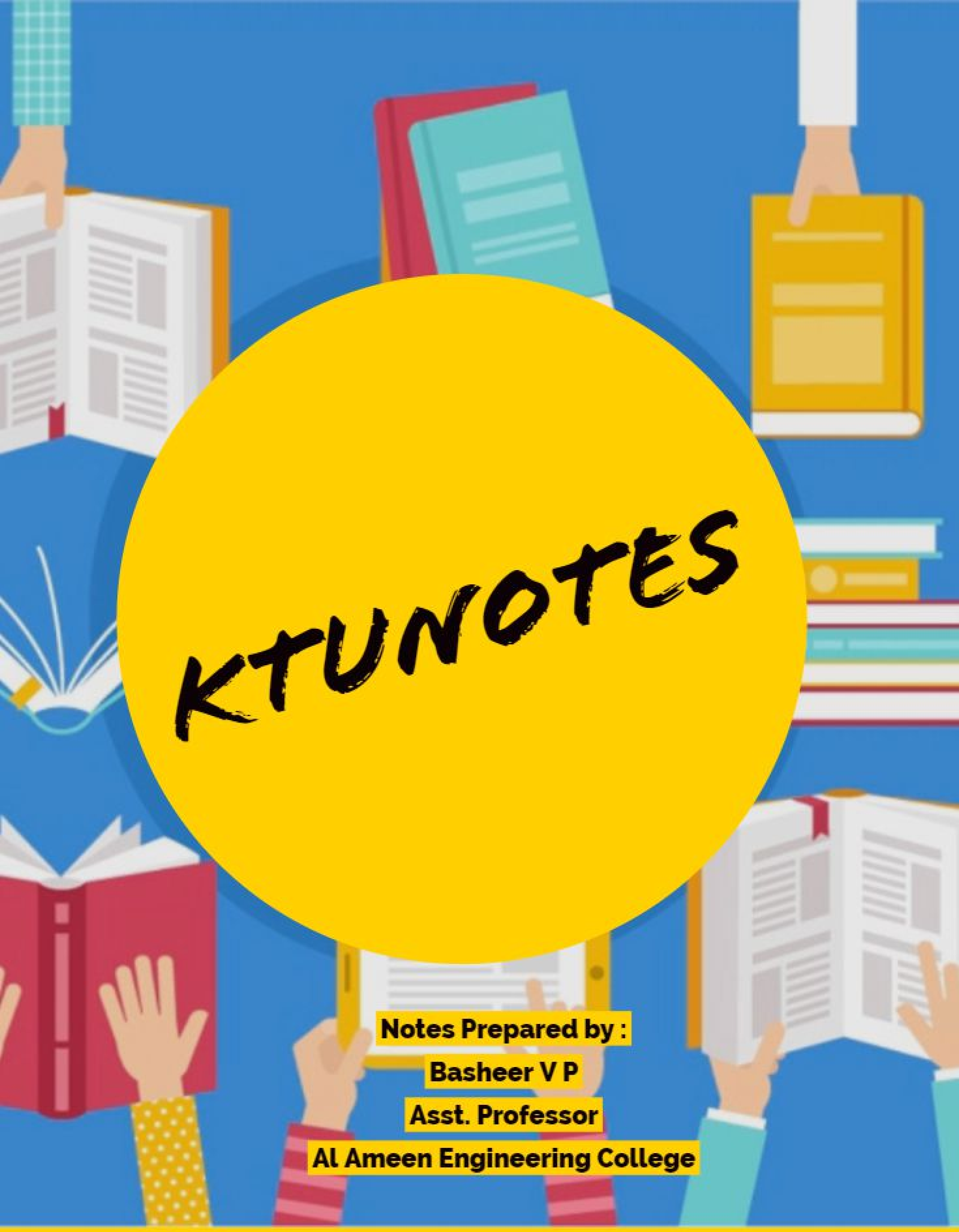


Notes Prepared by :
Basheer V P
Asst. Professor
Al Ameen Engineering College

EC 308 EMBEDDED SYSTEMS

MODULE-3

SYLLABUS

Memory devices and systems - memory map – DMA - I/O Devices – Interrupts - ISR – Device drivers for handling ISR – Memory Device Drivers – Device Drivers for on-board bus.

3.1 Memory devices

Memory is the storage device in a system used to store data and instructions.

Most systems consist of two types of memory:

- Read-only Memory (ROM) and Random-Access Memory (RAM).

ROM

- ROM stands for Read Only Memory.
- The memory from which we can only read but cannot write onit.
- This type of memory is non-volatile.
- An embedded system has ROM unit(s) for storing ROM image and flash to save non- volatile data and results
- ROM image holds the programs, operating system, and data required by the system.

Following are the various types of ROM:

MROM (Masked ROM)

- The very first ROMs were hard-wired devices that contained a pre programmed set of data or instructions. These kind of ROMs are known as masked ROMs.
- MROM programming is performed during IC fabrication.
- It is inexpensive ROM.
- Used for large scale manufacturing

Uses:

1. A finalised ROM image of system program.
2. Data, pictograms, image pixels, pixels for the fonts of a language.

PROM (One Time Programmable ROM –OTP ROM)

- PROM is read-only memory that can be modified only once by a user.
- The user buys a blank PROM and enters the desired contents using a PROM programmer.
- A PROM once written is not erasable.

