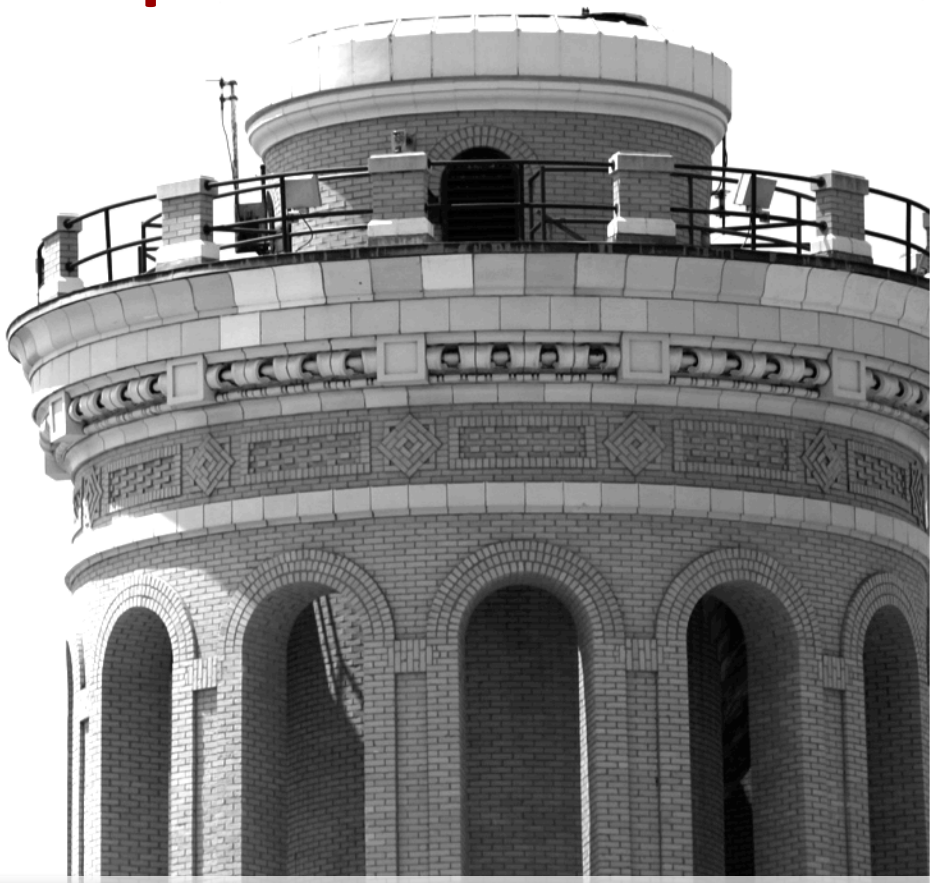
18-349: Introduction to Embedded Real-Time Systems

