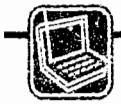


Practice Exercises

15. How do the following device features help in embedded systems? (a) Schmitt trigger input (b) low voltage 3.3 V IOs (c) Dynamically controlled impedance matching (c) PCS subunit (d) PMA subunit and (e) SerDes. Give one exemplary application of each.
16. PPP protocol for point to point networking has 8 starting flag bits, 8 address bits, 8 protocol specification bits, variable number of data bits, 16-bit CRC and 8 ending flag bits. The maximum number of bits per PPP frame can be 12064. How many maximum number of bytes can be transferred per PPP frame? What is the minimum percentage of overhead in the payload (frame)?
17. List the applications of the free running counter, periodically interrupting timer and pulse accumulator counter (PACT). How do you get PWM output from a PACT? How do you get DAC output from a PWM device?
18. A 16-bit counter is getting inputs from an internal clock of 12 MHz. There is a prescaling circuit, which prescales by a factor of 16. What are the time intervals at which overflow interrupts occur from this timer? What will be period before which these interrupts must be serviced?
19. What do you mean by a software timer (SWT)? How do the SWTs help in scheduling multiple tasks in real time? Suppose three SWTs are programmed to timeout after 1024, 2048 and 4096 times from the overflow interrupts from the timer. What will be rate of timeout interrupts from each SWT?
20. What are the advantages and disadvantages of negative acknowledgement bit?
21. A new generation automobile has about 100 embedded systems. How do the bus arbitration bits, control bits for address and data length, data bits, CRC check bits, acknowledgement bits and ending bits in CAN bus help the networking of devices distributed in an automobile system.
22. How does the USB protocol provide for the device attachment, configuration, reset, reconfiguration, bandwidth sharing with other devices, device detachment (while others are in operation) and reattachment?
23. Design a table that compares the maximum operational speeds and bus lengths and give two example of the uses of each of the following serial devices: (a) UART (b) 1-wire CAN (c) Industrial I^2C (d) SM I^2C Bus (e) SPI of 68 Series Motorola Microcontrollers (f) Fault tolerant CAN (g) Standard Serial Port (h) FireWire (i) I^2C (j) High Speed CAN (k) IEEE 1284 (l) High Speed I^2C (m) USB 1.1 Low Speed Channel and High Speed Channel (n) SCSI parallel (o) Fast SCSI (p) Ultra SCSI-3 (q) FireWire/IEEE 1394 (r) High Speed USB 2.0.
24. Use web search. Design a table that compares the maximum operational speeds and bus lengths and give two example of the uses of each of the following parallel devices: (a) ISA (b) EISA (c) PCI (d) PCI-X (e) COMPACT PCI (f) GMII (Gigabit Ethernet MAC Interchange Interface) (g) XGMI (10 Gigabit Ethernet MAC Interchange Interface) (h) CSIX-1. 6.6 Gbps 32-bit HSTL with 200 MHz performance (i) RapidIO™ Interconnect Specification v1.1 at 8 Gbps with 500 MBps performance or 250 MHz dual direction registering performance using 8-bit LVDS (Low Voltage Data Bus).
25. Use web search and design a table that gives the features of the following latest generation serial buses. (a) IEEE 802.3-2000 [1 Gbps bandwidth Gigabit Ethernet MAC (Media Access Control)] for 125 MHz performance (b) IEE P802.3oe draft 4.1 [10 Gbps Ethernet MAC] for 156.25 MHz dual direction performance] (c) IEE P802.3oe draft 4.1 [12.5 Gbps Ethernet MAC] for four channel 3.125 Gbps per channel transreceiver performance] (d) XAUI (10 Gigabit Attachment Unit) (e) XSBI (10 Gigabit Serial Bus Interchange) (f) SONET OC-48, OC-192 and OC-768 (g) ATM OC-12/46/192.
26. Take a mobile smart phone with a T9 keypad. Write a table for the states of each key. Write another table for the new states generated by a combination of two keys.
27. Compare the parallel ports interfaces for the keypad, printer, LCD-controller and touchscreen.
28. Show the use of USB devices in the digital camera, printer and computer for downloading a picture from camera to computer, printing the pictures in camera and saving in flash memory. What is the difference between USB host and USB device in a system?
29. Compare different serial buses.
30. Compare different wireless protocols.

Device Drivers and Interrupts Service Mechanism

4

We have learnt the following in previous chapter:

- **Embedded system hardware has devices which communicate through serial and parallel ports and buses. There may also be ports for real-time voice and video I/Os.**
- **A microcontroller has serial communication and timing devices. It may have keypad, stepper motor, LCD and touch screen controllers.**
- **Serial or parallel buses interconnect the distributed ports and devices.**
- **I^2C bus is used for inter-IC communication. It interconnects multiple distributed ICs.**
- **CAN bus is used at control network of the distributed devices. It is used in automobiles and industrial systems.**
- **USB is used for the fast serial transmission and reception between the embedded-system and serial devices such as the keyboard, printer and scanner.**
- **FireWire (IEEE 1394) is bus used for the high-speed interfacing of 800 Mbps multimedia devices.**

- Parallel buses, ISA and PCI/PCI-X are used for bus communication of devices between the host computer or system and PC-based devices or systems or cards, for example, NIC (Network Interface Card).
- Wireless protocols are used for the communication and synchronization of distributed devices in wireless personal area network.
- Internet-enabled embedded systems can network to the Internet using the TCP/IP suite of protocols.

We also learnt

- A device-access is required for opening, connecting, binding, reading, writing, disconnecting or closing it. Processor accesses a device using the addresses of device registers and buffers. A processor accesses the internal devices, devices at the I/O ports, peripheral devices and other off-chip devices using the addresses.
- A simple device such as SPI port (Section 3.2.4) has addresses for three sets of its registers: data register(s) (or buffers), control register(s) and status register(s).
- A device can also have number of registers (Table 3.4). For example, PCI bus-driven device (Section 3.12.2) has 64 bytes standard device-independent configuration registers.

In this chapter, we will learn how the concept of interrupt service routines is used to address and service the device IOs, requests and interrupts. We will learn the following:

1. *Programmed I/O busy and wait method and problems with this I/O method.*
2. *Interrupts and working of interrupts service mechanism in the system and simple examples of hardware and software interrupts.*
3. *Interrupt service routine (ISRs) are called by the system when device-hardware interrupts take place.*
4. *Software functions for the signals and exceptions also call ISRs. An ISR is also called on a trap or execution of a software instruction for interrupt.*
5. *Use of interrupt vectors, vector table and masking.*
6. *Interrupt latency and deadline for an interrupt service.*
7. *Context and context switching on an interrupt.*
8. *New methods for the fast context switching adopted in the processors.*
9. *Classification of processors for an interrupt service that 'Save' or 'Don't Save' the context other than the program counter.*
10. *Use of the DMA channel for facilitating the small interrupt latency period for the multiple data transfers in quick succession.*
11. *Assignment of software and hardware priorities among the multiple sources of interrupts.*
12. *Methods of service in case of simultaneous service demand from multiple interrupting sources.*
13. *Device drivers for a device or port initialization and accesses.*
14. *Use of device drivers, for example, Linux Internals.*
15. *Examples of device initialization and device driver coding for the parallel ports and serial-line UART.*

4.1 PROGRAMMED-I/O BUSY-WAIT APPROACH WITHOUT INTERRUPT SERVICE MECHANISM

Example 4.1 shows an example of programming a device service with programmed I/O *busy-wait* approach without using a device interrupt and the corresponding ISR. This example will make clear the problems in this approach and advantages of using an interrupt-based service mechanism.

Example 4.1

Assume a 64 kbps network. Using a UART that transmits in the format of 11-bit per character, the network transmits at most 64 kbps ÷ 11 = 5818 characters per second, which means that for every 171.9 µs a character is expected. Before 171.9 µs, the receiver port must be checked to find and read another character assuming that all the received characters are in succession without any time gap.

Let port A be at an Ethernet interface card in a PC, and port B be its modem input which puts the characters on the telephone line. Let *In_A_Out_B* be a routine that receives an input character from port A and re-transmits an output character to port B. Assume that there is no interrupt generation and interrupt service (handling) mechanism. Let *In_A_Out_B* routine has to call the following steps *a* to *e* and executes the cycles of functions *i* to *v*, thus ensuring that the modem port A does not miss reading the character.

In_A_Out_B routine:

1. Call function *i*
2. Call function *ii*
3. Call function *iii*
4. Call function *iv*
5. Call function *v*
6. Loop back to step 1

In_A_Out_B routine calls the following steps.

Step *a*: Function *i*: Check for a character at port A. If not available, then wait.

Step *b*: Function *ii*: Read port A bytes (characters for message) and return to step *a* instruction, which will call function *iii*.

Step *c*: Function *iii*: Decrypt the message and return to step *a* instruction, which will call function *iv*.

Step *d*: Function *iv*: Encode the message and return to step *a* instruction, which will call function *v*.

Step *e*: Function *v*: Transmit the encoded message to port B and return to step *a* last instruction, which will start step *a* from the beginning.

Step *a* is also called polling. Polling a port means to find the status of the port, ready with a character (byte) at input. Polling must start before 171.9 μs because characters are expected at 64 kbps. If the program instructions in the steps *b*, *c*, *d* and *e* and functions *ii* to *v* take a total running time of less than 171.9 μs then this approach works.

Problems with the busy-wait programming approach is as follows.

1. The program must switch to execute the *In_A_Out_B* cycle of steps *a* to *e* within a period less than 171.9 μs. Programmer must ensure that steps of *In_A_Out_B* and any other device program steps never exceed this time.
2. When the characters are not received at Port A in regular succession, the waiting period during step *a* for polling the port can be very significantly. Wastage of processor time for the waiting periods is the most significant disadvantage of the busy-wait approach.
3. When other ports and devices are also present in the system, the programming problem is to poll each port and device and ensure that the program switches to execute the *In_A_Out_B* step *a* as well as switches to poll each port or device on time and then execute each service routines related to the functions of other ports and devices within a specific time interval and ensure that each one is polled on time.
4. The program and functions are processor- and device-specific in the previous busy-wait approach and all system functions must execute in synchronization and the timings are completely dependent on periods taken for software execution.

Instead of continuously checking for characters at the port A by executing function (*i*), when a modem receives an input character and sets a status bit in its status register, an interrupt from port A should be generated. In response to the interrupt an interrupt service routine ISR_ PortA _Character should then be executed (Example 4.2 in Section 4.2). This will be the efficient solution instead of wait at step *a*.

Device service without using an ISR is by the routine (function) call similar to *In_A_Out_B*.

Each routine (function) call has the following features.

1. A function call after executing any instruction in any program is a planned (user-programmed) diversion from the current sequence of instructions to another sequence of instructions and this sequence of instructions executes till the return from that.
2. On a function call, the instructions are executed as a function in the 'C' or a method in Java.
3. Function calls are nested. Nesting can be explained as follows: when a function 1 calls another function 2 which in turn calls another function 3, then on return from 3, the return is to function 2 and then to function 1.

Figure 4.1 shows the *In_A_Out_B* routine steps *a* to *e* for the five functions *i* to *v* called by *In_A_Out_B* and how each called function processes on a *call* and on a *return* from that. Numberings on the arrows show the sequences during the program run (flow).

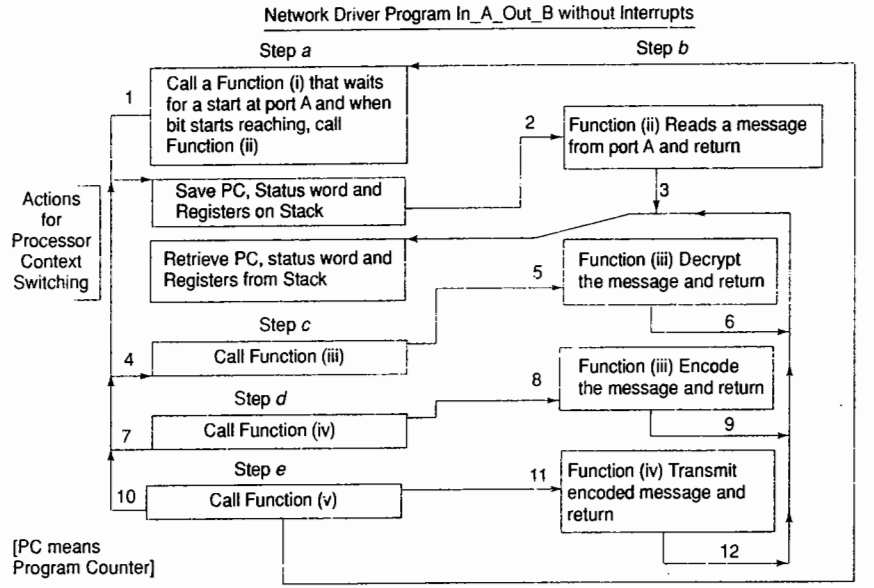


Fig. 4.1 Steps *a* to *e* for five function calls in an exemplary network drive program. *IN_A_OUT_B,* also shown is how each called function processes on a call and on a return. Numberings on the arrows show the program running sequences

One approach is 'programmed IO' transfer, also called 'busy and wait' transfer for service (accessing the device addresses for input or output or any other action). System functions in synchronization and the timings are completely software-dependent. When waiting periods are a significant fraction of the total program's execution period, wastage of the processor's time in waiting is the most significant disadvantage of this approach. Programmed IO approach can be used in single-purpose processors (controllers).

4.2 ISR CONCEPT

Interrupt means event, which invites the attention of processor for some action on the hardware or software event.

1. When a device or port is ready, a device or port generates an interrupt or when it completes the assigned action, it generates an interrupt. This interrupt is called hardware interrupt.
2. When software run-time exception condition is detected, either processor hardware or a software instruction generates an interrupt. This interrupt is called software interrupt or *trap* or *exception*.
3. Software can execute the software instruction for interrupt to *signal* the execution of ISR. The interrupt due to signal is also a software interrupt [The *signal* differs from the function in the sense that the execution of the signal handler function (ISR) can be masked and till the mask is reset, the handler will not execute. Function on the other hand always executes on the call after a call-instruction.]

In response to the interrupt, the routine or program, which is running at present gets interrupted and an ISR is executed. ISR is also called device driver ISR in the case of devices and is called *exception* or *signal* or *trap handler* in the case of software interrupts. Device driver ISRs execute on software interrupts from device open ( ), close ( ), read ( ), write ( ) or other device functions.

Examples in Sections 4.2.1 and 4.2.2 show the importance of interrupts and accessing of the devices using the ISRs and the importance of using the ISRs which generate on *traps* or *exceptions* or *signals*.

4.2.1 Examples of Port or Device Interrupts and ISRs

Following are the examples of interrupt events and accessing of devices using the ISRs.

Example 4.2

Recapitulate Example 4.1. Assume that a character input to the modem generates a port A interrupt and sets a status bit in the status register. On interrupt, a service routine ISR_PortA_Character runs so that it ensures that the modem port A does not miss reading the character. ISR_ PortA_Character executes step *f* in place of the step *a* function *i* and step *b* function *ii* of *In_A_Out_B* routine in Example 4.1. It places the read character in a memory buffer. Steps *c*, *d* and *e* are independent and are now parts of a function-call Out_B.

ISR_ PortA _Character executes as follows:

1. Step *f* function *vi*: *Read* Port A character. *Reset* the status bit so that the modem is ready for the next character input (resetting of the status bit is generally automatic without the need for specific instruction). *Put* it in a memory buffer. Memory buffer is a set of memory addresses where the bytes (characters) are queued for processing later.
2. Return from the ISR.

Out_B routine is as follows:

1. Step *g*: Call function *vi* to decrypt the message characters at the memory buffer and return for the next instruction step *h*.
2. Step *h*: Call function *vii* to encode the message character and return for the next instruction step *k*.
3. Step *k*: Call function *viii* to transmit the encoded character to port B.
4. Return from the function.

Figure 4.2 shows the step *f* executing on ISR_ PortA _Character on port A interrupt and steps *g* to *k* in Out_B routine. Numberings on the arrows show the program running sequences.