Uses:

1. Smart card identity number and personal information.
2. Storing boot programs.
3. ATM card or credit card or identity card.

EPROM (Erasable and Programmable ROM)

- Used in place of masked ROM during development phase.
- UV Erasable and Electrically programmable by a device programmer

EPROM (Erasable and Programmable ROM)

- The EEPROM is programmed and erased electrically.
- It can be erased and reprogrammed about ten thousand times.
- In EEPROM, any location can be selectively erased and programmed.
- EEPROMs can be erased one byte at a time, rather than erasing the entire chip. Hence, the process of re programming is flexible but slow.

Uses:

1. Storing current date and time in a machine.
2. Storing port status.
3. Storing driving, malfunctions and failure history in an automobile.

Flash Memory

- Flash memory is a form of EEPROM in which a sector of bytes can be erased in a flash (very short duration corresponding to a single clock cycle)

Uses:

1. Storing pictures in digital camera.
2. Storing SMS, MMS messages in a mobile phone.
3. Storing voice compressed form in a voice recorder.

RAM

- The RAM can be both read and, write.
- RAM is used to hold the programs, operating system, and data required by a computer system.
- In embedded systems, it holds the stack and temporary variables of the programs, operating system, and data.
- RAM is generally volatile, (does not retain the data stored in it when the system 's power is turned off)
- Used for saving the variables, stacks, process control blocks, input buffer, output buffer, decompressed format of program and data at the ROM image.

Following are the various types of ROM:

SRAM (static RAM)

- Storage elements in SRAM are transistors.
- SRAM is most commonly used for designing cache memory.

DRAM (dynamic RAM)

- Storage elements in SRAM are capacitors.
- SRAM is most commonly used in high performance computers or high memory density systems.

EDO (Extended Data Out) RAM

- EDO RAM is used in systems with clock rates up to 100 MHz.
- Zero wait state is needed between two fetches.
- Single cycle read or write is possible.

SDRAM (Synchronous DRAM)

- SDRAM Synchronises the read operation and keeps next word ready while previous one is being fetched;
- SDRAM is used in systems with clock rates up to 1 GHz.

RDRAM (Rambus* DRAM)

- RDRAM accesses in bursts of four successive words in single fetch; .
- used for 1 GHz + performance of the system

Parameterised Distributed RAM

- Parameterised Distributed RAM is the RAM distributes in various system subunits.
- IO units and transceiver sub units can have a slice of RAM each.

3.2 Memory Map

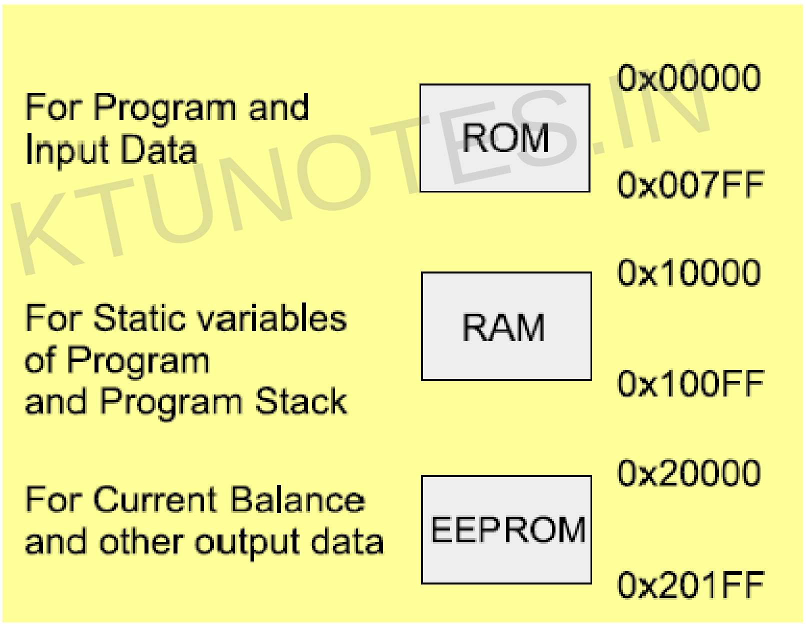
- A **memory map** is a massive table, in effect a database that comprises complete information about how the memory is structured in a computer **system**.
- Map to show the program and data allocation of the addresses to ROM, RAM, EEPROM or Flash in the system.
- It reflects the addresses available to the memory blocks and IO devises.

- It also reflects a description of memory and IO devices in the system hardware.
- It reflects memory allocation for the programs, data and IO operations by a locator program.

