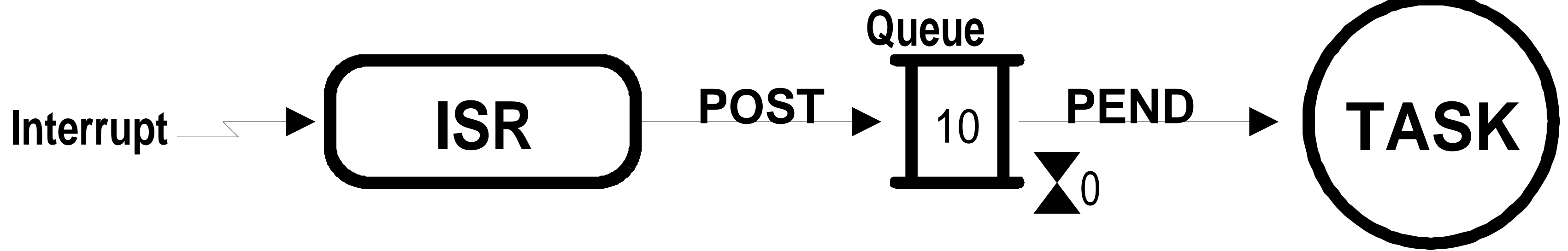
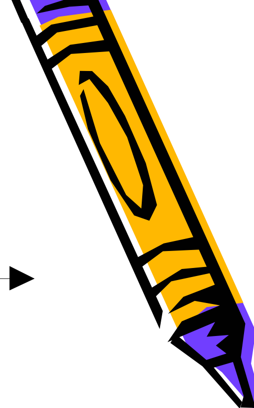

Interrupt
ISR
POST
Queue
10
0
PEND
TASK

Interrupts

- An interrupt is a hardware mechanism used to inform the CPU that an asynchronous event had occurred.
- The CPU saves the context of the current running task and jumps to the corresponding service routine (ISR).
- Common interrupts: clock tick (triggering scheduling), I/O events, hardware errors.
- Disabling interrupts affects interrupt latency.
- The ISR processes the event, and upon completion of the ISR, the program returns to
  - The background for a foreground/background system
  - The interrupted task for a non-preemptive kernel
  - The highest priority task ready to run for a preemptive kernel



TIME

TASK

ISR #1

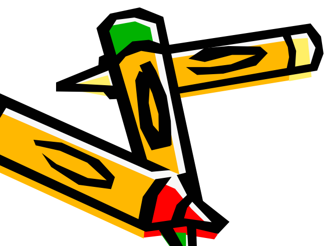
ISR #2

ISR #3

Interrupt #1

Interrupt #2

Interrupt #3

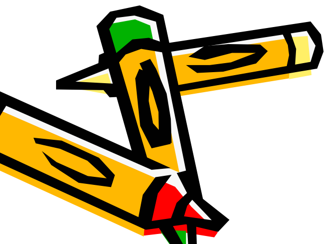


Interrupt Latency

- Real-time systems disable interrupts to manipulate critical sections of code and enable interrupts when critical section has executed.

- The longer interrupts disabled, the higher the interrupt latency is.

interrupt latency =
max. amount of interrupts are disabled +
Time to start executing the first instruction in the ISR



Interrupt Response

- Interrupt response: the time between the reception of the interrupt and the **start of the user code** that handles the interrupt – accounts for all the overhead involved in handling an interrupt

- For a foreground/background system and a non-preemptive kernel:

  Response time = Interrupt latency + Time to save the CPU's context

- For preemptive kernel

  Response time = Interrupt latency + Time to save the CPU's context + Execution time of the kernel ISR entry function

(to notify the kernel that an ISR is in progress and allows kernel to keep track of interrupt nesting, OSIntEnter( ) in uC/OS-2)

Interrupt Recovery

- The time required for the processor to return to the interrupted code.

