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 22, 2021

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Objectives

- Configure and use the timers in the EFM8LB1 microcontroller.
- Configure and use interrupts.
- Attach (and use) pushbuttons.
- Attach and use speaker with the EFM8LB1 microcontroller.
- Macros (time permitting).

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Timing & machine cycles

- For the EFM8, one machine cycle takes 1 oscillator period. In the examples provided for lab 2 the clock is set to 24 MHz: One cycle takes 41.67 ns.
- If we use delay loops for timing, the processor is busy wasting valuable computing time!
- A better solution is to use dedicated hardware for timing and counting: Timers and Counters!
- The timers and counters of other processors maybe/are different. The idea is the same!

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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). Available in the EFM8! Very powerful, other architectures have exactly the same PCA! (The 68HC11 from Freescale for example)
  4. Additional timer/counters: time 3, 4, 5, etc. Timers 3 to 5 are available in the EFM8LB1!.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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). DO NOT USE THIS MODE!
  - 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. The EFM8LB1 does no have a dedicated baud rate generator for UART0! (UART1 has a dedicated BR generator)

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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

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

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

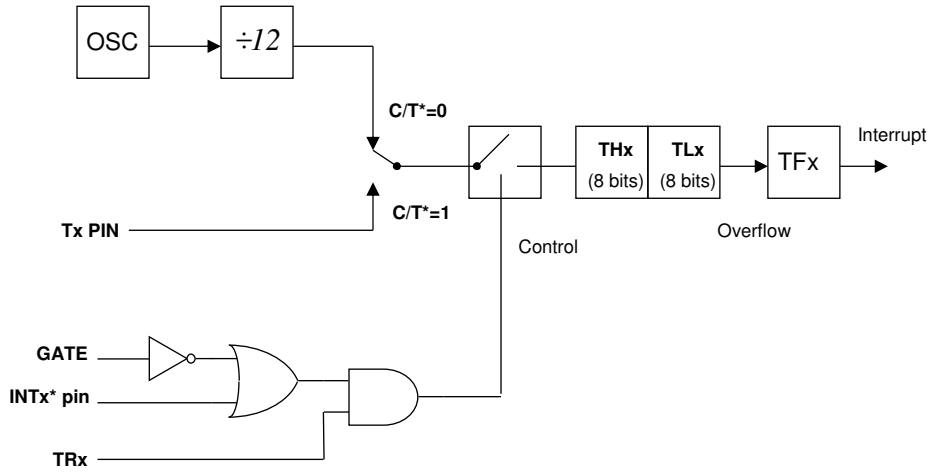
TCON: timer/counter control register. (Address 88H)

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.

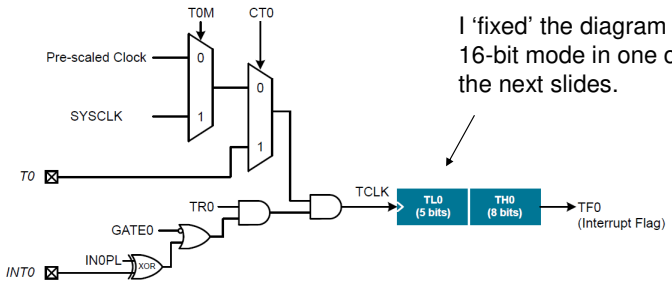
Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Timer/Counter 0 or 1 in Mode 1 Original 8051



Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

EFM8 Timer/Counter 0/1 in Mode 1



I 'fixed' the diagram for 16-bit mode in one of the next slides.

