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 21, 2022

Objectives

- Configure and use the timers in the AT89LP51RC2 microcontroller.
- Configure and use interrupts.
- Attach (and use) pushbuttons.
- Attach and use speaker with the AT89LP51RC2 microcontroller.

Timing & machine cycles

- For the AT89LP51RC2, one machine cycle takes 1 oscillator period. This year the clock is set to 22.1184 MHz: One cycle takes 45.21 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!

Timers/Counters

- Timers/Counters have 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 CPU 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). Available in the AT89LP51RC2! Very powerful, other architectures have exactly the same PCA!
  4. Additional timer/counters: time 3, 4, 5, 6, etc. Not available in the AT89LP51RC2.
- Let us begin with timers 0 and 1:

The more timers the merrier!

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 AT89LP51RC has a dedicated baud rate generator!

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

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

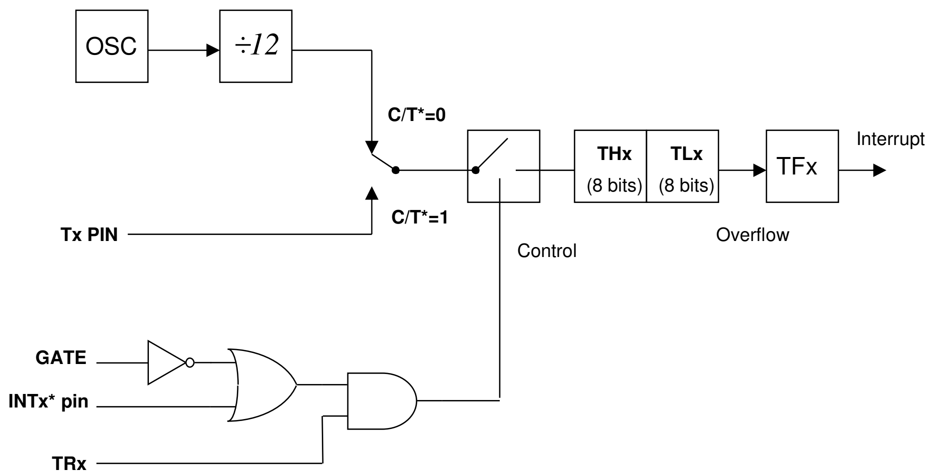
Bit	**Name**		**Description**
7 & 3	GATE		1: uses either INT0 or INT1 pins to enable/disable the timer/counter
6 & 2	C/T*		0: timer; 1: counter (pins T0 and T1)
All the other pins!	M1	M0	
	0	0	13-bit timer/counter
	0	1	16-bit timer/counter
	1	0	8-bit auto-reload timer/counter
	1	1	Special mode

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

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

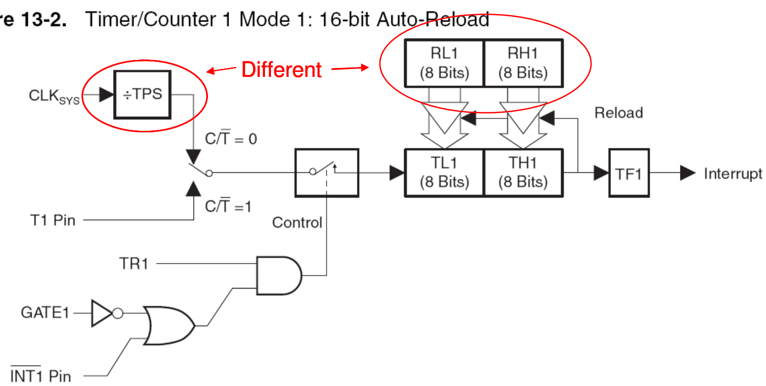
Bit	**Name**	**Description**
7	TF1	Timer 1 overflow flag.
6	TR1	Timer 1 run control.
5	TF0	Timer 0 overflow flag.
4	TR0	Timer 0 run control.
3	IE1	Interrupt 1 flag.
2	IT1	Interrupt 1 type control bit.
1	IE0	Interrupt 0 flag.
0	IT0	Interrupt 0 type control bit.

Timer/Counter 0 or 1 in Mode 1
Original 8051



AT89LP51RC2 Timer/Counter 0/1 in Mode 1 in 'Fast' mode.

Figure 13-2. Timer/Counter 1 Mode 1: 16-bit Auto-Reload



CLKREG

Table 6-8. CLKREG – Clock Register

CLKREG = AEH | Reset Value = 0101 XXXXB

Not Bit Addressable

	TPS3	TPS2	TPS1	TPS0	—	—	—	—
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, PCA 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 TPS is set to 5 for counting every six cycles (AT89C51RB2/RC2/IC2 compatibility). The prescaler is always enabled in Compatibility mode. In Fast mode the prescaler is off by default and can be individually enabled for the peripherals through the CKCON0 and CKCON1 SFRs.

