Διακοπές (Interrupts)

K.Harteros – G.Kornaros

Περίγραμμα

- Interrupts
  - ✓ Διαφορετικά είδη interrupts στον επεξεργαστή ARM
- Καθυστέρηση των Interrupt
- Interrupt handlers
  - ✓ Διαφορετικά είδη interrupt handlers
- Θέματα συγχρονισμού με τους interrupt handlers

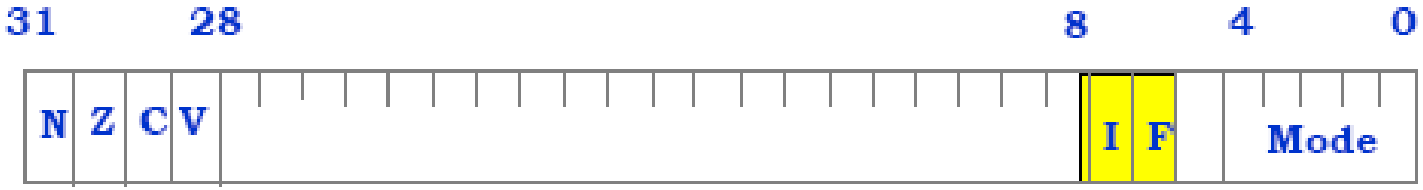
Events

- Two types of events
  - ✓ Planned (key pressed, timer periodical interrupt, software interrupt) called Interrupts
  - ✓ Unplanned (data abort, (undefined) instruction abort), called exceptions
- Manage events
  - ✓ the normal flow of execution from one instruction to the next is halted and re-directed to another instruction that is designed specificallyto handle that event
  - ✓ Once the event has been serviced the processor can resume normal execution by setting the program counter to point to the instruction after the instruction that was halted
  - ✓ IRQ goes high
    - o synchronous with clock
    - o Asychronous with clock

Interrupts στον επεξεργαστή ARM

- Πολλά είδη exceptions (interrupts)
  - ✓ SWIs: Reserved για κλήση ρουτινών του Λειτουργικού Συστήματος (supervisor mode)
  - ✓ IRQ/FIQ: προερχόμενα από ένα εξωτερικό περιφερειακό που απαιτεί προσοχή - εξυπηρέτηση
- IRQ (Interrupt Request)
  - ✓ General-purpose interrupts
  - ✓ Παράδειγμα: periodic timer interrupt to force a context switch
- FIQ (Fast Interrupt Request)
  - ✓ Reserved για μια πηγή interrupt που απαιτεί γρήγορο χρόνο απόκρισης
  - ✓ Παράδειγμα : direct memory access to move blocks of memory
- Σύγκριση των δύο:
  - ✓ Τα IRQs έχουν μικρότερη προτεραιότητα από τα FIQs
  - ✓ Τα IRQs έχουν μεγαλύτερη καθυστέρηση διακοπής από τα FIQs

Program Status Registers (CPSR & SPSR)



- Interrupt Disable bits
  - ✓I = 1, disables the IRQ
  - ✓F = 1, disables the FIQ
- Mode bits: processor mode

M[4:0]	Mode
10000	User
10001	FIQ
10010	IRQ
10011	SVC
10111	Abort
11011	Undef
11111	System

Προτεραιότητες εξαιρέσεων

Exceptions	Priority	I bit (1⇨IRQ Disabled)	F bit (1⇨FIQ Disabled)
Reset	1 (highest)	1	1
Data Abort	2	1	
Fast Interrupt Request (FIQ)	3	1	1
Interrupt Request (IRQ)	4	1	
Prefetch Abort	5	1	
Software Interrupt	6	1	
Undefined Instruction	6 (lowest)	1	