Example-1

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

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


For Program and Input Data
ROM
0x00000
0x007FF
For Static variables of Program and Program Stack
RAM
0x10000
0x100FF
For Current Balance and other output data
EEPROM
0x20000
0x201FF

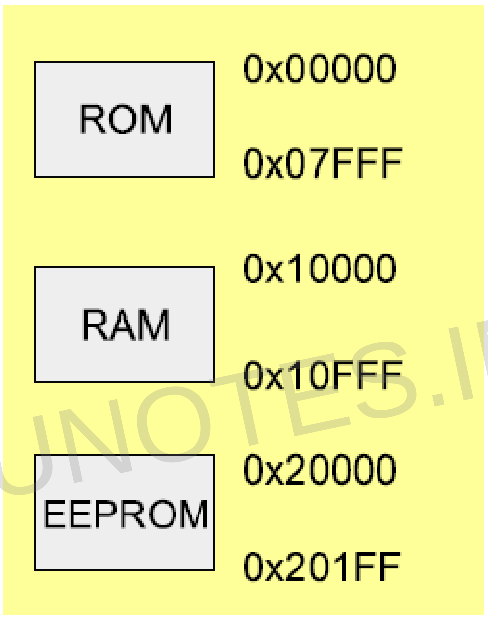
Memory map for example-1

Example-2

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

Memory

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



Memory map for example-2

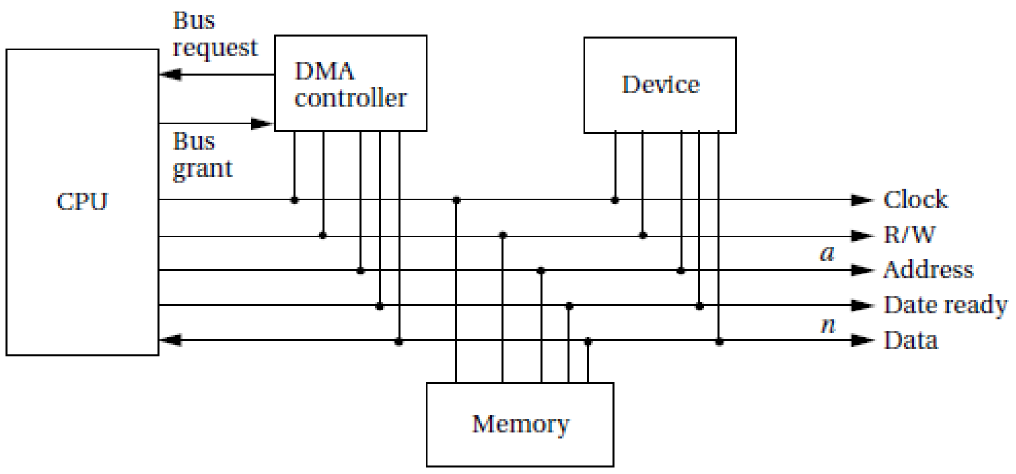
- There are memory space gaps between the origin of ROM,RAM and EEPROM in spite of very small lengths of available memory.
- This gap is due to a design feature of its hardware designer providing expansion of these memories in the future.

3.3 <u>DMA (Direct Memory Access)</u>

.

- ***Direct memory access (DMA)*** is a bus operation that allows read and write between IO devices and memory not controlled by the CPU.
- A DMA transfer is controlled by a ***DMA controller***, which requests control of the bus from the CPU.

- After gaining control, the DMA controller performs read and write operations directly between devices and memory.
- Figure shows the configuration of a bus with a DMA controller.



DMA

- The DMA requires the CPU to provide two additional bus signals:

  ■ The ***bus request*** is an input to the CPU through which DMA controllers ask for ownership of the bus.

  ■ The ***bus grant*** signals that the bus has been granted to the DMA controller.

- The CPU controls the DMA operation through registers in the DMA controller.
- A typical DMA controller includes the following three registers:

  ■ A starting address register specifies where the transfer is to begin.

  ■ A length register specifies the number of words to be transferred.

  ■ A status register allows the DMA controller to be operated by the CPU.

DMA Process:

- The bus request is asserted by the DMA controller when it wants to control the bus.
- The CPU will finish all pending bus transactions before granting control of the bus to the DMA controller.
- The bus grant is asserted by the CPU when the bus is ready.

- The CPU initiates a DMA transfer by setting the starting address and length registers appropriately.
- Upon becoming bus master, the DMA controller has control of all bus signals
- Once the DMA controller is bus master, it can perform read and write using the same bus protocol as with any CPU-driven bus transaction.
- After the transaction is finished, the DMA controller returns the bus to the CPU by deasserting the bus request, causing the CPU to deassert the bus grant.

3.4 IO Devices

- Some of the input and output devices commonly used in embedded computing systems are explained below.
- Some of these devices are often found as on-chip devices in micro-controllers; others are generally implemented separately but are still commonly used.

1. **Timers and Counters**

- ***Timers*** and ***counters*** are distinguished from one another largely by their use, not their logic.
- Both are built from adder logic with registers to hold the current value, with an increment input that adds one to the current register value.
- A timer has its count connected to a periodic clock signal to measure time intervals,
- A counter has its count input connected to an a periodic signal in order to count the number of occurrences of some external event.
- Because the same logic can be used for either purpose, the device is often called a ***counter/timer***.
- Figure shows the internals of a counter/timer to illustrate its operation.

