University of British Columbia
Electrical and Computer Engineering
ELEC291/292

Timers, Interrupts, and Pushbuttons

Dr. Jesús Calviño-Fraga P.Eng.
Department of Electrical and Computer Engineering, UBC
Office: KAIS 3024
E-mail: jesusc@ece.ubc.ca
Phone: (604)-827-5387

January 18, 2016

Objectives

- Understand the advantages of using timers/counters.
- Understand the different operation modes of the 8051's timers/counters.
- Configure and use the timers/counters in the 8051.
- Understand how interrupts operate.
- Configure interrupt service vectors in the 8051 microcontroller.
- Enable disable interrupts.
- Save/restore interrupt service routine registers.
- Attach (and use) pushbuttons to the microcontroller.

Timing & machine cycles

- For example a 40µs delay wit a 24MHz clock and 12 clock periods per cycle:

```
mydelay:
   mov R0, #37
L1:
   djnz R0, L1
   ret
```

lcall: two machine cycles.

The ***djnz*** instruction takes two cycles or 24 clocks: 24/24MHz=1µs, for a total of 37 µs.

Two machine cycles each.

- For many applications this is ok, but it has disadvantages:
  1. It keeps the MCU busy just wasting time.
  2. Timing is tricky to achieve, especially if interrupts are used.

Timing & machine cycles

- For the original 8051 one machine cycle takes 12 oscillator periods. For newer parts the machine cycle could be 6, 4, 2, and frequently just 1 oscillator period.
- For the AT89LP52, one machine cycle takes 1 oscillator period. This year the clock is set to 22.1184 MHz: One cycle takes 45.21 ns.
- The ***nop*** (no operation) instruction can be used to adjust timing loops. It takes one machine cycle.
- Another solution is to use dedicated hardware for timing and counting: Timers and Counters!

Timers/Counters

- Timers/Counters have some advantages over timing loops:
  - The processor is not tied while counting.
  - Combined with interrupts, produces very efficient (small and fast) code.
  - They are usually independent on how many clocks per cycle the MCU takes.
  - Many timers/counters can be set to work concurrently.

8051's Timers/Counters

- The original 8051 has only two timers/counters: 0 and 1.
- Newer 8051 microcontrollers usually have:
  1. The 8051 timers/counters: timers 0 and 1
  2. The 8052 timer/counter: timer 2
  3. The Programmable Counter Array (PCA). Not available in the AT89LP52.
  4. Additional timer/counters: time 3, 4, 5, 6, etc. Not available in the AT89LP52.
- Let us begin with timers 0 and 1:

Timer 0 and Timer 1 Operation Modes

- Timer 0 and 1 have four modes of operation:
  - Mode 0: 13-bit timer/counter (compatible with the 8048 microcontroller, the predecessor of the 8051).
  - Mode 1: 16-bit timer/counter.
  - Mode 2: 8-bit auto reload timer counter.
  - Mode 3: Special mode 8-bit timer/counter (timer 0 only).
- Timer 1 can be used as baud rate generator for the serial port. Some 8051/8052 microcontrollers have a dedicated baud rate generator. We will use timer 1 as a baud rate generator for next lab.

TMOD timer/counter mode control register (Address 89H)

Is this SFR bit addressable?

Timer 1				Timer 0			
GATE	**C/T***	**M1**	**M0**	**GATE**	**C/T***	**M1**	**M0**