Χειρισμός IRQ και FIQ

IRQ handling

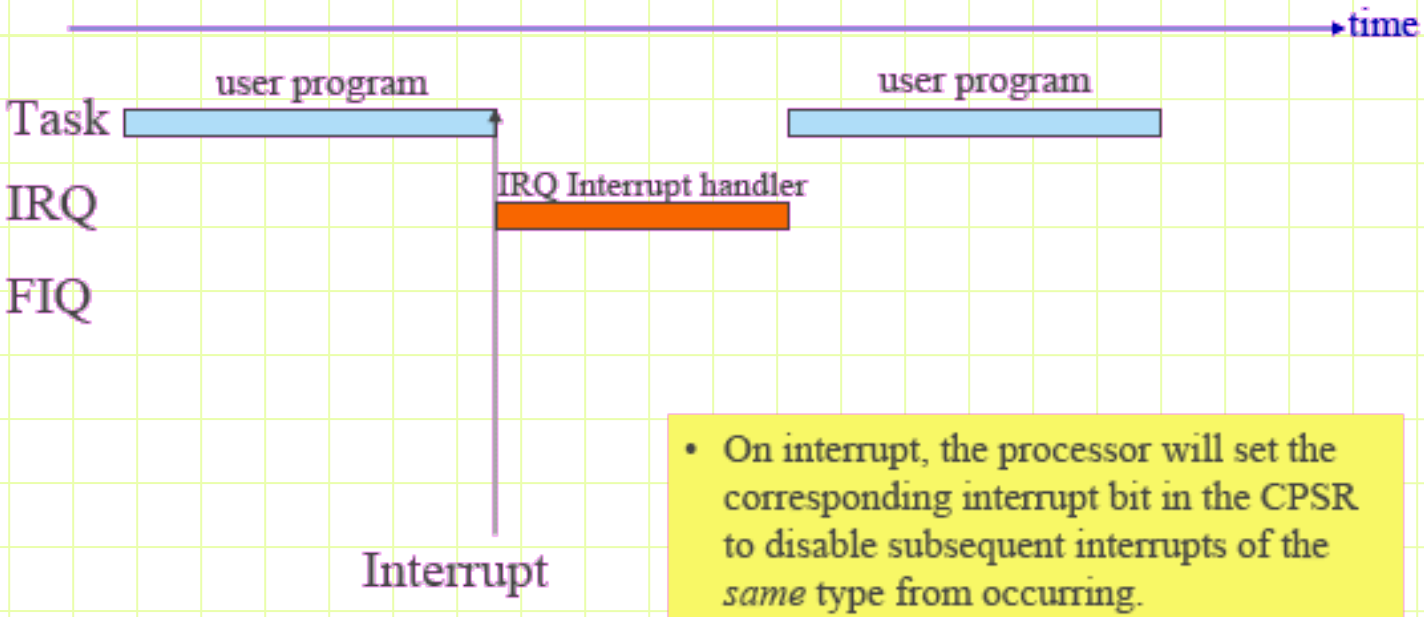
- Όταν συμβαίνει ένα IRQ τότε ο επεξεργαστής:
  - ✓ Copies CPSR into SPSR_irq
  - ✓ Sets appropriate CPSR bits
- Sets mode field bits to 10010
- Disable further IRQs
  - ✓ Maps in appropriate banked registers
  - ✓ Stores the “return address” in LR_irq
  - ✓ Sets PC to vector address 0x00000018
- To return, exception handler needs to:
  - ✓ Restore CPSR from SPSR_irq
  - ✓ Restore PC from LR_irq
  - ✓ Return to user mode

FIQ handling

- Όταν συμβαίνει ένα FIQ τότε ο επεξεργαστής:
  - ✓ Copies CPSR into SPSR_fiq
  - ✓ Sets appropriate CPSR bits
- Sets mode field bits to 10001
- Disable further IRQs and FIQs
  - ✓ Maps in appropriate banked registers
  - ✓ Stores the “return address” in LR_fiq
  - ✓ Sets PC to vector address 0x0000001c
- To return, exception handler needs to:
  - ✓ Restore CPSR from SPSR_fiq
  - ✓ Restore PC from LR_fiq
  - ✓ Return to user mode

Interrupt Handlers

- Όταν συμβεί ένα interrupt, ο επεξεργαστής θα εξυπηρετήσει το interrupt εκτελώντας έναν interrupt handler (επίσης ονομάζεται ρουτίνα εξυπηρέτησης interrupt (ISR))



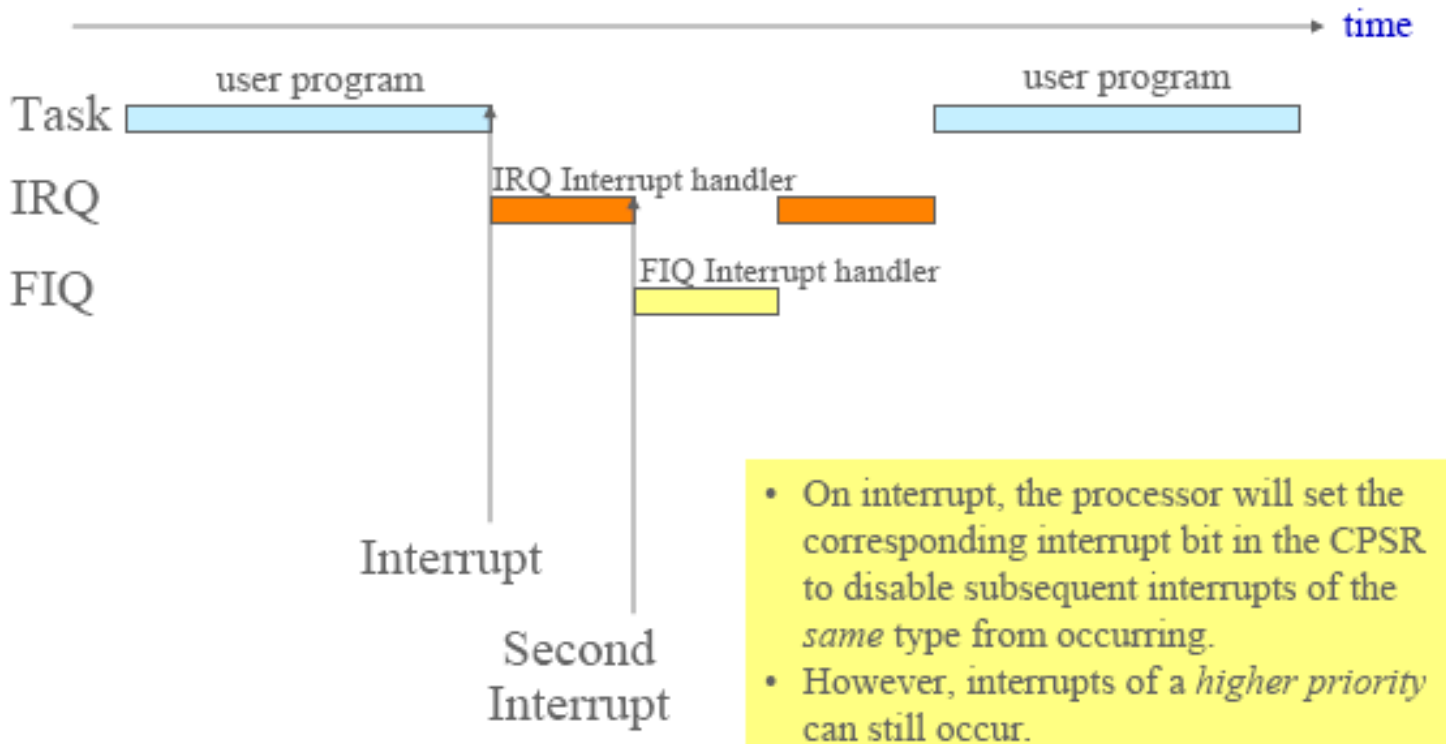
- On interrupt, the processor will set the corresponding interrupt bit in the CPSR to disable subsequent interrupts of the *same* type from occurring.
- However, interrupts of a *higher priority* can still occur.

