

Computer Labs: The i8254 Timer/Counter

2º MIEIC

Pedro F. Souto (pfs@fe.up.pt)

September 28, 2020

Lab 2: The PC's Timer/Counter - Part I

- ▶ Write a set of functions:

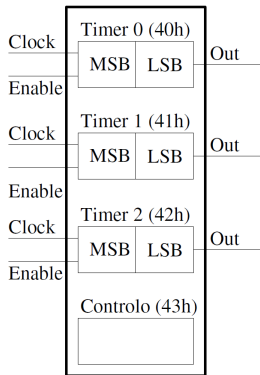
```
int timer_test_read_config(uint8_t timer,  
                           enum timer_status_field field)  
int timer_test_time_base(uint8_t timer, uint32_t freq)
```

that require programming the PC's Timer/Counter

- ▶ These functions are at a high level for pedagogical reasons
 - ▶ The idea is that you design the lower level functions (with the final project in mind)
 - ▶ In this lab we have also defined the lower level functions
- ▶ What's new?
 - ▶ Program an I/O controller: the PC's timer counter (i8254)
 - ▶ Use interrupts (Part II)

The i8254

- ▶ It is a programmable timer/counter
 - ▶ Each PC has a functionally equivalent circuit, nowadays it is integrated in the so-called south-bridge
 - ▶ Allows to measure time in a precise way, independently of the processor speed
- ▶ It has 3 16-bit counters, each of which
 - ▶ May count either in binary or BCD
 - ▶ Has 6 counting modes
 - ▶ The counting mode determines how the Out pin changes with the value of the timer/counter



i8254 Counting Modes (4 of 6)

Mode 0 Interrupt on terminal count – for counting events

- ▶ `OUT` goes high and remains high when count reaches 0

Mode 1 Hardware retriggerable one-shot

- ▶ `OUT` goes low and remains low until count reaches 0, the counter is reloaded on a rising edge of the `ENABLE` input

Mode 2 Rate Generator (divide-by-N counter)

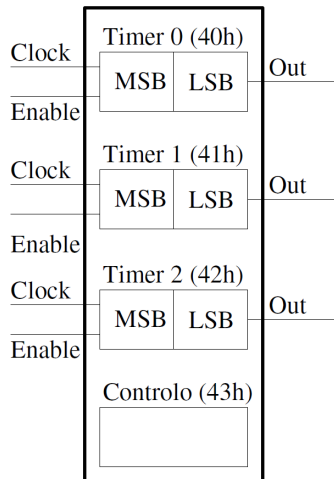
- ▶ `OUT` goes low for one clock cycle when count reaches 0, the counter is reloaded with its initial count afterwards, and ...

Mode 3 Square Wave Generator – for Lab 2

- ▶ Similar to mode 2, except for the duty-cycle: `OUT` will be high for half of the cycle and low for the remaining half of the cycle

Note In all modes, the counters perform a down count from a programmable initial counting value

i8254 Block Diagram



- ▶ Three independent 16-bit counters
 - ▶ Ports 40h, 41h and 42h
 - ▶ MSB and LSB addressable separately
 - ▶ Independent counting modes
 - ▶ Independent initial counting values
- ▶ An 8 bit-control register
 - ▶ Port 43h
 - ▶ Programming of each counter independently

i8254 Control Word

- ▶ Used to program the timers, one at a time
- ▶ The control word must be written to the Control Register (0x43)
- ▶ The initial counting value must be written on the timer's port (one of 0x40, 0x41, 0x42)
 - ▶ If programming the initial value of a single byte, the other byte will be initialized to 0

Bit	Value	Function
7,6		Counter selection
	00	0
	01	1
	10	2
5,4		Counter Initialization
	01	LSB
	10	MSB
	11	LSB followed by MSB
3,2,1		Counting Mode
	000	0
	001	1
	x10	2
	x11	3
	100	4
	101	5
0		BCD
	0	Binary (16 bits)
	1	BCD (4 digits)

i8254 Control Word: Example

Bit	Value	Function
7,6		Counter selection
	00	0
	01	1
	10	2
5,4		Counter Initialization
	01	LSB
	10	MSB
	11	LSB followed by MSB
3,2,1		Counting Mode
	000	0
	001	1
	x10	2
	x11	3
	100	4
	101	5
0		BCD
	0	Binary (16 bits)
	1	BCD (4 digits)

Example

- ▶ Timer 2 in mode 3
- ▶ Binary counting
- ▶ Initial counting value: 1234 = 0x04D2

Control Register: 10110110

- ▶ **"NOTE:** Don't care bits (X) should be 0 to insure compatibility with future Intel products."

