CG2271

Real-Time Operating Systems

Lecture 5

Task Management

colintan@nus.edu.sg

National University of Singapore

School of Computing

Learning Objectives

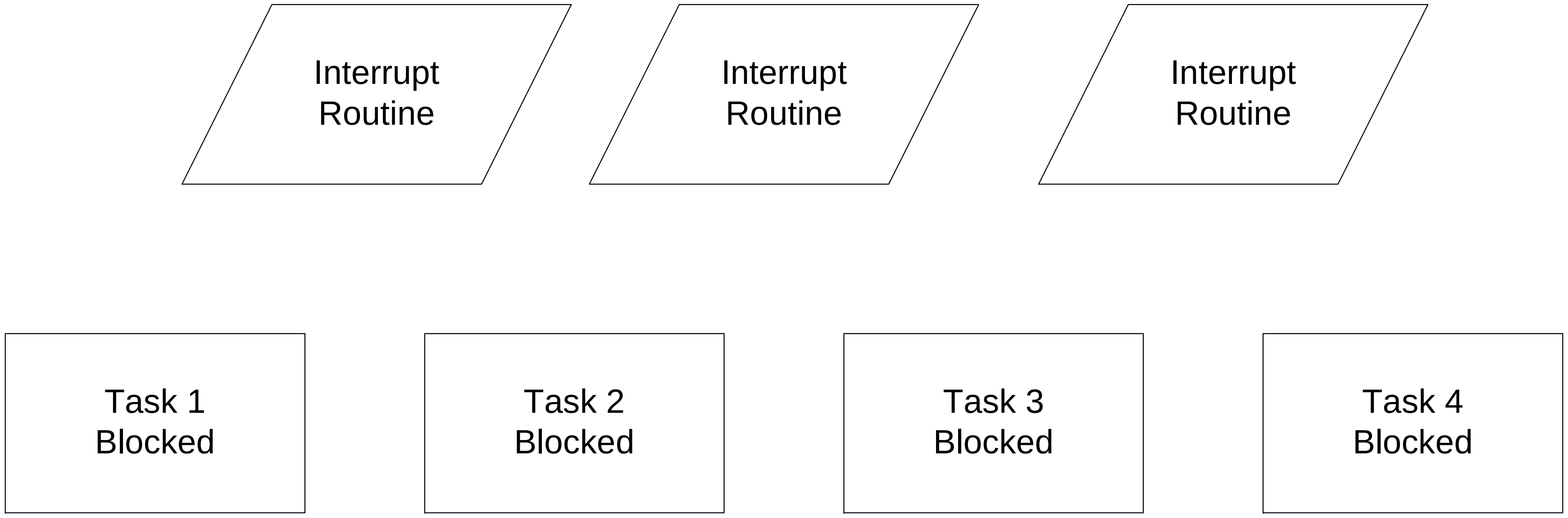
- **By the end of this lecture you will be able to:**
  - Understand the concept of a task, and how tasks behave in an RTOS environment.
  - Understand how tasks are created and managed in an RTOS environment.
  - Understand how an RTOS can simulate multiple tasks running even on a single core single processor system.

Task Management

GENERAL OPERATION OF EMBEDDED APPLICATIONS

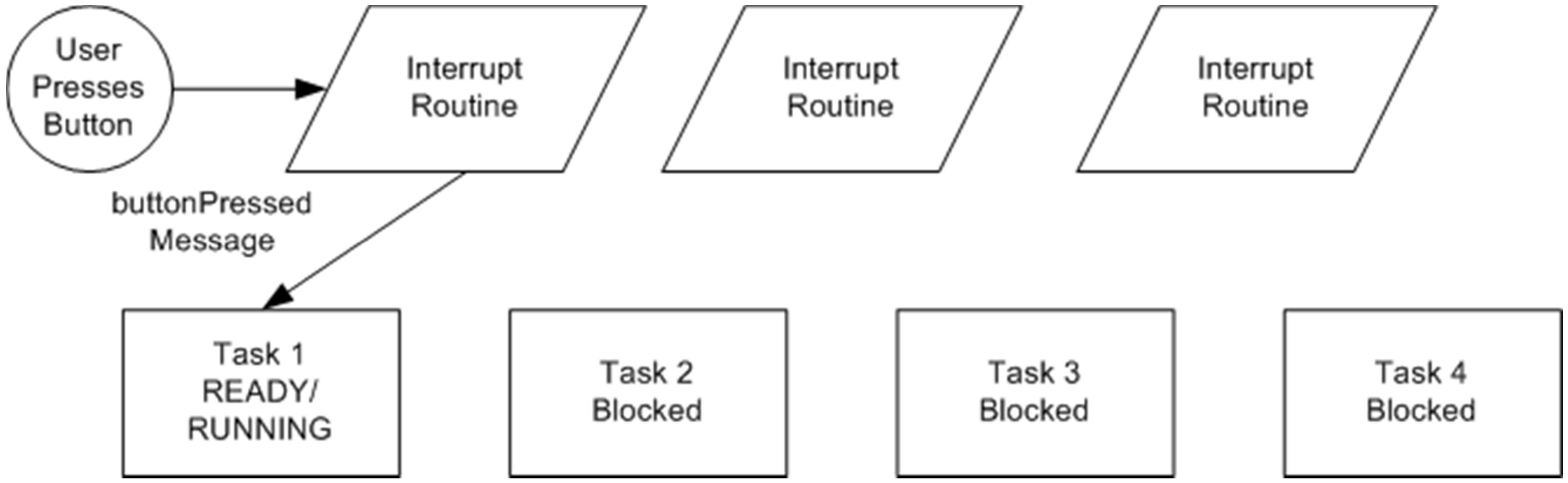
General Operation of an Embedded Application

Most of the time, tasks are blocked, waiting for something to happen.



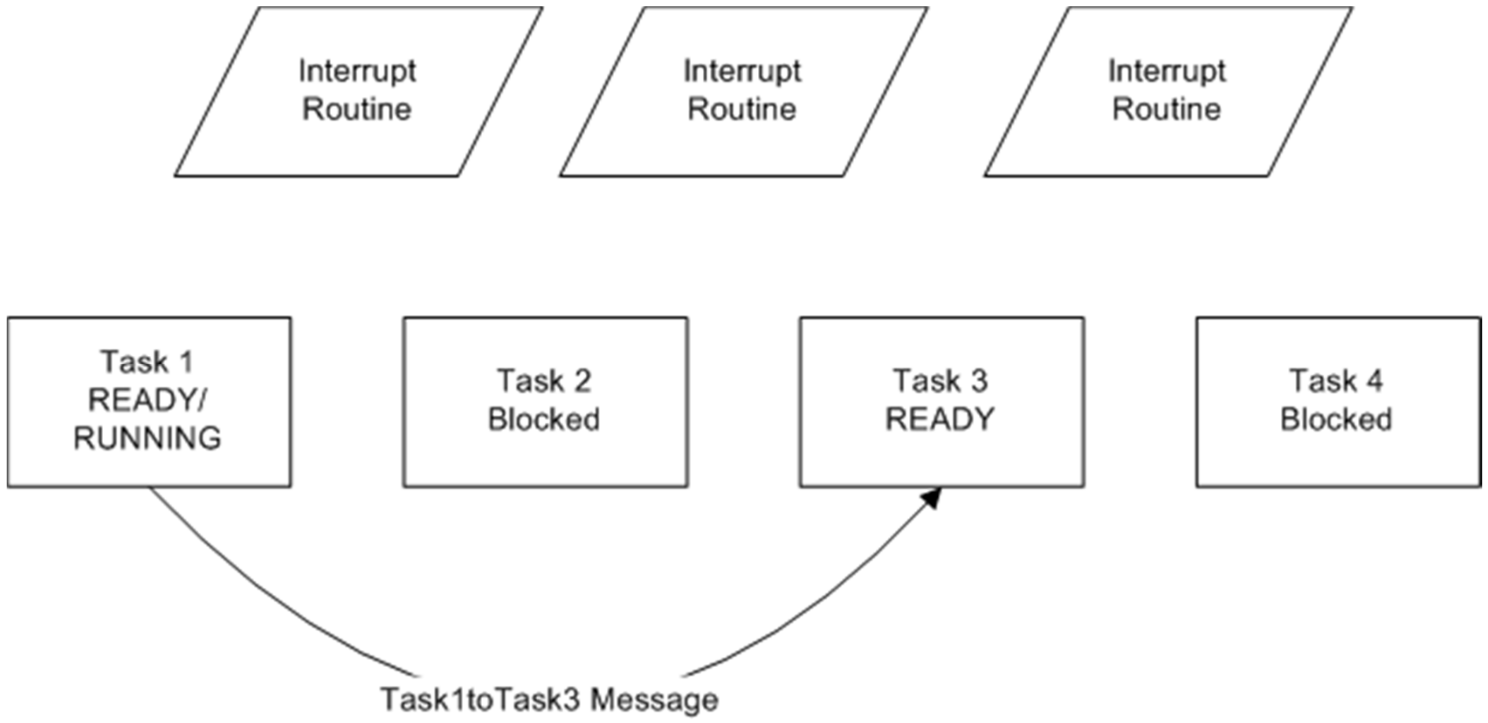
General Operation of an Embedded Application

Something happens, triggering an interrupt. Interrupt handler sends a message, causing a task to unblock.