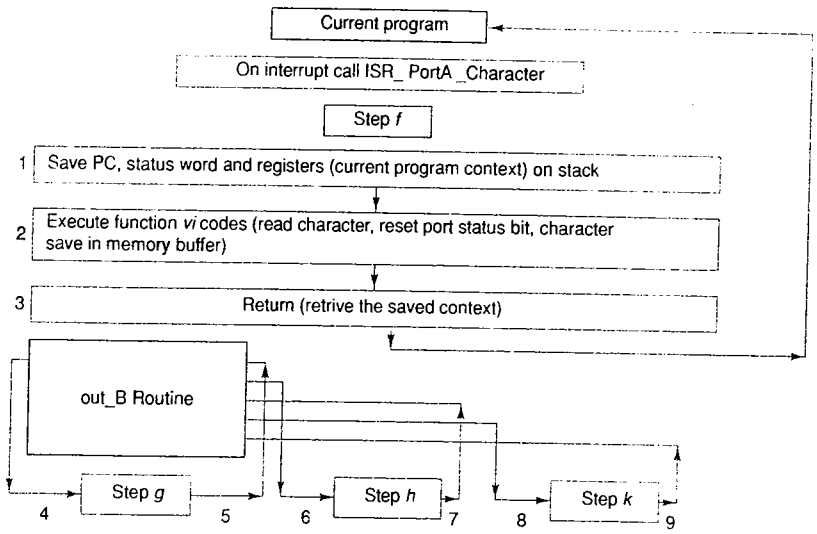
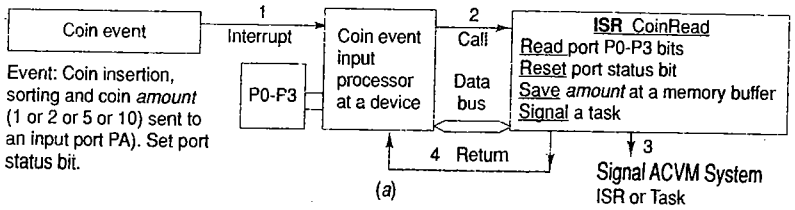


Fig. 4.2 Step *f* executing on ISR_ PortA _Character on port A interrupt and steps *g* to *k* in Out_B routine. Numberings on the arrows show the sequences of running the program-steps *f, g, h* and *k*

Example 4.3

Assume a device for coin amount input in an automatic chocolate-vending machine (Section 1.10.2). Without interrupt mechanism, one way is the busy wait transfer by which the device waits for the coin continuously, activates on sensing the coin and runs the service routine.

In the event-driven method the device should awaken and activate on each *interrupt* after sensing each coin-inserting event. The device is at an input port. It collects a coin inserted by a child. The system awakens and activates on interrupt through a hardware interrupt. The system on port hardware *interrupt* collects the coin by running a service routine. This routine is called interrupt handler routine or ISR or *device driver function* for the coin-port read. Figure 4.3(a) shows the ISR in the ACVM example.



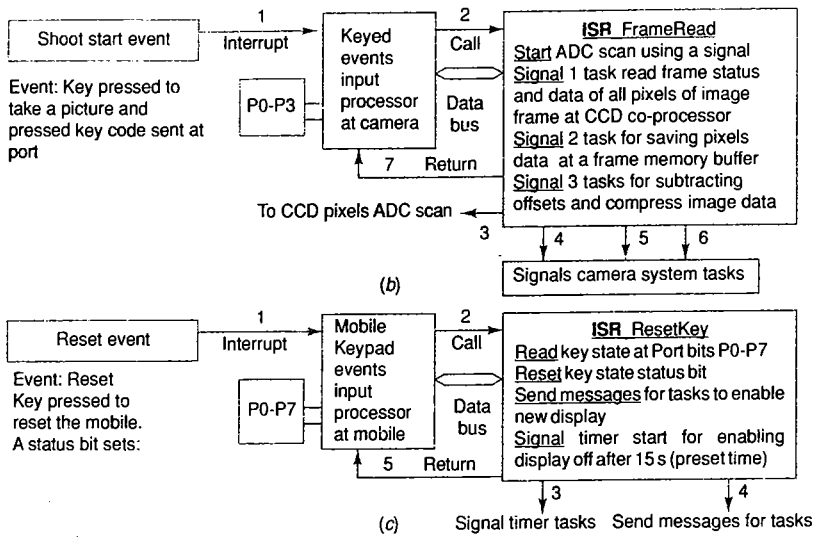


Fig. 4.3 (*a*) Use of ISR in the automatic chocolate vending machine (*b*) Use of ISR and three signals 1, 2 and 3 for three tasks in the digital camera example (*c*) Use of ISR in the mobile phone reset-key interrupt example

Example 4.4

Assume a digital camera system (Section 1.10.4). It has an image input device. When the system activates the device should grab image-frame data. The system awakens and activates on a switch *interrupt*. The interrupt is through a hardware signal from the device switch. On the *interrupt*, an ISR (can also be considered as camera's imaging device-driver function) for the *read* starts execution, it passes a message (signal) 1 to a function or program thread or task, which senses the image and then the function reads the CCD device frame buffer; then the routine passes signal 2 to another function or program thread or task to process and then signal 3. Subtracts offsets using a task and compresses image-data using a task. This task also saves the image frame data-compressed file in a flash memory. The camera system again awakens and activates on *interrupt* through a hardware signal from a device switch and prints the file picture image after file decompression. The system on *interrupt* then runs another ISR. The ISR routine is the device-driver write-function for the outputs to printer through the USB bus connected to the printer. Figure 4.3(b) shows the use of the ISR for frame read in the digital camera example.

ISR accesses a device for service (configuring, initializing, activating, opening, attaching, reading, writing, resetting, deactivating or closing). ISRs thus function as the device drivers.

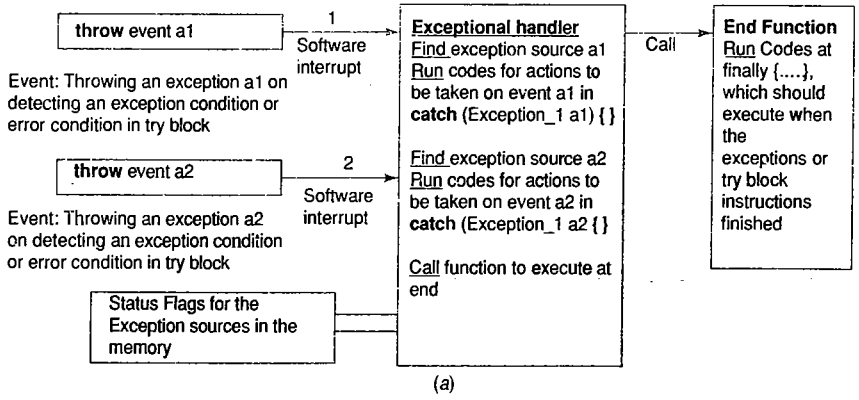
Example 4.5

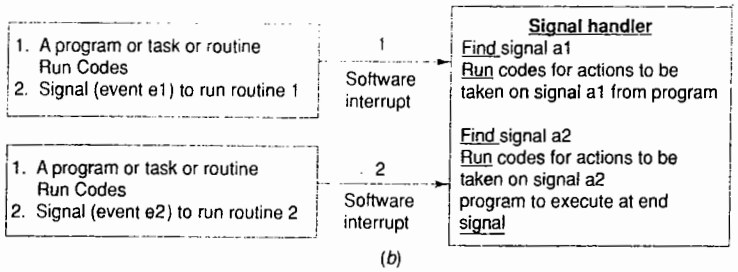
Assume a mobile phone system (Section 1.10.5). It has a system reset key, which when pressed resets the system to an initial state. When reset key is pressed the system awakens and activates a *reset interrupt* through a hardware signal from reset key circuit. On the *interrupt*, an ISR (can also be considered as reset-key device-driver function) suspends all activity of the system, sends messages to the display functions for the program threads or the tasks for displaying the initial reset state menu and graphics on LCD screen, and also signals to activate LCD display-off timer device for 15s timeout (for example). After the timeout the system again awakens and activates on *interrupt* through the internal hardware signal from timer device and runs another ISR to send a control bit to the LCD device. The ISR routine is the device-driver LCD off-function for the LCD device. The devices switch off by reset of control bit. Figure 4.3(c) shows the use of the ISR in the mobile-phone reset-key interrupt.

4.2.2 Examples of Software Interrupts and ISRs

Examples 4.2 to 4.5 clearly show that interrupts and ISRs (device-drivers) play the major role in using the system hardware and devices. Think of any system hardware and it will have devices and thus needs device drivers. The embedded software or the operating system for application software must consist of the codes for the device (i) configuring (initializing), (ii) activating (also called opening or attaching), (iii) driving function for read, (iv) driving function for write and (iv) resetting (also called deactivating or closing or detaching). Each device task is completed by first using an ISR—a device-driver function calls the ISR by using a software interrupt instruction (SWI).

A program must detect error condition or run-time exceptional condition encountered during the running. In a program either the hardware detects this condition (called trap) or an instruction SWI is used that executes on detecting the exceptional run-time condition during computations or communication. For example, detecting that the square root of a negative number is being calculated or detecting the illegal argument in the function or detecting that the *connection* to network is not found. Detection of exceptional run-time condition is called *throwing* an exception by the program. An interrupt service routine (exceptional handler routine) executes, which is called *catch* function as it executes on catching the exception thrown by executing an SWI. Figure 4.4(a) shows use of SWI instruction for calling an ISR in the function for throwing and catching the exceptional run-time conditions encountered during computations. Figure 4.4(b) shows the use of signal generated by SWI, and signal handling after that.





(b)

Fig. 4.4 (*a*) Use of software interrupt (SWI) instruction for calling an interrupt service routines (ISR) in the software on throwing and catching the exceptional run-time conditions encountered during computations (*b*) Use of SWIs to signal another routines or program tasks or program threads to start

Example 4.6 is given here to clearly show that SWI and execution of the ISRs (also called *exception handlers* or just *exceptions*) on SWIs. The SWIs also play a major role in embedded system software by the use of ISRs for device driver functions—create ( ), open ( ), read ( ) or other.

Example 4.6

Consider the following codes.

```
try {
/* Codes for execution in which a run-time exception or number of run-
    time exception conditions may encounter, for example square root
    of a negative number or a percentage value exceeding 100% or decreasing
    below 0%. The condition is trapped and on trapping throws an
    exception*/
.
If ((A - B) < 0.1) throw a1; x = y + sqrt (A - B); /* Find if A - B is
    -ve number. If yes, throw an exception a1 and call a catch
    function). */
.
y = ..../ * Calculate y */
If ((y > 100 | | y < 0) throw a2; /* Find if y > 100. If yes, throw
    exception a2 and call a catch function*/
.
}
```

```
catch (Exception_1 a1) {
 /* Code for action on throwing (trapping) A - B < 0.0 exception */
.
.
}
```

```
catch (Exception_1 a2
 /* Code for action on throwing (trapping) y < 0 or y > 100
    exception*/
.
.
}
```

```
finally {
/* Final codes, which should execute on exception or after
    try block instructions over */
.
.
}
```


High-level Java or C++ codes when compiled, during compilation the SWI instructions will be inserted for trapping (A – B) as a negative number and for trapping y > 100 or less than 0 as follows:

1. Software instruction SWI a1 will cause processor interrupt. In response, the software ISR function 'catch (Exception_1 a1) { }' executes on throwing of the exception a1 at try block execution. SWI a1 is used after catching the exception a1 whenever it is thrown.
2. Software instruction SWI a2 will cause processor interrupt. In response, the software ISR function 'catch (Exception_2 a2) { }' executes on throwing of the exception a2 during try block execution. SWI a2 is used after catching the exception a2 whenever it is thrown.
3. Software instruction SWI a3 will cause processor interrupt and in response will *signal* software ISR function 'finally { }' to execute either at the end of the try or at the end of the catch function codes. SWI a3 is used after the try and catch functions finish, then *finally* function will perform final task, for example, exit from the program or call another function.

A user program under execution currently by the processor does not know when its try function will throw the exceptions a1 or a2 or when the signal handler throws a3.

An ISR call has the following features.

1. An ISR call due to interrupt executes an event. Event can occur at any moment and event occurrences are asynchronous.
2. ISR call is event-based diversion from the current sequence of instructions (routine or program) to another sequence of instructions (routine or program). This sequence of instructions executes till the return instruction.
3. Event can be a device or port event or software computational exceptional condition detected by hardware or detected by a program, which throws the exception. An event can be signalled by software interrupt instruction SWI used in device driving functions create ( ), open ( ), etc.
4. An interrupt service mechanism exists in a system to call the ISRs from multiple sources (Section 4.4).

5. Diversion to ISR may or may not take place on finishing the execution of any instruction in the presently running routine. The execution of the ISRs can be masked by an instruction to set a mask bit and can be unmasked by another instruction to reset the mask bit. [Except a few interrupt sources called non-maskable source (Section 4.4.3).] An instruction in a function or program thread or task can disable or enable an ISR call or all ISR calls (Section 4.4.3).
6. On an interrupt call, the instructions do not execute continuously exactly like a C function or a Java method. These execute as per the interrupt mechanism of the system. For example, *'return'* from an ISR differs in certain important aspects. An interrupt mechanism may be such that an ISR on beginning the execution may disable automatically other device(s) interrupt services. These are automatically re-enabled if they were enabled before a service call. Another interrupt mechanism may be such that an ISR on beginning the execution does not disable automatically other device(s) interrupt services and there can be in-between diversion in the case of the unmasked higher priority interrupts (Section 4.5.1).
7. There can be multiple interrupt calls during running of an ISR for diversion to other ISRs. The ISR calls need not be the nesting of the ISRs unlike the case of the function calls and there is diversion to pending higher priority interrupt either at the end or in-between the interrupted ISR.

Section 7.6 will explain the distinction between functions, ISRs and tasks by their characteristics.

Interrupt is an event from a device or hardware action or software instruction. In response to the interrupt, a presently running program is interrupted and a service routine executes. The routine is called ISR. It is also called *device driver* in case of interrupts from the devices. It is also called *exception* handler in case of interrupts from the software. ISR-based approach facilitates an efficient synchronization of the function-calls and ISR-calls. The timings when an ISR executes are hardware or software interrupt event dependent. There is therefore no waiting period due to no need of device polling.

4.2.3 Interrupt Service Threads as Second-Level Interrupt Handlers

An ISR can be executed in two parts.

1. One part is the short execution time taking service routine and can be called as first-level ISR (FLISR). It runs the critical part of the ISR and execute a *signal* function to enable the OS to schedule for running the remaining part later. It can also send a message using a function to enable the OS to iniate a task later on after return from the ISR. The task waits during execution of interrupts routines and signal functions. The FLISR does the device-dependent handling only. For example, it does not perform decryption of data received from the network. It simply does the transfer of data to the memory buffer for the device data.
2. The second part is the long service routine called interrupt service thread (IST) or second-level ISR (SLISR), which executes on the *signal* of the first part. The OS schedules the IST as per its priority. IST does the device-independnet handling. IST is also the software interrupt thread as it is triggered by an SWI (software interrupt instruction) for the *signal* in FLISR.

Figure 4.3(b) showed used of *signal* in ISR-FrameRead in digital camera system. Figure 4.5 shows how ADC scan is initiated by an SLISR call from FLISR. Figure 4.5 shows the FLISR and second-level IST approach to handle the device hardware interrupts followed by software interrupts in upper part and the use of this approach in a camera in lower part.

Interrupt service can be done in two parts: a hardware device-dependent code in the FLISR, which has a short execution time and a software interrupt initiated SLISR, which is also called IST. A task can also be sent message by FLISR. The task runs after the IST.

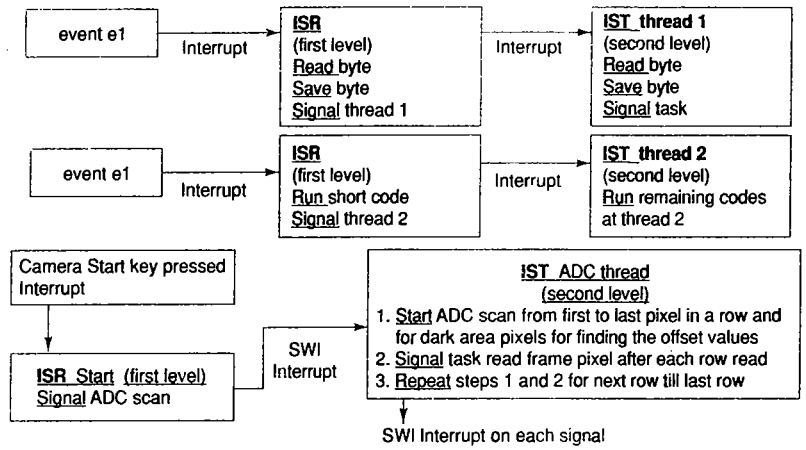


Fig. 4.5 First-level interrupt service routine and second-level interrupt service thread approach to handle the device hardware interrupt followed by the software interrupt to call SLISR–an IST and the use of this approach in a camera

4.2.4 Device Driver

Each device in a system needs device driver routines. An ISR relates to a device driver function. A device driver is a function used by a high-level language programmer and does the interaction with the device hardware and communicates data to the device, sends control commands to the device and runs the codes for reading the device data. A programmer uses generic commands for the device driver for using a device. The OS provides these generic commands.