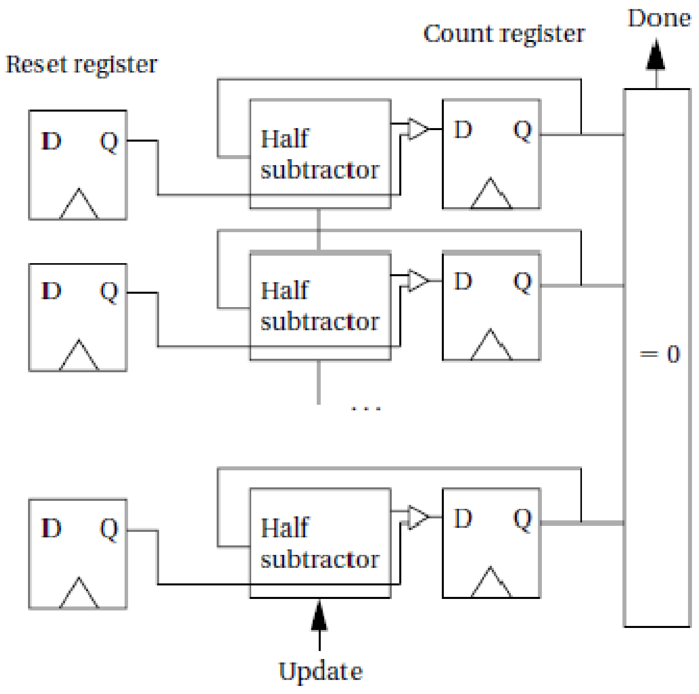


Figure: Internals of a counter/timer

- An *n*-bit counter/timer uses an *n*-bit register to store the current state of the count and an array of *half subtractors* to decrement the count when the count signal is asserted.
- Combinational logic checks when the count equals zero and the *done* output signals the zero count.
- It is often useful to be able to control the time-out, rather than require exactly 2^n events to occur.
- For this purpose, a reset register provides the value with which the count register is to be loaded.
- The counter/timer provides logic to load the reset register.
- Most counters provide both cyclic and acyclic modes of operation. In the cyclic mode, once the counter reaches the *done* state, it is automatically reloaded and the counting process continues.

2. A/D and D/A Converters

- ***Analog/digital (A/D)*** and ***digital/analog (D/A)*** converters (typically known as **ADCs** and **DACs**, respectively) are often used to interface non digital devices to embedded systems.
- Analog/digital conversion requires sampling the analog input before converting it to digital form.
- A control signal causes the A/D converter to take a sample and digitize it.
- A typical A/D interface has, in addition to its analog inputs, two major digital inputs. A data port allows A/D registers to be read and written, and a clock input tells when to start the next conversion.
- D/A conversion is relatively simple, so the D/A converter interface generally includes only the data value. The input value is continuously converted to analog form.

3. Keyboards

- A keyboard is basically an array of switches, but it may include some internal logic to help simplify the interface to the microprocessor.
- A switch uses a mechanical contact to make or break an electrical circuit.
- An ***encoded keyboard*** uses some code to represent which switch is currently being depressed.
- At the heart of the encoded keyboard is the scanned array of switches shown in Figure.

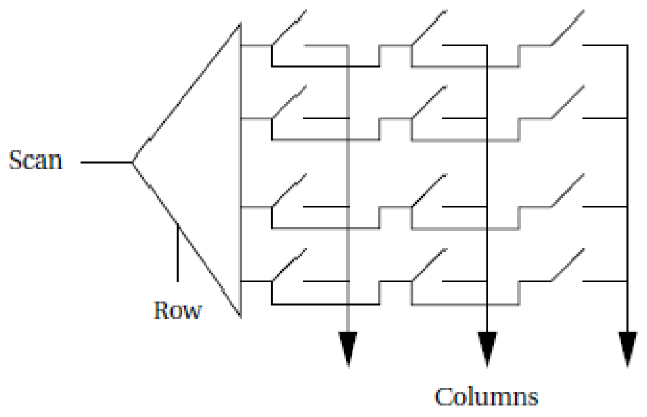
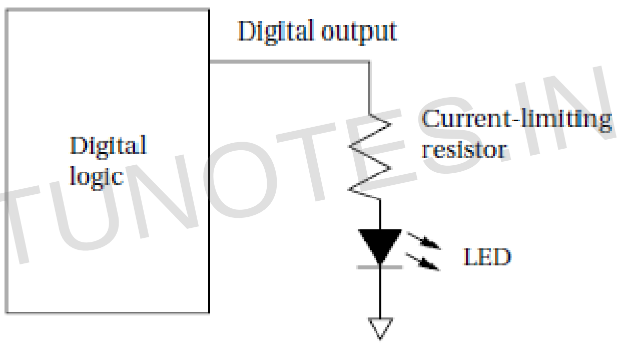


Figure: A scanned key array

- The scanned keyboard array reads only one row of switches at a time.
- The demultiplexer at the left side of the array selects the row to be read.
- When the scan input is 1, that value is transmitted to one terminal of each key in the row.
- If the switch is depressed, the 1 is sensed at that switch's column. Since only one switch in the column is activated, that value uniquely identifies a key.
- The row address and column output can be used for encoding, or circuitry can be used to give a different encoding.

