Sensors like temperature sensors, light sensors, PIR sensors, gas sensors are widely used in application specific circuitry.

Address bus and data bus:

- According to computer architecture, a bus is defined as a system that transfers data between hardware components of a computer or between two separate computers.
- Initially, buses were made up using electrical wires, but now the term bus is used more broadly to identify any physical subsystem that provides equal functionality as the earlier electrical buses.
- Computer buses can be parallel or serial and can be connected as multidrop, daisy chain or by switched hubs.
- System bus is a single bus that helps all major components of a computer to communicate with each other.
- It is made up of an address bus, data bus and a control bus. The data bus carries the data to be stored, while address bus carries the location to where it should be stored.

Address Bus

- Address bus is a part of the computer system bus that is dedicated for specifying a physical address.
- When the computer processor needs to read or write from or to the memory, it uses the address bus to specify the physical address of the individual memory block it needs to access (the actual data is sent along the data bus).
- More correctly, when the processor wants to write some data to the memory, it will assert the write signal, set the write address on the address bus and put the data on to the data bus.
- Similarly, when the processor wants to read some data residing in the memory, it will assert the read signal and set the read address on the address bus.
- After receiving this signal, the memory controller will get the data from the specific memory block (after checking the address bus to get the read address) and then it will place the data of the memory block on to the data bus.

The size of the memory that can be addressed by the system determines the width of the data bus and vice versa. For example, if the width of the address bus is 32 bits, the system can address 232 memory blocks (that is equal to 4GB memory space, given that one block holds 1 byte of data).

Data Bus

- A data bus simply carries data. Internal buses carry information within the processor, while external buses carry data between the processor and the memory.
- Typically, the same data bus is used for both read/write operations. When it is a write operation, the processor will put the data (to be written) on to the data bus.
- When it is the read operation, the memory controller will get the data from the specific memory block and put it in to the data bus.

What is the difference between Address Bus and Data Bus?

- Data bus is bidirectional, while address bus is unidirectional. That means data travels in both directions but the addresses will travel in only one direction.
- The reason for this is that unlike the data, the address is always specified by the processor. The width of the data bus is determined by the size of the individual memory block, while the width of the address bus is determined by the size of the memory that should be addressed by the system.

Power supply:

- Most of the embedded systems now days work on battery operated supplies.
- Because low power dissipation is always required. Hence the systems are designed to work with batteries.

Clock:

- The clock is used to control the clocking requirement of the CPU for executing instructions and the configuration of timers. For ex: the 8051 clock cycle is $(1/12)10^{-6}$ second (1/12µs) because the clock frequency is 12MHz. A simple 8051 instruction takes 12 cycles (1ms) to complete. Of course, some multi-cycle instructions take more clock cycles.
- A timer is a real-time clock for real-time programming. Every timer comes with a counter which can be configured by programs to count the incoming pulses. When the counter overflows (resets to zero) it will fire a timeout interrupt that triggers predefined actions. Many time delays can be generated by timers. For example ,a timer counter configured to 24,000 will trigger the timeout signal in 24000x 1/12µs = 2ms.
- In addition to time delay generation, the timer is also widely used in the real-time embedded system to schedule multiple tasks in multitasking programming. The watchdog timer is a special timing device that resets the system after a preset time delay in case of system anomaly. The watchdog starts up automatically after the system power up.
- One need to reboot the PC now and then due to various faults caused by hardware or software. An embedded system cannot be rebooted manually, because it has been embedded into its system. That is why many microcontrollers come with an on-chip watchdog timer which can be configured just like the counter in the regular timer. After a system gets stuck (power supply voltage out of range or regular timer does not issue timeout after reaching zero count) the watchdog eventually will restart the system to bring the system back to a normal operational condition.

Application Specific software:

It sits above the O.S. The application software is developed according to the features of the development tools available in the OS.

These development tools provide the function calls to access the services of the OS. These function calls include, creating a task ,to read the data from the port and write the data to the memory etc.

The various function calls provided by an operating system are

i. To create ,suspend and delete tasks.

ii. To do task scheduling to providing real time environment.

iii. To create inter task communication and achieve the synchronization between tasks.

iv. To access the I/O devices.

v. To access the communication protocol stack .

The designer develops the application software based on these function calls.

Recent trends in Embedded systems : With the fast developments in semiconductor industry and VLSI technology ,one can find tremendous changes in the embedded system design in terms of processor speed , power , communication interfaces including network capabilities and software developments like operating systems and programming languages etc.

- **Processor speed and Power :** With the advancements in processor technology ,the embedded systems are now days designed with 16,32 bit processors which can work in real time environment. These processors are able to perform high speed signal processing activities which resulted in the development of high definition communication devices like 3G mobiles etc.Also the recent developments in VLSI technology has paved the way for low power battery operated devices which are very handy and have high longevity. Also , the present day embedded systems are provided with higher memory capabilities ,so that most of them are based on tiny operating systems like android etc.
- **Communication interfaces :** Most of the present day embedded systems are aimed at internet based applications. So,the communication interfaces like Ethernet, USB, wireless LAN etc.have become very common resources in almost all the embedded systems. The developments in memory technologies also helped in porting the TCP/IP protocol stack and the HTTP server software on to the embedded systems. Such embedded systems can provide a link between any two devices anywhere in the globe.
- **Operating systems :** With recent software developments ,there is a considerable growth in the availability of operating systems for embedded systems. Mainly new operating systems are developed which can be used in real time applications. There are both commercial RTOSes like Vx

Works , QNX,WIN-CE and open source RTOSes like RTLINUX etc. The Android OS in mobiles has revolutionized the embedded industry.

- **Programming Languages :** There is also a remarkable development in the programming languages. Languages like C++, Java etc. are now widely used in embedded application programming. For example by having the Java virtual machine in a mobile phones ,one can download Java applets from a server and can be executed on your mobile.

In addition to these developments, now a days we also find new devices like ASICs and FPGAs in the embedded system market. These new hardware devices are popular as programmable devices and reconfigurable devices.

Msp430 introduction:

- The MSP430 was introduced in the late 1990s. It is a particularly straightforward 16-bit processor with a von Neumann architecture, designed for low-power applications.
- Both the address and data buses are 16 bits wide. The registers in the CPU are also all 16 bits wide and can be used interchangeably for either data or addresses.
- This makes the MSP430 simpler than an 8-bit processor with 16-bit addresses. Such a processor must use its general-purpose registers in pairs for addresses or provide separate, wider registers.
- In many ways, the MSP430 fits between traditional 8- and 16-bit processors.
- The 16-bit data bus and registers clearly define it as a 16-bit processor. On the other hand, it can address only 216 =64KB of memory.
- The MSP430 has 16 registers in its CPU, which enhances efficiency because they can be used for local variables, parameters passed to subroutines, and either addresses or data.
- This is a typical feature of a RISC, but unlike a "pure" RISC, it can perform arithmetic directly on values in main memory.
- Microcontrollers typically spend much of their time on such operations. The MSP430 is the simplest microcontroller in Texas Instrumentations.

Several features make the MSP430 suitable for low-power and portable applications:

Features of MSP430:

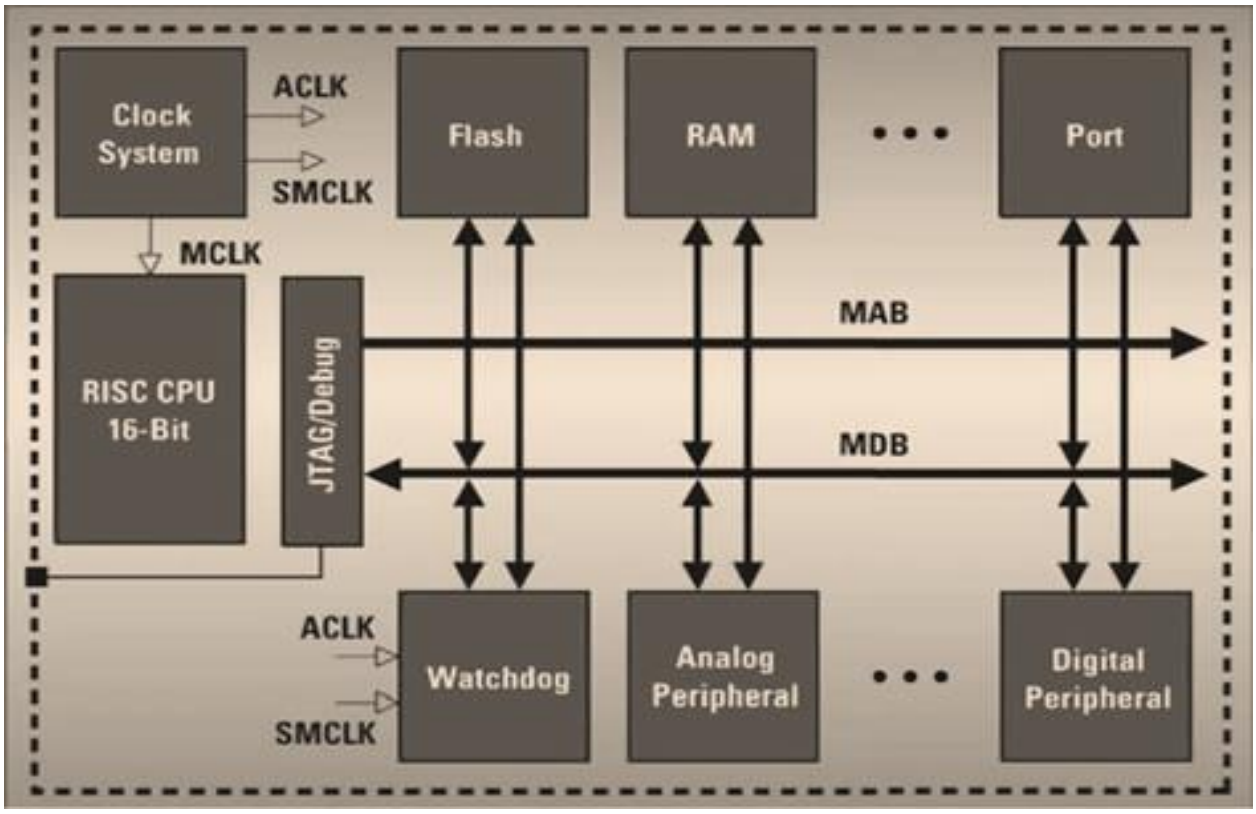
- The CPU is small and efficient, with a large number of registers.

- It is extremely easy to put the device into a low-power mode. No special instruction is needed.
- The mode is controlled by bits in the status register. There are several low-power modes, depending on how much of the device should remain active and how quickly it should return to full-speed operation.
- There is a wide choice of clocks. Typically, a low-frequency watch crystal runs continuously at 32KHz and is used to wake the device periodically. The CPU is clocked by an internal, digitally controlled oscillator (DCO), which restarts in less than 1us in the latest devices.
- Therefore the MSP430 can wake from a standby mode rapidly, perform its tasks, and return to a low-power mode.
- A wide range of peripherals is available, many of which can run autonomously without the CPU for most of the time.
- Many portable devices include liquid crystal displays, which the MSP430 can drive directly.
- Some MSP430 devices are classed as application-specific standard products (ASSPs) and contain specialized analog hardware for various types of measurement.

- Ultra-low-power (ULP) architecture and flexible clock system extend battery life
- Low power consumption:
    - μ A for RAM data Retention,
    - 0.8 μ A for RTC mode operation
    - 250 μA /MIPS at active operation.
- Low operation voltage (from 1.8 V to 3.6 V).
    - Zero-power Brown-Out Reset (BOR).
- Enhanced libraries to benefit several applications such as capacitive touch, metering metrology, low power design and debugging
- Extensive interrupt capability relieves need for polling
- Flexible and powerful processing capabilities:
- Seven source-address modes
- Four destination-address modes
- Only 27 core instructions
- Prioritized, nested interrupts
- Large register file
- Efficient table processing
- Fast hex-to-decimal conversion

ARCHITECTURE OF MSP430:

The Inside View—Functional Block Diagram Figure shows a block diagram of the F2013.



- These are its main features: On the left is the CPU and its supporting hardware, including the clock generator. The emulation, JTAG interface and Spy-Bi-Wire are used to communicate with a desktop computer when downloading a program and for debugging.
- The main blocks are linked by the memory address bus (MAB) and memory data bus (MDB).
- These devices have flash memory, 1KB in the F2003 or 2KB in the F2013, and 128 bytes of RAM.
- Six blocks are shown for peripheral functions (there are many more in larger devices).
- All MSP430s include input/output ports, Timer_A, and a watchdog timer, although the details differ.
- The universal serial interface (USI) and sigma–delta analog-to-digital converter (SD16_A) are particular features of this device.
- The brownout protection comes into action if the supply voltage drops to a dangerous level. Most devices include this but not some of the MSP430x1xx family.
- There are ground and power supply connections. Ground is labeled VSS and is taken to define 0V.
- The supply connection is VCC. For many years, the standard for logic was VCC =+5V but most devices now work from lower voltages and a range of 1.8–3.6V is specified for the F2013.

- The performance of the device depends on VCC. For example, it is unable to program the flash memory if VCC < 2.2V and the maximum clock frequency of 16MHz is available only if VCC ≥3.3V.
- TI uses a quaint notation for the power connections. The S stands for the source of a field-effect transistor, while the C stands for the collector of a bipolar junction transistor, a quite different device.
- The MSP430, like most modern integrated circuits, is built using complementary metal–oxide–silicon (CMOS) technology and field-effect transistors. I doubt if it contains any bipolar junction transistors except possibly in some of the analog peripherals.
- There is only one pair of address and data buses, as expected with a von Neumann architecture. Some addresses must therefore point to RAM and some to flash, so it is a good idea to explore the memory map next.

Clock Generator:

- Clocks for microcontrollers used to be simple. Usually a crystal with a frequency of a few MHz would be connected to two pins.
- It would drive the CPU directly and was typically divided down by a factor of 2 or 4 for the main bus.
- Unfortunately, the conflicting demands for high performance and low power mean that most modern microcontrollers have much more complicated clocks, often with two or more sources.
- In many applications the MCU spends most of its time in a low-power mode until some event occurs, when it must wake up and handle the event rapidly. It is often necessary to keep track of real time, either so that the MCU can wake periodically (every second or minute, for instance) or to time-stamp external events.
- Therefore, two clocks with quite different specifications are often needed:
    1. Afast clock to drive the CPU, which can be started and stopped rapidly to conserve energy but usually need not be particularly accurate.
    2. Aslow clock that runs continuously to monitor real time, which must therefore use little power and may need to be accurate.

The MSP430 addresses the conflicting demands for high performance, low power, and a precise frequency by using three internal clocks, which can be derived from up to four sources. These are the internal clocks, which are the same in all devices:

- Master clock, MCLK, is used by the CPU and a few peripherals.

- Subsystem master clock, SMCLK, is distributed to peripherals.
- Auxiliary clock, ACLK, is also distributed to peripherals. Typically SMCLK runs at the same frequency as MCLK, both in the megahertz range. ACLK is often derived from a watch crystal and therefore runs at a much lower frequency. Most peripherals can select their clock from either SMCLK or ACLK.
- ACLK comes from a low-frequency crystal oscillator, typically at 32KHz.
- Both MCLK and SMCLK are supplied by an internal digitally controlled oscillator (DCO), which runs at about 0.8MHz in the MSP430x1xx and 1.1MHz in the MSP430F2xx.

Execution of a program usually proceeds predictably, but there are two classes of exception to this rule: interrupts and resets:

Interrupts: Usually generated by hardware (although they can be initiated by software) and often indicate that an event has occurred that needs an urgent response. Apacket of data might have been received, for instance, and needs to be processed before the next packet arrives. The processor stops what it was doing, stores enough information (the contents of the program counter and status register) for it to resume later on and executes an interrupt service routine (ISR). It returns to its previous activity when the ISR has been completed.Thus an ISR is something like a subroutine called by hardware (at an unpredictable time) rather than software. Asecond use of interrupts, which is particularly important in the MSP430, is to wake the processor from a low-power state.

Resets: Again usually generated by hardware, either when power is applied or when something catastrophic has happened and normal operation cannot continue. This can happen accidentally if the watchdog timer has not been disabled, which is easy to forget. Areset causes the device to (re)start from a well-defined state.

Pin out diagram of MSP430:

Typical pins can be configured for either input or output and some inputs may generate interrupts when the voltage on the pin changes. This is useful to awaken a system that has entered a low-power mode while it waits for input from a (slow) human. Pins usually have further functions as well as described in the section “The Outside View—Pin-Out” shown in fig;

The pin-out shows which interior functions are connected to each pin of the package. There are several diagrams for each device, corresponding to the different packages in which it is produced. The

F2013 is available in a traditional 14-pin plastic dual-in-line package (PDIP).

- VCC and VSS are the supply voltage and ground for the whole device (the analog and digital supplies are separate in the 16-pin package).
- P1.0–P1.7, P2.6, and P2.7 are for digital input and output, grouped into ports P1 and P2.
- TACLK, TA0, and TA1 are associated with Timer_A; TACLK can be used as the clock input to the timer, while TA0 and TA1 can be either inputs or outputs. These can be used on several pins because of the importance of the timer.
- A0−,A0+, and so on, up toA4±, are inputs to the analog-to-digital converter. It has four differential channels, each of which has negative and positive inputs.
- VREF is the reference voltage for the converter.
- ACLK and SMCLK are outputs for the microcontroller's clock signals. These can be used to supply a clock to external components or for diagnostic purposes.
- SCLK, SDO, and SCLare used for the universal serial interface, which communicates with external devices using the serial peripheral interface (SPI) or inter-integrated circuit (I2C) bus.
- XIN and XOUT are the connections for a crystal, which can be used to provide an accurate, stable clock frequency.
- RST is an active low reset signal. Active low means that it remains high near VCC for normal operation and is brought low near VSS to reset the chip.Alternative notations to show the active low nature are _RST and /RST.
- NMI is the nonmaskable interrupt input, which allows an external signal to interrupt the normal operation of the program.
- TCK, TMS, TCLK, TDI, TDO, and TEST form the full JTAG interface, used to program and debug the device.
- SBWTDIO and SBWTCK provide the Spy-Bi-Wire interface, an alternative to the usual JTAG connection that saves pins.
- A less common feature of the MSP430 is that some functions are available at several pins rather than a single one.

- The emulation, JTAG interface and Spy-Bi-Wire are used to communicate with a desktop computer when downloading a program and for debugging.

Addressing Modes:

- A key feature of any CPU is its range of addressing modes, the ways in which operands can be specified.
- The MSP430 has four basic modes for the source but only two for the destination in instructions with two operands.
- These modes are made more useful by the way in which they interact with the CPU's registers.
- All 16 of these are treated on an almost equal basis, including the four special-purpose registers R0–R3 or PC, SP, SR/CG1, and CG2. The combination of the basic addressing modes and the registers gives the seven modes listed in the user's guides, although eight could reasonably be claimed.

Double operand (Format I): Arithmetic and logical operations with two operands such as add.w src,dst, which is the equivalent of dst+=src in C. Note the different ordering of src and dst in C and assembly language. Both operands must be specified in the instruction. This contrasts with accumulator-based architectures, where an accumulator or working register is used automatically as the destination and one operand.

Single operand (Format II): Amixture of instructions for control or to manipulate a single operand, which is effectively the source for the addressing modes. The nomenclature in TI's documents is inconsistent in this respect. Jumps: The jump to the destination rather than its absolute address, in other words the offset that must be added to the program counter.

Register Mode: This uses one or two of the registers in the CPU. It is the most straightforward addressing mode and is available for both source and destination. For example,

Syntax: mov.w R5,R6 ; move (copy) word from R5 to R6

The registers are specified in the instruction word; no further data are needed. It is also the fastest mode and this instruction takes only 1 cycle. Any of the 16 registers can be used for either source or destination but there are some special cases:

- The PC is incremented by 2 while the instruction is being fetched, before it is used as a source.
- The constant generator CG2 reads 0 as a source.
- Both PC and SP must be even because they address only words, so the lsb is discarded if they are used as the destination.

- SR can be used as a source and destination in almost the usual way although there are some details about the behavior of individual bits.

For byte instructions,

- Operands are taken from the lower byte; the upper byte is not affected.
- The result is written to the lower byte of the register and the upper byte is cleared.

The upper byte of a register in the CPU cannot be used as a source. If this is needed, the 2 bytes in a word must first be swapped with swpb.

Indexed Mode:

This looks much like an element of an array in C. The address is formed by adding a constant base address to the contents of a CPU register; the value in the register is not changed. Indexed addressing can be used for both source and destination. For example, suppose that R5 contains the value 4 before this instruction:

Syntax: mov.b 3(R5),R6 ; load byte from address 3+(R5)=7 into R6

The address of the source is computed as 3+(R5)=3+4=7. Thus a byte is loaded from address 7 into R6. The value in R5 is unchanged. There is no restriction on the address for a byte but remember that words must lie on even addresses.

Indexed addressing can be used for the source, destination, or both. The base addresses, just the single value 3 here because only one address is indexed, are stored in the words following the instruction. They cannot be produced by the constant generator.

Symbolic Mode (PC Relative):

In this case the program counter PC is used as the base address, so the constant is the offset to the data from the PC. TI calls this the symbolic mode although it is usually described as PC-relative addressing. It is used by writing the symbol for a memory location without any prefix. For example, suppose that a program uses the variable LoopCtr, which occupies a word. The following instruction stores the value of LoopCtr in R6 using symbolic mode:

Syntax: mov.w LoopCtr,R6 ; load word LoopCtr into R6, symbolic mode

mov.w X(PC),R6 ; load word LoopCtr into R6, symbolic mode

where X = LoopCtr−PC is the offset that needs to be added to PC to get the address of LoopCtr. This calculation is performed by the assembler, which also accounts for the automatic incrementing of PC.

Absolute Mode:

The constant in this form of indexed addressing is the absolute address of the data. This is already the complete address required so it should be added to a register that contains 0. The MSP430 mimics this

by using the status register SR. It makes no sense to use the real contents of SR in an address so it behaves as though it contains 0 when it is used as the base for indexed addressing. This is one of its roles as constant generator CG1. Absolute addressing is shown by the prefix & and should be used for special function and peripheral registers, whose addresses are fixed in the memory map. This example copies the port 1 input register into register R6:

Syntax: mov.b &P1IN,R6 ; load byte P1IN into R6, absolute mode

The assembler replaces this by the indexed form

Syntax: mov.b P1IN(SR),R6 ; load byte P1IN into R6, absolute mode

where P1IN is the absolute address of the register.

SP-Relative Mode:

TI does not claim this as a distinct mode of addressing, but many other companies do! The stack pointer SP can be used as the register in indexed mode like any other. Recall from the section "Stack Pointer (SP)" on page 120 that the stack grows down in memory and that SP points to (holds the address of) the most recently added word. Suppose that we wanted to copy the value that had been pushed onto the stack before the most recent one. The following instruction will do this:

Syntax: mov.w 2(SP),R6 ; copy most recent word but one from stack For example, suppose that the stack were as shown in Figure 5.2(d) with SP=0x027C. Then the preceding instruction would load 0x1234 into R6.

Indirect Register Mode

This is available only for the source and is shown by the symbol @ in front of a register, such as @R5. It means that the contents of R5 are used as the address of the operand. In other words, R5 holds a pointer rather than a value. (The contents of R5 would be the operand itself if the @ were omitted.) Suppose that R5 contains the value 4 before this instruction:

Syntax: mov.w @R5,R6 ; load word from address (R5)=4 into R6

The address of the source is 4, the value in R5. Thus a word is loaded from address 4 into R6. The value in R5 is unchanged. This has exactly the same effect as indexed addressing with a base address of 0 but saves a word of program memory, which also makes it faster. This is very loosely the equivalent of r6 = *r5 in C, without worrying about the types of the variables held in the registers.

Indirect addressing cannot be used for the destination so indexed addressing must be used instead. Thus the reverse of the preceding move must be done like this:

Syntax: mov.w R6,0(R5) ; store word from R6 into address 0+(R5)=4

The penalty is that a word of 0 must be stored in the program memory and fetched from it. The constant generator cannot be used.

Indirect Autoincrement Register Mode:

Again this is available only for the source and is shown by the symbol @ in front of a register with a + sign after it, such as @R5+. It uses the value in R5 as a pointer and automatically increments it afterward by 1 if a byte has been fetched or by 2 for a word. Suppose yet again that R5 contains the value 4 before this instruction:

Syntax: mov.w @R5+,R6

Aword is loaded from address 4 into R6 and the value in R5 is incremented to 6 because a word (2 bytes) was fetched. This is useful when stepping through an array or table, where expressions of the form *c++ are often used in C. This mode cannot be used for the destination. Instead the main instruction must use indexed mode with an offset of 0, followed by an explicit increment of the register by 1 or 2. The reverse of this move therefore needs two instructions:

Syntax: mov.w R6,0(R5) ; store word from R6 into address 0+(R5)=4 incd.w R5 ; R5 += 2

This is undoubtedly a bit clumsy.

Autoincrement is usually called postincrement addressing because many processors have a complementary predecrement addressing mode, equivalent to *--c in C, but the MSP430 does not.

An important feature of the addressing modes is that all operations on the first address are fully completed before the second address is evaluated. This needs to be considered when moving blocks of memory. The move itself might be done by a line like this:

Syntax: mov.w @R5+,0x0100(R5)

Suppose as usual that R5 initially contains the value 4. The contents of address 4 is read and R5 is double-incremented to 6 because a word is involved. Only now is the address for the destination calculated as 0x0100+0x0006=0x0106. Thus a word is copied from address 0x0004 to 0x0106; the offset is not just the value of 0x0100 used as the base address for the destination. The compiler takes care of these tricky details if you write in C, but you are on your own with assembly language.

Immediate Mode This is a special case of autoincrement addressing that uses the program counter PC. Look at this example:

Syntax: mov.w @PC+,R6 ; load immediate word into R6

The PC is automatically incremented after the instruction is fetched and therefore points to the following word. The instruction loads this word into R6 and increments PC to point to the next word, which in this case is the next instruction. The overall effect is that the word that followed the original instruction has been loaded into R6. This is how the MSP430 handles immediate or literal values, which are encoded into the stream of instructions. It is the equivalent of r6 = constant. This is available only for the source of an instruction because it is a type of autoincrement addressing but it is hard to see when it would be useful for

the destination.

Currently four families of MSP430 are available. The letter after MSP430 shows the type of memory. Most part numbers include F for flash memory but some have C for ROM.

Instruction set:

The instruction set

All instructions are 16 bits long, and there are only three instruction formats:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	0	0	1	0	0	Opcode			B/W	Ad		Destreg			
0	0	1	Condition			PC offset (10 bit)									
Opcode				Source reg				Ad	B/W	As		Destreg			

As and *Ad* are the source and destination addressing modes. *B/W* is a bit that is set to 1 for byte instructions. 2-operand opcodes begin at 0100 = 4.

As you can see, there are at most 8+8+12 = 28 instructions to keep track of, which is nice and simple.

One-operand instructions:

000	RRC(.B)	9-bit rotate right through carry. C->msbit->...->lsbit->C. Clear the carry bit beforehand to do a logical right shift.
001	SWPB	Swap 8-bit register halves. No byte form.
010	RRA(.B)	Badly named, this is an 8-bit arithmetic right shift.
011	SXT	Sign extend 8 bits to 16. No byte form.
100	PUSH(.B)	Push operand on stack. Push byte decrements SP by 2. CPU BUG: PUSH #4 and PUSH #8 do not work when the short encoding using @r2 and @r2+ is used. The workaround, to use a 16-bit immediate, is trivial, so TI do not plan to fix this bug.
101	CALL	Fetch operand, push PC, then assign operand value to PC. Note the immediate form is the most commonly used. There is no easy way to perform a PC-relative call; the PC-relative addressing mode fetches a word and uses it as an absolute address. This has no byte form.
110	RETI	Pop SP, then pop PC. Note that because flags like CPUOFF are in the stored status register, the CPU will normally return to the low-power mode it was previously in. This can be changed by adjusting the SR value stored on the stack before invoking RETI (see below). The operand field is unused.
111	Not used	The MSP430 actually only has 27 instructions.

The status flags are set by RRA, RRC, SXT, and RETI.

The status flags are NOT set by PUSH, SWPB, and CALL.

Relative jumps. These are all PC-relative jumps, adding twice the sign-extended offset to the PC, for a jump range of -1024 to +1022.

000	JNE/JNZ	Jump if Z==0 (if !=)
001	JEQ/Z	Jump if Z==1 (if ==)
010	JNC/JLO	Jump if C==0 (if unsigned <)
011	JC/JHS	Jump if C==1 (if unsigned >=)
100	JN	Jump if N==1 Note there is no "JP" if N==0!
101	JGE	Jump if N==V (if signed >=)
110	JL	Jump if N!=V (if signed <)
111	JMP	Jump unconditionally

Two-operand instructions. These basically perform *dest = src op dest* operations. However, MOV doesn't fetch the destination, and CMP and BIT do not write to the destination. All are valid in their 8 and 16 bit forms.

Operands are written in the order *src,dest*.

0100	MOV src,dest	dest = src	The status flags are NOT set.
0101	ADD src,dest	dest += src	
0110	ADDC src,dest	dest += src + C	
0111	SUBC src,dest	dest += ~src + C	
1001	SUB src,dest	dest -= src	Implemented as dest += ~src + 1.
1001	CMP src,dest	dest – src	Sets status only; the destination is not written.
1010	DADD src,dest	dest += src + C, BCD.	
1011	BIT src,dest	dest&src	Sets status only; the destination is not written.
1100	BIC src,dest	dest&= ~src	The status flags are NOT set.
1101	BIS src,dest	dest |= src	The status flags are NOT set.
1110	XOR src,dest	dest ^= src	
1111	AND src,dest	dest&=- src	

There are a number of zero- and one-operand pseudo-operations that can be built from these two-operand forms. These are usually referred to as "emulated" instructions:

NOP	MOV r3,r3	Any register from r3 to r15 would do the same thing.
POP dst	MOV @SP+,dst	

Note that other forms of a NOP instruction can be constructed as emulated instructions, which take different

numbers of cycles to execute. These can sometimes be useful in constructing accurate timing patterns in software.

Branch and return can be done by moving to PC (r0):

BR dst	MOV dst,PC
RET	MOV @SP+,PC

The constants were chosen to make status register (r2) twiddling efficient:

CLRC	BIC #1,SR
SETC	BIS #1,SR
CLRZ	BIC #2,SR
SETZ	BIS #2,SR
CLRN	BIC #4,SR
SETN	BIS #4,SR
DINT	BIC #8,SR
EINT	BIC #8,SR

Shift and rotate left is done with add:

RLA(.B) dst	ADD(.B) dst,dst
RLC(.B) dst	ADDC(.B) dst,dst

Some common one-operand instructions:

INV(.B) dst	XOR(.B) #-1,dst
CLR(.B) dst	MOV(.B) #0,dst
TST(.B) dst	CMP(.B) #0,dst

Increment and decrement (by one or two):

DEC(.B) dst	SUB(.B) #1,dst
DECD(.B) dst	SUB(.B) #2,dst
INC(.B) dst	ADD(.B) #1,dst
INCD(.B) dst	ADD(.B) #2,dst

Adding and subtracting only the carry bit:

ADC(.B) dst	ADDC(.B) #0,dst
DADC(.B) dst	DADD(.B) #0,dst
SBC(.B) dst	SUBC(.B) #0,dst

Comparisions of MSP430x1xx, MSP430x2xx, x3xx,x4xx,x5xx,x6xx:

MSP430x1xx series

The MSP430x1xx Series is the basic generation without an embedded LCD controller. They are generally smaller than the '3xx generation. These flash- or ROM-based ultra-low-power MCUs offer 8 MIPS, 1.8–3.6 V operation, up to 60 KB flash, and a wide range of analog and digital peripherals.

- Power specification overview, as low as:
  - 0.1 μA RAM retention
  - 0.7 μA real-time clock mode
  - 200 μA / MIPS active
  - Features fast wake-up from standby mode in less than 6 μs.