The examples of generic functions used for the commands to the device are device *create* ( ), *open* ( ), *connect* ( ), *bind* ( ), *read* ( ), *write* ( ), *ioctl* ( ) [for IO control], *delete* ( ) and *close* ( ). Device driver code is different in different OS. Same device may have different codes for the driver when the system is using different OS.

A device driver function uses SWI, which initiates the interrupt service. The device uses the system and IO buses required for the device service. Device driver can be considered as a function in software layer of an application program and the device.

For example, the application program sends the commands to write on display screen of a mobile the *contact names* from the contact database. LCD display device driver calls an SWI, and an ISR does that without the application programmer knowing how does LCD device interface in the system, what are the addresses which are used, what and where and how are the control (command) and status registers used.

For a programmer, using the device driver's generic functions for reading or writing from and to the device is analogous to reading or writing any other device or data file except that the device and file have different device identity numbers.

The driver routine controls a device without requiring understanding of the device configuration, control, status, data and other registers, when using the generic functions. Device driver runs the ISRs of the device. Each ISR is the low-level part of the device driver generic function, which executes on software interrupt instruction.

The driver translates a program generic function for using the device and sends the necessary commands to the device configuration and control registers. The driver uses the device control, status and data registers.

The driver does the opening, configuring, initializing, attaching, reading, writing, closing and detaching the device by initiating the corresponding ISRs.

Drivers of many devices, such as printers, touch screen, LCD display, keypad, keyboard, are part of the OS. Section 4.9 will describe device drivers in detail.

Each device high-level language program in a system uses device driver functions. A programmer uses generic commands, create ( ), open ( ), connect ( ), bind ( ), read ( ), write ( ), ioctl ( ), delete ( ) and close ( ) and uses for each device a device identity number. Device driver executes the SWIs, which call the ISRs for using the device hardware and memory allotted to that. SWIs are dedicated for the device service and perform all the necessary actions.

4.3 INTERRUPT SOURCES

Hardware sources can be from internal devices or external peripherals, which interrupt the ongoing routine and thereby cause diversion to corresponding ISR. Software sources for interrupt are related to (*i*) processor detecting (trapping) computational error for an illegal op-code during execution or (*ii*) execution of an SWI instruction to cause processor-interrupt of ongoing routine.

Each of the interrupt sources (when not masked) (or groups of interrupt sources) demands a temporary transfer of control from the presently executed routine to the ISR corresponding to the source.

The internal sources and devices differ in different processors or microcontrollers or devices and their versions and families. Table 4.1 gives a classification as hardware and software interrupts from several sources. Not all the given types of sources in the table may be present or enabled in a given system. Further, there may be some other special types of sources provided in the system.

Hardware Interrupts Related to Internal Devices There are number of hardware interrupt sources which can interrupt an ongoing program. These are processor or microcontroller or internal device hardware specific. An example of a hardware-related interrupt is timer overflow interrupt generated by the microcontroller hardware. Row 1 of Table 4.1 lists common internal devices interrupt sources.

Hardware Interrupts Related to External Devices – 1 There can be external hardware interrupt source for interrupting an ongoing program that also provides the ISR address or vector address (Section 4.4.1) or interrupt-type information through the data bus. Row 2 of Table 4.1 lists these interrupt sources. External hardware interrupts with ISR addresses information sent by the devices themselves (Section 4.4.1) and are device hardware-specific.

Example 4.7

An example of external hardware-related interrupt with device sending the interrupt on INTR pin is the 80x86 processor. When INTR pin activates on an interrupt from the external device, the processor issues two cycles of acknowledgements in two clock cycles through the INTA (interrupt acknowledgement) pin. During the second cycle of acknowledgement, the external device sends the type of interrupt information on data bus. Information is for one byte n. 80x86 internally signals instruction INT n, which means that it executes interrupt of type n, where n can be between 0 and 255. INT n causes the processor vectoring to address 0x00004 × n. [SWI in 80x86 is denoted by INT.]

Hardware Interrupts Related to External Devices – 2 External hardware interrupts with their ISR vector addresses (Section 4.4.1) are processor or microcontroller-specific interrupts of an ongoing program. External interrupting source does not send interrupt-type or ISR address-related information. An example of external hardware-related interrupt in which the interrupt-type information internally generates is an interrupt on NMI (non-maskable interrupt) pin in the 80x86 processor. Row 3 of Table 4.1 lists these interrupt sources.

Table 4.1 Classification and Sources of Interrupts[1]

Sources	*Examples*
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 [e.g. TDRE (transmitter data register Empty); 5. Synchronous transmission of byte completed; 6. ADC start of conversion; 7. ADC end of conversion; 8. Pulse-accumulator overflow; 9. Real-time clock time-outs (Section 3.8); 10. Watchdog timer resets (Section 3.7); 11. Timers overflows on time-out (Section 3.6); 12. Timer comparison with output compare register; 13. Timer capture on input (Section 3.6)
External hardware devices providing the ISR address or vector address or type externally[2]	INTR in 8086 and 80x86
External hardware devices with internal vector address generation	1. Non-maskable pin [NMI in 8086 and 80x86]; 2. Within first few clock cycles unmaskable declarable pin (interrupt request pin) but otherwise maskable XIRQ [in 68HC11]; 3. Maskable pin (interrupt request pin) [INT0 and INT 1 in 8051, IRQ in 68HC11]
Software error-related sources (exceptions[3] *or SW-traps)*	1. Division by zero detection (or *trap*) by hardware; 2. Over-flow by hardware; 3. Under-flow by hardware; 4. Illegal opcode by hardware
Software instruction-related sources (exceptions[4] *or SW-traps SW-signal)*	Programmer-defined exceptions[3] or traps for handling exceptional run-time conditions or programmer-defined *signal* for executing ISR to handle further actions or *signals* from device driver functions

[1] Processor-specific examples are in bracket.
[2] Example 4.7 explains this.
[3] The processor internally generates a trap or exception. An example is *division by zero* in 80x86. Example 4.8 explains this.
[4] The second type of exception is the user program-defined exception. Example 4.6 explained this. *Signal* is a term sometimes used in high level program for software interrupt instruction in assembly language. For example, in VxWorks RTOS. [Refer Section 9.3.] *Signal* or *exception* is an interrupt on the setting of certain conditions or on obtaining certain results or output during a program run or a signal for some action. The condition examples are square root of a negative number or percentage computation resulting in values greater than 100% or an IO connection not found.

Software Error-Related Hardware interrupts There can be the software-error related interrupts generated by processor hardware. Each processor has a specific instruction set. It is designed for that set only. An illegal code (instruction in the software) is an instruction, which does not correspond

to any instruction in this set. Whenever the processor fetches illegal code, an interrupt occurs in certain processors. The error-related interrupts are also called hardware-generated *software traps* (or *software exceptions*). A software error called *trap* or *exception* may generate in the processor hardware for an illegal or not-implemented opcode found during execution. The examples are as follows: (i) There is an *illegal opcode trap* in 68HC11. This error causes an interrupt to a vector address (Section 4.4.1). (ii) Non-implementable opcode error causes an interrupt to a vector address in 80196.

Software error *exception* or *trap*-related sources cause the interrupt of an ongoing program computations in certain processors. Examples are the division by zero (also known as type 0 interrupt as it is also generated by a software interrupt instruction *INT 0* in 80x86) and overflow (also known as type 2 interrupt as it is also generated by Int 2 instruction) in 80x86. These two interrupts, types 0 and 2 are generated by the hardware within the ALU of the processor. Row 4 of Table 4.1 lists these interrupt sources. Example 4.8 explains a software-related *trap* or *exception*, which is an interrupt generated by the processor hardware on division by 0.

Example 4.8

Assume that a division by zero occurs during execution of a certain instruction of a program. An ISR is needed which must execute whenever the division by zero occurs. This ISR could be to display 'A division by zero error at .....' on the screen and then terminate or pause the ongoing program.

A user program under execution currently by the processor does not know when its ALU will issue this internal error flag (a hardware signal). The service routine executes by using an interrupt mechanism which is meant for service on a zero-division error-signal. On setting of the signal, an interrupt of the ongoing program happens just after completing the current instruction that is being executed, and then the ISR executes for postzero division tasks after resetting the flag.

Executing software error-related processor interrupts are needed to respond to errors such as division by zero or illegal opcode, which is detected by the processor hardware. These are called traps and some time also called exceptions. These are essential for handling run-time errors detected by the system hardware.

Software Instruction-Related Interrupts Sources A program can also *handle* specific computational errors or run-time conditions or signalling some condition. For instance, Example 4.6 showed the handling of negative number square root SWI, which is handled by SWI instruction in the instruction set of a processor. Processors provide for software instruction(s) related to the traps, signals or exceptions.

1. There are certain software instructions for interrupting and then diverting to the ISR also called the signal handler. These are used for signalling (or switching) to another routine from an ongoing routine or task or thread (Section 7.10). Figure 4.4(b) showed the signal generated by SWI and signal handling.
2. Software instructions are also used for trapping some run-time error conditions (called throwing exceptions) and executing exceptional handlers on catching the exceptions (Example 4.6).

An example of a software interrupt is the interrupt generated by a software instruction INT n in the 80x86 processor or SWI in ARM7. Row 5 of Table 4.1 lists these interrupt sources. SWI instruction differs from a function *call* instruction as follows.

1. Software interrupt in 68HC11 is caused by instruction, SWI.
2. There is a single byte instruction INT0 in 80x86. It generates type 0 interrupt, which means that the interrupt should be generated with the corresponding vector address 0x00000. Instead of the type 0 interrupt that 8086 and 80x86 hardware may also generate on a division by zero, the instruction INT0 does exactly that.

3. There is another single byte 8086 and 80x86 instruction TYPE3 (corresponding vector address $0 \times 00C0H$). This generates an interrupt of type 3, called break point interrupt. This instruction is like a PAUSE instruction. PAUSE is a temporary stoppage of a running program. It enables a program to do some housekeeping, and return to the instruction after the break point by pressing any key.
4. There are another 80x86 two-byte instruction INT n, where n represents the type and is the second byte. This means 'generate type n interrupt' and processor hardware get the ISR address by computing by the vector address $0x00004 \times n$. When n = 1, it represents single-step trap in 8086 and 80x86.
5. There is another 80x86 instruction, which uses a flag called trap and is denoted by TF. This flag is at the FLAG register and EFLAG register of 8086 and 80x86, respectively. This means when TF sets (written '1'), automatically after every instruction, the processor action causes an interrupt of type 1 repeatedly. The processor fetches each time the ISR address from the vector address 0x00004 (same as type 1 interrupt address). INT 1 software instruction will also cause type 1 interrupt once but the TF flag set instruction action is identical to the action caused at the end of each instruction after type 1 interrupt.
6. There is instruction in 80196 called *Trap*. It enables debugging of instructions. Till the next instruction after the Trap is executed, no interrupt source can interrupt the process and cause diversion to ISR.

SWI-related details in the instruction set help in programming the program diversion to ISR on exception. The exceptional condition if occurs (sets) during execution, causes a diversion to the ISR called *exception handler* or *signal handler* using the software instruction for interrupt in the set.

A programmer can program for the exception on a *queue* (a memory buffer similar to a print buffer) getting full. This is an exceptional run-time condition. It should cause the diversion to routine called *exception* handler function that initiates the appropriate action. *Exceptions* are important routines for handling the run-time errors.

Software instruction-related or software-defined condition-related software interrupts are used in the embedded system. They are essential to design ISRs like error-handling ISRs, software timer-driving ISRs and signalling another routines to run. These interrupts are also called *traps* or *exceptions* or *signals*.

4.4 INTERRUPT SERVICING (HANDLING) MECHANISM

Each system has an interrupt servicing (handling) mechanism. The OS also provides for mechanism for interrupt-handling (Section 8.7).

4.4.1 Interrupt Vector

Interrupt vector is a memory address to which the processor vectors. The processor transfers the program counter to the interrupt vector new address on an interrupt. Using this address, the processor services that interrupt by executing corresponding ISR. The memory addresses for vectoring by the processor are processor- or microcontroller-specific. Vectoring is as per the provisions in interrupt-handling mechanism. The various mechanisms are as follows:

Processor Vectoring to the ISR_ VECTADDR On an interrupt, a processor vectors to a new address, ISR_VECTADDR. It means that the PC (program counter), which has the instruction address of next instruction, saves that address on stack or in some CPU register, called link register and the processor loads the ISR_VECTADDR into the PC. The stack pointer register of CPU provides the saved address to enable return from the ISR using the stack. When the PC saves at the link register it is part of the CPU register set. Section 4.6 will explain the mechanism for saving the CPU registers in detail. The ISR last instruction is RETI (return from interrupt) instruction.

A processor provides for one of the following ways of using the ISR_VECTADDR-based addressing mechanism.

Processor Vector Address

1. A system has internal devices like the on-chip timer and A/D converter. In a given microcontroller, each internal device interrupt source or source-group has a separate ISR_VECTADDR address. Each external interrupt pin has separate ISR_VECTADDR. An example is 8051. Figure 4.6(a) shows the ISR_VECTADDRs for the hardware interrupt sources. A very commonly used method is that the internal device (interrupt source or interrupt source group) in the microcontroller autogenerates the corresponding interrupt vector address, ISR_VECTADDR. Thus vector addresses are specific for specific microcontroller or processor with that internal device. An internal hardware signal from the device is sent for the interrupt source or source group.
2. In 80x86 processor architecture, a software instruction, for example, *INT* n explicitly also defines the *type* of interrupt and the *type* defines the ISR_VECTADDR. Figure 4.6(b) shows the ISR_VECTADDRs with different vector addresses for different interrupt types. This mechanism results in the handling of n number of exception handling routines or ISRs for n interrupt types.

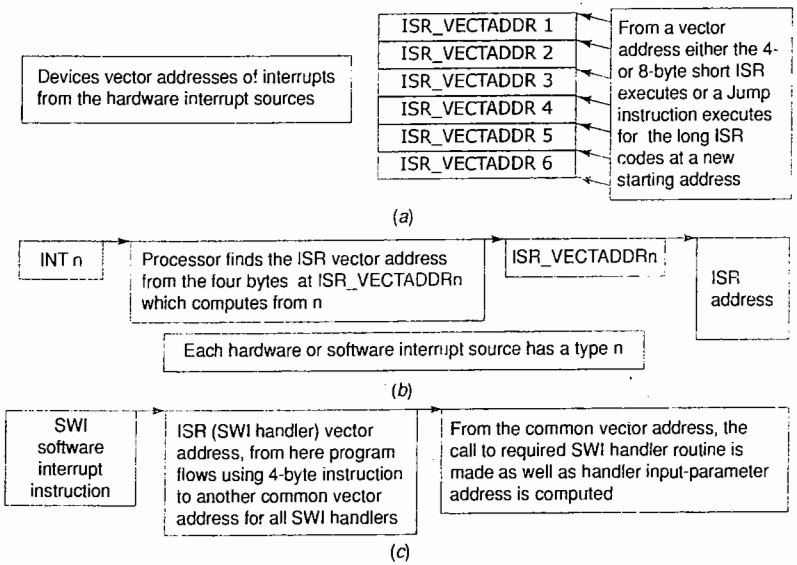


Fig. 4.6 (*a*) ISR_VECTADDRs for hardware interrupt sources (*b*) ISR_VECTADDRs with different vector address for different interrupt types using INT n instruction (*c*) The ISR_VECTADDR with common vector addresses for different exceptions, traps and signals using software interrupt instruction SWI

3. In ARM processor architecture, the software instruction SWI does not explicitly define the type of interrupt for generating different vector address and instead there is a common ISR_VECTADDR for each *exception* or *signal* or *trap* generated using SWI instruction. ISR that executes after vectoring has to find out which exception caused the processor to interrupt and divert the program. Such a mechanism in the processor architecture results in provisioning for the unlimited number of exception handling routines in the system all having the common interrupt vector address. Figure 4.6(c) shows the ISR_VECTADDR with common vector address for all *exceptions, traps* and *signals* resulting from SWI.