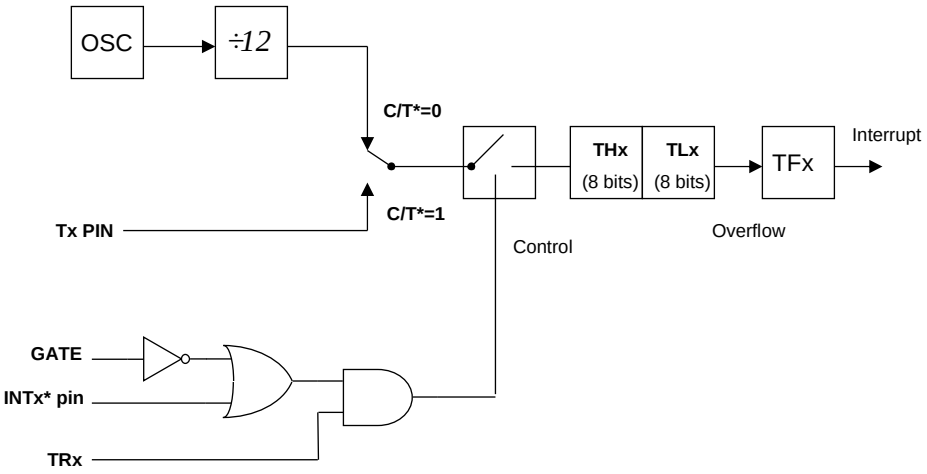
Bit	Name		Description
7 & 3	GATE		1: uses either INT0 or INT1 pins to enable/disable the timer/counter
6 & 2	C/T*		0: timer; 1: counter (pins T0 and T1)
All the other pins!	M1	M0	
	0	0	13-bit timer/counter
	0	1	16-bit timer/counter
	1	0	8-bit auto-reload timer/counter
	1	1	Special mode

TCON: timer/counter control register. (Address 88H)

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

Bit	Name	Description
7	TF1	Timer 1 overflow flag.
6	TR1	Timer 1 run control.
5	TF0	Timer 0 overflow flag.
4	TR0	Timer 0 run control.
3	IE1	Interrupt 1 flag.
2	IT1	Interrupt 1 type control bit.
1	IE0	Interrupt 0 flag.
0	IT0	Interrupt 0 type control bit.

Timer/Counter 0 or 1 in Mode 1 Original 8051



CLKREG

Table 6-2. CLKREG – Clock Control Register

CLKREG = 8FH — Reset Value = 0?0? 00?0B

Not Bit Addressable

	TPS3	TPS2	TPS1	TPS0	CDV2	CDV1	CDV0	—
Bit	7	6	5	4	3	2	1	0

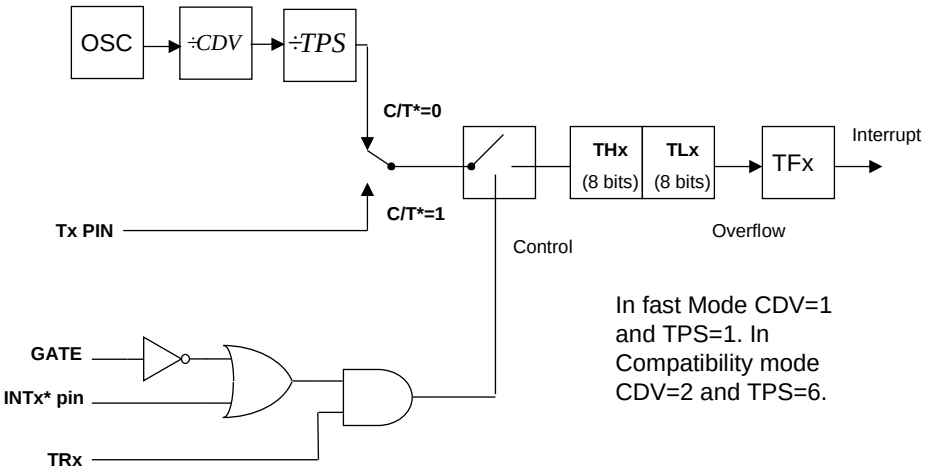
Symbol	Function
TPS[3-0]	Timer Prescaler. The Timer Prescaler selects the time base for Timer 0, Timer 1, Timer 2 and the Watchdog Timer. The prescaler is implemented as a 4-bit binary down counter. When the counter reaches zero it is reloaded with the value stored in the TPS bits to give a division ratio between 1 and 16. By default the timers will count every clock cycle in Fast mode (TPS = 0000B) and every six cycles in Compatibility mode (TPS = 0101B).
CDV[2-0]	System Clock Division. Determines the frequency of the system clock relative to the oscillator clock source.

CDIV2	CDIV1	CDIV0	System Clock Frequency
0	0	0	$f_{OSC}/1$
0	0	1	$f_{OSC}/2$
0	1	0	$f_{OSC}/4$
0	1	1	$f_{OSC}/8$
1	0	0	$f_{OSC}/16$
1	0	1	$f_{OSC}/32$
1	1	0	Reserved
1	1	1	Reserved

Note: The reset value of CLKREG is 0000000B in Fast mode and 01010010B in Compatibility mode.

