CpE 213
Digital Systems Design
Interrupts

Lecture 21

Wednesday 11/16/2005

Overview

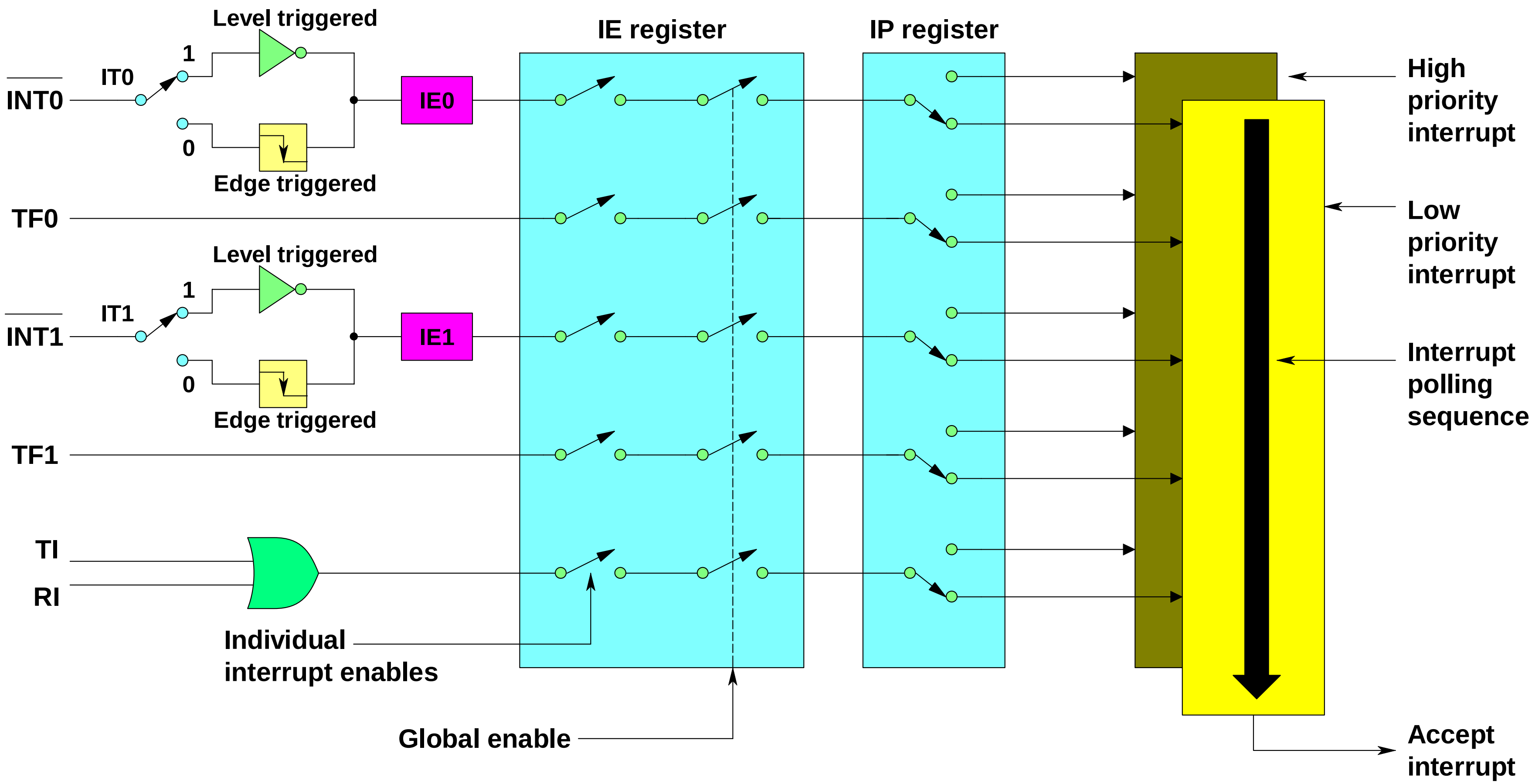
- Greater good
- Interrupts (Chapter 11) continued
- Multitasking

Greater Good

- Blood drive today in TJ South Lounge
  - Ends at 5pm
  - First time donors need picture ID
  - You get a free T-shirt
  - Tomorrow from 2 to 8 pm in Havener Center
- Cutest baby contest at Bookstore
  - Each vote costs you a penny
  - Proceeds go to Women and Children's Shelter

Interrupts on the 8051
Chapter 11

Overview of 8051 Interrupt Structure



Another Example

```
#include <REG51.H>
unsigned char count=0;
void counter(void) interrupt 0 {
   count++;
}
main(){
   unsigned char x=0;
   int i;

   IT0=0;
   INT0=1;
   PX0=0;
   EX0=1;
   EA=1;
```

(continued on next slide)

Example continued

```
for(i=1;i<=10;i++) {
      x++;

      if(x==5){
            x=0;
            INT0=1;
            INT0=0;
            INT0=1;
      }
  }
}
```

Example in ASM

```
;declare segments
mydata segment data
mycode segment code

;declare variables
rseg mydata
  x: DS 1
  count: DS 1
  stack: DS 20

;place jump to main at 1st code mem location
CSEG AT 0000H
  JMP main ; will jump to main function
```

Example continued

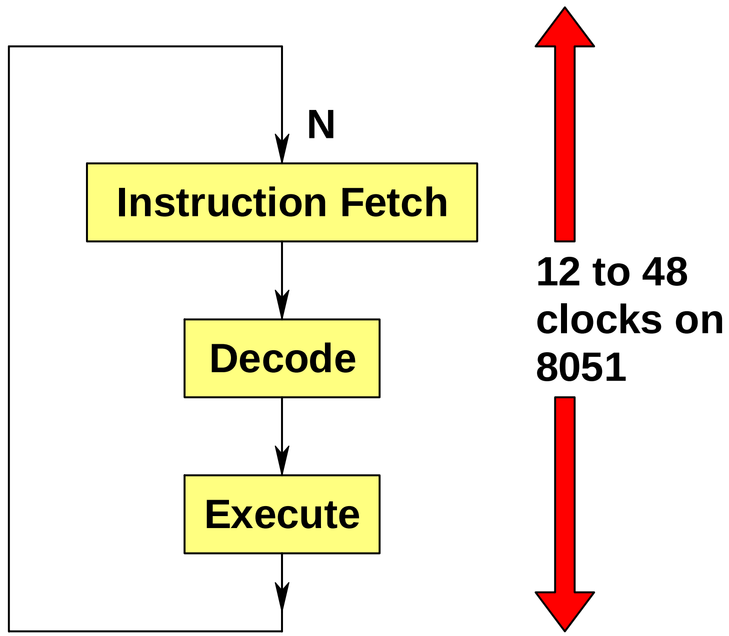
```
CSEG AT 0003H
       PUSH ACC
       PUSH PSW
       MOV A,count
       ADD A,#1
       MOV count, A
       POP PSW
       POP ACC
       RETI

rseg mycode
 main: MOV x,#00H
       mov count,#00H
       MOV SP, #stack
```

Example continued

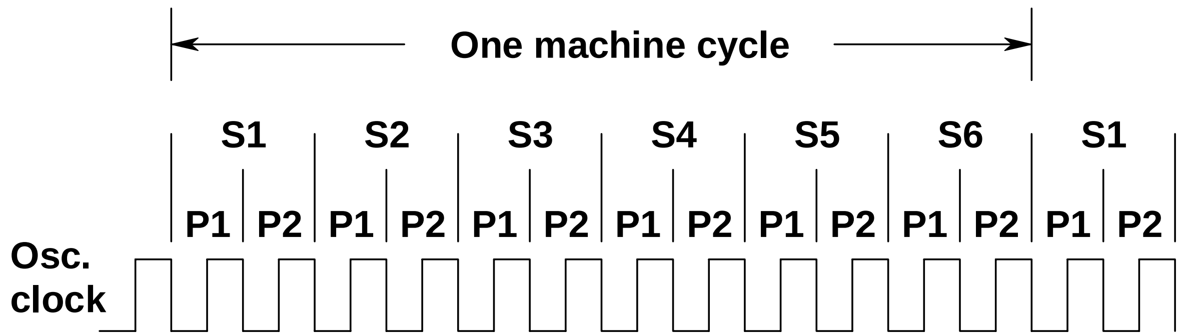
```
    SETB IT0
    SETB INT0
    CLR PX0
    SETB EX0
    SETB EA

 while:      INC x
    MOV A,#128
    CJNE A,x,while
    SETB INT0
    CLR INT0
    SETB INT0
    JMP while
END
```

Basic Instruction Execution

- Instruction execution proceeds by
  - (1) Instruction fetch
  - (2) Instruction decode
  - (3) Instruction execute
- On the 8051 this takes from one to four machine cycles, where each cycle lasts 12 clock cycles.
- This means that it can take from one to four machine cycles to start processing an interrupt after it is detected.