A group of Interrupt Sources having Common Vector Address A source group in the hardware may have the same ISR_VECTADDR.

Example 4.9

Consider 8051, TI (transmitter interrupt) and RI (receiver interrupt) are the sources in the same group having identical ISR_VECTADDR. TI is an interrupt that is generated when the serial buffer register for transmission completes serial transmission, and RI is when the buffer receives a byte from the serial receiver. ISR at the ISR_ADDR to which the program jumps or which is called from bytes at the ISR_ VECTADDR must first identify the interrupt source (whether TI or RI) in case of the identical vector address or ISR address for a group of sources. Identification is from a flag in the status register. Setting of a specific status flag in the device flag register enables identification of the interrupt source in the group by the ISR that runs after vectoring.

There are two types of handling mechanisms in processor hardware. The processor-handling mechanism provides for fetching into the PC either (i) the ISR instruction at the ISR_VECTADDR or (ii) the ISR address from the bytes at the ISR_VECTADDR.

1. There are some processors, which use ISR_VECTADDR directly as ISR address and the processor fetches the ISR instruction from there, for example, ARM or 8051. The ARM permits the use of 4-byte instruction for the jump to the ISR routine for the interrupt servicing). Figure 4.7(a) shows the use of ISR_ VECTADDR in ARM for the jump to the routine for the interrupt servicing. The 8051 microcontroller permits the use of short ISR of maximum 8 bytes for the internal devices. The short ISR codes can also use a call instruction to call a detailed routine. Figure 4.7(b) and (c) shows the use of ISR_ VECTADDR in 8051 in case of short-code and long-code ISR, respectively.
2. There are some processors, which use ISR_VECTADDR indirectly as ISR address and the processor fetches the ISR address from the bytes saved at the ISR_VECTADDR, for example, 80x86. Figure 4.7(d) shows the use of ISR_ VECTADDR address in 8086. Processor of interrupt of type n vectors to address 0x00004 × n and fetches 16 bits for sending into IP (instruction pointer register) and another 16 bits for sending into CS (code segment register) The ISR for interrupt will execute from address 0x100000 × CS + IP.

Interrupt Vector Table System software designer must provide for specifying the bytes at each ISR_VECTADDR address. The bytes are for either ISR short code [Figure 4.7(b)] or jump instruction to the ISR first instruction [Figure 4.7(a)] or ISR short code with call to the full code of the ISR [Figure 4.7(*c*)] or for fetching the bytes for finding the ISR address [Figure 4.7(d)].

A table facilitates the service of the multiple interrupting sources for each internal device. Each row of table has an ISR_VECTADDR and the bytes are saved at each ISR_VECTADDR. Vector table location in the memory depends on the processor. It is located at the higher memory addresses, 0xFFC0 to 0xFFFB in 68HC11. It is at the lowest memory addresses 0x00000 to 0x003FF in 80x86 processor. It starts from the lowest memory addresses 0x00000000 in ARM7. Figure 4.8 shows a vector table in the memory in case of multiple interrupt sources or source groups.

An external device may also send to the processor the ISR_VECTADDR through the data bus (row 2, Table 4.1).

An interrupt vector is an important part of interrupts service mechanism, which associates a processor. The processor first saves the PC and/or other registers of CPU on interrupt and then loads a vector address into the PC. Vector address provides the ISR or ISR address to the processor for an interrupt source or a group of sources or for the given interrupt type. The interrupt vector table is an important part of interrupts service mechanism, which associates the system provisioning for the multiple interrupt sources and source groups.

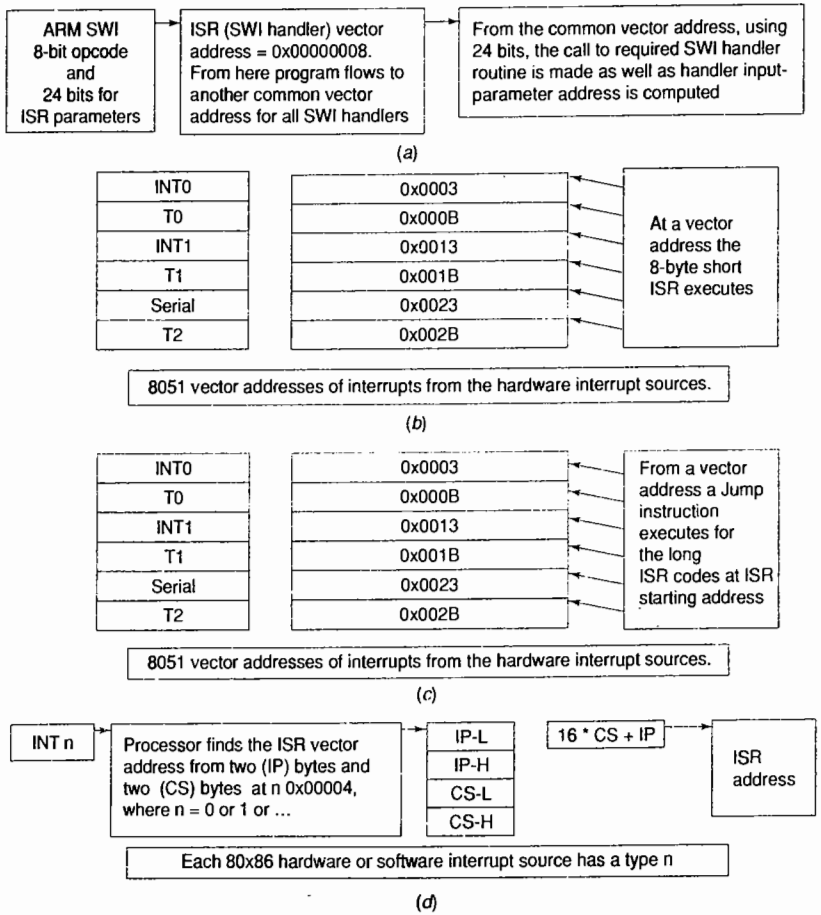


Fig. 4.7 (*a*) Use of ISR_ VECTADDR in ARM for the jump to the routine for the interrupt servicing (*b*) Use of ISR_ VECTADDR in 8051 in case of short-code interrupt service routine (ISR) (*c*) Use of ISR_ VECTADDR in 8051 in case of long-code ISR (*d*) Use of ISR_ VECTADDR address in 80x86 processors

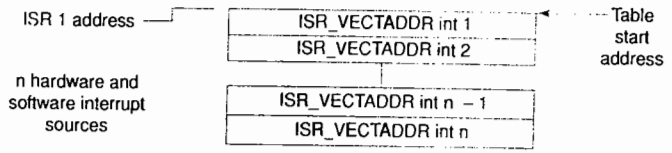


Fig. 4.8 Vector table in memory in case of multiple interrupt sources or source groups

4.4.2 Classification of All Interrupts as Non-Maskable and Maskable Interrupts

Maskable sources of interrupts provide for masking (no diversion) and unmasking the interrupt services (diversion to the ISRs). Execution of ISR for each device interrupt source or source group can be masked or unmasked. An external interrupt request can also be masked. Execution of a software interrupt (trap or exception or signal) can also be masked. Most interrupt sources are maskable. A few specific interrupts cannot be masked. A few specific interrupts can be declared non-maskable within few clock cycles of the processor reset, else that is maskable. There are three types of interrupt sources in a system.

1. *Non-maskable*: Examples are RAM parity error in a PC and error interrupts like division by zero. These must be serviced.
2. *Maskable*: Maskable interrupts are those for which the service may be temporarily disabled to let higher priority ISRs be executed first uninterruptedly.
3. *Non-maskable only when defined so within few clock cycles after reset*: Certain processors like 68HC11 has this provision. For example, an external interrupt pin, XIRQ interrupt, in 68HC11. XIRQ interrupt is non-maskable only when defined so within few clock cycles after 68HC11 is reset.

4.4.3 Enabling (Unmasking) and Disabling (Masking) in Case of Maskable Interrupt Sources

There can be interrupt control bits in devices. There may be one bit EA (enable all), also called the primary-level bit for enabling or disabling the complete interrupt system. When a routine or ISR is executed by the codes in a critical section, an instruction DI (disable interrupts) is executed at the beginning of the critical section and another instruction EI (enable interrupts) is executed at the end of the critical section. DI instruction resets the EA (enable all) bit and EI instruction sets primary level bit denoted by EA (enable all). An example of a critical section code is as follows. Assume that an ISR transfers data to the printer buffer, which is common to the multiple ISRs and functions. No other ISR of the function should transfer the data to the print buffer, else the bytes at the buffer will be from multiple sources. Data shared by several ISRs and routines need to be generated or used by protecting its modification by another ISR or routine.

There may be multiple bits denoted by E_0, ... E_{n-1} for n source group of interrupts in case of multiple devices. These bits are called mask bits and are also called secondary-level bits for enabling or disabling specific sources or source-groups in the system. By appropriate instructions in the user software, a write to the primary enable bit and secondary level enable bits (or the opposite of it, mask bits) either all or a part of the total maskable interrupt sources are disabled.

Example 4.10

Consider a system in which there are two timers and each timer has an interrupt control bit. Timer interrupt control bits are ET0 and ET1. Consider a system in which there is an SI device and an interrupt control bit ES, common to serial transmission and serial reception. There is an EA bit to interrupt control for disabling all interrupts.

When EA = 0, no interrupt is recognized and timers as well as SI interrupts service is disabled. When EA = 1, ET0 = 0, ET1 = 1 and ES = 1, interrupts from timer 1 and SI are enabled and timer 0 interrupt is disabled (masked).

4.4.4 Status Register or Interrupt Pending Register

An identification of a previously occurred interrupt from a source is performed by one of the following:

1. A local-level flag (bit) in a status register, which can hold one or more status flags for the one or several of the interrupt sources or groups of sources.
2. A processor-interrupt service pending flag (boolean variable) in an interrupt-pending register (IPR), that sets by the source (setting by hardware) and auto-resets immediately by the internal hardware when at a later instant, the corresponding source service starts diversion to the corresponding ISR.

Example 4.11

Consider a system in which there are two timers and each timer has a status bit TF0 and TF1. Consider a system in which there is SI device there are the status bits TxEMPTY and RxReady for serial transmission completed and receiver data ready.

1. The ISR_T1 corresponding to timer 1 device reads the status bit TF1 = 1 in the status register to find that timer 1 has overflowed; as soon as the bit is read the TF1 resets to 0.
2. The ISR_T0 corresponding to timer 0 device reads the status bit TF1 = 0 in the status register to find that timer 0 has overflowed; as soon as the bit is read the TF0 resets to 0.
3. The ISR corresponding to the SI device is common for the transmitter and the receiver. The ISR reads the status bits TxEMPTY and RxReady in the status register to find whether a new byte is to be sent to the transmit buffer or whether the byte is to be read from the receiver buffer. As soon as the byte is read the RxReady resets and as soon as the byte is written into the SI for transmission, TxEMPTY resets.

Some processor hardware provide for use of status register bits and some IPR bits. The IPR and status registers differ as follows. The status register is read only. (i) A status register bit (an identification flag) is *read only*, and is cleared (auto-reset) during the *read*. An IPR bit either clears (auto-resets) on the service of the corresponding ISR or clears only by a *write* instruction for resetting the corresponding bit. (ii) An IPR bit can be set by a write instruction as well as by an interrupt occurrence that waits for the service. A status register bit is set by the interrupting source hardware only. (iii) An IPR bit can correspond to a pending interrupt from a group of interrupt sources, but identification flags (bits) are separate for each source among the multiple interrupts.

Properties of the interrupt flags are as follows. A separate flag for every identification of an occurrence from each of the interrupt sources must exist. The flag sets on occurrence of interrupt: (i) It is present either in the internal hardware circuit of processor or in the IPR or in the status register. (ii) It is used for a *read* by processor or instruction after a *write* by the interrupting source hardware. (iii) It *resets* (becomes inactive) as soon as it is *read*. This is auto-reset characteristic provided in most hardware designs in order to enable this flag to indicate the next occurrence from same interrupt source. (iv) If set at once, it does not necessarily mean that it will be recognized and serviced later using an ISR. When a mask bit corresponding to that interrupt is set, even if the flag sets, the processor may ignore it unless the mask (or enable) bit modifies later. This makes it possible to prevent an unwanted interrupt from being serviced.

Example 4.12

Consider a touch screen.

It generates an interrupt when a screen position is touched. A status bit b_t is also set. It activates a interrupt request (IRQ). From the status bit, which is set, the interrupting source is recognized among the sources group (multiple sources of interrupts from same device or devices). The ISR_VECTOR_{IRQ} and ISR_{IRQ} are common for all the interrupts at IRQ.

IRQ results in processor vectoring to an ISR_VECTOR_{IRQ}. Using ISR_VECTOR_{IRQ} when the ISR_{IRQ} starts, ISR_{IRQ} instruction reads the status register and discovers that bit b_t as set. It calls for service function (get_touch_position), which reads register R_{pos} for touched screen position information. This action of reading b_t also resets the b_t if the touch screen controller-processing element provides for auto-resetting of b_t. This enables next IRQ interrupt and thus reading next-position on next touch.

4.5 MULTIPLE INTERRUPTS

4.5.1 Multiple Interrupt Calls

When there are multiple interrupt sources, each occurrence of interrupt from a source (or source group) is identifiable from a bit in the status register and/or in the IPR (Section 4.4.4). There can be interrupt service calls in succession case higher priority interrupt sources activate in succession. Then return from high priorit ISR is to lower priority pending ISR.

Let us understand two processor interrupt service mechanisms for the case of multiple interrupts.

1. Certain processors do not provide for in-between routine diversion to higher priority interrupts and presume that all interrupts or interrupts of priority greater than the presently running routine are masked till the end of the routine. Figure 4.9(a) shows diversion to higher priority interrupts at the end of the present interrupt service routine only.
2. Certain processors permit in-between routine diversion to higher priority interrupts. Figure 4.9(b) shows the actions in such processors. These processors provide, in order to prevent diversion in-between, a mechanism as follows: There is provisioning for masking of all interrupts by a primary-level bit. These processors also provision selective diversion by provisioning for masking the interrupt service selectively by secondary-level bits (Section 4.4.3).

4.5.2 Hardware Assigned Priorities

There is assigned priority order by hardware. ARM7 provides for two types of external interrupt sources (requests), IRQs and FIQs (fast interrupt requests). 8051 provides for priority order in order of interrupt vector addresses. Lower address has highest and higher has the lower priority. Interrupts in 80x86 are assigned priority order according to interrupt-types. Interrupt of type 0 has highest priority and 255 has lowest assigned priority.

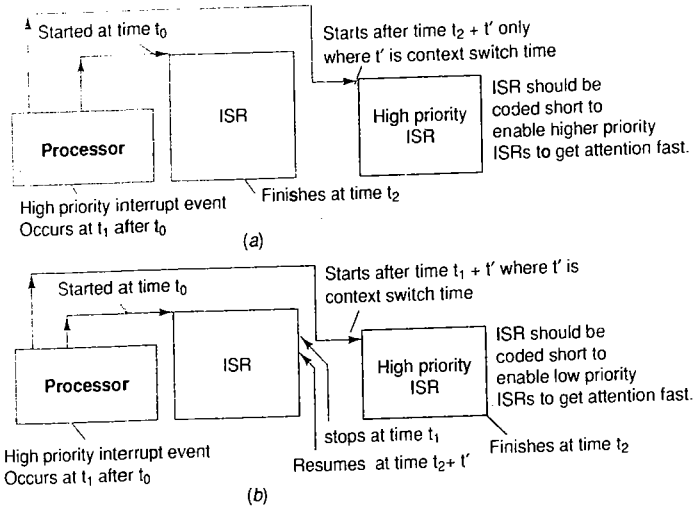


Fig. 4.9 (*a*) Diversion to higher priority interrupts at the end of the present interrupt service routine only (*b*) In-between routine diversion to higher priority interrupts unless all interrupts or interrupts of priority greater than the presently running routine are masked

When there are multiple sources of interrupts from the multiple devices, the processor hardware assigns to each source (including traps or exceptions) or source group a preassumed priority (or level or type). Let us assume a number, p_{hw} that represents the hardware-presumed priority for the source (or group). Let the number be among 0, 1, 2, ..., k, ..., m – 1. Let $p_{hw} = 0$ mean the highest; $p_{hw} = 1$ next to highest.....: $p_{hw} = m - 1$ assigned the lowest. Why does the hardware assign the presumed priority? Several interrupts occur at the same time during the execution of a set of instructions, and either all or a few are enabled for service. The service using the source corresponding to the ISRs can only be done in a certain order of priority. (There is only one processor.) Assume that there are seven devices or interrupt source groups. The processor's hardware can assign $p_{hw} = 0, 1, 2, ..., 6$. The hardware service priorities will be in the order $p_{hw} = 0, 1, 2, ..., 6$.