Καθυστέρηση των Διακοπών

- Καθυστέρηση διακοπής
  - ✓ Το χρονικό διάστημα από τη στιγμή που ένα εξωτερικό σήμα ζητάει interrupt-request έως το πρώτο fetch μιας εντολής ενός συγκεκριμένου ISR
- Επηρεάζεται από τον αριθμό των ταυτόχρονων πηγών interrupt που πρέπει να χειριστεί.
- Παράγοντες που επηρεάζουν την καθυστέρηση ενός interrupt:
  - ✓ Time taken to recognize the interrupt
  - ✓ Time taken by the CPU to complete its current instruction (depends on whether the CPU is executing a multi-cycle or a single-cycle instruction)
  - ✓ Time taken by the CPU to perform a context switch (switching in banked registers, saving program counter, etc.)
  - ✓ Time taken to fetch the interrupt vector
  - ✓ Time taken to start the interrupt service routine executing

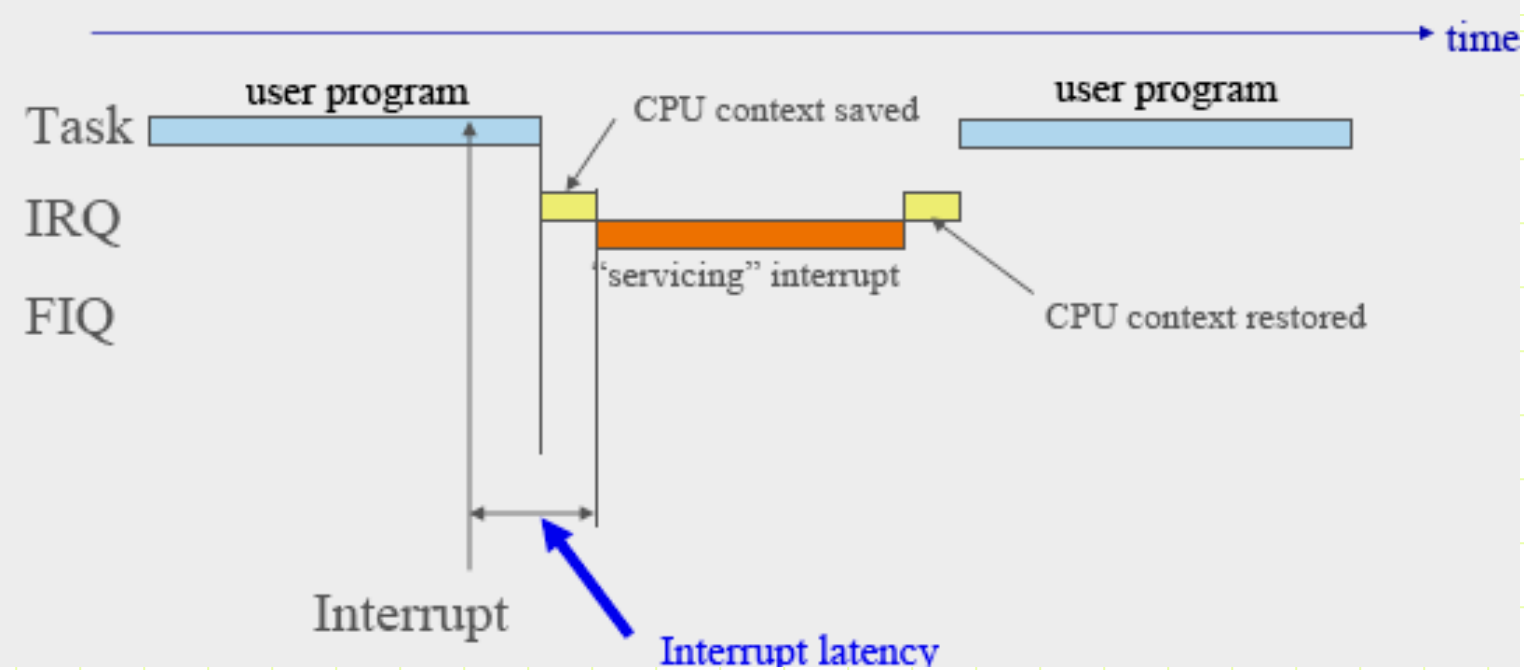
FIQs Within IRQ Servicing

- Interrupts μπορούν να συμβούν ενδιάμεσα στα interrupt handlers



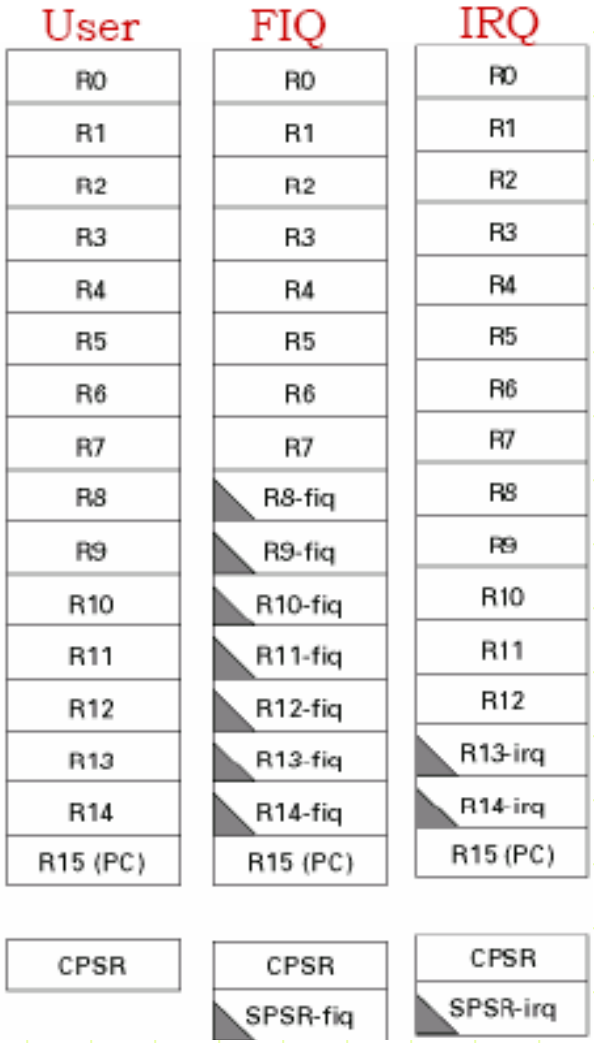
Θέματα χρονισμού στα Interrupts

- Προτού μία ISR κάνει οτιδήποτε, πρέπει να αποθηκεύσει τους καταχωρητές του τρέχοντος προγράμματος (if the ISR overwrites those registers)
- Γι'αυτό το λόγο το FIQ έχουν πολλούς extra registers – για να ελαχιστοποιηθεί το overhead να αποθηκευτεί το περιβάλλον του CPU



