I/O ports and Hardware Interrupts

The emulator does not reproduce any input/output devices of the original IBM PC ®, however theoretically it may be possible to create emulation of the original ibm pc devices. emu8086 supports user-created virtual devices that can be accessed from assembly language program using **in** and **out** instructions. devices that can be created by anyone with basic programming experience in any high or low level programming language. the simplest virtual device in assembly language can be found in examples: **simplest.asm**

Input / Output ports

emu8086 supports additional devices that can be created by anyone with basic programming experience in any language device can be written in any language, such as: java, visual basic, vc++, delphi, c#, .net or in any other programming language that allow to directly read and write files. for more information and sample source code look inside this folder: **c:\emu8086\DEVICES\DEVELOPER**

The latest version of the emulator has no reserved or fixed I/O ports, input / output addresses for custom devices are from **0000 to 0FFFFh** (0 to 65535), but it is important that two devices that use the same ports do not run simultaneously to avoid hardware conflict.

Port 100 corresponds to byte 100 in this file: **c:\emu8086.io** , port 0 to byte 0, port 101 to byte 101, etc...

Emulation of Hardware Interrupts

External hardware interrupts can be triggered by external peripheral devices and microcontrollers or by the 8087 mathematical coprocessor.

Hardware interrupts are disabled when interrupt flag (IF) is set to 0. when interrupt flag is set to 1, the emulator continually checks first 256 bytes of this file **c:\emu8086.hw** if any of the bytes is none-zero the microprocessor transfers control to an interrupt handler that matches the triggering byte offset in **emu8086.hw** file (0 to 255) according to the interrupt vector table (memory 0000-0400h) and resets the byte in **emu8086.hw** to 00.

These instructions can be used to disable and enable hardware interrupts:

cli - clear interrupt flag (disable hardware interrupts).
sti - set interrupt flag (enable hardware interrupts).

by default hardware interrupts are enabled and are disabled automatically when software or hardware interrupt is in the middle of the execution.

Examples of Custom I/O Devices

Ready devices are available from **virtual devices** menu of the emulator.

- **Traffic Lights** - port 4 (word)

  the traffic lights are controlled by sending data to i/o port 4.
  there are 12 lamps: 4 green, 4 yellow, and 4 red.

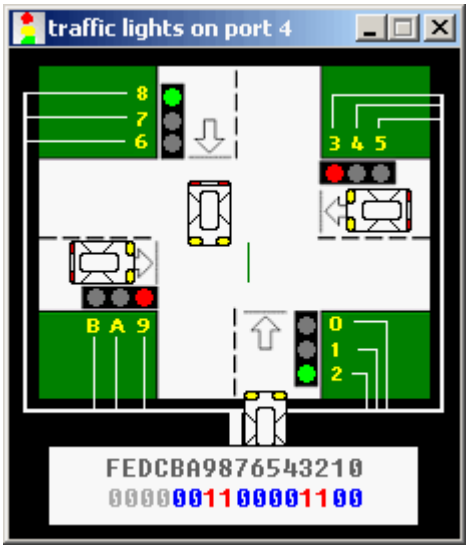
  you can set the state of each lamp by setting its bit:

  1 - the lamp is turned on.
  0 - the lamp is turned off.

  only 12 low bits of a word are used (0 to 11), last bits (12 to 15) are unused.

  for example:

  MOV AX, 0000001011110100b
  OUT 4, AX



  we use yellow hexadecimal digits in caption (to achieve compact view), here's a conversion:

  Hex - Decimal

  A - 10
  B - 11
  C - 12 (unused)
  D - 13 (unused)
  E - 14 (unused)
  F - 15 (unused)

first operand for **OUT** instruction is a port number (**4**), second operand is a word (**AX**) that is written to the port. first operand must be an immediate byte value (0..255) or **DX** register. second operand must be **AX** or **AL** only.

see also **traffic_lights.asm** in c:\emu8086\examples.

---

if required you can read the data from port using **IN** instruction, for example:

```
IN AX, 4
```

first operand of **IN** instruction (**AX**) receives the value from port, second operand (**4**) is a port number. first operand must be **AX** or **AL** only. second operand must be an immediate byte value (0..255) or **DX** register.

- **Stepper Motor** - port 7 (byte)

  the stepper motor is controlled by sending data to i/o port 7.

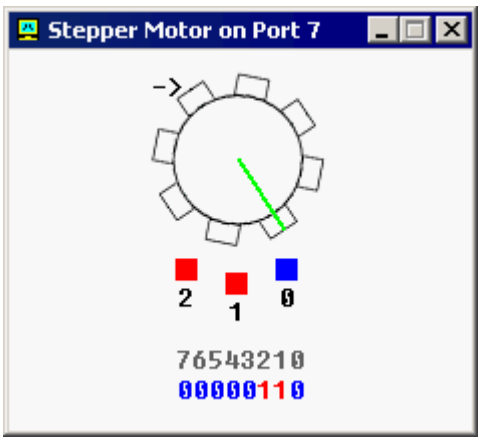
  stepper motor is electric motor that can be precisely controlled by signals from a computer.

  the motor turns through a precise angle each time it receives a signal.

  by varying the rate at which signal pulses are produced, the motor can be run at different speeds or turned through an exact angle and then stopped.

  This is a basic 3-phase stepper motor, it has 3 magnets controlled by bits **0, 1 and 2**. other bits (3..7) are unused.

  When magnet is working it becomes red. The arrow in the left upper corner shows the direction of the last motor move. Green line is here just to see that it is really rotating.