- For a foreground/background system and a non-preemptive kernel:

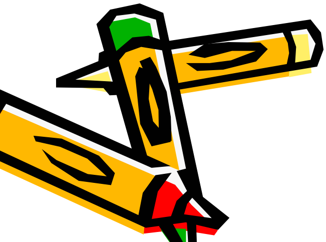
  Interrupt recovery
  = Time to restore the CPU's context
  + Time to execute the return from interrupt instruction

- For preemptive kernel:

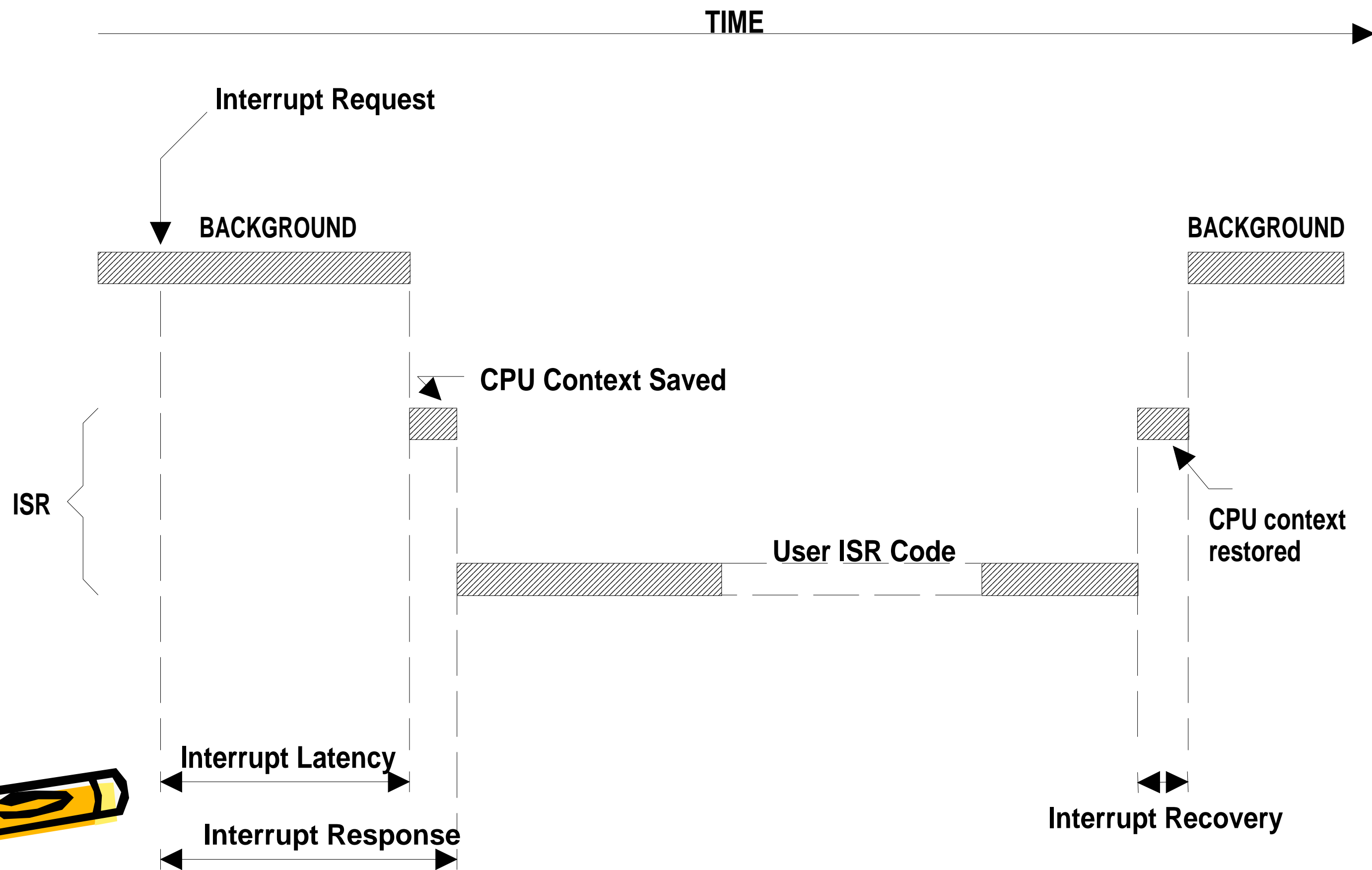
  Interrupt recovery
  = Time to determine if a higher priority task is ready
  + Time to restore the CPU's context of the highest priority task
  + Time to execute the return from interrupt instruction