Timer2 LSB 0xD2

Timer2 MSB 0x04

How to assemble the control word?

How to assemble the control word?

Use bitwise operations

Use the macros defined in i8254.h

i8254: Read-Back Command

The command

- ▶ Allows to retrieve
 - ▶ the programmed configuration
 - ▶ and/or the current counting value
- of one or more timers
 - ▶ The bars over COUNT and STATUS means that these bits are active in 0
- ▶ Written to the Control Register (0x43)

Reading of the status/count

- ▶ The configuration (status) is read from the timer's data register
 - ▶ The 6 LSBs match those of the Control Word

Read-Back Command Format

Bit	Value	Function
7,6		Read-Back Command
	11	
5		$\overline{\text{COUNT}}$
	0	Read counter value
4		$\overline{\text{STATUS}}$
	0	Read programmed mode
3		Select Timer 2
	1	Yes
2		Select Timer 1
	1	Yes
1		Select Timer 0
	1	Yes
0		Reserved

Read-Back Status Format

Bit	Value	Function
7		Output
6		Null Count
5,4		Counter Initialization
3,2,1		Programmed Mode
0		BCD

How to parse the the status word?

Use bitwise operations

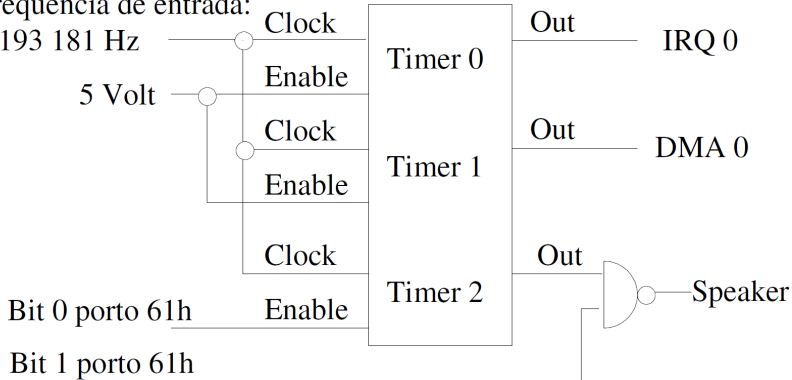
Use the macros defined in i8254.h

i8254: Use in the PC (1/2)

Frequência de entrada:

1 193 181 Hz

5 Volt



- ▶ Timer 0 is used to provide a time base.
- ▶ Timer 1 is used for DRAM refresh
 - ▶ Via DMA channel 0(Not sure this is still true.)
- ▶ Timer 2 is used for tone generation

i8254: Use in the PC (2/2)

- ▶ The i8254 is mapped in the I/O address space:

Timer 0: 0x40

Timer 1: 0x41

Timer 2: 0x42

Control Register: 0x43

- ▶ Need to use IN/OUT assembly instructions

- ▶ Minix 3 provides the `SYS_DEVIO` kernel call for doing I/O

```
#include <minix/syslib.h>
```

```
int sys_inb(int port, u32_t *byte);
```

```
int sys_outb(int port, u32_t byte);
```

- ▶ **Note** that the second argument of `sys_inb()` must be the address of a 32-bit unsigned integer variable.

- ▶ **Hint** (must) implement

```
util_sys_inb(int port, u8_t *byte)
```

- ▶ This is a wrapper to `sys_inb()`

- ▶ You can use it thereafter instead of `sys_inb()`

- ▶ Need to write to the control register before accessing any of the timers

- ▶ Both to program (control word) a timer, or to read its configuration (read-back command)

Minix 3 and Timer 0

- ▶ At boot time, Minix 3 programs Timer 0 to generate a square wave with a fixed frequency
 - ▶ Timer 0 will generate an interrupt at a fixed rate:
 - ▶ Its output is connected to `IRQ0`
- ▶ Minix 3 uses these interrupts to measure time
 - ▶ The interrupt handler increments a global variable on every interrupt
 - ▶ The value of this variable increments at a fixed, known, rate
- ▶ Minix 3 uses this variable mainly for:
 - ▶ Keeping track of the date/time
 - ▶ Implementing SW timers

Lab 2: Part 1 - Reading Timer Configuration (1/2)

What to do? Read timer configuration in Minix

```
int timer_test_read_config(uint8_t timer,  
                           enum timer_status_field field)
```

1. Write read-back command to read input timer configuration:
 - ▶ Make sure 2 MSBs are both 1
 - ▶ Select only the status (not the counting value)
 - ▶ Remember, these are active low, i.e. when the bit value is 0
2. Read the timer port
3. Parse the configuration read
4. Call the function `timer_print_config()` that **we provide you**