8051 Machine Cycle



- Each 8051 machine cycle consists of twelve clock cycles.
- It consists of a sequence of 6 states, numbered S1 through S6. Each state lasts for two oscillator periods.
- Each state is divided into a Phase 1 half and a Phase 2 half.
- Thus, a machine cycle takes 12 oscillator periods or 1 μs if the oscillator frequency is 12 MHz.

8051 Interrupt Timing

- Interrupts are sampled and latched during S5P2 (State 5 Phase 2) of each machine cycle.
- They are polled on the next cycle.
- If an interrupt exists, it will be accepted if ALL of the following conditions are satisfied:

  (1) no other interrupt of equal or higher priority is in progress.

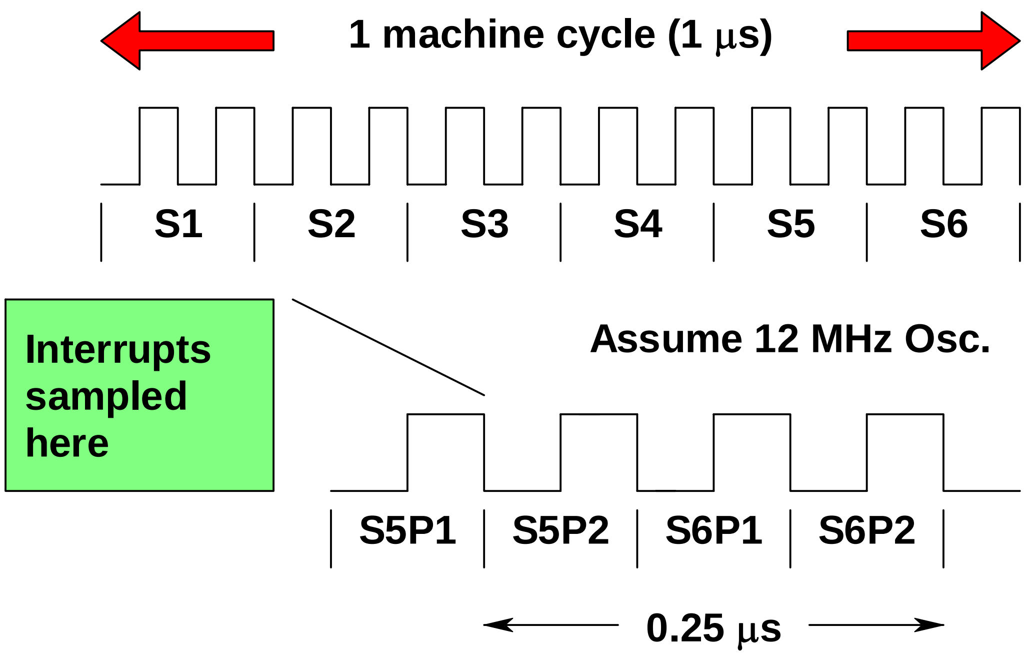
  (2) the polling cycle is the last cycle of an instruction.

  (3) the current instruction is not RETI, and does not access the IE or IP registers.
- Condition 2 ensures that the current instruction will be completed before the ISR begins.

8051 Interrupt Timing

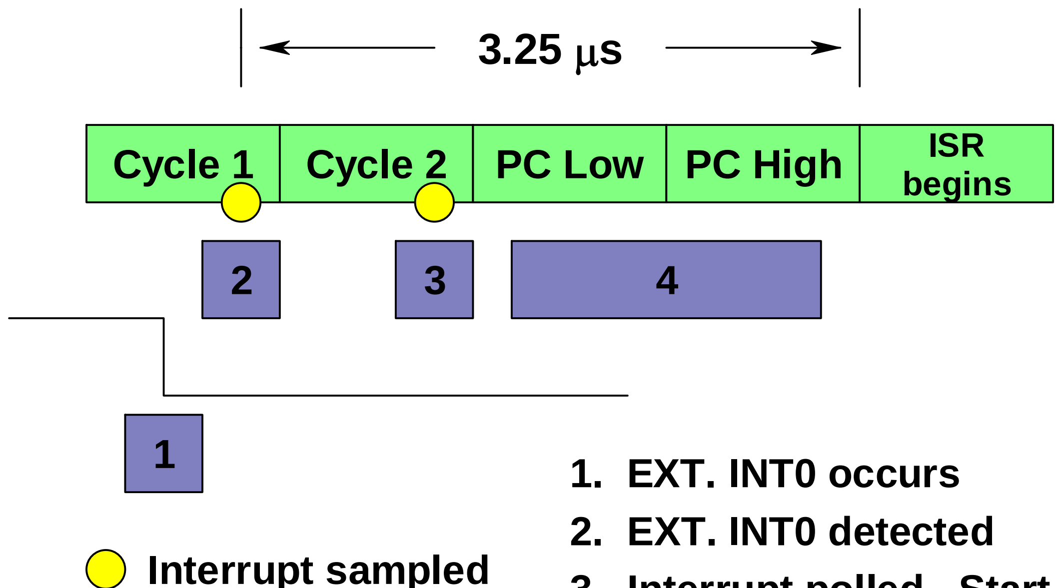
- Failure of condition 3 implies that at least one more instruction will be executed before the ISR is started.
- If even one of the conditions is not satisfied, the interrupt is blocked.
- After the processor recognizes the interrupt, it takes 2 more cycles to save the Program Counter (PC) and jump to the ISR.
- *Interrupt Latency* is the time elapsed from when an interrupt is generated to when the interrupt service routine begins execution.

8051 Interrupt Timing (Cont.)



- Interrupts are sampled at S5P2, which is 0.25 μs before the end of the machine cycle.