ISR Processing Time

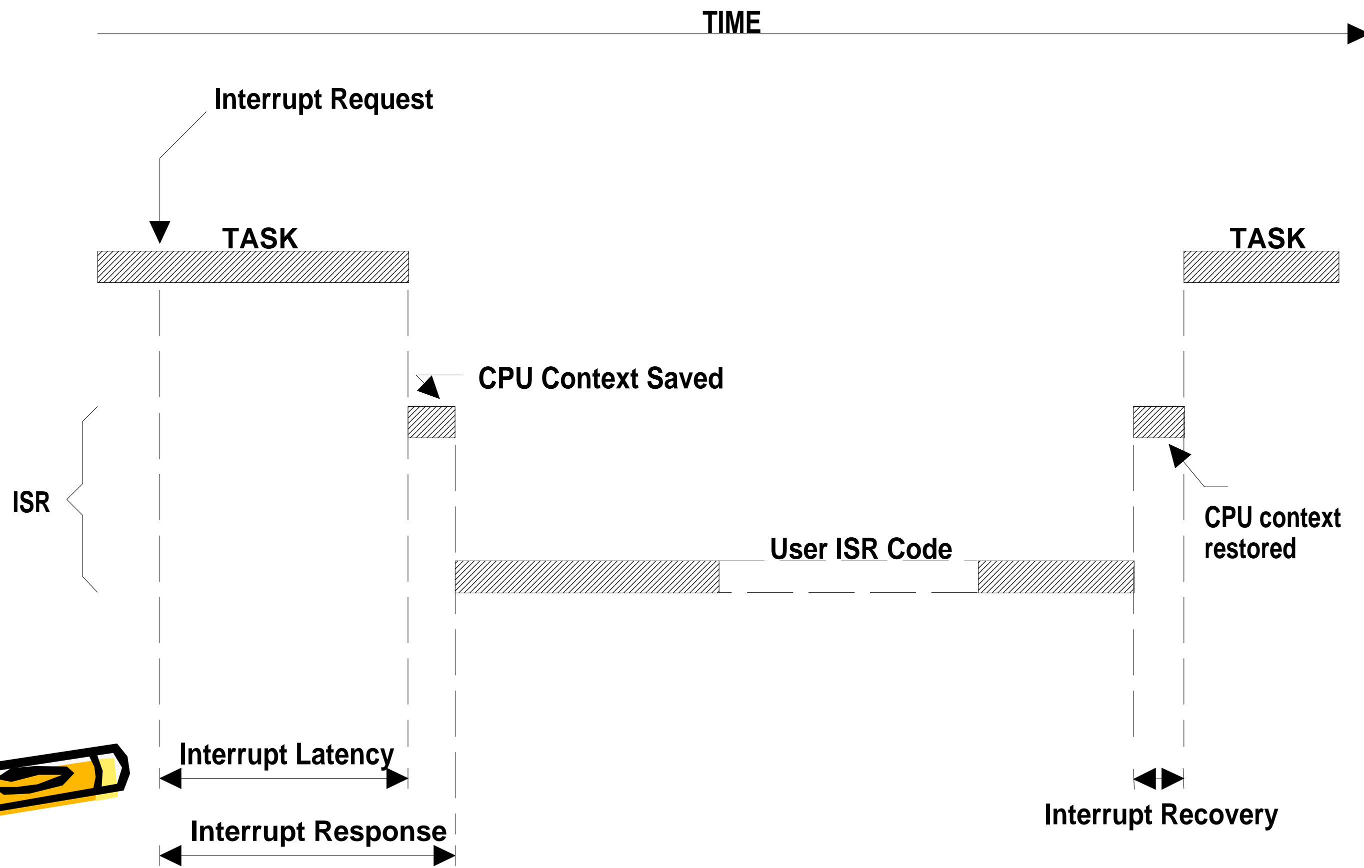
- ISRs should be as short as possible.
  - there are no absolute limits on the amount of time for an ISR.
- If the ISR code is the most important code that needs to run at any given time, it could be as long as it needs to be.
- In most cases, the ISR should
  - Recognize the interrupt
  - Obtain data or status from the interrupting device
  - Signal a task to perform a actual processing
- Overhead involved in signaling task
  - the processing of the interrupt