- Device parameters
  - Flash options: 1–60 KB
  - ROM options: 1–16 KB
  - RAM options: 128 B–10 KB
  - GPIO options: 14, 22, 48 pins
  - ADC options: Slope, 10 & 12-bit SAR
  - Other integrated peripherals: 12-bit DAC, up to 2 16-bit timers, watchdog timer, brown-out reset, SVS, USART module (UART, SPI), DMA, 16×16 multiplier, Comparator_A, temperature sensor

MSP430F2xx series

The MSP430F2xx Series are similar to the '1xx generation, but operate at even lower power, support up to 16 MHz operation, and have a more accurate (±2%) on-chip clock that makes it easier to operate without an external crystal. These flash-based ultra-low power devices offer 1.8–3.6 V operation. Includes the very-low power oscillator (VLO), internal pull-up/pull-down resistors, and low-pin count options.

- Power specification overview, as low as:

    - 0.1 μA RAM retention
    - 0.3 μA standby mode (VLO)
    - 0.7 μA real-time clock mode
    - 220 μA / MIPS active
    - Feature ultra-fast wake-up from standby mode in less than 1 μs

- Device parameters
    - Flash options: 1–120 KB
    - RAM options: 128 B – 8 KB
    - GPIO options: 10, 11, 16, 24, 32, and 48 pins
    - ADC options: Slope, 10 & 12-bit SAR, 16 & 24-bit Sigma Delta
    - Other integrated peripherals: operational amplifiers, 12-bit DAC, up to 2 16-bit timers, watchdog timer, brown-out reset, SVS, USI module (I^2C, SPI), USCI module, DMA, 16×16 multiplier, Comparator_A+, temperature sensor

MSP430x3xx series

The MSP430x3xx Series is the oldest generation, designed for portable instrumentation with an embedded LCD controller. This also includes a frequency-locked loop oscillator that can automatically synchronize to a low-speed (32 kHz) crystal. This generation does not support EEPROM memory, only mask ROM and UV-eraseable and one-time programmable EPROM. Later generations provide only flash memory and mask ROM options. These devices offer 2.5–5.5 V operation, up to 32 KB ROM.

- Power specification overview, as low as:
    - 0.1 μA RAM retention
    - 0.9 μA real-time clock mode
    - 160 μA / MIPS active
    - Features fast wake-up from standby mode in less than 6 μs.

- Device parameters:
    - ROM options: 2–32 KB
    - RAM options: 512 B–1 KB
    - GPIO options: 14, 40 pins
    - ADC options: Slope, 14-bit SAR
    - Other integrated peripherals: LCD controller, multiplier

MSP430x4xx series

The MSP430x4xx Series are similar to the '3xx generation, but include an integrated LCD controller, and are larger and more capable. These flash or ROM based devices offers 8–16 MIPS at 1.8–3.6 V operation, with FLL, and SVS. Ideal for low power metering and medical applications.

- Power specification overview, as low as:
  - 0.1 μA RAM retention
  - 0.7 μA real-time clock mode
  - 200 μA / MIPS active
  - Features fast wake-up from standby mode in less than 6 μs.

- Device parameters:
  - Flash/ROM options: 4 – 120 KB
  - RAM options: 256 B – 8 KB
  - GPIO options: 14, 32, 48, 56, 68, 72, 80 pins
  - ADC options: Slope, 10 & 12-bit SAR, 16-bit Sigma Delta
  - Other integrated peripherals: SCAN_IF, ESP430, 12-bit DAC, Op Amps, RTC, up to 2 16-bit timers, watchdog timer, basic timer, brown-out reset, SVS, USART module (UART, SPI), USCI module, LCD Controller, DMA, 16×16 & 32x32 multiplier, Comparator_A, temperature sensor, 8 MIPS CPU Speed

MSP430x5xx series

The MSP430x5xx Series are able to run up to 25 MHz, have up to 512 KB flash memory and up to 66 KB RAM. This flash-based family features low active power consumption with up to 25 MIPS at 1.8–3.6 V operation (165 uA/MIPS). Includes an innovative power management module for optimal power consumption and integrated USB.[3]

- Power specification overview, as low as:
  - 0.1 μA RAM retention
  - 2.5 μA real-time clock mode
  - 165 μA / MIPS active
  - Features fast wake-up from standby mode in less than 5 μs.

- Device parameters:

- Flash options: up to 512 KB
- RAM options: up to 66 KB
- ADC options: 10 & 12-bit SAR
- GPIO options: 29, 31, 47, 48, 63, 67, 74, 87 pins
- Other integrated peripherals: High resolution PWM, 5 V I/O's, USB, backup battery switch, up to 4 16-bit timers, watchdog timer, Real-Time Clock, brown-out reset, SVS, USCI module, DMA, 32x32 multiplier, Comp B, temperature sensor

MSP430x6xx series

The MSP430x6xx Series are able to run up to 25 MHz, have up to 512 KB flash memory and up to 66 KB RAM. This flash-based family features low active power consumption with up to 25 MIPS at 1.8–3.6 V operation (165 uA/MIPS). Includes an innovative power management module for optimal power consumption and integrated USB.

- Power specification overview, as low as:
  - 0.1 μA RAM retention
  - 2.5 μA real-time clock mode
  - 165 μA / MIPS active
  - Features fast wake-up from standby mode in less than 5 μs.

- Device parameters:
  - Flash options: up to 512 KB
  - RAM options: up to 66 KB
  - ADC options: 12-bit SAR
  - GPIO options: 74 pins
  - Other integrated peripherals: USB, LCD, DAC, Comparator_B, DMA, 32x32 multiplier, power management module (BOR, SVS, SVM, LDO), watchdog timer, RTC, Temp sensor

UNIT II
TYPICAL EMBEDDED SYSTEM:

Embedded systems overview

An embedded system is nearly any computing system other than a desktop computer. An embedded system is a dedicated system which performs the desired function upon power up, repeatedly.

Embedded systems are found in a variety of common electronic devices such as consumer electronics ex. Cell phones, pagers, digital cameras, VCD players, portable Video games, calculators, etc.,

Embedded systems are found in a variety of common electronic devices, such as: (a)consumer electronics -- cell phones, pagers, digital cameras, camcorders, videocassette recorders, portable video games, calculators, and personal digital assistants; (b) home appliances
-- microwave ovens, answering machines, thermostat, home security, washing machines, and lighting systems; (c) office automation -- fax machines, copiers, printers, and scanners; (d) business equipment

-- cash registers, curbside check-in, alarm systems, card readers, product scanners, and automated teller machines; (e) automobiles --transmission control, cruise control, fuel injection, anti-lock brakes, and active suspension

Classifications of Embedded systems

1. Small Scale Embedded Systems: These systems are designed with a single 8- or 16-bit microcontroller; they have little hardware and software complexities and involve board- level design. They may even be battery operated. When developing embedded software for these, an editor, assembler and cross assembler, specific to the microcontroller or processor used, are the main programming tools. Usually, _C‘ is used for developing these systems. _C‘ program compilation is done into the assembly, and executable codes are then appropriately located in the system memory. The software has to fit within the memory available and keep in view the need to limit power dissipation when system is running continuously.

2. Medium Scale Embedded Systems: These systems are usually designed with a single or few 16- or 32-bit microcontrollers or DSPs or Reduced Instruction Set Computers (RISCs). These have both hardware and software complexities. For complex software design, there are the following programming tools: RTOS, Source code engineering tool, Simulator, Debugger and Integrated Development Environment (IDE). Software tools also provide the solutions to the hardware complexities. An assembler is of little use as a programming tool. These systems may also employ the readily available ASSPs and IPs (explained later) for the various functions—for example, for the bus interfacing, encrypting, deciphering, discrete cosine transformation and inverse transformation, TCP/IP protocol stacking and network connecting functions.
3. Sophisticated Embedded Systems: Sophisticated embedded systems have enormous hardware and software complexities and may need scalable processors or configurable processors and programmable logic arrays. They are used for cutting edge applications that need hardware and software co-design and integration in the final system; however, they are constrained by the processing speeds available in their hardware units. Certain software functions such as encryption and deciphering algorithms, discrete cosine transformation and inverse transformation algorithms, TCP/IP protocol stacking and network driver functions are implemented in the hardware to obtain additional speeds by saving time. Some of the functions of the hardware resources in the system are also implemented by the software. Development

tools for these systems may not be readily available at a reasonable cost or may not be available at all. In some cases, a compiler or retarget able compiler might have to be developed for these.

The processing units of the embedded system

1. Processor in an Embedded System A processor is an important unit in the embedded system hardware. A microcontroller is an integrated chip that has the processor, memory and several other hardware units in it; these form the microcomputer part of the embedded system. An embedded processor is a processor with special features that allow it to be embedded into a system. A digital signal processor (DSP) is a processor meant for applications that process digital signals.

2. Commonly used microprocessors, microcontrollers and DSPs in the small-, medium-and large scale embedded systems
3. A recently introduced technology that additionally incorporates the application-specific system processors (ASSPs) in the embedded systems.
4. Multiple processors in a system.

Embedded systems are a combination of hardware and software as well as other components that we bring together inti products such as cell phones,music player,a network router,or an aircraft guidance system.they are a system within another system as we see in Figure 1.1

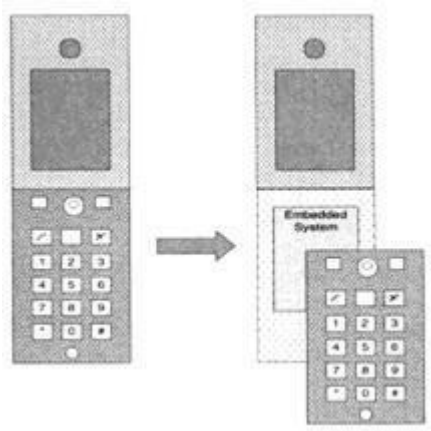


Figure 1.1: A simple embedded system

Building an embedded system

we embed 3 basic kinds of computing engines into our systems: microprocessor, microcomputer and microcontrollers. The microcomputer and other hardware are connected via A system bus is a single computer bus that connects the major components of a computer system. The technique was developed to reduce costs and improve modularity. It combines the functions of a data bus to carry information, an address bus to determine where it should be sent, and a control bus to determine its operation.

The system bus is further classified int address ,data and control bus.the microprocessor controls the whole system by executing a set of instructions call firmware that is stored in ROM.

An instruction set, or instruction set architecture (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor. To run the application, when power is first turned ON, the microprocessor addresses a predefined location and fetches, decodes, and executes the instruction one after the other. The implementation of a microprocessor based embedded system combines the individual pieces into an integrated whole as shown in Figure 1.2, which represents the architecture for a typical embedded system and identifies the minimal set of necessary components.

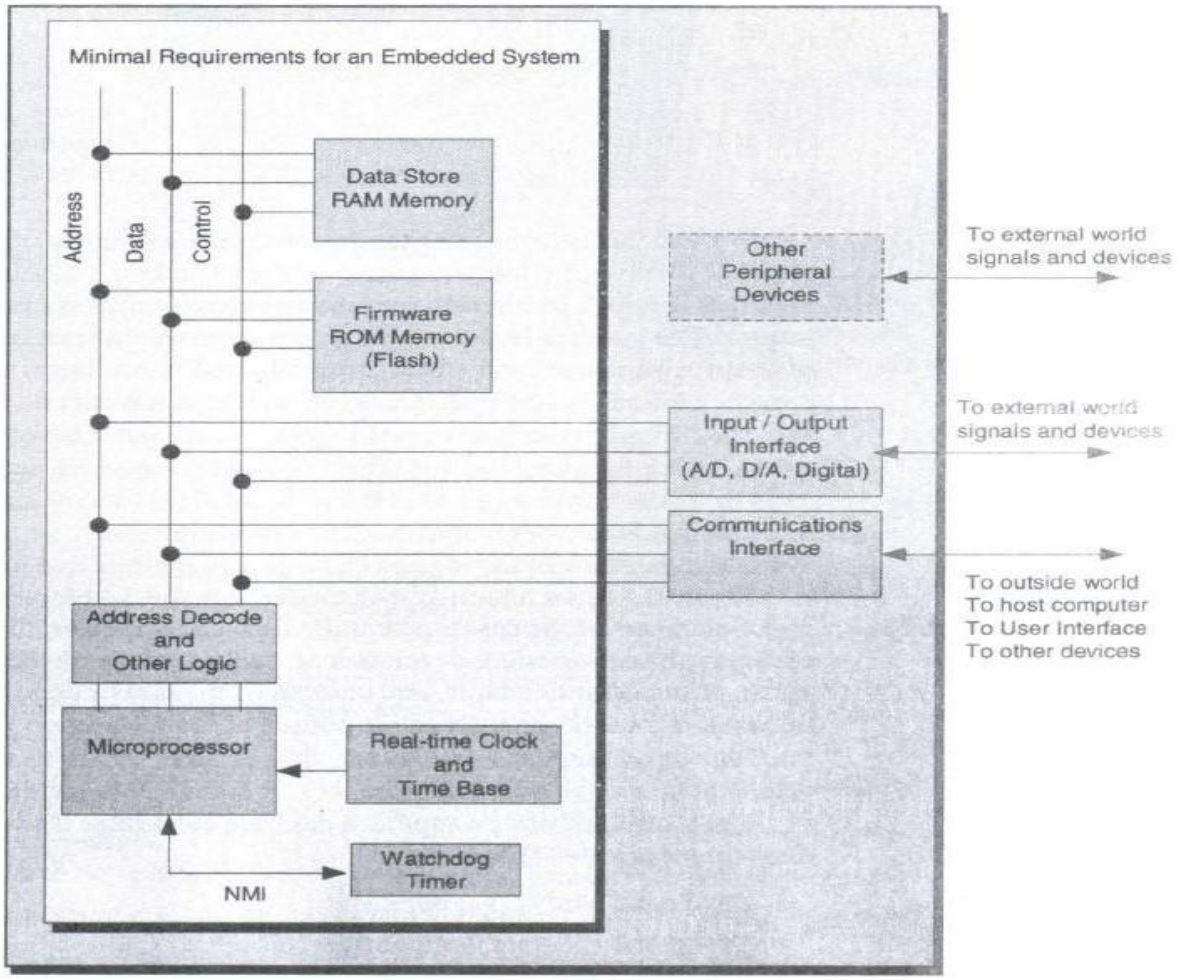


Figure 1.2 :A Microprocessor based Embedded system

Embedded design and development process

Figure1.3 shows a high level flow through the development process and identifies the major elements of the development life cycle.

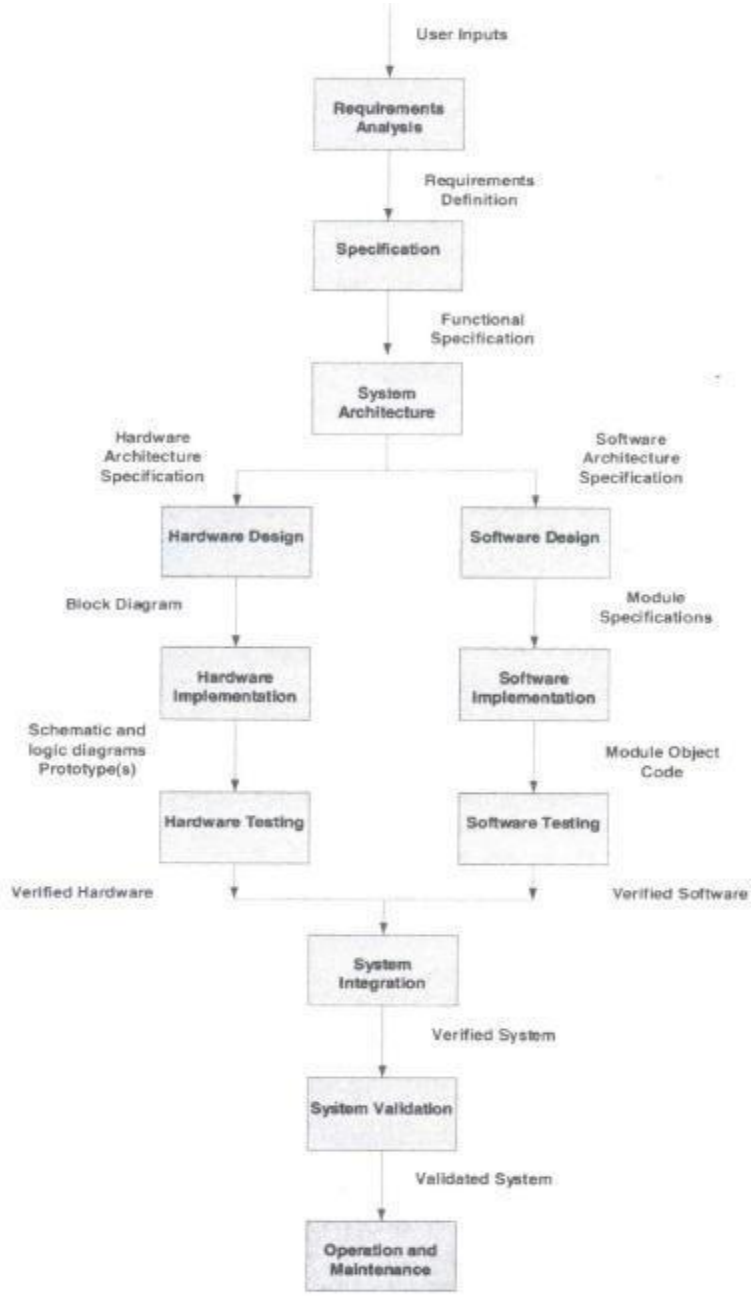


Figure 1.3 Embedded system life cycle

The traditional design approach has been traverse the two sides of the accompanying diagram separately, that is,

- Design the hardware components • Design the software components. • Bring the two together.
- Spend time testing and debugging the system.

The major areas of the design process are

- Ensuring a sound software and hardware specification.
- Formulating the architecture for the system to be designed. • Partitioning the h/w and s/w.
- Providing an iterative approach to the design of h/w and s/w.

The important steps in developing an embedded system are • Requirement definition.

- System specification. • Functional design
- Architectural design • Prototyping.

The major aspects in the development of embedded applications are

- Digital hardware and software architecture
- Formal design , development, and optimization process. • Safety and reliability.
- Digital hardware and software/firmware design.
- The interface to physical world analog and digital signals.
- Debug, troubleshooting and test of our design.

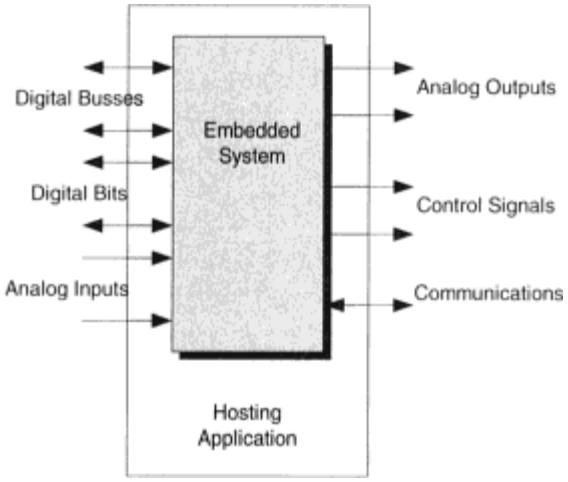


Figure 1.4: Interfacing to the outside world

Embedded applications are intended to work with the physical world, sensing various analog and digital signals while controlling, manipulating or responding to others. The study of the interface to the external world extends the I/O portion of the von-Neumann machine as shown in figure 1.4 with a study of buses, their constitutes and their timing considerations.

Exemplary applications of each type of embedded system

Embedded systems have very diversified applications. A few select application areas of embedded systems are Telecom, Smart Cards, Missiles and Satellites, Computer Networking, Digital Consumer Electronics, and Automotive. Figure 1.9 shows the applications of embedded systems in these areas.

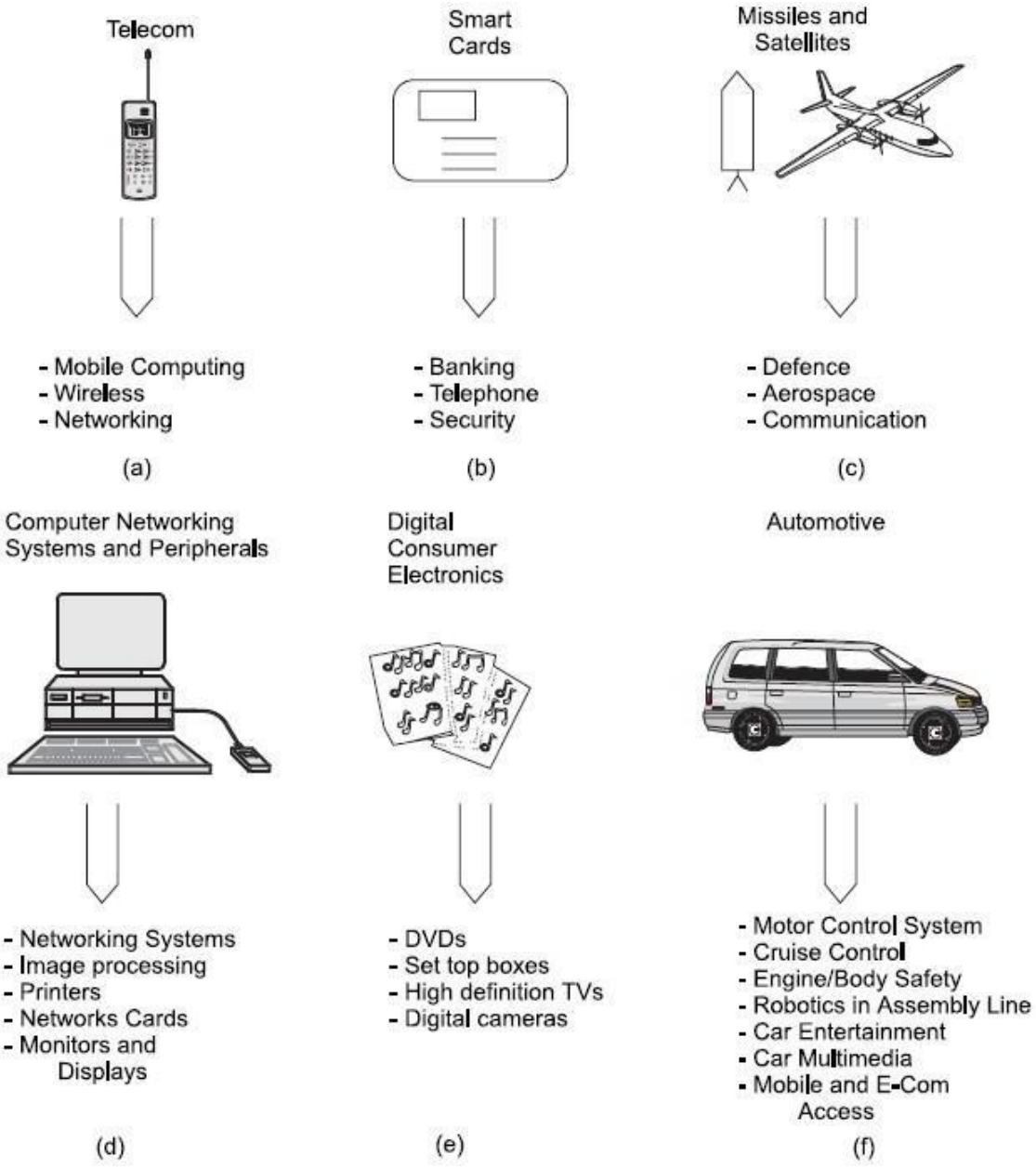


Figure 1.9 Applications of embedded systems

THE HARDWARE SIDE

In today‘s hi-tech and changing world, we can put together a working hierarchy of hard ware components. At the top, we find VLSI circuits comprising of significant pieces of functionality: microprocessor, microcontrollers, FPGA‘s, CPLD, and ASIC.

Our study of hardware side of embedded systems begins with a high level view of the computing core of the system. we will expand and refine that view of hardware both inside and outside of the core. Figure 2.1 illustrates the sequence.

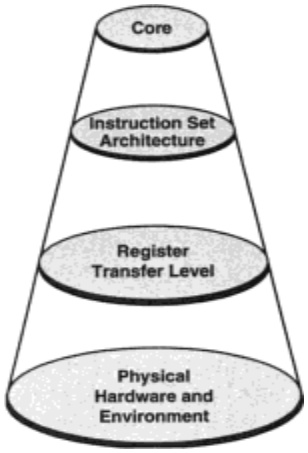


Figure 2.1 Exploring embedded systems

The core level

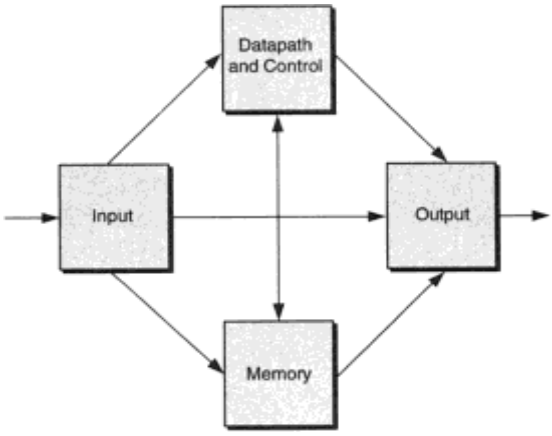


Figure 2.2 Four major blocks of an embedded hardware core

At the top, we begin with a model comprising four major functional blocks i.e., input, output, memory and data path and control depicting the embedded hardware core and high level signal flow as illustrated in figure 2.2.

The source of the transfer is the array of eight bit values; the destination is perhaps a display. in figure 2.3, we refine the high level functional diagram to illustrate a typical bus configuration comprising the address, data and control lines.

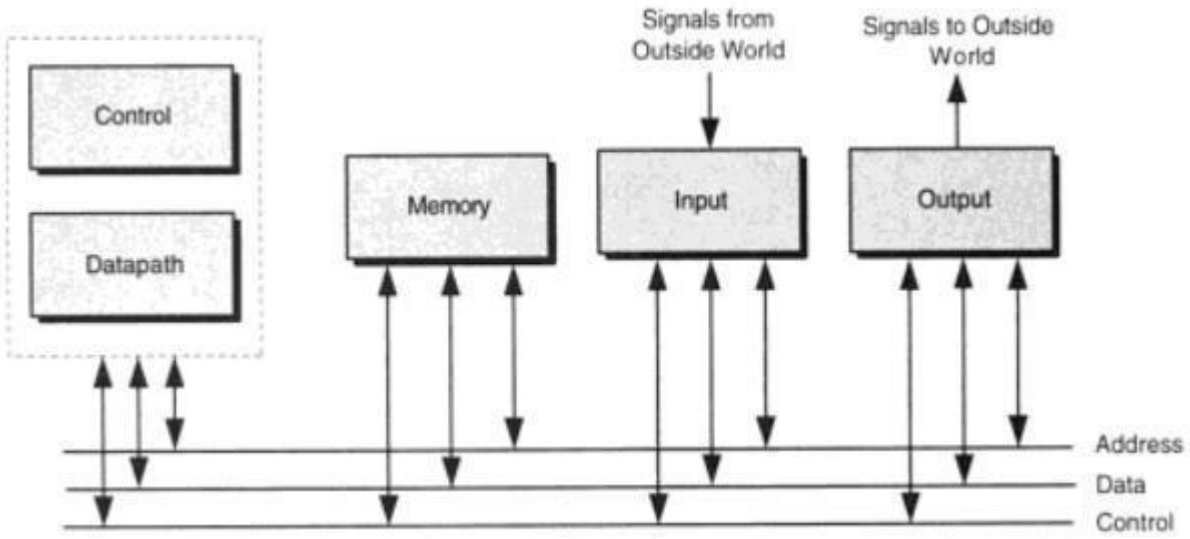


Figure 2.3 A typical Bus structure comprising address, data and control signals.

The Microprocessor

A microprocessor (sometimes abbreviated µP) is a programmable digital electronic component that incorporates the functions of a central processing unit (CPU) on a single semiconducting integrated circuit (IC). It is a multipurpose, programmable device that accepts digital data as input, processes it according to instructions stored in its memory, and provides results as output. It is an example of sequential digital logic, as it has internal memory. Microprocessors operate on numbers and symbols represented in the binary numeral system.

A microprocessor control program can be easily tailored to different needs of a product line, allowing upgrades in performance with minimal redesign of the product. Different features

can be implemented in different models of a product line at negligible production cost. Figure 2.4 shows a block diagram for a microprocessor based system.

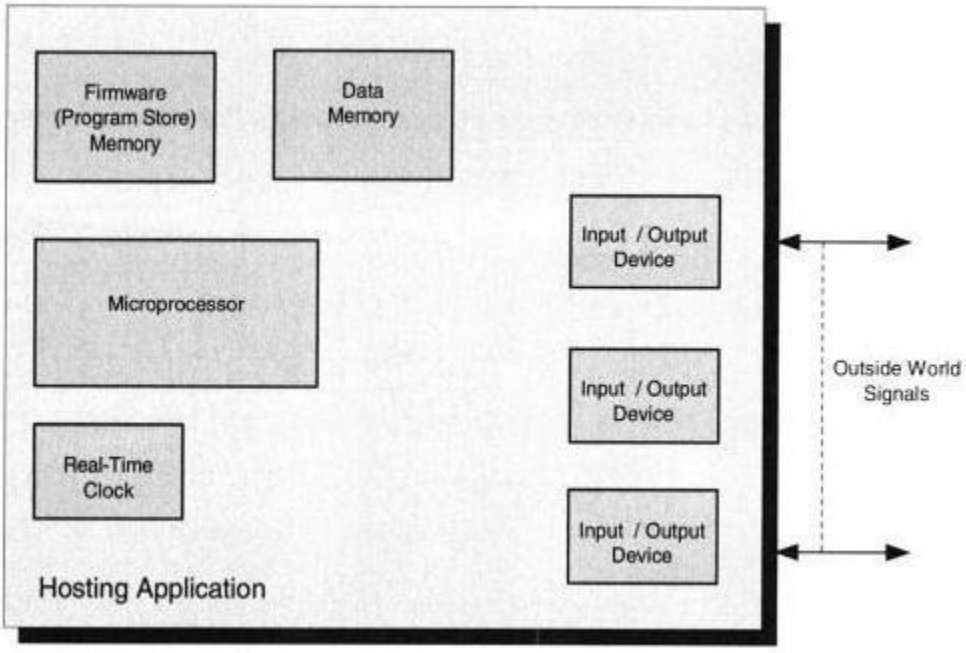


Figure 2.4 : A block diagram for a microprocessor based system

The microcomputer

The microcomputer is a complete computer system that uses a microprocessor as its computational core. Typically, a microcomputer will also utilizes numerous other large scale integrated circuits to provide necessary peripheral functionality. The complexity of microcomputers varies from simple units that are implemented on a single chip along with a small amount of on chip memory and elementary I/O system to the complex that will augment the microprocessor with a wide array of powerful peripheral support circuitry.

The microcontroller

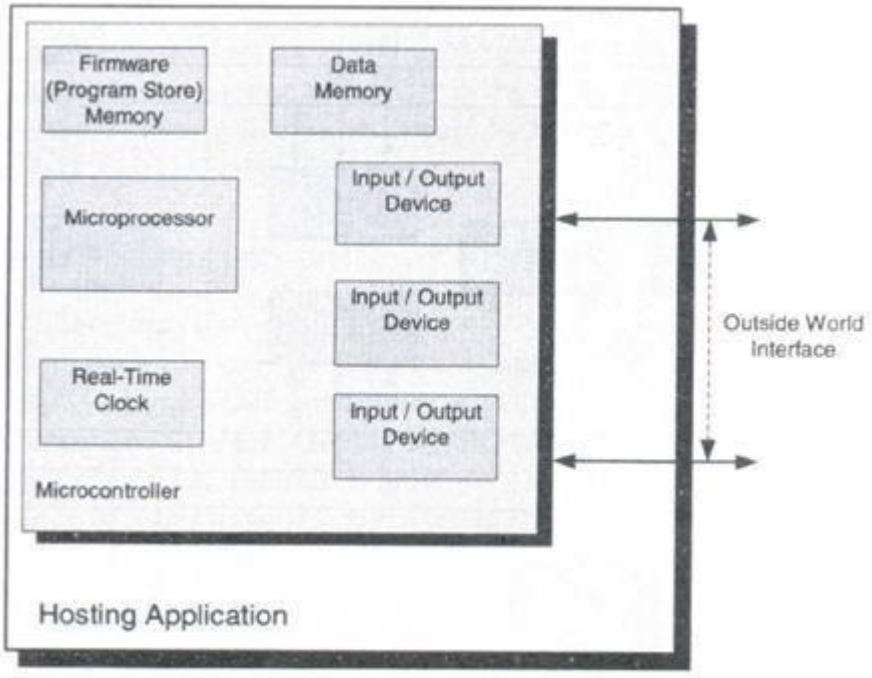
A microcontroller (sometimes abbreviated µC, uC or MCU) is a small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals. Program memory in the form of NOR flash or OTP

ROM is also often included on chip, as well as a typically small amount of RAM. Microcontrollers are designed for embedded applications, in contrast to the microprocessors used in personal computers or other general purpose applications.