Lecture 6: Timers and Interrupts

Anthony Rowe
Electrical and Computer Engineering
Carnegie Mellon University



Electrical & Computer ENGINEERING

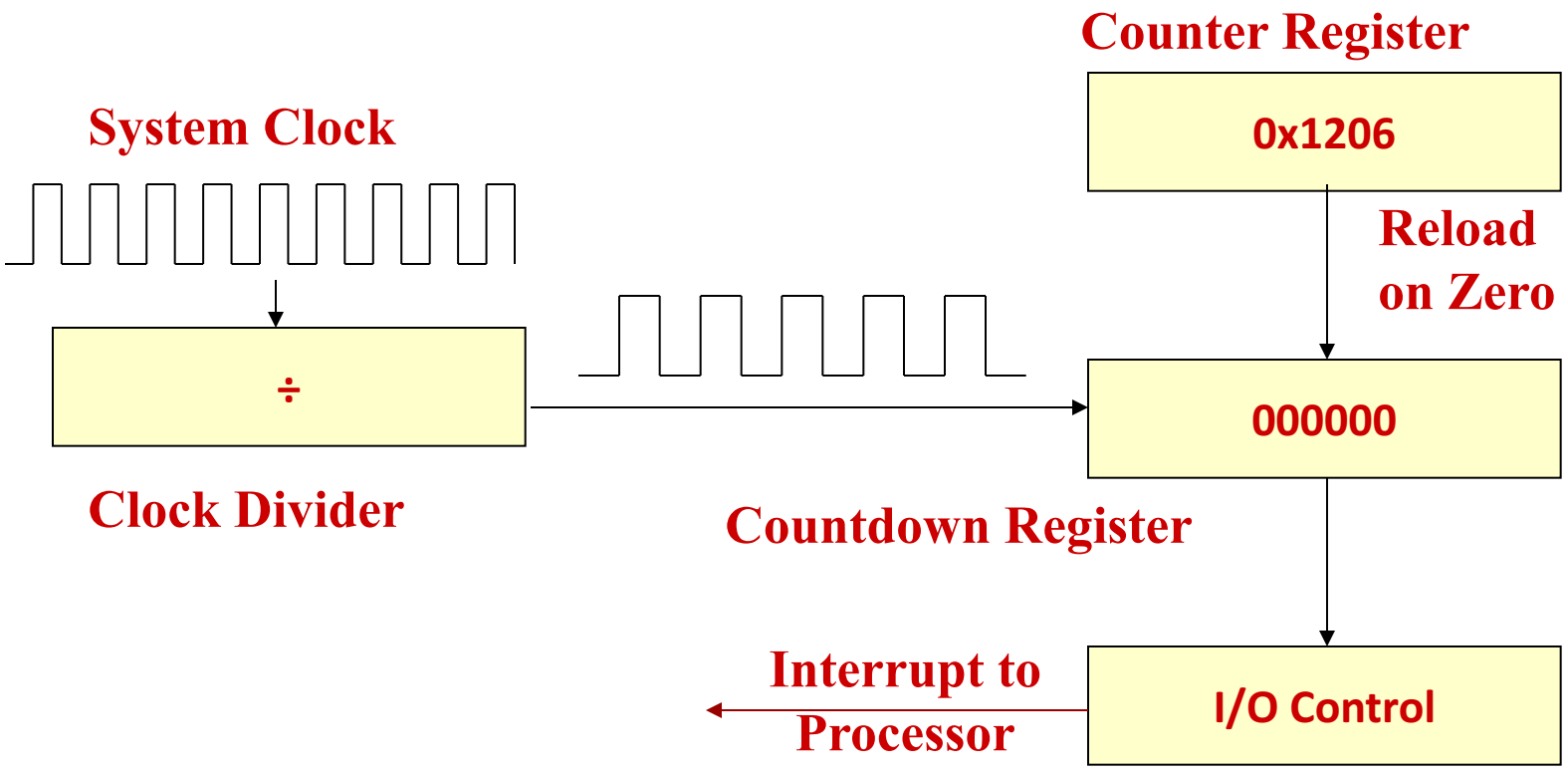
Carnegie Mellon University

Lecture Overview

- Timers
- Interrupts
  - Interrupt Latency
  - Interrupt Handlers
- Concurrency issues with interrupt handlers

What is a Timer?

- A device that uses a highspeed clock input to provide a series of time or count-related events

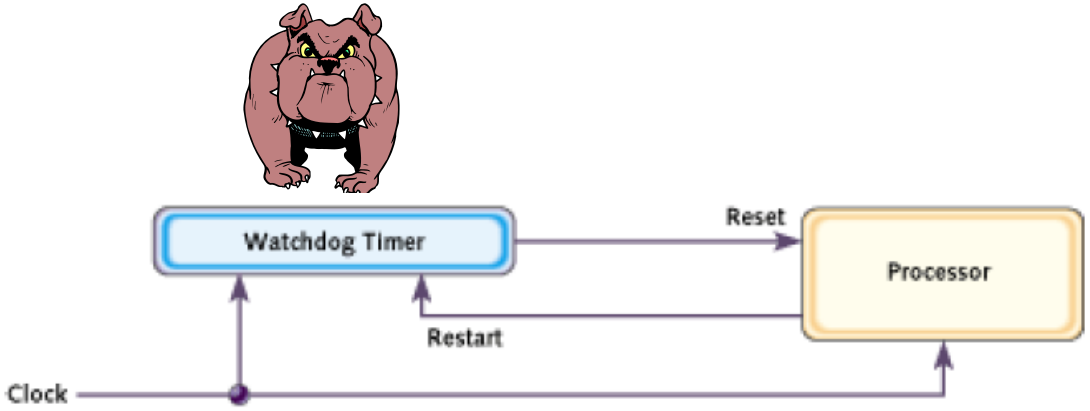


Uses of Timers

- Pause Function
  - Suspends task for a specified amount of time
- One-shot timer
  - Single one-time-only timeout
- Periodic timer
  - Multiple renewable timeouts
- Time-slicing
  - Chunks of time to each task
- Watchdog timer

Watchdog Timers

- A piece of hardware that can be used to reset the processor in case of anomalies
- Typically a timer that counts to zero
  - Reboots the system if counter reaches zero
  - For normal operation – the software has to ensure that the counter never reaches zero ("kicking the dog")



Care of Your Watchdog

- A watchdog can get the system out of many dangerous situations
- But, be very careful
  - Bugs in the watchdog timer could perform unnecessary resets
  - Bugs in the application code could perform resets
- Choosing the right kicking interval is important
  - System initialization process is usually lengthy
    - Some watchdogs can wait longer for the first kick than for the subsequent kicks
  - What should you do, for example, if some functions in a for loop *can* take longer than the maximum timer interval?

Interrupts

Merriam-Webster:

- "to break the uniformity or continuity of"

- Informs a program of some external events
- Breaks execution flow

Key questions:

- Where do interrupts come from?
- How do we save state for later continuation?
- How can we ignore interrupts?
- How can we prioritize interrupts?
- How can we share interrupts?