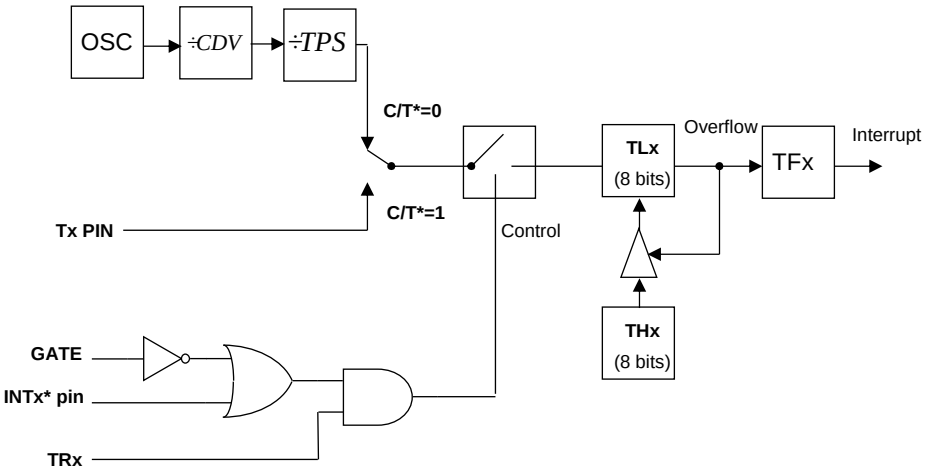
1

Timer/Counter 0 or 1 in Mode 1 AT89LP52



In fast Mode CDV=1 and TPS=1. In Compatibility mode CDV=2 and TPS=6.

Timer/Counter 0 or 1 in Mode 2 AT89LP52



Timer/Counter 0 in Mode 2

```
myprogram:
    ; After reset, the stack pointer register is set to 07h
    ; We may need space for variables, so move the SP
    mov SP, #7fH
    ; Enable timer 0
    mov a, TMOD
    anl a, #0f0H
    orl a, #00000010B ; GATE=0, C/T*=0, M1=1, M0=0: 8-bit auto reload timer
    mov TMOD, a
    mov TH0, #080H ; Set the interrupt rate
    setb TR0 ; Enable timer 0
    setb ET0 ; Enable timer 0 interrupt (future lecture!)
    setb EA
Blink:
    cpl P1.0
    mov R0, #200
L0: djnz R0, L1
    jmp Blink
L1: mov R1, #200
L2: djnz R1, L2
    jmp L0
```

Timer/Counter 2

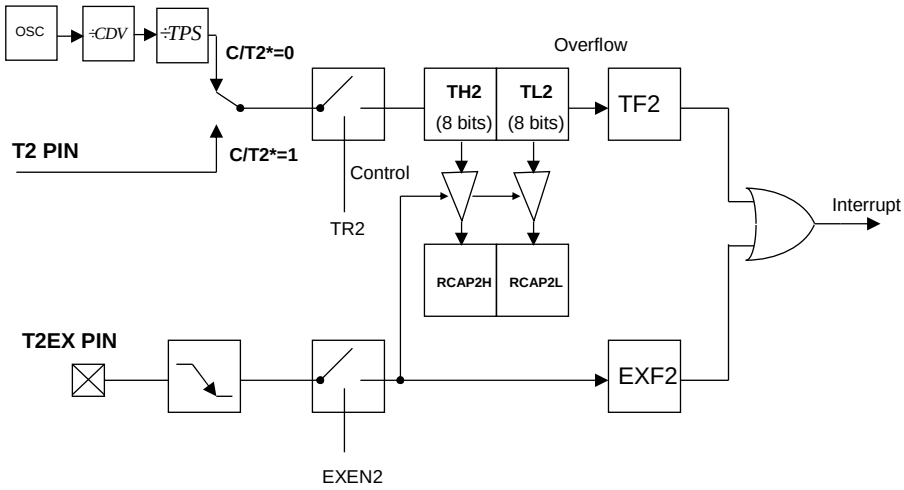
- It is a 16-bit timer/counter.
- It has four modes of operation:
  - Capture
  - Auto-reload
  - Baud rate generation
  - Programmable clock out