Figure 21.1. T0 Mode 0 Block Diagram

Mode 1: 16-bit Counter/Timer

Mode 1 operation is the same as Mode 0, except that the counter/timer registers use all 16 bits. The counter/timers are enabled and configured in Mode 1 in the same manner as for Mode 0. The overflow rate for Timer 0 in 16-bit mode is:

$$F_{TIMER0} = \frac{F_{Input\ Clock}}{2^{16} - TH0{:}TL0} = \frac{F_{Input\ Clock}}{65536 - TH0{:}TL0}$$

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

T0M

21.4.1 CKCON0: Clock Control 0

Bit	7	6	5	4	3	2	1	0
Name	T3MH	T3ML	T2MH	T2ML	T1M	T0M	SCA	
Access	RW	RW	RW	RW	RW	RW	RW	
Reset	0	0	0	0	0	0	0x0	
SFR Page = 0x0, 0x10, 0x20; SFR Address: 0x8E								

Bit	Name	Reset	Access	Description
2	T0M	0	RW	**Timer 0 Clock Select.** Selects the clock source supplied to Timer 0. Ignored when C/T0 is set to 1. Value / Name / Description 0 / PRESCALE / Counter/Timer 0 uses the clock defined by the prescale field, SCA. 1 / SYSCLK / Counter/Timer 0 uses the system clock.
1:0	SCA	0x0	RW	**Timer 0/1 Prescale.** These bits control the Timer 0/1 Clock Prescaler: Value / Name / Description 0x0 / SYSCLK_DIV_12 / System clock divided by 12. 0x1 / SYSCLK_DIV_4 / System clock divided by 4. 0x2 / SYSCLK_DIV_48 / System clock divided by 48. 0x3 / EXTOSC_DIV_8 / External oscillator divided by 8 (synchronized with the system clock).

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Timer/Counter 2,3,4,5

- They are 16-bit timer/counter.
- They have at least two modes of operation:
  - Capture
  - Auto-reload

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Timer/Counter 2 in auto-reload mode for EFM8

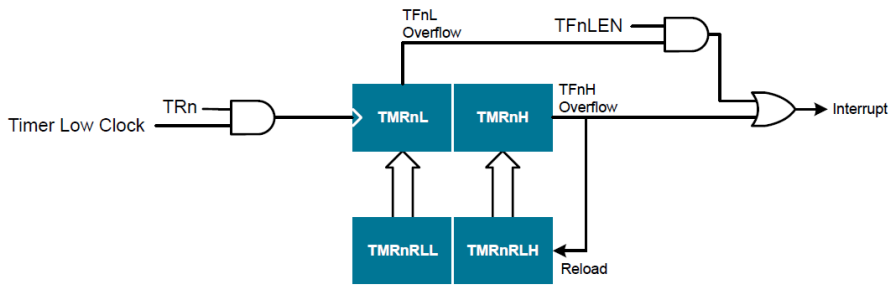


Figure 21.6. 16-Bit Mode Block Diagram

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

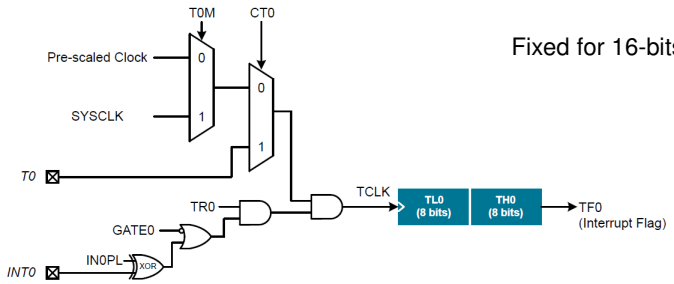
Time Delay Using a Timer

- Implement a 1 ms delay subroutine using timer 0. Assume the routine will be running in a EFM8LB1 microcontroller with a 24MHz clock and using the system clock (SYSCLK) without pre-scaling.

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

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

EFM8 Timer/Counter 0/1 in Mode 1



Fixed for 16-bits!