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

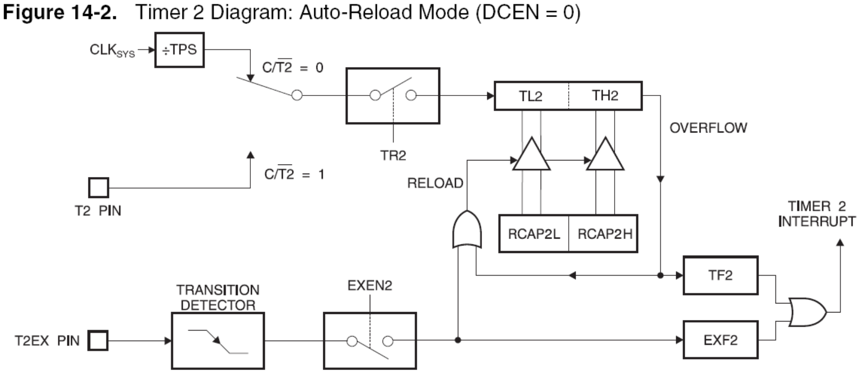
T2CON: timer/counter 2 control register. (Address C8H)

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 auto-reload mode AT89LP51RC2

Figure 14-2. Timer 2 Diagram: Auto-Reload Mode (DCEN = 0)



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.

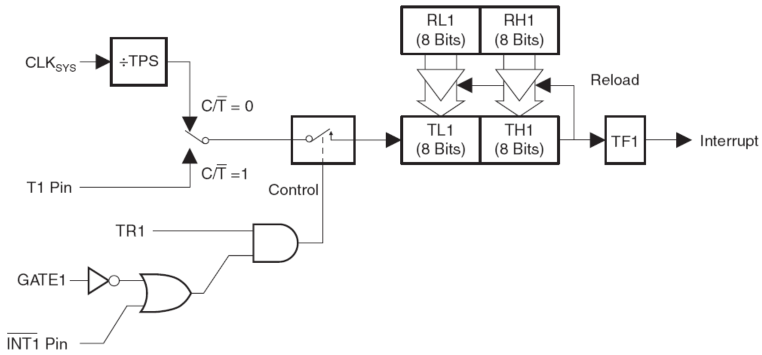
Time Delay Using a Timer

- Implement a 1 ms delay subroutine using timer 0. Assume the routine will be running in a AT89LP51RC2 microcontroller wit a 22.1184MHz in fast mode.

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

AT89LP51RC2 Timer/Counter 0/1 in Mode 1

Figure 13-2. Timer/Counter 1 Mode 1: 16-bit Auto-Reload



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?

