use of interrupt stack to store the link reg.
- subtracts an offset from link reg. & then stores it onto interrupt stack.

handler

```
SUB R14, R14, #4          ; R14=R14-4
STMFD R13!, {R0-R3, R14}  ; store context
---
<handler code>
---
LDMFD R13!, {R0-R3, PC}^  ; return
```

^ forces cpsr to be restored from spsr.

Interrupts : 2 types on ARM processor

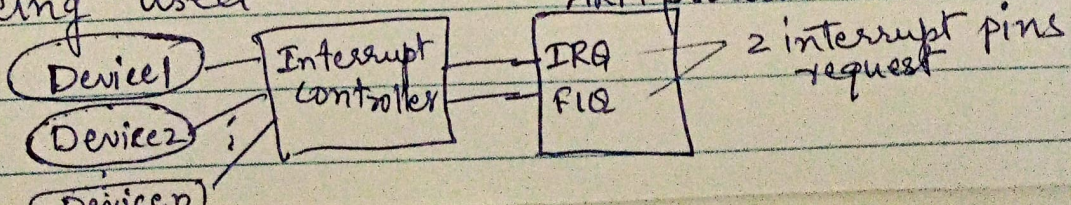
① causes exception raised by external peripheral IRQ & FIQ.

② causes an exception — SWI instr

Both types suspend the normal flow of a program

Assigning interrupts :-

A system designer can decide which hardware peripheral can produce which interrupt request.
- This decision can be implemented in h/w or s/w (or both) & depends upon the embedded system being used.



An interrupt controller connects multiple external interrupts to one of the two ARM interrupt requests

System designer adopt a standard design practice while assigning interrupts

- Software Interrupts : reserved to call priviledged operating systems routines
  Ex:- It can be used to change a pgm running in user mode to privileged mode

- Interrupt Request : assigned for general purpose interrupts. IRQ has lower priority & higher interrupt latency
  Ex:- a periodic timer interrupt to force a context switch.

- Fast Interrupt Requests : reserved for a single interrupt source that requires a fast response time
  Ex:- DMA specifically used to move blocks of mem.

Interrupt Latency: The interval of time from an external interrupt request signal being raised to the first fetch of an instr of a specific ISR.

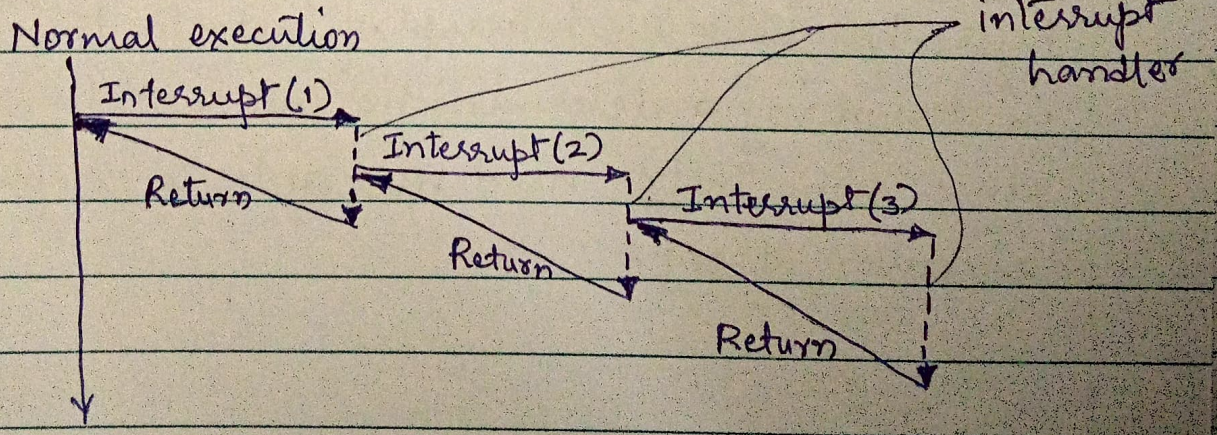
Interrupt Latency depends on a combination of h/w & s/w.