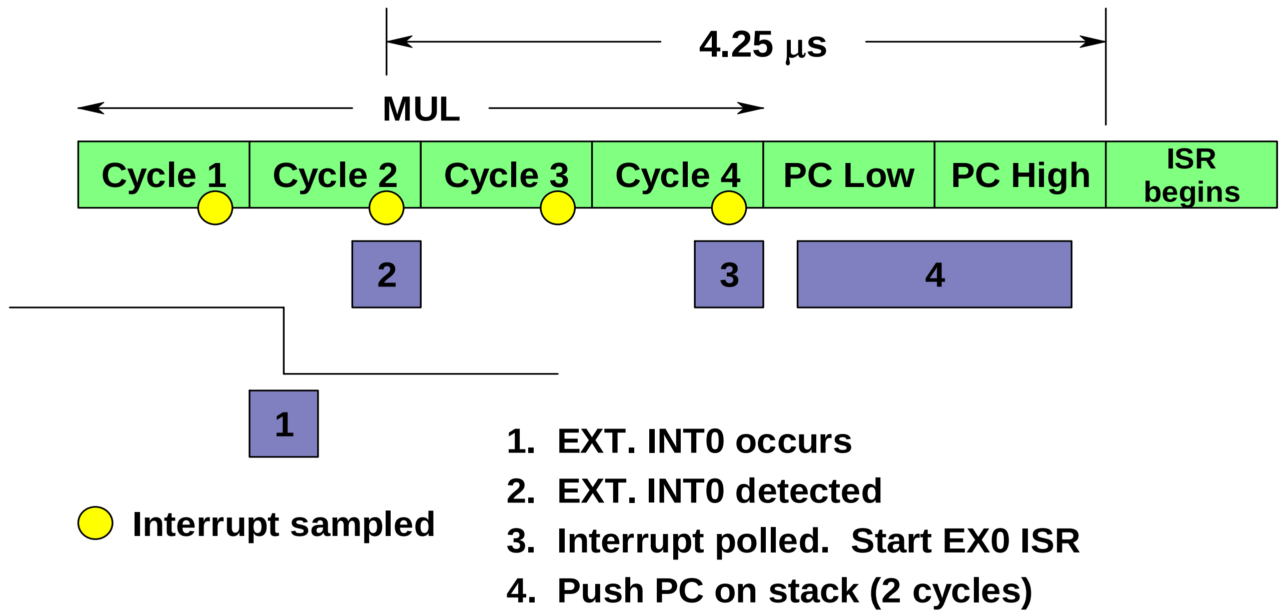
Best-Case Interrupt Latency



1. EXT. INT0 occurs
2. EXT. INT0 detected
3. Interrupt polled. Start EX0 ISR
4. Push PC on stack (2 cycles)

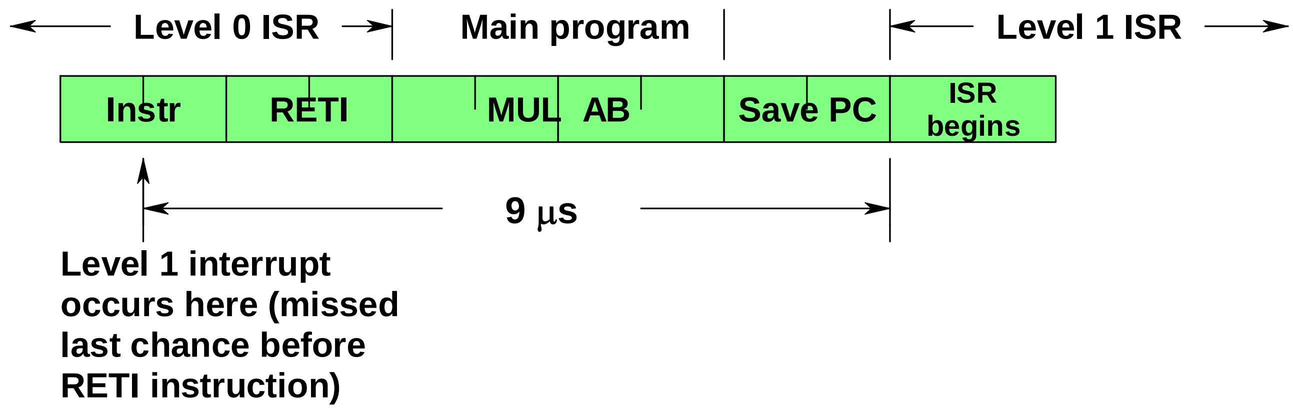
- The shortest interrupt latency is 3.25 μs, assuming a 12 MHz crystal.

Typical Interrupt Latency



- Interrupt event occurs between cycles 1 and 2 of a MUL instruction.
- The event is detected 0.25 μs before the end of cycle 2.
- It is polled during cycle 4.
- Two more cycles are taken to push the PC onto the stack.
- The interrupt latency in this example is 4.25 μs, assuming a 12 MHz crystal.

Worst-Case Interrupt Latency



1. Level 1 event occurs just after sample in second to last cycle of instruction (0.25 μs).
2. Interrupt is sampled in last cycle of instruction (1 cycle)
3. Interrupt polled in the last cycle of RETI (RETI takes 2 cycles).
4. RETI allows one more instruction: 4 cycles for MUL.
5. Plus two cycles to save PC

6. Equals 9.25 μs worst-case, assuming 12 MHz crystal.

What is the absolute worst-case latency?

Important Considerations

- Consider this scenario:
  - Subroutine FOO stores a variable in memory location 0020H and plans to use it later.
  - FOO is called by the main program, begins executing, and stores its data in 0020H.
  - An interrupt occurs and the ISR starts executing. The ISR calls FOO with different data than in the main program.
  - FOO uses location 0020H; most likely changing its contents, finishes executing, and returns.
  - The ISR returns and the original version of FOO resumes executing, but the data in 0020H has changed.

Register Protection is Important

- One very important rule applies to all interrupt routines:
  - Interrupts must leave the processor in the same state it was in when the interrupt initiated.
- That means if your ISR uses any register, you must ensure that the value of the register is the same at the end of the ISR as it was at the beginning.

Register Protection

- In general, your interrupt routine must protect the following registers:
  - PSW
  - DPTR (DPH/DPL)
  - ACC
  - B
  - Registers R0 - R7
- This is generally accomplished with a PUSH and POP sequence.
- Remember that the PSW consists of many individual bits that are set by various 8051 instructions.
- It is generally a good idea to protect the PSW in your ISR.

Register Protection Example

Assume this example is an ISR for INT0:

```
        ORG   200H
EXT0:   PUSH  ACC
        PUSH  PSW
        MOV   A,#0FFH
        ADD   A,#02H
        .
        .
        POP   PSW
        POP   ACC
        RETI
```

Common Problems

- If you are using interrupts and your program is crashing or does not seem to be performing as you expect, always review the following interrupt-related issues:
  - Register Protection: Make sure you save any registers altered by the ISR (crucial!).
  - Forgetting to restore protected values: Make sure you POP the same number of values off the stack as you PUSH onto it.
  - Use of RET instead of RETI: Verify that ISRs are terminated with the RETI instruction.
  - Keep the ISR short: Remember that low-priority interrupts can be delayed by the time taken to service high-priority interrupts. To improve the latency, keep ISRs short.

Multitasking

Multitasking

- Main (background) with interrupts (foreground) is a simple multitasking system
- Preemptive vs. non-preemptive tasking
  - non-preemptive: tasks executed in order
  - preemptive: one task can interrupt another
  - like single level vs. multilevel priority interrupts

Non-Preemptive Tasking

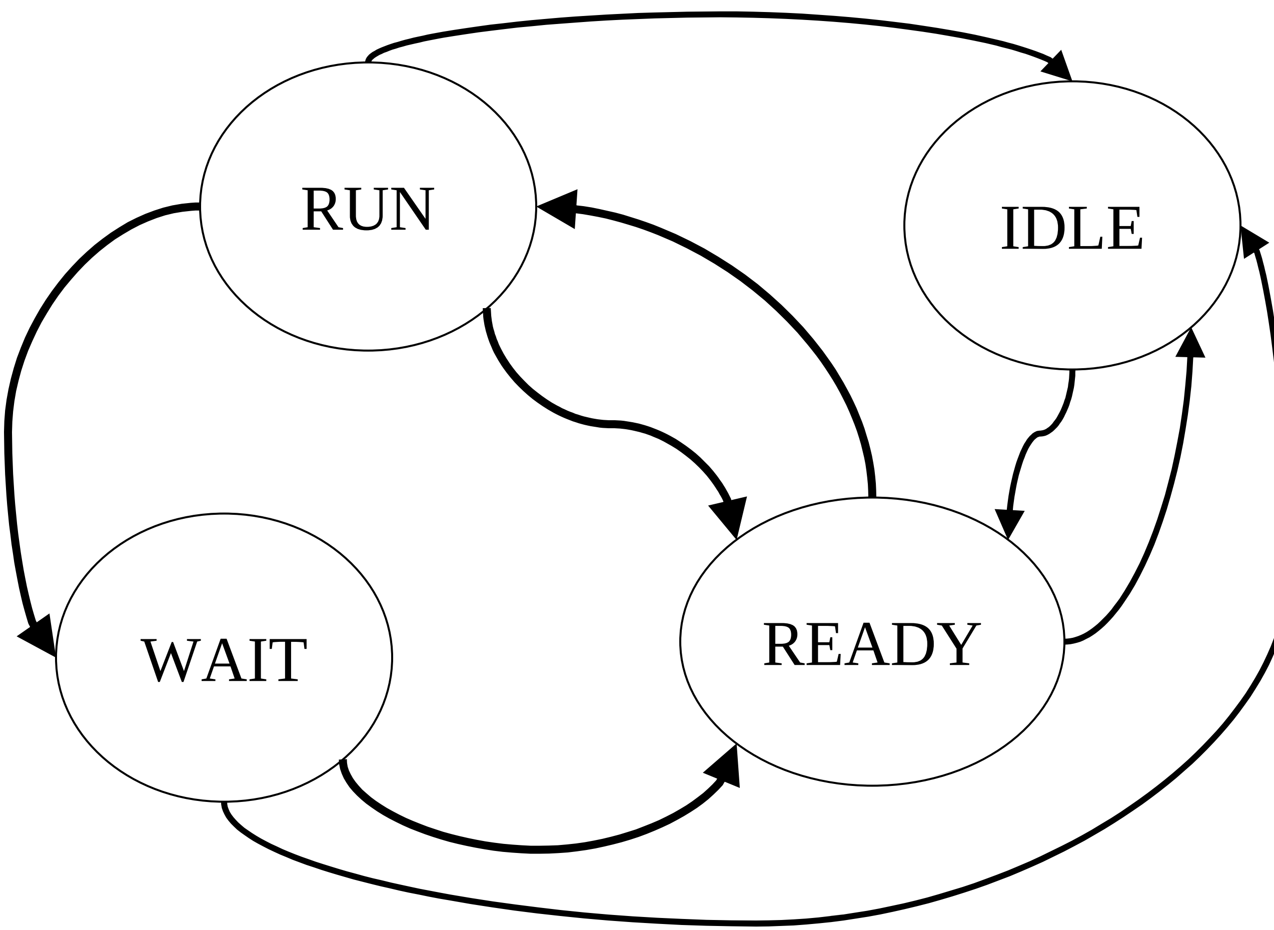
- Tasks executed in sequential order
- Sometimes called a ‘message loop’
- Examples:
  - X - windows
  - Microsoft Windows™
- Tasks must be well-behaved
- A task executes until it must wait
- Easy for one task to ‘hog’ the CPU