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

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

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

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
```

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 is from "Microcontroller Technology" by Peter Spasov.

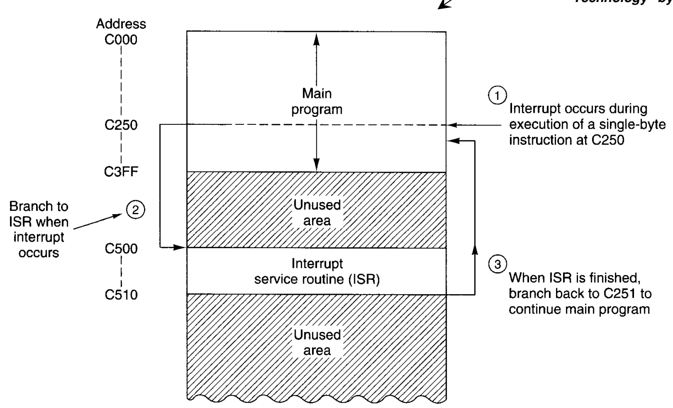


FIGURE 8.11 Example of how an interrupt causes a change in the execution of a program.

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 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) (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.

() 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".

Interrupts in the AT89LP51RC2

Table 9-3. IEN0 – Interrupt Enable Register 0

IEN0 = A8H Reset Value = 0000 0000B

Bit Addressable

	EA	EC	ET2	ES	ET1	EX1	ET0	EX0
Bit	7	6	5	4	3	2	1	0

Symbol	Function
EA	**Global Interrupt Enable** All interrupts are disabled when EA = 0. When EA = 1, each interrupt source is enabled/disabled by setting /clearing its own enable bit.
EC	**PCA Interrupt Enable** Clear to disable the PCA interrupt. Set to enable the PCA interrupt when EA = 1.
ET2	**Timer 2 Interrupt Enable** Clear to disable the Timer 2 interrupt. Set to enable the Timer 2 interrupt when EA = 1.
ES	**Serial Port Interrupt Enable** Clear to disable the UART interrupt. Set to enable the UART interrupt when EA = 1.
ET1	**Timer 1 Interrupt Enable** Clear to disable the Timer 1 interrupt. Set to enable the Timer 1 interrupt when EA = 1.
EX1	**External Interrupt 1 Enable.** Clear to disable the $\overline{INT1}$ interrupt. Set to enable the $\overline{INT1}$ interrupt when EA = 1.
ET0	**Timer 0 Interrupt Enable** Clear to disable the Timer 0 interrupt. Set to enable the Timer 0 interrupt when EA = 1.
EX0	**External Interrupt 0 Enable** Clear to disable the $\overline{INT0}$ interrupt. Set to enable the $\overline{INT0}$ interrupt when EA = 1.

Interrupts in the AT89LP51RC2

Table 9-4. IEN1 – Interrupt Enable Register 1

IEN1 = B1H						Reset Value = 0000 0000B		
Bit Addressable								
	–	–	EADC	ECMP	–	ESPI	ETWI	EKBD
Bit	7	6	5	4	3	2	1	0

Symbol	Function
EADC	**ADC Interrupt Enable.** Clear to disable the ADC interrupt. Set to enable the ADC interrupt when EA = 1.
ECMP	**Analog COmparator Interrupt Enable** Clear to disable the Analog Comparator interrupt. Set to enable the Analog Comparator interrupt when EA = 1.
ESPI	**SPI Interrupt Enable** Clear to disable the SPI interrupt. Set to enable the SPI interrupt when EA = 1.
ETWI	**TWI Interrupt Enable** Clear to disable the TWI interrupt. Set to enable the TWI interrupt when EA = 1.
EKBD	**Keyboard Interrupt Enable** Clear to disable the Keyboard interrupt. Set to enable the Keyboard interrupt when EA = 1.

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.
- The AT89LP51RC2 has the “Keyboard Interrupts” in port P1.

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 **OVERWRITE**/**CORRUPT** 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 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 AT89LP51RC2, see next slide:

Interrupts in the AT89LP51RC2

Table 9-1. Interrupt Vector Addresses and Priority

Polling Priority	Interrupt	Source	Vector Address
0	System Reset	RST or POR or BOD	0000H
1	External Interrupt 0	IE0	0003H
2	Timer 0 Overflow	TF0	000BH
3	External Interrupt 1	IE1	0013H
4	Timer 1 Overflow	TF1	001BH
6	Serial Port Interrupt	RI or TI	0023H
7	Timer 2 Interrupt	TF2 or EXF2	002BH
5	PCA Interrupt	CF, CCF0, CCF1, CCF2, CCF3 or CCF4	0033H
8	Keyboard Interrupt	KBF_{7-0}	003BH
9	Two-Wire Interrupt	SI	0043H
10	SPI Interrupt	SPIF or MODF or TXE	004BH
11	reserved		0053H
12	Analog Comparator Interrupt	CFA or CFB	005BH
13	ADC Interrupt	ADIF	0063H

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.

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.

Example: Saving and Restoring Registers in the Stack

```
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

    .
    . ; Some code that uses R1 and 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

```
CLK            EQU 22118400 ; 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!
    .
    .
    .
    .
```

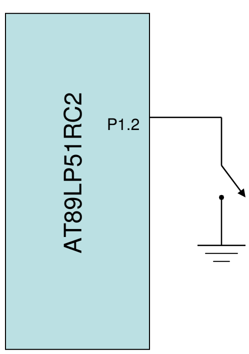

Interrupt Programming with the 8051 in Assembly (summary)

- 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

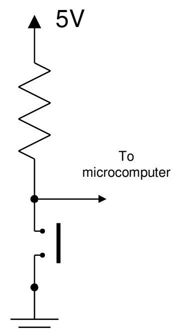
- Before using a pin for input we need to configure it:
  - Original 8051: Write ‘1’ to the pin to be used as input.
  - Newer 8051s: configure the pin as input using designated SFRs .
- In the original 8051 any pin can be used as output or input. In newer 8051s some pins can be only input and/or outputs.
- 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!