System architects must balance the system design to handle multiple simultaneous interrupt sources & minimize interrupt latency

If the interrupts are not handled in timely manner, then the system will exhibit slow response time.

Two main methods to minimize interrupt latency

① use a nested interrupt handler – which allows further interrupts to occur even when currently servicing an existing interrupt.



Once a nested interrupt has been serviced, then the control is relenquished to original ISR

(2) 2nd method involves prioritization.
You program the interrupt controller to ignore interrupts of same or lower priority than the interrupt you are handling, so that only a higher priority task can interrupt your handler. You then at the end reenable all disabled interrupts.

IRQ & FIQ Exceptions :
when interrupts are masked in cpsr register then these will not be executed.

IRQ & FIQ exceptions only occur when a specific interrupt mask is cleared (ie interrupt is enabled) in the cpsr. The ARM processor will continue executing the current instr in the execution stage of the pipeline before handling the interrupt. [important factor in designing a deterministic interrupt handler since some instrs need more cycles to complete execution stage].

IRQ or FIQ exception causes the processor h/w to go through a standard procedure (provided the interrupts are not masked):

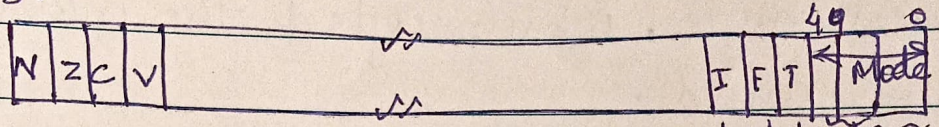
1. The processor changes to a specific interrupt request mode, which reflects the interrupt being raised.
2. The previous mode's cpsr is saved into spsr of the new interrupt request mode.
3. The pc is saved in the lr of the new interrupt request mode.
4. Interrupt(s) are disabled - either the IRQ or both IRQ & FIQ exceptions are disabled in the cpsr. This immediately stops another interrupt request of the same type being raised.
5. The processor branches to a specific entry in the vector table.

The procedure varies slightly depending upon the type of interrupt being raised.

Ex:- what happens when IRQ exception is raised when processor is in user mode.

The processor starts in state 1. Here both IRQ & FIQ exception bits in cpsr are enabled.