Pre-emptive Tasking

- One task may interrupt another
- Tasks are assigned a priority
- A task executes until it must wait or a higher priority task becomes ready
- Examples:
  - QNX (a *real time* unix for x86 architecture)
  - RTX51/RTXtiny (RTOS for 8051)
  - VRTX (RTOS for large microprocessors)

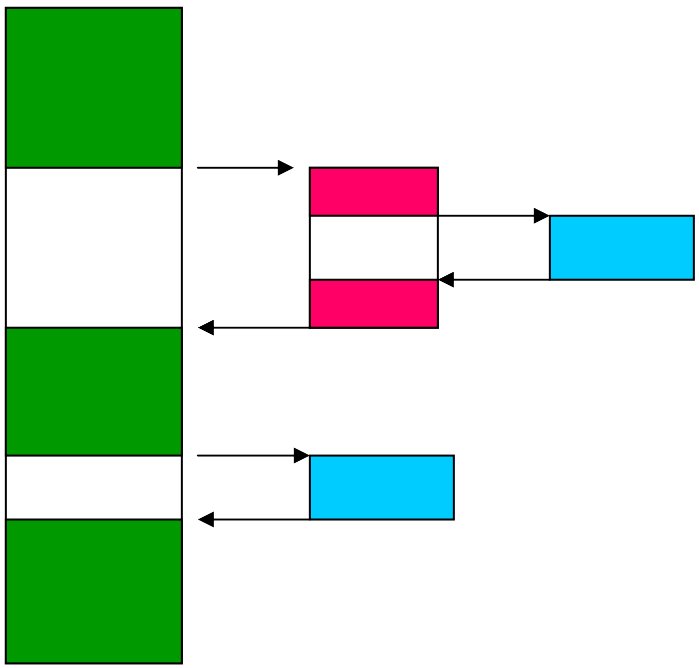
Task States

- Controlled by *task scheduler*

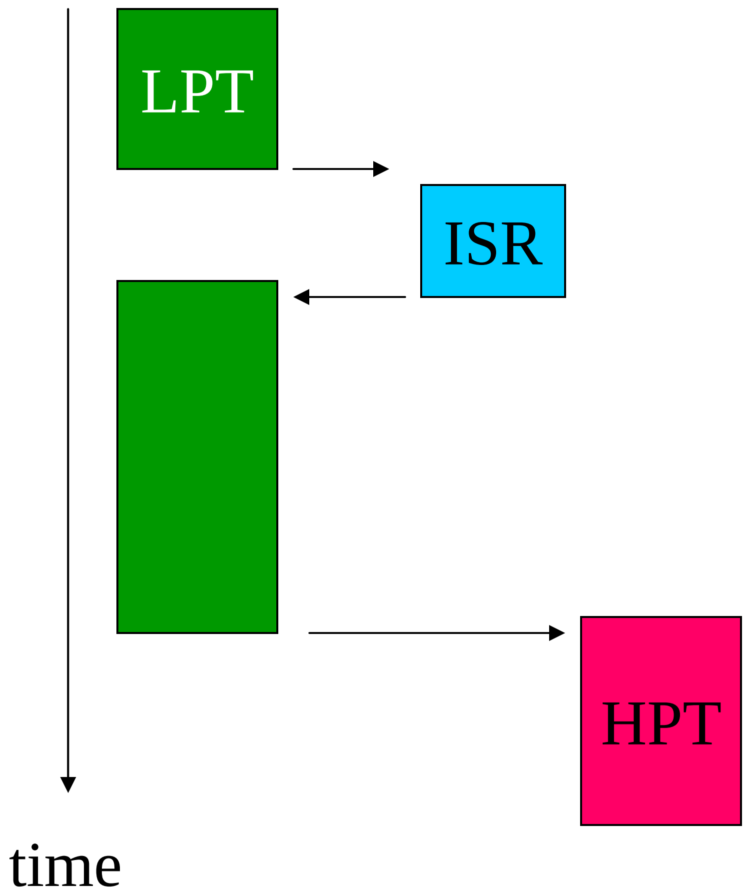


Main Program with interrupts

- Background task (main)
- Foreground interrupt service routines
  - timer ticks
  - I/O interrupts