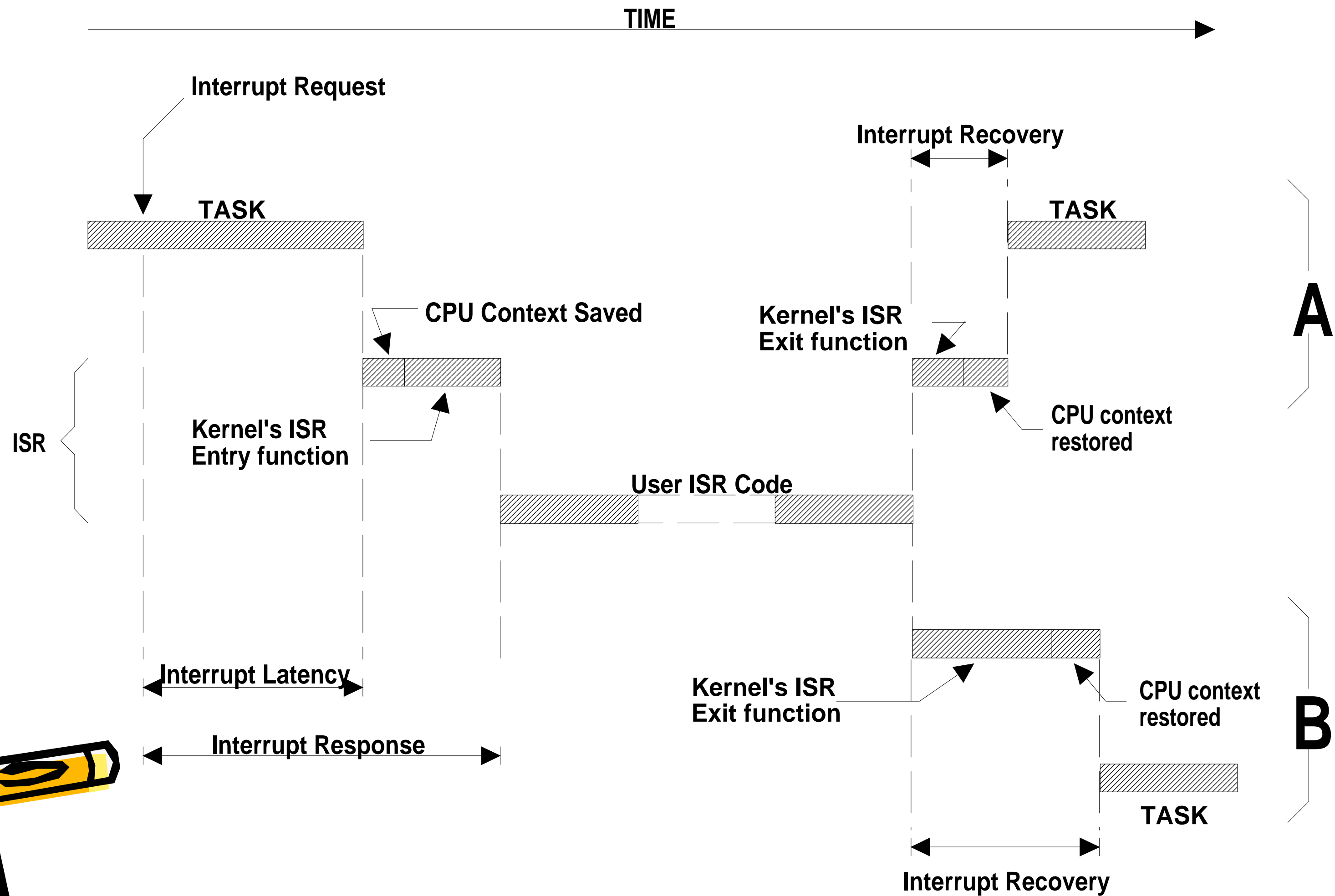
Interrupt latency, response, and recovery (Foreground/Background)



Interrupt latency, response, and recovery (Non-preemptive kernel)



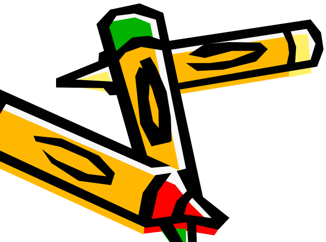
Interrupt latency, response, and recovery (Preemptive kernel)