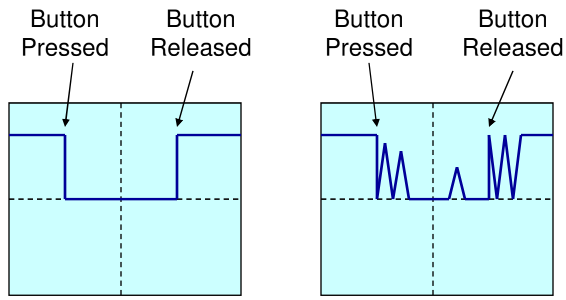
Reading Push Buttons




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

Problem: Contact Bounce

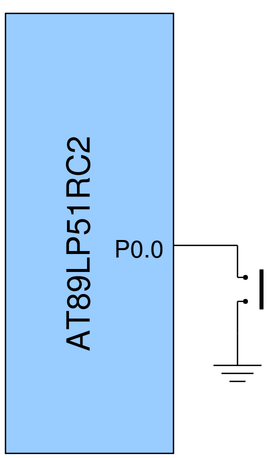




What we want ☺ What we get ☹

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

Software Debouncing




```
mov P0M0, #0 ; Configure P0
mov P0M1, #0
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

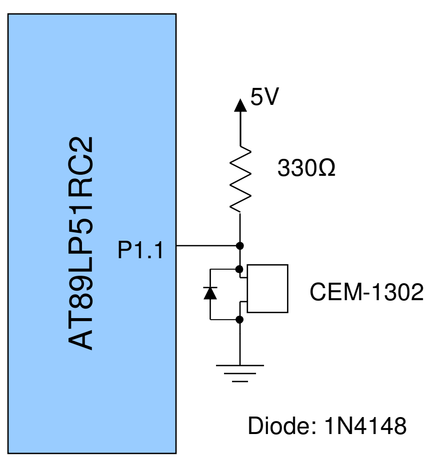
Speaker

Specifications

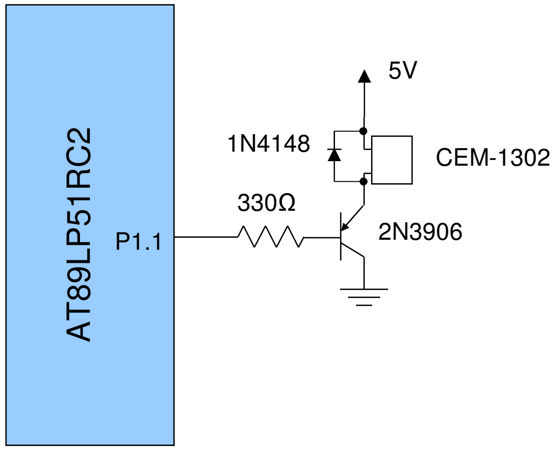
Rated voltage	3.5 Vo-p	
Operating voltage	3.0 - 5.0 Vo-p	
Mean current	35 mA max.	Applying rated voltage, 2048 Hz square wave, ½ duty
Coil resistance	42 ±6.3 Ω	
Sound output	Min. 85 (Typical 95) dBA	Distance at 10cm (A-weight free air). Applying rated voltage of 2048 Hz, square wave, 1/2 duty.
Rated frequency	2,048 Hz	
Operating temperature	-20 ~ +60° C	
Storage temperature	-30 ~ +70° C	
Dimensions	ø12.0 x H8.5 mm	See attached drawing
Weight	1.4 g	
Material	PPO (Black)	
Terminal	Pin type (AU Plating)	See attached drawing
RoHS	yes	

Louder at
2.048 kHz!

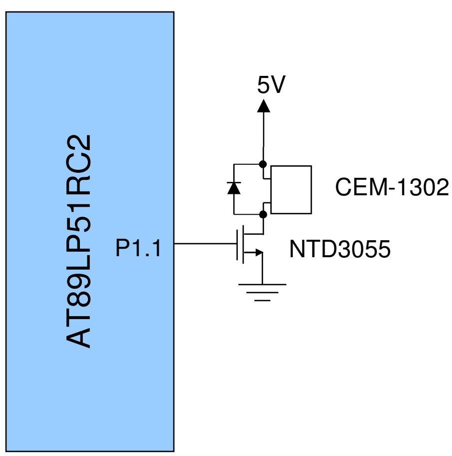
Attaching Speaker (simple way)



Attaching Speaker (louder!)



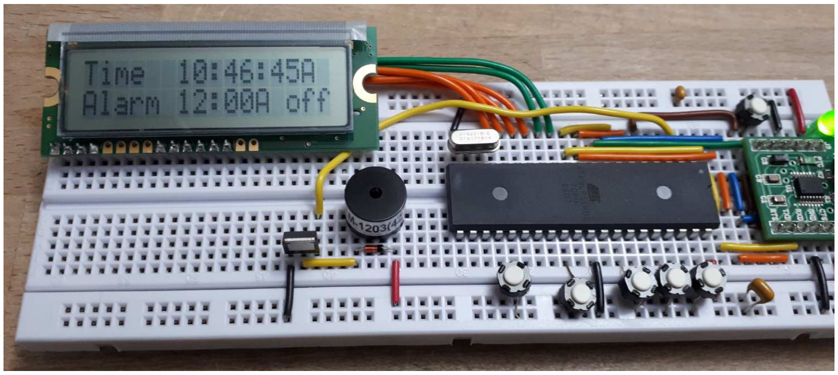
Attaching Speaker (loud and simple!)



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.

Lab 2



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.

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
```

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
```


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:

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:
.
.
.
```


Can you use macros in ARM assembly?

YES! ARM GNU Assembler manual:

.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
```