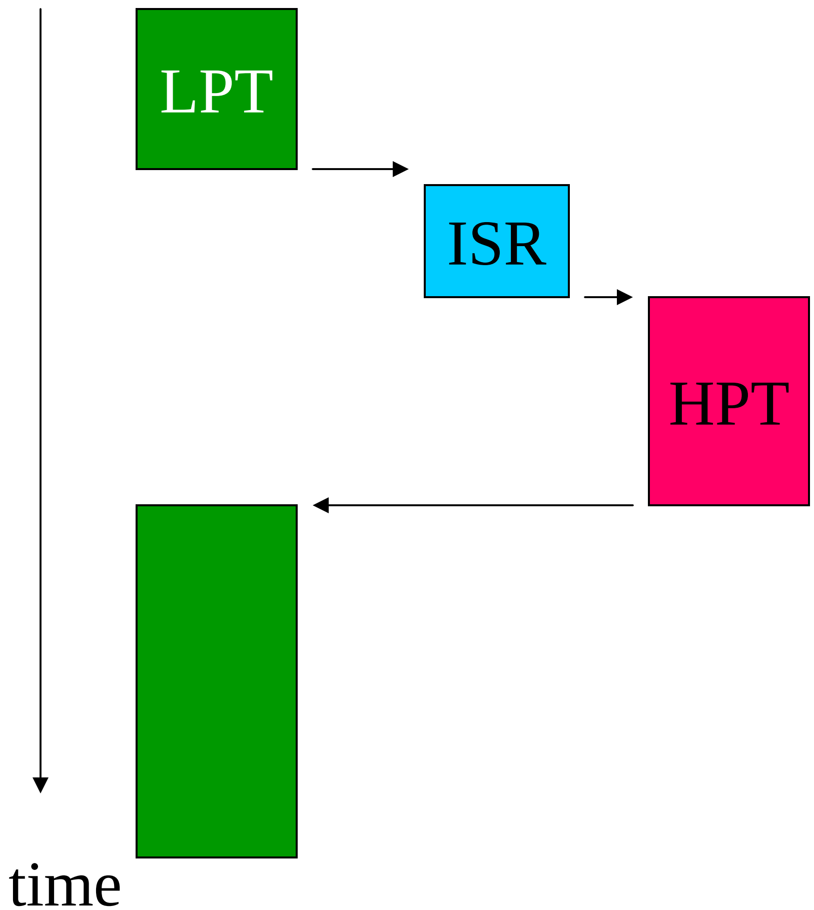


Non-Preemptive Multitasking



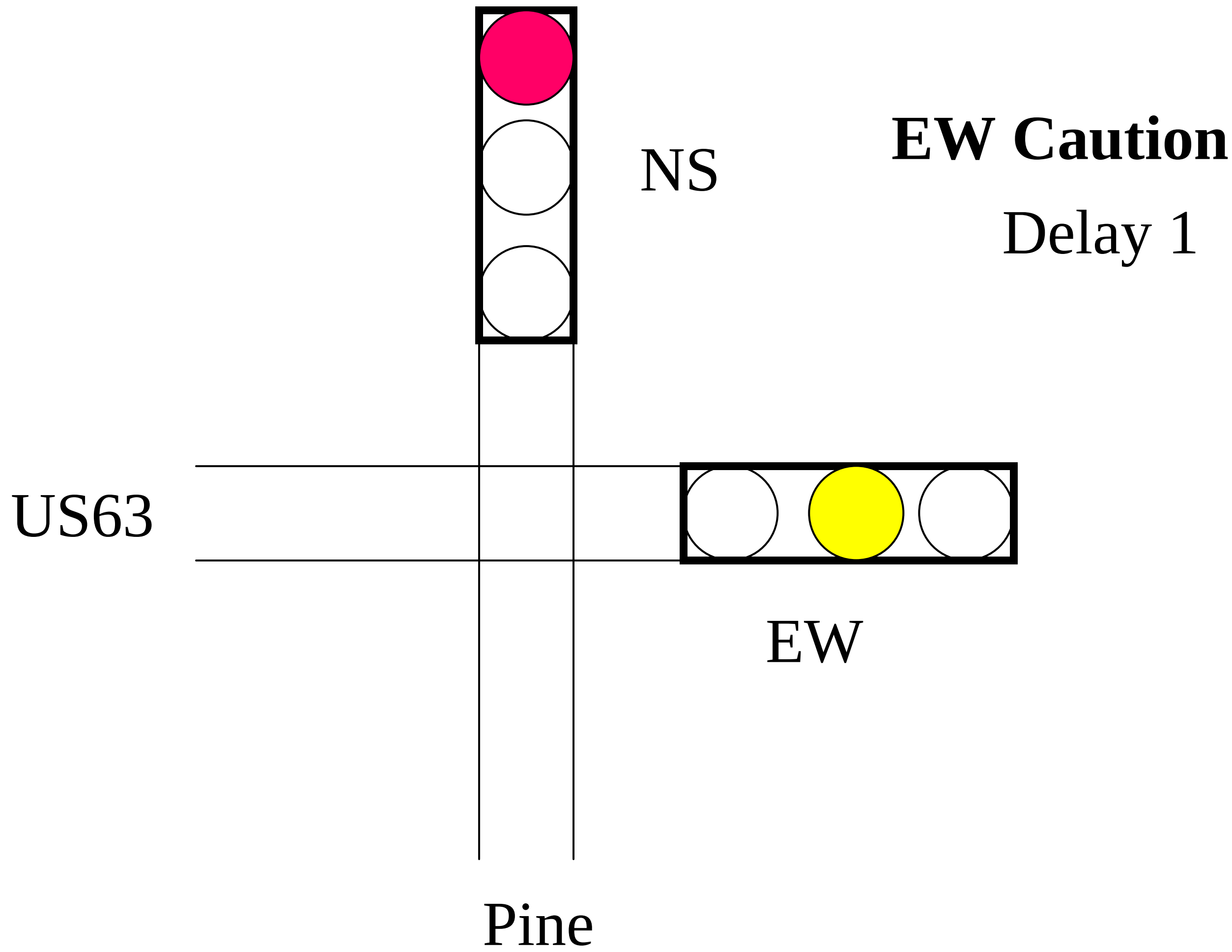
- Low priority task executes
- ISR makes high priority task ready
- LPT finishes
- Relinquishes control to HPT
- HPT executes...

Preemptive Multitasking

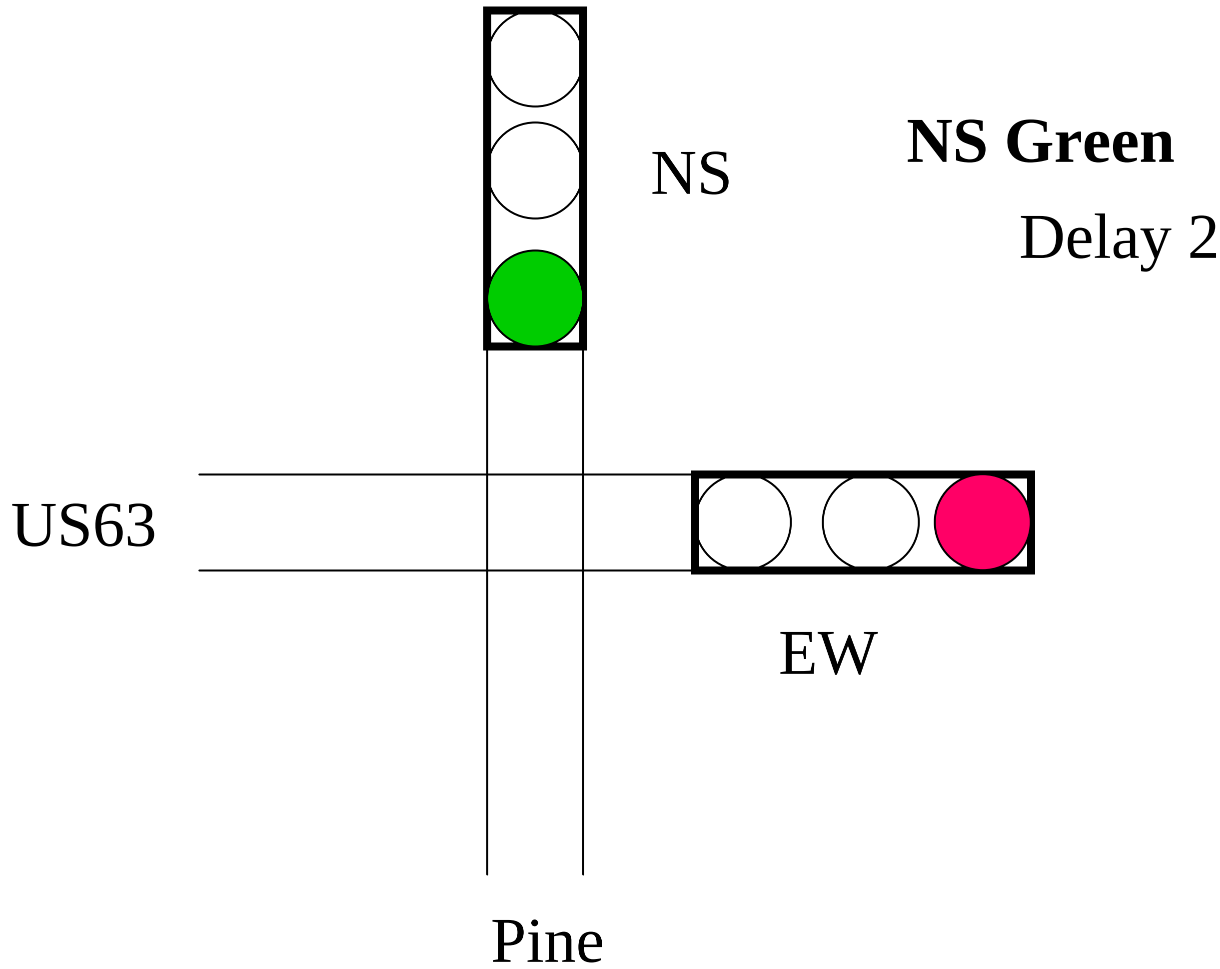


- Low priority task executes
- ISR makes high priority task ready
- Scheduler enables HPT
- HPT finishes
- LPT resumes execution...