Software assigned priorities override these priorities, for example in 8051. Section 4.6.3 will explain this point.

Consider the example of the 80x86 family processor. Consider its six interrupt sources; division by zero, single step, NMI (non-maskable interrupt from RAM parity error and so on), break point, overflow and print screen. These interrupts are presumed to be of $p_{hw} = 0, 1, 2, 3, 4$ and 5, respectively. The hardware processor assigns the highest priority for a *division by zero*. This is so because it is an exceptional condition found in user software. The processor assigns the *single stepping* as the next priority as the user enables this source of interrupt because of the need to have a break point at the end of each instruction whenever a debugging software is run. NMI is the next priority because external memory *read* error needs urgent attention. Print screen has the lowest priority.

Which is the Interrupt to be Serviced First among those Pending? Some Way of Polling Resolves this Question. The 8086 has a 'Vectored Priority Polling Method' A processor interrupt mechanism may internally provide for the number of vectors, ISR_VECTADDRs. The *vectored priority' method* means that the interrupt mechanism assigns the ISR_VECTADDR as well as p_{hw}. There is a

call at the end of each instruction cycle (or at the return from an ISR) for a highest priority source among those enabled and pending. Vectored priorities in 80x86 are as per the n_{type}. $n_{type} = 0$ highest priority and $n_{type} =$ 0xFF (=255) lowest priority.

When there are multiple device drivers, traps, exceptions and signals as a result of hardware and software interrupts the assignment of priorities for each source or source group is required so that the ISRs of smaller deadline execute earlier by assigning them higher priorities. Hardware-defined priorities are used as default. Software assigned priorities override these priorities, for example, in 8051.

4.6 CONTEXT AND THE PERIODS FOR CONTEXT SWITCHING, INTERRUPT LATENCY AND DEADLINE

Getting an address (pointer) from where the new function begins, loading that address into the PC and then executing the called function's instructions will change a running function at the CPU to another. Before executing new instructions of the new function the processor or the OS also saves the current program's status word, registers and program contexts. If not done automatically by the processor or the OS, then the new functions, instruction should do that. This is because these (status word and registers) may be needed by the newly called function. *CPU registers including processor status word, registers, stack pointer and program current address in the PC define a function's context.* Figure 4.10(a) shows a current program context. What should exactly constitute the context? It depends on the processor or the operating system supervising the program.

The context must save if a function program or routine left earlier has to run again from the state which was left. When there is a *call* to a function (called *routine* in assembly language, *function* in C and C++, *method* in Java also called task or process or thread when it runs under supervison of the OS), the function or ISR or *exception*-handling function executes by three main steps.

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

These three actions are known as *context switching*. Figure 4.10(b) shows a current program's context switching to the new context.

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

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

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

We can say that on interrupt or function call and return the context switches and a new program is executed whenever the new context loads into the processor CPU registers. Figure 4.10(c) and (d) shows context switching for new routine and another context switch on return or on in-between call to another routine. Nesting means one function calling the second which in turn calls the third and so on and the return to the calling functions will be in the reverse order. In case of function calls there is nesting and in the case of multiple ISRs because of the presence of multiple interrupts there may or may not no nesting.

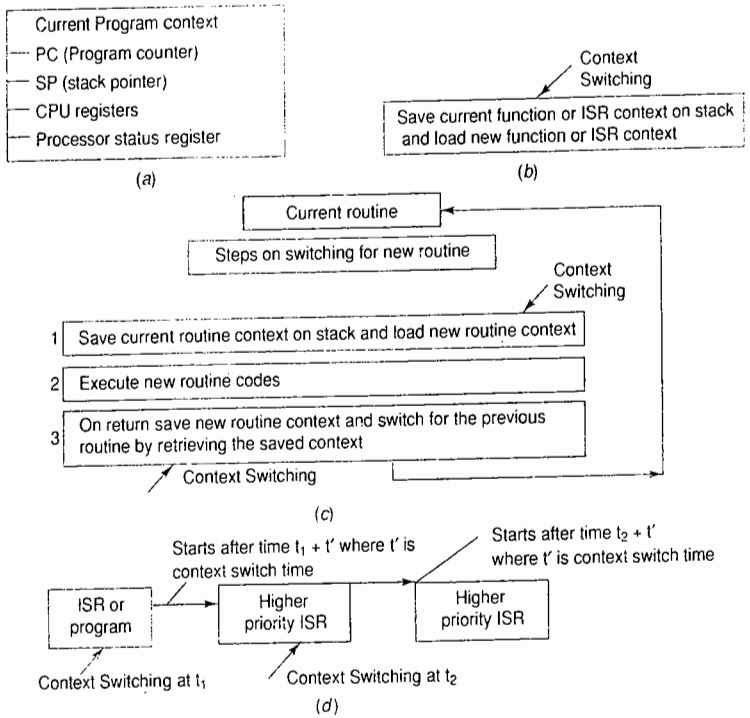


Fig. 4.10 (*a*) Current program context (*b*) New program executes with the new context of the called function or routine (*c*) Context switching for new routine and another switch on the return from routine (*d*) Context switching for new routine and another switch on return or in-between the call to another routine

Context switching means saving the context of the interrupted routine (or function) and then retrieving or loading the new context of the called routine. Example 4.13 shows how the context switching takes place in the ARM processor.

Example 4.13

Context switching is as follows in the ARM7 processor on *ISR call*. (i) The interrupt mask (disable) flags are set. (Disable low priority interrupts.) (ii) Next instruction PC is saved at link register. (iii) Current program status register (CPSR) copies into the saved program status register (SPSR) and CPSR stores the new status during new instructions. (iv) PC gets the new value as per the interrupt source from the vector table. An ISR return context switching back to the previous context is as follows. (i) PC is retrieved from link register. (ii) The corresponding SPSR copies back into the CPSR. (iii) The interrupt mask (disable) flags are reset. (Enable again the earlier disabled low priority interrupts.)

The time taken in context switching, T_{switch} has to be included in a period called *interrupt latency* period, T_{lat}. Example 4.14 shows how to calculate the context switching time period, which is to be accounted in calculating the interrupt latency (Section 4.6.1).

Example 4.14

1. ARM7 processor context switching's minimum period equals two clock cycles plus 0–20 clock cycles for finishing an ongoing instruction plus 0–3 cycles for aborting the data. The 0 cycle when an interrupt occurs just before the end and 3–20 when during an instruction. Longest time taken for an ARM instruction is 20 cycles.
2. During context switching for new routine call or for return. CPSR copies into SPSR on switching from a routine. CPSR means current program status register and SPSR means saved program status register. 3 cycles are taken in switching the CPSR.
3. Two clock cycles are needed for the start of the execution stage of switched routine's first instruction.

Aborting the processor data means CPSR not coping into an SPSR. Then step 2 three cycles are not taken up.

1. Minimum period is thus four (2 + 2) for data abort interrupt. [Steps 1 and 2 above]
2. Maximum is 27 clock cycles (2 + 20 + 3 + 2) for other than data abort interrupt. Maximum is when the interrupt occurs just at the start of execution of the longest time taking instruction in the processor. [Steps 1, 2 and 3 above]

Thus for any latency period calculation, 27 clock cycle periods as context switching time are taken into account when estimating latency in an ARM-based system.

Each running program has a context at an instant. Context reflects a CPU state (PC, stack pointer(s), registers and program state (variables that should not be modified by another routine). Context saving on the call of another ISR or task or routine is essential before switching to another context.

4.6.1 Interrupt Latency

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

1. When the interrupt service starts immediately on context switching the interrupt latency T_{switch} equals the context switching period. When instructions in a processor take variable clock cycles, maximum clock cycles for an instruction are taken into account for calculating the latency. Figure 4.11(a) shows latency in case the interrupt service starts immediately.
2. When the interrupt service does not start immediately but context switching starts after all the ISRs corresponding to the higher priority interrupts complete the execution. If the sum of time intervals for completing the higher priority ISRs equals ΣT_{exec}, then interrupt latency equals $T_{switch} + \Sigma T_{exec}$. Figure 4.11(b) shows latency in case the interrupt service starts after present ISR of higher priority interrupt completes the execution.
3. We disable the interrupt system when a routine enters a critical section and enable the interrupts when the routine exits the critical section codes. A routine of function or ISR may consist of codes for critical region instructions and before the critical section codes all the interrupts are disabled and enabled by the

end of the critical section. $T_{disable}$ may or may not be included depending on the programmer's approach. Let $T_{disable}$ be the period for which a routine is disabled in its critical section. The interrupt service latency from the routine with the interrupt-disabling instruction (because of the presence of the routine with critical section) for an interrupt source will be $T_{switch} + \Sigma T_{exec} + T_{disable}$. Figure 4.11(c) shows interrupt latency as sum of the periods for T_{switch}, ΣT_{exec} and $T_{disable}$ when the presently running routine to be interrupted is executing critical section codes.

Worst case latency is sum of the periods T_{switch}, ΣT_{exec} and $T_{disable}$ where the sum is for the interrupts of higher priorities only. Minimum latency is the sum of the periods T_{switch} and $T_{disable}$ when the interrupt is of the highest priority. For latency computations, worst case is taken into account.

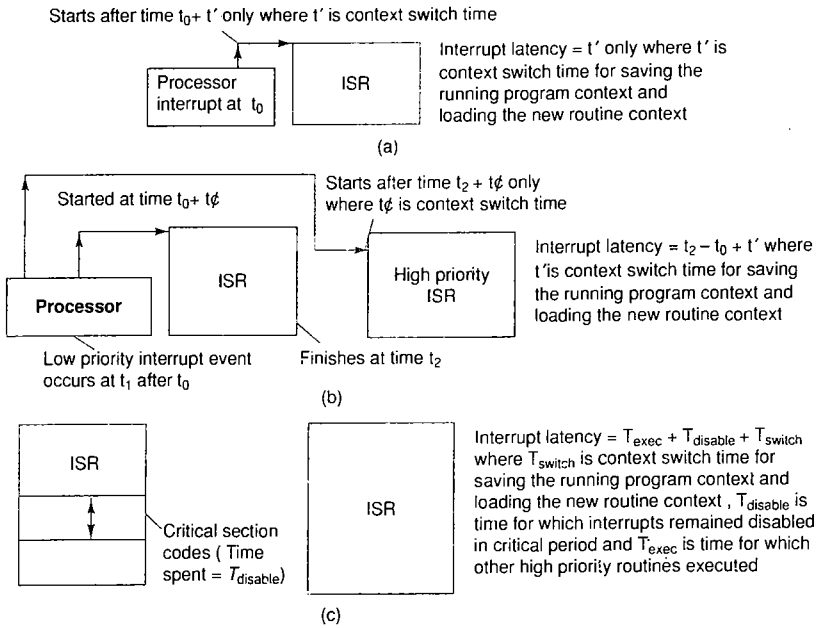


Fig. 4.11 (*a*) Latency in case the interrupt service starts immediately (*b*) Latency in case the interrupt service starts only after the interrupt service routine presently running completes execution (*c*) Interrupt latency as sum of the periods for T_{switch}, ΣT_{exec} and $T_{disable}$ when the presently running low priority routine to be interrupted is having critical section codes

Example 4.15

80196 microcontroller has an SI device which has a FIFO (first in first out) buffer and the SI reads the bytes and puts it in the buffer. SI generates three interrupts: RI on one byte reception, FIFO_4thEntry interrupt when FIFO is half full and FIFO_Full interrupt when the FIFO is full. Assume that a microcontroller has two devices: SI similar to 80196 and timer T. SI has a serial input buffer of 8 bytes (a FIFO of 8 bytes

0 – 7). Assume that a serial input at the SI reads a byte from a network and generates three types of interrupts. T generates timer-overflow and timer-capture interrupts, called TF and TCAPTURE interrupts. Worst-case interrupt latencies are as follows.

1. When the seventh byte is received, the controller generates interrupt FIFO_FULL and a FIFO_FULL flag sets in S0. Assume that it is the top priority serial interrupt and the ISR execution time is T_{exec} (FIFO_FULL). For the FIFO_FULL interrupt, the interrupt latency is $T_{switch} + T_{disable}$ because it is a top priority ISR.
2. When the zeroth byte is received, the SI generates an interrupt RI and an RI flag sets in the status register S0. Assume RI has the lowest priority serial interrupt and RI ISR execution time is T_{exec} (RI). For the RI interrupt, the interrupt latency is $T_{switch} + T_{exec}$ (TCAPTURE) + T_{exec} (TF) + $T_{disable}$, because it has the lowest priority than the timer interrupts.
3. When the third byte is received, the SI generates an interrupt FIFO_4thEntry and a FIFO_Half flag sets in S0. Assume that it is the middle priority serial interrupt and has priorities lower than the TCAPTURE interrupt but higher than the timer overflow. Assume that the ISR execution time is T_{exec} (FIFO_Half). For FIFO_4thEntry interrupt, the interrupt latency is $T_{switch} + T_{exec}$ (TCAPTURE) + $T_{disable}$, because it has higher priority than timer overflow but it has lower priority that TCAPTURE. The T_{exec} (RI) is not taken into account because if RI is not responded then only FIFO_Half interrupt occurs. Both interrupts RI and FIFO_Half belong to the same SI device.

Each running program when interrupts, the interrupting source service routine takes some time before starting the servicing codes. That time interval is called interrupt latency. It is the sum of the execution time of higher priority interrupts and the context switching period. If an interrupted routine is having a critical section (interrupts disabled), the interrupt latency increases by period equal to the interrupts disabled period.

4.6.2 Interrupt Service Deadline

For every source, the service of its ISR instructions can be kept pending up to a maximum period. This period defines the deadline during which the service must be completed. It should not be less than the worst-case interrupt latency. Figure 4.12(a) shows interrupt latency period and deadline for an interrupt.

A 16-bit timer device on overflow raises TF interrupt on transition of counts from 0xFFFF to 0x0000. It has to be responded by executing an ISR for TF before the next overflow of the timer occurs, else the counting period between 0x0000 after overflow and 0x0000 after the next-to-next overflow will not be accounted. The timer counts increment every 1 µs; the interrupt service deadline is 65536 µs.

Video frames in video conferencing reach after every 1 ÷ 15s. The device on getting the frame interrupts the system and the interrupt service deadline is 1 ÷ 15s, else the next frame will be missed.

Example 4.15 FIFO_Full interrupt must be executed fast as it has shorter deadline compared with RI and the fourth entry interrupt. If ISR for FIFO_Full interrupt does not execute before the next character at the SI device, the character will be missed. If ISR for FIFO_4th entry interrupt does not execute fast, it does not matter, because eventually there is a cushion of SI raising the FIFO_Full interrupt. If ISR for RI interrupt does not execute fast, it does not matter, because eventually there is a cushion of SI raising FIFO_4th entry interrupt as well as FIFO_Full interrupt. FIFO_Full interrupt is said to have a service interrupt service deadline. If SI device is receiving characters at 64 kbps and in 11-bit UART format, the FIFO_Full interrupt service deadline is 171.9 µs. FIFO_RI interrupt service deadline is 171.9 µs if SI device does not have the buffer and provisions for FIFO_Half and FIFO_full interrupts.