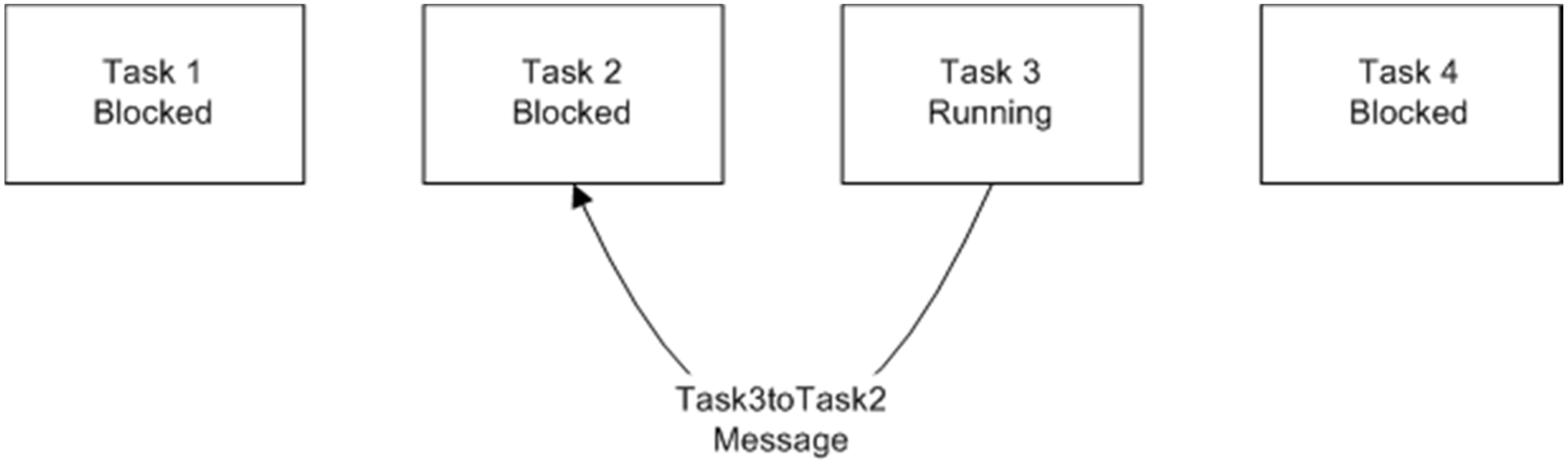
General Operation of an Embedded Application

Task 1 computes something and passes it to Task 3, which moves from BLOCKED to READY.



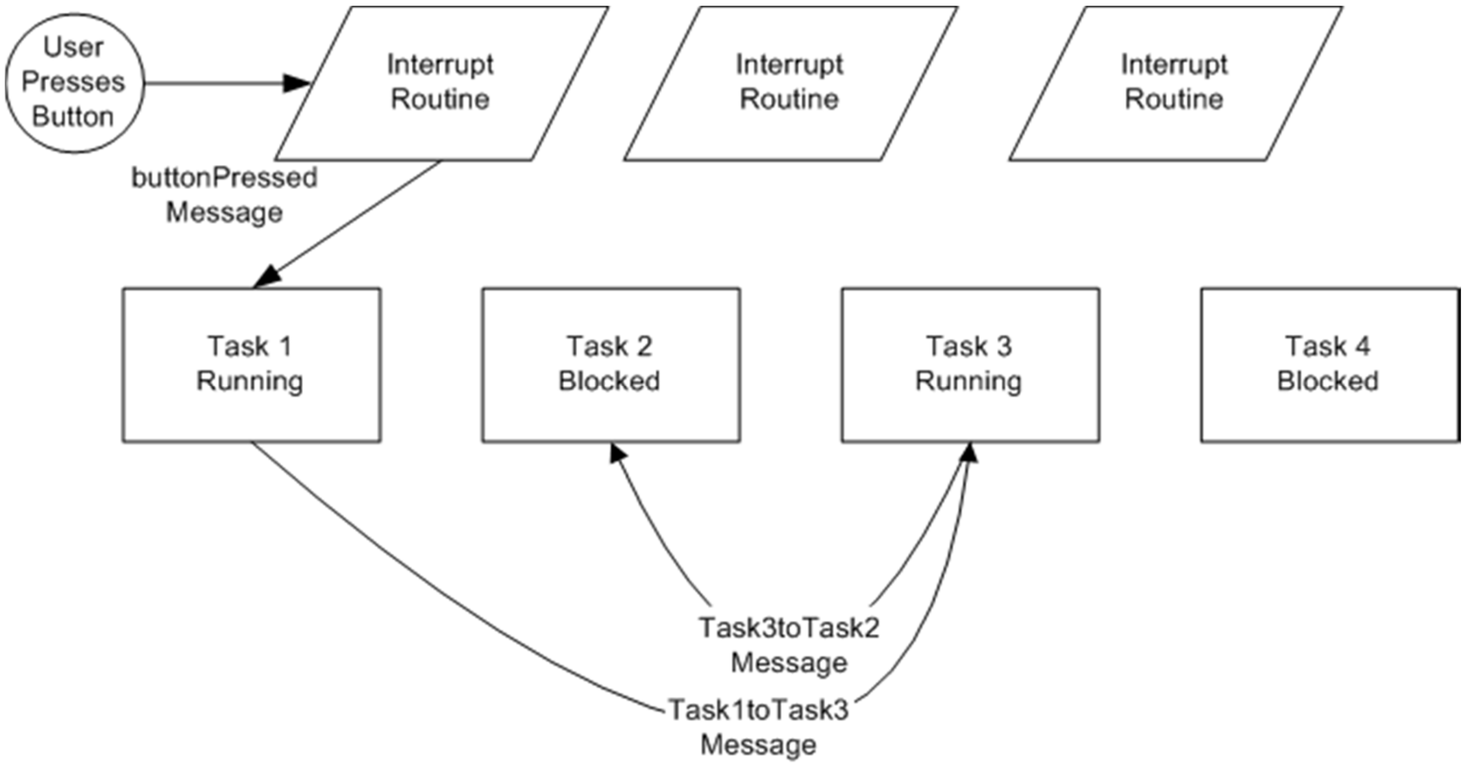
General Operation of an Embedded Application

Task 3 starts running, sending messages to other tasks, that then start up and send more messages.



General Operation of an Embedded Application

- **Thus a single interrupt generates a flurry of task and message activity.**



Task Management

INTERRUPT ROUTINES

Interrupt Routines

- **Interrupt routines are thus the backbone of a Real Time System.**
- **Always write short interrupt handlers:**
  - Even the lowest priority interrupt handler will interrupt the highest priority tasks!

Interrupt Routines

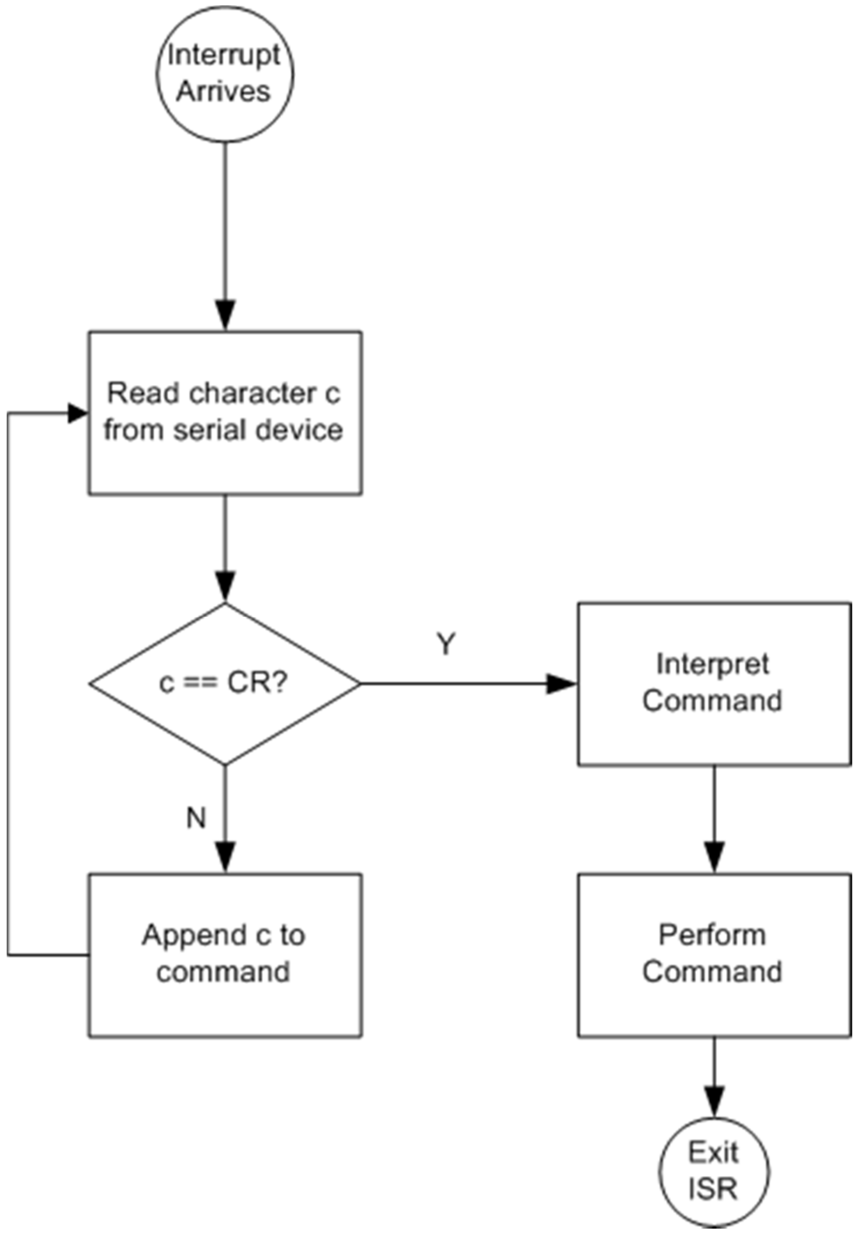
- **Example System:**
  - System responds to commands coming in from a serial port.
  - Commands always end with a carriage return.
  - Commands arrive one at a time. Next command will not come till system responds to previous command.