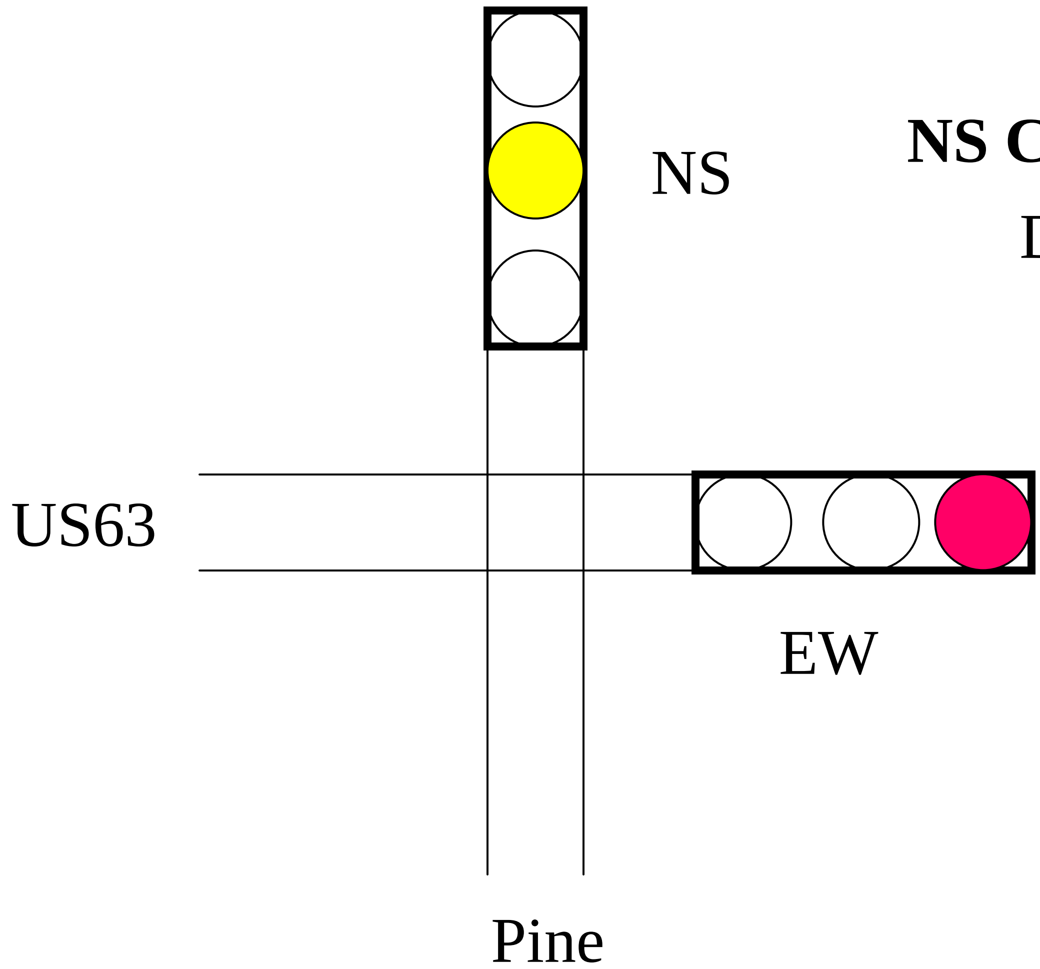
Simple Light Sequencer TLC1



Simple Light Sequencer TLC1



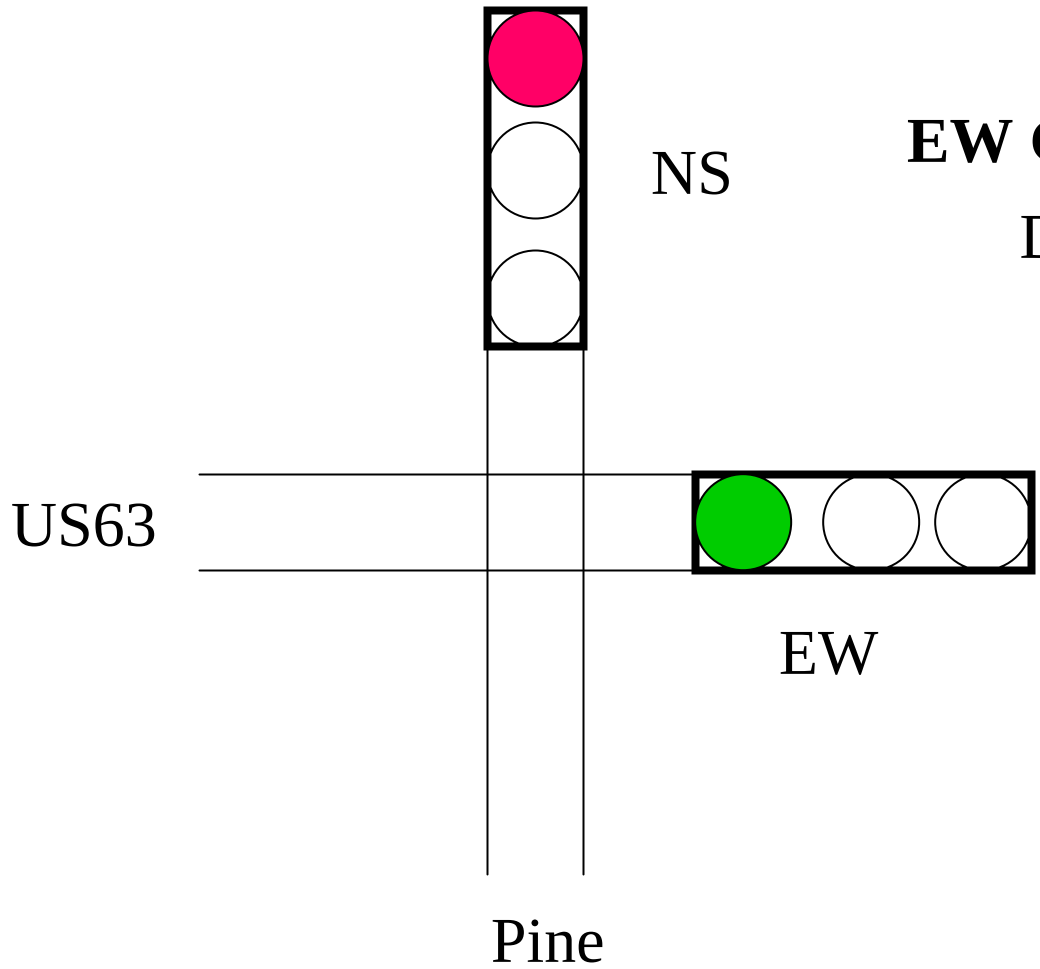
Simple Light Sequencer TLC1



NS Caution

Delay 1

Simple Light Sequencer TLC1



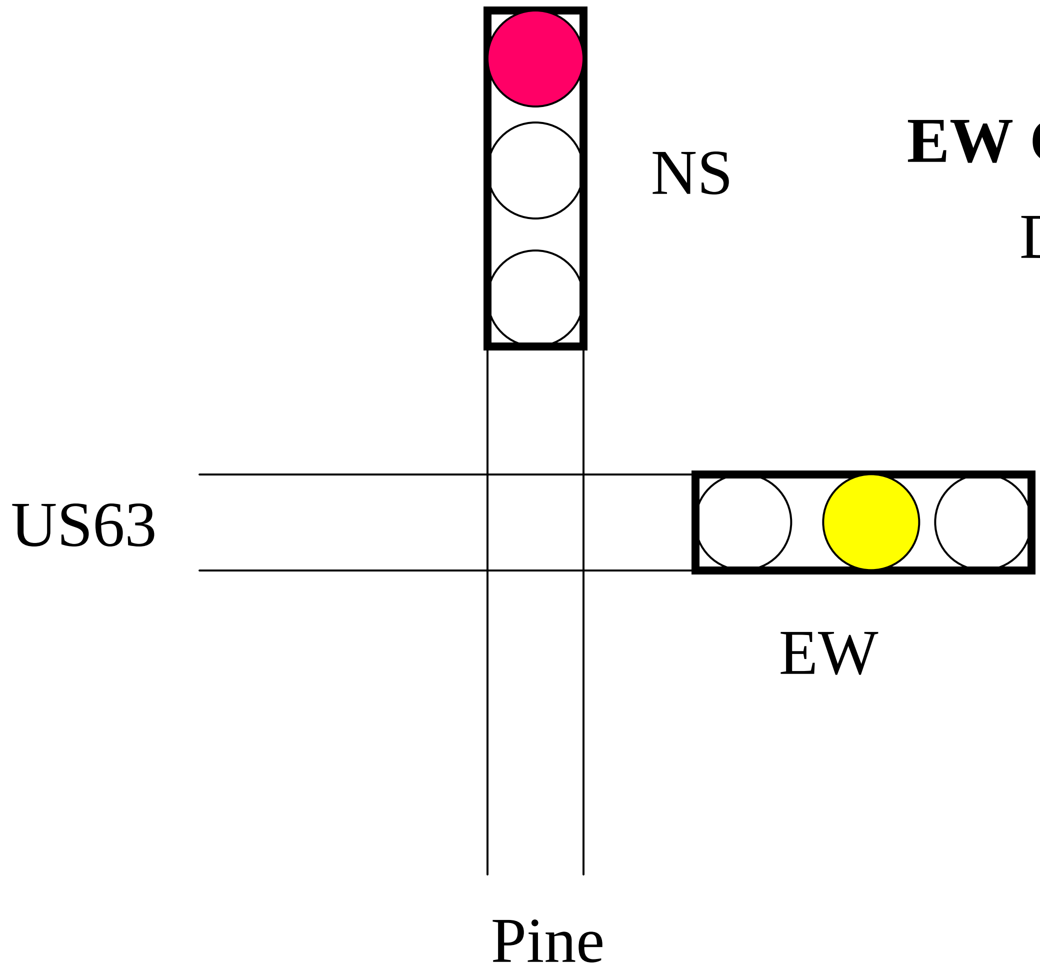
EW Green

Delay 3

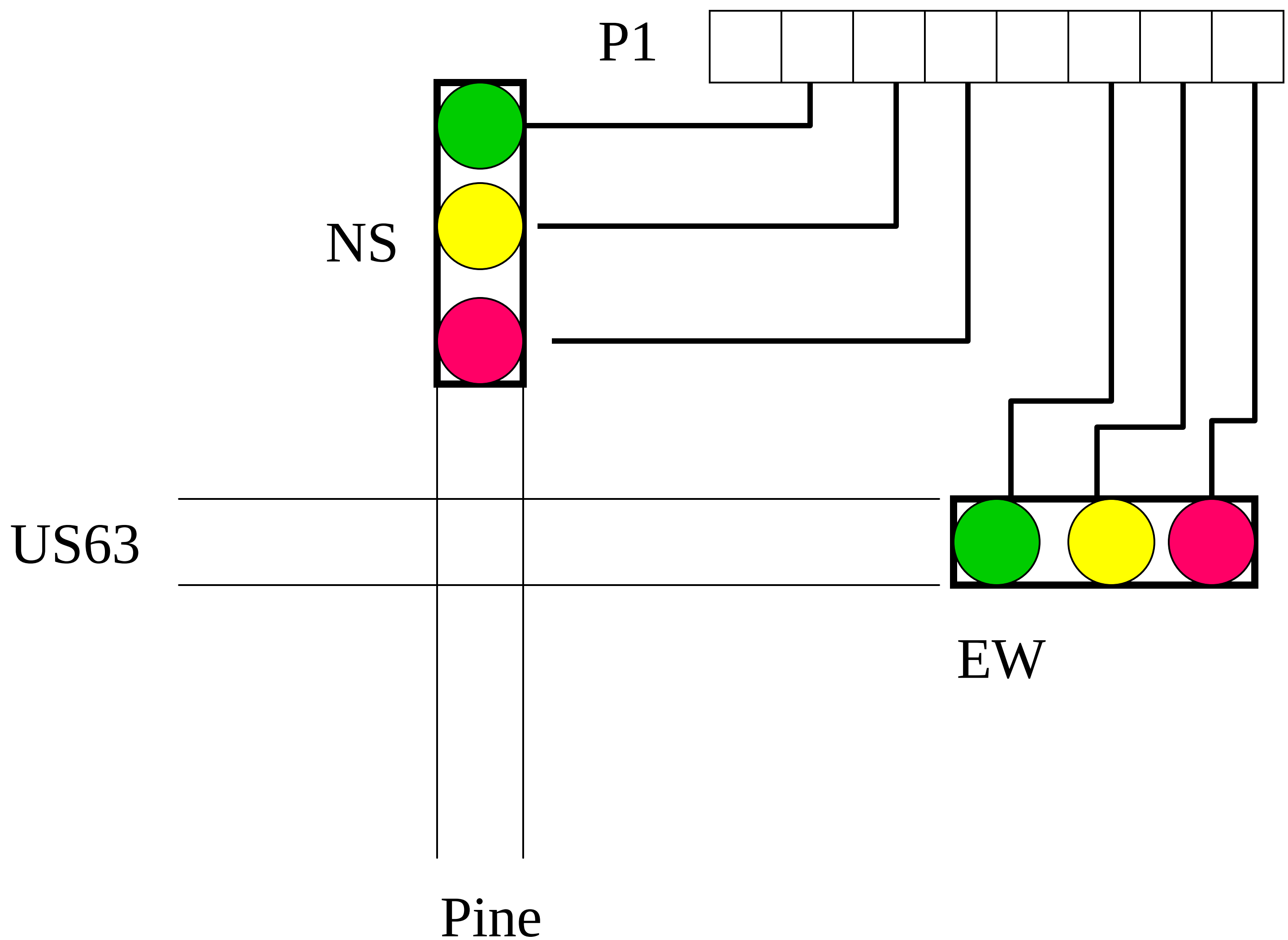
Simple Light Sequencer TLC1

EW Caution, etc

Delay 1



TLC1 Outputs