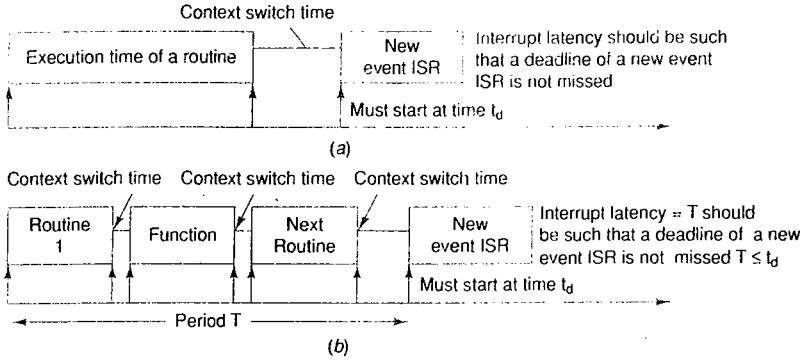


Fig. 4.12 (*a*) Interrupt latency period and deadline for an interrupt (*b*) Short interrupt service routines (ISR) and functions, which run at later instances so that the other ISR deadlines are not missed

A good software design principle for multiple interrupt sources is to keep the ISR as short as possible. Why? This is service the in-between pending interrupts and leave the functions that can be executed afterwards for a later time. When this principle is not adhered to, a specific interrupting source may not be serviced within the deadline (maximum permissible pending time). Section 4.2.3 described use of interrupt service threads, which are the second-level interrupt handlers. Figure 4.12(b) shows a short ISR and functions, which run at later instances so that the other ISR deadlines are not missed.

The system therefore has to meet the deadlines set for service of each system device. This can be understood by the following examples. Consider the example of a video system. When the system is running, two device-driver ISRs also run. One driver is for the voice device and the other for the image device. The ISRs and the other system software design for these two device drivers have to maintain synchronization else the next set of images and the next set of voice signals will be missed.

Therefore, the system software designer designs the appropriate ISRs for multiple device interrupts so that all device interrupt calls are serviced within the stipulated deadlines of each interrupt. The design should provide optimum latencies and set appropriate deadlines for each service routine and functions.

Each ISR may have a interrupt service deadline when interrupts. An ISR with a deadline must have interrupt latency less than the deadline.

4.6.3 Software Over-riding of Hardware Priorities to Meet Service Deadlines

Which source or source group has higher priority with respect to the others that is first decided among the ISRs that have been assigned higher priority in the user software. If user-assigned priorities are equal then the highest priority is that which is preassigned at the processor internal hardware. The 8051 internal interrupt mechanism is as follows. There is the interrupt priority (IP) register at 8051 in which there are five priority bits for the five interrupt sources in 8051. Also there are the five interrupt-enable bits in the IE register. These are secondary-level enable bits of the processor's service of ISRs. When a

priority bit at IP is set, the corresponding interrupt source gets a high priority, and if reset, it gets a lower priority. The 8051 first selects by polling among the high priority according to the bits at IP register.

There is a need for over-riding the priority order by assigning priorities. The need of reassigning priorities over hardware pre-assigned priorities can be understood from the following example.

Example 4.16

Assume that there are two sources of interrupts: serial port input and A/D conversion. A/D conversion time is 200 μs and SI device with no data buffer receiving inputs at 64 kbps with minimum separation between the characters equals 171.9 μs. The A/D conversion should therefore have the lower priority than RI interrupts of SI. When the system hardware has the internal devices, it assigns lower priority to the A/D end of the conversion interrupt. Suppose that the SI device is used to receive input at 16 kpbs and assume that the UART mode has 11 bits per character. When the A/D conversion is needed continuously (e.g., when ECG signals are input), the software should assign the higher priority to A/D, because SI receives character every 11/16 ms = 687 μs and A/D every 200 μs, at a rate faster than the SI.

Software-assigned priorities can be used to over-ride the hardware priorities. OS provides the functions, which assign the software priorities to each ISR, IST and task of the real-time system.

4.7 CLASSIFICATION OF PROCESSORS INTERRUPT SERVICE MECHANISM FROM CONTEXT-SAVING ANGLE

1. The 8051 interrupt service mechanism is such that on occurrence of an interrupt service, the processor pushes the processor registers PCH (program counter higher byte) and PCL (program counter lower byte) onto the memory stack. The 8051 family processors do not save the context of the program (other than the absolutely essential PC) and a context can save only by using the specific set of instructions in the called routine. For example, using push instructions. It speeds up the *start* of ISR and *return* from ISR but at a cost. The onus of context saving is on the programmer in case the context (SP and CPU registers other than PCL and PCH) is to be modified on service or on function calls during execution of the remaining ISR instructions.
2. The 68HC11 interrupt mechanism is such that processor registers save onto the stack whenever an interrupt service occurs. These are in the order of PCL, PCH, IYL, IYH, IXL, IXH, ACCA, ACCB and CCR. The 68HC11 thus does automatically save the processor context of the program without being so instructed in the user program. As context saving takes processor time, it slows a little the *start* of ISR and *return* from the ISR but at the great advantage that the onus of context saving is not on the programmer and there is no risk in case the context modifies on service or function calls.
3. Certain processor provides for fast context switching two stack frames with each stack frame consisting of the same number of registers, for example, 16 or 32 registers. The PC, stack-pointer and link-register define one stack frame. When context switches from one routine to another, only the pointer to the stack frame changes. The ISR stack frame that is called has the current program context and the interrupted program context becomes the saved program context. ARM7 provides such a mechanism. Certain processors provide for more than two stack frames with each stacking a context.

The OS program also provides for memory blocks, which are used as multiple stack frames for the tasks (processes or threads). This enables multi-threading and multi-tasking.

Certain processors provide for saving only the PC. Certain processors provide for saving only the PC and other CPU registers before calling the ISR and context switching. Certain processors provide for fast context switching by providing internal register frames for the stack or providing sets of local (internal) stack for the contexts. Fast context switching reduces the interrupt latencies and enables the meeting of each function or routine deadline for service. The operating system provides for multiple stack frames to enable multitasking and context switching using the multiple stack frames.

4.8 DIRECT MEMORY ACCESS

Assume that the data transfer is to occur between hard disk system memory. The DMA is used in that case. When the I/Os are needed for large amount data from a peripheral device to the memory addresses in the system or large amount of data is to be transferred by the I/Os, the interrupt-based mechanism is not suitable.

A DMA facilitates a multi-byte data set or a burst of data or a block of data transfer between the external device and system or between two systems. A device that facilitates DMA transfer has a processing element (single purpose processor). The device is called DMAC (DMA Controller). Data transfer occurs efficiently between the I/O devices and system memory with the least processor intervention when using DMAC. The system address and data buses become unavailable to processor and available to the IO device that interconnects using DMAC during the data transfer. Figure 4.13 shows the interconnections using the DMAC. It also shows the buses and control signals between the processor, memory, DMAC and data-transferring I/O device.

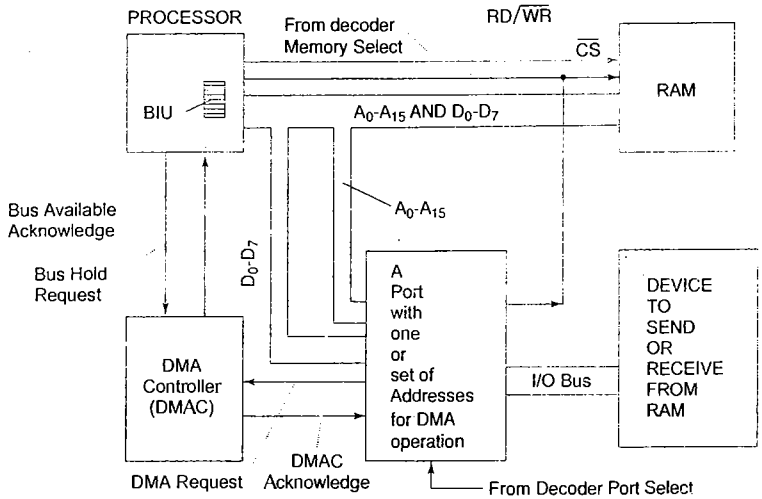


Fig. 4.13 The buses and control signals between the processor, memory, DMA controller and data-transferring I/O device

The DMAC sends a hold request to the CPU and the CPU acknowledges that if the system memory buses are free to use. Three modes are usually supported in DMA operations. (i) Single transfer at a time and then release IO bus hold on the system bus after each transfer. (ii) Burst transfer at a time and then release of the IO bus hold on the system bus. A burst may be of a few kilobytes. (iii) Bulk transfer and then release of the IO bus hold on the system bus only after the transfer is completed.

4.8.1 Use of DMAC

Whenever a DMA request is made to the DMAC for the I/Os, the DMAC is first initialized. It is programmed for (i) read or write, (ii) mode (bytes, burst or bulk) of DMA transfer, (iii) total number of bytes to be transferred and (iv) starting memory address. Consider a read operation (external device to memory transfer). DMA proceeds without the intervention of the CPU, except (i) at the start of DMAC programming and initializing and (ii) at the end. Whenever a DMA request by the external device is made to the DMAC, the CPU is requested the DMA transfer by DMAC at the start to initiate the DMA and at the end to notify the end of the DMA by DMAC. Example 4.16 explains the data transfer operation.

Example 4.17

Assume that 2 kb of data needs to be transferred. One method is that device interrupts the processor, when 1 or 4 or 8 bytes of data are ready and generate the interrupt. The ISR reads the 1 or 4 or 8 bytes and put these into the memory addresses. Assume that the device generates the interrupt and transfers the 8 bytes. Number of interrupts required will be 2 k/8 = 2048/8 = 256 and ISR has to be run 256 times. A DMA is the better approach.

An IO program initializes the DMAC for 2 kb burst mode transfer from a memory address for the I/O to an external device starting from memory address M_1. DMAC loads 2048 in a data count register and loads M_1 in address register on initialization.

On an external device requesting the DMA, the DMAC sends HOLD request signal to the processor. Processor acknowledges by the HLDA (hold acknowledge) signal that when the system buses are not in use.

DMAC transfers the bytes from I/O bus to the memory bus in burst from IO bus to the memory bus D0-D7 lines and keeps track of the data counts in the DC (data count) register. Transfer takes place to addresses from M_1 to M_1 + 2047. DC = 0 after the transfer completes.

DMAC interrupts the processor so that the processor is notified at the end of DMA transfer and an ISR can re-initialize the DMAC for the next transfer.

A DMAC may also provide memory access to multiple channels. A multi-channel DMAC provides DMA action from system memories and two (or more IO) devices. There is a separate set of registers for programming each channel. There may be the separate or common interrupt signals in the case of multi-channel DMAC.

The 80x86 processors do not have on-chip DMAC units. The 8051 family member 83C152JA (and its sister JB, JC and JD versions) have two DMA channels on-chip. The 80196KC has a PTS (peripheral transactions server) that supports DMA functions. (Only single and bulk transfer modes are supported, not the burst transfer mode.) The MC68340 microcontroller has two on-chip DMA channels. 80960CA has four-channel DMAC on chip, with a mode called demand transfer mode also provided.

On-chip or a separate DMAC facilitates fast direct byte transfers between memory and I/O devices compared with interrupt-driven data transfer as that has in-built processing element and uses the system buses as and when they are made available by the processor. Designers can use DMAC in sophisticated systems so that the system performance improves by separate processing of bulk or burst data transfer from and to the peripherals.

4.8.2 Use of DMA Channel in Case of Multiple Interrupts in Quick Succession from the Same Source

A good feature of DMA-based data transfer service is very small latency periods compared with data transfer using multiple IO interrupt sources and multi-byte bulk or burst data transfers. The interrupt service routine period from start to end can now be very small as the ISR that initiates the DMA to the interrupting source, simply programs the DMA registers for the command, data count, memory block address and I/O bus start address (Section 4.8.1).

The use of DMA channels for the IO services in place of processor interrupt-driven ISRs provides an efficient method when the device has to transfer large amount of data by I/O. This is because a DMA transfer uses the periods when the system buses are free.

4.9 DEVICE DRIVER PROGRAMMING

A system has number of physical devices (Chapter 3). A device may have multiple functions. Each device function requires a driver. Examples of multiple functions in a device are as follows.

1. A timer device performs timing functions as well as counting functions. It also performs the delay function and periodic system calls.
2. A *transceiver* device transmits as well as receives It may not be just a repeater. It may also do the *jabber control* and *collision control.* (Jabber control means prevention of continuous streams of unnecessary bytes in case of system fault. Collision control means that it must first sense the network bus availability then only transmit.)
3. Voice-data-fax modem device has transmitting as well as receiving functions for voice, fax as well as data.

A common driver or separate drivers for each device function are required. Device drivers and their corresponding ISRs are the important routines in most systems. The driver has following features.

1. *The driver provides a software layer (Interface) between the application and actual device*: When running an application, the devices are used. A driver provides a routine that facilitates the use of a device function in the application. For example, an application for mailing generates a stream of bytes. These are to be sent through a network driver card after packing the stream messages as per the protocol used in the various layers, for example, TCP/IP. The network driver routine will provide the software layer between the application and network for using the network interface card (device).
2. *The driver facilitates the use of a device by executing an ISR*: The driver function is usually written in such a manner that it can be used like a black box by an application developer. Simple commands from a task or function can then drive the device. Once a driver function is available for writing the codes, the application developer does not need to know anything about the mechanism, addresses, registers, bits and flags used by the device. For example, consider a case when the system clock is to be set to tick every 10,000 μs (100 times each second). The user application simply makes a call to an OS function like OS_Ticks (100). It is not necessary for the user of this function to know which timer device will perform it. What are the addresses, which will be used by the driver? Which will be the device register where value 100 registers for the ticks? What are the control bits that will be set or reset? OS_Ticks (100) when run, simply interrupts the system and executes the SWI instruction which calls the signalled routine (driver ISR) for the system ticking device. Then the driver ISR which executes takes 100 as *input* and

configures the real time clock (Section 3.8) to let the system clock tick each 10,000 μs and generate the system clock interrupts continuously every 10,000 μs to get 100 ticks each second.

Generic device driver functions in high level language are used in high level language program. The functions are open, close, read, write, listen, accept etc.

Device driver ISR programming in assembly needs an understanding of the processor, system and IO buses and the addresses of the device registers in the specific hardware. It needs in-depth understanding of how the software application program will seek the device data or write into the device data and what is the platform. Platform means the operating system and hardware, which interfaces with the system buses.

A common method of using the drivers is as follows: a device (or device function module) is opened (or registered or attached) before using the driver. If means device is first initialized and configured by setting and resetting the control bits of device control register and use of the interrupt service is enabled. Using a user function or an OS function, a device (or device function module) can also be closed or de-registered or detached by another process. After executing that process, the device driver is not accessible till the device is re-opened (re-registered or re-attached).

4.9.1 Writing Physical Device-Driving ISRs in a System

For writing the software for driver in assembly, the following points must be clear.

1. Information about how the device communicates.
2. Information about the three sets of device registers–data registers or buffers, control registers and status registers. A device *initializes* (configures, registers, attaches) by setting the control register bits. A device *closes* (resets, de-registers, detaches) by resetting the control register bits. (Example 4.18)
3. Information of other registers and common addresses to a device register.
4. Control register bits control all actions of the device. A control bit can even control which address corresponds to which data register at an instant. For example, at the instance when the DLAB control bit is set, the 0x2F8 corresponds to the divisor-latch lower byte (Example 4.19).
5. Status register bits reflect the flags for status of the device at an instant and change after performing actions as per the device driver. A status flag at a status register reflects the present status of device. For example, an instance between finishing the transmission of bits from a TRH buffer register and obtaining the new bits for next transmission, a transmitter empty flag (TDRE) reflects it (Example 4.19).
6. Either setting of an enable bit (interrupt control flag) is used by the system to initiate a call for executing an ISR related to the device driver function. ISR executes if: (i) it is enabled (not masked at the system) and (ii) the interrupt system itself is also enabled.

The following information must therefore be available when writing a device control and configuring and driver codes.

1. *Addresses for each register*: Physical device hardware and its interfacing circuit fix the addresses for a physical device and they usually cannot be relocated. The device becomes the *owner* of these addresses. For example, IBM PC hardware is designed such that the device addresses are as following:
   a. *Timer* addresses between 0x0040-5F;
   b. *Keyboard* addresses between 0x00600-6FD, real-time clock (system clock) addresses between 0x0070-7F;
   c. *Serial COM port 2* addresses between 0x02F8-2F and *serial COM port 1* addresses between 0x03F8-3F.
2. There may be input-buffer register as well as output-buffer register at a common address. This is because during device write and read instructions at the control bus the different signals RD and WR

are issued. The physical device can thus select the appropriate register when taking action. For example, there is a register SBUF at 8051. It addresses both the output serial buffer and input serial buffer.