Figure 21.1. T0 Mode 1 Block Diagram

Mode 1: 16-bit Counter/Timer

Mode 1 operation is the same as Mode 0, except that the counter/timer registers use all 16 bits. The counter/timers are enabled and configured in Mode 1 in the same manner as for Mode 0. The overflow rate for Timer 0 in 16-bit mode is:

$$F_{TIMER0} = \frac{F_{Input\ Clock}}{2^{16} - TH0{:}TL0} = \frac{F_{Input\ Clock}}{65536 - TH0{:}TL0}$$

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Calculating TH0 and TL0

$$\text{Rate} = \frac{\text{CLK}}{2^{16} - [\text{THn,TLn}]} = \frac{24\text{MHz}}{65536 - [\text{THn,TLn}]}$$

$$[\text{THn,TLn}] = 65536 - \frac{24\text{MHz}}{\text{Rate}} = 65536 - \frac{24\text{MHz}}{(1/1\text{ms})} = 41536$$

Maximum delay achievable?

$$\text{Rate} = \frac{24\text{MHz}}{2^{16} - [\text{THn,TLn}]} = \frac{24\text{MHz}}{65536 - [\text{THn,TLn}]}$$

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

$$\text{Rate} = \frac{24\text{MHz}}{65536} = 366.21Hz \rightarrow 2.73ms$$

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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(41536)
    mov TL0, #low(41536)
    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:

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Time Delay Using Timer 0

```
; Let the Assembler do the calculation for us!
XTAL equ 24000000
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
```


Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Interrupts

Figure is from "Microcontroller Technology" by Peter Spasov.

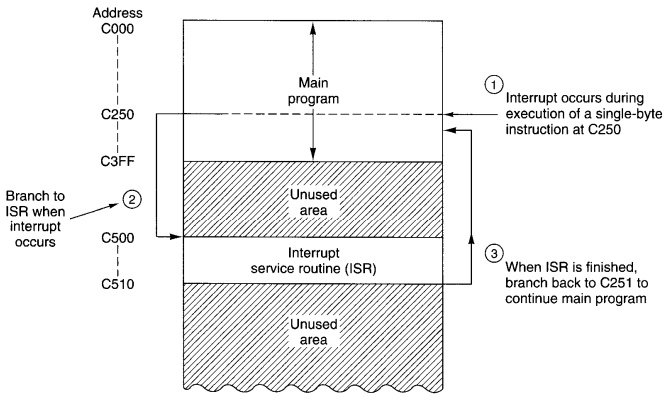


FIGURE 8.11 Example of how an interrupt causes a change in the execution of a program.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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 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:

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

IE: INTERRUPT ENABLE REGISTER. (Address A8H) (Original 8051)

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.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

() 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! Then you can access the bits like “setb EA”.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Interrupts in the EFM8

6.3.1 IE: Interrupt Enable

Bit	7	6	5	4	3	2	1	0
Name	EA	ESPI0	ET2	ES0	ET1	EX1	ET0	EX0
Access	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0
SFR Page = ALL; SFR Address: 0xA8 (bit-addressable)								

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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
```


Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Interrupt Service Routines (ISR) Vectors

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

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

- For the EFM8LB1, see next slide:

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Interrupts in the EFM8LB1

Table 6.2. Interrupt Priority Table

Interrupt Source	Vector	Priority	Primary Enable	Auxiliary Enable(s)	Pending Flag(s)
Reset	0x0000	Top	-	-	-
External Interrupt 0	0x0003	0	IE_EX0	-	TCON_IE0
Timer 0 Overflow	0x000B	1	IE_ET0	-	TCON_TF0
External Interrupt 1	0x0013	2	IE_EX1	-	TCON_IE1
Timer 1 Overflow	0x001B	3	IE_ET1	-	TCON_TF1
UART0	0x0023	4	IE_ES0	-	SCON0_RI SCON0_TI
Timer 2 Overflow / Capture	0x002B	5	IE_ET2	TMR2CN0_TF2CEN TMR2CN0_TF2LEN	TMR2CN0_TF2H TMR2CN0_TF2L
SPI0	0x0033	6	IE_ESPI0	SPI0FCN0_RFRQE SPI0FCN0_TFRQE SPI0FCN1_SPIFEN	SPI0CN0_MODF SPI0CN0_RXOVRN SPI0CN0_SPIF SPI0CN0_WCOL SPI0FCN1_RFRQ SPI0FCN1_TFRQ
SMBus 0	0x003B	7	EIE1_ESMB0	-	SMB0CN0_SI
Port Match	0x0043	8	EIE1_EMAT	-	-

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Example

```
; Basic interrupt setup

; We need the register definitions for the 8051:
$MOD52

org 0h
ljmp myprogram

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

org 3h ; External interrupt 0
reti

org 0bh ; Timer 0 interrupt
reti
```

WARNING: org directives must be sequential!

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Example (cont.)