T2CON: timer/counter 2 control register. (Address C8H)

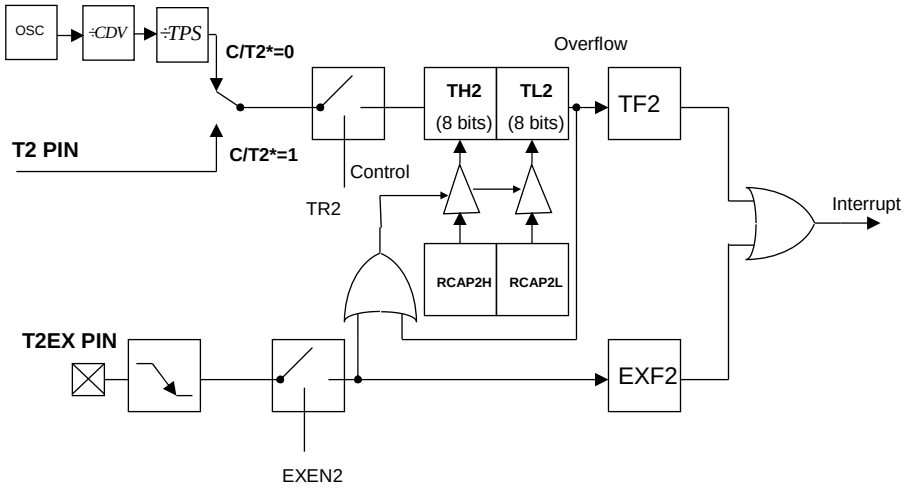
TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/T2*	CP/RL2*

Bit	Name	Description
7	TF2	Timer/counter 2 overflow flag.
6	EXF2	Timer/counter 2 external flag.
5	RCLK	Receive clock flag.
4	TCLK	Transmit clock flag.
3	EXEN2	Timer/Counter 2 external enable.
2	TR2	Start/stop for timer/counter 2.
1	C/T2*	Timer or Counter select.
0	CP/RL2*	Capture/Reload Flag.

Timer/Counter 2 in capture mode AT89LP52



Timer/Counter 2 in auto-reload mode AT89LP52



Example: Time Delay Using a Timer

- To use a timer to implement a delay we need to:
  - Initialize the timer: use TMOD SFR.
  - Load the timer: use THx and TLx.
  - Clear the timer overflow flag: TFx=0;
  - Start the timer: Use TRx.
  - Check the timer overflow flag: Use TFx.

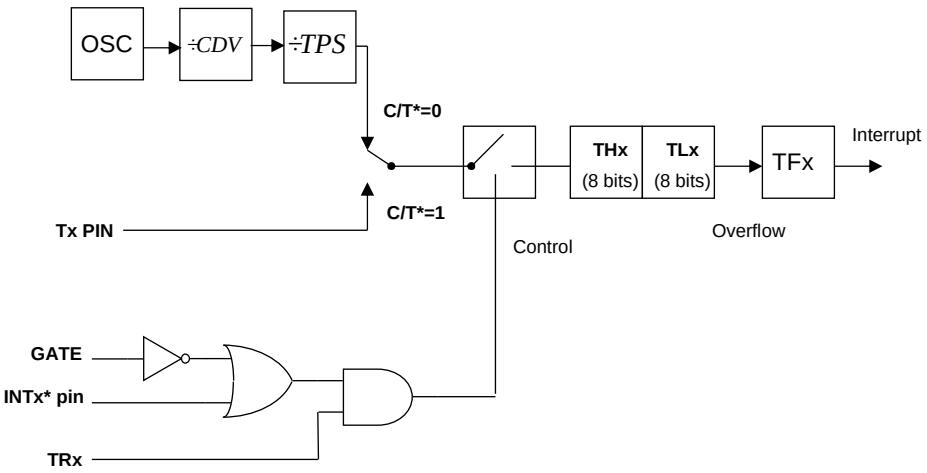
For the registers above 'x' is either '0' for timer 0, or '1' for timer 1.

Time Delay Using a Timer

- Implement a 1 ms delay subroutine using timer 0. Assume the routine will be running in a AT89LP52 microcontroller wit a 22.1184MHz in fast mode.