cpsr

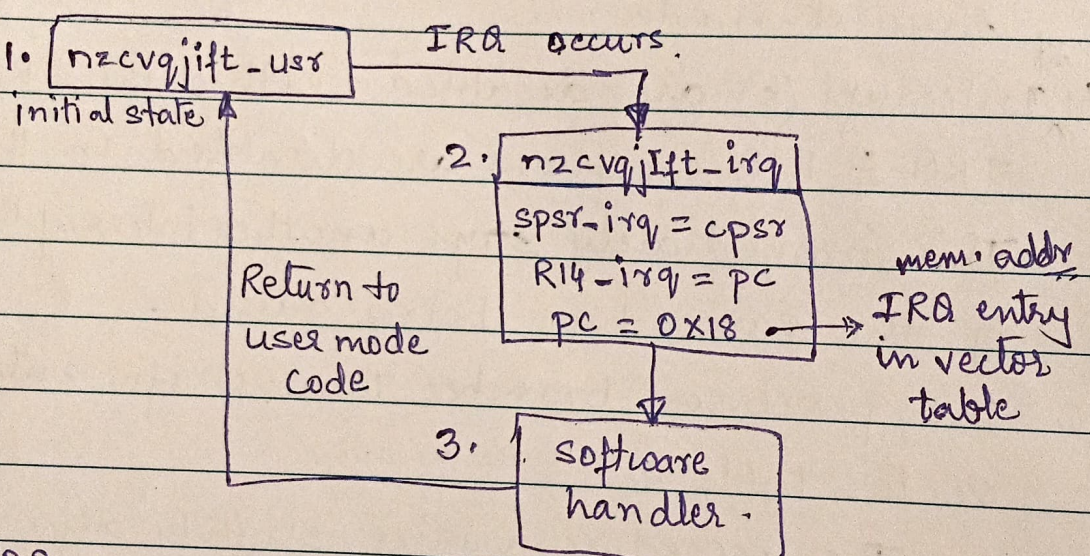


if I = 1 disables IRQ interrupts
F = 1 disables FIQ interrupts

interrupt masks

Thumb state

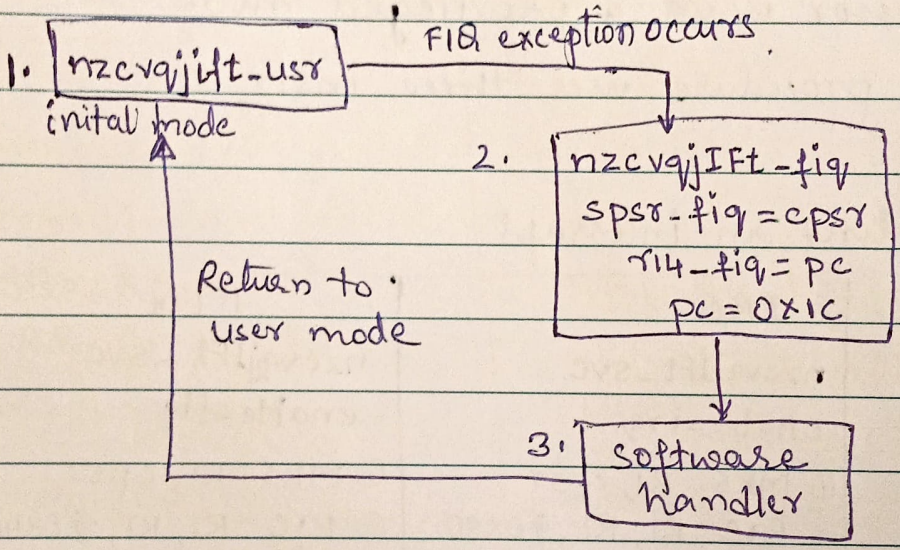
processor mode



* when IRQ occurs, the processor moves into state 2. This transition automatically sets IRQ bit to 1, disabling any further IRQ exceptions.
* FIQ however remains enabled because FIQ has higher priority ∴ does not get disabled when a low-priority IRQ exception is raised.
* The cpsr processor mode changes to IRQ mode
* The user mode cpsr is automatically copied into spsr_irq.

r14-irq register is assigned the value of pc when interrupt is raised.
The pc is then set to IRQ entry + 0x18 in vector table
In state 3 the s/w handler takes over & calls the appropriate ISR to service the source of interrupt
Upon completion, the processor mode reverts back to original user mode code in state 1.

FIQ exception occurs; when processor is in user mode.



* The processor goes thro' similar procedure as with IRQ exception, but the processor masks out further IRQ exceptions & FIQ exceptions [∴ IF in cpsr]
This means that both interrupts are disabled when entering s/w handler in state 3.
* changing to FIQ mode means there is no requirement to save registers R8 to R12 since these registers are banked in FIQ mode.

These registers can be used to hold temporary data such as buffer pointers or counters. This makes FIQ ideal for servicing a single source high priority, low Latency interrupt.