Interrupts

Interrupt (a.k.a. exception or trap):

- An event that causes the CPU to stop executing current program
- Begin executing a special piece of code
  - Called an **interrupt handler** or **interrupt service routine** (ISR)
    - Typically, the ISR does some work
    - Then resumes the interrupted program

Interrupts are really glorified procedure calls, except that they:

- **can occur between any two instructions**
- are “transparent” to the running program (usually)
- are not explicitly requested by the program (typically)
- call a procedure at an address determined by the type of interrupt, not the program

Two basic types of interrupts (1/2)

- Those caused by an instruction
  - Examples:
    - TLB miss
    - Illegal/unimplemented instruction
    - div by 0
    - SVC (supervisor call, e.g.: SVC #3)

- Names:
    - Trap, exception

Two basic types of interrupts (2/2)

- Those caused by the external world
  - External device
  - Reset button
  - Timer expires
  - Power failure
  - System error

- Names:
  - interrupt, external interrupt

How it works

- Something tells the processor core there is an interrupt
- Core transfers control to code that needs to be executed
- Said code “returns” to old program
- Much harder then it looks.
  - Why?

Devil is in the details

- How do you figure out *where* to branch to?
- How to you ensure that you can get back to where you started?
- Don't we have a pipeline? What about partially executed instructions?
- What if we get an interrupt while we are processing our interrupt?
- What if we are in a "critical section?"

Interrupt vs. Polled I/O

- Polled I/O requires the CPU to *ask* a device (e.g. Ethernet controller) if the device requires servicing
  - For example, if the Ethernet controller has changed status or received packets
  - Software plans for polling the devices and is written to know when a device will be serviced
- Interrupt I/O allows the device to *interrupt* the processor, announcing that the device requires attention
  - This allows the CPU to ignore devices unless they request servicing (via interrupts)
  - Software cannot plan for an interrupt because interrupts can happen at any time therefore, software has no idea when an interrupt will occur
- Processors can be programmed to ignore or mask interrupts
  - Different types of interrupts can be masked (IRQ vs. FIQ)

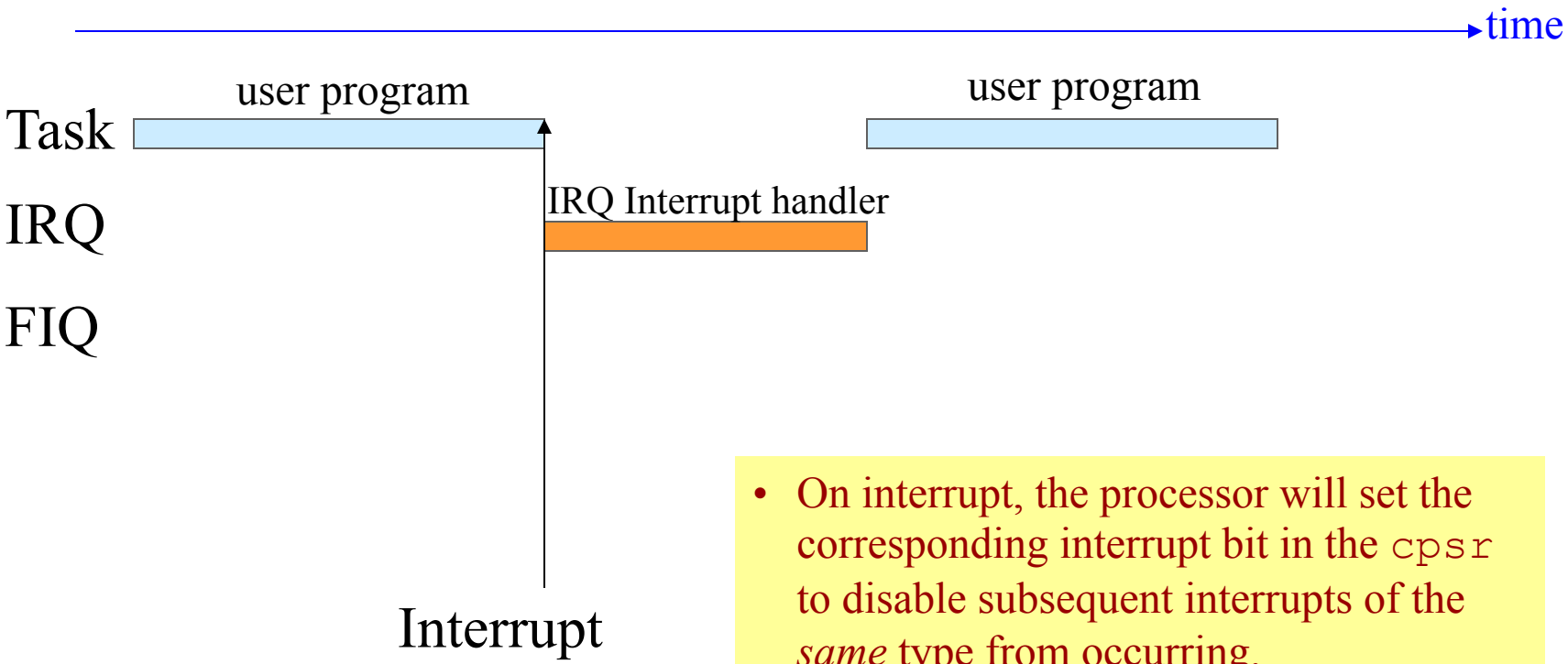
Polling vs. InterruptDriven I/O

- Polling requires code to loop until device is ready
  - Consumes *lots* of CPU cycles
  - Can provide quick response (guaranteed delay)