TLC1-the old way

- Output EW Caution. P1=0001 0010 (-ryg -ryg)
- Delay 1 (short)
- Output NS Green. P1=0100 0001
- Delay 2 (medium)
- Output NS Caution. P1=0010 0001
- Delay 1 (short)
- Output EW Green. P1=0001 0100
- Delay 3 (long)

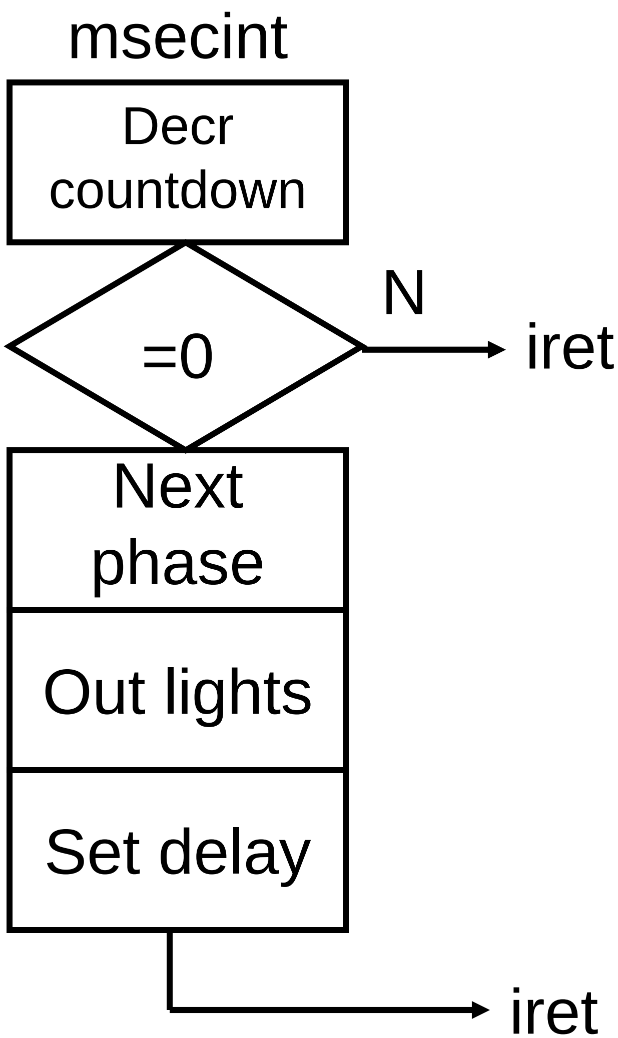
Problem

- Delays implemented by for loop
- Software delays waste CPU time
- Difficult to do anything else while sequencing lights.