Enabling & Disabling FIQ & IRQ exceptions

* ARM processor core has a simple procedure to manually enable & disable interrupts which involves modifying the cpsr when the processor is in a privileged mode.
* The procedure uses three instr s

Enabling an interrupt

cpsr value	IRQ	FIQ
Pre:	nzcvqjIFt_SVC enable_irq	nzcvqjIFt_SVC enable_fiq
	① MRS R1, cpsr ② BIC R1, R1, #0X80 ③ MSR cpsr_c, R1	① MRS R1, cpsr ② BIC R1, R1, #0X40 ③ MSR cpsr_c, R1
Post:	nzcvqjiFt_SVC	nzcvqjIft_SVC
Mask pattern:	0X80 = 10000000 (cpsr; I = 1 = I, F, Thumb, mode)	0X40 = 01000000 (I, F, Thumb, mode)
BIC →	Logical bit clear (AND NOT) of 2 32-bit values	
AND { cpsr / mask	00000000 10000000 (nzcvqjIFt mode) 00000000 00000000	00000000 10000000 (nzcvqjIFt mode) AND mask 01000000
then complement that bit	10000000 NOT 01111111	NOT 01000000 10111111

nzcvqj I F t ⇒ both = 1 ie masked
= 0 ⇒ not masked ie enable

① 1st instr MRS copies content of cpsr into R1
② 2nd instr clears IRQ or FIQ mask bit. [only that bit BIC works which unmasks ie enables it without affecting the other bits]
③ 3rd instr copies updated contents in R1, thus enabling the interrupt request.

* Disabling of FIQ is used only with critical section of your code.

Disabling an interrupt:

cpsr	IRQ	FIQ
Pre:	nzcvqjift_SVC	nzcvqjift-SVC
Code	disable_irq	
	MRS R1, cpsr	MRS R1, cpsr
	ORR R1, R1, #0x80	ORR R1, R1, #0x40
	MSR cpsr_c, R1	MSR cpsr_c, R1
Post:	nzcvqjIft_SVC	nzcvqjiFt-SVC

* The interrupt request is either enabled or disabled only when the MSR instr has completed the execution stage of the pipeline.

But in between ie before completion of execution stage interrupts can still be raised or masked,
∴ to disable or enable both IRQ & FIQ, the ORR or BIC, the mask pattern is 0xC0.

Interrupt handling schemes:-

There are many schemes

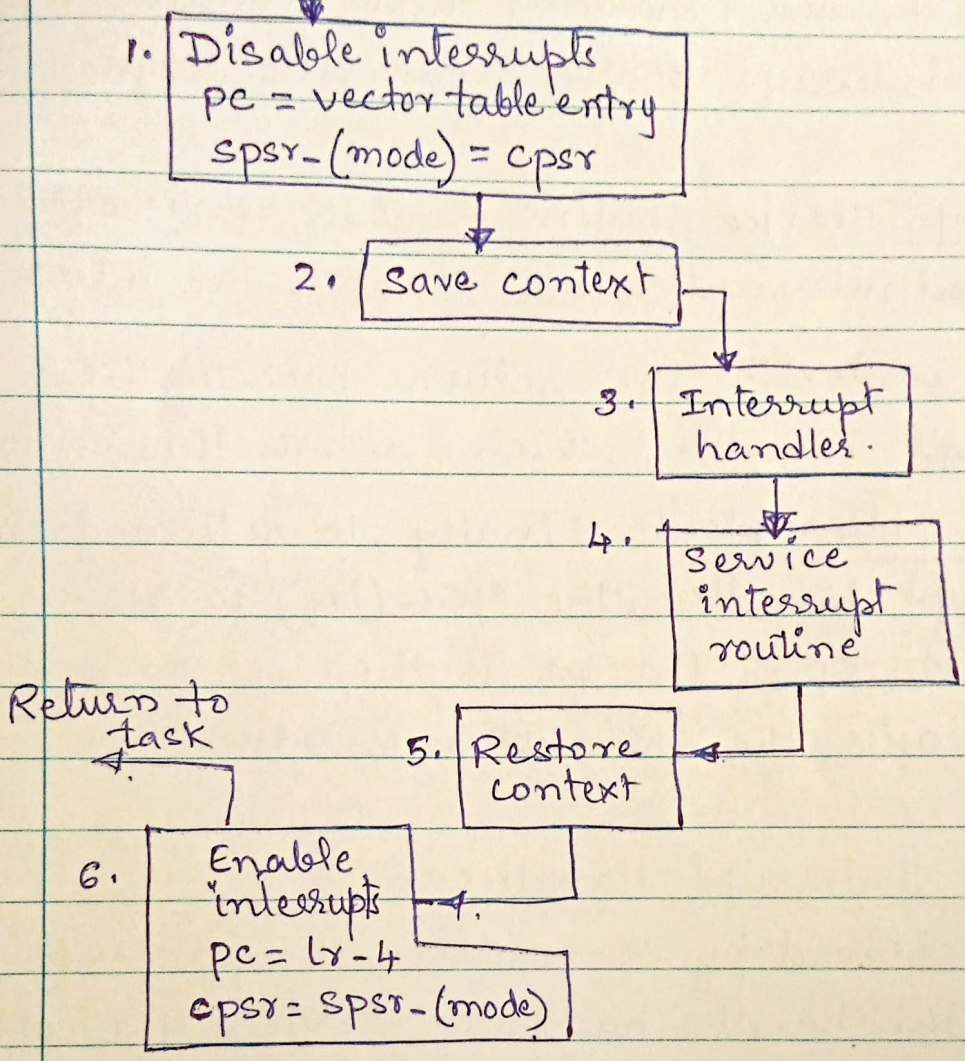
1. Nonnested interrupt handler
2. Nested interrupt handler
3. Reentrant " "
4. Prioritized simple " "
5. Prioritized standard " "
6. Prioritized direct " "
7. Prioritized grouped interrupt handler
8. VIC PL190 based interrupt service routine
   (vectored Interrupt Controller)