Figure 2.5 shows together the microprocessor core and a rich collection of peripherals and I/O capability into a single integrated circuit.

Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, toys and other embedded systems. By reducing the size and cost compared to a design that uses a separate microprocessor, memory, and input/output devices, microcontrollers make it economical to digitally control even more devices and processes. Mixed signal microcontrollers are common, integrating analog components needed to control non-digital electronic systems.

Figure 2.5 :A block diagram for a microcontroller based system



The digital signal processor

A digital signal processor (DSP) is a specialized microprocessor with an architecture optimized for the operational needs of digital signal processing. A DSP provides fast, discrete-

time, signal-processing instructions. It has Very Large Instruction Word (VLIW) processing capabilities; it processes Single Instruction Multiple Data (SIMD) instructions fast; it processes Discrete Cosine Transformations (DCT) and inverse DCT (IDCT) functions fast. The latter are a must for fast execution of the algorithms for signal analyzing, coding, filtering, noise cancellation, echo-elimination, compressing and decompressing, etc. Figure 2.6 shows the block diagram for a digital signal processor

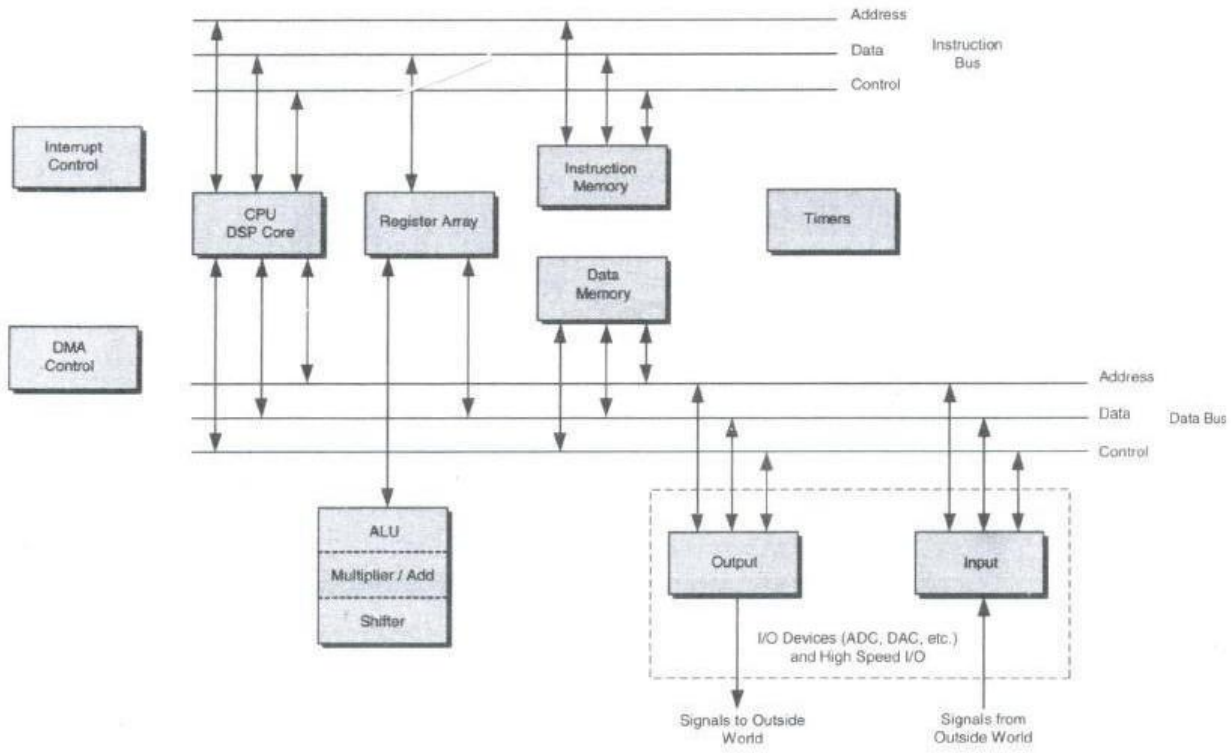


Figure 2.6 A block diagram for a digital signal processor

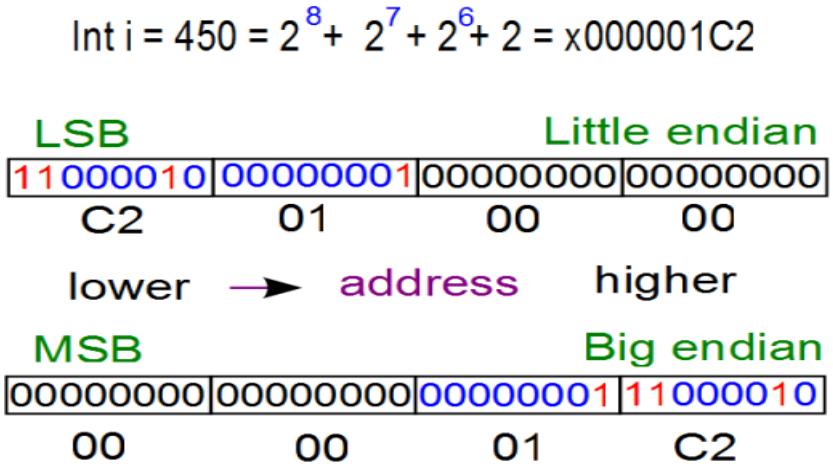
By the standards of general-purpose processors, DSP instruction sets are often highly irregular. One implication for software architecture is that hand-optimized assembly-code routines are commonly packaged into libraries for re-use, instead of relying on advanced compiler technologies to handle essential algorithms.

Hardware features visible through DSP instruction sets commonly include:

- Hardware modulo addressing, allowing circular buffers to be implemented without having to constantly test for wrapping.

- Memory architecture designed for streaming data, using DMA extensively and expecting code to be written to know about cache hierarchies and the associated delays.
- Driving multiple arithmetic units may require memory architectures to support several accesses per instruction cycle
- Separate program and data memories (Harvard architecture), and sometimes concurrent access on multiple data busses
- Special SIMD (single instruction, multiple data) operations
- Some processors use VLIW techniques so each instruction drives multiple arithmetic units in parallel
- Special arithmetic operations, such as fast multiply–accumulates (MACs). Many fundamental DSP algorithms, such as FIR filters or the Fast Fourier transform (FFT) depend heavily on multiply–accumulate performance.
- Bit-reversed addressing, a special addressing mode useful for calculating FFTs
- Special loop controls, such as architectural support for executing a few instruction words in a very tight loop without overhead for instruction fetches or exit testing
- Deliberate exclusion of a memory management unit. DSPs frequently use multi-tasking operating systems, but have no support for virtual memory or memory protection. Operating systems that use **virtual memory** require more time for context switching among processes, which increases latency.

Representing Information



Big endian systems are simply those systems whose memories are organized with the most significant digits or bytes of a number or series of numbers in the upper left corner of a memory page and the least significant in the lower right, just as in a normal spreadsheet.

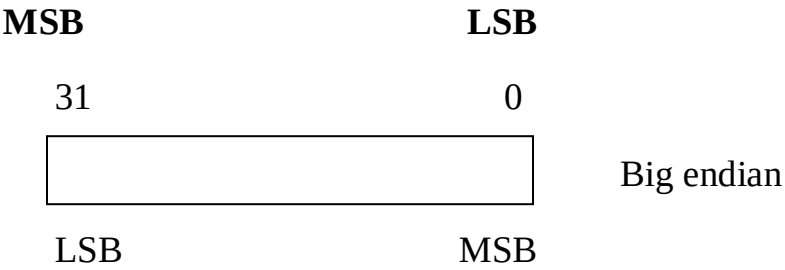
Little endian systems are simply those system whose memories are organized with the least significant digits or bytes of a number or series of numbers in the upper left corner of a memory page and the most significant in the lower right. There are many examples of both types of systems, with the principle reasons for the choice of either format being the underlying operation of the given system.

Understanding numbers

We have seen that within a microprocessor, we don‘t have an unbounded numbers of bits with which to express the various kinds of numeric information that we will be working with in an embedded application. The limitation of finite word size can have unintended consequences of results of any mathematical operations that we might need to perform. Let‘s examine the effects of finite word size on resolution, accuracy, errors and the propagation of errors in these operation. In an embedded system, the integers and floating point numbers are normally represented as binary values and are stored either in memory or in registers. The expensive power of any number is dependent on the number of bits in the number.

Addresses

In the earlier functional diagram as well as in the block diagram for a microprocessor, we learned that information is stored in memory. Each location in memory has an associated address much like an index in the array. If an array has 16 locations to hold information, it will have 16 indices. if a memory has 16 locations to store information ,it will have 16 addresses. Information is accessed in memory by giving its address.



Big endian

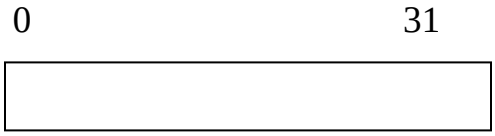


Figure Expressing Addresses

Instructions

An **instruction set**, or **instruction set architecture** (**ISA**), is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor.

The entities that instructions operate on are denoted Operand. The number of operands that an instruction operates on at any time is called the arity of the operation.

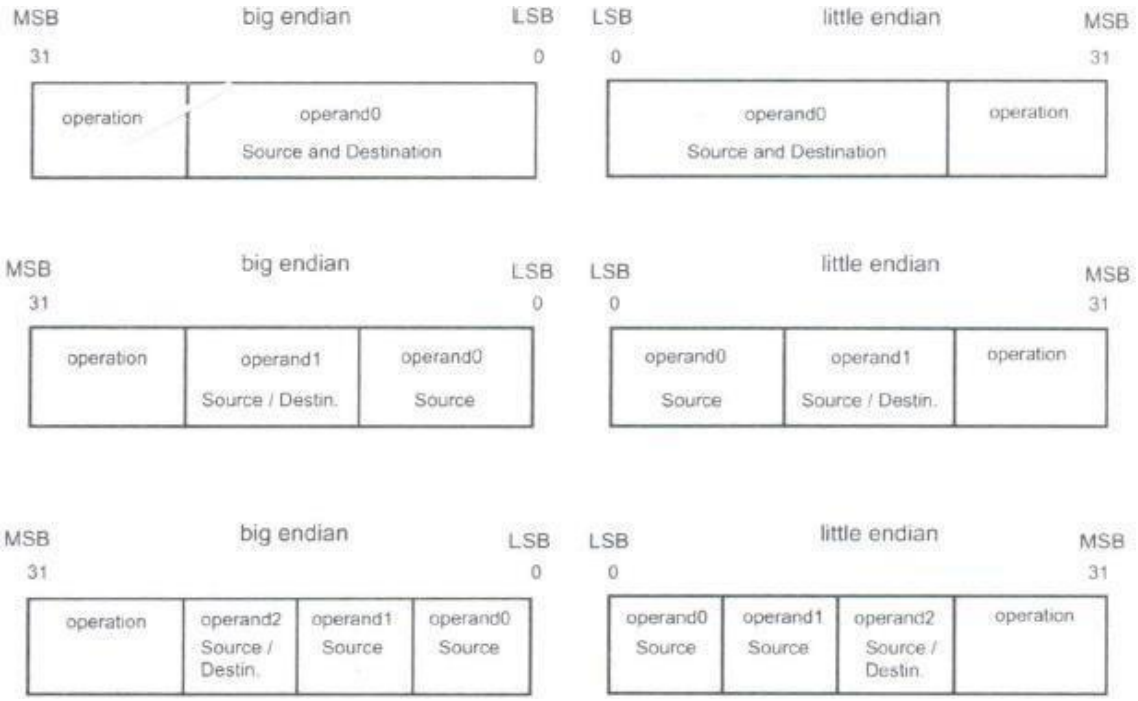


Figure 2.7 Expressing Instructions

In figure 2.7 ,we see that within the 32 bit word, the bit are aggregated into groups or fields. Some of the fields are interpreted as the operation to be performed, and others are seen as the operands involved in the operation.

Embedded systems-An instruction set view

A microprocessor instruction set specifies the basic operations supported by the machine. From the earlier functional model, we see that the objectives of such operations are to transfer or store data, to operate on data, and to make decisions based on the data values or outcome of the operations, corresponding to such operations, we can classify instructions into the following groups

- Data transfer • Flow control
- Arithmetic and logic

Data transfer Instructions

Data transfer instructions are responsible for moving data around inside the processor as well as for bringing data in from the outside world or sending data out. The source and destination can be any of the following:

- A register • Memory
- An input or output As shown in figure

Addressing modes

There are five addressing modes in 8085. 1.Direct Addressing Mode

1. Register Addressing Mode

2. Register Indirect Addressing Mode

3. Immediate Addressing Mode

4. Implicit Addressing Mode

Direct Addressing Mode

In this mode, the address of the operand is given in the instruction itself.

- LDA is the operation.
- 2500 H is the address of source. • Accumulator is the destination.

1. Immediate addressing mode:

In this mode, 8 or 16 bit data can be specified as part of the instruction.

OP Code	Immediate Operand

Example 1 : MOV CL, 03 H

Moves the 8 bit data 03 H into CL Example 2 : MOV DX, 0525 H

Moves the 16 bit data 0525 H into DX

In the above two examples, the source operand is in immediate mode and the destination operand is in register mode. A constant such as ‑VALUE‖ can be defined by the assembler EQUATE directive such as VALUE EQU 35H

Example : MOV BH, VALUE

Used to load 35 H into BH

2. Register addressing mode

The operand to be accessed is specified as residing in an internal register of 8086. Example below shows internal registers, any one can be used as a source or destination operand, however only the data registers can be accessed as either a byte or word.

Example 1 : MOV DX (Destination Register) , CX (Source Register) Which moves 16 bit content of CS into DX.

Example 2 : MOV CL, DL

Moves 8 bit contents of DL into CL MOV BX, CH is an illegal instruction.

* The register sizes must be the same.

3. Direct addressing mode

The 20 bit physical address of the operand in memory is normally obtained as PA = DS : EA

But by using a segment override prefix (SOP) in the instruction, any of the four segment registers can be referenced,

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \{ \text{Direct Address} \}$$

The Execution Unit (EU) has direct access to all registers and data for register and immediate operands. However the EU cannot directly access the memory operands. It must use the BIU, in order to access memory operands.

In the direct addressing mode, the 16 bit effective address (EA) is taken directly from the displacement field of the instruction.

Example 1 : MOV CX, START

If the 16 bit value assigned to the offset START by the programmer using an assembler pseudo instruction such as DW is 0040 and [DS] = 3050.

Then BIU generates the 20 bit physical address 30540 H. The content of 30540 is moved to CL The content of 30541 is moved to CH

Example 2 : MOV CH, START

If [DS] = 3050 and START = 0040

8 bit content of memory location 30540 is moved to CH. Example 3 : MOV START, BX

With [DS] = 3050, the value of START is 0040. Physical address : 30540

1. Register indirect addressing mode :

The EA is specified in either pointer (BX) register or an index (SI or DI) register. The 20 bit physical address is computed using DS and EA.

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} = \begin{Bmatrix} BX \\ SI \\ DI \end{Bmatrix}$$

Example : MOV [DI], BX

If [DS] = 5004, [DI] = 0020, [Bx] = 2456 PA=50060.

The content of BX(2456) is moved to memory locations 50060 H and 50061 H.

2. Based addressing mode:

when memory is accessed PA is computed from BX and DS when the stack is accessed PA is computed from BP and SS.

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} BX \\ or \\ BP \end{Bmatrix} + \text{displacement}$$

Example : MOV AL, START [BX] or MOV AL, [START + BX] EA : [START] + [BX]

PA : [DS] + [EA]

The 8 bit content of this memory location is moved to AL.

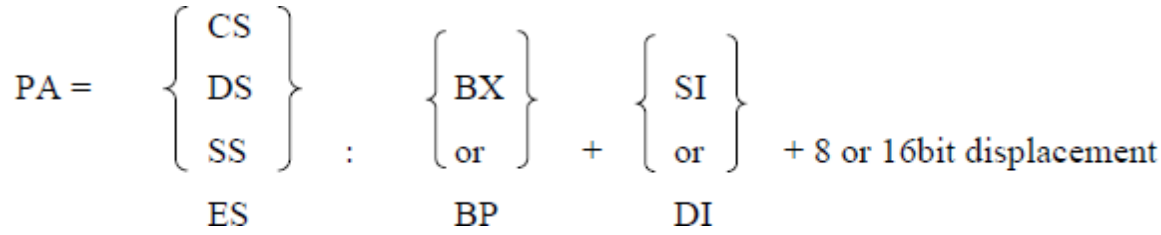
Indexed addressing mode:

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} SI \\ or \\ DI \end{Bmatrix} + 8 \text{ or } 16\text{bit displacement}$$

Example : MOV BH, START [SI] PA : [SART] + [SI] + [DS]

The content of this memory is moved into BH

Based Indexed addressing mode:



Example : MOV ALPHA [SI] [BX], CL

If [BX] = 0200, ALPHA – 08, [SI] = 1000 H and [DS] = 3000

Physical address (PA) = 31208

8 bit content of CL is moved to 31208 memory address.

Execution flow

The execution flow or control flow captures the order of evaluation of each instruction comprising the firmware in an embedded application, we can identify these as

- Sequential • Branch
- Loop
- Procedure or functional call

Sequential flow-sequential control flow describes the fundamental movement through a program. Each instruction contained in the program is executed in sequence one after the other.

Branch-

The control-flow of a language specify the order in which computations are performed The if-else statement is used to express decisions. Formally the syntax is

if (expression) statement1

else

statement2

Where the else part is optional. The expression is evaluated; if it is true (that is, if expression has a nonzero value), statement1 is executed. If it is false (expression is zero) and if there is an else part, statement2 is executed instead.

Since a if tests the numeric value of an expression, certain coding shortcuts are possible. The most obvious is writing

```
if (expression)
```

Instead of

```
if (expression != 0)
```

Sometimes this is natural and clear; at other times it can be cryptic

Procedure or function call

The procedure or function invocation is the most complex of the flow of control constructs. CALL operand - when PC is unconditionally saved and replaced by specified operand; the control is transferred to specified memory location.

RET – Previously saved contents of PC are restored, and control is returned to previous context.

Arithmetic and logic

Arithmetic and logic operations are essential elements in affecting what the processor is to do. such operations are executed by any og several hardware components comprising the ALU. Figure 2.8 presents a block diagram for a possible functional ALU architecture.

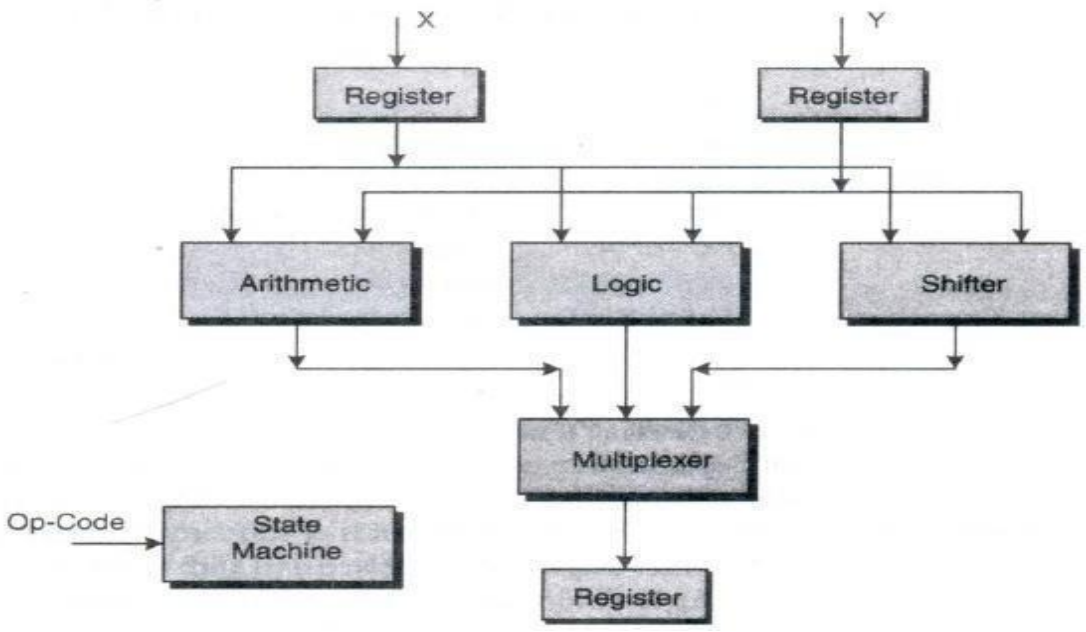


Figure 2.8 A block diagram for a possible functional ALU architecture.

Data is brought into the ALU and held in the local registers. The opcode is decoded, the appropriate operation is performed on the selected operands, and the result is stored in another local register.

Embedded system-A register view

At the ISA level, the instruction set specifies the basic operations supported by the machine-that is , the external view of the processor from the developer‘s perspective. The instruction set expresses the machine‘s ability to transfer data, store data , operate on data, and make decision. The core hardware comprises a control unit and a data path as illustrated in figure 2.9

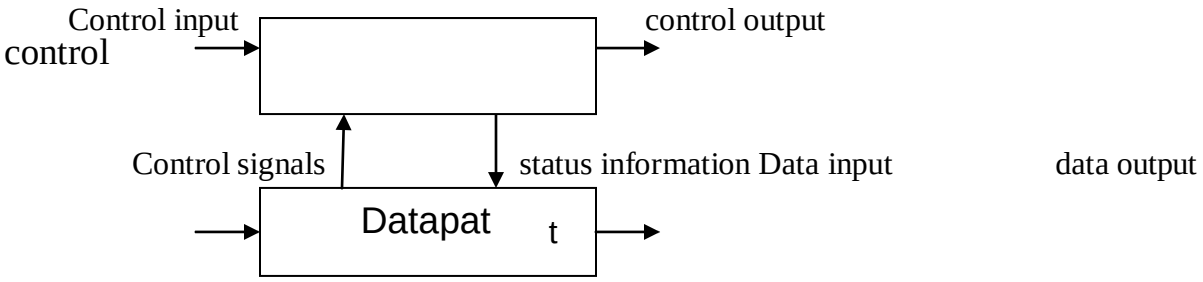


Figure 2.9 A control and datapath block diagram

The data path is a collection of registers and an associated set of micro operations on the data held in the registers. The control unit directs the ordered execution of the micro operations so as to effect the desired transformation of the data. Thus the system‘s behavior can be expressed by the movement of data among these registers, by operations and transformations performed on the register‘s contents, and by the management of how such movements and operations take place.the operations on data found at the instruction level are paralled by a similar, yet more detailed, set of operations at the register level.

Register view of a Microprocessor The datapath

Figure 2.10 expresses the architecture of the datapath and the memory interface for a simple microprocessor at the register transfer level.

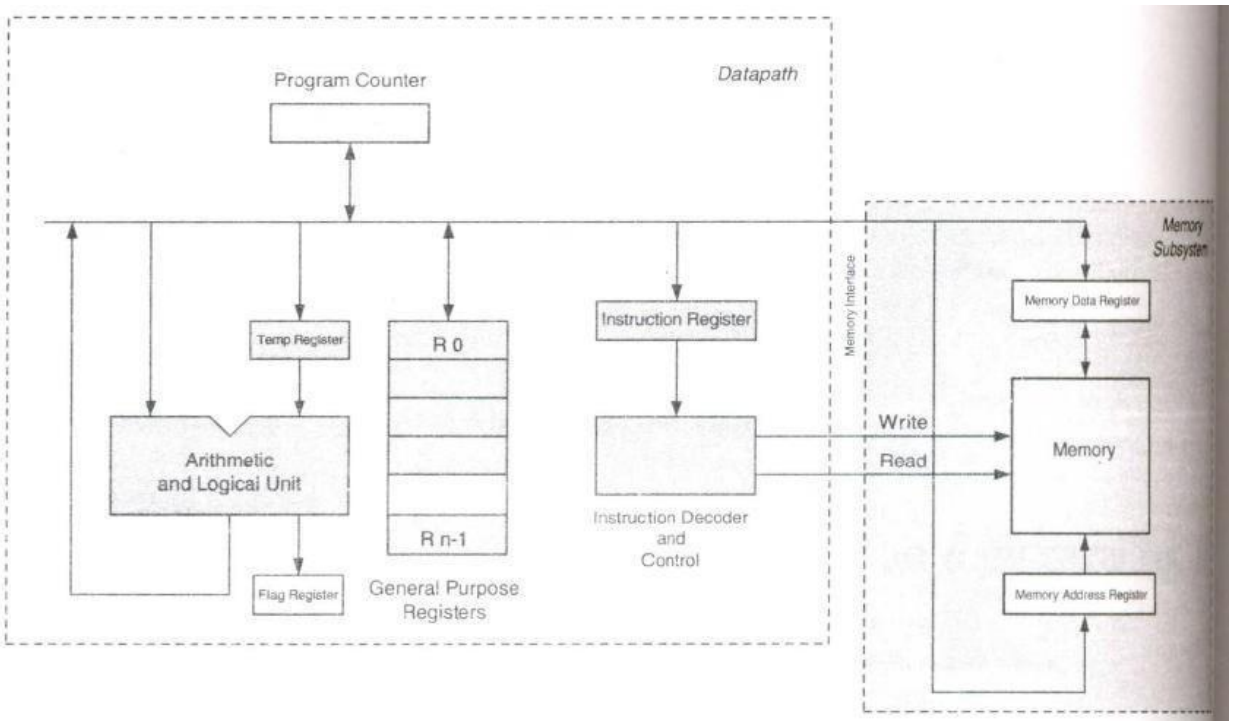


Figure 2.10 Architecture of the datapath and the memory interface

Processor control

The control of the microprocessor data path comprises four fundamental operations defined as the instruction cycle . The steps are identified in figure 2.11 .

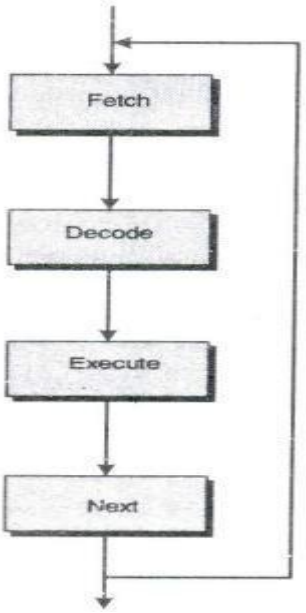


Figure 2.11 The instruction cycle

Fetch-The fetch operation retrieves an instruction from memory. That instruction is identified by its address, which is the contents of the PC.

Decode-The decode step is performed when the opcode field in the instruction is extracted from the instruction and decoded by the decoder. That information is forwarded to the control logic, which will initiate the execute portion of the instruction cycle.

Execute-Based on the value contained in the opcode field, the control logic performs the sequence of steps necessary to execute the instruction.

Next-The address of the next instruction to be executed is dependent on the type of instruction to be executed and potentially, on the state of the condition flags as modified by the recently completed instruction.

The Hardware side-storage elements and finite state machines

The logic devices that we have studied so far are combinational. The outputs of such circuitry are a function of inputs only, they are valid as long as the inputs are true. If the input changes, the output changes.

The concepts of State and Time

Time - A combinational logic system has no notion of time or history. The present output does not depend in any way on how the output values are achieved. Neglecting the delays through the system, we find that the output is immediate and a direct function of the current input set. Time is an integral part of the behavior of a system.

State -In an analog circuits, we define branch and mesh currents and branch or node voltages. The values these variables assume over time characterize the behavior of that circuit. If we know the values of the specified variables over time, we know the behavior of the circuit. Such variables are called state variables. We define the state of a system at any time as a set of values for such variables; each set of values represents a unique state.

State changes

In traditional logic, a simple memory device, represented by a single variable, has two states binary 1 and 0. The device will remain in the state until changed. For a set of variables, the state changes with time are called the behavior of a system.

The state diagram

In the embedded world, the state diagram is one of the means used to capture, describe and specify the behavior of a system. In a state diagram, each state is represented by a circle,node,or vertex. We label each node to identify the state. A memory device has two states-its output is a logical 0 or 1,thus to express its behavior we will need two nodes as shown in figure 2.12.

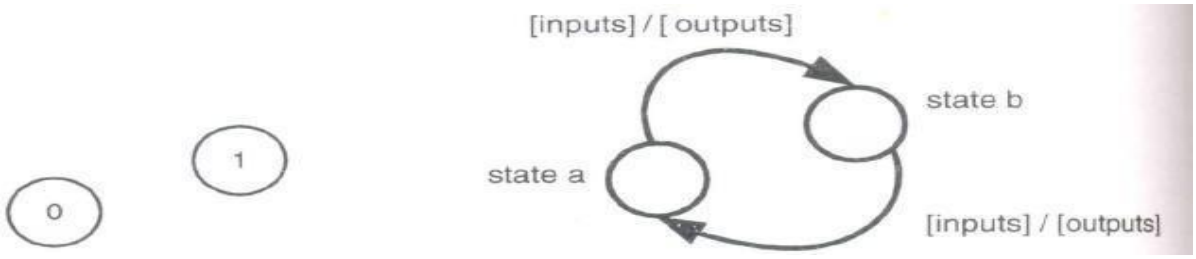


Figure 2.12 transition between states in a digital memory device

We show the transition between two states using a labeled directed line or arrow called arc as illustrated in figure because the line has a direction, the state diagram is referred to as a directed graph. The head o point of the arrow identifies the final state, and the tail or back of the arrow identifies the initial state.

Finite-state machine (FSM)- A theoretical model

A finite-state machine (FSM) or finite-state automaton (plural: automata), or simply a state machine, is a mathematical model of computation used to design both computer programs and sequential logic circuits. It is conceived as an abstract machine that can be in one of a finite number of states. The machine is in only one state at a time; the state it is in at any given time is called the current state. It can change from one state to another when initiated by a triggering event or condition; this is called a transition. A particular FSM is defined by a list of its states, and the triggering condition for each transition.

The behavior of state machines can be observed in many devices in modern society which perform a predetermined sequence of actions depending on a sequence of events with which they are presented. Simple examples are vending machines which dispense products when the proper combination of coins are deposited, elevators which drop riders off at upper floors before going down, traffic lights which change sequence when cars are waiting, and combination locks which require the input of combination numbers in the proper order.

Finite-state machines can model a large number of problems, among which are electronic design automation, communication protocol design, language parsing and other engineering applications. In biology and artificial intelligence research, state machines or hierarchies of state machines have been used to describe neurological systems and in linguistics—to describe the grammars of natural languages.

Figure shows a simple finite-state machine having no inputs other than a clock and have only primitive outputs. such machines are referred to as autonomous clocks. A high level block diagram for a finite-state machine begins with the diagram in figure 2.13.

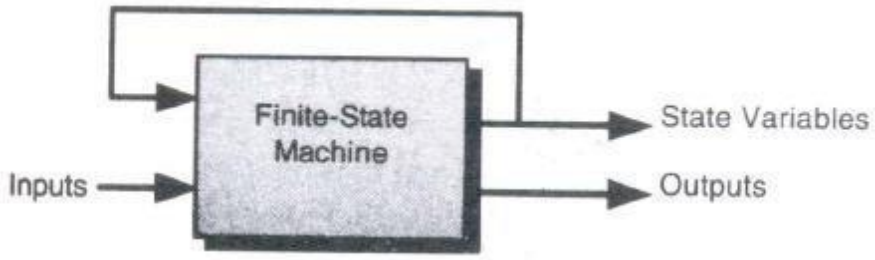


Figure 2.13 A high level block diagram for a finite-state machine

The output shown in the diagram may be the values of the state variables, combinations of the state variables ,or combinations of the state variables and the inputs.