Interrupt Routines

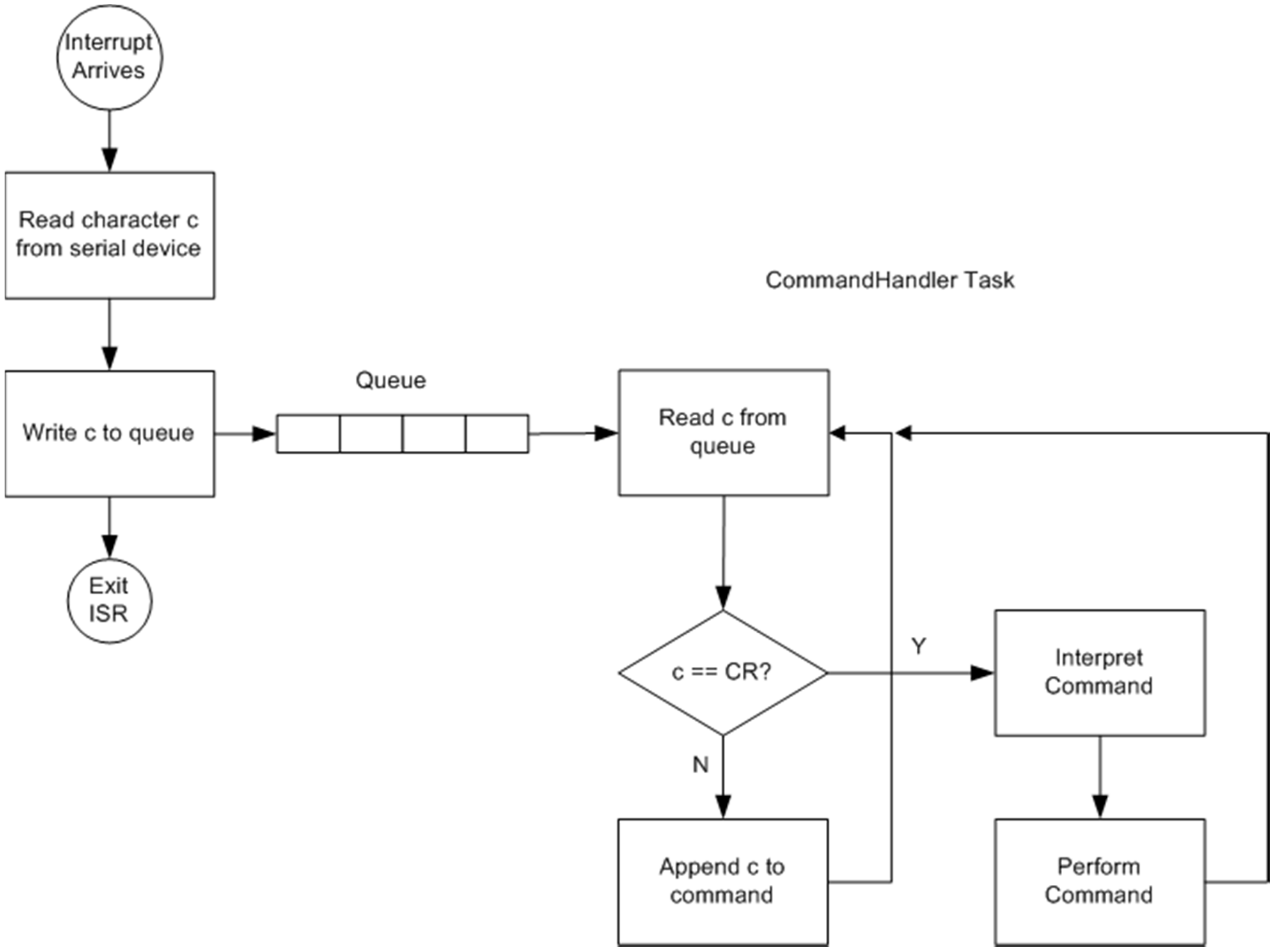
  - Serial port can only store one character at a time.
  - Characters may come quickly.
- **We can have three solutions!**

Solution I
ISR Does Everything

- **Do everything in the ISR.**
- **BAD solution!**
  - This ISR will take a really long time to complete!
  - High priority tasks are meanwhile suspended.

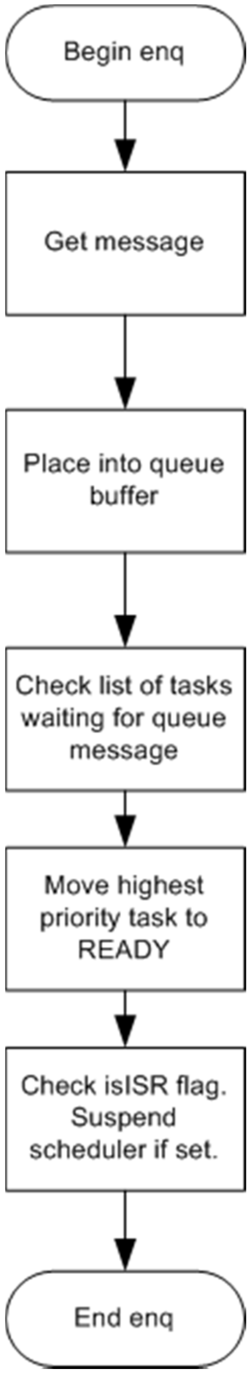


Solution II
ISR Does Almost Nothing



Solution II
ISR Does Almost Nothing

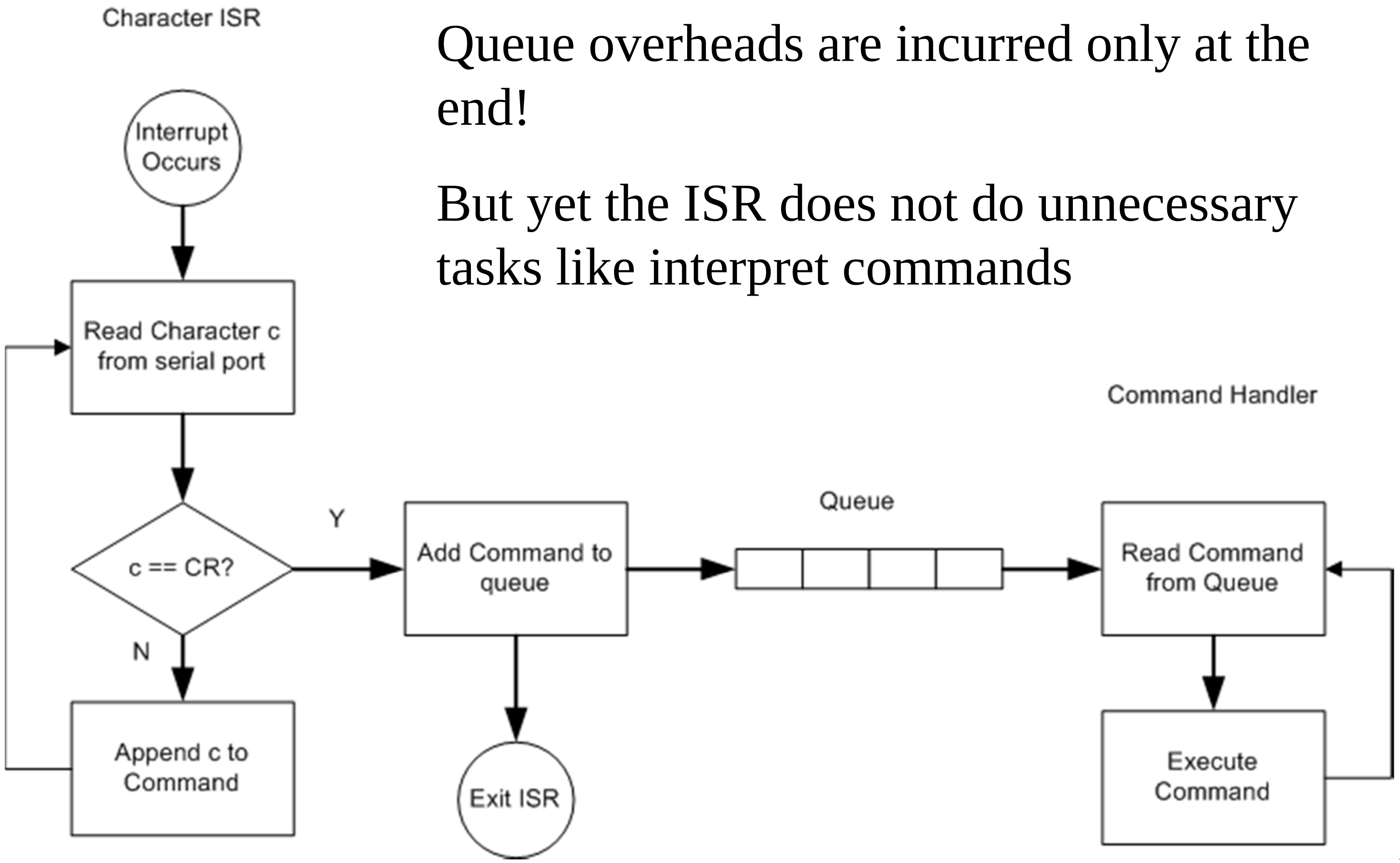
- **This looks like a good idea:**
  - ISR kept *really* short.
- **Trouble is:**
  - ISR sends lots of messages to the task
  - Sending messages takes time!
    - ✓**See flow chart on the side.**
- **This can cause a lot of overheads.**
  - ISR may even start to miss characters.


Begin enq
Get message
Place into queue buffer
Check list of tasks waiting for queue message
Move highest priority task to READY
Check isISR flag. Suspend scheduler if set.
End enq

Solution III
ISR Sends Message only at End

Queue overheads are incurred only at the end!

But yet the ISR does not do unnecessary tasks like interpret commands



Task Management

BASIC IDEAS

Some Basic Ideas

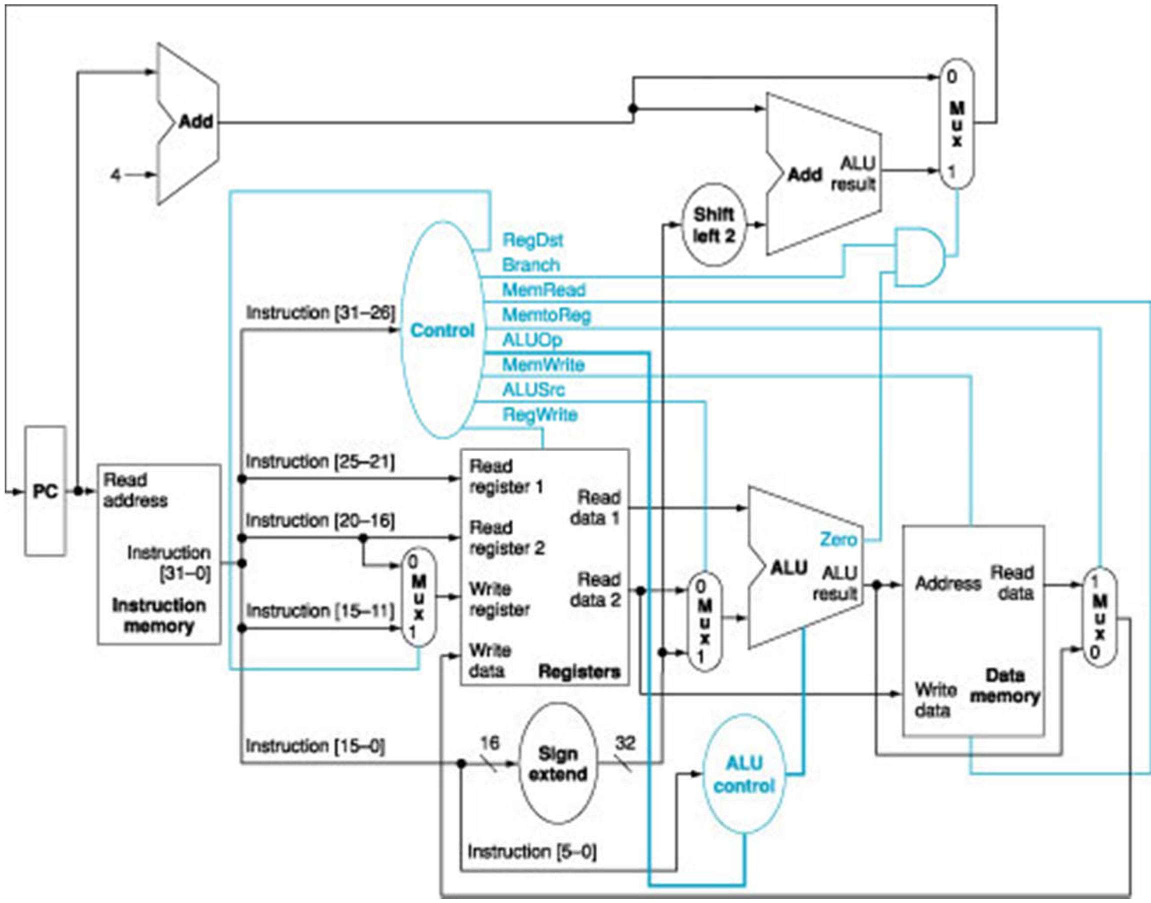
- **Since tasks are the work horses of a real-time system, we will now look at tasks in more detail.**
- **Before we do that however, let’s look at some basic ideas first:**
  - CPU Registers.
  - Kernel and User Spaces.