First, we have to find the divider (TH0, TL0) needed for a 1 ms delay…

Timer/Counter 0 or 1 in Mode 1



Calculating TH0 and TL0

$$\text{Rate}=\frac{\text{CLK}}{2^{16}-[\text{THn,TLn}]}=\frac{22.1184\text{MHz}}{65536-[\text{THn,TLn}]}$$

$$[\text{THn,TLn}]=65536-\frac{22.1184\text{MHz}}{\text{Rate}}=65536-\frac{22.1184\text{MHz}}{(1/1\text{ms})}=43417$$

Maximum delay achievable?

$$\text{Rate}=\frac{22.1184\text{MHz}}{2^{16}-[\text{THn,TLn}]}=\frac{22.1184\text{MHz}}{65536-[\text{THn,TLn}]}$$

$$[\text{THn,TLn}]=0$$

$$\text{Rate}=\frac{22.118400\text{MHz}}{65536}=337.5Hz\rightarrow 2.963ms$$

This is for CDV=1 and TPS=1 (Fast Mode). In Compatibility mode, for example, CDV=2 and TPS=6 and we use 22.1184MHz/12 instead.

Time Delay Using Timer 0

```
Wait1ms:
    ; Initialize the timer
    mov a, TMOD
    anl a, #11110000B ; Clear bits for timer 0
    orl a, #00000001B ; GATE=0, C/T*=0, M1=0, M0=1: 16-bit timer
    mov TMOD, a
    clr TR0 ; Disable timer 0
    ; Load the timer [TH0, TL0]=65536-(22118400/(1/0.001))
    mov TH0, #high(43417)
    mov TL0, #low(43417)
    clr TF0 ;Clear the timer flag
    setb TR0 ; Enable timer 0
Wait1ms_L0:
    jnb TF0, Wait1ms_L0 ; Wait for overflow
    ret
```

Not bad, but we can do better:

Time Delay Using Timer 0

```
; Let the Assembler do the calculation for us!
XTAL equ 22118400
FREQ equ 1000 ; 1/1000Hz=1ms
RELOAD_TIMER0_1ms equ 65536-(XTAL/FREQ)

Wait1ms:
    ; Initialize the timer
    mov a, TMOD
    anl a, #11110000B ; Clear bits for timer 0
    orl a, #00000001B ; GATE=0, C/T*=0, M1=0, M0=1: 16-bit timer
    mov TMOD, a
    clr TR0 ; Disable timer 0
    mov TH0, #high(RELOAD_TIMER0_1ms )
    mov TL0, #low(RELOAD_TIMER0_1ms )
    clr TF0 ;Clear the timer flag
    setb TR0 ; Enable timer 0
Wait1ms_L0:
    jnb TF0, Wait1ms_L0 ; Wait for overflow
    ret
```

Timers Summary

- The original 8051 had only two timer/counters. They can be configured in four different modes.
- Newer 8051s can have many timers/counters (3 or more), and one or more multi channel PCA.
- Usually, any of the timers can be used to generate periodic interrupts (interrupts will be covered next).
- Timers 1 and 2 can be used as baud rate generators for serial communications (next lecture).

Interrupts

- Interrupt uses:
  - Handshake I/O thus preventing CPU from being tied up.
  - Providing a way to handle some errors: illegal opcodes, dividing by 0, power failure, etc.
  - Getting the CPU to perform periodic tasks: generate square waves, keep time of day, measure frequency, etc.

Interrupts

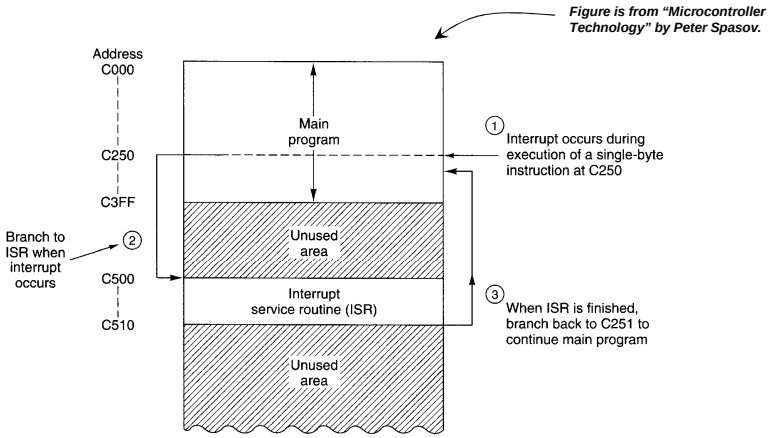


FIGURE 8.11 Example of how an interrupt causes a change in the execution of a program.