Γιατί είναι πιο γρήγορα τα FIQs

- Τα FIQs είναι ταχύτερα από τα IRQs όσον αφορά την καθυστέρηση του interrupt
- FIQ mode has five extra registers at its disposal
  - ✓ No need to save registers r8 – r12
  - ✓ These registers are banked in FIQ mode
  - ✓ Convenient to store status between calls to the handler
- FIQ vector is the last entry in the vector table
  - ✓ The FIQ handler can be placed directly at the vector location and run sequentially starting from that location
- Cache-based systems: Vector table + FIQ handler all locked down into one block



Διακοπές και Στοίβα

- Τα Stacks είναι σημαντικά στην διαχείριση των interrupt
  - ✓ Especially in handling nested interrupts
  - ✓ Who sets up the IRQ and FIQ stacks and when?
- Το μέγεθος του Stack εξαρτάται από τον τύπο του ISR
  - ✓ Nested ISR requires more memory space
  - ✓ Stack grows in size with the number of nested interrupts
- Η καλή σχεδίαση του stack design πρέπει να αποφύγει την υπερχείλιση του stack (where stack extends beyond its allocated memory) – δύο μέθοδοι:
  - ✓ Memory protection
  - ✓ Call stack-check function at the start of each routine
- Είναι σημαντικό στα ενσωματωμένα συστήματα να γωρίζουμε το μέγεθος του stack πριν την εκτέλεση (ως μέρος της σχεδίασης της εφαρμογής)