```
org 13h ; External interrupt 1
reti

org 1bh ; Timer 1 interrupt
reti

org 23h ; Serial port interrupt
reti

org 2bh ; Timer 2 interrupt
reti

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


Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

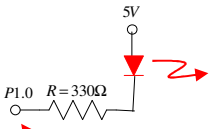
Example: Do something (useful) in the main program.

```
; 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

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Example: Enable timer 0 interrupt and setup an ISR

```
CLK           EQU 24000000 ; Crystal frequency
TIMER0_RATE   EQU 4096     ; 2048Hz square wave
TIMER0_RELOAD EQU ((65536-(CLK/TIMER0_RATE)))
myprogram:
        mov a, TMOD
        anl a, #0xf0 ; Clear the bits for timer 0
        orl a, #0x01 ; Configure timer 0 as 16-timer
        mov TMOD, a
        ; Set auto-reload value
        mov RH0, #high(TIMER0_RELOAD)
        mov RL0, #low(TIMER0_RELOAD)
        ; Enable the timer and interrupts
       setb ET0  ; Enable timer 0 interrupt
       setb TR0  ; Start timer 0
       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
```

WARNING: old 8051 code. Missing disable of WDT for EFM8LB1.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Example (cont.) the ISR.

```
; Timer 0 interrupt
org 0bh
   cpl P3.7 ; Check this pin with the scope!
   reti
```


Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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 P3.7
   reti
```


Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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 register "A" was 33 when the ISR was called, it must be set back to 33 before the ***reti***.
- 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.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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 P3.7
    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…

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Setting the SP register

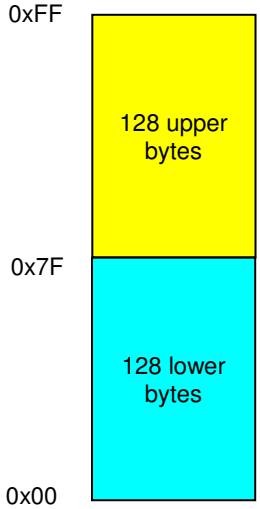
```
CLK           EQU 24000000 ; Crystal frequency
TIMER0_RATE   EQU 4096     ; 2048Hz square wave
TIMER0_RELOAD EQU ((65536-(CLK/TIMER0_RATE)))
myprogram:
        mov SP, 0x7f ; Do it once in your program!
        mov a, TMOD
        anl a, #0xf0 ; Clear the bits for timer 0
        orl a, #0x01 ; Configure timer 0 as 16-timer
        mov TMOD, a
        ; Set auto-reload value
        mov RH0, #high(TIMER0_RELOAD)
        mov RL0, #low(TIMER0_RELOAD)
        ; Enable the timer and interrupts
       setb ET0  ; Enable timer 0 interrupt
       setb TR0  ; Start timer 0
       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
```

WARNING: old 8051 code. Missing disable of WDT for EFM8LB1.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Why we put the SP at 0x7F?



Accessible only using a pointer:

mov R0, #0xa0

mov a,@R0

Accessible both directly or using a pointer:

mov R0, #0x40

mov a,@R0

Or

mov a, 0x40 ; Way easier in assembly

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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.

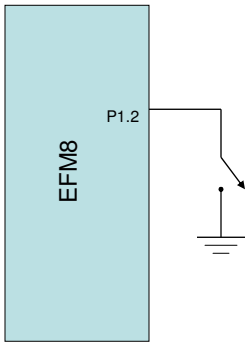
Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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!