1. Non-nested interrupt handler:-

- the interrupts are disabled until control is returned back to the interrupted task or process
- It can service a single interrupt at a time, ie this form is not suitable for complex embedded systems which require servicing of multiple interrupts with differing priority levels.

- the various stages which occur when an interrupt is raised in a system:

simple nonnested interrupt handler



① **Disable interrupts :-**

- when IRQ exception is raised, ARM will disable further IRQ exceptions from occuring.
- Processor mode is set to appropriate interrupt request mode & previous cpsr is copied into spsr_(irq)
- Processor then sets pc to point to correct entry in vector table & execute instr.

(2) Save context : On entry the handler code saves a subset of the current processor mode nonbanked regs.

(3) Interrupt handler:- handler then identifies the external interrupt source & executes appropriate ISR.

(4) Interrupt service routine :- ISR services the external interrupt source & resets the interrupt.

(5) Restore context:- ISR returns back to the interrupt handler, which restores the context.

(6) Enable interrupts :- Finally, to return from interrupt handler, the spsr-(irq) is restored back into cpsr. The pc is then set to next instr after the interrupt was raised.

Nested Interrupt Handler :-

- It is acheived by re-enabling the interrupts before the handler has fully serviced the current interrupt.
- For a real time system this feature increases the complexity of the system but also improves its performance.
- this additional complexities introduces the possibilities of subtle timing issues which can cause system failure.

Nested interrupt handler (contd.)

- the problems introduced due to increase in complexity is overcome by protecting the context restoration from interruption, so that the next interrupt will not fill the stack.

main pgm.
=
=
- interrupt (INTR1) occurs.
  - µP checks IVT gets ISR1 addr.
  - pc value(main) is stored
  - ISR1 starts
    Again interrupt occurs (say ISR6)
    - µP checks IVT gets ISR6 addr.
    - Save the pc value (ISR1)
      - ISR6 is executed.
        once completed upon return gets back pc value (ISR1)

Now ISR1 is executed
once completed, pc value (main) is got back into pc, then remaining instrs in main pgm are executed.

=
=
end (main)

Nested Interrupt handler :-