Διακοπές και Στοίβα

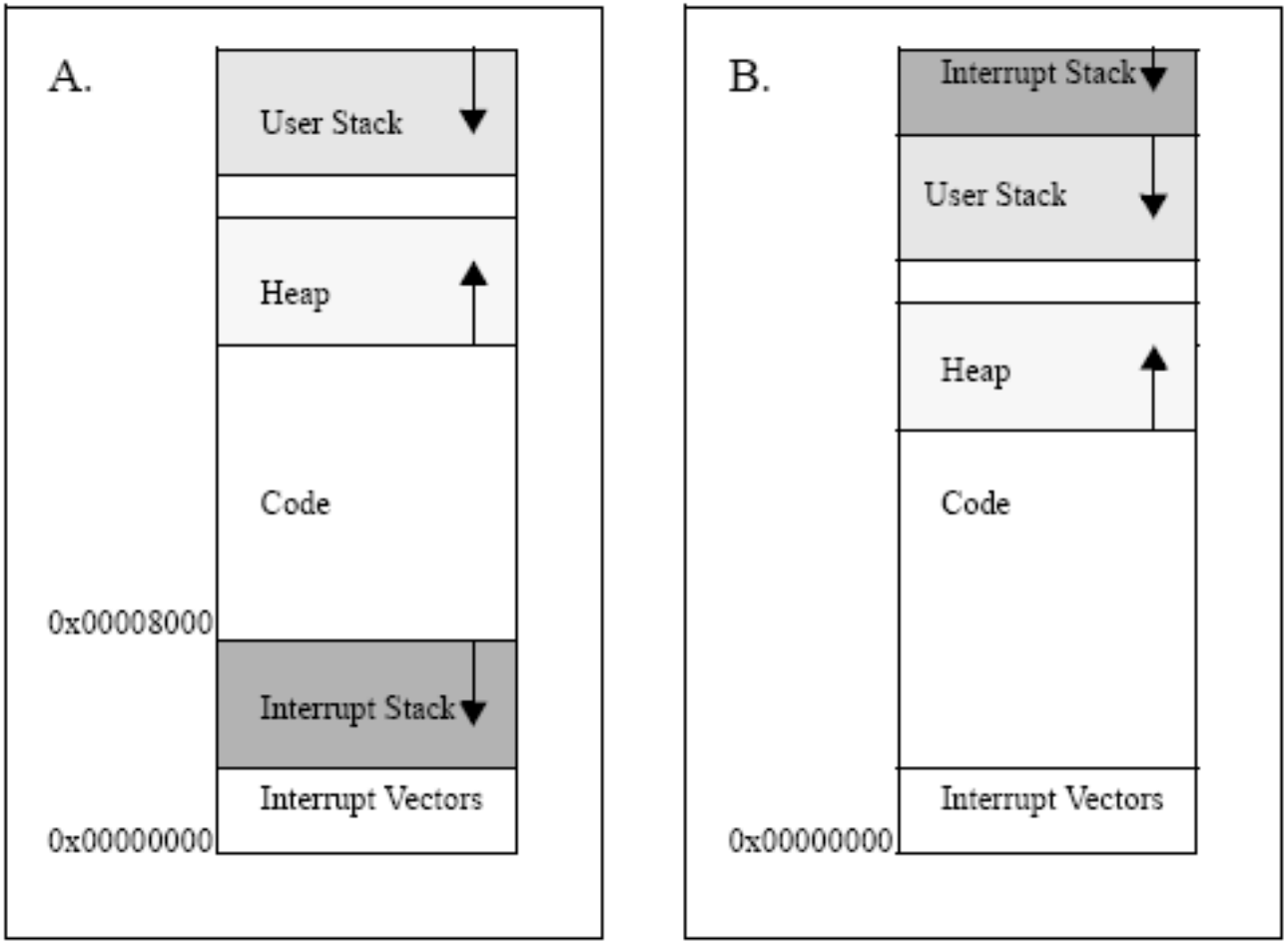


Figure 1.14 Typical stack design layouts

Ελαχιστοποίηση της καθυστέρησης Διακοπών

- Nested interrupt handler (problem with stack grow)
  - ✓ Allows further interrupts to occur when a current interrupt is being serviced
  - ✓ Re-enable interrupts at a safe point in the current ISR
  - ✓ Ultimately, nested interrupts roll back to the original ISR
- Προτεραιότητες στις Διακοπές
  - ✓ Ignore interrupts of the same/lower priority than the current interrupt
  - ✓ Higher-priority tasks can interrupt the current ISR
  - ✓ Higher-priority interrupts have a lower average interrupt latency

Τύποι χειριστών διακοπών

- Μη εμφωλευμένος χειριστής interrupt (simplest possible)
  - ✓ Services individual interrupts sequentially, one interrupt at a time
- εμφωλευμένος χειριστής interrupt
  - ✓ Handles multiple interrupts without priority assignment
- Re-entrant (prioritized) interrupt handler
  - ✓ Handles multiple interrupts that can be prioritized

Μη εμφωλευμένος χειριστής interrupt

- Does not handle any further interrupts until the current interrupt is serviced and control returns to the interrupted task
- Not suitable for embedded systems where interrupts have varying priorities and where interrupt latency matters
  - ✓ However, relatively easy to implement and debug
- Inside the ISR (after the processor has disabled interrupts, copied cpsr into spsr_mode, set the etc.)
  - ✓ Save context – subset of the current processor mode's nonbanked registers
  - ✓ Not necessary to save the spsr_mode – why?
  - ✓ ISR identifies the external interrupt source – how?
  - ✓ Service the interrupt source and reset the interrupt
  - ✓ Restore context
  - ✓ Restore cpsr and pc

Εμφωλευμένος χειριστής interrupt

- Επιτρέπει σε κάποιο άλλο interrupt να συμβεί μέσα στο τρέχον handler που εκτελείται
  - ✓ By re-enabling interrupts at a *safe point* before ISR finishes servicing the current interrupt
- Προσοχή πρέπει να δοθεί στην υλοποίηση
  - ✓ Protect context saving/restoration from interruption
  - ✓ Check stack and protect against register corruption
  - ✓ Increases code complexity, but improves interrupt latency
- Δεν υπάρχει διαφορά-διάκριση μεταξύ interrupts υψηλής και χαμηλής προτεραιότητας
  - ✓ Time taken to service an interrupt can be high for high-priority interrupts

Re-entrant (Prioritized) Interrupt Handler

- Allows for higher-priority interrupts to occur within the currently executing handler
  - ✓ By re-enabling higher-priority interrupts within the handler
  - ✓ By disabling all interrupts of lower priority within the handler
- Same care needs to be taken in the implementation
  - ✓ Protect context saving/restoration from interruption, check stack overflow
- Does distinguish between high and low priority interrupts
  - ✓ Interrupt latency can be better for high-priority interrupts