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

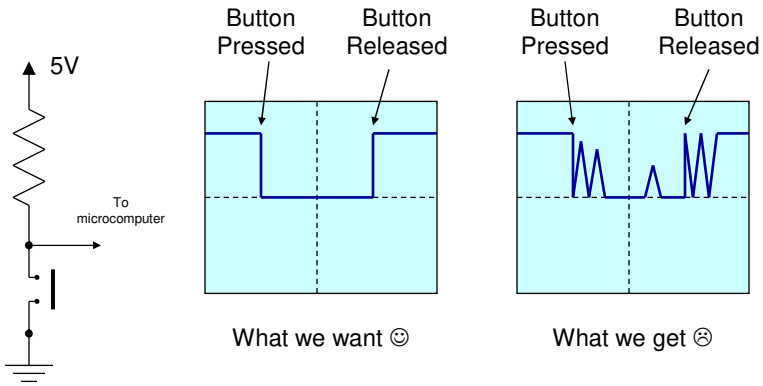
Reading Push Buttons




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


Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

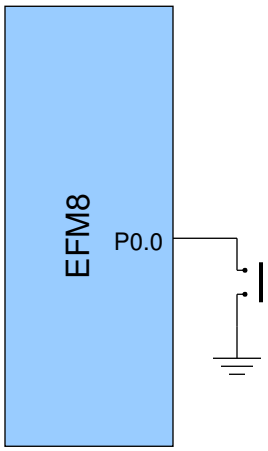
Problem: Contact Bounce



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

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

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

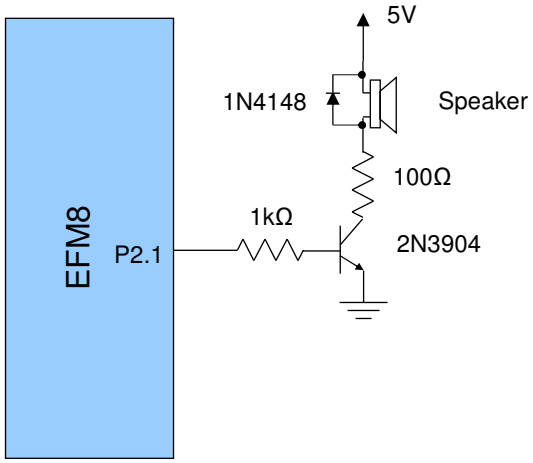
Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Speaker

2.25" Speaker, 1W, 32 Ohm

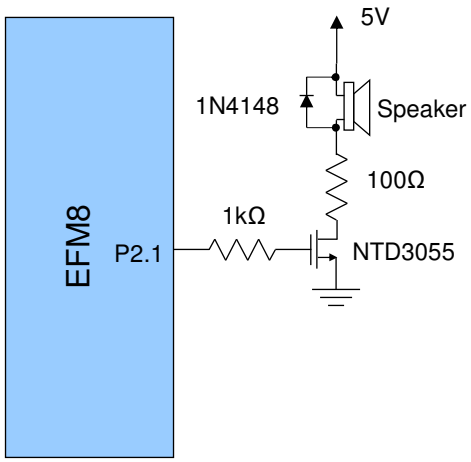
Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Driving the Speaker Using a BJT



Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Driving the Speaker Using a MOSFET



Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Lab 2

- 12h Alarm clock (AM/PM).
- Use LCD, speaker, and push buttons.
- Sample code provided:
  - ISR_example.asm: interrupt programming example.
  - LCD_4bit.inc: functions and macros to use the LCD.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Assembly Macros (time permitting)

- "A macro is a name assigned to one or more assembly statements or directives. Macros are used to include the same sequence of instructions in several places. This sequence of instructions may operate with different parameters, as indicated by the programmer."
- The MAC directive is used to define the start of a macro. A macro is a segment of instructions that is enclosed between the directives MAC and ENDMAC. The format of a macro is as follows:

```
name MAC ; comment
.
.
.
ENDMAC
```

- You can use macros to add some "flavour" of high level language to your assembly program.