CPU Registers

- **You have already seen registers.**
  - E.g. the DDRB and PORTB registers that control direction and data on the Port B GPIO pins on the Atmega328.
    - ✓**These are known as "hardware" or "I/O" registers, and are in actual fact memory locations.**
    - ✓**These are located outside of the CPU core, and are used to control or configure hardware.**
  - Here we will look at a different type of registers.
    - ✓**CPU registers are temporary locations inside the CPU core, and are used to store intermediate computation results.**

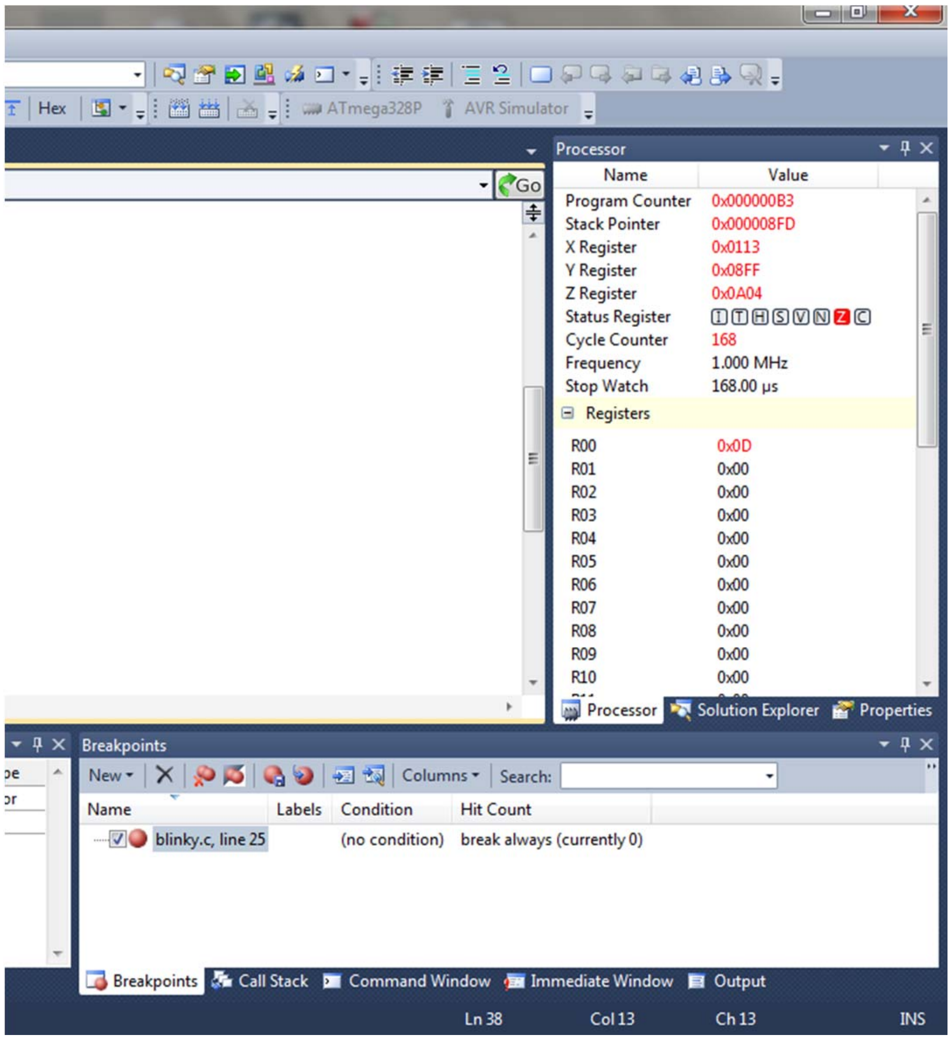
CPU Registers

- **Take ADD R0, R1, R2 as an example.**
  - This is assembly for R0=R1+R2.
    - ✓**Values are fetched from registers R1 and R2.**
    - ✓**The Arithmetic-Logic Unit (ALU) adds these values together.**
    - ✓**The answer is written to R0.**
  - R0, R1, and R2 are registers holding intermediate results!



CPU Registers

- **Registers are often labeled R0 to R31.**
  - Alternative schemes:
    - ✓**AX, BX, CX and DX in Intel processors.**
  - Some registers have aliases.
    - ✓**R26 and R27 are called "X", R28 and R29 are called "Y" and R30, R31 are called "Z" in the Atmega328.**
- **You can see the register contents in the "Processor" pane in the AVR Studio IDE.**



Processor

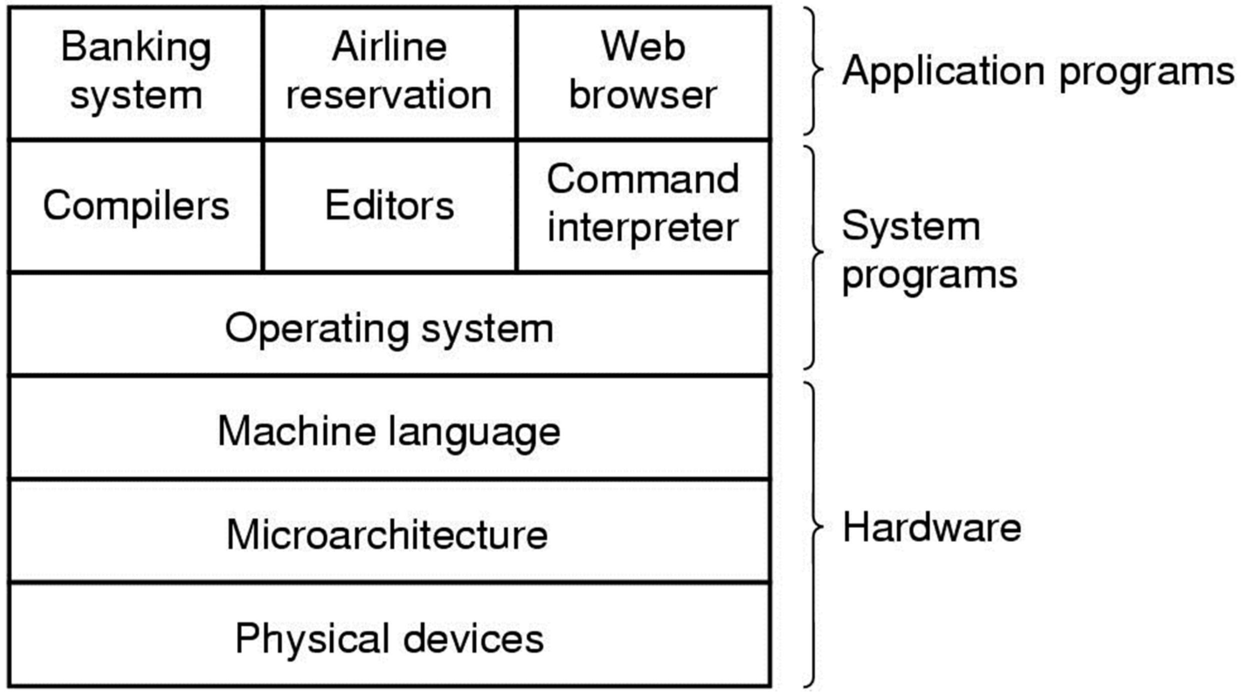
Name	Value
Program Counter	0x000000B3
Stack Pointer	0x000008FD
X Register	0x0113
Y Register	0x08FF
Z Register	0x0A04
Status Register	I T H S V N Z C
Cycle Counter	168
Frequency	1.000 MHz
Stop Watch	168.00 µs
⊟ Registers	
R00	0x0D
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00

Breakpoints

Name	Labels	Condition	Hit Count
☑ blinky.c, line 25		(no condition)	break always (currently 0)

Kernel and User Spaces

- **Kernel vs. User Spaces.**
  - The diagram below shows how the hardware, operating system and user programs are typically organized.