Re-enabling Interrupts in an ISR

- What problems can you anticipate if you re-enable interrupts in an ISR?
  - ✓ When you save context and interrupts have already been re-enabled, this can be a problem, why?
- Re-enable interrupts only after saving context
  - ✓ When you restore context and interrupts have been reenabled, this can be a problem
- Mask interrupts when you are restoring context
- If you haven't cleared the current interrupt source before re-enabling interrupts in a nested or re-entrant ISR, this can be a problem
  - ✓ You can get an infinite sequence of interrupts (or a race condition) – make sure to clear the current source before re-enabling interrupts
- Stack overflow issues

False/Ghost Interrupts

- When the interrupt handler executes the return to the interrupted instruction before clearing the original interrupt source
- Noise on the interrupt-assertion lines

Writing your Own ISR

- You can write your own ISR completely in C
- Use the __irq function declaration keyword
  - ✓ Preserves **all** registers used by the ISR on the stack
    - o Preserves the lr but only if another function is called from the ISR
  - ✓ Exits the function cleanly (sets PC and CPSR correctly on exit)

```
__irq void IRQHandler (void)
{
  volatile unsigned int *base = (unsigned int *)
  0x80000000;
  if (*base == 1) // which interrupt was it?
  {
          C_int_handler(); // process the interrupt
  }
  *(base+1) = 0; // clear the interrupt
}
```

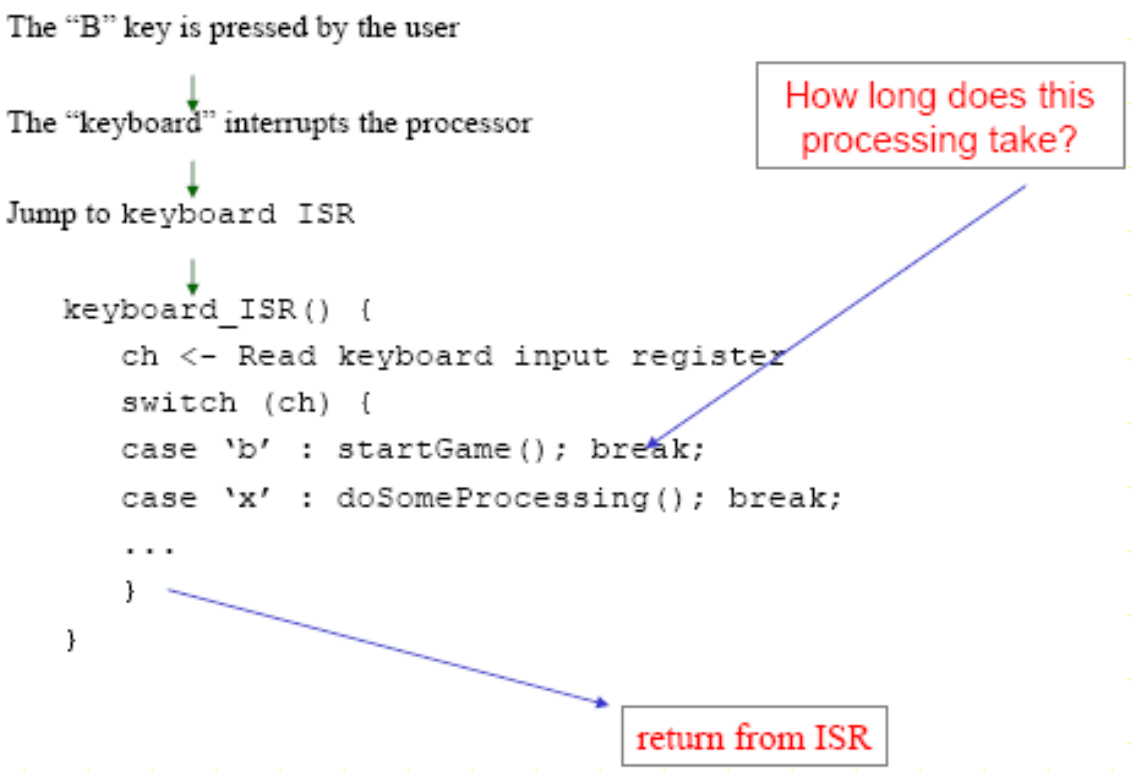
- Cannot be used for re-entrant interrupt handlers, Why?
  - ✓ What are the implications of this on nested interrupt handlers?

Διαμοιρασμός Πόρων Across Interrupts

- Need to control resource sharing across interrupts
- Interrupts can occur asynchronously (i.e., whenever they want)
- Access to shared resources and global variables must be handled in a way that does not corrupt the program
- Normally done by masking interrupts before accessing shared data and unmasking interrupts afterwards
  - ✓ Clearly, when interrupt-masking occurs, interrupt latency will be higher
- Start with a simple keyboard ISR and then understand what happens when it takes a while, when we try to improve its interrupt latency, and when we need to deal with other concurrent interrupts during the ISR

Ξεκινώντας με ένα απλό παράδειγμα

- Keyboard command processing



Θα χαθούν events ?

How fast is the keyboard_ISR()?