- Interrupts don't require code to loop until the device is ready
  - Device interrupts processor when it needs attention
  - Code can go off and do other things
  - Interrupts can happen at any time
    - Requires careful coding to make sure other programs (or your own) don't get messed up

- What do you think real-time embedded systems use?

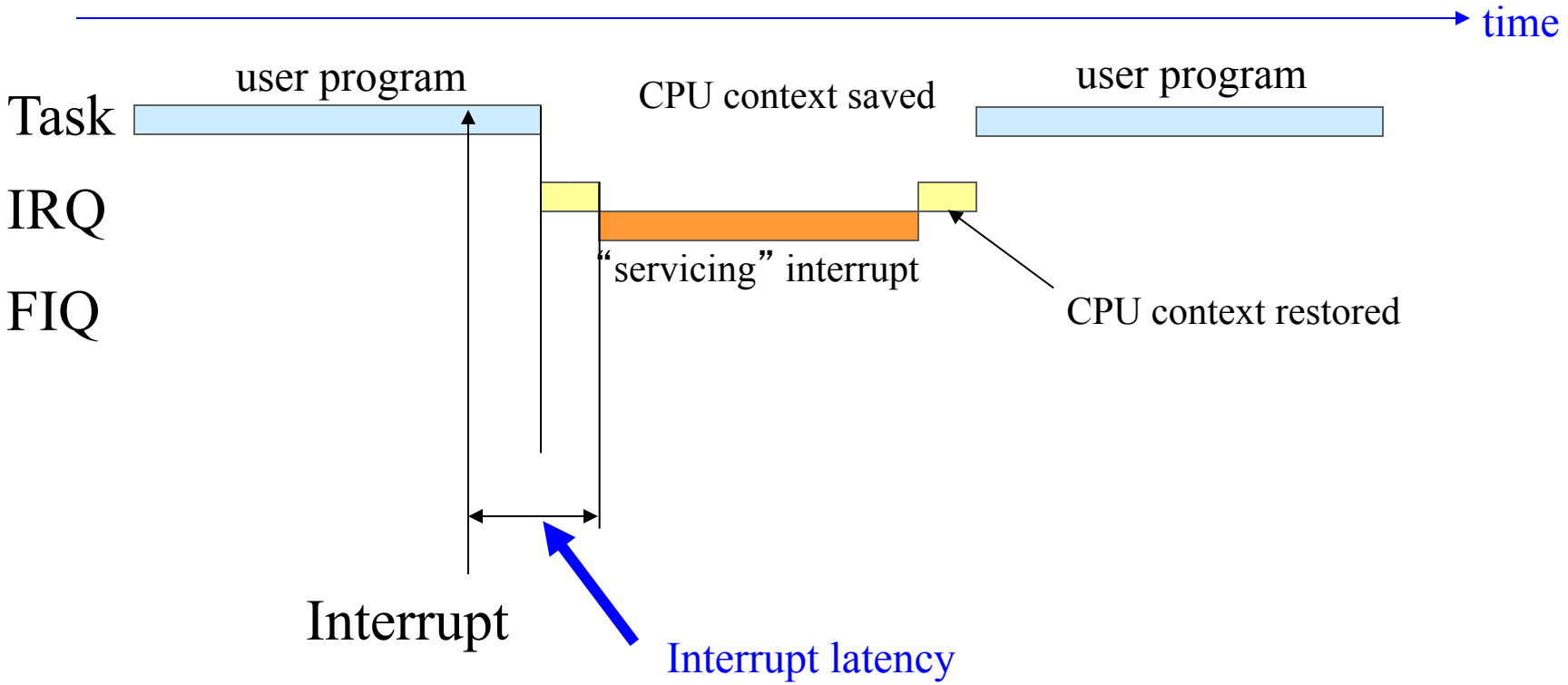
Onto IRQs & FIQs: Interrupt Handlers



- 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.