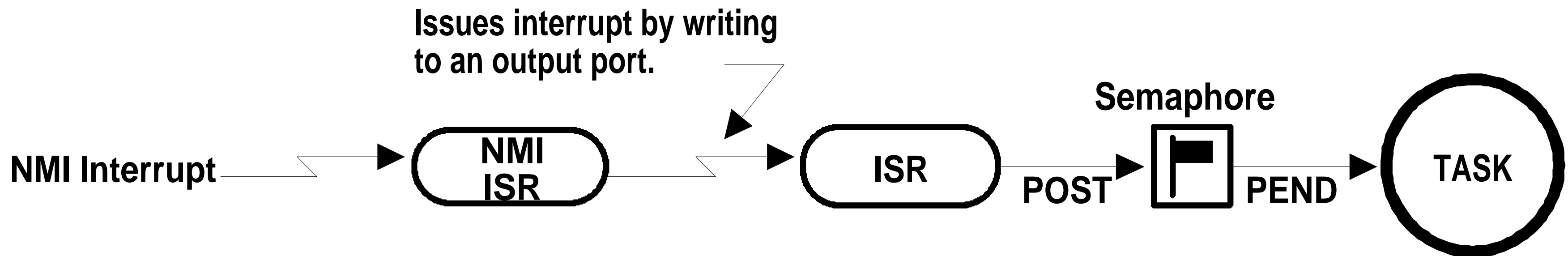
Non-Maskable Interrupts

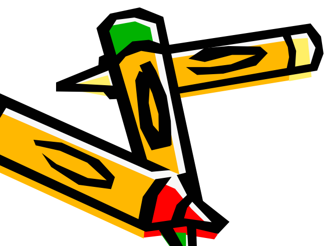
- NMI's can not be disabled.
  - They are generally reserved for drastic events, such as the power-source is almost exhausted.

- You can not use kernel services to signal a task in ISR's of NMI's.
  - Since interrupts can not be disabled in the ISR of an NMI.
  - The size of global variable under this situation must be atomic. (i.e., byte, word, dword)
  - Or, we can trigger another hardware interrupt which's ISR uses kernel services to signal the desired task.



Signaling a task from the ISR of an NMI





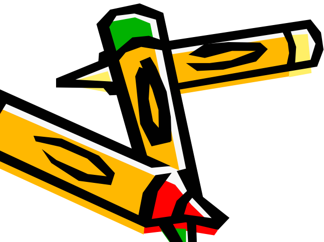
Non-Maskable Interrupts

Interrupt latency
= Time to execute longest instruction
+ Time to start executing the NMI ISR

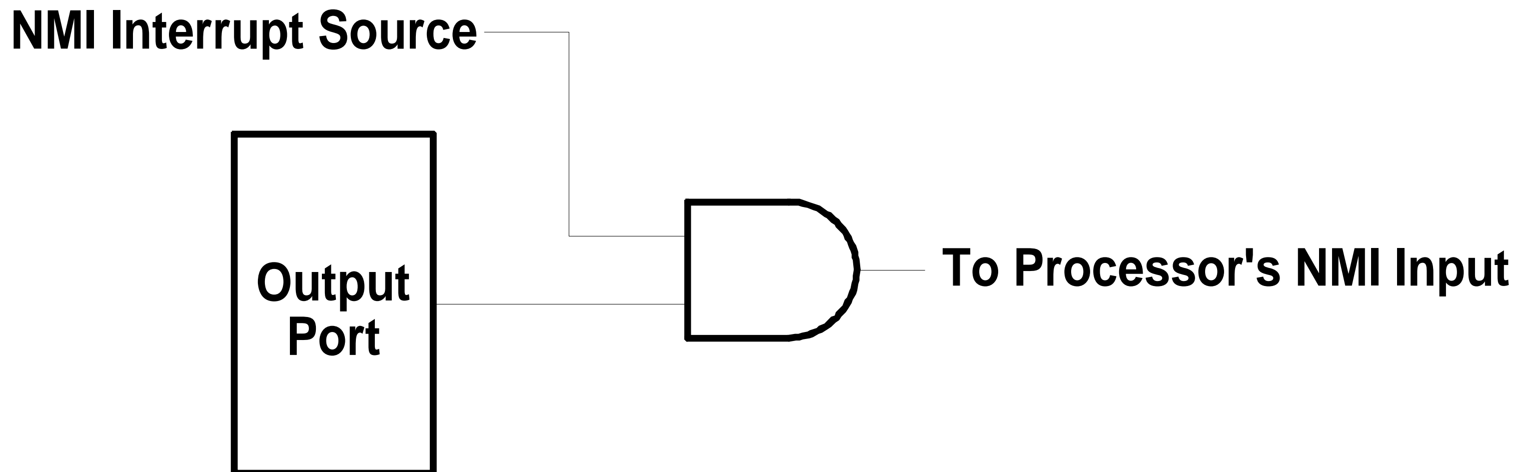
Interrupt response
= Interrupt latency
+ Time to save the CPU's context

Interrupt recovery
= Time to restore the CPU's context
+ Time to execute the return from interrupt instruction

- NMI can still be disable by adding external circuits.