4. LEDs

- ***Light-emitting diodes (LEDs)*** are often used as simple displays by themselves.
- Arrays of LEDs may form the basis of more complex displays.
- Figure shows how to connect an LED to a digital output.



An LED connected to a digital output

- A resistor is connected between the output pin and the LED to limit the current.
- When the digital output goes to 0, the LED voltage is in the device's off region and the LED is not on.

5. **Displays**

- A display device may be either directly driven or driven from a frame buffer.
- Typically, displays with a small number of elements are driven directly by logic, while large displays use a RAM frame buffer.
- The *n*-digit array, shown in Figure is a simple example of a display that is usually directly driven.

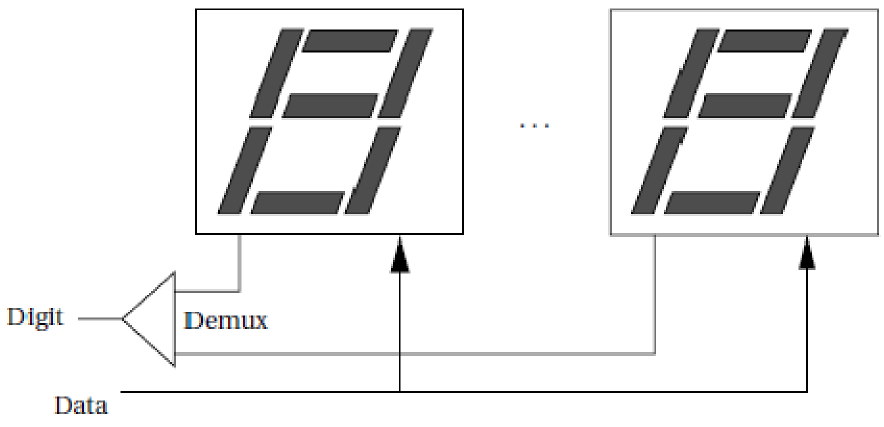


Figure: An n digit display

- A single-digit display typically consists of seven segments; each segment may be either an LED or a ***liquid crystal display (LCD)*** element.
- The digit input is used to choose which digit is currently being updated, and the selected digit activates its display elements based on the current data value
- Many large displays are built using LCD. Each pixel in the display is formed by a single liquid crystal.
- LCD displays present a very different interface to the system because the array of pixel LCDs can be randomly accessed.

6. Touch screens

- A ***touch screen*** is an input device overlaid on an output device. The touch screen registers the position of a touch to its surface. By overlaying this on a display, the user can react to information shown on the display.
- The two most common types of touch screens are resistive and capacitive. A resistive touch screen uses a two-dimensional voltmeter to sense position. As shown in Figure , the touch screen consists of two conductive sheets separated by spacer balls. The top conductive sheet is flexible so that it can be pressed to touch the bottom sheet.
- A voltage is applied across the sheet; its resistance causes a voltage gradient to appear across the sheet. The top sheet samples the conductive sheet's applied voltage at the contact point. An analog/digital converter is used to measure the voltage and resulting position.

- The touch screen alternates between *x* and *y* position sensing by alternately applying horizontal and vertical voltage gradients. information shown on the display.
- The two most common types of touch screens are resistive and capacitive. A resistive touch screen uses a two-dimensional voltmeter to sense position. As shown in Figure , the touch screen consists of two conductive sheets separated by spacer balls. The top conductive sheet is flexible so that it can be pressed to touch the bottom sheet.
- A voltage is applied across the sheet; its resistance causes a voltage gradient to appear across the sheet. The top sheet samples the conductive sheet's applied voltage at the contact point. An analog/digital converter is used to measure the voltage and resulting position.
- The touch screen alternates between *x* and *y* position sensing by alternately applying horizontal and vertical voltage gradients.

3.5 Interrupts

- In system programming, an **interrupt** is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention.
- An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing.
- The processor responds by suspending its current activities, saving its state, and executing a function called an *interrupt handler* (or an interrupt service routine, ISR) to deal with the event.
- This interruption is temporary, and, after the ISR finishes, the processor resumes normal activities

Interrupt Sources/ Types of Interrupts

There are two types of interrupts: hardware interrupts and software interrupts.

1. **Hardware Interrupt Sources:**

- Hardware sources can be internal or external for interrupt of ongoing routine and thereby diversion to corresponding ISR.