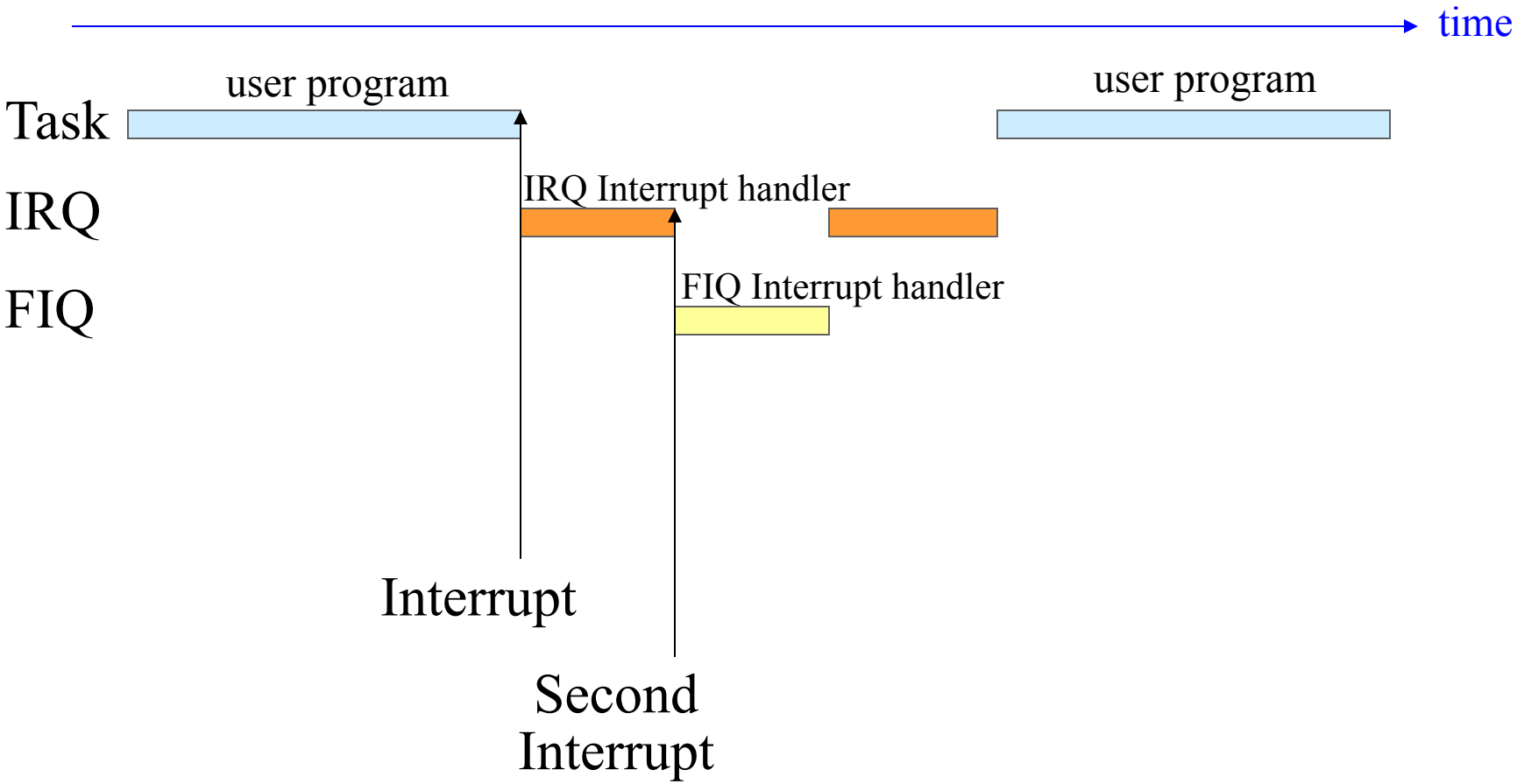
Timing Issues in Interrupts

- Before an interrupt handler can do anything, it must save away the current program's registers (if it touches those registers)
- That's why the FIQ has lots of extra registers, to minimize CPU context-saving overhead

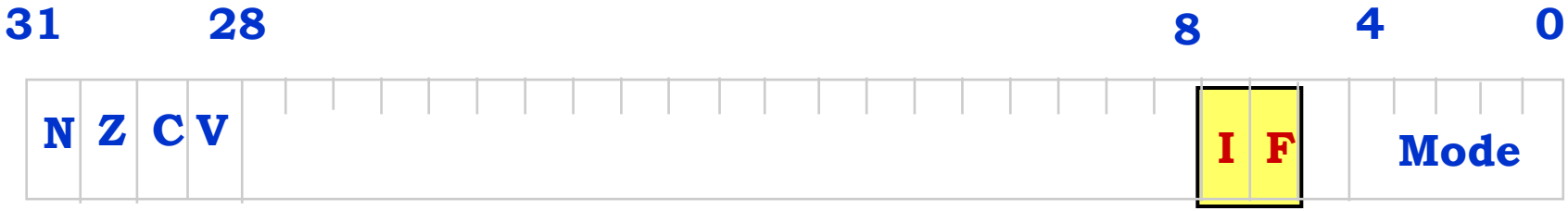


Servicing FIQs Within IRQ

- Interrupts can occur within interrupt handlers



cpsr & spsr for IRQs and FIQs



- Interrupt Disable bits
  - I = 1, *disables* the IRQ
  - F = 1, *disables* the FIQ
- Mode bits
  - Processor mode differs