MEMORY

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

(i) Internal RAM of 256 or 512 bytes in a microcontroller for registers, temporary data and stack.

(ii) (ii) Internal ROM/PROM/EPROM for about 4 kB to 16 kB of program (in the case of microcontrollers).

(iii) (iii) External RAM for the temporary data and stack (in most systems).

(iv) (iv) Internal caches (in the case of certain microprocessors).

(v) (v) EEPROM or flash (in many systems saving the results of processing in nonvolatile memory: for example, system status periodically and digital-camera images, songs, or speeches after a suitable format compression).

(vi) (vi) External ROM or PROM for embedding software (in almost all nonmicrocontroller- based systems).

(vii) (vii) RAM Memory buffers at the ports. (viii) Caches (in superscalar microprocessors).

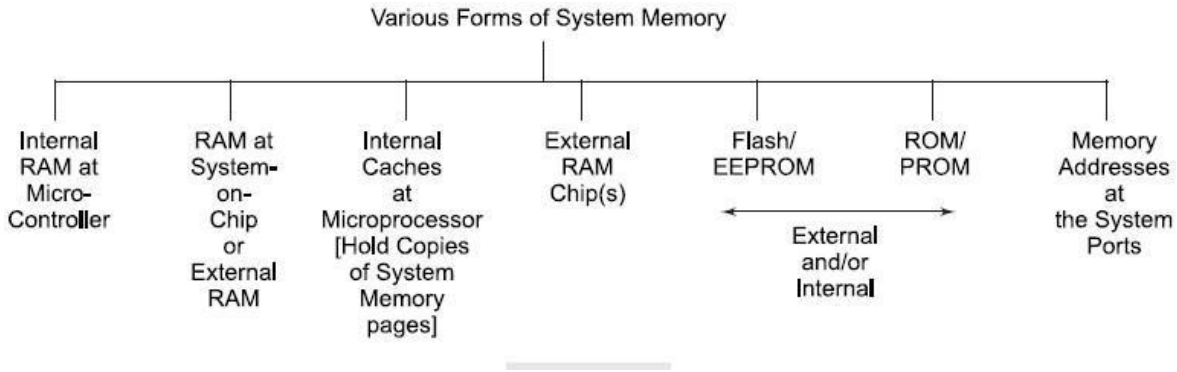


Table 1.1 gives the functions assigned in the embedded systems to the memories. ROM or PROM or EPROM embeds the embedded software specific to the system.

Table 1.1

Functions Assigned to the Memories in a System

Memory Needed	Functions
ROM or EPROM	Storing Application programs from where the processor fetches the instruction codes. Storing codes for system booting, initializing, Initial input data and Strings. Codes for RTOS. Pointers (addresses) of various service routines.
RAM (Internal and External) and RAM for buffer	Storing the variables during program run and storing the stack. Storing input or output buffers, for example, for speech or image.
EEPROM or Flash	Storing non-volatile results of processing.
Caches	Storing copies of instructions and data in advance from external memories and storing temporarily the results during fast processing.

The memory unit in an embedded system should have low access time and high density (a memory chip has greater density if it can store more bits in the same amount of space). Memory in an embedded system consists of ROM (only read operations permitted) and RAM (read and write operations are permitted). The contents of ROM are non-volatile (power failure does not erase the contents) while RAM is volatile. The classification of the ROM is given in Figure 3.1. ROM stores the program code while RAM is used to store transient input or output data. Embedded systems generally do not possess secondary storage devices such as magnetic disks. As programs of embedded systems are small there is no need for virtual storage.

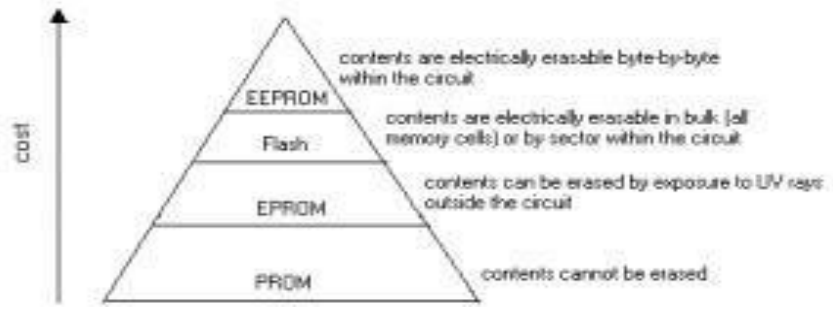


Figure 3.1: Classifications of ROM

Volatile memory

A primary distinction in memory types is volatility. Volatile memories only hold their contents while power is applied to the memory device. As soon as power is removed, the memories lose their contents; consequently, volatile memories are unacceptable if data must be retained when the memory is switched off. Examples of volatile memories include static RAM (SRAM),

synchronous static RAM (SSRAM), synchronous dynamic RAM (SDRAM), and FPGA on-chip memory.

Nonvolatile memory

Non-volatile memories retain their contents when power is switched off, making them good choices for storing information that must be retrieved after a system power-cycle. Processor boot-code, persistent application settings, and FPGA configurationdataaretypicallystoredinnon-volatilememory.Althoughnon-volatile memory has the advantage of retaining its data when power is removed, it is typically much slower to write to than volatile memory, and often has more complex writing and erasing procedures. Non-volatile memory is also usually only guaranteed to be erasable a given number of times, after which it may fail. Examples of non- volatile memories include all types of flash, EPROM, and EEPROM. Most modern embedded systems use some type of flash memory for non-volatile storage. Many embedded applications require both volatile and non-volatile memories because the two memory types serve unique and exclusive purposes. The following sections discuss the use of specific types of memory in embedded systems.

ROM Overview

Although there are exceptions, the ROM is generally viewed as read only device. A high level interface to the ROM is as shown in figure 3.2. when the ROM is implemented,positions in the array that are to store a logical 0 have a transistor connected as shown in figure. Those positions intended to store a logical 1 have none.

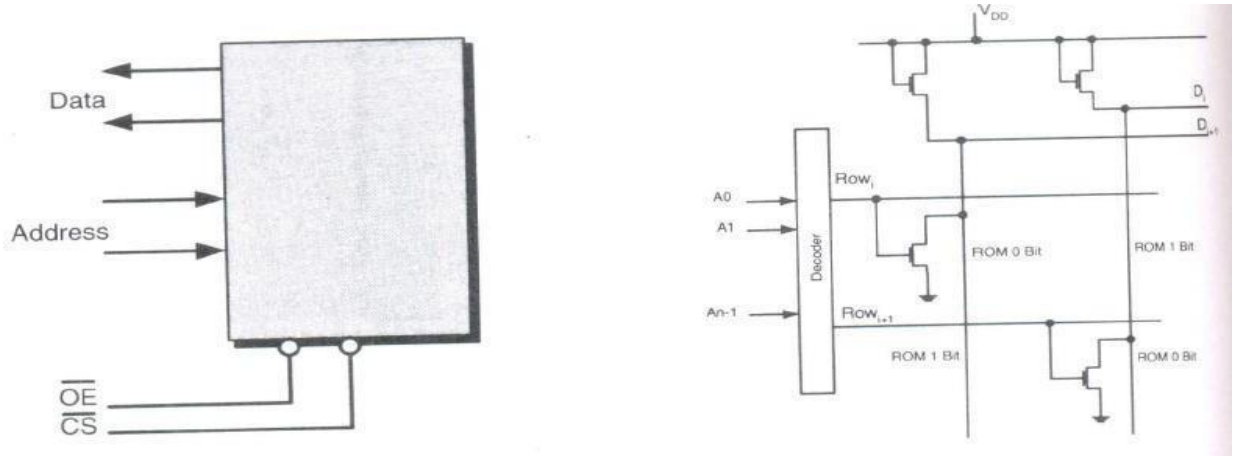


Figure 3.2 The ROM- outside and inside

Read Operation

A value is read from a ROM by asserting one of the row lines. Those rows in which there is a transistor will be pulled to ground thereby expressing a logical 0. Those without the transistor will express a logical 1. Timing for a ROM read operation is given in figure 3.3.

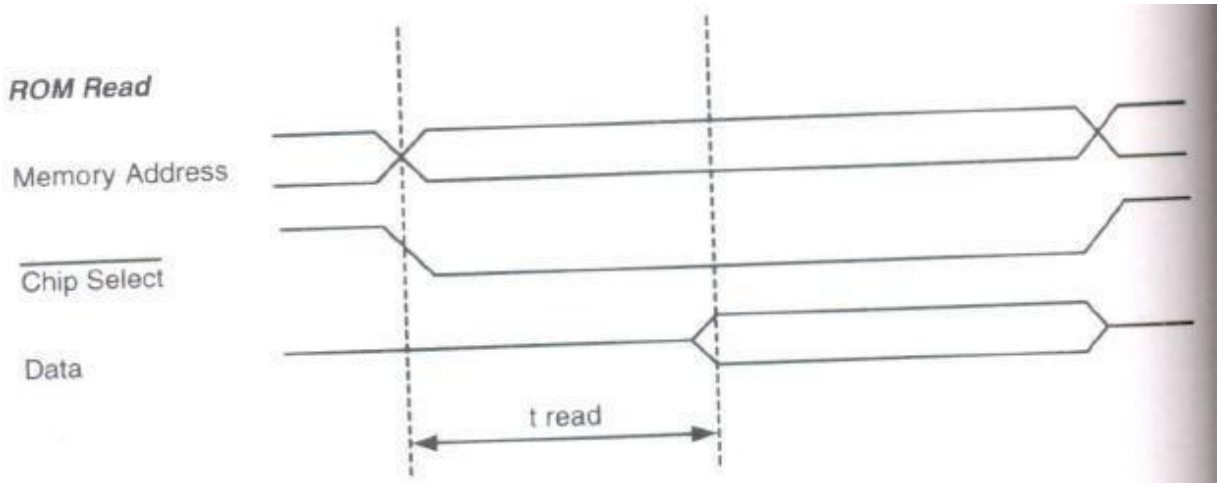


Figure 3.3 The ROM –read operation timing

Static RAM overview

A high level interface to the SRAM is very similar to that for the ROM.The major differences arise from support for write capability. Figure 3.4 represents the major I/O signals and a typical cell in an SRAM array.

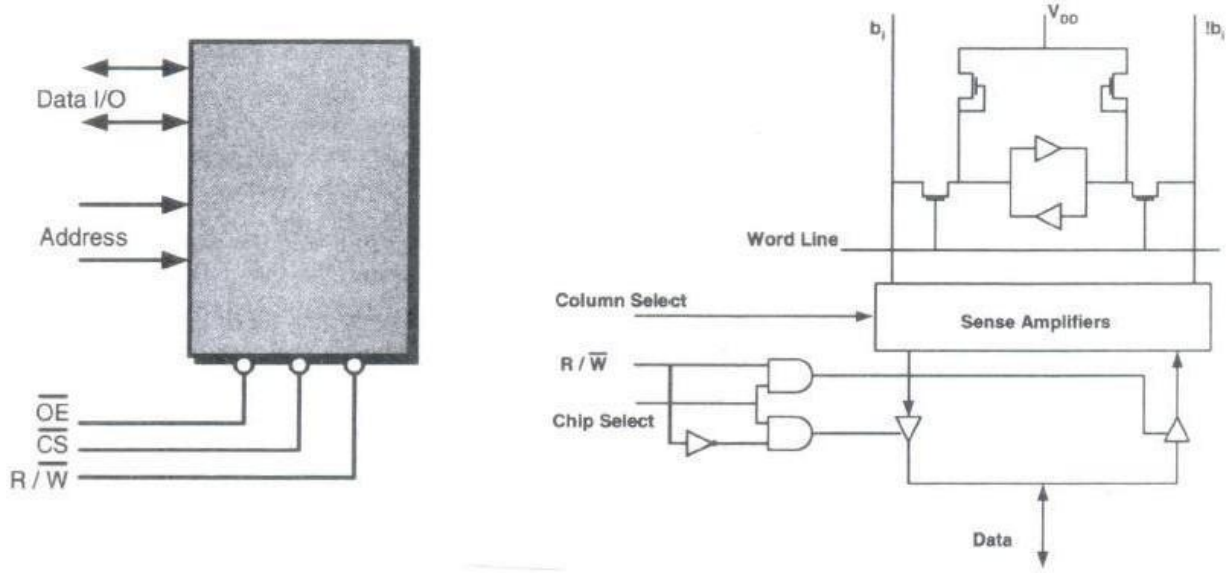


Figure 3.4 The SRAM – inside and outside

Write operation

A value is written into the cell by applying a signal to bi and bi_{bar} through the write/sense amplifiers. Asserting the word line causes the new value to be written into the latch.

Read Operation

A value is read from the cell by first precharging bi and bi_{bar} to a voltage that is halfway between a 0 and 1.the values are sensed and amplified by write/sense amplifier.

Typical timimg for a read and write operation is shown in Figure 3.5 .

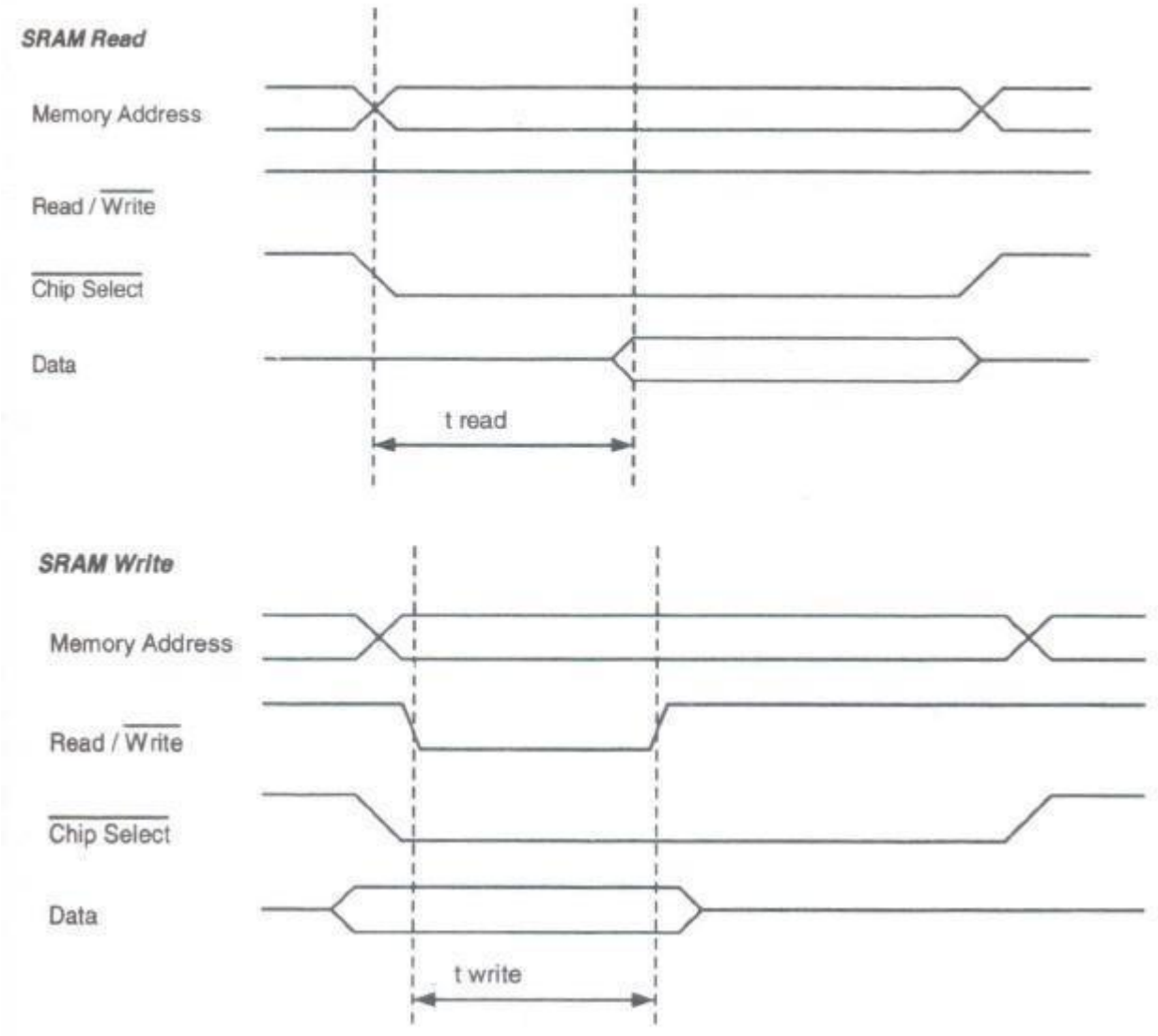


Figure 3.5 timing for the SRAM-read and write operation

SDRAM

SDRAM is another type of volatile memory. It is similar to SRAM, except that it is dynamic and must be refreshed periodically to maintain its content. The dynamic memory cells in SDRAM are much smaller than the static memory cells used in SRAM. This difference in size translates into very high-capacity and low-cost memory devices. In addition to the refresh requirement, SDRAM has other very specific interface requirements which typically necessitate the use of special controller hardware. Unlike SRAM, which has a static set of address lines, SDRAM divides up its memory space into banks, rows, and columns. Switching between banks and rows incurs some overhead, so that efficient use of SDRAM involves the careful ordering of accesses. SDRAM also multiplexes the row and column addresses over the same address lines, which reduces the pin count necessary to implement a given size of SDRAM. Higher speed varieties of SDRAM such as DDR, DDR2, and DDR3 also have strict signal integrity requirements which need to be carefully considered during the design of the PCB. SDRAM devices are among the least expensive and largest-capacity types of RAM devices available, making them one of the most popular. Most modern embedded systems use SDRAM. A major part of an SDRAM interface is the SDRAM controller. The SDRAM controller manages all the address-multiplexing, refresh and row and bank switching tasks, allowing the rest of the system to access SDRAM without knowledge of its internal architecture.

Dynamic RAM Overview

Larger microcomputer systems use Dynamic RAM (DRAM) rather than Static RAM (SRAM) because of its lower cost per bit. DRAMs require more complex interface circuitry because of their multiplexed address bus and because of the need to refresh each memory cell periodically.

A typical DRAM memory is laid out as a square array of memory cells with an equal number of rows and columns. Each memory cell stores one bit. The bits are addressed by using half of the bits (the most significant half) to select a row and the other half to select a column.

Each DRAM memory cell is very simple – it consists of a capacitor and a MOSFET switch. A DRAM memory cell is therefore much smaller than an SRAM cell which needs at least two gates to implement a flip-flop. A typical DRAM array appears as illustrated in figure 3.6 .

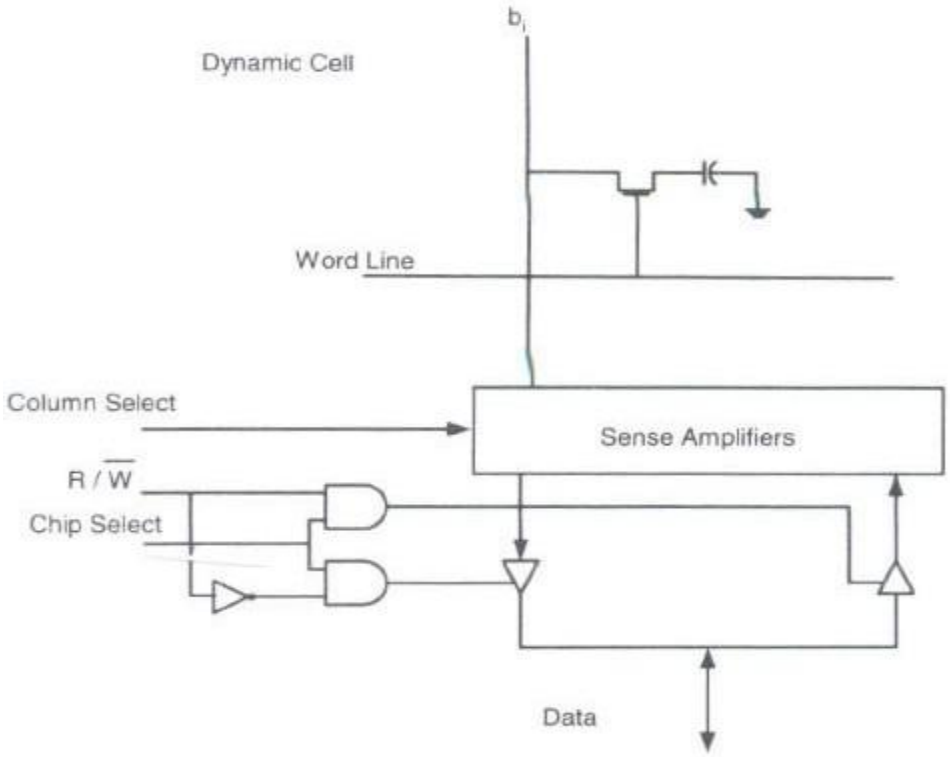


Figure 3.6 The DRAM inside

Read Operation

A value is read from the cell by first precharging bi and bi_{bar} to a voltage that is halfway between a 0 and 1. Asserting the word line enables the stored signal onto bi. If the stored value is a logical 1,through charge sharing,the value on line bi will increase. Conversely,if the stored value is a logical 0, charge sharing will cause the value on bi to decrease. The change in the values are sensed and amplified by write/sense amplifier

Write operation

A value is written into the cell by applying a signal to bi and bi_{bar} through the write/sense amplifiers. Asserting the word line charges the capacitor if a logical 1 is to be stored and discharges if it a logical 0 to be stored.

Typical DRAM read and write timing is given in figure 3.7 .

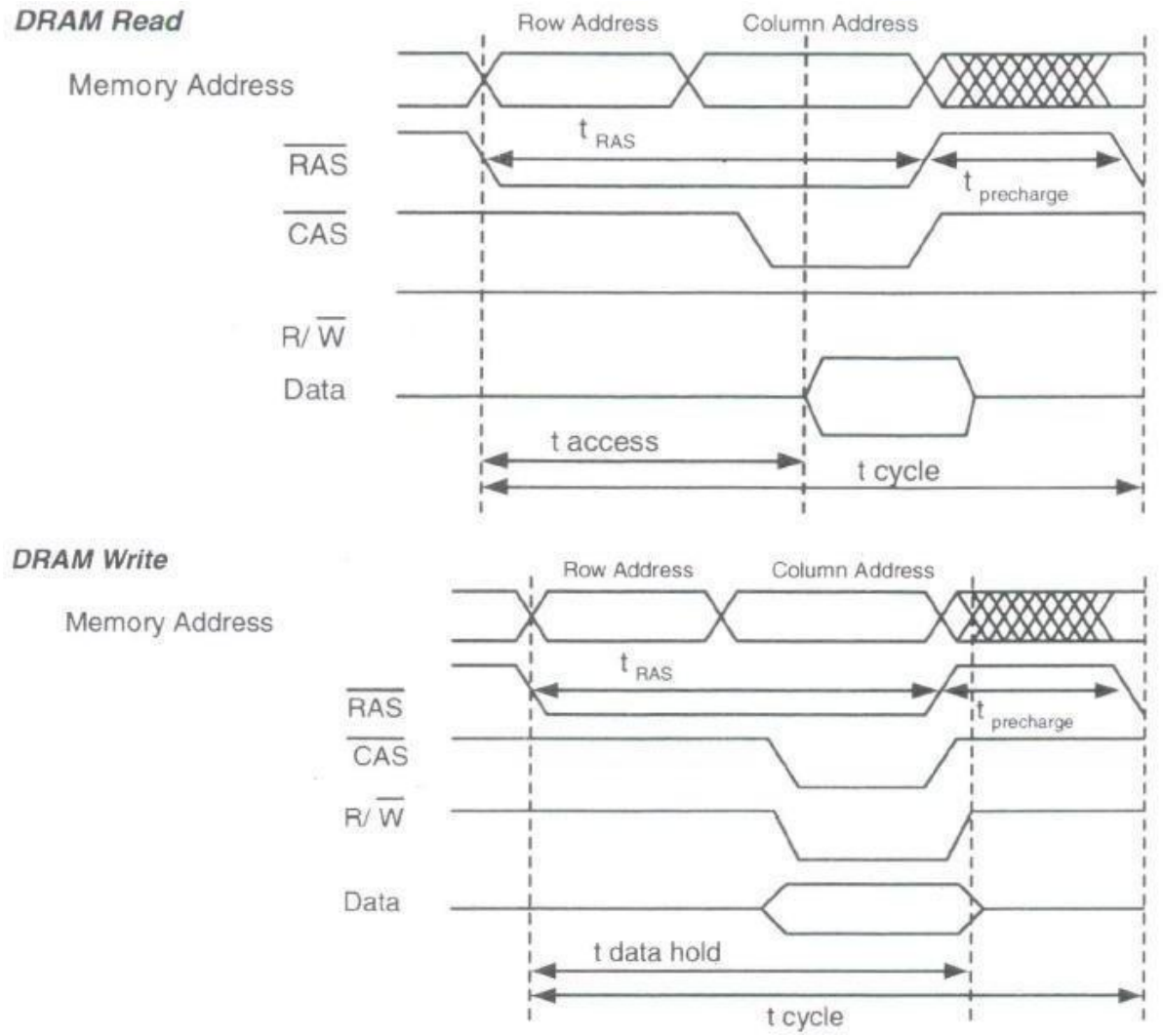


Figure 3.7 : Timing for the DRAM read and write cycles

Chip organization

Independent type of internal storage, the typical memory chip appears as is shown in figure 3.8

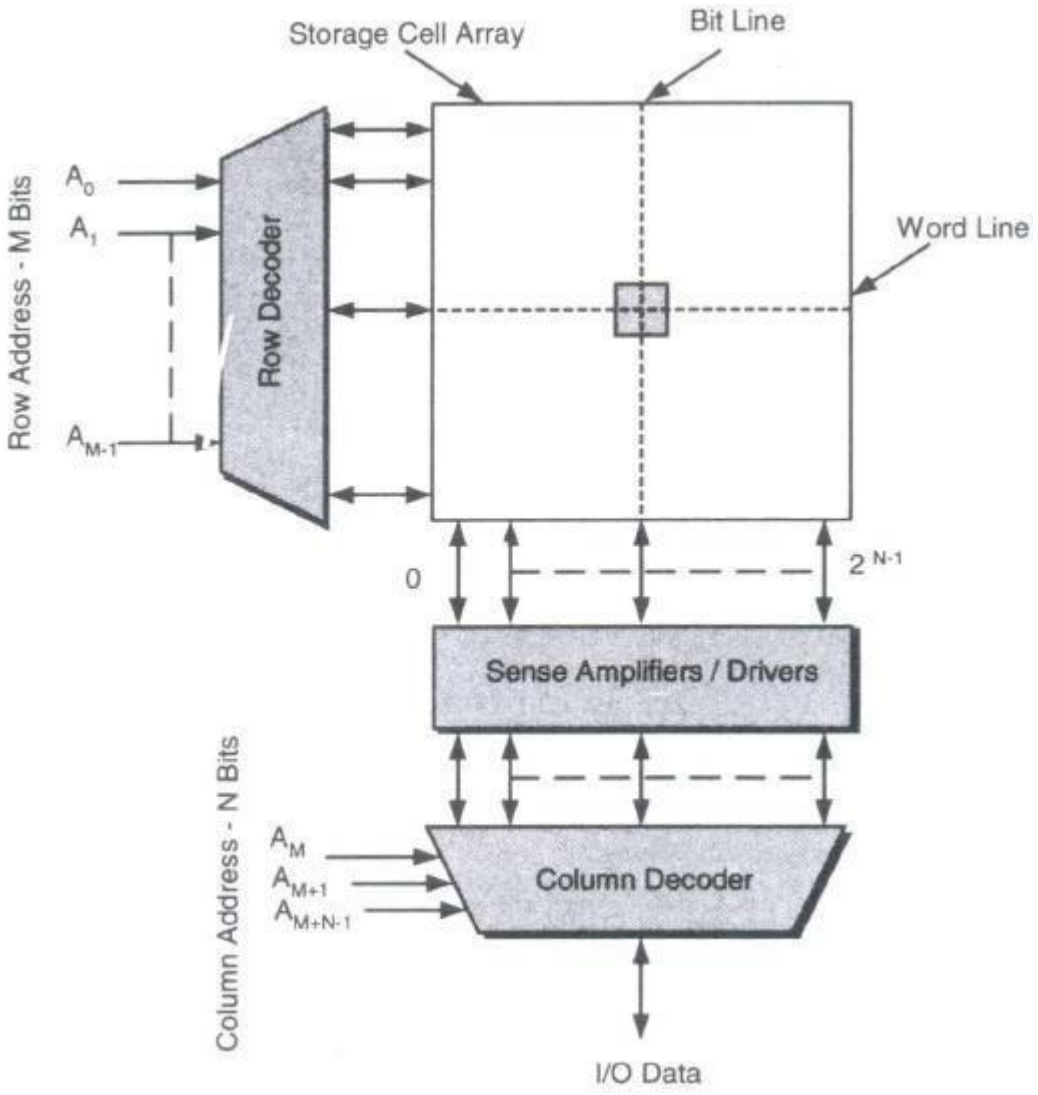


Figure 3.8 Typical memory chip internal architecture.

A Memory interface in detail

If a single ROM or RAM chip is large enough and the address and the data I/O are wide enough to satisfy system memory requirements,then the interface is rather straightforward. We will look at the SRAM system and then the DRAM design.

An SRAM design

A system specification requires an SRAM system that can store upto 4K 16 bit words..but the largest memory device available is 1K by 8. Thus it can store upto 1024 8 bit words. Consequently ,the design will require 8 of the smaller memorydevices:2 sets of 4.

In worst case, to support 4K 16 bit words,12 address lines and 16 data lines are required. The architecture of such system is given in figure 3.9

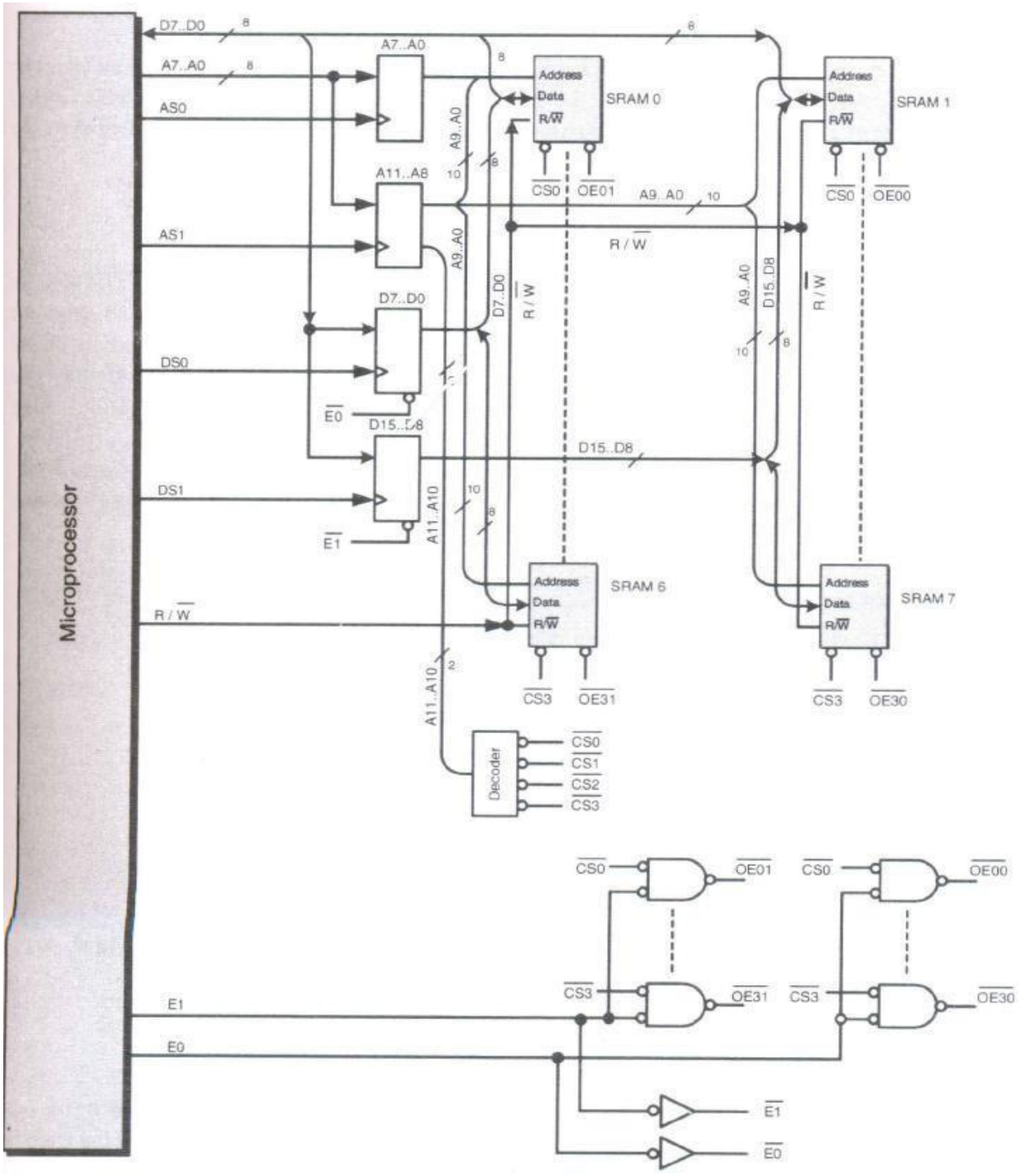


Figure 3.9: Design a 4K *16 SRAM system

A DRAM Design

The DRAM system will utilize an architecture that duplicates most of the previous SRAM systems. One major difference is the potential need to manage the refresh function. A second difference results from a memory size versus IC package size difficulty.