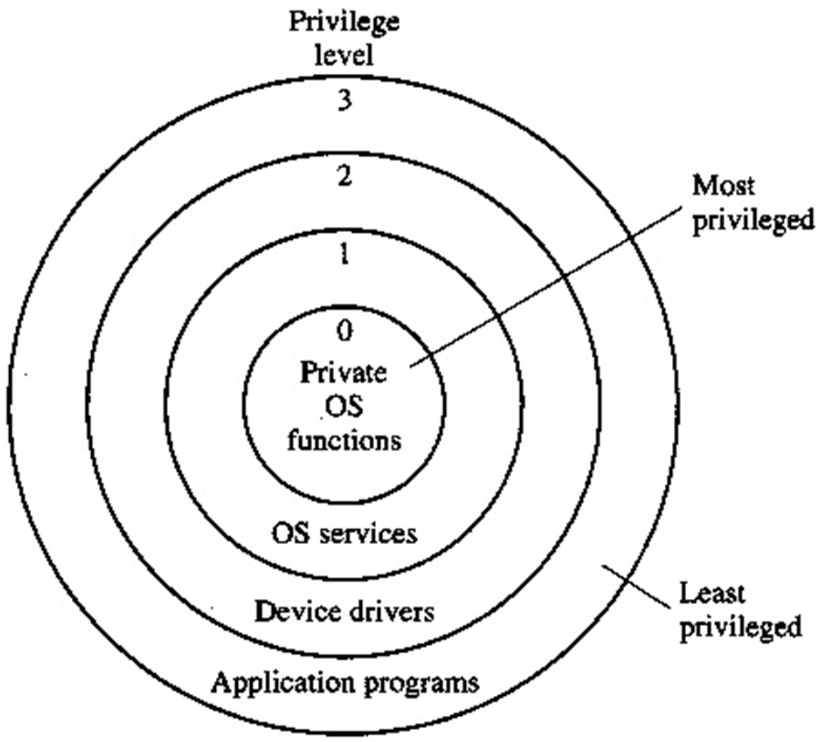


Kernel and User Spaces

- The basic services in the operating system (e.g. to manage memory, manage multiple processes, access hardware, etc), are held in a "kernel".
  - ✓**All instructions and data used by the operating system operate within the "kernel space".**
- User programs interact with the kernel through an application program interface (API).
  - ✓**All instructions and data used by user programs operate in the "user space".**
- Calls to the OS can involve switching between user and kernel spaces.
  - ✓**This means switching CPU privilege levels.**

Kernel and User Spaces

- **The diagram shows “privilege rings” of a the Intel x86 CPU.**
  - The innermost ring has the highest privileges, allowing access to all hardware and memory resources.
  - The kernel typically operates at the inner rings while user programs operate at the outermost ring.
- **Switching between ring levels takes a lot of time!**
  - This means switching from user space to kernel space is SLOW.
    - ✓**Avoid if possible!**



Task Management

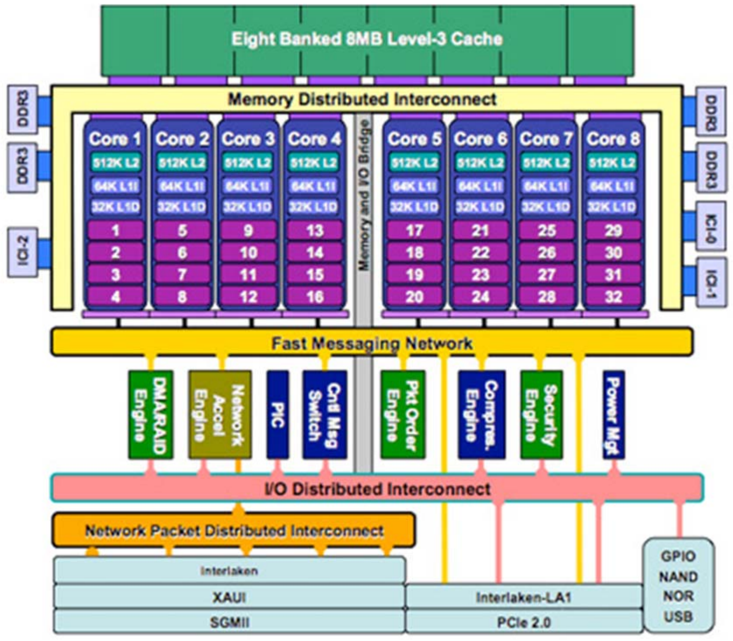
TASKS AND TASK MANAGEMENT

Tasks

- **While interrupts are the backbones of real-time systems, tasks are the workhorses.**
  - ISRs get information from sensors, trigger when uses press a button etc.
  - It is tasks that process these information or act on a button press.

The Process Model

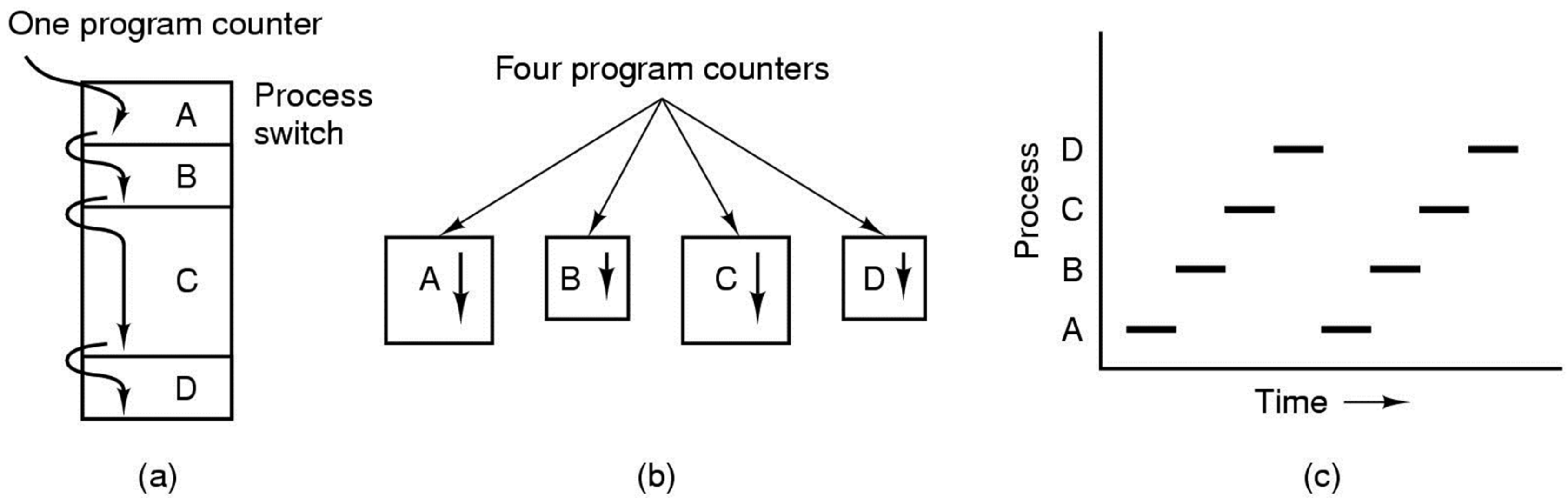
- **In this lecture we will assume a single processor with a single core.**
  - This is very typical of a microcontroller used in a real-time system.
    - ✓**Some exceptions include the ARM Cortex-A9 used in the iPhone.**
  - Most modern desktops however have "multi-core" processors:
    - ✓**Each microprocessor actually consists of multiple execution units.**



XLP832 Processor Block Diagram

The Process Model

- **The materials for this lecture come from Modern Operating Systems.**
  - This book uses the term "process" instead of "task". We will assume that they mean the same thing.
- **Since we have only a single-core single processor:**
  - At any one time, at most one process can execute.



The Process Model

- **Figure (b) shows what “appears” to be happening in a single processor system running multiple processes:**
  - There are 4 processes each with its own program counter (PC) and registers.
  - All 4 processes run independently of each other at the same time.

The Process Model

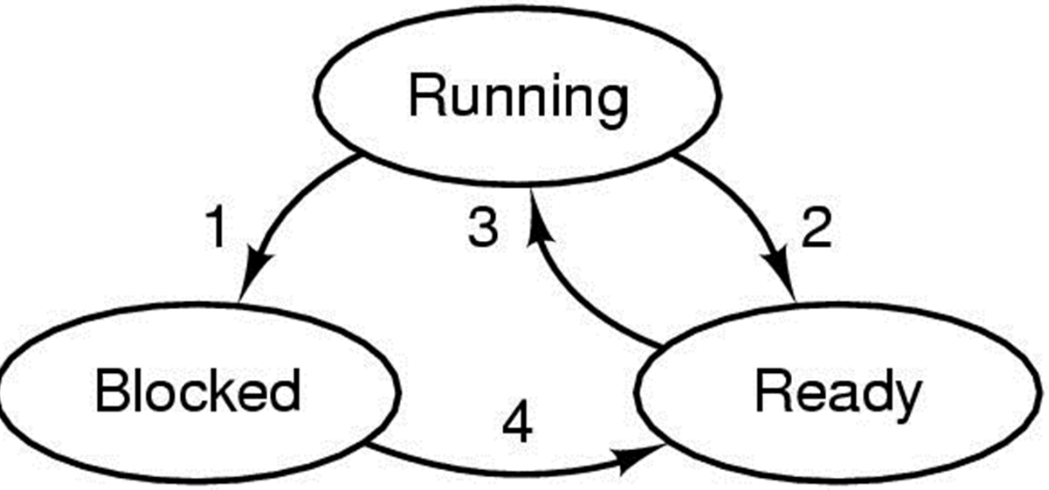
- **Figure (a) shows what actually happens.**
  - ▪There is only a single PC and a single set of registers.
  - ▪When one process ends, there is a "context switch" or "process switch":
    - ✓**PC, all registers and other process data for Process A is copied to memory.**
    - ✓**PC, register and process data for Process B is loaded and B starts executing, etc.**
  - ▪Figure (c) illustrates how processes A to D share CPU time.

Process States

- **A process can be in one of 3 possible states:**
  - Running
    - ✓**The process is actually being executed on the CPU.**
  - Ready
    - ✓**The process is ready to run but not currently running.**
    - ✓**A “scheduling algorithm” is used to pick the next process for running.**
  - Blocked.
    - ✓**The process is waiting for “something” to happen so it is not ready to run yet.**
    - ✓**E.g. include waiting for inputs from another process.**

Process States

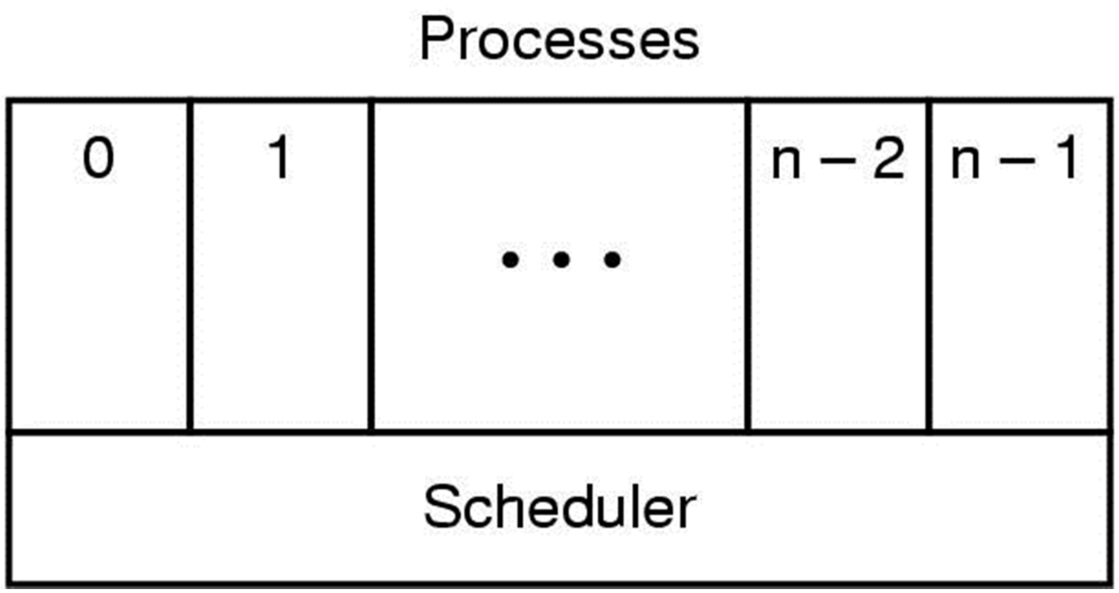
- **The diagram below shows the 3 possible states and the transitions between them.**