3. There may be multiple registers at the same address. Refer Example 4.19. This example shows the following. RBR (receiver data buffer register) and TRH (transmitter holding register) are at the same address (0x2F8) in PC COM2 serial device. This address is also common for the lower byte of divisor latch, which is used for presetting the device baud rate. A control bit is made 1 to write this byte when setting the device baud rate and later it is made 0 for using the same address as RBR and TRH during the device '*read*' and '*write*' instructions, respectively.
4. Purpose of each bit of the control register.
5. Purpose of each status flag in the status register. Which status bit when set and reflects a device interrupt, calls to which ISR.
6. Whether control bits and status flags are at the same address. The processor reads the status from this address during the read instructions. The processor writes the control bits at that address during the write instructions.
7. Whether both, control bits and flags coexist in the same register.
8. Whether the status flag, which sets on a device interrupt, auto-resets on executing the ISR or if an ISR instruction should reset.
9. Whether control bits need to be changed, reset or set again before return to the interrupted process.
10. List of actions required by the driver at the data buffers, control registers and status registers.

Section 8.6.1 will describe in detail the device management functions at an OS. The OS usually provides device-related functions so that for the new device also the drivers are written in an identical manner. For example, Unix device driver components are: (i) device ISR, (ii) device initialization codes (codes for configuring device control registers) and (iii) system initialization codes, which run just after the system resets (at bootstrapping). Microsoft OS Windows provides the Windows driver functions (WDF) and user-mode driver framework (UDMF). Linux provides device drivers (Section 4.9.6). Using object-oriented programming approach, *Class drivers* are also written for operation on large number of similar type of devices using identical bus or network protocol, for example, printers or CD drives or class drivers for the USB-based devices.

When a device driver function such as read or write or open is called, the OS first intiates the logical layer part. The logical layer then initiates the physical layer, which implements OS device function using driver ISR functions written in assembly so that the device hardware performs the actions accordingly. Similary the device sends the response of the commands to the logical layer of the driver through the physical layer.

The device drivers execute according to the device hardware, interrupt service mechanism, OS, system and IO buses. A device driving ISR is designed using the device addresses and three sets of device registers– data registe(s) or buffer(s), control register(s) and status register(s). A device is configured and controlled by the control bits. The driver ISR initiates and executes on status flag change. A list of actions required by the driver at the data buffers, control registers and status registers is needed and is prepared before writing the driver codes. The driver codes are sensitive to the processor and memory. This is because: (i) when the device addresses change, the program should also be also be modified and (ii) when a processor changes, the interrupt service mechanism changes. The OS usually provides device drivers for the system devices.

4.9.2 Virtual Device Drivers

Virtual device drivers emulate the device hardware, for example, hard disk and generate software interrupts similar to physical device drivers. The file and pipe are two examples of virtual devices.

Both virtual devices and physical device drivers have functions for device *open*, *read*, *write* and *close*. Consider the analogies of a file device with a physical device. (i) Just as a *file* needs to be *opened* to enable

read and write operations, a device may need to be sent an *interrupt call* for initializing and configuring it (opening, registering or attaching it. Setting control bits appropriately does this). (ii) Just as a file is sent a *read call*, a device must be sent another *interrupt call* when its input buffer(s) is to be read. (iii) Just as a file is sent a *write call*, a device needs to be sent *another interrupt call when its output buffer is to be written*. (iv) Just as a file is sent a *close-call* a device needs to be sent *another interrupt call to disable* (close or deregister or detach) it from the system for further read and write operations.

The concept of virtual (software) device drivers is very important in programming. Examples are as follows.

1. A memory block can have data buffers in analogy to buffers at an IO device and can be accessed from a *char* driver or a *block* driver. The device is called the *char* device or the *block* device when it can access a character or a block of characters, respectively.
2. A physical device transceiver (with input–output block buffer) or repeater is equivalent to a virtual device called *loop back device*. It stores allocated memory blocks using a block device driver and returns the data back from the memory.
3. A bounded buffer device in memory can be like a printer buffer. A data stream is sent by one routine (driver) and read by another routine (driver). Bounded buffer device is a virtual device, usually called *pipe* device.
4. A program can store in a set of memory blocks called *RAM disk* in the analogous way a file system does at the hard disk. RAM disk is a device that consists of multiple internal file devices.

The virtual device is an innovative concept for system software design. Drivers for these are also written like the physical device drivers. Important devices are char device, block device, loop back device, file device, pipe, socket and RAM disk. Device configuring is equivalent to creating a file. Device activation on the interrupt is equivalent to opening a file. Device resetting is equivalent to closing a file. Device detaching is equivalent to freeing the memory space allotted for a file data.

4.9.3 Parallel Port Drivers in a System

Device driver *read* ( ) function can be implemented by calling an ISR is Port_ISR_Input, which handle the port input. Figure 4.14(a) shows control and status bits used in the ISRs in the device drivers and port pins interface with the data bus. Figure 4.14(b) shows step *A* for port initialization, step *B* for calling the driver and steps 0 to 5 for driver Port_ISR_Input. The driver reads byte from port and puts it into a queue that builds in memory on successive inputs to the port.

Port_ISR_Input does the following:

1. Step *A* sets the device control bit for read. Step *B* is no action till input event.
2. Steps 0 to 2 are for reading the input buffer(s) by emptying the buffer and storing the byte(s) in memory or using the bytes received as per the system requirement.
3. Step 3 resets the device receive-buffer ready flag (in status register) and thus prepares the device for the next read after step 4. In step 4, interrupt flag resets to enable next byte read on next interrupt.

An example for device driver *write* ( ) function is a driver ISR for handling the port outputs. The ISR does the following:

1. Sets the device control bit for write.
2. Sends into the device output buffer (s) the byte(s) from the memory.
3. Resets the device-transmit buffer-empty flag (in status register) on completion of transmission of the byte(s) and prepare the device for the next write.

Example 4.18 gives a device driver ISR example using 68HC11 microcontroller port C (68HC11 microcontroller knowledge is presumed here).

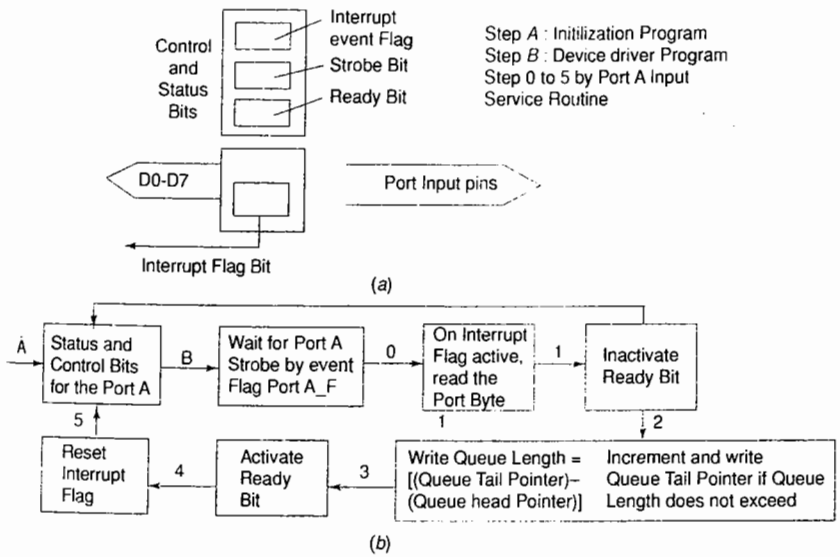


Fig. 4.14 (*a*) Control and status bits used in the interrupt service routines (ISRs) called by the device drivers and port pins used to interface the data bus (*b*) Step *A* for initialization, step *B* for the interrupt of the driver and steps 0 to 5 for driver Port_ISR_Input execution. The driver reads a byte from a port and puts it into a queue that builds in memory on successive inputs to the port

Example 4.18

Device driver *read* ( ) function calls an ISR PortC_ISR for handling the port C inputs in 68HC11. Port C uses the hand-shaking signals. Figure 4.15(a) shows hand-shaking signal to port C. Figure 4.15(b) shows control and status bits used in the *call* to the driver. Figure 4.15(c) shows port C as input and its interface with the data bus. Figure 4.15(d) shows port C as output. Figure 4.15(e) shows step *A* for initialization, step *B* of the interrupt and step *C* for executing PortC_ISR. The ISR reads from the port and inserts the byte into a queue. The latter builds in memory on successive inputs to the port. An external peripheral activates STRA pin. The peripheral requests a transfer of its byte to port C through the STRA. When STRA pin activates by '0', the port C gives an acknowledgment in case *STAI* (STRA interrupt mask bit) at a control register is not set (*STAI is not at* '1'). STRB pin sends hardware signal for the ready status (or acknowledgement) from the port C to the peripheral. When *STAI* is programmed to '0', the peripheral puts the byte into the port buffer as soon as STRB pin sends acknowledgement. As soon as the peripheral completes by putting the byte at the port C, the *STAF* sets (=0). *STAF* is at status register. *STAF* is the interrupt flag, which sets when the external device completes putting the byte at the port C.

Port C memory address is 0xp003, when page address configured on 68HC11 is 0xp000 (p is 4-bit maximum-significant nibble). A call to the device driver ISR for port C device *open* ( ), three actions occur by the device initialization program. (i) Define port C address as follows. # define PortC 0x1003 /* p bits as 0001 */. (ii) Reset all eight bits to 0s at DDRC so that port C becomes an input parallel port. DDRC is data direction

register for port C at memory address 0xp007. (iii) On initialization call, *STAI* sets to '1' for enabling interrupting by the peripheral, which connects to port C. *STAI* is the sixth bit of PIOC (port I/O control register). It is at memory address 0xp002.

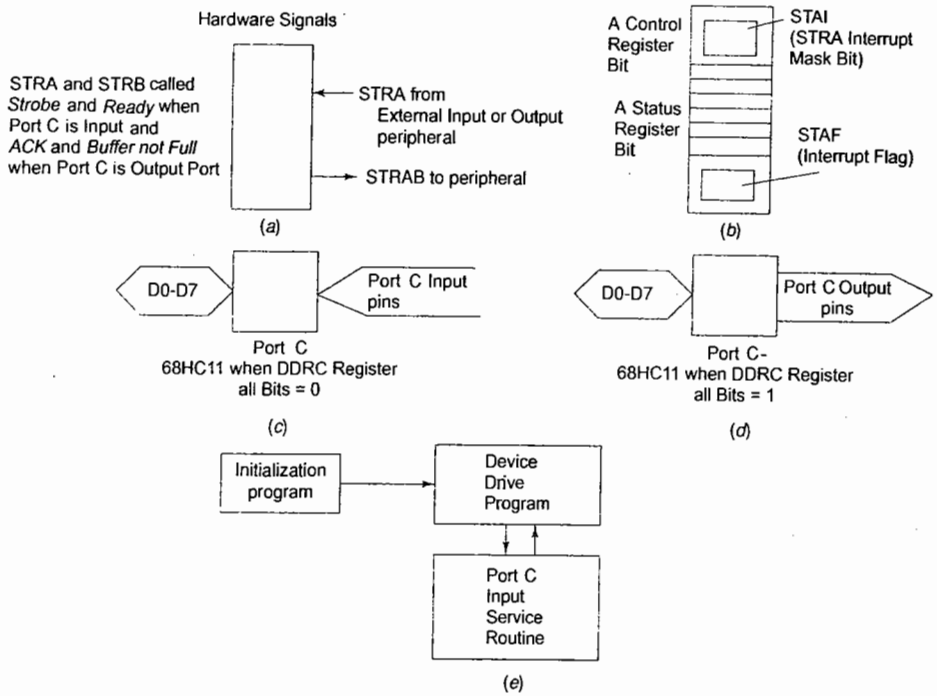


Fig. 4.15 (*a*) Hand-shaking signal to a parallel port (*b*) Control and status bits used in the system calls by driver functions (*c*) Port C as input and its interface with data bus (*d*) Port C as output (*e*) Step *A* for initialization, step *B* for system call to the driver and step *C* for driver PortC_ISR

A driver ISR program for Port C read will execute after the following actions.

1. If *STAI* is set '0' then read *STAF*. (*STAF* is the seventh bit at PIOC. PIOC also provides the status bits. It is for control cum status bits.)
2. If *STAF* is set '0' then interrupt for call to *portC_ISR* (port C service routine), otherwise wait.
3. There is no need to software reset *STAF* as there is automatic hardware reset of it by 68HC11 as soon as *portC_ISR* is called.

Driver routine *portC_ISR* programming is done as follows. Assume the name of pointers and variables are as following: (i) **portC_Queueback* is a pointer that points to a memory address where the byte from port C inserts into to a queue. (ii) *portC_Queuelength* is present queue length. (iii) *portC_Max QueueSize* is the maximum queue length defined for the port C received bytes.

1. If *quasi_bidir* bit does not equal to false, write 0xFF to port C.
2. Read port C.
3. If *portC_Queuelength* is less than the *portC_MaxQueueSize* store port bits at the address defined by **portC_Queuetail*.
4. If *portC_Queuelength* is not equal to *portC_MaxQueueSize* then increment **portC_Queuetail* to let it now point to next address. When equal then call an exception (*error* routine) for port C.

4.9.4 Serial Port Drivers in a System

There is IEEE standard called POSIX (portable operating system interface) standard. Portability of the UART drivers in different systems is essential. In a PC with 80x86 processor an UART 8250 or a new generation UART device UART 16550, which includes the 16 byte FIFO input and output buffer is used. Example 4.19 gives all three sets of the registers (data, control and status) for a serial-line UART device in a PC. All PCs have this device.

Example 4.19

A serial-line device 8250 or 16550 in a 80x86-based IBM PC has the addresses of device registers as follows. These addresses are fixed by hardware configuration of the UART port interface circuit in IBM PC system employing the 80x86 processor. They are from 0x2F8 to 0x2FE at COM2 port in a PC and 0x3F8 to 0x3FE at COM1 port. Consider COM 2.

1. Two I/O data buffer registers (RBR for receiving and TRH for transmitting) are at a common address, 0x2F8—
   (a) Provided a control bit at address 0x2FB is 0, (i) during read from the address, the processor accesses from the RBR or (ii) during write to the address, processor accesses the TRH.
   (b) Provided a control bit at address 0x2FB is 1, data of two bytes of *divisor latch* are at distinct addresses, 0x2F8 (LSB) and 0x2F9 (MSB). Divisor latch holds a 16-bit value for dividing the system clock. This then selects the rate of serial transmission of bits at the serial line. [While writing a device driver, remember that a bit in another register (control register) changes the 0x2F8 access from access to the IO register to the lower byte register at divisor latch register.
2. Three control registers are at three distinct addresses 0x2FA, 0x2FB and 0x2FC. These are for writing in registers as follows—
   (a) IER (interrupt-enabling register). It enables the device interrupts.
   (b) LCR (line control register). It defines how and how many bits will be on the line.
   (c) MCR (modem control register). It defines how the modem handshakes and communicates.
3. Three status registers of the device are also at three addresses 0x2FA, 0x2FD and 0x2FE and are used during read from these. These are as follows—
   (a) IIR (interrupt identification register) for flags at 0x2FA. A flag sets on a device interrupt and resets at the servicing of corresponding device interrupt.
   (b) LSR at 0x2FD. It is for reading line status the number of bits that will be present on the line.
   (c) MSR at 0x2FE. It specifies the modem status bits during handshakes and communication.

Assume that device has been given identity number 5. It is also a file descriptor for the device that points to description parameters of the device.

(a) A serial device high-level driver function, open (5, baudrate) will configure and initialize the device. It sets the reset flag in IIR. The device initializes by unmasking the device interrupts and writing the control bits for clock divisor latch at the specified address. Divisor latch bits will define the baud rate configured for the device.
(b) A serial device high-level driver function *write* (5, *length*1, *memTxaddress*) will send bytes into TRH one by one and the device transmits total bytes = *length*1 from addresses memTxAddress to memTxAddress + *length*1 – 1.
(c) A serial device high-level driver function *read* (5, len2, memRxaddress) will receive bytes through RBR and the device receives the bytes one by one and len2 number of bytes are put in the buffer at memory address from memRxaddress to memRxaddress + len2 – 1.
(d) A serial device high-level driver function *close* (5) will close the device. It can then be reused only after opening it by open ( ). The device closes by masking the device interrupts.

4.9.5 Device Drivers for Internal Programmable Timing Devices

Generally, there is at least one hardware timer T as an internal device in any systems needing functions related to timers. Using the time-outs (ticks) from T (using overflow interrupts) several needed software timers (SWTs) as can also be driven.