The relative timing for some of the signals in the base case is illustrated in figure 3.10.

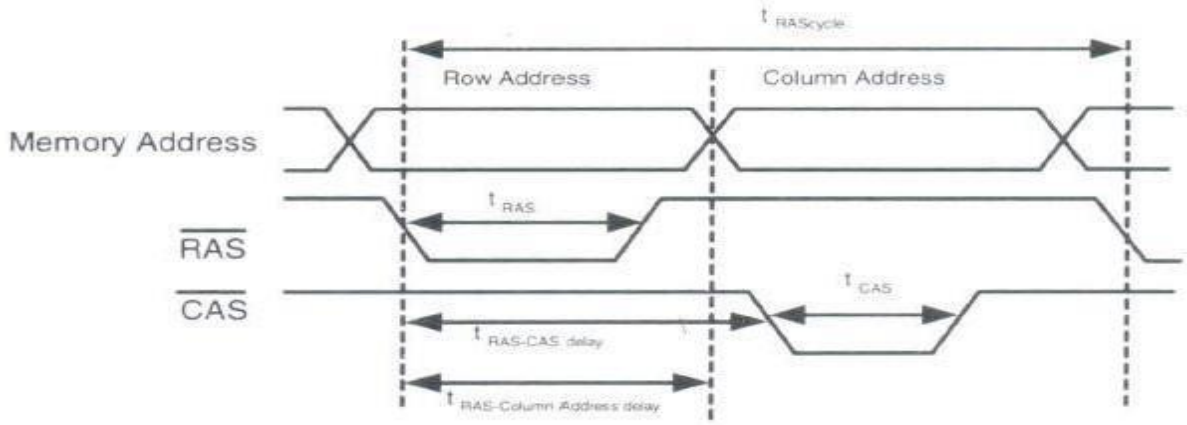


Figure 3.10 Basic DRAM timing

The memory map

As a first step towards understanding the memory subsystem in an embedded application, we begin with a memory map. The map specifies the allocation and use of each location in the physical memory address space. A typical memory map for a small 16 bit machine is presented in figure 3.11

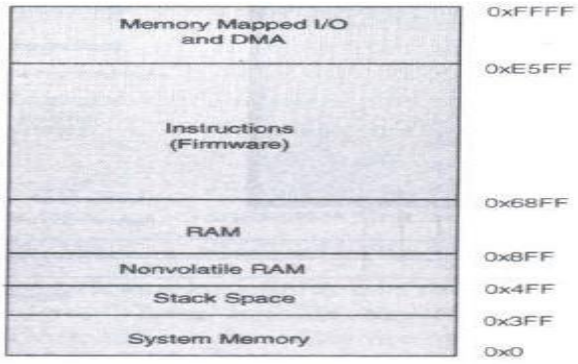


Figure 3.11 : Basic Memory map

This is the primary physical memory. From a high level perspective, the memory subsystem is comprised of two basic types:ROM and RAM. It is possible for the required code and data space to exceed total available primary memory.under such circumstances, one must use techniques called virtual memory and overlays to accommodate the expanded needs.

Memory subsystem Architecture

The block labeled memory in the diagrrm for a vonneumann machine is sctually comprised of a memory components of different size, kinds, and speeds arranged in a heirarchial manner and designed to coopereate with each other.such a hierarchy is given in figure 3.12 .

Figure 3.12 : Typical memory hierarchy utilizing a variety of memory types.

At the top are the slowest,largest and least expensive memories. These are known as secondary memory. At the bottom are the smallest,fastest,called cache memory. These are typically higher speed SRAM. These devices are more expensive.

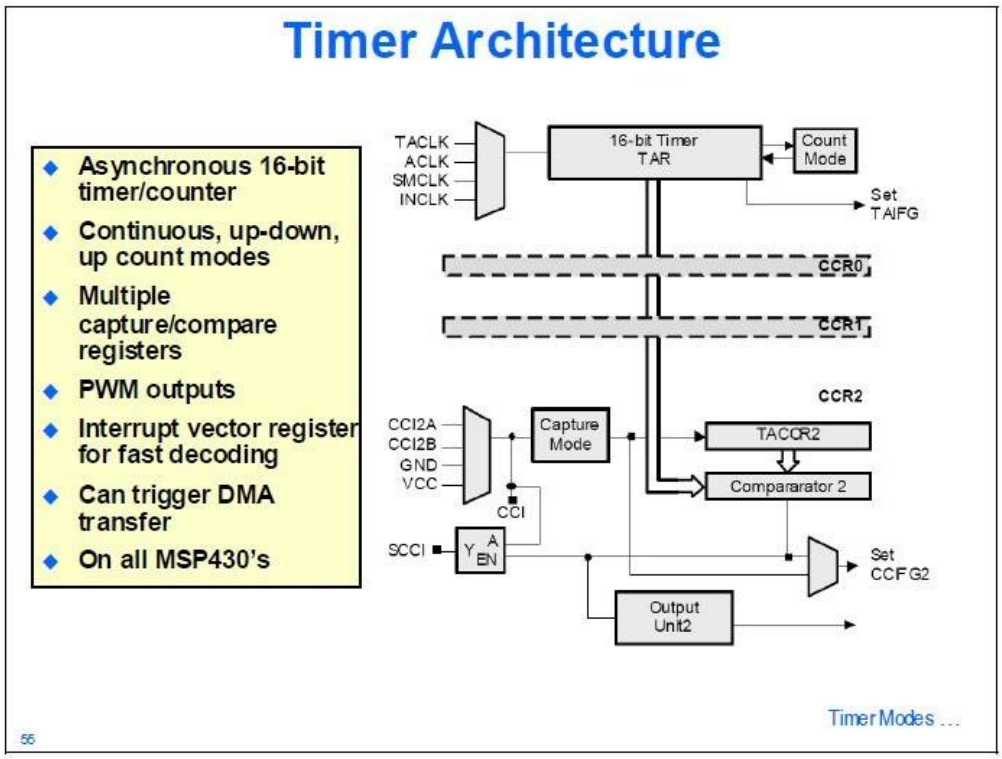
UNIT III

EMBEDDED FIRMWARE

Timer & Real Time Clock (RTC), PWM control, timing generation and measurements. Analog interfacing and data acquisition: ADC and Comparator in MSP430, data transfer using DMA.

Case Study: MSP430 based embedded system application using ADC & PWM demonstrating peripheral intelligence. “Remote Controller of Air Conditioner Using MSP430”.

Timer Architecture



Timer

Timer_A is a 16-bit timer/counter with multiple capture/compare registers. Timer_A is a 16-bit timer/counter with up to seven capture/compare registers. Timer_A can support

multiple capture/compares, PWM outputs, and interval timing. Timer_A also has extensive interrupt capabilities. Interrupts may be generated from the counter on overflow conditions and from each of the capture/compare registers.

Timer_A features include:

Asynchronous 16-bit timer/counter with four operating modes

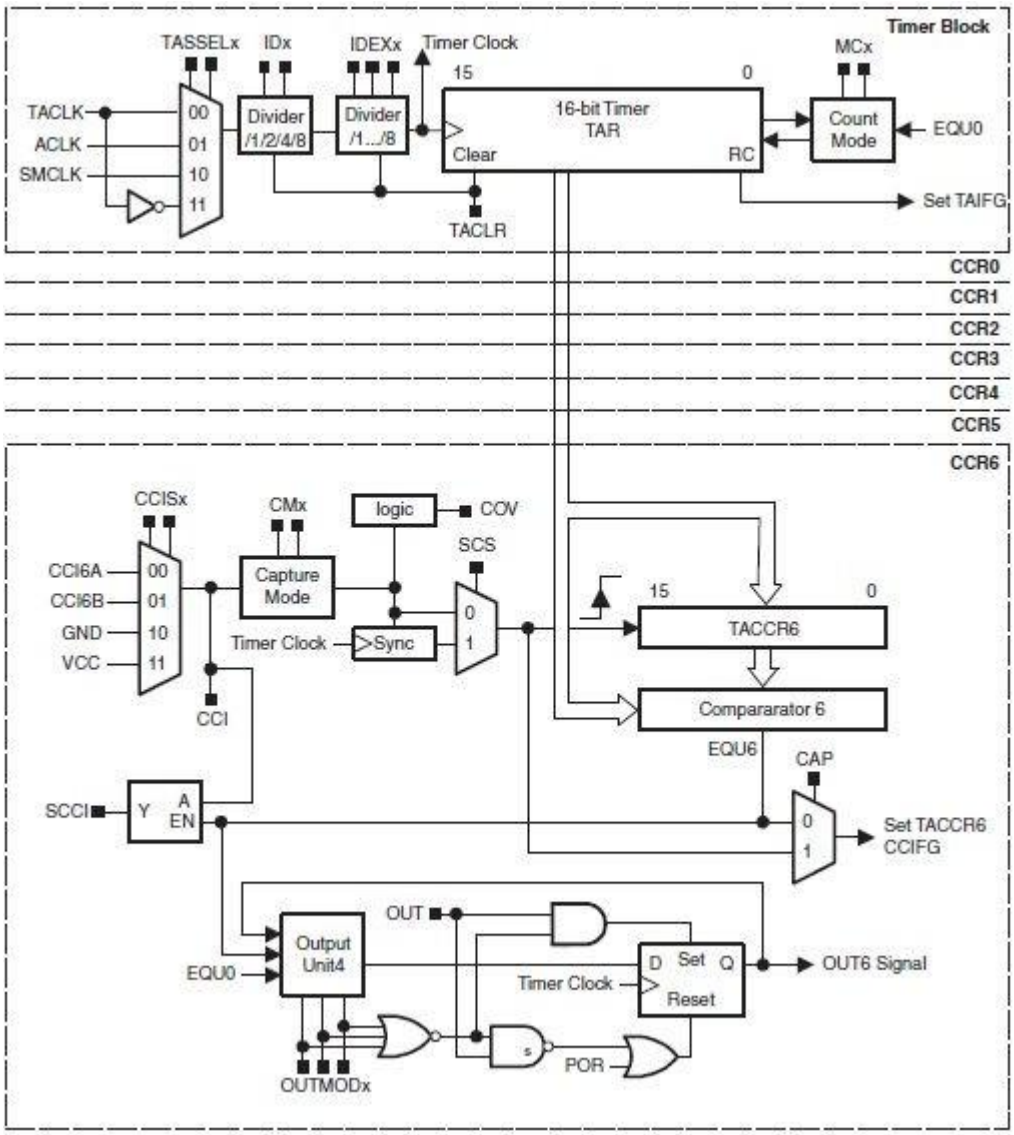
Selectable and configurable clock source

Up to seven configurable capture/compare registers

Configurable outputs with PWM capability

Asynchronous input and output latching

Interrupt vector register for fast decoding of all Timer_A interrupts



The 16-bit timer/counter register, TAR, increments or decrements (depending on mode of operation) with each rising edge of the clock signal. TAR can be read or written with software. Additionally, the timer can generate an interrupt when it overflows.

TAR may be cleared by setting the TACLR bit. Setting TACLR also clears the clock divider

and count direction for up/down mode.

Clock Source Select and Divider

The timer clock TACLK can be sourced from ACLK, SMCLK, or externally via TACLK. The clock source is selected with the TASSELx bits. The selected clock source may be passed directly to the timer or divided by 2, 4, or 8, using the IDx bits The selected clock source can be further divided by 2, 3, 4, 5, 6, 7, or 8 using the IDEXx bits.The TACLK dividers are reset when TACLR is set.

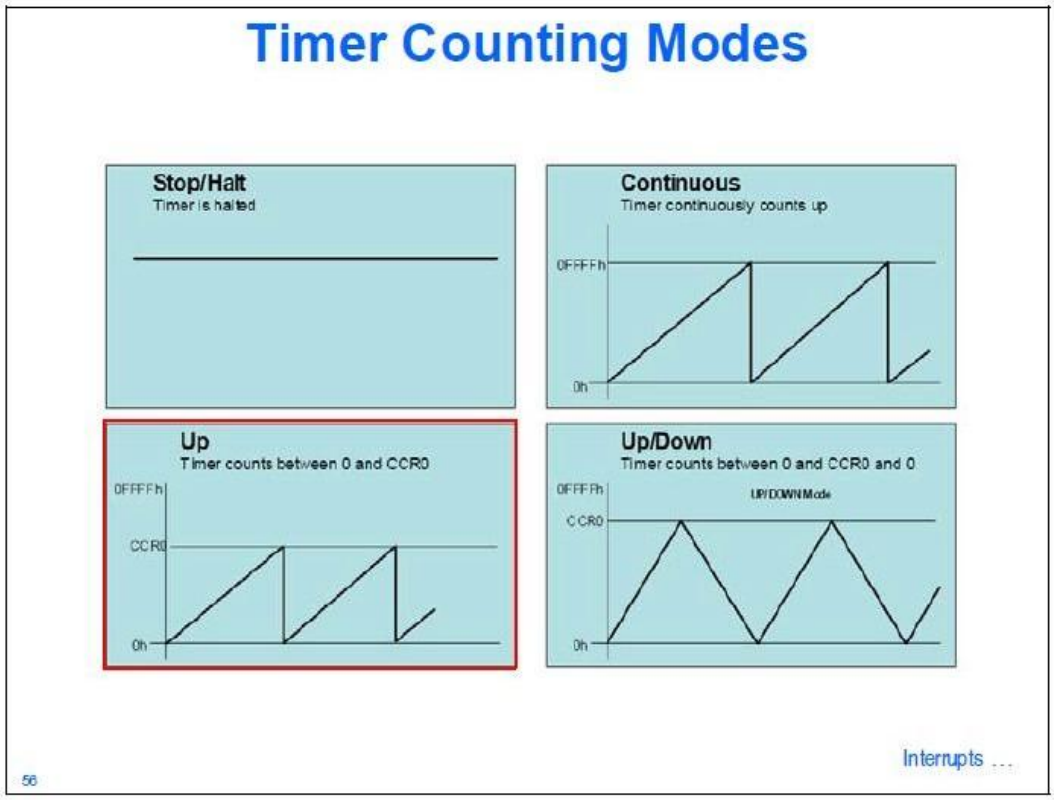
The timer may be started, or restarted in the following ways:

The timer counts when MCx > 0 and the clock source is active.

When the timer mode is either up or up/down, the timer may be stopped by writing 0 to TACCR0. The timer may then be restarted by writing a nonzero value to TACCR0. In this scenario, the timer starts incrementing in the up direction from zero.

MCx	Mode	Description
00	Stop	The timer is halted.
01	Up	The timer repeatedly counts from zero to the value of TACCR0
10	Continuous	The timer repeatedly counts from zero to 0FFFFh.
11	Up/down	The timer repeatedly counts from zero up to the value of TACCR0 and backdown to zero.

Counting Modes



Up Mode

The up mode is used if the timer period must be different from 0FFFFh counts. The timer repeatedly counts up to the value of compare register TACCR0, which defines the period, as shown in Figure. The number of timer counts in the period is TACCR0+1. When the timer value equals TACCR0 the timer restarts counting from zero. If up mode is selected when the timer value is greater than TACCR0, the timer immediately restarts counting from zero

The TACCR0 CCIFG interrupt flag is set when the timer counts to the TACCR0 value. The TAIFG interrupt flag is set when the timer counts from TACCR0 to zero.

Continuous Mode:

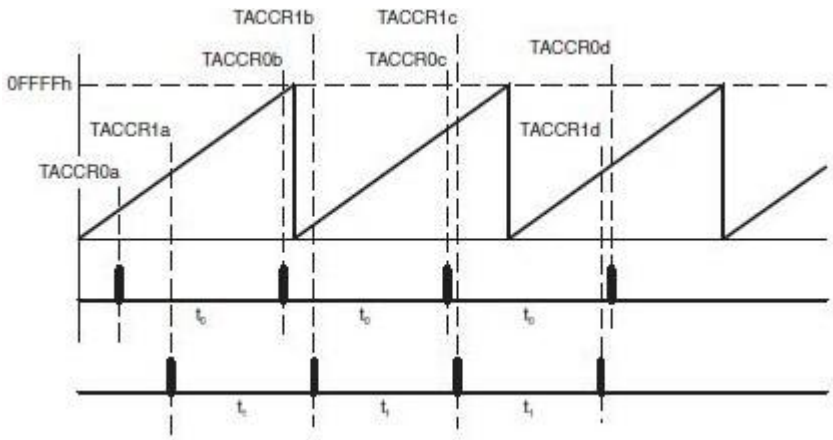
In the continuous mode, the Continuous Mode timer repeatedly counts up to 0FFFFh and restarts from zero as shown in Figure. The capture/compare register TACCR0 works the same way as the other capture/compare registers.

The TAIFG interrupt flag is set when the timer counts from 0FFFFh to zero

Use of the Continuous Mode

The continuous mode can be used to generate independent time intervals and output frequencies. Each time an interval is completed, an interrupt is generated. The next time interval is added to the TACCRx register in the interrupt service routine. Figure shows two separate time intervals t0 and t1 being added to the capture/compare registers. In this usage, the time interval is controlled by hardware, not software, without impact from interrupt latency. Up to n (Timer_An), where n = 0 to 7, independent time intervals or output frequencies can be generated using capture/compare registers.

Time intervals can be produced with other modes as well, where TACCR0 is used as the period register. Their handling is more complex since the sum of the old TACCRx data and the new period can be higher than the TACCR0 value. When the previous TACCRx value plus tx is greater than the TACCR0 data, the TACCR0 value must be subtracted to obtain the correct time interval.



Up/Down Mode

The up/down mode is used if the timer period must be different from 0FFFFh counts, and if symmetrical pulse generation is needed. The timer repeatedly counts up to the value of compare register TACCR0 and back down to zero,

The count direction is latched. This allows the timer to be stopped and then restarted in the same direction it was counting before it was stopped. If this is not desired, the TACLR bit must be set to clear the direction. The TACLR bit also clears the TAR value and the TACLK divider.

In up/down mode, the TACCR0 CCIFG interrupt flag and the TAIFG interrupt flag are set

only once during a period, separated by 1/2 the timer period. The TACCR0 CCIFG interrupt flag is set when the timer counts from TACCR0-1 to TACCR0, and TAIFG is set when the timer completes counting down from 0001h to 0000h.

Use of the Up/Down Mode

The up/down mode supports applications that require dead times between output signals (see section *Timer_A Output Unit*). For example, to avoid overload conditions, two outputs driving an H-bridge must never be in a high state simultaneously. In the example shown in Figure 12-9 the t_{dead} is:

$t_{dead} = t_{timer} \times (TACCR1 - TACCR2)$

With:

t_{dead} = Time during which both outputs need to be inactive
t_{timer} = Cycle time of the timer clock
TACCRx = Content of capture/compare register x

The TACCRx registers are not buffered. They update immediately when written to. Therefore, any required dead time will not be maintained automatically.

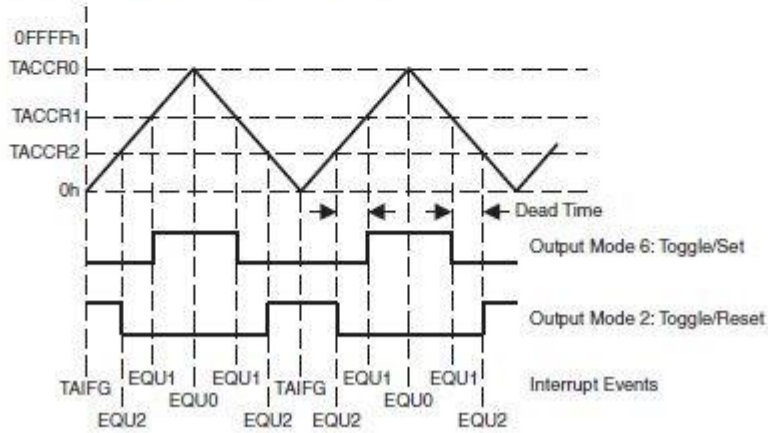


Figure 12-9. Output Unit in Up/Down Mode

Capture/Compare Blocks:

Three or five identical capture/compare blocks, TACCRx, are present in Timer_A. Any of the blocks may be used to capture the timer data, or to generate time intervals. Capture Mode The capture mode is selected when CAP = 1. Capture mode is used to record time events. It can be used for speed computations or time measurements. The capture inputs CCIxA and CCIxB are connected to external pins or internal signals and are selected with the CCISx bits. The CMx bits select the capture edge of the input signal as rising, falling, or both. A capture occurs on the selected edge of the input signal. If a capture occurs:

The timer value is copied into the TACCRx register

The interrupt flag CCIFG is set

The input signal level can be read at any time via the CCI bit. MSP430x5xx family devices may have different signals connected to CCIxA and CCIxB. Refer to the device-specific data sheet for the connections of these signals. The capture signal can be asynchronous to the timer clock and cause a race condition. Setting the SCS bit will synchronize the capture with the next timer clock. Setting the SCS bit to synchronize the capture signal with the timer clock is recommended.

Overflow logic is provided in each capture/compare register to indicate if a second capture was performed before the value from the first capture was read. Bit COV is set

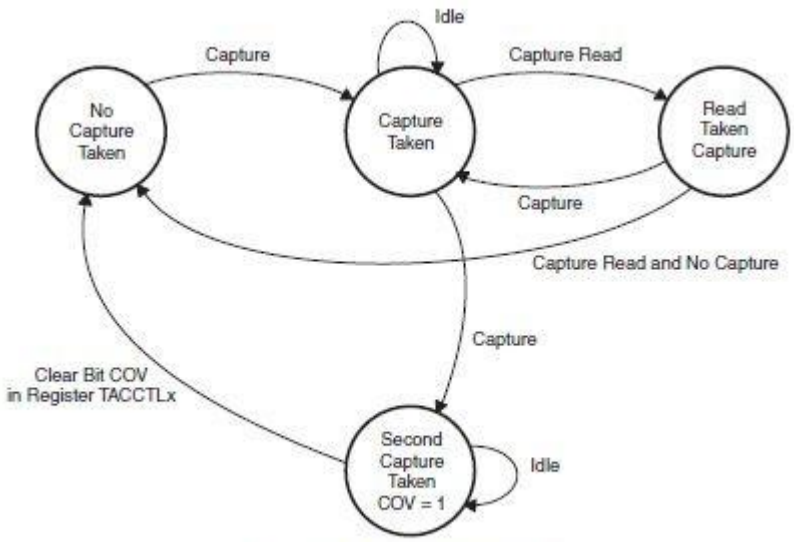


Figure 12-11. Capture Cycle

Capture Mode:

Captures can be initiated by software. The CMx bits can be set for capture on both edges. Software then sets CCIS1 = 1 and toggles bit CCIS0 to switch the capture signal between VCC and GND, initiating a capture each time CCIS0 changes state:

MOV #CAP+SCS+CCIS1+CM_3,&TACCTLx ; Setup TACCTLx XOR #CCIS0,&TACCTLx ; TACCTLx = TAR

Compare Mode

The compare mode is selected when CAP = 0. The compare mode is used to generate PWM output signals or interrupts at specific time intervals. When TAR counts to the value in a TACCRx:

Interrupt flag CCIFG is set

Internal signal EQUx = 1

EQUx affects the output according to the output mode

The input signal CCI is latched into SCCI

Output Unit:

Each capture/compare block contains an output unit. The output unit is used to generate output signals such as PWM signals. Each output unit has eight operating modes that generate signals based on the EQU0 and EQUx signals.

Output Modes

The output modes are defined by the OUTMODx bits and are described in Table. The OUTx signal is changed with the rising edge of the timer clock for all modes except mode 0. Output modes 2, 3, 6, and 7 are not useful for output unit 0 because EQUx = EQU0.

Output Example—Timer in Up Mode

The OUTx signal is changed when the timer *counts* up to the TACCRx value, and rolls from TACCR0 to zero, depending on the output mode. An example is shown in Figure 12-12 using TACCR0 and TACCR1.

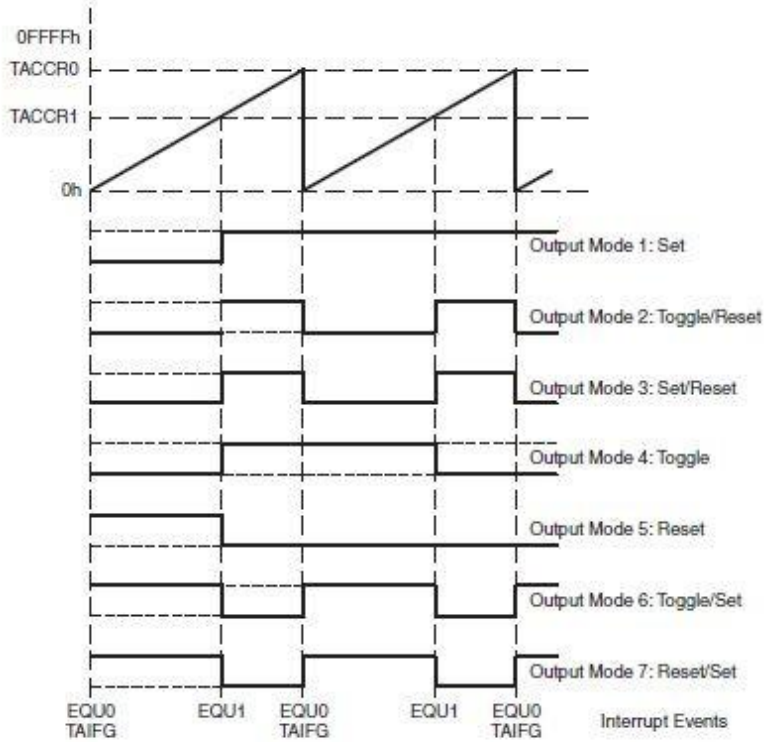


Figure 12-12. Output Example—Timer in Up Mode

Table 12-2. Output Modes

OUTMODx	Mode	Description
000	Output	The output signal OUTx is defined by the OUTx bit. The OUTx signal updates immediately when OUTx is updated.
001	Set	The output is set when the timer *counts* to the TACCRx value. It remains set until a reset of the timer, or until another output mode is selected and affects the output.
010	Toggle/Reset	The output is toggled when the timer *counts* to the TACCRx value. It is reset when the timer *counts* to the TACCR0 value.
011	Set/Reset	The output is set when the timer *counts* to the TACCRx value. It is reset when the timer *counts* to the TACCR0 value.
100	Toggle	The output is toggled when the timer *counts* to the TACCRx value. The output period is double the timer period.
101	Reset	The output is reset when the timer *counts* to the TACCRx value. It remains reset until another output mode is selected and affects the output.
110	Toggle/Set	The output is toggled when the timer *counts* to the TACCRx value. It is set when the timer *counts* to the TACCR0 value.
111	Reset/Set	The output is reset when the timer *counts* to the TACCRx value. It is set when the timer *counts* to the TACCR0 value.

Output Example—Timer in Continuous Mode

The OUTx signal is changed when the timer reaches the TACCRx and TACCR0 values, depending on the output mode. An example is shown in Figure 12-13 using TACCR0 and TACCR1.

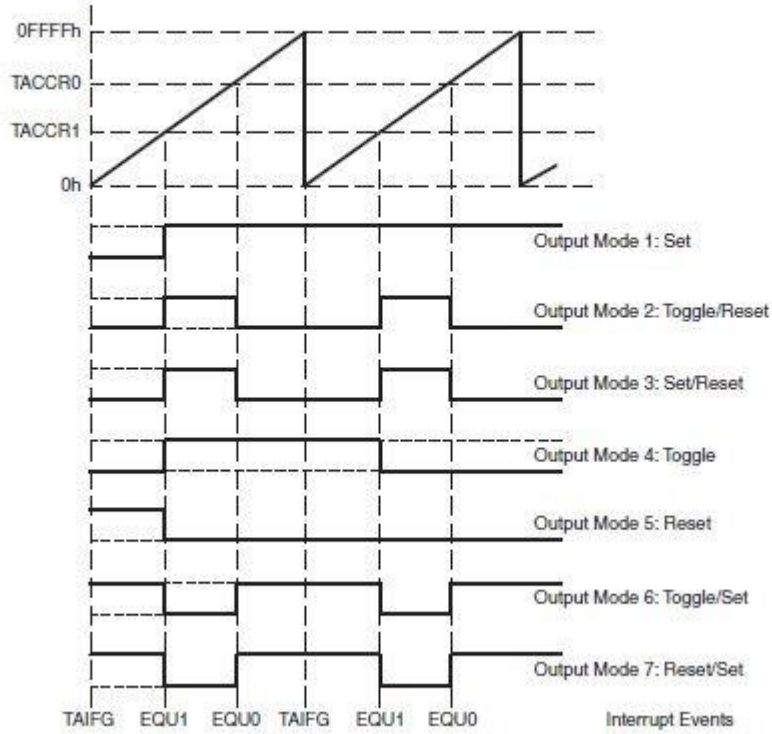


Figure 12-13. Output Example—Timer in Continuous Mode

Timer_A Interrupts

Two interrupt vectors are associated with the 16-bit Timer_A module:

TACCR0 interrupt vector for TACCR0 CCIFG

TAIV interrupt vector for all other CCIFG flags and TAIFG

In capture mode any CCIFG flag is set when a timer value is captured in the associated TACCRx register. In compare mode, any CCIFG flag is set if TAR counts to the associated TACCRx value. Software may also set or clear any CCIFG flag. All CCIFG flags request an interrupt when their corresponding CCIE bit and the GIE bit are set.