- The internal sources from devices differ in different processor or microcontroller or device and their versions and families
- External sources and ports also differ in different processors or microcontrollers
- There are four possible sources of hardware interrupts as follows

a) Internal Hardware Device Sources

1 Parallel Port
2. UART Serial Receiver Port - [Noise, Overrun,
Frame-Error, IDLE, RDRF in 68HC11]
3. Synchronous Receiver byte Completion
4. UART Serial Transmit Port-Transmission Complete, [For example, TDRE (transmitter data register Empty]
5. Synchronous Transmission of byte completed
6. ADC Start of Conversion
7. ADC End of Conversion
8. Pulse-Accumulator overflow
9. Real Time Clock time-outs
10. Watchdog Timer Reset
11. Timer Overflow on time-out
12. Timer comparison with Output compare Registers
13. Timer capture on inputs

b) External Hardware interrupts with also sending vector address

- INTR in 8086 and 80x86 ─ The device provides the ISR Address or Vector Address or Type externally on data bus after interrupt at INTR pin

c) External Hardware Interrupts with Internal Vector Address Generation

- Non-Maskable Pin─ NMI in 8086 and 80x86
- Maskable Pins (interrupt request pin) ─INT0 and INT 1 in 8051, IRQ in 68HC11

d) Sources of interrupts due to Processor Hardware detecting Software error

- Software sources for interrupt are related to processor detecting computational error during execution such as division by 0, illegal opcode or overflow (for example, multiplication of two numbers exceeding the limit)
  1. Division by zero detection (or *trap*) by hardware
  2. Over-flow detection by hardware
  3. Under-flow detection by hardware
  4. Illegal opcode detection by hardware

2. Software Interrupts/ Software Interrupt Sources

- Software sources for interrupt are related to software detecting computational error or exceptional condition during execution and there up on executing a SWI (software interrupt) instruction, which causes processor interrupt of ongoing routine

- Software interrupt is interrupt-generated, for example, by a software instruction Int *n* in 80x86 processor or SWI *m* in ARM7, where n is interrupt type and *m* is 24 bits related to ISR address pointer and ISR input parameters pointer.

3.6 ISR (Interrupt Service Routine) and Interrupt Mechanism

- ISR is a small program that the processor executes when the corresponding Interrupt is being requested.
- For every interrupt, there must be an interrupt service routine (ISR), or **interrupt handler**. When an interrupt occurs, the processor/microcontroller runs the interrupt service routine.

When an interrupt occurs

- Processor masks further external interrupts.
- Save the current context on stack
- Then executes the appropriate interrupt service routine (ISR).
- Upon return from the ISR restores the context and enables interrupts and return.

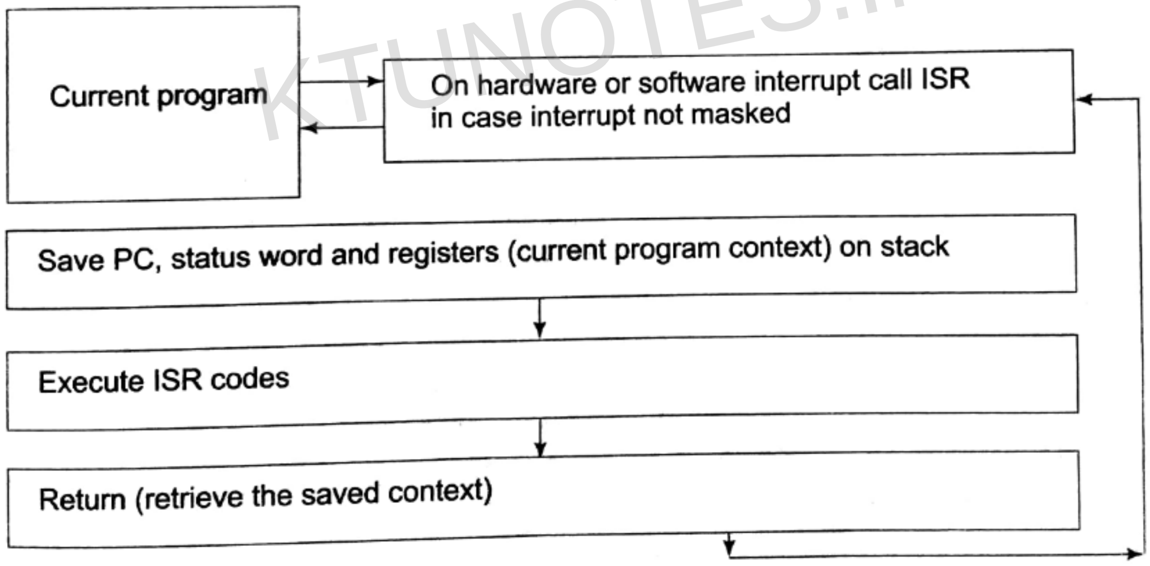

Current program
On hardware or software interrupt call ISR in case interrupt not masked
Save PC, status word and registers (current program context) on stack
Execute ISR codes
Return (retrieve the saved context)