Interrupts

- Most processors provide a way of enabling / disabling all maskable interrupts. For the 8051:

  clr EA ;Disable interrupts

  setb EA ;Enable interrupts

- Some other interrupts are non-maskable and they MUST be serviced. For example, the X86 has the "Non-Maskable Interrupt" NMI.
- Maskable interrupts can be enabled/disabled individually. For the 8051 use register IE:

IE: INTERRUPT ENABLE REGISTER. (Address A8H)

EA	--	ET2	ES	ET1	EX1	ET0	EX0

Bit	Name	Description
7	EA	Interrupt Enable Bit: EA = 1 interrupt(s) can be serviced, EA = 0 interrupt servicing disabled.
6	--	Reserved
5	ET2	Timer 2 Interrupt Enable. (8052)
4	ES	Serial Port Interrupt Enable
3	ET1	Timer 1 Overflow Interrupt Enable.
2	EX1	External Interrupt 1 Enable.
1	ET0	Timer 0 Overflow Interrupt Enable.
0	EX0	External Interrupt 0 Enable.

() Bit addressable registers

- If the location address of an special function register (SFR) is a multiple of 8, then the register is bit addressable and you can use the ***setb*** and ***clr*** instructions.
- IE is bit addressable!

Two external Interrupts

- Connected to pins P3.2 (INT0) and P3.3 (INT1) in the standard 8051.
- Can be configured to be edge sensitive or level sensitive. Use bits IT0 and IT1 in SFR TCON to specify falling edge or low level sensitivity.
- There is an application note (somewhere) on how to use the timer inputs T0 and T1 as additional external interrupts.

Interrupts and the stack

- Interrupts in the 8051 make use of the stack.
- The stack is an area of memory where variables can be stacked. It is a LIFO memory: the last variable you put in is the first variable that comes out.
- Register SP (stack pointer) points to the beginning of the stack. SP in the 8051 is incremented **before** is used (push), or used and them decremented (pop).
- After reset, SP is set to 07H. If you have variables in internal RAM, any usage of the stack is likely to damage them. Therefore, at the beginning of your program set the SP:

```
mov SP, #7FH ; Set the stack pointer to idata start
```

Interrupts and the stack

- Additionally, these two instructions can be used to push/pull registers to/from the stack: ***push*** **&** ***pop***
- After an interrupt is asserted the CPU:
  - Pushes the address of the next instruction into the stack (two bytes). Some processors also push some or all of the registers into the stack as well (not the 8051 though!).
  - All interrupts of equal or lower priority are disabled.
  - Then the program counter (PC) is set to the Interrupt Service Routine (ISR) vector.
  - The PC will be restored to the interrupted point once the ***reti*** instruction is executed in the ISR and all interrupts of equal or lower priority are re-enabled.

Interrupt Service Routines (ISR) Vectors

- The 8051 will ***lcall*** to an specific memory location when an interrupt occurs. The may be different for different 8051 variants. For the AT89LP52:

Interrupt source	Address
External 0	0003H
Timer 0	000BH
External 1	0013H
Timer 1	001BH
Serial port	0023H
Timer 2	002BH

Interrupt Service Routines (ISR) Vectors

- Notice that there are only 8 bytes available between vectors. Not enough for a decent ISR, but more than enough for a *ljmp* instruction!
- IF you enable a particular interrupt, there **MUST** be an ISR, or your program **WILL** crash. A fool proof code technique is to setup all the ISR vectors and place a ***reti*** (return from interrupt) instruction for those that are not used (next example).
- In assembly language you can use the "*org*" directive to set an ISR vector.
- To return from an ISR use the ***reti*** instruction. To return from a normal routine use the ***ret*** instruction.