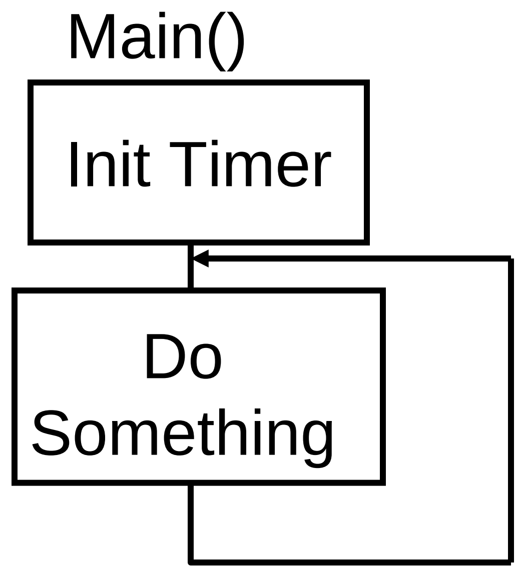
The Solution

- Use two ***tasks***
- One to sequence lights, another to “do something”
- Use timer interrupt to ***schedule*** the sequence after a delay
- New program is ***event driven*** (timer interrupt is the event)

Foreground Task



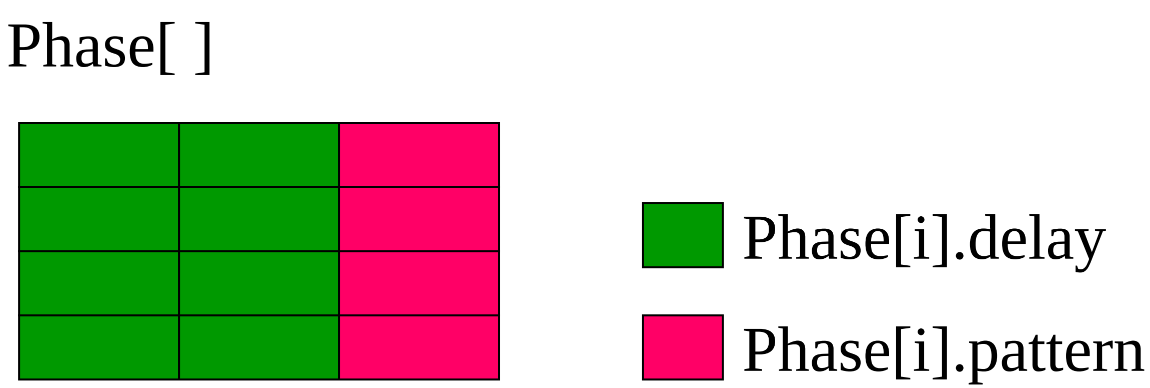
Background Task



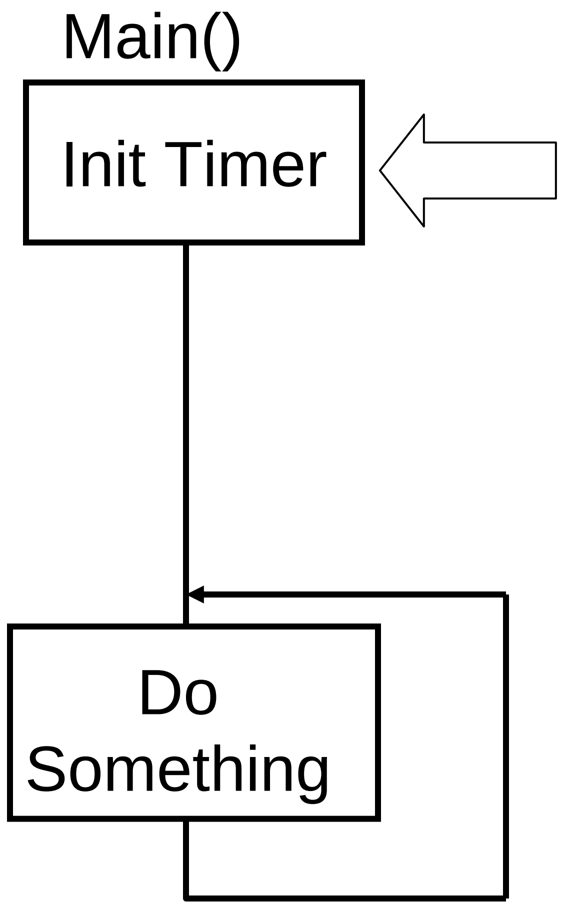
No wait loops in the foreground or the background!

Next Phase struct

```
Struct {uint delay; uchar pattern;}
cycle;

cycle phase[]= { {1000,0x12},
                 {3000,0x41},
                 {1000,0x21},
                 {5000,0x14}};
```

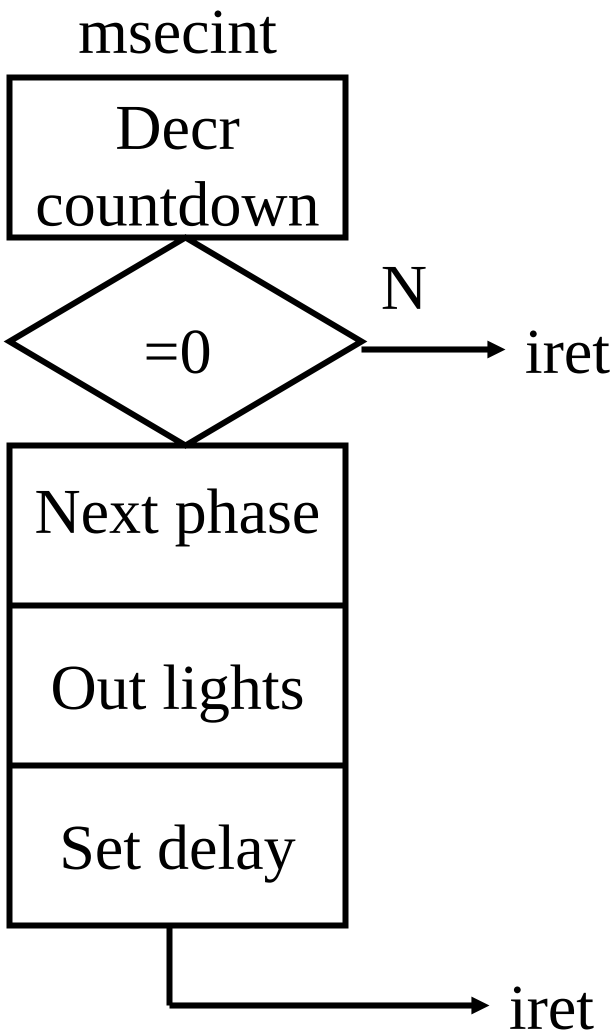


Background Task




```
TMOD=0x10;  //T1=ctr16
TH1= -1000>>8;
TL1= -1000&0xff;
TR1=1; //start T1
IE=0x11; //enable interrupts
```

Foreground Task




```
Cntdwn-=1;
if(!cntdwn){ i=(i+1)&3;
 P1=phase[i].pattern;
 cntdwn= phase[i].delay;
}
```

For the next lecture

- Review lecture notes and Chapter 11.