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

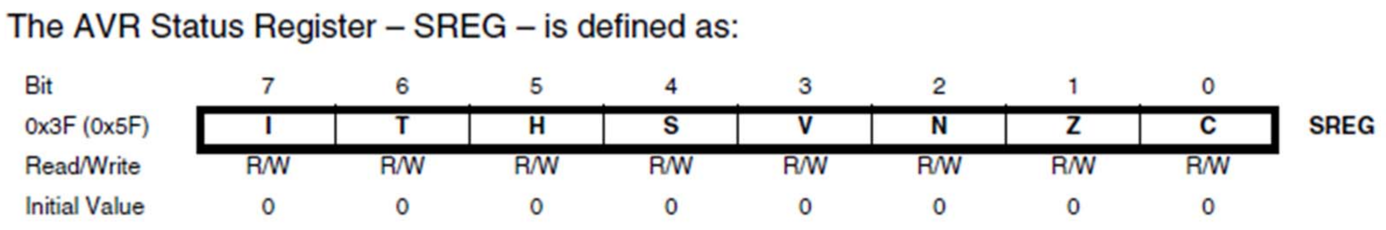
Process States

- **The figure below shows how the processes are organized.**
  - ▪The lowest layer selects (schedules) which process to run next.
    - ✓**This is subject to "scheduling policies" which we will look at in a later lecture.**



Switching between Processes

- **When a process runs, the CPU needs to maintain a lot of information about it. This is called the "process context".**
  - CPU register values.
  - Stack pointers.
  - CPU Status Word Register.
    - ✓**This maintains information about whether the previous instruction resulted in an overflow or a "zero", whether interrupts are enabled, etc.**
    - ✓**This is needed for branch instructions – assembly equivalents of "if' statements.**



The AVR Status Register – SREG – is defined as:

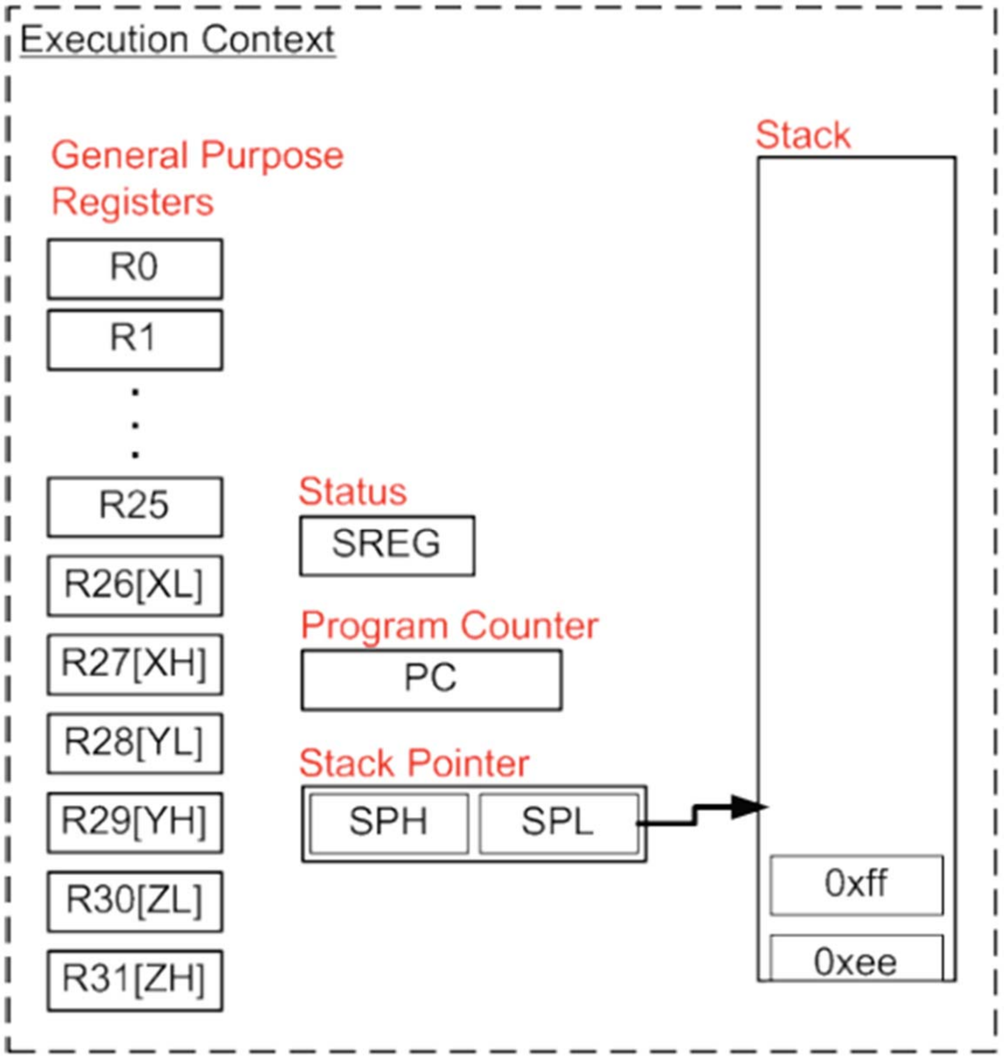
Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Switching between Processes

- **All of these values change as a process runs.**
- **When a process is blocked or put into a READY state, a new process will be picked to take control of the CPU.**
  - All the information for the current process must be saved!
  - The information for the new process must be loaded into the registers, stack pointer and status registers!
    - ✓**This is to allow the new process to run like as though it was never interrupted!**
- **This process is known as "context switching".**

Context Switching on the FreeRTOS Atmega Port

- **Each process is allocated a stack.**
  - ▪Exactly what you learnt in CG1103.
  - ▪The stack pointer are two 8-bit registers SPH and SPL, together forming a single 16-bit SP.
- **The diagram shows the complete Atmega context.**
  - ▪Registers R0-R31, PC.
  - ▪Status register SREG.
  - ▪Stack pointer SPH/SPL.



Context Switching on the FreeRTOS Atmega Port

- **FreeRTOS implements context saving in a macro called "portSAVE_CONTEXT".**

```
#define portSAVE_CONTEXT()\
asm volatile (\
      "push r0  \n\t"\            // Save R0
      "in r0, __SREG__   \n\t"\   // Read in status register SREG to R0
      "cli      \n\t"\            // Disable all interrupts for atomicity
      "push r0  \n\t"\            // Save SREG
      "push r1  \n\t"\            // Save R1
      "clr r1   \n\t"\            // AVR C expects R1 to be 0, so clear it.
      "push r2  \n\t"\            // Save R2 to R31
      ...
      "push r31 \n\t"\
      "in r26, __SP_L__  \n\t"\   // Read in stack pointer low byte
      "in r27, __SP_H__  \n\t"\   // and high byte
      "sts pxCurrentTCB+1, r27    \n\t"\   // And save it to pxCurrentTCB
      "sts pxCurrentTCB, r26      \n\t"\
      "sei      \n\t" : : :\               // Re-enable interrupts
);
```

Context Switching on the FreeRTOS Atmega Port

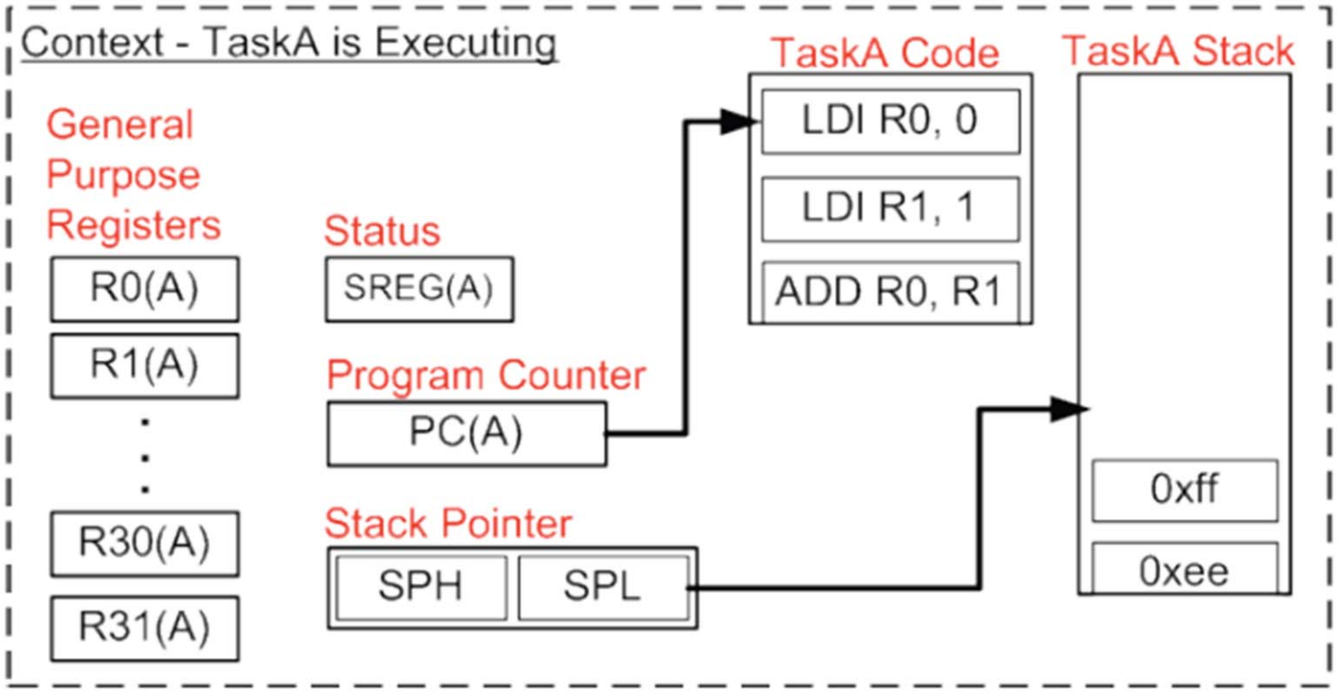
- **The reverse operation is portRESTORE_CONTEXT. The stack pointer for the process being restored must be in pxCurrentTCB.**

```
#define portRESTORE_CONTEXT()\
asm volatile (\
      "out __SP_L__, %A0 \n\t"\    // Copy SP_L and SP_H from the
      "out __SP_H__, %B0 \n\t"\    // pxCurrentTCB variable.
      "pop r31  \n\t"\             // Restore registers r31 to r1.
      ...
      "pop r0   \n\t"\             // Pop out SREG
      "out __SREG__, r0\n\t"\      // And restore it.
      "pop r0   \n\t": : "r" (pxCurrentTCB):\        // Restore R0

      );
```