Example 4.20

A given hardware timer time-outs every 2^{16} (16384) counts and let the timer clock give input at every 2 μs. Assume that an SWT is to be programmed to tick every 31 × 32.768 μs = 1.015808 s. SWT should interrupt after swt counts *swtcnt* becomes equal to *numTicks* (preset number of ticks). The SWT is first initialized to *swtcnt* equals 0 and on every T overflow an ISR increments *swtcnt* and when *swtcnt* equals 31 then *swtcnt* is reset to 0 after generating software interrupt by an SWI instruction. An SWT-ISR then executes to perform required actions on SWT interrupt.

Timer device driver function call an ISR. The ISR programming needs an understanding of the programming of each bit of timer control register(s) and status register(s). An important step is programming of each bit of one or two control registers present and the use of status register. The programmer must also take into account the following. (i) Instead of interrupt enable, a device may have a mask bit. Mask bit means interrupt disables on set and enables on reset. Its actions are opposite to that of the enable bit. The programmer must also remember that a certain interrupt cannot disable (cannot mask, NMI). These enable or mask bits are the secondary bits. There is an overall interrupt system enable bit, which is like a master key (primary-level bit) for all maskable interrupt sources. The driver must set that bit also.

Step I: Write in a register that holds the timer maximum count value, the number of count inputs, *numTicks* for the SWT.

Step II: Write in status register the timer status flag(s) equal to reset [in case the device does not reset flag(s) automatically on a read of the status flag(s)].

Step III: Write each bit present in the control register(s). Write interrupt secondary- and primary-level enable bits equals true in control register, write other bits according to their uses. It is essential to write the device enable bit to let the device work. Definition of each bit in the mode register, if present is also essential.

Assume that a free running counter (FRC) is used as a timing device. Device-driver ISR programming steps require use of 68HC11 RTC described in Section 3.8. Consider following example. (68HC11

microcontroller knowledge is presumed here). The example gives the details of bits that are initialized for using FRC device of 68HC11.

Example 4.21

1. *Step I*: Define the output compare register(s) to hold count instance(s) of the FRC when OC flag(s) sets and OC interrupt(s) occurr(s).
2. *Step II*: Flag(s) on its read from status register must be reset in case the device does reset automatically. The flags that may be present are the FRC overflow status flag, OC flag(s), ICAP_F flag(s), RTC flag and SWT flag(s). These are to be reset on a read of the status register.
3. *Step III*: Define control register(s) bits. Here, definition for each bit present is essential. The bits may be as follows. Prescaling bits for count input clock, overflow interrupt enable bit, RTC interrupt enable, OC interrupt enable bit(s), OC enable bit(s), OC output level bit(s), ICAP enable bit(s), ICAP input edge bit, ICAP input bit(s), ICAP interrupt(s) enable bit, SWT enable bits and SWT interrupts enable bit(s).
4. *Step IV*: Also enable the primary level interrupt enable bit, if already not enabled.

4.9.6 Linux Internals as Device Drivers and Network Functions

Drivers for port, keypad, display, timer and network devices (Sections 3.3, 3.6 and 3.9) are most commonly used in the systems. Drivers for PCS (physical coding sublayer) and PMA (physical media attachment) are required in media devices for most voice and video systems. It becomes impractical for a programmer to write the codes for each function of device. For commonly used devices, a programmer most often relies on drivers that are readily available in the thoroughly tested and debugged operating system (Refer to Chapters 9 and 10 for µCOS II, VxWorks OS, Windows CE, OSEK and Real Time Linux).

The Linux operating system is a tested and debugged operating system and is used throughout. It has a large number of drivers (Table 4.2) that are, moreover, in the public domain. Public domain means non-proprietary and usable by anyone. A programmer may therefore choose Linux drivers when the embedded system being designed has the devices that have the drivers available in Linux (refer http://www.linuxdoc.org).

Linux has internal functions called Internals. Internals exist for device drivers and network-management functions. Examples of useful Linux drivers for the embedded system are given in Table 4.2.

Linux internal functions exist for sockets, handling of socket buffers, firewalls, network protocols (e.g., NFS, IP, IPv6 and ethernet) and bridges. These are in the *net* directory. They work separately as drivers and also form a part of the network management function of the OS. (The reader can refer a standard textbook for bit-wise meaning of UDP, PPP and SLIP and for socket functions, firewall and network protocols. For example, refer *Internet and Web Technologies* from Tata McGraw-Hill, 2002 for bit-wise description of PPP, SLIP, TCP, IP, ethernet and other protocols).

Device drivers play a key role in most embedded system as these provide software layers between the application and devices. Drivers control almost all devices except the memory devices and the processor in a system. Linux device drivers are also used popularly because they are tested and debugged and are in the public domain. The Linux OS has internals and a large number of readily available device drivers for the most common physical and virtual devices and has the functions for the network sockets and protocols.

Table 4.2 Useful Linux Device Drivers

Driver type	*Explanation*
char	Drivers for char devices. A char device is a device for handling a stream of characters (bytes).
block	Drivers for block devices. A block device is a device that handles a block or part of a block of data. For example, 1 kB of data handled at a time. (*Note*: Unix block driver does not facilitate use of a part of the block during read or write.)
net	Drivers for network devices. A net device is a device that handles network interface device (card or adapter) using a line protocol, for example, tty or PPP or SLIP.
input	Drivers for the standard input devices. An input device is a device that handles inputs from a device, for example, keyboard.
media	Drivers for the voice and video input devices. Examples are video-frame grabber device, teletext device, radio-device (actually a streaming voice, music or speech device).
video	Drivers for the standard video output devices. A video device is a device that handles the frame buffer from the system to other systems as a char device does or a UDP network packet sending device does.
sound	Drivers for the standard audio devices. A sound device is a device that handles audio in a standard format.
system	Platform-specific drivers. Recently, system processor-specific drivers have also become available in this operating system. Examples are drivers for an ARM processor-based system.

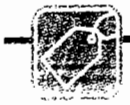
Summary

The following is a summary of the important points that were discussed in this chapter.

- Interrupt means event, which invites attention of the processor for the action of hardware. Event can be a hardware or software event. In response to the interrupt, running routine or program interrupts and a service routine executes.
- When a device or port is ready, a device or port generates an interrupt or when it completes the assigned action, it generates an interrupt. These interrupts are called hardware interrupts. When software run-time exception condition is detected, either processor hardware or a software instruction SWI generates an interrupt for *exception*. An SWI instruction is *INT* n in 80x86.
- Device, run-time error and software instruction-related interrupts are studied. Various possible sources of the software and hardware interrupts are listed.
- Device driver functions execute software-interrupt routines for servicing the device, and drive a peripheral or internal device by create, open, read, write, close or other device function. A device is configured and initialized by using the control bits at its control register(s). The device driver executes on hardware or software interrupts as per set flags in status register(s).
- Physical device drivers, virtual device drivers and ISRs for the software instruction, software-defined condition and error condition are used to program the system.
- Every system has an interrupt service mechanism.
- We learnt *device initialization, driver ISR and function coding* for the parallel ports, serial-line UART and internal timing device in 68HC11.
- Virtual devices like char device, block device or file device, RAM disk, socket, pipe, loop back device are used during programming. These are treated in a way analogous to the physical devices.
- There are the device interrupts as well as other interrupt sources, driver functions and ISRs for which must be written by the programmer. A list of the various possible sources of software and hardware interrupts is given. A

software instruction or a condition during running-related or run-time error-related or device driver function interrupts are important in the systems.

- Interrupt system and individual device interrupt enabling and disabling, interrupt vectors, interrupt pending registers and status registers, non-maskable, maskable, non-maskable only when so defined within a few clock cycles after reset.
- Each running program has a *context* at each running code instance. Context means a CPU state (PC, stack pointer(s), registers and program state (variables that should not be modified by another routine). The context must be saved on a *call* to another ISR or *task* or routine. It must be done before processor switching to another context. Processor switches to another context by retrieving called program context. Certain processors like those from the ARM family provide for fast context switching. These have the internal stack frames or sets of local registers for the contexts. Fast context switching reduces the interrupt latencies and enables the meeting of each real-time task deadline. The OS program provides for memory blocks (allocates the blocks) to be used as stack frames in a multitasking system.
- Programming should be such that interrupt latencies are made as short as possible. This helps in meeting the deadlines for each interrupt service. Use of interrupt service threads (slow-level ISRs initiated by SWIs) helps in having the main first-level ISR codes short. The use of a DMA channel facilitates the small interrupt latency periods of an IO interrupt source requiring bulk or burst data transfers.
- There may be simultaneous service demands from multiple sources. Assignment of software priorities among the multiple sources of interrupts keeping in view the available hardware priorities is essential.
- Linux has a large number of device drivers, which are open-source.



Keywords and their Definitions

Context : PC, stack pointer as well as the program status word and processor registers for a foreground program or ISR or task. It can also include memory block addresses allotted to the program or routine.

Context switching : Saving the foreground program [interrupted routine (or function)] context and retrieving or loading the new context of the called routine. The time taken in context switching is included in the interrupt latency period.

Deadline : A period during which service to an interrupt must start.

Device attaching (adding) : Configuring a device and enabling the use of its driver.

Device detaching (removing) : Disabling the use of a device driver by the system.

Device driver ISR codes : Codes for read and write or other operations at the device addresses after reading device status on interrupt.

Device initialization codes : Codes for programming the control register of a device.

Device opening : Resetting the device control bits and preparing it for the use of its driver.

Device closing : Resetting the device control bits and its next time use is then possible only by opening it again.

Exception : An interrupt on detection of a run-time event during computations or communication. Setting of a condition that may be defined by the programmer.

Exception handler : Programmer defines the exception handler ISR also for handling service for that condition. Error conditions are handled by the exception handlers. Exception handler is called on executing an SWI instruction.

Foreground program : Foreground program is one that is executed when no interrupt call is being serviced.

Hardware-assigned Priority : Priority assigned by the processor itself to service a source when several interrupts need the interrupt service.

Hardware Interrupt : Interrupt of devices or ports at the system.

Hardware timer : A timer present in the system as hardware and which gets inputs from the internal clock with the processor. A device-driver program programs it like any other physical device.

Interrupt : CPU on interrupt event may initiate a further action by vectoring to a vector address and calling an ISR or else it continues with the current process (task) if the interrupt is disabled or masked.

Interrupt enable bit : It enables (unmasks) the interrupts from a source(s).

Interrupt flag : A register bit for a Boolean variable that sets to reflect a need for executing an ISR. It resets when corresponding ISR starts executing.

Interrupt latency : A period for waiting for service after a service demand is raised (source status flag sets).

Interrupt mask bit : When this bit is reset (= false) the request for initiation of interrupt service is responded, otherwise it is not responded.

Interrupt pending register : A register to show the interrupt sources or source groups from various devices that are pending for service by executing the corresponding ISRs. It is a '*read and write*' register. A bit auto resets in it when the corresponding interrupt service starts. A user instruction can also reset a bit in the register.

Interrupt service mechanism : A mechanism for interrupt-driven service of the devices and ports. It saves the processor waiting time, because it lets the processor process the multiple devices and virtual devices. The mechanism also sets the priorities and provides for enabling and disabling the services.

ISR : A program that is executed on interrupt after saving the necessary parameters called context onto the stack so that the same can be retrieved on return from the routine last instruction. An ISR is also a device driver ISR when a high level language device driver function executes SWI and it services a device-interrupt. An ISR is also a trap when it services a software error or other condition-related interrupts with error detected by processor hardware. An ISR is also called *exception handler* on *exception*, which is *thrown* when it services software run-time condition detection and the condition is detected in the software routine. It is also called *signal* handler. When a *signal* function is called in a program. Each signal or exception throwing function executes an SWI, which initiates an ISR.

Interrupt vector : A memory address where there are bytes to provide the corresponding ISR address. The system has the specific vector addresses assigned by the hardware for each interrupting source for each internal device.

Interrupt vector table : A table for the interrupt vectors in the memory. The table facilitates the service of the multiple interrupting sources or source-groups for each internal device. In each row for an interrupt vector address, there are bytes to provide the corresponding interrupt service routine address.

Linux : An open source OS. It has a large number of device drivers and network management functions.

Linux device drivers : Device drivers taken from the Linux source.

Maskable interrupt source : A source, service routine call for which can be disabled or service of which masked.

Non-maskable interrupt source : A source, which cannot be disabled and which is used for the highest priority interrupt service cases, like RAM parity error.

Polling : A method to find the status of a peripheral or device. It is also a method by which at the end of an instruction or at the end of an ISR, the pending interrupts are searched by the processor from the status register or interrupt pending register to service the one with the highest priority.

Primary-level enable bit	:	A bit, which enables or disables any service on interrupt by all the maskable sources. It helps in executing critical section codes and preventing service to any other maskable source during disabling of service.
Secondary-level mask bit	:	It disables service from an individual source or source group.
Signal	:	Signal is function to initiate software-interrupt on an instruction to call a software-initiated ISR. An *exception* or trap may also be called a *signal*. Signal is called by SWI instruction in ARM and INT n in 80x86.
Software-assigned priority	:	(Hardware signal is different.) A priority for a source or source group. It is defined at a register called interrupt priority register. When several interrupts occur at the same time, software-assigned priorities over-ride the hardware priorities.
Software Interrupt	:	An interrupt by an error condition trap or illegal opcode or an SWI instruction (or INT n instruction 80x86) in a routine or ISR or software timer or signal.
Stack frame	:	A set of registers or a memory block that stores the context for a program or ISR.
Status register	:	A read only register for a device to set a flag on arising of an interrupt. A user instruction can also reset a bit in it. If a device has a number of sources the status register has a number of flags and a distinct source for each source. When it is read by a processor instruction, the flag resets.
Trap	:	An interrupt on detection by hardware a run-time computational or other event. The processor may also signal an exception on the *trap*. Example of a trap is division by zero in 80x86.
Virtual device	:	A device, which emulates the physical device and drives by virtual device drivers provided in an operating system. Examples are file, pipe, socket, etc.
Worst-case latency	:	Maximum interrupt latency found in the worst possible case.

Review Questions

1. What are the disadvantages and advantages of *busy and wait transfer* mode for the I/O devices?
2. What are the advantages and disadvantages of interrupt-driven data transfer?
3. What are the advantages of DMA based or peripheral-transcation-server based data transfer over the interrupt-driven data transfer?
4. How is the vector address used for an interrupt source?
5. Interrupt vector addresses are prefixed in the interrupt mechanism for the known internal peripherals in a microcontroller. How are the vector addresses assigned for exceptions and user-defined interrupts?
6. interrupt mechanism in each processor differs from a processor family to another. Explain, why the device drivers are processor-sensitive programs.
7. How do you intialize and configure a device? Take an example of serial-line driver at COM port of PC.
8. How is a *file* at the memory act handled as a device?
9. What are the advantages of RAM disk?
10. Make a list of Linux internal *net* directory functions for sockets, handling of socket buffers, firewalls, network protocols (e.g., NFS, IP, IPv6 and ethernet) and bridges. Why are these device drivers assigned in a separate directory of network management function of Linux OS..
11. Define *context*, *interrupt latency* and *interrupt service deadline*.
12. Why is the context switching in an embedded processor faster than saving the pointers and variables on the stack using a stack pointer? How does the context switching time reduce in processor architectures for embedded systems?

13. How is the context switching handled in ARM7?
14. DMA helps in reducing the processor load by providing direct access for the IOs. How does it help in faster task execution in a multi tasking system by the reduced interrupt service latencies?
15. What do you mean by throwing an exception? How is the exception condition during execution of a function (routine) handled?
16. How do the device driver functions and ISRs differ? How do the ISR calls differ in 80x86 and 8051?
17. How do you assign service priority to the multiple device drivers of a system? How do you assign priorities to the timer devices and ADC device?
18. What are the uses of hardware-assigned priorities in an interrupt service mechanism?
19. What are the uses of software-assigned priorities in an interrupt service mechanism?
20. How is break point interrupt important for debugging embedded software?
21. What do you mean by POSIX function?

Practice Exercises

22. How do you write device driver? List the steps involved in writing a device driver.
23. Search web and design a table to show features in device driver modules of embedded Linux OS. Explain with examples each a char device, block device and block device configurable as char device. UART is a char device. Why is it a char device?
24. Give software-related interrupt examples. What are the interrupts in 8086, which generate software error?
25. Show the state machine-generated states in a key marked as number 4 in the mobile device. How will you use the SWI instruction to generate an SMS message in a mobile phone having a T9 keypad?