Example

```
; Basic interrupt setup

; We need the register definitions for the 8052:
$MODLP52

org 0h
ljmp myprogram

; Notice that there is not much space to put code between
; service routines, but enough to put a ljmp!

; External interrupt 0
org 3h
reti

; Timer 0 interrupt
org 0bh
reti
```

Dummy ISRs

WARNING: org directives must be sequential!

Example (cont.)

```
; External interrupt 1
org 13h
reti

; Timer 1 interrupt
org 1bh
reti

; Serial port interrupt
org 23h
reti

; Timer 2 interrupt
org 2bh
reti

; Dummy program, just to compile and see...
myprogram:
    mov R1, #00H ; do something
    sjmp myprogram
END
```

Dummy ISRs

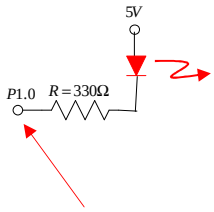
Example: Do something (useful) in the main program.

```
; This program makes an LED connected to P1.0 blink

myprogram:
        cpl P1.0

        mov R0, #200
L0:
        djnz R0, L1
        jmp myprogram
L1:
        mov R1, #200
L2:
        djnz R1, L2
        jmp L0
```

This is the Complement bit instruction



In general, MCU pins are better at sinking current than sourcing current

Example: Enable timer 0 interrupt and setup an ISR

```
myprogram:
        ; Enable timer 0
        mov a, TMOD
        anl a, #0f0H
        orl a, #00000010B ; 8-bit auto reload timer (this is in binary)
        mov TMOD, a
        mov TH0, #080H ; Set the interrupt rate
        setb TR0 ; Enable timer 0
        setb ET0 ; Enable timer 0 interrupt

        setb EA ; Enable all interrupts!

Blink:
        cpl P1.0
        mov R0, #200
L0:
        djnz R0, L1
        jmp Blink
L1:
        mov R1, #200
L2:
        djnz R1, L2
        jmp L0
```


Example (cont.) the ISR.

```
; Timer 0 interrupt
org 0bh
   cpl P1.1 ; Check this pin with the scope!
   reti
```

Example: use a *ljmp* to go to the ISR

```
; Timer 0 interrupt
org 0bh
   ljmp timer0_ISR

; Other ISR vectors come here! (Not shown to save space)

; Actual ISR for timer 0.
timer0_ISR:
   cpl P1.1
   reti
```


Saving and Restoring Registers in the Stack

- If your ISR routine uses a register, you must make sure that it will remain **unmodified** before returning to the interrupted program. Example, if ACC was 33 when the ISR was called, it must be set back to 33 before the ***reti***.
- As mentioned before you use the instructions ***push***/***pop*** to save/restore registers to/from the stack.
- Additionally, you could use one of four available register banks in your ISR.

Example: Saving and Restoring Registers in the Stack

```
; Actual ISR for timer 0. There must be a ljmp at address 0BH
timer0_ISR:
    ; The main loop is using both registers R0 and R1,
    ; so if we want to use them in this ISR we should push then
    ; into the stack and restore them before reti.
    push AR0
    push AR1

    cpl P1.1
    mov R1, #55H ;Wreck R1 and R0 so to show that program works!
    inc R0

    ; Restore the register to their original values
    pop AR1
    pop AR0

    reti ; Return from interrupt
```

The 'A' stands for address…

Saving and Restoring Registers in the Stack

- Before using the stack make sure you set the SP register.
- Popular registers to push/pop in ISRs: ACC, DPL, DPH, PSW, R0 to R7. Of course, only if they are used in the ISR.
- Pop registers from the stack in the **REVERSE** order you pushed them! Remember the stack is a LIFO.

Setting the SP register

```
myprogram:
    ; After reset, the stack pointer register is set to 07h
    ; We may need space for variables, so move the SP
    mov SP, #7FH

    ; Enable timer 0
    mov a, TMOD
    anl a, #0f0H
    orl a, #00000010B ; 8-bit auto reload timer
    mov TMOD, a
    mov TH0, #080H ; Set the interrupt rate (see formula in the book)
    setb TR0 ; Enable timer 0
    setb ET0 ; Enable timer 0 interrupt
    setb EA

Blink:
    cpl P1.0
    mov R0, #200
L0:
    djnz R0, L1
    jmp Blink
L1:
    mov R1, #200
L2:
    djnz R1, L2
    jmp L0
```


Register Banks

- Bits RS1 (bit 4) and RS0 (bit 3) of the Program Status Word (PSW) select one of four available register banks.
- After a power-on reset, register bank 0 is selected.
- Normally you will use a register bank for each different interrupt priority level.