For example, the code below will do three clock-wise half-steps:

```
MOV AL, 001b ; initialize.
OUT 7, AL

MOV AL, 011b ; half step 1.
OUT 7, AL

MOV AL, 010b ; half step 2.
OUT 7, AL

MOV AL, 110b ; half step 3.
OUT 7, AL
```

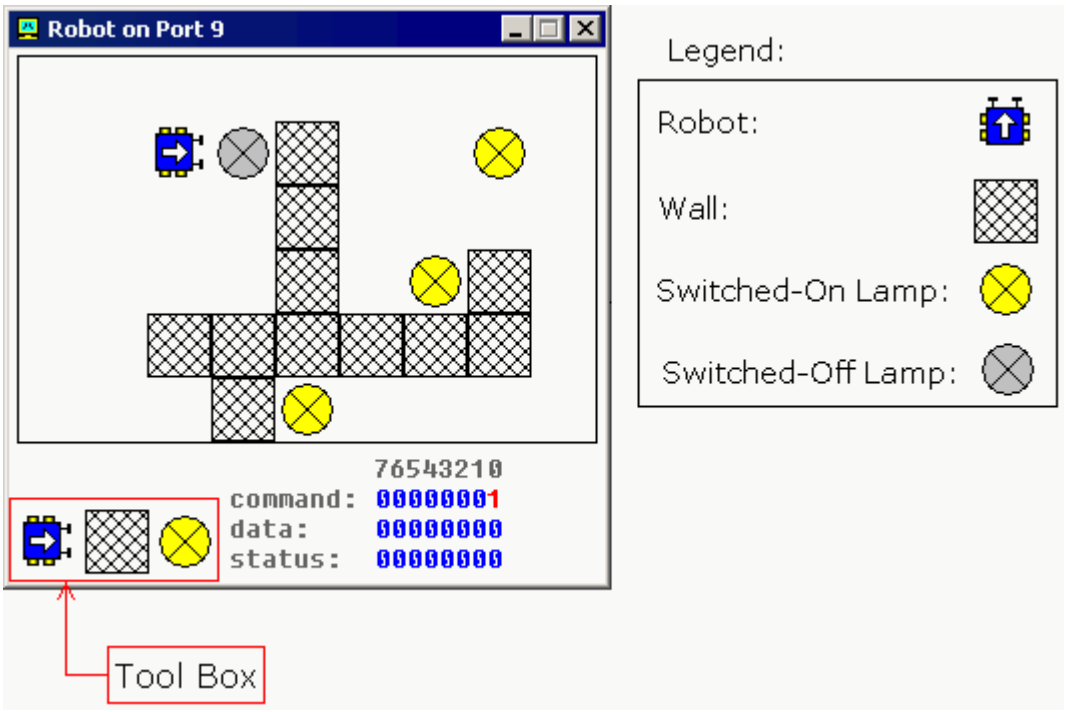
If you ever played with magnets you will understand how it works. try experimenting, or see **stepper_motor.asm** in c:\emu8086\examples.

If required you can read the data from port using **IN** instruction, for example:

```
IN AL, 7
```

Stepper motor sets topmost bit of byte value in port 7 when it's ready.

- **Robot** - port 9 (3 bytes)



The robot is controlled by sending data to i/o port 9.

The first byte (port 9) is a **command register**. set values to this port to make

robot do something. supported values:

decimal value	binary value	action
0	00000000	do nothing.
1	00000001	move forward.
2	00000010	turn left.
3	00000011	turn right.
4	00000100	examine. examines an object in front using sensor. when robot completes the task, result is set to **data register** and **bit #0** of **status register** is set to **1**.
5	00000101	switch on a lamp.
6	00000110	switch off a lamp.

The second byte (port 10) is a **data register**. this register is set after robot completes the **examine** command:

decimal value	binary value	meaning
255	11111111	wall
0	00000000	nothing
7	00000111	switched-on lamp
8	00001000	switched-off lamp

The third byte (port 11) is a **status register**. read values from this port to determine the state of the robot. each bit has a specific property:

bit number	description
bit #0	**zero** when there is no new data in **data register**, **one** when there is new data in **data register**.
bit #1	**zero** when robot is ready for next command, **one** when robot is busy doing some task.
bit #2	**zero** when there is no error on last command execution, **one** when there is an error on command execution (when robot

	cannot complete the task: move, turn, examine, switch on/off lamp).

example:

```
MOV AL, 1  ; move forward.
OUT 9, AL  ;

MOV AL, 3  ; turn right.
OUT 9, AL  ;

MOV AL, 1  ; move forward.
OUT 9, AL  ;

MOV AL, 2  ; turn left.
OUT 9, AL  ;

MOV AL, 1  ; move forward.
OUT 9, AL  ;
```

keep in mind that robot is a mechanical creature and it takes some time for it to complete a task. you should always check bit#1 of **status register** before sending data to port 9, otherwise the robot will reject your command and "**busy!**" will be shown. see **robot.asm** in c:\emu8086\examples.

Creating Custom Robo-World Map

It is possible to change the default map for the robot using the tool box.

if you click the robot button and place robot over existing robot it will turn 90 degrees counter-clock-wise. to manually move the robot just place it anywhere else on the map.

If you click lamp button and click switched-on lamp the lamp will be switched-off, if lamp is already switched-off it will be deleted. click over empty space will create a new switched-on lamp.

Placing wall over existing wall deletes the wall.

Current version is limited to a single robot only. if you forget to place a robot on the map it will be placed in some random coordinates.

When robot device is closed the map is automatically saved inside this file:
c:\emu8086\devices\robot_map.dat
It is possible to have several maps by renaming and coping this file before starting the robot device.

The right-click over the map brings up a popup menu that allows to switch-on or switch-off all the lamps at once.

For a list of frequently asked questions click **here**.