TACTL, Timer_A Control Register

15	14	13	12	11	10	9	8
Unused						TASSELx	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

7	6	5	4	3	2	1	0
IDx		MCx		Unused	TACLR	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	w-(0)	rw-(0)	rw-(0)

Unused	Bits 15-10	Unused
TASSELx	Bits 9-8	Timer_A clock source select 00 TACLK 01 ACLK 10 SMCLK 11 Inverted TACLK
IDx	Bits 7-6	Input divider. These bits along with the IDEXx bits select the divider for the input clock. 00 /1 01 /2 10 /4 11 /8
MCx	Bits 5-4	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power. 00 Stop mode: the timer is halted 01 Up mode: the timer counts up to TACCR0 10 Continuous mode: the timer counts up to 0FFFFh 11 Up/down mode: the timer counts up to TACCR0 then down to 0000h
Unused	Bit 3	Unused
TACLR	Bit 2	Timer_A clear. Setting this bit resets TAR, the TACLK divider, and the count direction. The TACLR bit is automatically reset and is always read as zero.
TAIE	Bit 1	Timer_A interrupt enable. This bit enables the TAIFG interrupt request. 0 Interrupt disabled 1 Interrupt enabled
TAIFG	Bit 0	Timer_A interrupt flag 0 No interrupt pending 1 Interrupt pending

TACCTLx, Capture/Compare Control Register

15	14	13	12	11	10	9	8
CMx		CCISx		SCS	SCCI	Unused	CAP
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r-(0)	r-(0)	rw-(0)

7	6	5	4	3	2	1	0
OUTMODx			CCIE	CCI	OUT	COV	CCIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	rw-(0)	rw-(0)	rw-(0)

CMx	Bit 15-14	Capture mode 00 No capture 01 Capture on rising edge 10 Capture on falling edge 11 Capture on both rising and falling edges
CCISx	Bit 13-12	Capture/compare input select. These bits select the TACCRx input signal. See the device-specific data sheet for specific signal connections. 00 CCIxA 01 CCIxB 10 GND 11 Vcc
SCS	Bit 11	Synchronize capture source. This bit is used to synchronize the capture input signal with the timer clock. 0 Asynchronous capture 1 Synchronous capture
SCCI	Bit 10	Synchronized capture/compare input. The selected CCI input signal is latched with the EQUx signal and can be read via this bit.
Unused	Bit 9	Unused. Read only. Always read as 0.
CAP	Bit 8	Capture mode 0 Compare mode 1 Capture mode
OUTMODx	Bits 7-5	Output mode. Modes 2, 3, 6, and 7 are not useful for TACCR0 because EQUx = EQU0. 000 OUT bit value 001 Set 010 Toggle/reset 011 Set/reset 100 Toggle 101 Reset 110 Toggle/set 111 Reset/set
CCIE	Bit 4	Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag. 0 Interrupt disabled 1 Interrupt enabled
CCI	Bit 3	Capture/compare input. The selected input signal can be read by this bit.
OUT	Bit 2	Output. For output mode 0, this bit directly controls the state of the output. 0 Output low 1 Output high
COV	Bit 1	Capture overflow. This bit indicates a capture overflow occurred. COV must be reset with software. 0 No capture overflow occurred 1 Capture overflow occurred

Timer B

Timer_B is a 16-bit timer/counter with three or seven capture/compare registers. Timer_B can support multiple capture/compares, PWM outputs, and interval timing. Timer_B also has extensive interrupt capabilities. Interrupts may be generated from the counter on overflow conditions and from each of the capture/compare registers.

Timer_B features include :

Asynchronous 16-bit timer/counter with four operating modes and four selectable lengths

Selectable and configurable clock source

Up to seven configurable capture/compare registers

Configurable outputs with PWM capability

Double-buffered compare latches with synchronized loading

Interrupt vector register for fast decoding of all Timer_B interrupts

Timer_B is identical to Timer_A with the following exceptions:

The length of Timer_B is programmable to be 8, 10, 12, or 16 bits.

Timer_B TBCCRx registers are double-buffered and can be grouped.

All Timer_B outputs can be put into a high-impedance state.

The SCCI bit function is not implemented in Timer_B.

Interrupts

Timer_B Interrupts

The Timer_B Capture/Comparison Register 0 Interrupt Flag (TBCCR0) generates a single interrupt vector:

TBCCR0 CCIFG → **TIMERB0_VECTOR**

No handler is required

TBCCR1-6 and TB interrupt flags are prioritized and combined using the Timer_B Interrupt Vector Register (TBIV) into another interrupt vector

TBCCR1 CCIFG

TBCCR2 CCIFG → **TBIV** → **TIMER_B1_VECTOR**

TBIFG

Your code must contain a handler to determine which Timer_B interrupt triggered

The same applies to Timer_A

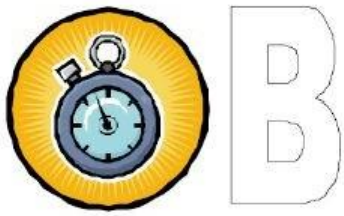
B vs A ...

57

Timer_B vs Timer_A

Timer_B vs Timer_A

- Default function identical to Timer_A
- 8, 10, 12, or 16-bit timer or counter (16-bit only for Timer_A)
- Outputs double-buffered for simultaneous loading
- CCRx registers can be grouped for simultaneous updates
- Tri-state function from external pin



Lab 4 Flowchart ...

58

Real Time Clock:

The Real-Time Clock module provides a clock with calendar that can also be configured as a general

purpose counter.

Real-Time Clock features include:

Configurable for Real-Time Clock mode or general purpose counter

Provides seconds, minutes, hours, day of week, day of month, month and year in calender mode.

Interrupt capability.

Selectable BCD or binary format in Real-Time Clock mode

Programmable alarms in Real-Time Clock mode

Calibration logic for time offset correction in Real-Time clock mode

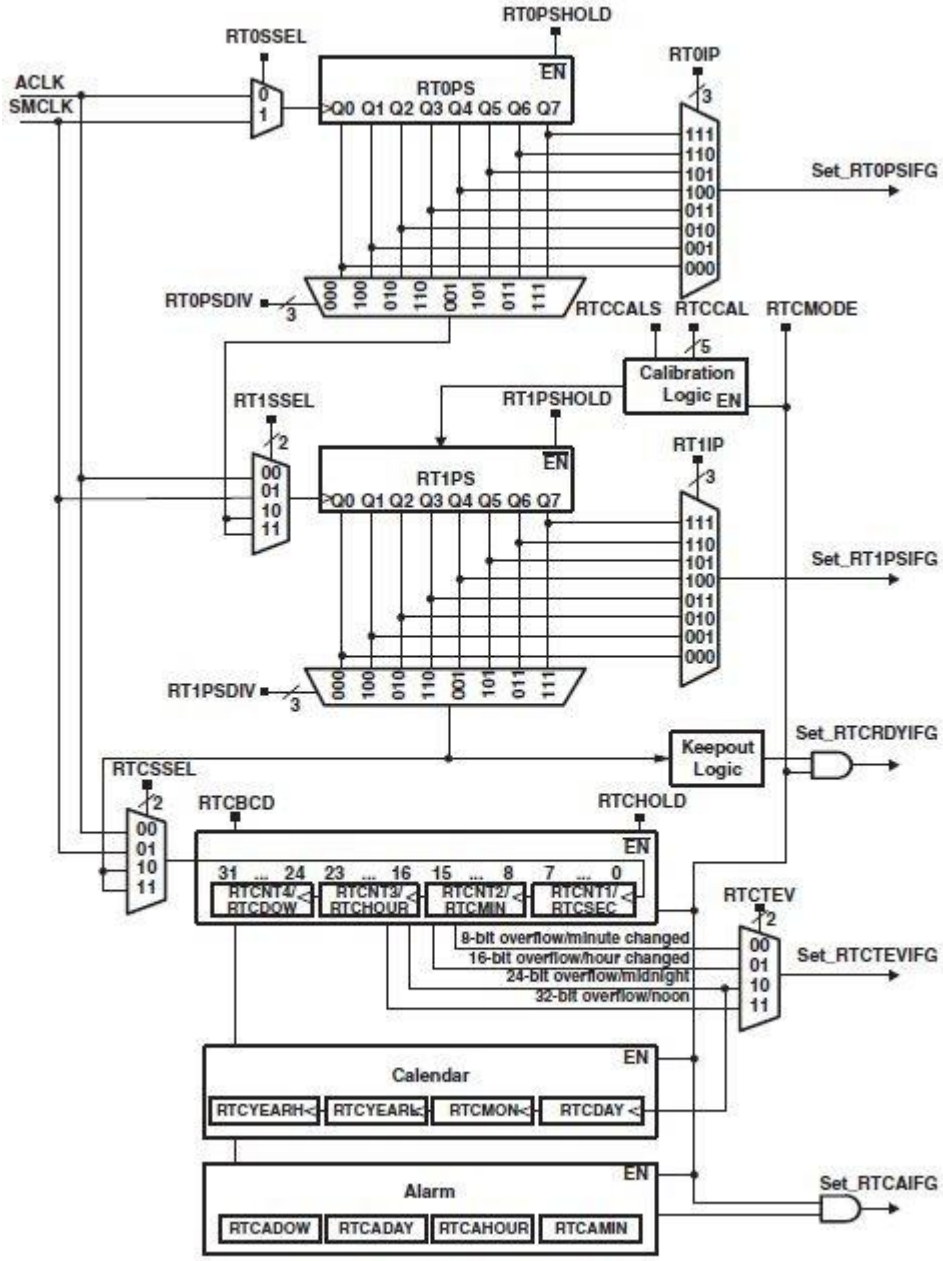


Figure 14-1. Real-Time Clock

The Real-Time Clock module can be configured as a real-time clock with calendar function or as a 32-bit general purpose counter with the RTCMODE bit.

Counter mode

Counter mode is selected when RTCMODE is reset. In this mode, a 32-bit counter is provided that isdirectly accessible by software. Switching from calendar mode to counter mode resets the count value (RTCNT1, RTCNT2, RTCNT3, RTCNT4), as well as, the prescale counters (RT0PS, RT1PS). The clock to increment the counter can be sourced from ACLK, SMCLK, or prescaled versions of ACLK or SMCLK. Prescaled versions of ACLK or SMCLK are sourced from the prescale dividers , RT0PS and RT1PS. RT0PS and RT1PS output /2, /4, /8, 16, /32,

/64, /128, /256 versions of ACLK and SMCLK, respectively. The output of RT0PS can be cascaded with RT1PS. The cascaded output can be used as a clock source input to the 32-bit counter.

Four individual 8-bit counters are cascaded to provide the 32-bit counter. This provides 8-bit, 16-bit, 24-bit, or 32-bit overflow intervals of the counter clock. The RTCTEV bits select the respective trigger event. An RTCTEV event can trigger an interrupt by setting the RTCTEVIE bit. Each counter RTCNT1 through RTCNT4 is individually accessible and may be written to. RT0PS and RT1PS can be configured as two 8-bit counters or cascaded into a single 16-bit counter. RT0PS and RT1PS can be halted on an individual basis by setting their respective RT0PSHOLD and RT1PSHOLD bits. When RT0PS is cascaded with RT1PS, setting RT0PSHOLD will cause both RT0PS and RT1PS to be halted. The 32-bit counter can be halted several ways depending on the configuration. If the 32-bit counter is sourced directly from ACLK or SMCLK, it can be halted by setting RTCHOLD. If it is sourced from the output of RT1PS, it can be halted by setting RT1PSHOLD or RTCHOLD. Finally, if it is sourced from the cascaded outputs of RT0PS and RT1PS, it can be halted by setting RT0PSHOLD, RT1PSHOLD, or RTCHOLD.

Calendar mode

Calendar mode is selected when RTCMODE is set. In calendar mode, the Real-Time Clock module provides seconds, minutes, hours, day of week, day of month, month, and year in selectable BCD or hexadecimal format. The calendar includes a leap year algorithm that considers all years evenly divisible by 4 as leap years. This algorithm is accurate from the year 1901 through 2099.

The prescale dividers, RT0PS and RT1PS are automatically configured to provide a one second clock interval for the Real-Time Clock. RT0PS is sourced from ACLK. ACLK must be set to 32768 Hz, nominal for proper Real-Time Clock calendar operation. RT1PS is cascaded with the output ACLK/256 of RT0PS. The Real-Time Clock is sourced with the /128 output of RT1PS, thereby providing the required one second interval. Switching from counter to calendar mode clears the seconds, minutes, hours, day-of-week, and year counts and sets day-of-month and month counts to 1. In addition, the RT0PS and RT1PS are cleared.

When RTCBCD = 1, BCD format is selected for the calendar registers. The format must be selected before the time is set. Changing the state of RTCBCD clears the seconds, minutes, hours, day-of-week, and year counts and sets day-of-month and month counts to 1. In addition, RT0PS and RT1PS are cleared.

In calendar mode, the RT0SSEL, RT1SSEL, RT0PSDIV, RT1PSDIV, RT0PSHOLD,

RT1PSHOLD, and RTCSSEL bits are do not care. Setting RTCHOLD halts the real-time counters and prescale counters, RT0PS and RT1PS.

Real-Time Clock and Prescale Dividers

The prescale dividers, RT0PS and RT1PS are automatically configured to provide a one second clock interval for the Real-Time Clock. RT0PS is sourced from ACLK. ACLK must be set to 32768 Hz, nominal for proper Real-Time Clock calendar operation. RT1PS is cascaded with the output ACLK/256 of RT0PS. The Real-Time Clock is sourced with the

/128 output of RT1PS, thereby providing the required one second interval. Switching from counter to calendar mode clears the seconds, minutes, hours, day-of-week, and year counts and sets day-of-month and month counts to 1. In addition, the RT0PS and RT1PS are cleared.

When RTCBCD = 1, BCD format is selected for the calendar registers. The format must be selected before the time is set. Changing the state of RTCBCD clears the seconds, minutes,

hours, day-of-week, and year counts and sets day-of-month and month counts to 1. In addition, RT0PS and RT1PS are cleared.

In calendar mode, the RT0SSEL, RT1SSEL, RT0PSDIV, RT1PSDIV, RT0PSHOLD,

RT1PSHOLD, and RTCSSEL bits are do not care. Setting RTCHOLD halts the real-time counters and prescale counters, RT0PS and RT1PS.

Real-Time Clock Alarm Function

The Real-Time Clock module provides for a flexible alarm system. There is a single, user programmable alarm that can be programmed based on the settings contained in the alarm registers for minutes, hours, day of week, and day of month. The user programmable alarm function is only available in calendar mode of operation.

Each alarm register contains an alarm enable bit, AE that can be used to enable the respective alarm register. By setting AE bits of the various alarm registers, a variety of alarm events can be generated. For example, a user wishes to set an alarm every hour at 15 minutes past the

hour i.e. 00:15:00, 01:15:00, 02:15:00, etc. This is possible by setting RTCAMIN to 15. By setting the AE bit of the RTCAMIN, and clearing all other AE bits of the alarm registers, the alarm will be enabled. When enabled, the AF will be set when the count transitions from 00:14:59 to 00:15:00, 01:14:59 to 01:15:00, 02:14:59 to 02:15:00,etc.

For example, a user wishes to set an alarm every day at 04:00:00. This is possible by setting RTCAHOUR to 4. By setting the AE bit of the RTCHOUR, and clearing all other AE bits of the alarm registers, the alarm will be enabled. When enabled, the AF will be set when the count transitions from 03:59:59 to 04:00:00.

For example, a user wishes to set an alarm for 06:30:00. RTCAHOUR would be set to 6 and RTCAMIN would be set to 30. By setting the AE bits of RTCAHOUR and RTCAMIN, the alarm will be enabled. Once enabled, the AF will be set when the the time count transitions from 06:29:59 to 06:30:00. In this case, the alarm event will occur every day at 06:30:00.

For example, a user wishes to set an alarm every Tuesday at 06:30:00. RTCADOW would be set to 2, RTCAHOUR would be set to 6 and RTCAMIN would be set to 30. By setting the AE bits of RTCADOW, RTCAHOUR and RTCAMIN, the alarm will be enabled. Once enabled, the AF will be set when the the time count transitions from 06:29:59 to 06:30:00 and the RTCDOW transitions from 1 to 2.

For example, a user wishes to set an alarm the fifth day of each month at 06:30:00. RTCADAY would be set to 5, RTCAHOUR would be set to 6 and RTCAMIN would be set to

30. By setting the AE bits of RTCADAY, RTCAHOUR and RTCAMIN, the alarm will be enabled. Once enabled, the AF will be set when the the time count transitions from 06:29:59 to 06:30:00 and the RTCDAY equals 5.

Reading or Writing Real-Time Clock Registers in Calendar Mode

must be used when accessing the Real-Time Clock registers.

In calendar mode, the real-time clock registers are updated once per second. In order to prevent reading any real-time clock register at the time of an update that could result in an invalid time being read, a keepout window is provided. The keepout window is centered approximately - 128/32768 seconds around the update transition. The read only RTCRDY bit is reset during the keepout window period and set outside the keepout the window period.

Any read of the clock registers while RTCRDY is reset, is considered to be potentially invalid, and the time read should be ignored.An easy way to safely read the real-time clock registers is

to utilize the RTCRDYIFG interrupt flag. Setting RTCRDYIE enables the RTCRDYIFG interrupt. Once enabled, an interrupt will be generated based on the rising edge of the RTCRDY bit, causing the RTCRDYIFG to be set. At this point, the application has nearly a complete second to safely read any or all of the real-time clock registers. This synchronization process prevents reading the time value during transition. The RTCRDYIFG flag is reset automatically

when the interrupt is serviced, or can be reset with software. In counter mode, the RTCRDY bit remains reset. The RTCRDYIE is a do not care and the RTCRDYIFG remains reset.

Real-Time Clock Interrupts

Real-Time Clock Interrupts in Calendar Mode

In calendar mode, five sources for interrupts are available, namely RT0PSIFG, RT1PSIFG, RTCRDYIFG, RTCTEVIFG, and RTCAIFG. These flags are prioritized and combined to source a single interrupt vector. The interrupt vector register RTCIV is used to determine which flag requested an interrupt. The highest priority enabled interrupt generates a number in the RTCIV register (see register description).

This number can be evaluated or added to the program counter to automatically enter the appropriate software routine. Disabled RTC interrupts do not affect the RTCIV value. Any access, read or write, of the RTCIV register automatically resets the highest pending interrupt flag. If another interrupt flag is set, another interrupt is immediately generated after servicing the initial interrupt. In addition, all flags can be cleared via software.

The user programmable alarm event sources the real-time clock interrupt, RTCAIFG. Setting the RTCAIE enables the interrupt. In addition to the user programmable alarm, The Real-Time Clock Module provides for an interval alarm that sources real-time clock interrupt, RTCTEVIFG. The interval alarm can be selected to cause an alarm event when RTCMIN changed, RTCHOUR changed, every day at midnight (00:00:00), or every day at noon (12:00:00). The event is selectable with the RTCTEV bits Setting the RTCTEVIE bit enables the interrupt.

The RTCRDY bit sources the real-time clock interrupt, RTCRDYIFG and is useful in synchronizing the read of time registers with the system clock. Setting the RTCRDYIE bit enables the interrupt. The RT0PSIFG can be used to generate interrupt intervals selectable by

the RT0IP bits. In calendar mode, RT0PS is sourced with ACLK at 32768 Hz, so intervals of 16384 Hz, 8192 Hz, 4096 Hz, 2048 Hz, 1024 Hz, 512 Hz, 256 Hz, or 128 Hz are possible. Setting the RT0PSIE bit enables the interrupt. The RT1PSIFG can be used to generate interrupt intervals selectable by the RT1IP bits. In calendar mode, RT1PS is sourced with the output of RT0PS, which is 128Hz (32768/256 Hz). Therefore, intervals of 64 Hz, 32 Hz, 16 Hz, 8 Hz, 4 Hz, 2 Hz, 1 Hz, or 0.5 Hz are possible. Setting the RT1PSIE bit enables the interrupt.

Real-Time Clock Interrupts in Counter Mode

In counter mode, a three interrupt sources are available, namely RT0PSIFG, RT1PSIFG, and RTCTEVIFG. The RTCAIFG and RTCRDYIFG are cleared. RTCRDYIE and RTCAIE are do not care.The RT0PSIFG can be used to generate interrupt intervals selectable by the RT0IP bits. In counter mode, RT0PS is sourced with ACLK or SMCLK so divide ratios of /2, /4, /8, /16, /32, /64, /128, /256 of the respective clock source are possible. Setting the RT0PSIE bit enables the interrupt.

The RT1PSIFG can be used to generate interrupt intervals selectable by the RT1IP bits. In counter mode, RT1PS is sourced with ACLK, SMCLK, or the output of RT0PS so divide ratios of /2, /4, /8, /16, /32, /64, /128, /256 of the respective clock source are possible. Setting the RT1PSIE bit enables the interrupt. The Real-Time Clock Module provides for an interval timer that sources real-time clock interrupt, RTCTEVIFG. The interval timer can be selected to cause an interrupt event when an 8-bit, 16-bit, 24-bit, or 32-bit overflow occurs within the 32-bit counter. The event is selectable with the RTCTEV bits Setting the RTCTEVIE bit enables the interrupt.

RTCCTL0, Real-Time Clock Control Register 0

7	6	5	4	3	2	1	0
Reserved	RTCTEVIE	RTCAIE	RTCRDYIE	Reserved	RTCTEVIFG	RTCAIFG	RTCRDYIFG
r0	rw-0	rw-0	rw-0	r0	rw-(0)	rw-(0)	rw-(0)

Reserved	Bit 7	Reserved. Always read as 0.
RTCTEVIE	Bit 6	Real-time clock time event interrupt enable
		0 Interrupt not enabled
		1 Interrupt enabled
RTCAIE	Bit 5	Real-time clock alarm interrupt enable. This bit remains cleared when in counter mode (RTCMODE = 0).
		0 Interrupt not enabled
		1 Interrupt enabled
RTCRDYIE	Bit 4	Real-time clock alarm interrupt enable
		0 Interrupt not enabled
		1 Interrupt enabled
Reserved	Bit 3	Reserved. Always read as 0.
RTCTEVIFG	Bit 2	Real-time clock time event flag
		0 No time event occurred.
		1 Time event occurred.
RTCAIFG	Bit 1	Real-time clock alarm flag. This bit remains cleared when in counter mode (RTCMODE = 0).
		0 No time event occurred.
		1 Time event occurred.
RTCRDYIFG	Bit 0	Real-time clock alarm flag
		0 RTC can not be read safely
		1 RTC can be read safely

Table 14-1. Real-Time Clock Registers

Register	Short Form	Register Type	Address Offset	Initial State
Real-Time Clock control register 0	RTCCTL0	Read/write	00h	00h
Real-Time Clock control register 1	RTCCTL1	Read/write	01h	40h
Real-Time Clock control register 2	RTCCTL2	Read/write	02h	00h
Real-Time Clock control register 3	RTCCTL3	Read/write	03h	00h
Real-Time Prescale Timer 0 control register	RTCPS0CTL	Read/write	08h	10h
Real-Time Prescale Timer 1 control register	RTCPS1CTL	Read/write	0Ah	10h
Real-Time Prescale Timer 0	RTCPS0	Read/write	0Ch	Unchanged
Real-Time Prescale Timer 1	RTCPS1	Read/write	0Dh	Unchanged
Real Time Clock Interrupt vector	RTCIV	Read	0Eh	00h
Real-Time Clock Second Real-Time Counter register 1	RTCSEC/RTCNT1	Read/write	10h	Unchanged
Real-Time Clock Minute Real-Time Counter register 2	RTCMIN/RTCNT2	Read/write	11h	Unchanged
Real-Time Clock Hour Real-Time Counter register 3	RTCHOUR/RTCNT3	Read/write	12h	Unchanged
Real-Time Clock Day of Week Real-Time Counter register 4	RTCDOW/RTCNT4	Read/write	13h	Unchanged
Real-Time Clock Day of Month	RTCDAY	Read/write	14h	Unchanged
Real-Time Clock Month	RTCMON	Read/write	15h	Unchanged
Real-Time Clock Year (Low Byte)	RTCYEARL	Read/write	16h	Unchanged
Real-Time Clock Year (High Byte)	RTCYEARH	Read/write	17h	Unchanged
Real-Time Clock Minute Alarm	RTCAMIN	Read/write	18h	Unchanged
Real-Time Clock Hour Alarm	RTCAHOUR	Read/write	19h	Unchanged
Real-Time Clock Day of Week Alarm	RTCADOW	Read/write	1Ah	Unchanged
Real-Time Clock Day of Month Alarm	RTCADAY	Read/write	1Bh	Unchanged

RTCCTL1, Real-Time Clock Control Register 1

7	6	5	4	3	2	1	0
RTCBCD	RTCHOLD	RTCMODE	RTCRDY	RTCSSEL		RTCTEV	
rw-(0)	rw-(1)	rw-(0)	r-(0)	rw-0	rw-0	rw-(0)	rw-(0)

RTCBCD Bit 7

Real-time clock BCD select. Selects BCD counting for real-time clock. Applies to calendar mode (RTCMODE = 1) only - setting will be ignored in counter mode. Changing this bit will clear seconds, minutes, hours, day of week, and year are to 0 and sets day of month and month to 1. The real-time clock registers need to be set by software afterwards.

0 Binary/hexadecimal code selected

1 BCD (Binary Coded Decimal) code selected

RTCHOLD Bit 6

Real-time clock hold

0 Real-Time Clock (32-bit counter or calendar mode) is operational

1 In counter mode (RTCMODE = 0) only the 32-bit counter is stopped. In calendar mode (RTCMODE = 1) the calendar is stopped as well as the Prescale counters, RT0PS and RT1PS. RT0PSHOLD and RT1PSHOLD are do not care.

RTCMODE Bit 5

Real-time clock mode

0 32-bit counter mode

1 Calendar modeSwitching between counter and calendar mode will reset the real-time clock/counter registers. Switching to calendar mode clears seconds, minutes, hours, day of week, and year are to 0 and sets day of month and month to 1. The real-time clock registers need to be set by software afterwards. The Basic Timer counters, BT0CNT and BT1CNT, are also cleared.

RTCRDY Bit 4

Real-time clock ready

0 RTC time values in transition (calendar mode only).

1 RTC time values safe for reading (calendar mode only)This bit indicates when the RTC time values are safe for reading (calendar mode only). In counter mode, RTCRDY signal remains cleared.

RTCSSEL Bits 3-2

Real-time clock source select. Selects clock input source to the RTC/32-bit counter. In Real-Time Clock calendar mode, these bits are do not care. The clock input is automatically set to the output of RT1PS.

00 ACLK

01 SMCLK

10 Output from RT1PS

11 Output from RT1PS

RTCTEV Bits 1-0

Real-time clock time event

RTC Mode	RTCTEVx	Interrupt Interval
Counter Mode (RTCMODE = 0)	00	8-bit overflow
	01	16-bit overflow
	10	24-bit overflow
	11	32-bit overflow
Calendar Mode (RTCMODE = 1)	00	Minute changed
	01	Hour changed
	10	Every day at midnight (00:00)
	11	Every day at noon (12:00)

DMA Controller

The direct memory access (DMA) controller module transfers data from one address to another, without CPU intervention. This chapter describes the operation of the DMA controller.

Direct Memory Access (DMA) Introduction

The DMA controller transfers data from one address to another, without CPU intervention, across the entire address range. For example, the DMA controller can move data from the ADC conversion memory to RAM. Devices that contain a DMA controller may have up to eight DMA channels available. Therefore, depending on the number of DMA channels available, some features described in this chapter are not applicable to all devices. See the device-specific data sheet for number of channels supported. Using the DMA controller can increase the throughput of peripheral modules. It can also reduce system power consumption by allowing the CPU to remain in a low-power mode, without having to awaken to move data to or from a peripheral.

DMA controller features include:

- Up to eight independent transfer channels
- Configurable DMA channel priorities
- Requires only two MCLK clock cycles per transfer
- Byte or word and mixed byte/word transfer capability
- Block sizes up to 65535 bytes or words
- Configurable transfer trigger selections
- Selectable-edge or level-triggered transfer
- Four addressing modes
- Single, block, or burst-block transfer modes

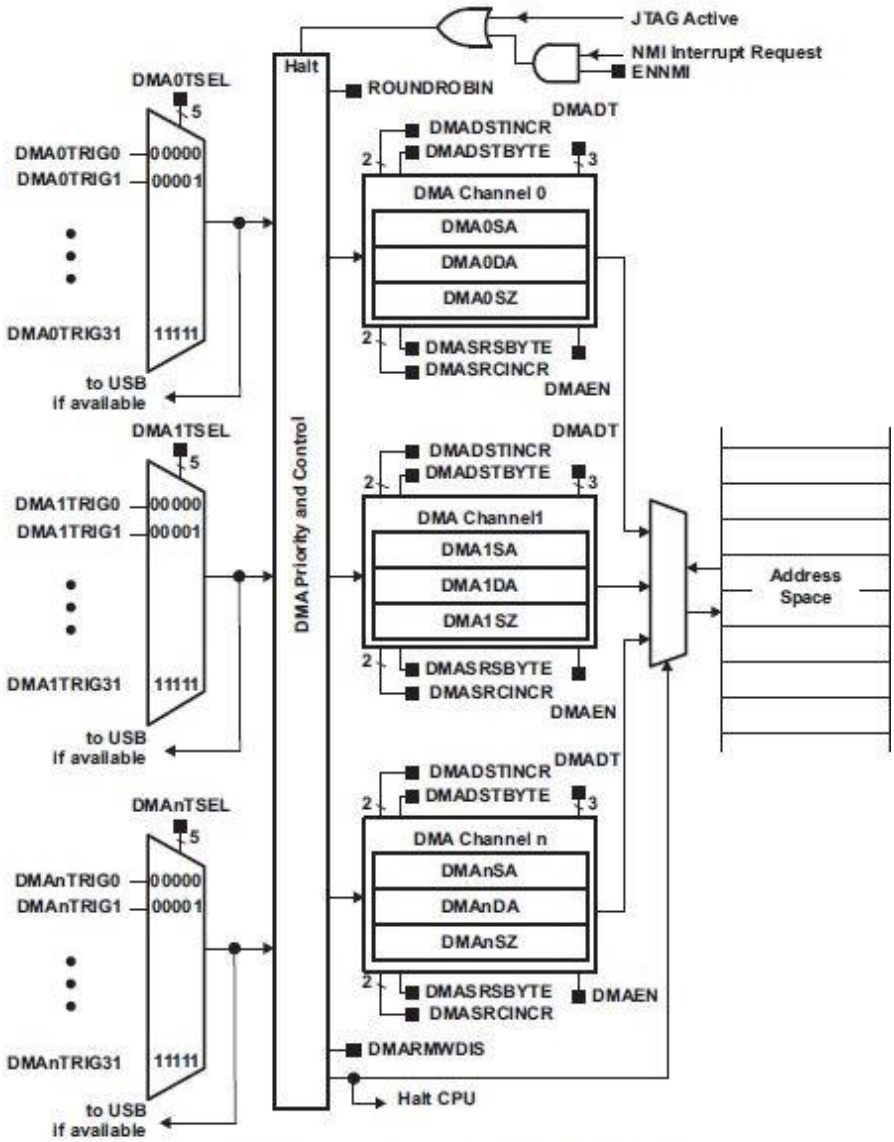
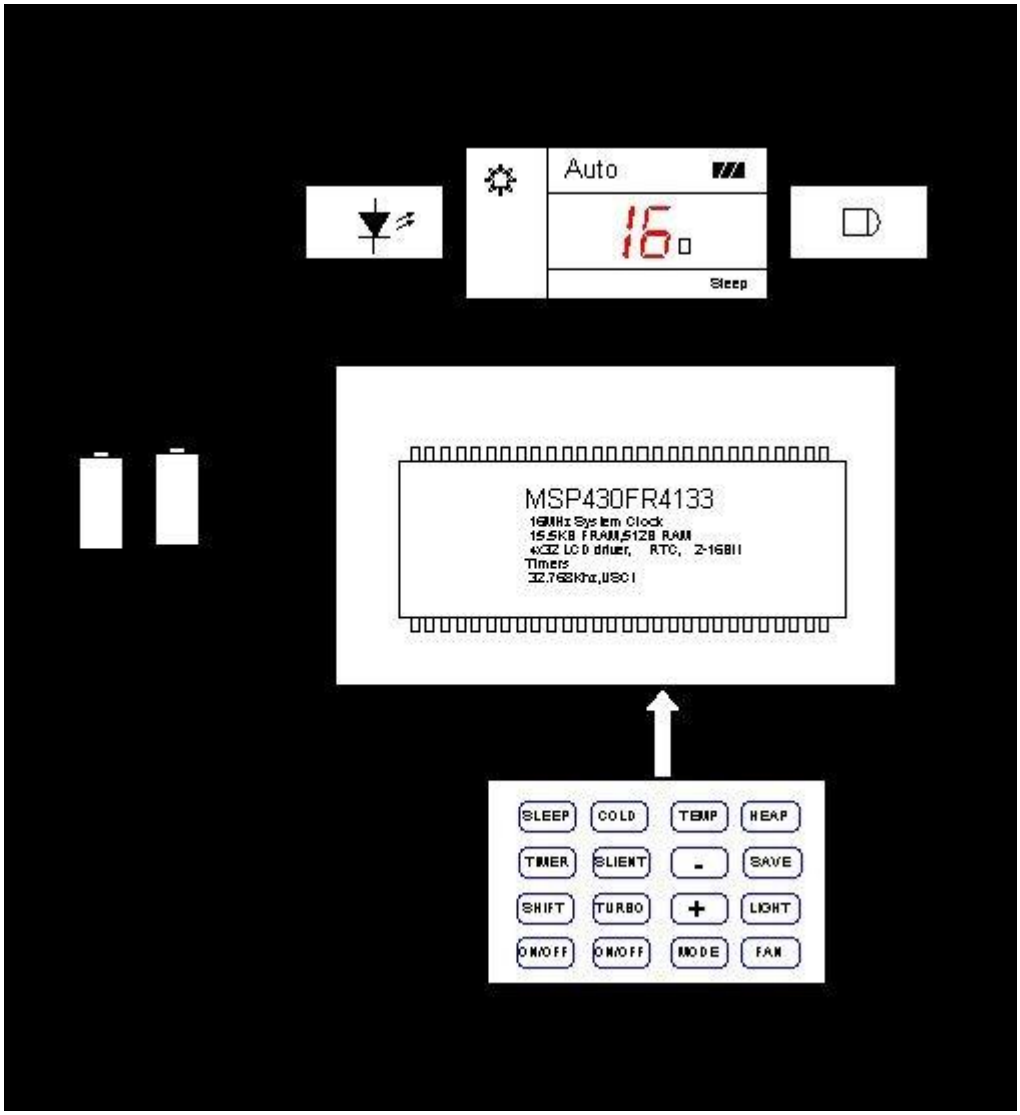


Figure 9-1. DMA Controller Block Diagram

CASE STUDY:

Remote Controller of Air Conditioner Using MSP430



System Description

This board demonstrates an ultra-low power, general purpose, infrared remote controller solution. The board uses a FRAM-based MCU MSP430FR4133, which supports features such as real time clock, button scan, infrared encoding, LED backlight, and LCD display.