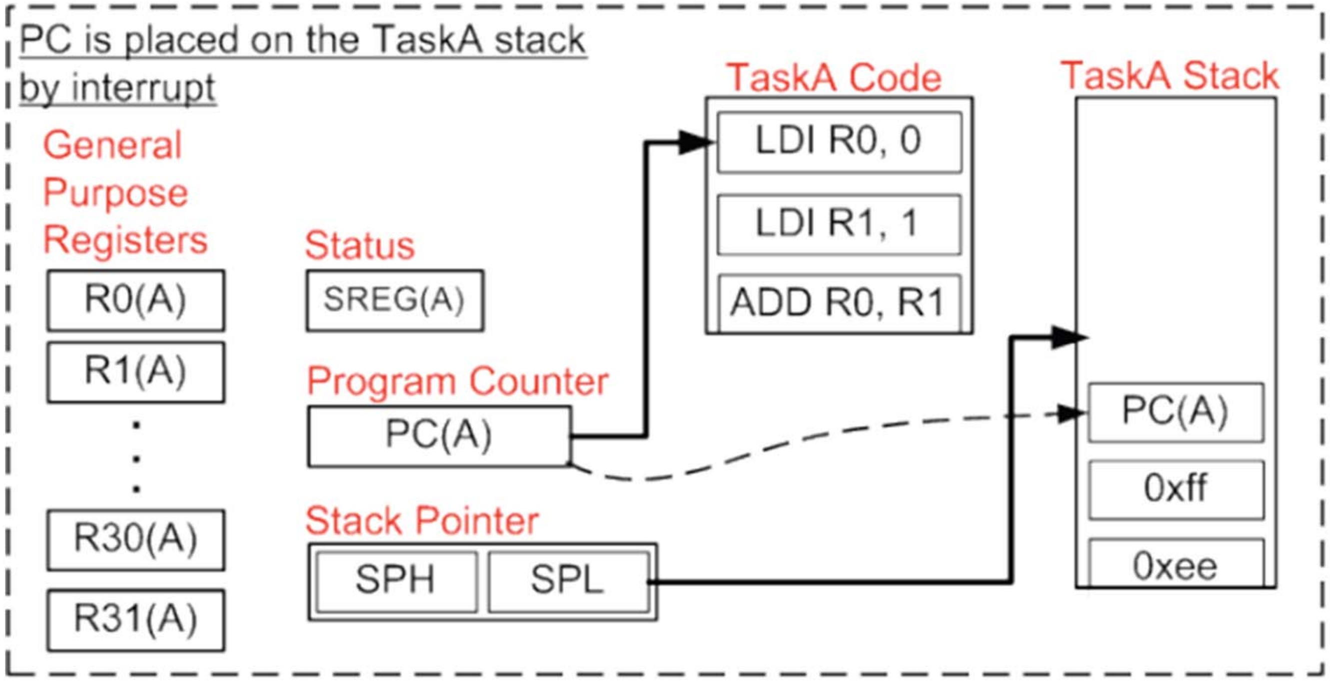
Context Switching on the FreeRTOS Atmega Port

- **We will now see step-by-step how this works.**
- **Assume that at first Task A is executing.**
  - PC would be pointing at Task A code, SPH/SPL pointing at Task A stack, Registers R0-R31 contain Task A data.



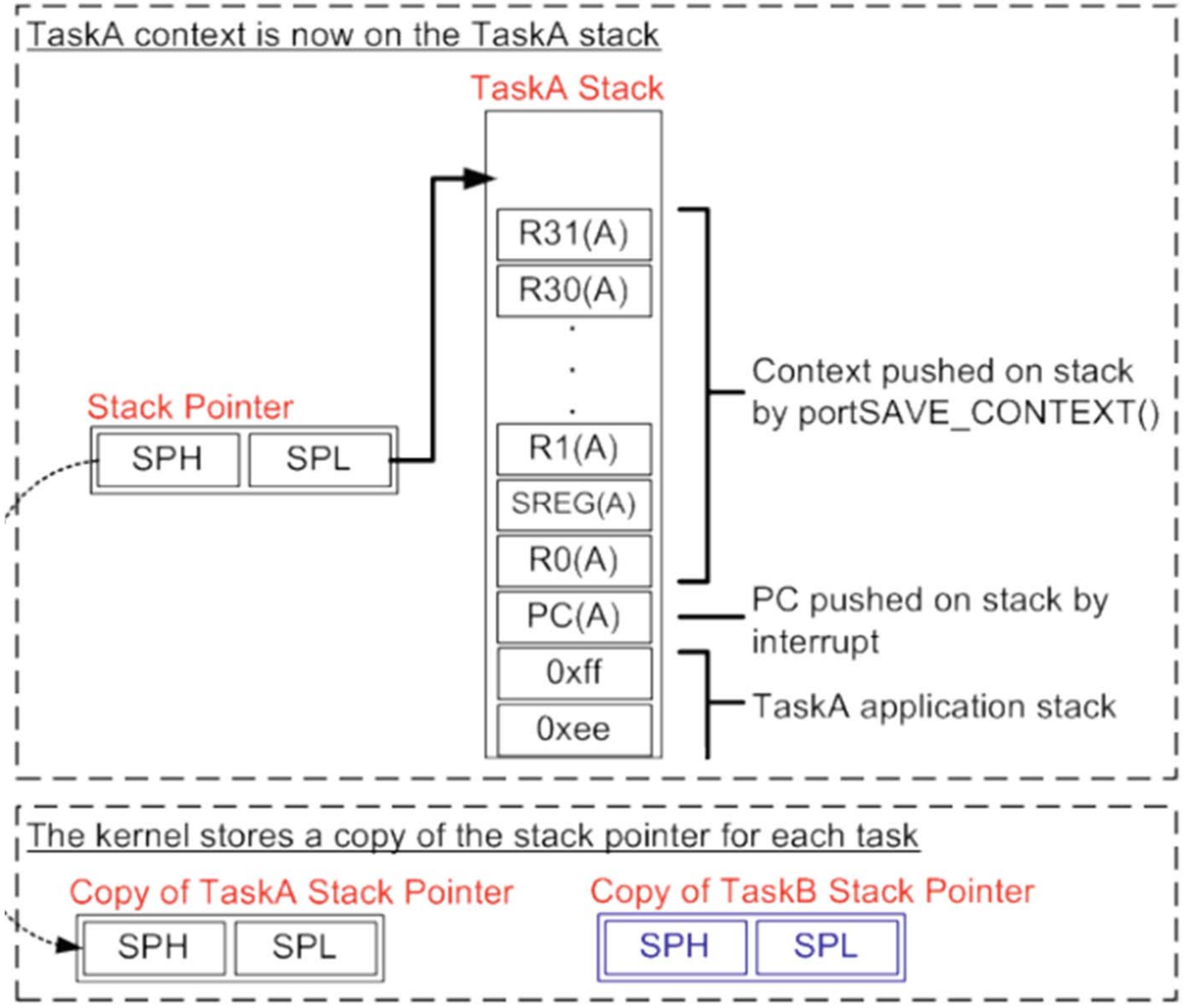
Context Switching on the FreeRTOS Atmega Port

- **FreeRTOS relies on regular interrupts from Timer 0 to switch between tasks. When the interrupt triggers, PC is placed onto Task A's stack.**



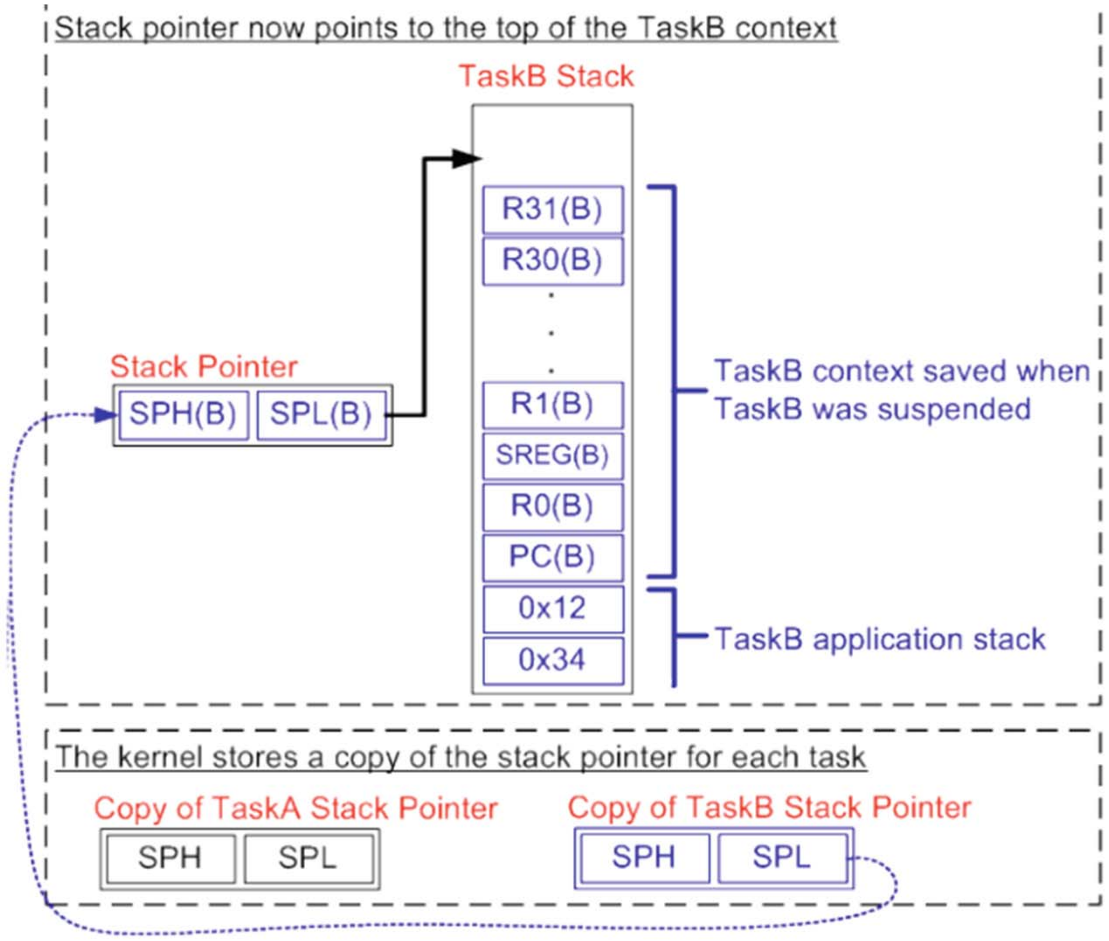
Context Switching on the FreeRTOS Atmega Port