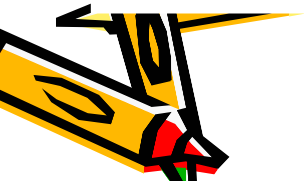


Disabling NMI’s



Clock Tick

- Clock tick is a periodically hardware event (interrupt) generated by a timer.
- The kernel utilize the timer to delay tasks and to periodically perform scheduling.
- The higher the tick rate,
  - the better the responsiveness is.
  - the better the schedulability is.
    - Blocking due to clock tick resolution.
  - the higher the overhead is.



20 mS

Tick Interrupt

Tick ISR

All higher priority tasks

Call to delay 1 tick (20 mS)

Call to delay 1 tick (20 mS)

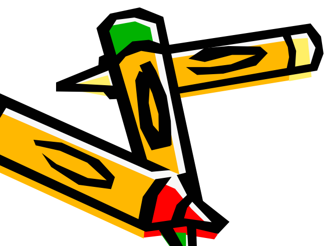
Call to delay 1 tick (20 mS)

Delayed Task

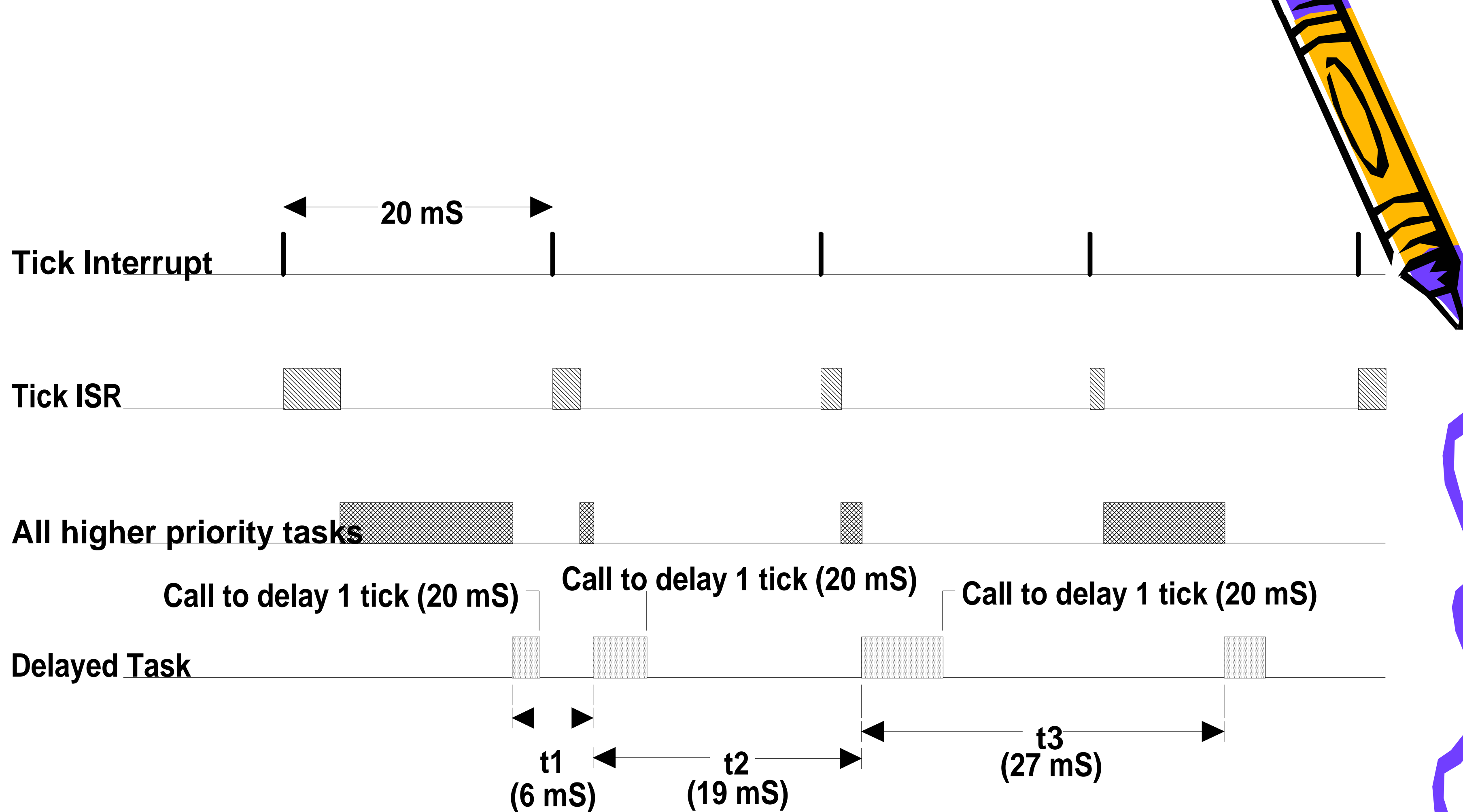
t1
(19 mS)