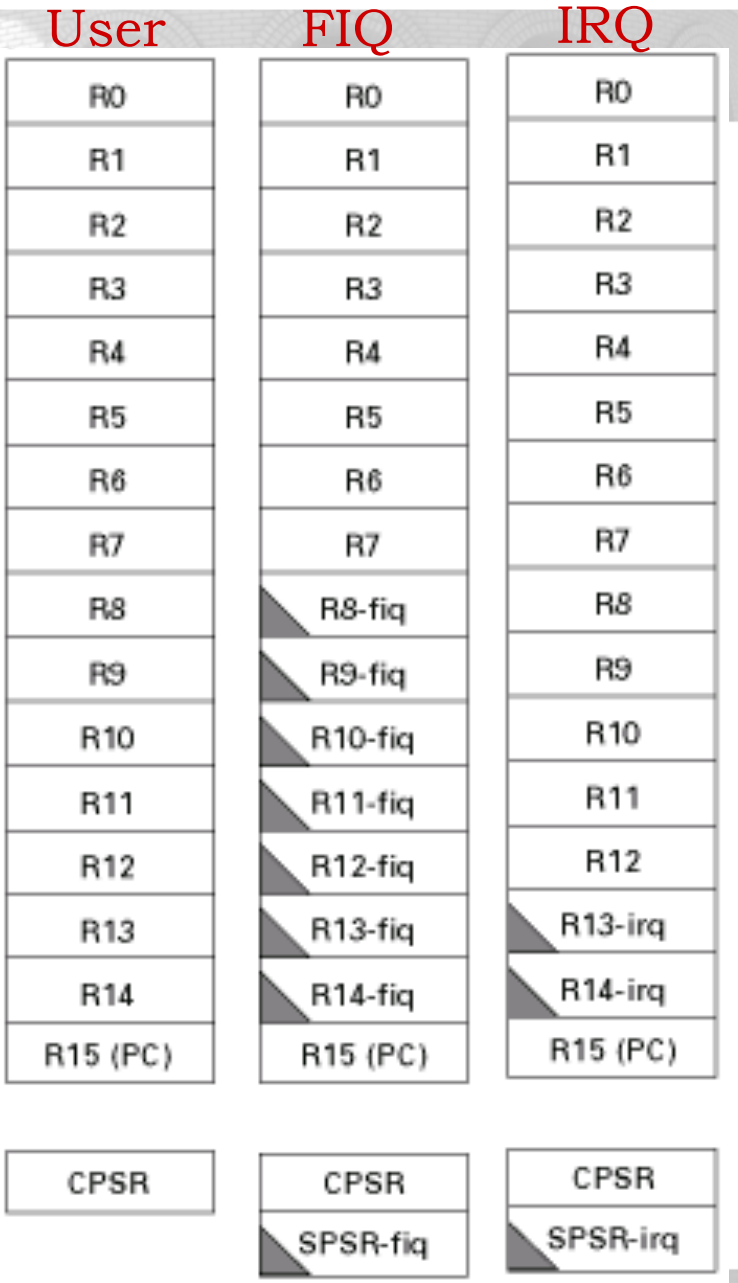
M[4:0]	Mode
10000	User
10001	FIQ
10010	IRQ
10011	SVC
10111	Abort
11011	Undef
11111	System

Exception Priorities

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	

How are FIQs Faster?

- FIQs are faster than IRQs in terms of interrupt latency
- 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



IRQ and FIQ ISR Handling

IRQ Handling

When an IRQ occurs, the processor

- Copies cpsr into spsr_irq
- Sets appropriate cpsr bits
  - Sets mode field bits to 10010
  - Disables further IRQs
- Maps in appropriate banked registers
- Stores the address of "*next instruction + 4*" 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