MSP430FR4133

The MSP430FR4133 is a FRAM-based ultra-low power mixed signal MCU. With the following features, the MSP430FR4133 is highly suitable for portable device applications.

16-bit RISC architecture up to 16 Mhz

Wide supply voltage range from 1.8 V to 3.6 V

64-Pin/56-Pin/48Pin TSSOP/LQFP package options

Integrated LCD driver with charge pump can support up to 4x36 or 8x32 segment LCD

Optimized 16-bit timer for infrared signal generation

Low power mode (LPM3.5) with RTC on:0.77 uA

Low power mode (LPM3.5) with LCD on: 0.936 uA

Active mode: 126 uA/MHz

10^15 write cycle endurance low power ferroelectric RAM (FRAM) can be used to store data

10-channel, 10-bit analog-to-digital converter (ADC) with built-in 1.5 V reference for battery powered

system

All I/Os are capacitive touch I/O

CIRCUIT:

A 4x28 segment LCD is directly connected to the MSP430FR4133 LCD driver pins. Designers can swap the COM and SEG pins to simplify the PCB layout. A 4x4 matrix is used to detect 15 buttons. The matrix columns are connected to interrupt-enabled GPIOs (P1) to wake up the MSP430FR4133 from low power mode. MCU internal pull up/pull down resistors are used as button scan matrix pull up resistors. No external resistor is needed for button detection, and no

external circuit is needed for battery voltage detection. The function is also realized by the MCU ADC module without any external component.

A 32.768 KHz watch crystal serves as the MCU FLL and RTC clock source. Two chip capacitors, C4 and C6, are used as the crystal loading capacitor. Designers must choose C4

and C6 values carefully according to crystal specification. Cautious PCB layout design for the crystal is strongly recommended, to secure system clock robustness.

Software Description

The software implements an interrupt-driven structure. In the main loop, the MCU stays in LPM3.5 mode.Interrupts from the button, RTC, and timer wake up the MCU for task processing. Inputs from the button are processed in task KeyProcess (), which handles system status and generates the content for the LCD display and infrared signal. RTC generates a 3S interval interrupt to inform the system of battery voltage measurement.

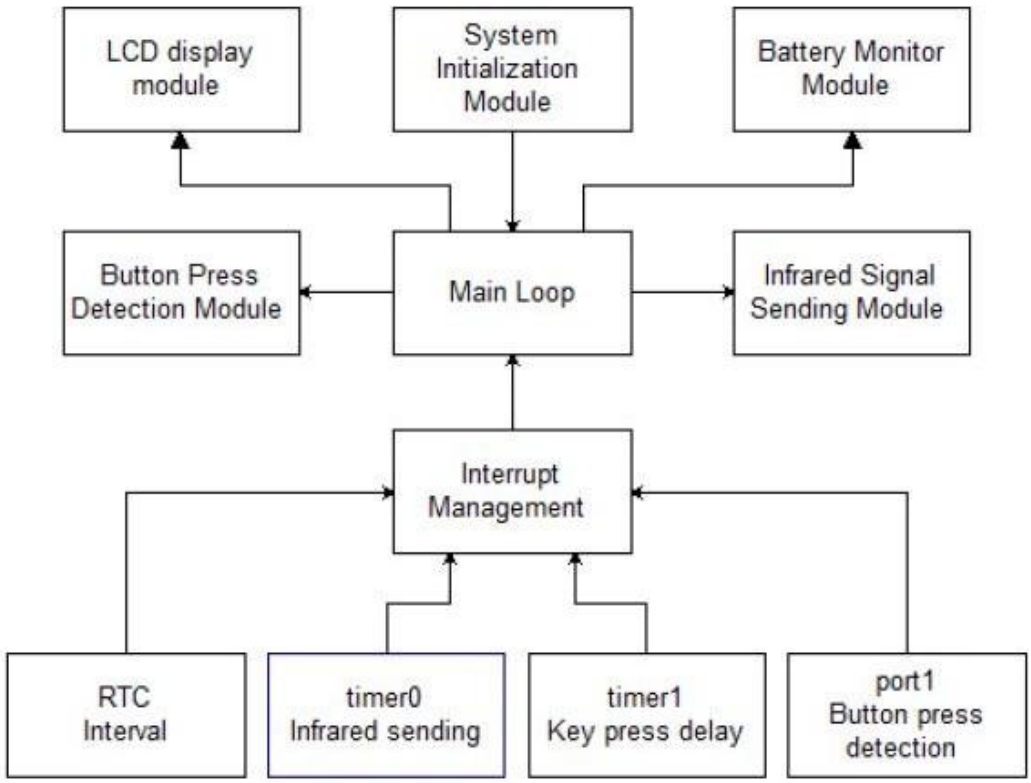


Figure 3. Software Structure

Infrared Signal Generation

There are several kinds of infrared modulation protocols in the industry. This design illustrates pulse distance protocol with data frame format, the most commonly-used format for air conditioner remote controllers. As shown in Figure 4, each bit is composed of a carrier-modulated pulse and a space. The space's width distinguishes logic 1 and logic 0 respectively. The carrier-modulated pulse width is constant. In this design, space length for 1 is 1690 uS, and 560 uS for digit 0. Modulated pulse width is 560 uS.

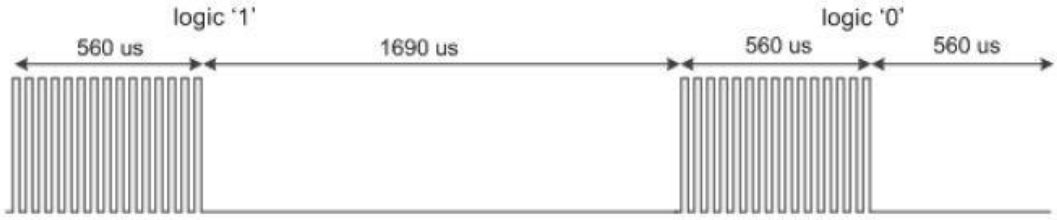


Figure 4. Pulse Distance Protocol, Bit Encoding

A complete data frame format is shown in Figure 5.

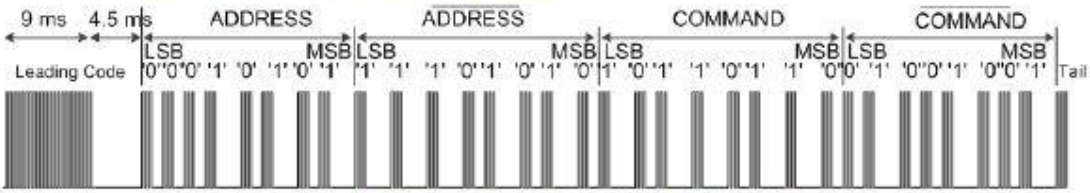


Figure 5. Pulse Distance Protocol, Data Frame Format

The envelope waveform depends on the frame format. Quantize all of the above items with a minimum time slot of 0.56 ms, as shown in Table 1.

Table 1. Table 1. Pulse Distance Protocol, Data Encoding Quantization Table

Items	Leading Code		Logic '1'		Logic '0'		Tail Code	
	Carrier Modulated Pulse	Space	Carrier Modulated Pulse	Space	Carrier Modulated Pulse	Space	Carrier Modulated Pulse	Space
Length	9 ms	4.5 ms	0.56 ms	1.69 ms	0.56 ms	0.56 ms	0.56 ms	0 ms
Quantization	16	8	1	3	1	1	1	0

TA1 is used to generate an envelope waveform, and each pair of carrier-modulated pulse and space must update the CCR0 and CCR2 once. The CCR0 depends on the carrier-modulated pulse period plus the space period, while the CCR2 depends on the carrier-modulated pulse period. For instance, if TA1 sources from SMCLK of 4 MHz and uses default divider configuration, CCR0 and CCR2 are individually configured as 54,000 and 36,000, to generate the leading code (9 ms carrier modulated pulse paired with 4.5 ms space), and updated to 9,000 and 2,240 for logic 1 (see Figure 6). To send one full data frame, CCR0 and CCR2 must be updated 34 (1+8*2+8*2+1) times, which is achieved in the TA1 interrupt routine.

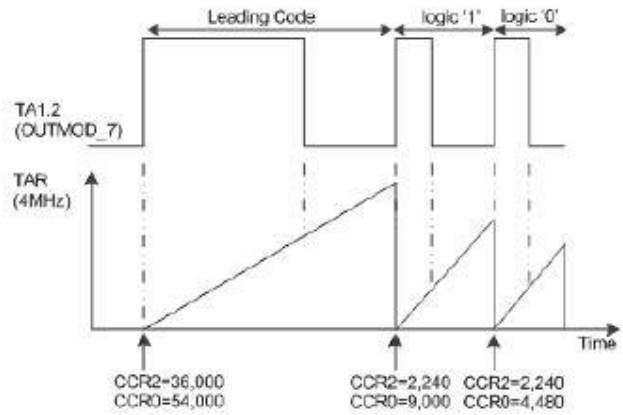


Figure 6. Pulse Distance Protocol, TA1 in Envelope Generation

UNIT IV

RTOS BASED EMBEDDED SYSTEM DESIGN

Introduction

Most embedded systems are bound to real-time constraints. In production control the various machines have to receive their orders at the right time to ensure smooth operation of a plant and to fulfill customer orders in time. Railway switching systems obviously have to act in a timely manner.

An embedded system with a single CPU can run only one process at an instance the process at any instance either be an ISR

Principles: semaphore and queues:

The principles will discuss the design considerations that have application to a broad range of embedded system.

Write Short interrupt routines

Limit the number of tasks

Avoid creating and destroying tasks

Avoid time-slicing

Encapsulate semaphores in separate functions

Encapsulate queues in separate functions

Encapsulating Queues:

consider encapsulating queues that tasks use to receive messages from other tasks. we wrote code to handle a shared flash memory. That code deals correctly with synchronizing the requests for reading from and writing to the flash memory. Since any task can write onto the flash memory task input queue, any programmer can blow it and send a message that does not contain a FLASH_MSG structure.

Hard Real-Time Scheduling Considerations:\

Hard Real Time Systems: Systems were time constraints are absolutely critical

Soft Real Time Systems: Systems with time constraints where minor errors are tolerated

The obvious issue that arises in hard real-time systems is that you must somehow guarantee that the system will meet the hard deadlines. The ability to meet hard deadlines comes from writing fast code To write some frequently called subroutine in assembly language.

If you can characterize your tasks, then the studies can help you determine if your system will meet its deadlines,

Saving Memory and Power:

Saving memory:

Embedded systems often have limited memory

RTOS: each task needs memory space for its stack.

The first method for determining how much stack space a task needs is to examine your code

The second method is experimental. Fill each stack withsome recognizable data pattern at startup, run the system for a period of time

Program Memory:

Limit the number of functions used

Check the automatic inclusions by your linker: may consider writing own functions

Include only needed functions in RTOS Consider using assembly language for large routines

Data Memory:

Consider using more static variables instead of stack variables

On 8-bit processors, use char instead of int when possible

Few ways to save code space:

Make sure that you are not using two functions to do the same thing.

Check that your development tools are not sabotaging you.

Configure your RTOS to contain only those functions that you need.

Look at the assembly language listings created by your cross-compiler to see if certain of your C statements translate into huge numbers of instructions.

Saving power:

The primary method for preserving battery power is to turn off parts or all of the system whenever possible.

Most embedded-system microprocessors have at least one power-saving mode; many have several.

The modes have names such as sleep mode, low-power mode, idle mode, standby mode, and so on.

A very common power-saving mode is one in which the microprocessor stops executing instructions, stops any built-in peripherals, and stops its clock circuit. This saves a lot of power, but the drawback typically is that the only way to start the microprocessor up again is to reset it.

Static RAM uses very little power when the microprocessor isn't executing instructions

Another typical power-saving mode is one in which the microprocessor stops executing instructions but the on-board peripherals continue to operate.

Another common method for saving power is to turn off the entire system and have the user turn it back on when it is needed.

Embedded software development tools:

Host and Target machines:

During development process, a host system is used

Then locating and burning the codes in the target board.

Target board hardware and software later copied to get the final embedded system

Final system function exactly as the one tested and debugged and finalized during the development process

Host system: at PC or workstation or laptop :

High performance processor with caches, large RAM memory

ROMBIOS (read only memory basic input-output system)

very large memory on disk

keyboard

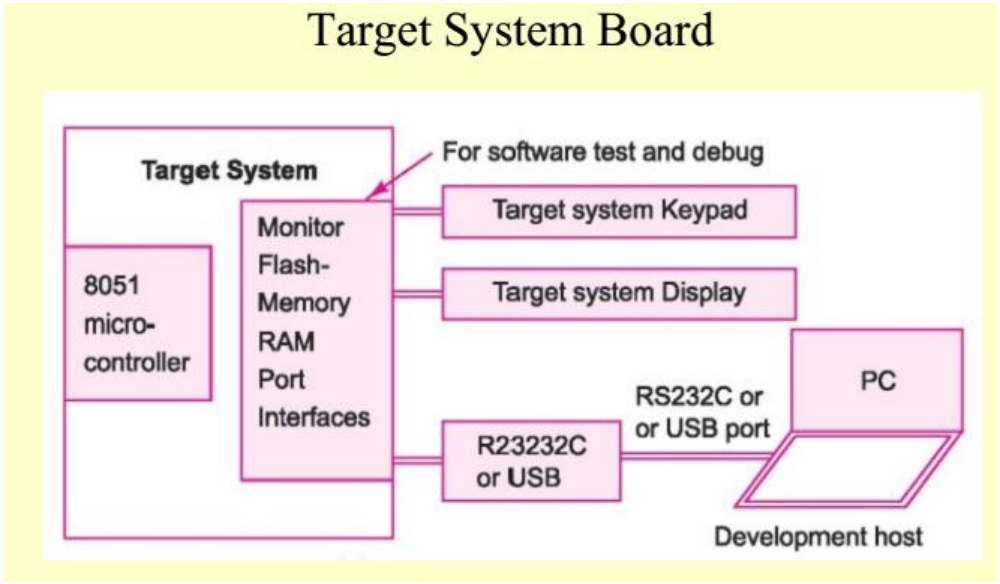
display monitor

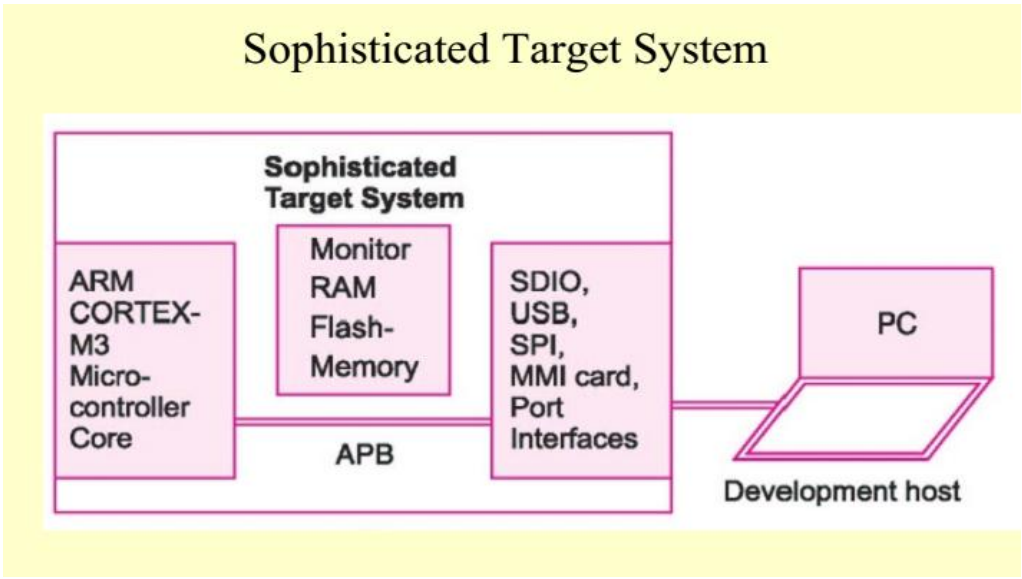
mice

network connection

Target machines:

A target system has a processor, ROM memory, for ROM image of the embedded software ,RAM for stack, temporary variable an memory before ,peripherals and interfaces





Some target systems have 8 or 16 MB flash memory and 64 MB SDRAM

Linker/Locators for Embedded Software:

Linker:

Determines addresses of labels that assembler could not resolve.

Typically extern functions & variables defined in other files.

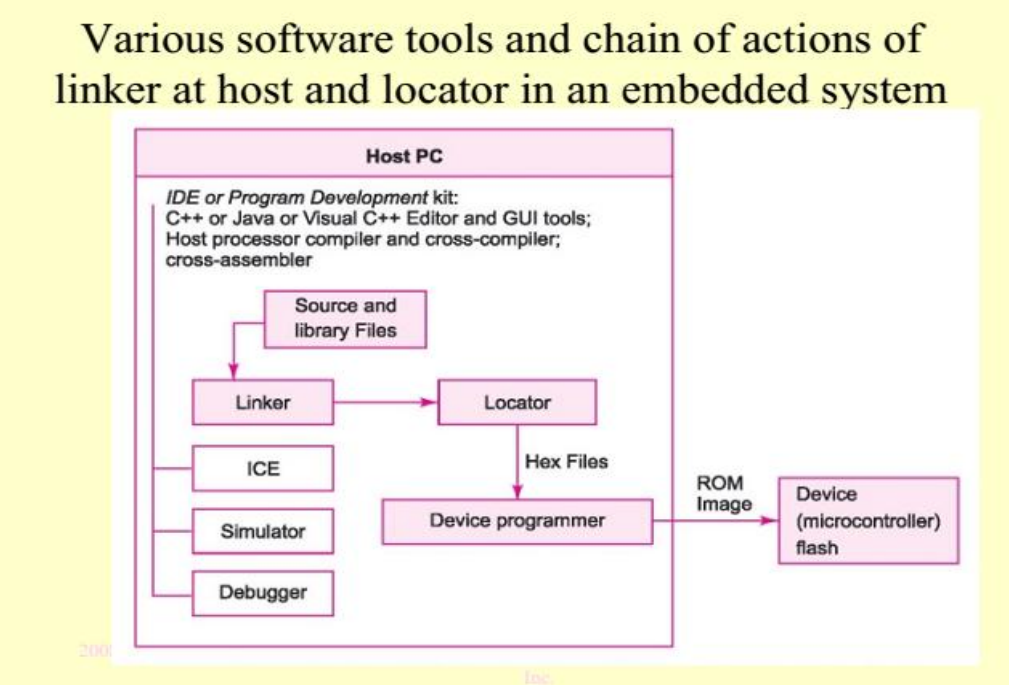
Assembler will have marked as needing to be fixed:

Instructions referencing these labels

Data (pointers) initialized to address of other variables

The linker puts together files, fixing up the address references between the files.

If unresolved labels exist at this point, it is a fatal error.



Locator:

Quite different functionality than native linker

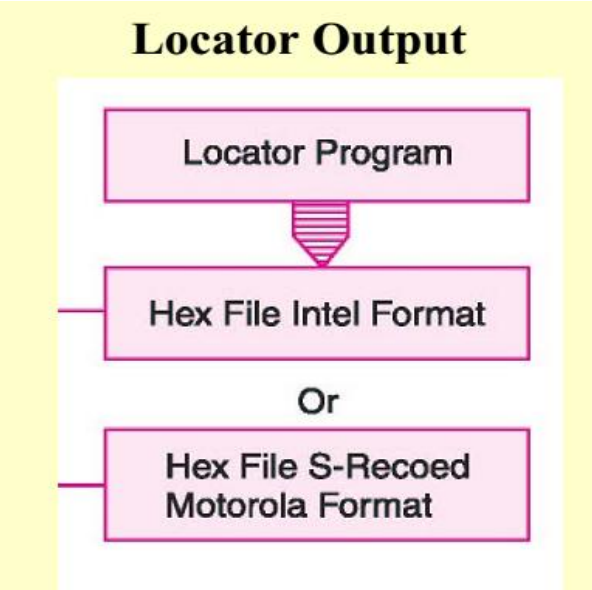
Determines final memory image of program

No loader will come along after and fix up addresses

Locator can do this because:

No other program will be in memory at runtime; no resource conflicts

Locator can determine final address of everything, including kernel and library functions called in application code



Locator includes a mechanism for programmer to determine placement in memory

Some parts need to be in RAM, others in ROM

Executing out of RAM:

RAM is often faster than Flash and ROM.

To exploit, startup code must copy program from ROM to RAM, then transfer control to it.

Consider new challenge for locator:

Build a program that is stored at one address (in ROM), but will run correctly at a different address (in RAM).

A bit tricky: requires support from the RTOS development system.

Getting Embedded Software into the Target System:

Several alternatives:

Write it to flash memory on target

Put it in ROM or PROM, then insert chip into system

Use a ROM emulator

Use an in-circuit emulator

Replaces microprocessor in target system

Overlay memory in emulator can be used instead of memory on target

Useful for debugging – not in shipped products

PROM Programmer

Used to program executable code into a PROM.

The PROM should be socketed so it can be replaced easily.

PROM approach good for production mode, but inconvenient for test and debug during development.

Painful to pull, reinsert chip for every new test.

Not surprisingly, other alternatives have been developed.

ROM emulator

Plugs into PROM/ROM socket.

Looks like ROM to target.

Has connection to host to allow easy changes to memory.

Much easier than burning a new PROM.

Used during development and debugging only.

Not shipped with working system!

Flash memory

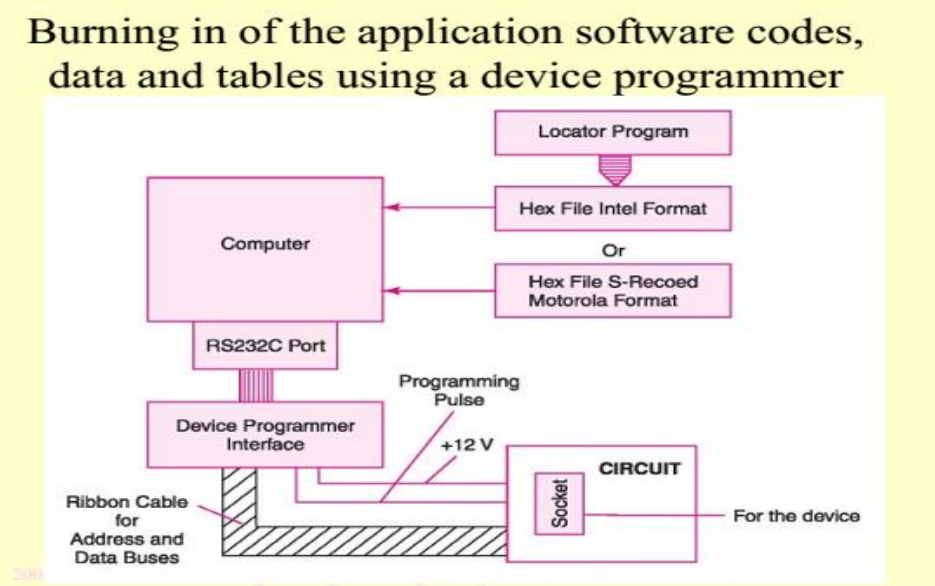
Flash is “field programmable”

Host can connect to target, cause flash to be reprogrammed directly without replacing any ships.

Software (bootstrap program) must be on target system to copy data from host to flash memory.

Tricky: cannot execute from flash while reprogramming it.

Must copy itself from ROM to RAM and then run from RAM.



Debugging Techniques:

Avoiding software bugs

The best approach is to produce bug-free code, but no software is completely error-free.

Although it can’t be your principal technique for ensuring software quality, you are foolish to not

do a lot of testing.

Unfortunately, embedded systems pose special challenges for testing.

Problems testing on target system

Target system may not be available or stable early on while code is being written and debugged.

Difficult to generate pathological timing scenarios.

Impossible to test all combinations, and difficult to know which combinations will cause a problem.

Bugs are often not repeatable.

Often show up with specific event sequence and timing.

Tough to generate using standard software test suites.

Embedded systems generally lack extensive logging capabilities to identify cause of failure.

Testing on Host Machine:

Test early (target may not ready or completely stable)

Exercise all code, including exceptions (real situations may be difficult to exercise)

Develop reusable, repeatable test (difficult to do in target environment, and likelihood of hitting the same bug is low)

Store test results (target may not even have disk drive to store results)

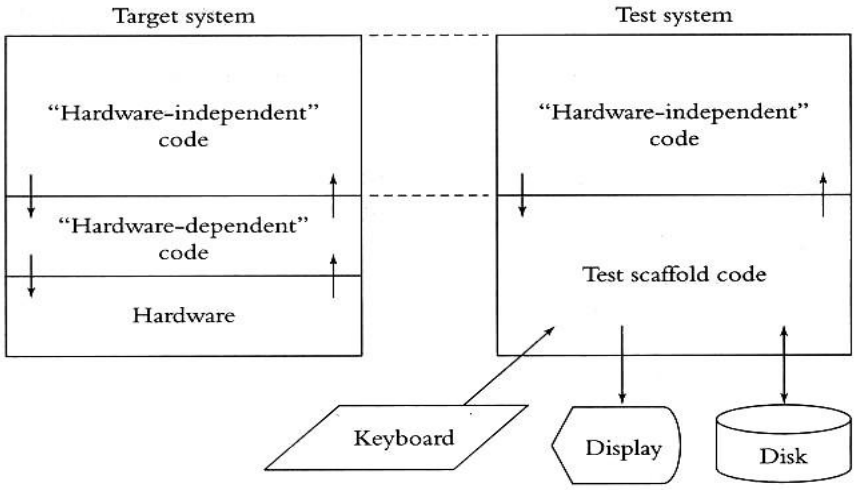
Basic Techniques

Target system on the left: (hardware-indep code, hardware-dep code, hw)

Test system (on host) on the right: (hardware-indep code – same, scaffold – rest)

Scaffold provides (in software) all functionalities and calls to hardware as in the hardware-dep and hardware components of the target system – more like a simulator for them

Figure 10.1 Test System



Laboratory Tools:

Hardware Diagnostic Laboratory Tools

Voltmeters, ohmmeters multi-meters	These are used for checking is the hardware working
Oscilloscopes	Graphs voltage vs. time, potentially multiple signals Can select trigger to start its operation
Logic analyzers	Capture signals, store in memory, graph on screen. Can track many signals simultaneously, Up to several hundred if you are willing to pay and make all the connections! Typical operation: trigger on symptom of problem, then look backward through captured data to see source of problem.
In-circuit emulators	Hardware emulator that plugs into CPU socket, appears to target system as regular microprocessor.

	Programmable or controlled by host.
Software-only monitor/debugging kernel	Small debugging program in ROM on target system that knows how to receive software over serial line, copy to RAM, and run it.

UNIT – V

TASK COMMUNICATION

Introduction

A more complex software architecture is needed to handle multiple tasks, coordination, communication, and interrupt handling – an RTOS architecture

Distinction:

Desktop OS – OS is in control at all times and runs applications, OS runs in different address space

RTOS – OS and embedded software are integrated, ES starts and activates the OS – both run in the same address space (RTOS is less protected)

RTOS includes only service routines needed by the ES application

RTOS vendors: VsWorks (we got it!), VTRX, Nucleus, LynxOS, uC/OS

Most conform to POSIX (IEEE standard for OS interfaces)

Desirable RTOS properties: use less memory, application programming interface, debugging tools, support for variety of microprocessors, already-debugged network drivers

What Is an O.S?

A piece of software

It provides tools to manage (for embedded systems)

Processes, (or tasks)

Memory space

What Is an Operating System?

What? It is a program (software) that acts as an intermediary between a user of a computer and the computer hardware.

Why? Make the use of a computer CONVENIENT and EFFICIENT.

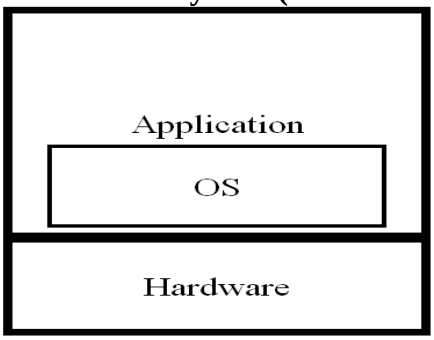
What Is an Operating System?For an Embedded System

Provides software tools for a convenient and prioritized control of tasks.

Provides tools for task (process) synchronization.