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Macro Example: Duplicated code

```
Loop:
    mov P3, #0 ; all bits zero!
    lcall mydelay

    setb P3.7
    lcall mydelay
    jnb P2.4, L1a
    clr P3.7
L1a:
    setb P3.6
    lcall mydelay
    jnb P2.4, L2a
    clr P3.6
L2a:
    setb P3.5
    lcall mydelay
    jnb P2.4, L3a
    clr P3.5
L3a:
.
.
[more code here]
```

Similar code for each bit, good candidate for a macro:

```
ADC_bit MAC
    ;ADC_bit(%0, %1, %2)
    setb %0
    lcall mydelay
    jnb %1, %2
    clr %0
%2:
ENDMAC
```


Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Macro Example: first try

```
ADC_bit MAC
    ;ADC_bit(%0, %1, %2)
    setb %0
    lcall mydelay
    jnb %1, %2
    clr %0
%2:
ENDMAC
.
.
myprogram:
.
.
.
```

```
Loop:
    mov P3, #0 ; all bits zero!
    lcall mydelay

    ADC_bit(P3.7, P2.4, L1a)
    ADC_bit(P3.6, P2.4, L2a)
    ADC_bit(P3.5, P2.4, L3a)
    ADC_bit(P3.4, P2.4, L4a)
    ADC_bit(P3.3, P2.4, L5a)
    ADC_bit(P3.2, P2.4, L6a)
    ADC_bit(P3.1, P2.4, L7a)
    ADC_bit(P3.0, P2.4, L8a)

    mov val, P3 ; Save the result

    ljmp Loop
```


Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Macro Example: better macro

```
ADC_bit MAC
    ;ADC_bit(%0, %1)
    setb %0
    lcall mydelay
    jnb %1, skip%M
    clr %0
skip%M:
ENDMAC
.
.
myprogram:
.
.
.
```

%M: Macro counter

```
Loop:
    mov P3, #0 ; all bits zero!
    lcall mydelay

    ADC_bit(P3.7, P2.4)
    ADC_bit(P3.6, P2.4)
    ADC_bit(P3.5, P2.4)
    ADC_bit(P3.4, P2.4)
    ADC_bit(P3.3, P2.4)
    ADC_bit(P3.2, P2.4)
    ADC_bit(P3.1, P2.4)
    ADC_bit(P3.0, P2.4)

    mov val, P3 ; Save the result

    ljmp Loop
```

Check the .lst file to see how the macro expanded:

Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Macros after expansion:

```
001E              36   Loop:
001E 75B000       37       mov P3, #0 ; all bits zero!
0021 120003       38       lcall mydelay
0024              39
0024              40       ;ADC_bit(P3.7, P2.4)
0024 D2B7         40       setb P3.7
0026 120003       40       lcall mydelay
0029 30A402       40       jnb P2.4, skip1
002C C2B7         40       clr P3.7
002E              40   skip1:
002E              41       ;ADC_bit(P3.6, P2.4)
002E D2B6         41       setb P3.6
0030 120003       41       lcall mydelay
0033 30A402       41       jnb P2.4, skip2
0036 C2B6         41       clr P3.6
0038              41   skip2:
0038              42       ;ADC_bit(P3.5, P2.4)
0038 D2B5         42       setb P3.5
003A 120003       42       lcall mydelay
003D 30A402       42       jnb P2.4, skip3
0040 C2B5         42       clr P3.5
0042              42   skip3:
.
.
.
```


Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Unrelated question: Can I use macros in ARM assembly?

YES! ARM GNU Assembler:

.macro <name> {<arg_1} {,<arg_2>} ... {,<arg_N>}

Defines an assembler macro called <name> with N parameters. The macro definition must end with .endm. To escape from the macro at an earlier point, use .exitm. These directives are similar to MACRO, MEND, and MEXIT in armasm. You must precede the dummy macro parameters by \. For example:

```
.macro SHIFTLEFT a, b
   .if \b < 0
      MOV \a, \a, ASR #-\b
      .exitm
   .endif
   MOV \a, \a, LSL #\b
.endm
```


Copyright © 2009-2020, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.