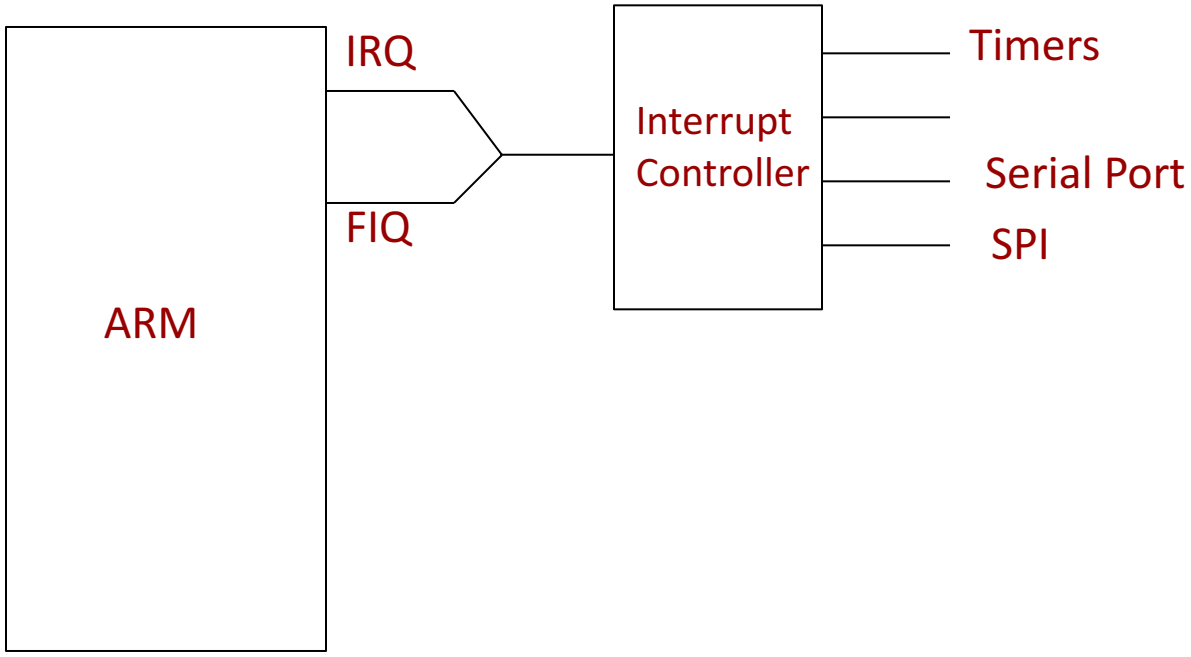
When an FIQ occurs, the processor

- Copies cpsr into spsr_fiq
- Sets appropriate cpsr bits
  - Sets mode field bits to 10001
  - Disables further IRQs and FIQs
- Maps in appropriate banked registers
- Stores the "*next instruction + 4*" in lr_fiq
- Sets pc to vector address 0x0000001c0

To return, exception handler needs to:

- Restore cpsr from spsr_fiq
- Restore pc from lr_fiq
- Return to user mode

Interrupt Controller



Jumping to the Interrupt Handler

- Non-vectored
  - Processor jumps to the same location irrespective of the kind of interrupt
  - Hardware simplification
- Vectored
  - Device supplies processor with address of interrupt service routine
  - Interrupt handler reads the address of the interrupt service routine from a special bus
- Why the different methods?
  - If multiple devices uses the same interrupt the processor must poll each device to determine which device interrupted the processor
    - This can be time-consuming if there is a lot of devices
  - In a vectored system, the processor would just take the address from the device (which dumps the interrupt vector onto a special bus).

Jumping to the Interrupt Handler

- Auto-vectored
  - Multiple CPU interrupt inputs for interrupts of different priority level
    - ARM has two – FIQ and IRQ
    - Other processors, like 68000, SPARC, may have 8 or more
  - Processor-determines address of interrupt service routine based on type of interrupt
  - For ARM, pseudo-auto vectored IRQs and FIQs is implemented using an on-chip interrupt controller

Types of Interrupt Handlers

- Non-nested interrupt handler (simplest possible)
  - Services individual interrupts sequentially, one interrupt at a time
- Nested interrupt handler
  - Handles multiple interrupts without priority assignment
- Prioritized (re-entrant) interrupt handler
  - Handles multiple interrupts that can be prioritized

Non-Nested Interrupt Handler

- 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`

Nested Interrupt Handler

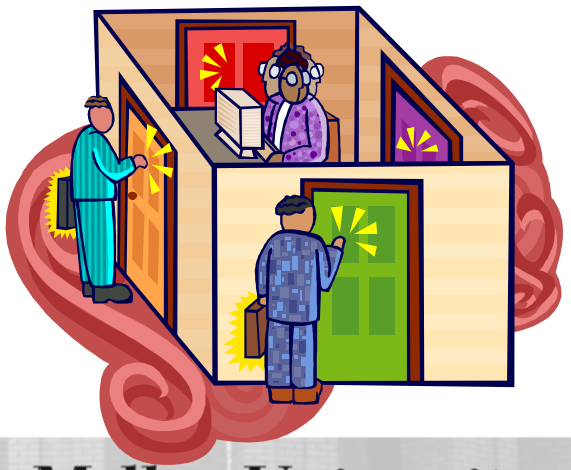
- Allows for another interrupt to occur within the currently executing handler
  - By re-enabling interrupts at a *safe point* before ISR finishes servicing the current interrupt

- Care needs to be taken in the implementation
  - Protect context saving/restoration from interruption
  - Check stack
  - Increases code complexity, but improves interrupt latency

- Does not distinguish between high and low priority interrupts
  - Time taken to service an interrupt can be high for high-priority interrupts

Prioritized (Re-entrant) 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



Interrupts and Stacks

- Stacks are important in interrupt handling
  - Especially in handling nested interrupts
  - Who sets up the IRQ and FIQ stacks and when?
- Stack size depends on the type of ISR
  - Nested ISRs require more memory space
  - Stack grows in size with the number of nested interrupts
- Good stack design avoids stack overflow (where stack extends beyond its allocated memory) – two common methods
  - Memory protection
  - Call stack-check function at the start of each routine
- Important in embedded systems to know the stack size ahead of time (as a part of the designing the application) – why?