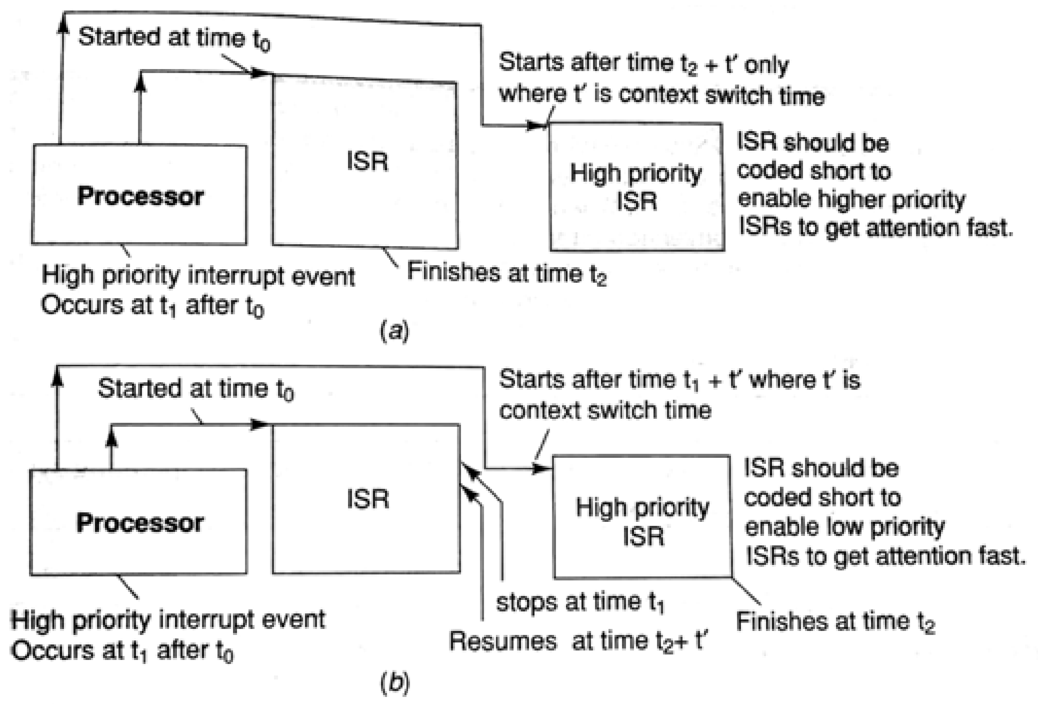
Figure: Interrupt handling mechanism

Multiple Interrupts: (Non Nested ISR and Nested ISR)

- In a system with multiple interrupts, each and every interrupt may be assigned a priority level.
- Here there is a possibility to occur multiple interrupts at a time.
- There are two types of interrupt service mechanisms for the case of multiple interrupts. Non Nested ISR and Nested ISR

Non Nested ISR:

- In this case, Processors don't permit in-between routine diversion to higher priority interrupts.
- These processors provide auto disabling of all maskable interrupt when ISR execution starts and auto re-enabling of all maskable interrupt on return from ISR.
- These processors may also provide for auto saving the CPU registers (context) when ISR execution starts and auto return of saved values into the CPU registers on return from ISR. This help in fast transfer to pending higher priority interrupt
- Figure (a) shows multiple interrupts with diversion to higher priority interrupts only at the end of present ISR.
- Whenever an interrupt occurs the processor diverts the execution to the corresponding ISR (at time t_0). During the execution of this first ISR, suppose another interrupt with higher priority occurs (at time t_1)but the processor will continue the first ISR and finished it (at time t_2) then only execution will be diverted to the second ISR (at time t_2+t').



Started at time t_0
Starts after time $t_2 + t'$ only where t' is context switch time
ISR
Processor
High priority ISR
ISR should be coded short to enable higher priority ISRs to get attention fast.
High priority interrupt event Occurs at t_1 after t_0
Finishes at time t_2
(a)
Started at time t_0
Starts after time $t_1 + t'$ where t' is context switch time
ISR
Processor
High priority ISR
ISR should be coded short to enable low priority ISRs to get attention fast.
stops at time t_1
Resumes at time $t_2+ t'$
Finishes at time t_2
High priority interrupt event Occurs at t_1 after t_0
(b)

(a) Diversion to higher priority interrupts, only at the end of the present interrupt service routine (Non Nested ISR)
(b) In-between routine diversion to higher priority interrupts (Nested ISR)

Nested ISR:

- In this case, processors permit in-between routine diversion to higher priority interrupts.
- Figure (b) shows multiple interrupts with diversion to higher priority interrupts in between ISR.
- Whenever an interrupt occurs the processor diverts the execution to the corresponding ISR (at time t_0). During the execution of this first ISR, suppose another interrupt with higher priority occurs (at time t_1) then the processor will stop the first ISR (pending) and execution will be diverted to the second ISR (at time t_1+t').
- After completion of second ISR (at time t_2) the pending ISR will be resumed at time t_2+t'. After finishing this first ISR, the execution will be returned to the main program.
- These processors may also provide for auto saving the CPU registers (context) when ISR execution starts and auto return of saved values into the CPU registers on return from ISR. This help in fast transfer to higher priority interrupt

Context

- Whenever an interrupt occurs, processor stops the currently doing program and saves some values including program counter, stack pointer, status register, CPU registers etc before diverting the execution to the corresponding ISR. The saved data during the ISR call is known as context.

Context switching

Whenever an interrupt occurs, before start the ISR the following steps carried out:

1. Saving all the CPU registers including processor status word, registers and function's current address for next instruction in the PC. Saving the address of the PC onto a stack is required if there is no link register to point to the PC of left instruction of earlier function. Saving facilitates the return from the new function to the previous state.
2. Load new context to switch to a new function.
3. Readjust the contents of stack pointer and execute the new function.