How to design it? Try to develop an API that can be used in the project.

```
int timer_get_conf(uint8_t timer, uint8_t *st);  
int timer_display_conf(uint8_t timer, uint8_t st,  
                       enum timer_status_field status);
```

Lab 2: Part 1 - Reading Timer Configuration (2/2)

Stuff we provide you

```
int timer_print_config(uint8_t timer,
                      enum timer_status_field field,
                      union timer_status_field_val val);

enum timer_status_field {
    tsf_all,          // configuration in hexadecimal
    tsf_initial,      // timer initialization mode
    tsf_mode,          // timer counting mode
    tsf_base           // timer counting base
};

enum timer_init {
    INVALID_val,
    LSB_only,
    MSB_only,
    MSB_after_LSB
};

union timer_status_field_val {
    uint8_t          byte;          // status, in hexadecimal
    enum timer_init  in_mode;       // initialization mode
    uint8_t          count_mode;    // counting mode: 0, 1, ..., 5
    bool             bcd;           // true, if counting in BCD
};
```

C Enumerated Types

- ▶ This is a user-defined type that can take one of a finite number of values

```
enum timer_status_field {  
    tsf_all,          // configuration in hexadecimal  
    tsf_initial,      // timer initialization mode  
    tsf_mode,         // timer counting mode  
    tsf_base          // timer counting base  
};  
enum timer_status_field info = tsf_base;
```

- ▶ The C compiler represents each possible value of an enumerated type by an integer value. By default:
 - ▶ The first value is represented with 0
 - ▶ Any other value, is one more than the previous value
- ▶ However, it is possible to assign to an enumerated value an integer value different from the default (e.g.
 `tsf_all = 255;`)
- ▶ The names of the members of an enumerated type have global scope
 - ▶ To avoid collisions we use the `tsf_` prefix
- ▶ The use of enumerated types makes the code more readable

C Unions

- ▶ Syntactically, a union data type appears like a struct:

```
union timer_status_field_val {  
    uint8_t      byte;           // status, in hexadecimal  
    enum timer_init in_mode;     // initialization mode  
    uint8_t      count_mode;    // counting mode: 0, 1, ...,  
    bool         bcd;           // true, if counting in BCD  
};
```

- ▶ Access to the members of a union is via the dot operator
- ▶ However, semantically, there is a big difference:
Union contains space to store **any** of its members, but **not all** of its members simultaneously

- ▶ The name **union** stems from the fact that a variable of this type can take values of **any** of the types of its members

Struct contains space to store **all** of its members simultaneously

In `timer_print_config()` we are using it to reduce the number of arguments passed

- ▶ But need another argument the kind of information passed

Lab 2: Part 1 - Setting the Time-Base (1/2)

What to do? Change the rate at which a timer 0 generates interrupts.

```
int timer_test_time_base(uint8_t timer, uint32_t freq)
```

1. Write control word to configure Timer 0:

- ▶ **Do not change 4 least-significant bits**

- ▶ Mode (3)
- ▶ BCD/Binary counting

You need to read the Timer 0 configuration first.

- ▶ Preferably, LSB followed by MSB

2. Load timer's register with the value of the divisor to generate the frequency corresponding to the desired rate

- ▶ Depends on the previous step
- ▶ Remember that the frequency of the `Clock` input of all timers is 1 193 181 Hz

Lab 2: Part 1 - Setting the Time-Base (2/2)

How to design it? Try to develop an API that can be used in the project.

```
int timer_set_frequency(uint8_t timer, uint32_t freq);
```

This function should work for every timer, not only Timer 0.

How do we know it works? Use the `date` command.

Minix 3 programs Timer 0 to generate interrupts at a fixed rate (60 Hz) at boot-time and assumes that rate is not changed thereafter

- ▶ By programming a different rate, Minix 3 will measure time incorrectly. E.g. with a 30 Hz rate ...
- ▶ Or, even better, use the test code provided.

Lab 2: Grading Criteria (in previous years)

SVN (5%) Whether or not your code is in the right place (under `lab2/`, of the repository's root)

► Also, evidence of incremental development approach

Execution (80%) Including automatic code grading.

Code (15%)

return values of function/kernel calls must be checked

global variables only if you cannot do what you want, or if they can be considered fields/members of an object (if using object oriented design)

symbolic constants i.e. use `#define`

modularity both at the level of functions and **at the level of files**

Self-evaluation **Must submit** it by filling a [Google Form](#) (check the handout)

IMPORTANT Please follow exactly the instructions, otherwise you may be penalized

Further Reading

- ▶ Lab 2 Handout
- ▶ i8254 Data-sheet