- **The ISR calls portSAVECONTEXT, resulting in Task A's context being pushed onto the stack.**
- **pxCurrentTCB will also hold SPH/SPL _after_ the context save.**
  - ▪This must be saved by the kernel.


TaskA context is now on the TaskA stack
TaskA Stack
R31(A)
R30(A)
R1(A)
SREG(A)
R0(A)
PC(A)
0xff
0xee
Stack Pointer
SPH
SPL
Context pushed on stack by portSAVE_CONTEXT()
PC pushed on stack by interrupt
TaskA application stack
The kernel stores a copy of the stack pointer for each task
Copy of TaskA Stack Pointer
SPH
SPL
Copy of TaskB Stack Pointer
SPH
SPL

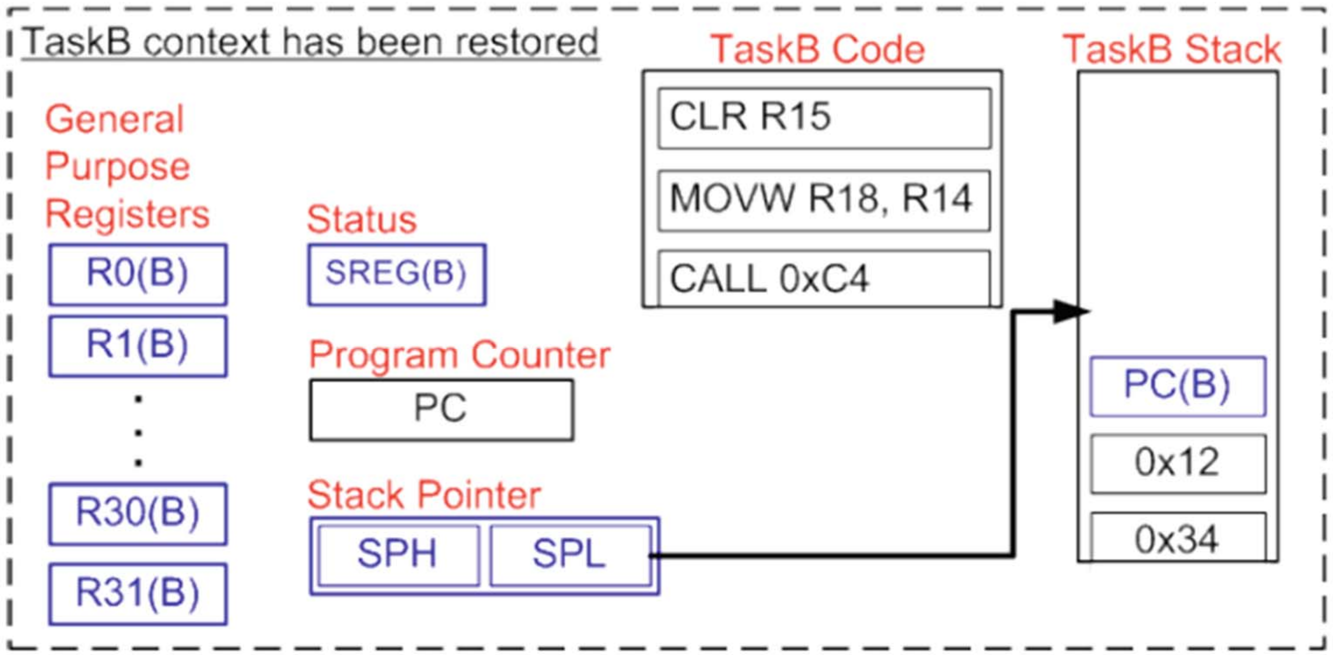
Context Switching on the FreeRTOS Atmega Port

- **The kernel then selects Task B to run, and copies its SPH/SPL values into pxCurrentTCB and calls portRESTORE_CONTEXT.**
  - The first two lines will copy pxCurrentTCB into SPH/SPL, causing SP to point to Task B's stack.


Stack pointer now points to the top of the TaskB context
TaskB Stack
R31(B)
R30(B)
R1(B)
SREG(B)
R0(B)
PC(B)
0x12
0x34
TaskB context saved when TaskB was suspended
TaskB application stack
Stack Pointer
SPH(B)
SPL(B)
The kernel stores a copy of the stack pointer for each task
Copy of TaskA Stack Pointer
SPH
SPL
Copy of TaskB Stack Pointer
SPH
SPL

Context Switching on the FreeRTOS Atmega Port

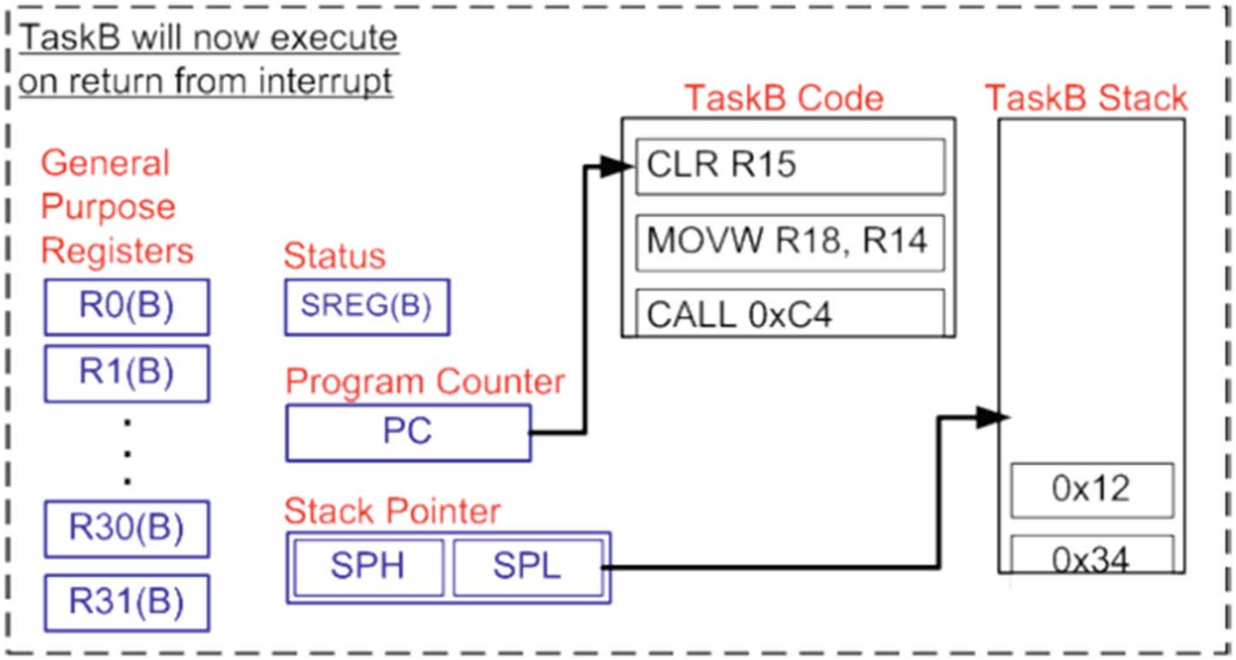
- **The rest of portRESTORE_CONTEXT is executed, causing Task B's data to be loaded into R31-R0 and SREG.**
  - Now Task B can resume like as though nothing happened!



Only the program counter remains on the stack.

Context Switching on the FreeRTOS Atmega Port

- **Only Task B's PC remains on the stack. Now the ISR exits, causing this value to be popped off onto the AVR's PC.**
  - PC points to the next instruction to be executed.
  - End result: Task B resumes execution, with all its data and SREG intact!



Context Switching on the FreeRTOS Atmega Port

- **Here we looked at context switching controlled by a timer.**
  - It can also be triggered by other things:
    - ✓**Currently running processed waiting for input.**
    - ✓**Currently running task blocking on a synchronization mechanism (see next lecture).**
    - ✓**Currently running task wants to sleep for a fixed period.**
    - ✓**Higher priority task becoming "READY".**
    - ✓…

Task Management

MODELING MULTI-PROGRAMMING

Modeling Multiprogramming

- **In a system running just one process, suppose the CPU is only 20% utilized.**
- **Then logically, in a system running 5 processes, the CPU will be 100% utilized!**
- **Sadly this is not the case.**
  - Much of the time the task might spend on I/O.
  - While the task is waiting for I/O it will be in the BLOCKED state and cannot run.
- **A better model would be to assume that the CPU spends *p* percent of the time doing I/O.**

Modeling Multiprogramming

- **The probability of *n* processes all doing I/O at the same time is therefore p^n**
- **The CPU utilization will therefore be $1-p^n$**
- **The figure below shows CPU utilization for differing *p* and *n*.**
  - Diagram omitted. Reserved for your tutorial question. ☺

Modeling Multiprogramming

- **The model given here is overly simplistic. It assumes that the tasks are independent. This cannot be the case:**
  - Since there is only 1 CPU, even when a task becomes READY, it cannot run if the CPU is BUSY.
- **A more accurate model can be derived using queuing theory.**
- **This model is nonetheless useful for giving "first-cut" guesses at CPU utilization.**

Summary

- **In this lecture we looked at:**
  - How real-time applications run.
  - How multiple tasks are managed.