These three actions are known as *context switching*.

The last instruction (action) of any routine or function is always a *return*. The following steps occur during return from the called function.

1. Before return, retrieve the previously saved status word, registers and other context parameters.
2. Retrieve into the PC the saved PC (address) from the stack or link register and load other part of saved context from stack and readjust the contents of stack pointer.
3. Execute the remaining part of the function, which called the new function.

These three actions are also known as *context switching*.

Context Switching Time

- Time taken to complete the context switching is called the context switch time.

Interrupt Latency

Interrupt Latency

When an interrupt occurs the service of the interrupt by executing the ISR may not start immediately by context switching. The interval between the occurrence of an interrupt and start of execution of the ISR is called interrupt latency.

Interrupt Vector Table

- For every interrupt, there is a fixed location in memory that holds the address of its interrupt service routine, ISR.
- The table of memory locations set aside to hold the addresses of ISRs is called as the Interrupt Vector Table.

3.7 Device Drivers

- Most embedded hardware requires some type of software initialization and management.
- A device driver is a function used by a high-level language programmer, which does the interaction with the device hardware, sends control commands to the device, communicates data to the device and runs the codes for reading device data.
- Device drivers are the software libraries that initialize the hardware, and manage access to the hardware by higher layers of software.

1. Device Drivers for Interrupt-Handling

- The software that handles interrupts on the master processor and manages interrupt hardware mechanisms (i.e., the interrupt controller) consists of the *device drivers* for interrupt-handling.
- These functions implemented in software usually depend on the following criteria:
  - The types, number, and priority levels of interrupts available (determined by the interrupt hardware mechanisms on-chip and on-board).
  - How interrupts are triggered.
  - The interrupt policies of components within the system that trigger interrupts, and the services provided by the master CPU processing the interrupts.

Following are some of the device drivers for handling interrupt.

Interrupt-Handling Startup:

- This driver is used for initialization of the interrupt hardware (i.e., interrupt controller, activating interrupts, etc.) upon power-on or reset.

Interrupt-Handling Shutdown;

- This driver is used for configuring interrupt hardware (i.e., interrupt controller, deactivating interrupts, etc.) into its power-off state.

Interrupt-Handling Disable:

- This driver is used for allowing other software to disable active interrupts on the-fly (not allowed for Non-Maskable Interrupts (NMIs), which are interrupts that cannot be disabled).

Interrupt-Handling Enable:

- This driver is used for allowing other software to enable inactive interrupts on-the-fly.

Interrupt-Handler Servicing:

- This driver is used for the interrupt-handling code itself, which is executed after the interruption of the main execution stream (this can range in complexity from a simple non-nested routine to nested and/or re entrant routines).

2. <u>Memory Device Drivers</u>

- The software must provide the processors in the system with the ability to access various portions of the memory map.
- The software involved in managing the memory on the master processor and on the board, as well as managing memory hardware mechanisms, consists of the *device drivers* for the management of the overall memory subsystem.

Following are some of the device drivers for handling memory

Memory Subsystem Startup

- This driver is used for initialization of the hardware upon power-on or reset (initialize TLBs for MMU, initialize/configure MMU).

Memory Subsystem Shutdown

- This driver is used for configuring hardware into its power-off state.

Memory Subsystem Disable

- This driver is used for allowing other software to disable hardware on-the-fly (disabling cache).

Memory Subsystem Enable

- This driver is used for allowing other software to enable hardware on-the-fly (enable cache).

Memory Subsystem Write

- This driver is used for storing in memory a byte or set of bytes (i.e., in cache, ROM, and main memory).

Memory Subsystem Read

- This driver is used for retrieving from memory a “copy” of the data in the form of a byte or set of bytes (i.e., in cache, ROM, and main memory).

3. <u>On-board Bus Device Drivers</u>

- Every bus is associated with some *protocol* that defines
  - How devices gain access to the bus.
  - The rules must followed by the devices to communicate over the bus and
  - The signals associated with the bus lines.
- Bus protocol is supported by the bus device drivers

Following are some of the device drivers for handling on board bus

Bus Startup

- This driver is used for initialization of the bus upon power-on or reset.

Bus Shutdown

- This driver is used for configuring bus into its power-off state.

Bus Disable

- This driver is used for allowing other software to disable bus on-the-fly.

Bus Enable

- This driver is used for allowing other software to enable bus on-the-fly.

Bus Acquire

- This driver is used for allowing other software to gain singular (locking) access to bus.

Bus Release

- This driver is used for allowing other software to free (unlock) bus.

Bus Read

- This driver is used for allowing other software to read data from bus.

Bus Write

- This driver is used for allowing other software to write data to bus.

Bus Install

- This driver is used for allowing other software to install new bus device on-the-fly for expandable buses.

Bus Uninstall

- This driver is used for allowing other software to remove installed bus device on-the-fly for expandable buses.

Which of the routines are implemented and how they are implemented depends on the actual bus.