Using different register banks in ISRs

```
; Actual ISR for timer 0.  Notice that there must be a ljmp at address 0BH
timer0_ISR:
    ; Another possibility is to use a different register bank.  First,
    ; Save the two registers we are going to use:
    push ACC
    push PSW
    mov PSW, #00001000B ; Select register bank 1!

    cpl P1.1
    mov R1, #55H ;Wreck R1 and R0 so to demonstrate that program works!
    inc R0
    clr a ; Change also the accumulator...

    ; Restore the registers to their original values
    pop PSW
    pop ACC
    reti ; Return from interrupt
```

Alternatively we can use:

```
clr RS1
setb RS0
```

Question: What register bank will be selected after *reti*? Why?

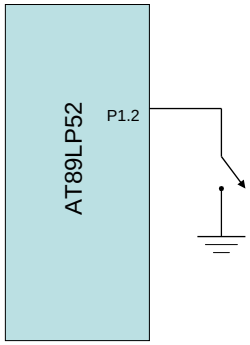
Interrupt Programming with the 8051 in Assembly (summary)

- Set a ***ljmp*** to the ISR into the corresponding memory address for each interrupt source.
- Setup the stack in the main program. (Do this only once!)
- Setup (including priority) and Enable the interrupt to use.
- In the ISR use ***push***/***pop*** to save restore used registers. You may also use a different register bank.
- Use a ***reti*** instruction to return from the ISR.

Reading Push Buttons

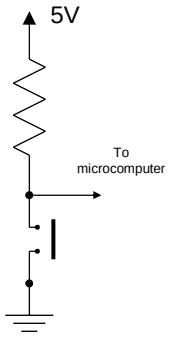
- Before using a pin for input we need to configure it:
  - Original 8051: Write ‘1’ to the pin to be used as input.
  - Newer 8051s: configure the pin as input using designated SFRs .
- In the original 8051 any pin can be used as output or input. In newer 8051s some pins can be only input and/or outputs.
- In the original 8051 pins in the same port can be independently used as inputs or outputs. For example pin P0.0 can be used as input, while P0.1 can be used as output!

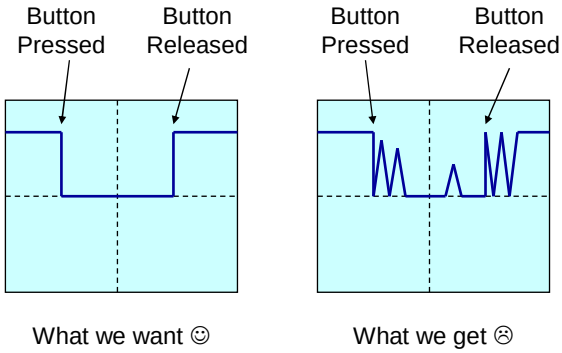
Reading Push Buttons




```
Setb P1.2; Make pin input
.
.
.
.
jnb P1.2, ButtonPressed
jb P1.2, ButtonNotPressed
```

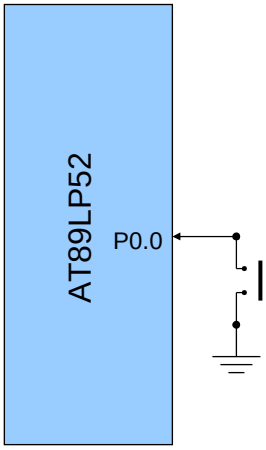
Problem: Contact Bounce





The time the contact bounces
can be as long as 50ms!

Software Debouncing




```
setb P0.0 ; Before using as input...
jb P0.0, not_pressed
lcall Wait50ms ; Wait and check again
jb P0.0, not_pressed
; Wait for the button to be released
L0: jnb P0.0, L0
sjmp pressed
```

This technique is called *wait-and-see*
in many textbooks