- Provides a simple memory management system

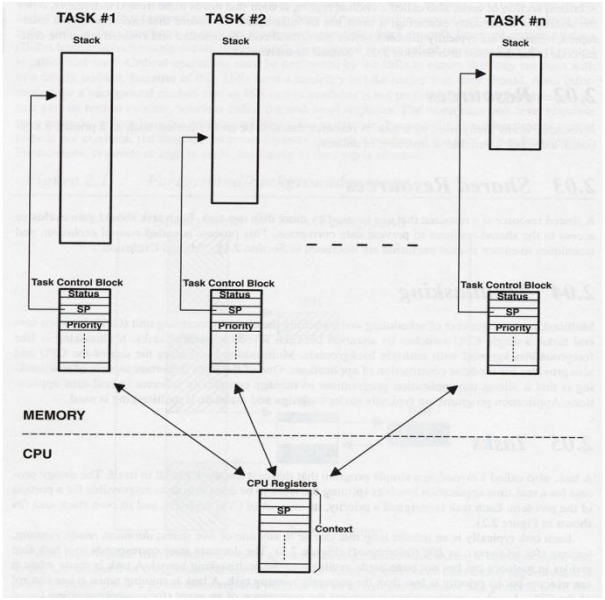
Abstract View of A System (*Embedded System*):



Process/Task Concept:

- Process is a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section

Multitasking:



Process/Task Concept:

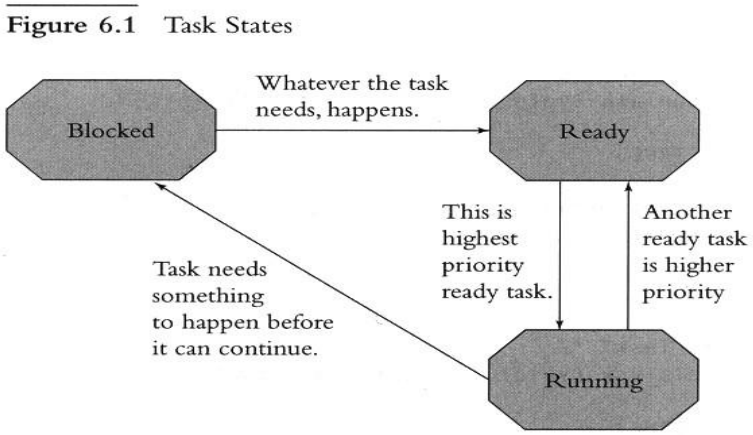
- ☐ Task States:
  - Running: Instructions are being executed
  - Ready: The process is waiting to be assigned to a process
  - Blocked: The process is waiting for some event to occur
  - terminated: The process has finished execution
  - new: The process is being created

Task states:

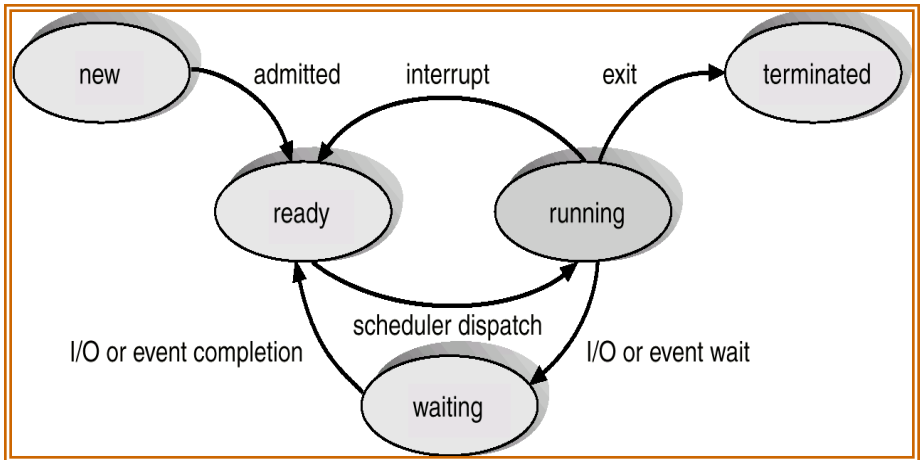
Tasks and Task States:

- A task – a simple subroutine
- ES application makes calls to the RTOS functions to start tasks, passing to the OS, start address, stack pointers, etc. of the tasks
- Task States:
  - Running
  - Ready (possibly: suspended, pended)
  - Blocked (possibly: waiting, dormant, delayed)
  - [Exit]
  - Scheduler – schedules/shuffles tasks between Running and Ready states

- Blocking is self-blocking by tasks, and moved to Running state via other tasks' interrupt signaling (when block-factor is removed/satisfied)
- When a task is unblocked with a higher priority over the 'running' task, the scheduler 'switches' context immediately (for all pre-emptive RTOSs)



Task State Transitions:



Tasks – 1:

- Issue – Scheduler/Task signal exchange for block-unblock of tasks via function calls
- Issue – All tasks are blocked and scheduler idles forever (not desirable!)
- Issue – Two or more tasks with same priority levels in Ready state (time-slice, FIFO)
- Example: scheduler switches from processor-hog vLevelsTask to vButtonTask (on user interruption by pressing a push-button), controlled by the main() which initializes the RTOS, sets priority levels, and starts the RTOS

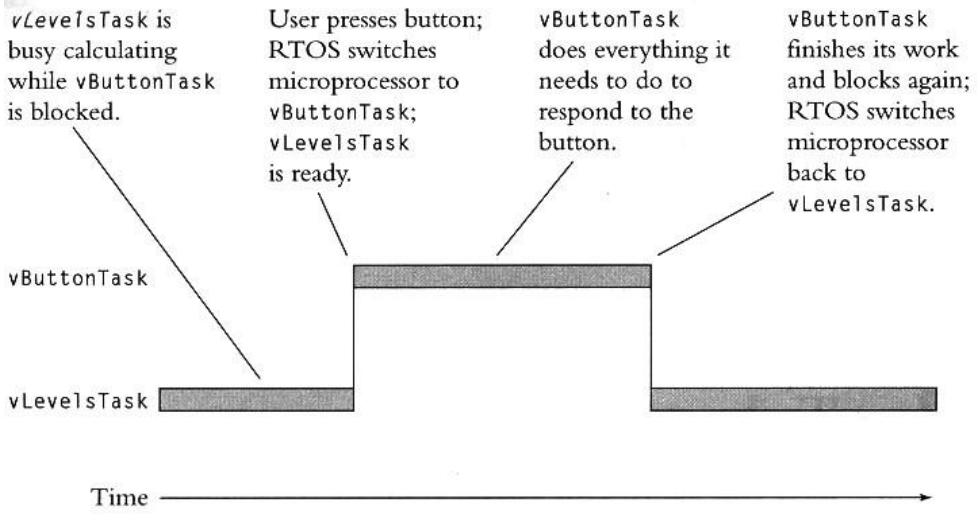
Figure 6.2 Uses for Tasks

```
/* "Button Task" */
void vButtonTask (void)    /* High priority */
{
   while (TRUE)
   {
      !! Block until user pushes a button
      !! Quick: respond to the user
   }
}

/* "Levels Task" */
void vLevelsTask (void)    /* Low priority */
{
   while (TRUE)
   {
      !! Read levels of floats in tank
      !! Calculate average float level                    (continued)
```

Figure 6.2 *(continued)*

```
      !! Do some interminable calculation
      !! Do more interminable calculation
      !! Do yet more interminable calculation

      !! Figure out which tank to do next
   }
}
```

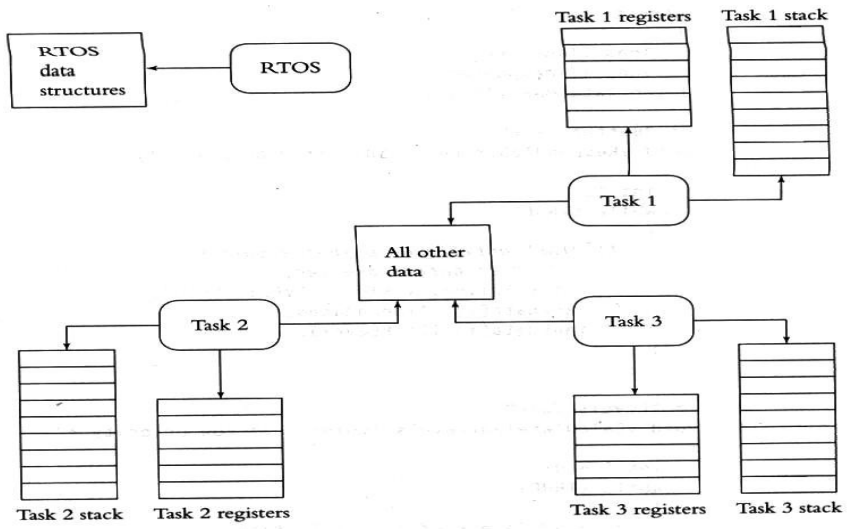
Figure 6.3 Microprocessor Responds to a Button under an RTOS



Tasks and Data:

- Each tasks has its won context - not shared, private registers, stack, etc.
- In addition, several tasks share common data (via global data declaration; use of 'extern' in one task to point to another task that declares the shared data
- Shared data caused the 'shared-data problem' without solutions discussed in Chp4 or use of 'Reentrancy' characterization of functions

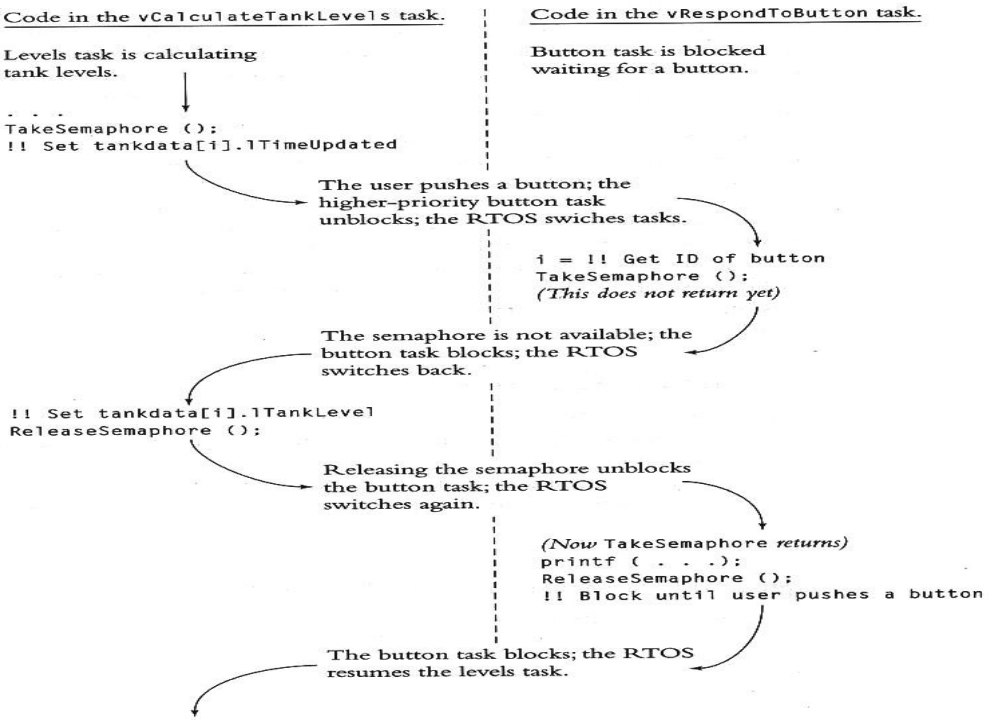
Figure 6.5 Data in an RTOS-Based Real-Time System



Semaphores and Shared Data – A new tool for atomicity

- Semaphore – a variable/lock/flag used to control access to shared resource (to avoid shared-data problems in RTOS)
- Protection at the start is via primitive function, called ***take***, indexed by the semaphore
- Protection at the end is via a primitive function, called ***release***, also indexed similarly
- Simple semaphores – Binary semaphores are often adequate for shared data problems in RTOS

Figure 6.13 Execution Flow with Semaphores



Semaphores and Shared Data – 1:

- RTOS Semaphores & Initializing Semaphores
- Using binary semaphores to solve the 'tank monitoring' problem
- The nuclear reactor system: The issue of initializing the semaphore variable in a dedicated task (not in a 'competing' task) before initializing the OS – timing of tasks and priority overrides, which can undermine the effect of the semaphores
- Solution: Call OSSemInit() before OSInit()

Figure 6.14 Semaphores Protect Data in the Nuclear Reactor

```
#define TASK_PRIORITY_READ 11
#define TASK_PRIORITY_CONTROL 12
#define STK_SIZE 1024
static unsigned int ReadStk [STK_SIZE];
static unsigned int ControlStk [STK_SIZE];

static int iTemperatures[2];
OS_EVENT *p_semTemp;
```

```
void main (void)
{
   /* Initialize (but do not start) the RTOS */
   OSInit ();

   /* Tell the RTOS about our tasks */
   OSTaskCreate (vReadTemperatureTask,  NULLP,
      (void *)&ReadStk[STK_SIZE], TASK_PRIORITY_READ);
   OSTaskCreate (vControlTask,  NULLP,
      (void *)&ControlStk[STK_SIZE], TASK_PRIORITY_CONTROL);

   /* Start the RTOS.  (This function never returns.) */
   OSStart ();
}
void vReadTemperatureTask (void)
{
   while (TRUE)
   {
      OSTimeDly (5); /* Delay about 1/4 second */

      OSSemPend (p_semTemp, WAIT_FOREVER);
      !! read in iTemperatures[0];
      !! read in iTemperatures[1];
      OSSemPost (p_semTemp);
   }
}

void vControlTask (void)
{
   p_semTemp = OSSemInit (1);
   while (TRUE)
   {
      OSSemPend (p_semTemp, WAIT_FOREVER);
      if (iTemperatures[0] != iTemperatures[1])
         !! Set off howling alarm;
      OSSemPost (p_semTemp);

      !! Do other useful work
   }
}
```

Semaphores and Shared Data – 2

- Reentrancy, Semaphores, Multiple Semaphores, Device Signaling,
- a reentrant function, protecting a shared data, cErrors, in critical section
- Each shared data (resource/device) requires a separate semaphore for individual protection, allowing multiple tasks and data/resources/devices to be shared exclusively, while allowing efficient implementation and response time
- example of a printer device signaled by a report-buffering task, via semaphore signaling, on each print of lines constituting the formatted and buffered report

Figure 6.15 Semaphores Make a Function Reentrant

```
void Task1 (void)
{
   :
   vCountErrors (9);
   :
}

void Task2 (void)
{
   :
   vCountErrors (11);
   :
}

static int cErrors;
static NU_SEMAPHORE semErrors;

void vCountErrors (int cNewErrors)
{
   NU_Obtain_Semaphore (&semErrors, NU_SUSPEND);
   cErrors += cNewErrors;
   NU_Release_Semaphore (&semErrors);
}
```

Figure 6.16 Using a Semaphore as a Signaling Device

```
/* Place to construct report. */
static char a_chPrint[10][21];

/* Count of lines in report. */
static int iLinesTotal;

/* Count of lines printed so far. */
static int iLinesPrinted;

/* Semaphore to wait for report to finish. */
static OS_EVENT *semPrinter;


void vPrinterTask(void)
{
   BYTE byError;   /* Place for an error return. */
   Int wMsg;

   /* Initialize the semaphore as already taken. */
   semPrinter = OSSemInit(0);

   while (TRUE)
   {
      /* Wait for a message telling what report to format. */
      wMsg = (int) OSQPend (QPrinterTask, WAIT_FOREVER, &byError);

      !! Format the report into a_chPrint
      iLinesTotal = !! count of lines in the report

      /* Print the first line of the report */
      iLinesPrinted = 0;
      vHardwarePrinterOutputLine (a_chPrint[iLinesPrinted++]);

      /* Wait for print job to finish. */
      OSSemPend (semPrinter, WAIT_FOREVER, &byError);
   }
}
```

(continued)

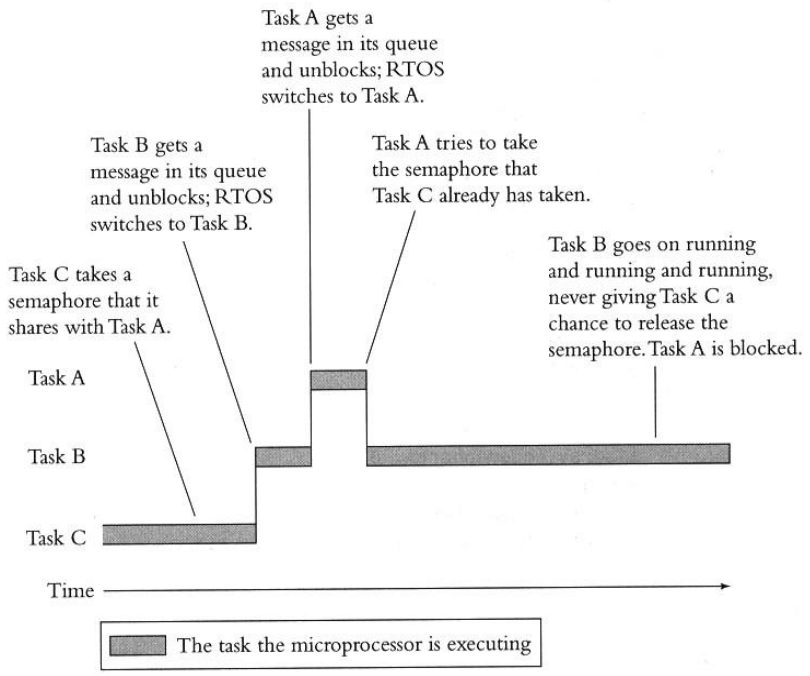
Figure 6.16 *(continued)*

```
void  vPrinterInterrupt (void)
{
   if (iLinesPrinted == iLinesTotal)
      /* The report is done.  Release the semaphore. */
      OSSemPost (semPrinter);

   else
      /* Print the next line. */
      vHardwarePrinterOutputLine (a_chPrint[iLinesPrinted++]);
}
```

semaphores and Shared Data – 3:

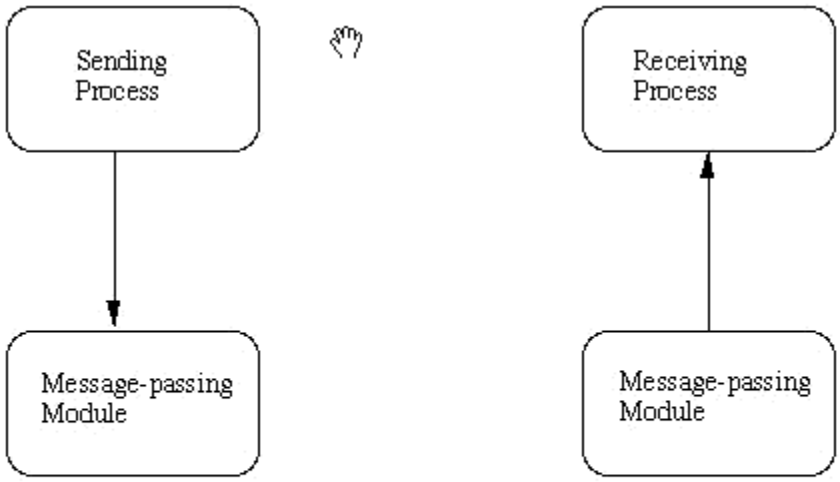
- Semaphore Problems – 'Messing up' with semaphores
  - The initial values of semaphores – when not set properly or at the wrong place
  - The 'symmetry' of takes and releases – must match or correspond – each 'take' must have a corresponding 'release' somewhere in the ES application
  - 'Taking' the wrong semaphore unintentionally (issue with multiple semaphores)
  - Holding a semaphore for too long can cause 'waiting' tasks' deadline to be missed
  - Priorities could be 'inverted' and usually solved by 'priority inheritance/promotion'

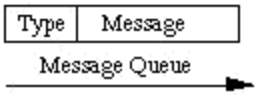
Figure 6.17 Priority Inversion

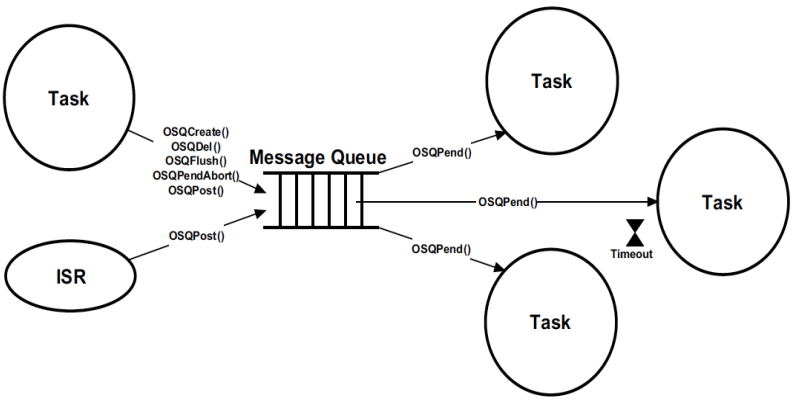


message queue :

Two (or more) processes can exchange information via access to a common system message queue. The *sending* process places via some (OS) message-passing module a message onto a queue which can be read by another process (Figure)Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place (subject to other permissions -- see below).







Basic Message Passing IPC messaging lets processes send and receive messages, and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length. Messages can be assigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Before a process can send or receive a message, the queue must be initialized (through the msgget function see below) Operations to send and receive messages are performed by the msgsnd() and msgrcv() functions, respectively.

When a message is sent, its text is copied to the message queue. The msgsnd() and msgrcv() functions can be performed as either blocking or non-blocking operations. Non-blocking operations allow for asynchronous message transfer -- the process is not suspended as a result of sending or receiving a message. In blocking or synchronous message passing the sending process cannot continue until the message has been transferred or has even been acknowledged by a receiver. IPC signal and other mechanisms can be employed to implement such transfer. A blocked message operation remains suspended until one of the following three conditions occurs:

- The call succeeds.
- The process receives a signal.
- The queue is removed.

Initialising the Message Queue :

The msgget() function initializes a new message queue:

```
int msgget(key_t key, int msgflg)
```

It can also return the message queue ID (msqid) of the queue corresponding to the key argument. The value passed as the msgflg argument must be an octal integer with settings for the queue's permissions and control flags.

The following code illustrates the msgget() function.

```
#include <sys/ipc.h>;
#include <sys/msg.h>;

...


key_t key; /* key to be passed to msgget() */
int msgflg /* msgflg to be passed to msgget() */
```

```
int msqid; /* return value from msgget() */

...
key = ...
msgflg = ...

if ((msqid = msgget(key, msgflg)) == &ndash;1)
  {
   perror("msgget: msgget failed");
   exit(1);
  } else
   (void) fprintf(stderr, &ldquo;msgget succeeded");
```

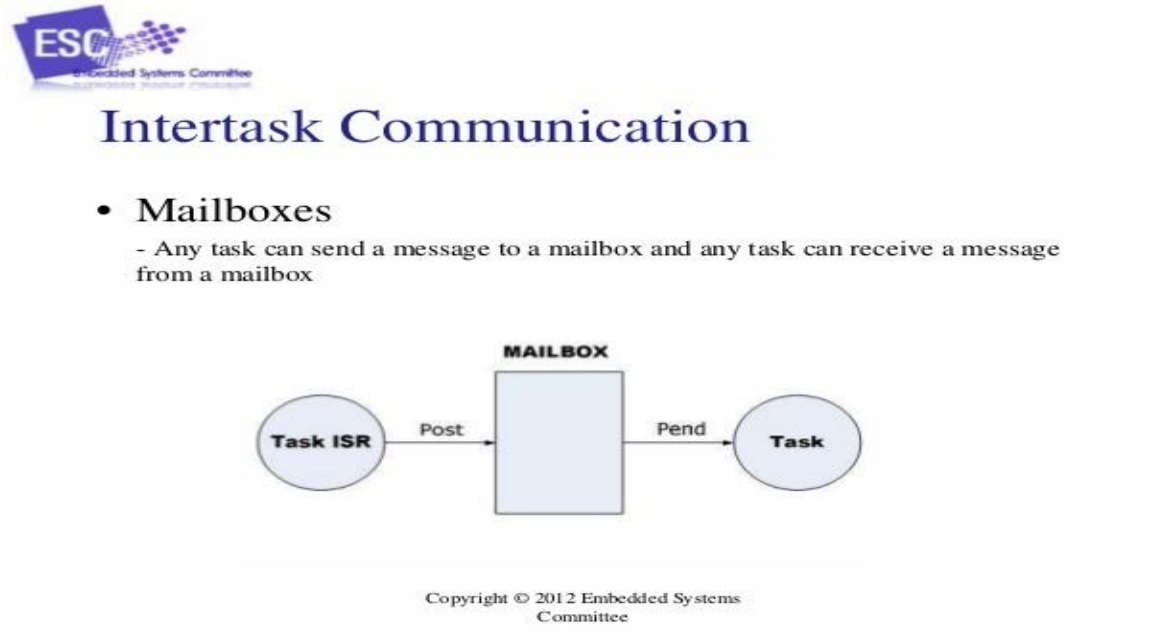
Mailbox:

- Mailbox (for message) is an IPC through a message-block at an OS that can be used only by a single destined task.
-
- A task on an OS function call puts (means post and also send) into the mailbox nly a pointer to a mailbox message

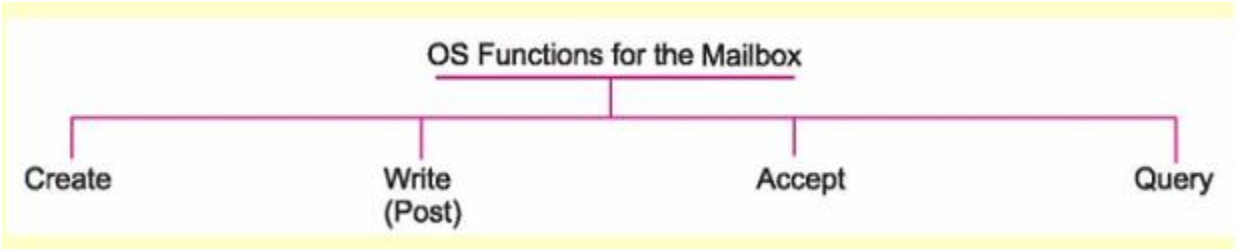
- Mailbox message may also include a header to identify the message-type specification.

Mailbox IPC features:

•OS provides for inserting and deleting message into the mailbox message- pointer. Deleting eans message-pointer pointing to Null.
•Each mailbox for a message need initialization (creation) before using the functions in the scheduler for the message queue and message pointer pointing to null



Mailbox Related Functions at the OS:



Pipe Function:

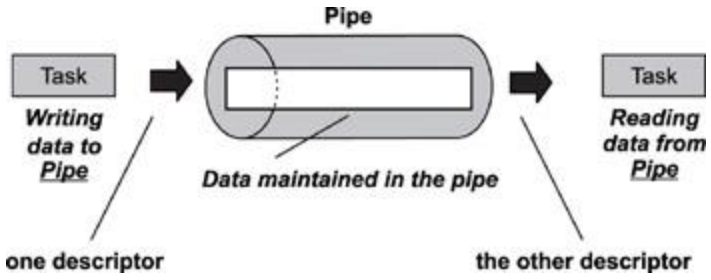
Pipe

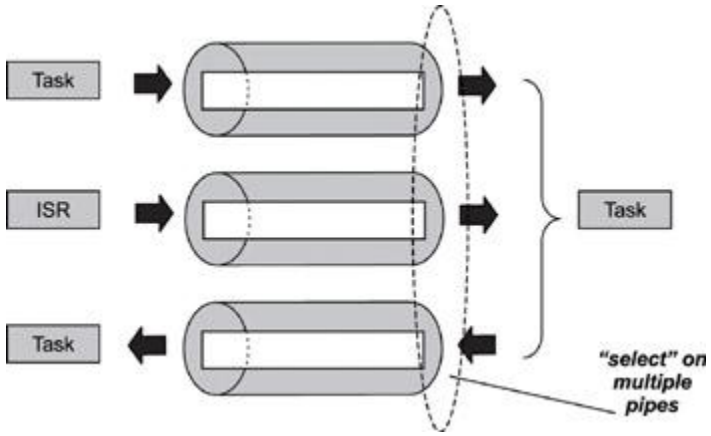
- Pipe is a device used for the interprocess communication
- Pipe has the functions create, connect and delete and functions similar to a device driver

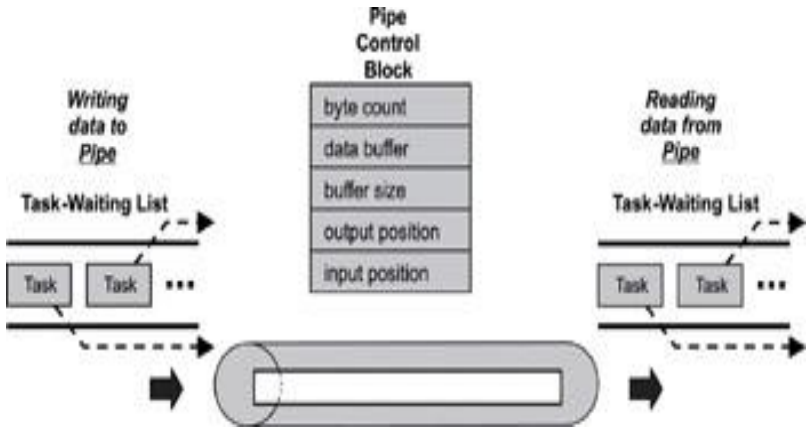
Writing and reading a Pipe:

• A message-pipe─ a device for inserting (writing) and deleting (reading) from that between two given inter-connected tasks or two sets of tasks.

• Writing and reading from a pipe is like using a C command*fwrite with a file name* to write into a named file, and C command *fread with a file name*to read into a named







Pipe function calls:

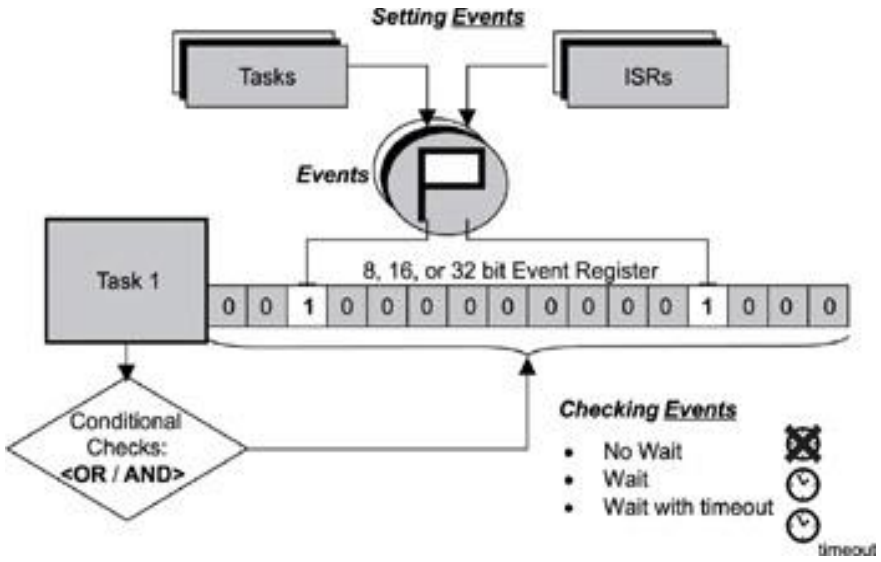
- Create a pipe
- Open pipe
- Close pipe
- Read from the pipe
- Write to the pipe

Event Functions:

- Wait for only one event (semaphore or mailboxmessage posting event)
- Event related OS functions can wait for number of events before initiating an action or wait for any of the predefined set of events
- Events for wait can be from different tasks or the ISRs

Event functions at OS:

Some OSes support and some don't support event functions for a group of event



Event registers function calls:

- Create an event register
- Delete an event register
- Query an event register
- Set an event register
- Clear an event register

- Each bit I an event register can be used to obtain the states of an event .
- A task can have an event register and other tasks can set/clear the bits in the event register

Signal:

- one way for messaging is to use an OS function signal ( ).
- Provided in Unix, Linux and several RTOSes.
- Unix and Linux OSes use signals profusely and have thirty-one different types of signals for the various events.
- A signal is the software equivalent of the flag at a register that sets on a hardware interrupt. Unless masked by a signal mask, the signal allows the execution of the Signal handling function and allows the handler to run just as a hardware interrupt allows the execution of an ISR
- Signal provides the shortest communication.

Signal management fuction calls:

- Install a signal handler
- Remove an installed signal handler
- Send a signal to another task
- Block a signal from being delivered
- Unblock a blocked signal
- Ignore a signal

Timers:

- Real time clock ─ system clock, on each tick SysClkIntr interrupts
- Based on each SysClkIntr interrupts─ there are number of OS timer functions
- Timer are used to message the elasped time of events for instance , the kernel has to keep track of different times

The following functions calls are provided to manage the timer

- Get time
- Set time
- Time delay(in system clock)
- Time delay(in sec.)
- Reset timer

Memory management:

Memory allocation:

- Memory allocation When a process is created, the memory manager allocates the memory addresses (blocks) to it by mapping the process address space.
- Threads of a process share the memory space of the process

Memory Managing Strategy for a system

- Fixed
- blocks allocation
- Dynamic
- blocks Allocation
- Dynamic Page
- Allocation
- Dynamic Data memory Allocation

Interrupt service routine (ISR):

- Interrupt is a hardware signal that informs the cpu that an important event has occurred when interrupt occured, cpu saves its content and jumps to the ISR
- In RTOS
    - Interrupt latency
    - Interrupt response
    - Interrupt recovery

Mutex:

Mutex standards for mutual exclusion ,mutex is the general mechanism used for both rsource synchronization as well as task synchronization

It has following mechanisms

- Disabling the scheduler
- Disabling the interrupts
- By test and set operations
- Using semaphore