t2
(17 mS)

t3
(27 mS)



- A task delaying itself for one tick
- Higher priority tasks and ISRs execute prior to the task, which needs to delay for 1 tick
- A jitter occurs.



- The execution times of all higher priority tasks and ISRs are slightly less than 1 tick
- As a result, if you need to delay at least one tick, you must specify one extra tick

20 mS

Tick Interrupt

Tick ISR

All higher priority tasks

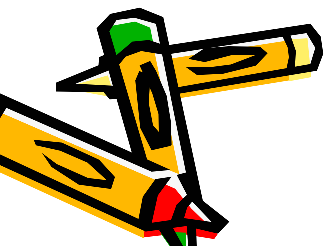
Call to delay 1 tick (20 mS)

Call to delay 1 tick (20 mS)

Delayed Task

t1
(40 mS)

t2
(26 mS)



- The execution times of all higher priority tasks and ISRs are more than 1 clock tick.
- The task that tries to delay for 1 tick actually executes two ticks later and violates its deadline.

Memory Requirements

- Most real-time applications are embedded systems. Memory requirements must be analyzable.
- A preemptible kernel requires more RAM/ROM space.
- Code size (ROM) = kernel size + application size
- RAM requirements can be significantly reduced if
  - Stack size of every task can be differently specified
  - A separate stack is used to handle ISR's. (uC/OS-2 doesn't, DOS does)
- RAM requirement = application requirement + kernel requirement + SUM(task stacks + MAX(ISR nesting))
- RAM requirement = application requirement + kernel requirement + SUM(task stacks) + MAX(ISR nesting)
  - If a separate stack is prepared for ISR's.

Memory Requirements

- We must be careful on the usages of tasks' stacks:
  - Large arrays and structures as local variables.
  - Recursive function call.
  - ISR nesting.
  - Function calls with many arguments.

Advantages and Disadvantages of Real-Time Kernels

- A real-time kernel (RTOS) allows real-time applications to be designed and expanded easily.
  – Functions can be added without requiring major changes to the software.
- The use of RTOS simplifies the design process by splitting the application code into separate tasks.
- With a preemptive RTOS, all time-critical events are handled as quickly and as efficiently as possible.
- An RTOS allows you to make better use of your resources by providing you with valuable services – semaphores, mailboxes, queues, time delays, timeouts, etc.
  - Extra cost of the kernel.
  - More ROM/RAM space.
  - 2 to 4 percent additional CPU overhead.
  - Cost of the RTOS: $70 ~ $30,000 !
  - The maintenance cost: $100 ~ $5,000 per year !

Real-Time Systems Summary

	Foreground/Background	Non-Preemptive Kernel	Preemptive Kernel
Interrupt Latency (Time)	MAX(Longest instruction, User int. disable) + Vector to ISR	MAX(Longest instruction, User int. disable, Kernel int. disable) + Vector to ISR	MAX(Longest instruction, User int. disable, Kernel int. disable) + Vector to ISR
Interrupt response (Time)	Int. latency + Save CPU's context	Int. latency + Save CPU's context	Interrupt latency + Save CPU's context + Kernel ISR entry function
Interrupt recovery (Time)	Restore background's context + Return from int.	Restore task's context + Return from int.	Find highest priority task + Restore highest priority task's context + Return from interrupt
Task response (Time)	Background	Longest task + Find highest priority task + Context switch	Find highest priority task + Context switch
ROM size	Application code	Application code + Kernel code	Application code + Kernel code
RAM size	Application code	Application code + Kernel RAM + SUM(Task stacks + MAX(ISR stack))	Application code + Kernel RAM + SUM(Task stacks + MAX(ISR stack))
Services available?	Application code must provide	Yes	Yes