The "B" key is pressed by the user

↓

The "keyboard" interrupts the processor

↓

Jump to keyboard ISR

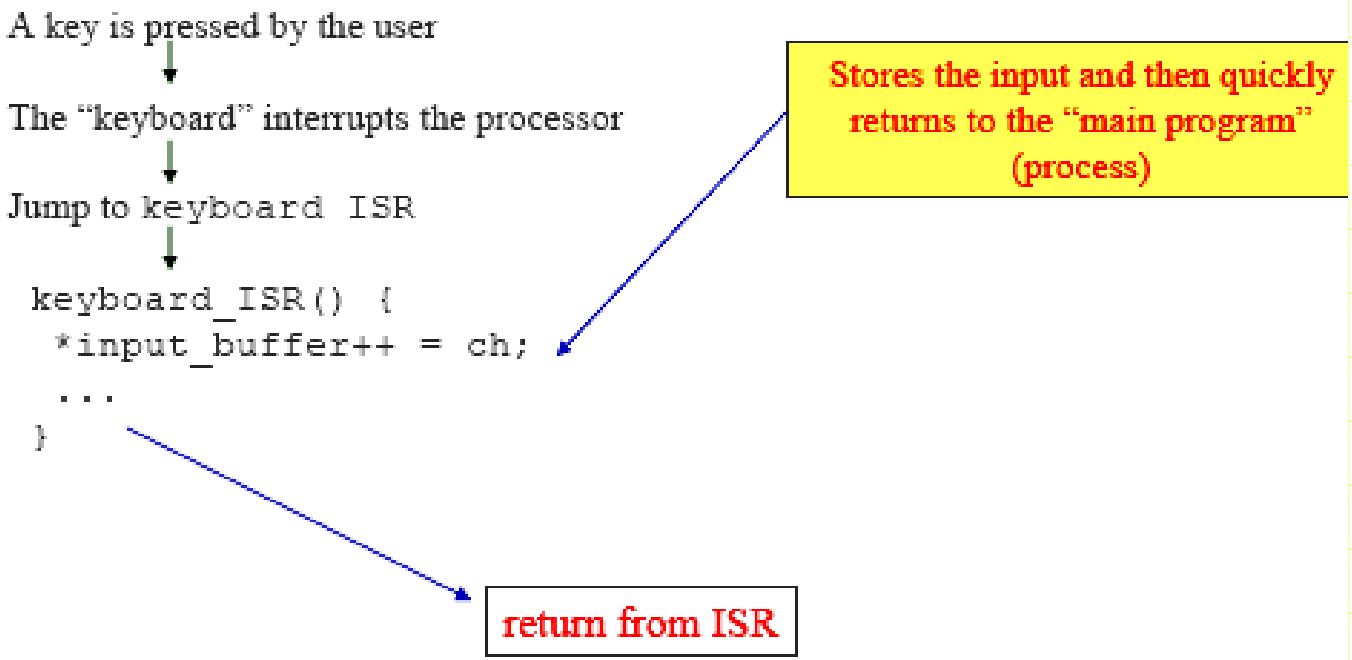
↓

```
keyboard_ISR(){
    ch <- Read keyboard input register
    switch (ch) {
    case 'b' : startGame(); break;
    case 'x' : doSomeProcessing();
               break;
    ...
    }
}
```

What happens if another key is pressed - or if a timer interrupt occurs?