Resource Sharing Across Interrupts

- Interrupts can occur asynchronously
- 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 (if needed) afterwards
  - Clearly, when interrupt-masking occurs, interrupt latency will be higher
- Up next – start with a simple keyboard ISR and then understand
  - What can happen when the ISR takes a while to execute
  - How do we improve its interrupt latency
  - What can go wrong

Starting With a Simple Example

- Keyboard command processing

The "B" key is pressed by the user

↓

The "keyboard" interrupts the processor

↓

Jump to `keyboard ISR (non-nested)`

↓

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

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

How long does this processing take?

return from ISR

Improving Interrupt Latency

- 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
  - Commands stored in the `input_buffer` can be processed in the user/application code

A key is pressed by the user

↓

The "keyboard" interrupts the processor

↓

Jump to `keyboard ISR`

↓

```
keyboard_ISR() {
 *input_buffer++ = ch;
 ...
}
```

return from ISR

Stores the input and then quickly returns to the "main program" (process)

Application Code

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

What Can Go Wrong? Buffer Processing


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

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?

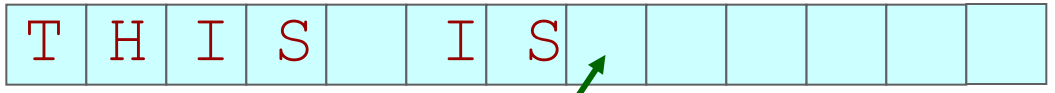
Another Concurrency Problem

- An application uses the serial port to print characters on the terminal emulator (Hyper Terminal)
  - The application calls a function `PrintStr` to print characters to the terminal
  - In the function `PrintStr`, the characters to be printed are copied into an output buffer (use of output buffer to reduce interrupt latency)
- In the serial port ISR
  - See if there is any data to be printed (whether there are new characters in the output buffer)
  - Copy data from the output buffer to the transmit holding register of the UART
- The (new app) display also needs to print the current time on the terminal – a timer is used (in interrupt mode) to keep track of time
- In the timer ISR
  - Compute current time
  - Call `PrintStr` to print current time on the terminal emulator

Another Example: Buffer for Printing Chars to Screen


```
PrintStr(*string)
char *string;
{
   while (*string) {
      outputBuffer[tail++] = *string++;
   }
}
```

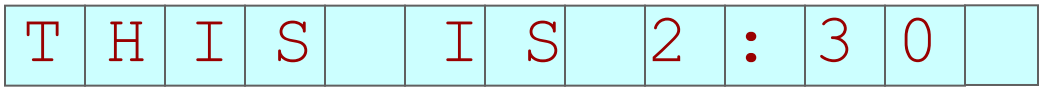
```
PrintStr("this is a line");
```



tail points here and a timer interrupt occurs

Jump to `timer_ISR`

```
timer_ISR(){
   clockTicks++;
   PrintStr(convert(clockTicks));
}
```

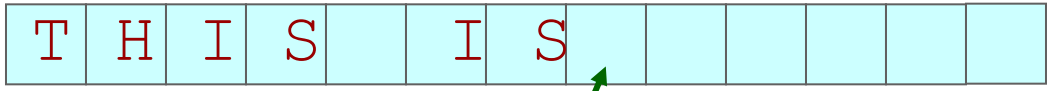


Critical Sections of Code

- Pieces of code that must appear as an **atomic action**

```
printStr(*string)
char *string;
{
   MaskInterrupts();
   while (*string){
      outputBuffer[tail++]= *string++;
}
   UnmaskInterrupts();
}
```

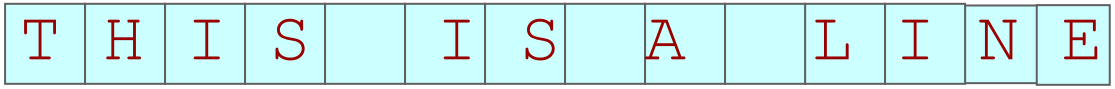
```
printStr("this is a line");
```



tail points here and a timer interrupt occurs

Jump to `timer_ISR` happens **after** `printStr()` completes

```
timer_ISR(){
   clockTicks++;
   printStr(convert(clockTicks));
}
```



Atomic action
action that "appears"
to take place in a
single indivisible operation

Shared-Data Problems

- Previous examples show what can go wrong when data is shared between ISRs and application tasks
- Very hard to find, and debug such concurrency problems (if they exist)
  - Problem may not happen every time the code runs
    - In the previous example, you may not have noticed the problem if the timer interrupt did not occur in the `PrintStr` function
- Lessons learned
  - Keep the ISRs short
  - Analyze your code carefully, if any data is shared between ISRs and application code

Summary

- Timers
- Interrupts
  - Interrupt Latency
  - Interrupt Handlers
- Concurrency issues with interrupt handlers
- **Next Lecture: ARM Optimization (NOT SWI and the Kernel)**