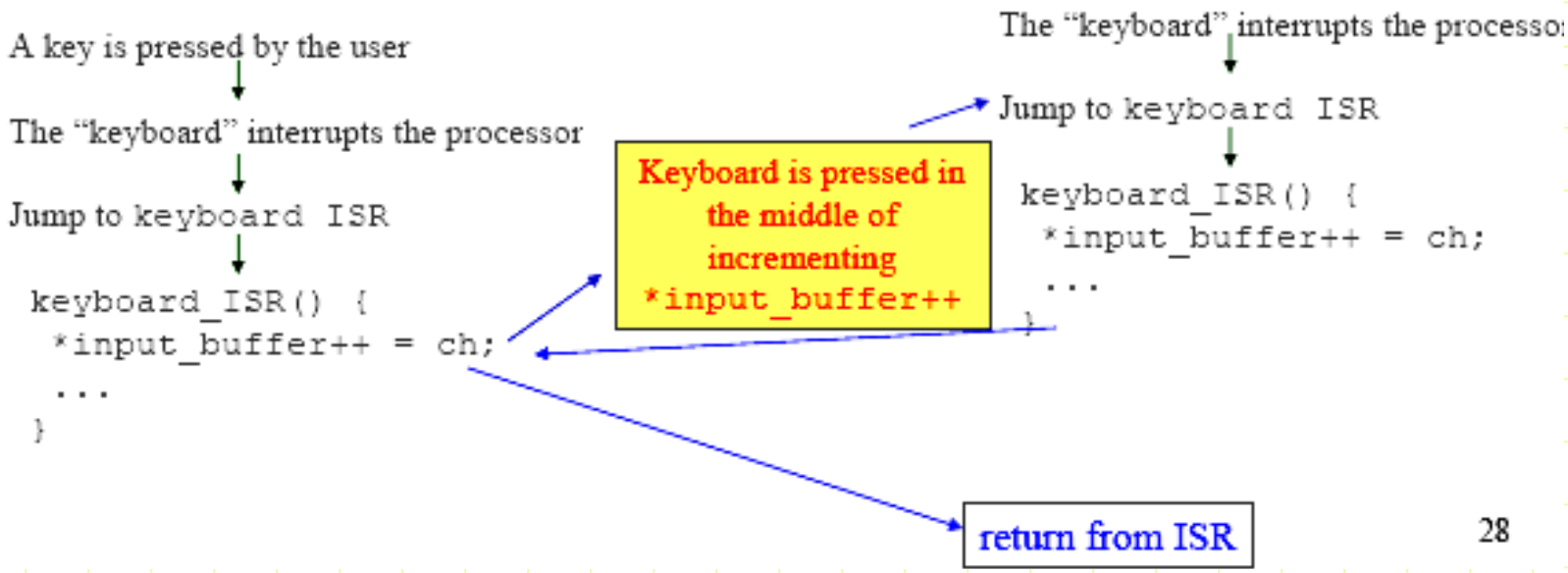
Βελτιώνοντας την καθυστέρηση Διακοπών

- Add a buffer (in software or hardware) for input characters.
  - ✓ This decouples the time for processing from the time between keystrokes, and provides a computable upper bound on the time required to service a keyboard interrupt.



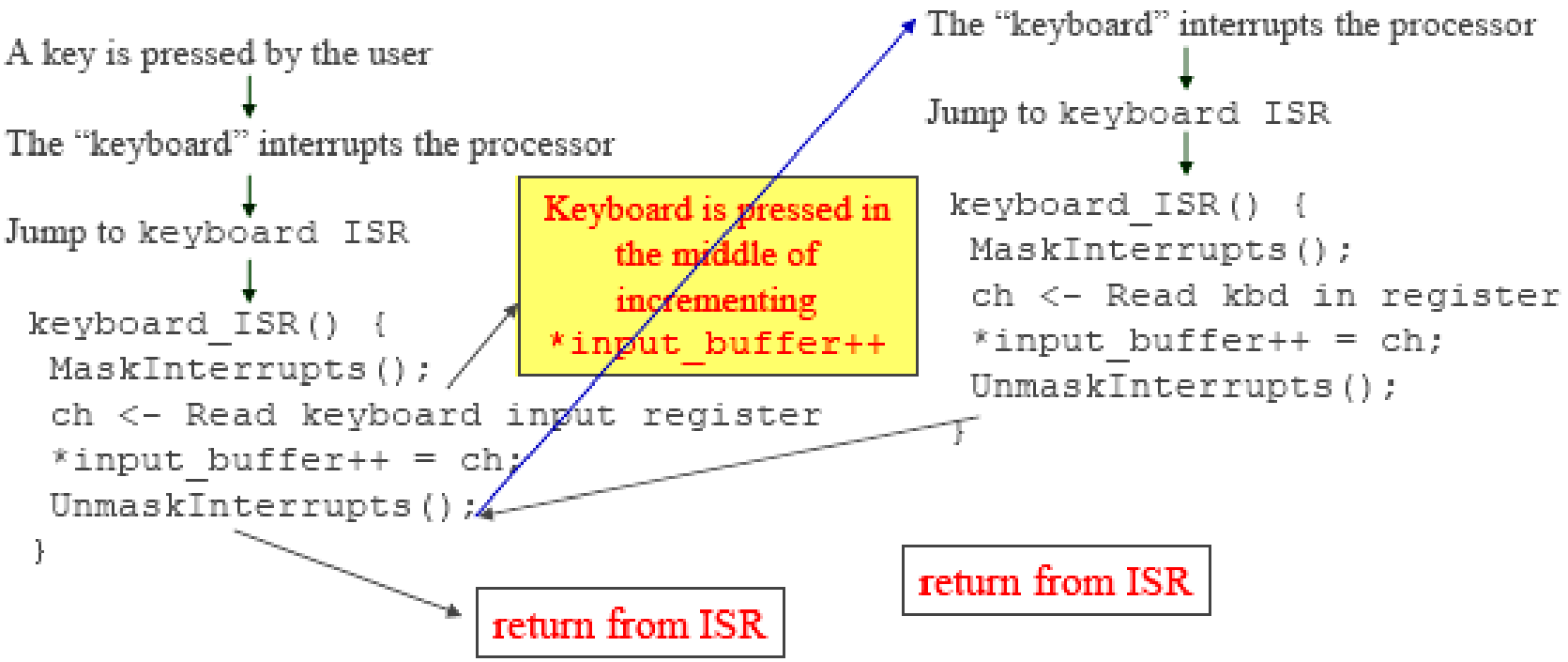
Τι μπορεί να πάει στραβά ?

- 1. ο Buffer μπορεί να υπερχειλίσει (bigger buffer helps, but there is a limit)
- 2. Could *another* interrupt occur while adding the current keyboard character to the input_buffer?
  - ✓ Result will depend upon whether interrupts have been enabled or not



Masking Interrupts

- If interrupts are masked (IRQ and FIQ disabled), nothing will be processed until the ISR completes and returns.



Επεξεργασία του buffer

- Απαιτείται προσοχή όταν τροποποιείται ο buffer όταν τα interrupts είναι ενεργοποιημένα

```
keyboard_ISR(){
   MaskInterrupts();
   ch <- Read ACIA input register
   *input_buffer++ = ch;
   UnmaskInterrupts();
}
```

return from ISR

application code

```
...
...
while (!quit){
   if (*input_buffer){
      processCommand(*input_buffer);
      removeCommand(input_buffer);
   }
}
...
...
```

What happens if another command is entered as you remove one from the inputBuffer?

30