

# Using Interrupts Version 1.0

---

**Disclaimer :** While every effort has been made to make sure the information in this document is correct, the author can not be liable for any damages whatsoever for loss relating to this document. Use this information at your own risk.

---

## Table of Contents

What are Interrupts?	Page 2
Interrupts and Intel Architecture	Page 3
Hardware Interrupts	Page 3
Implementing the Interrupt Service Routine (ISR)	Page 5
Using your new Interrupt Service Routine	Page 6
The Programmable Interrupt Controller	Page 7
IRQ2/IRQ9 Redirection	Page 9
Programmable Interrupt Controller's Addresses	Page 10
Initialization Command Word 1 (ICW1)	Page 11
Initialization Command Word 2 (ICW2)	Page 12
Initialization Command Word 3 (ICW3)	Page 12
Initialization Command Word 4 (ICW4)	Page 14
Operation Control Word 1 (OCW1)	Page 15
Operation Control Word 2 (OCW2)	Page 16
Operation Control Word 3 (OCW3)	Page 17

## *What are Interrupts?*

---

When receiving data and change in status from I/O Ports, we have two methods available to us. We can *Poll* the port, which involves reading the status of the port at fixed intervals to determine whether any data has been received or a change of status has occurred. If so, then we can branch to a routine to service the ports requests.

As you could imagine, polling the port would consume quite some time. Time which could be used doing other things such refreshing the screen, displaying the time etc. A better alternative would be to use *Interrupts*. Here, the processor does your tasks such as refreshing the screen, displaying the time etc, and when a I/O Port/Device needs attention as a byte has been received or status has changed, then it sends a Interrupt Request (IRQ) to the processor.

Once the processor receives an Interrupt Request, it finishes its current instruction, places a few things on the stack, and executes the appropriate Interrupt Service Routine (ISR) which can remove the byte from the port and place it in a buffer. Once the ISR has finished, the processor returns to where it left off.

Using this method, the processor doesn't have to waste time, looking to see if your I/O Device is in need of attention, but rather the device will *interrupt* the processor when it needs attention.

## *Interrupts and Intel Architecture*

---

Interrupts do not have to be entirely associated with I/O devices. The 8086 family of microprocessors provides 256 interrupts, many of these are only for use as software interrupts, which we do not attempt to explain in this document.

The 8086 series of microprocessors has an Interrupt Vector Table situated at 0000:0000 which extends for 1024 bytes. The Interrupt Vector table holds the address of the Interrupt Service Routines (ISR), all four bytes in length. This gives us room for the 256 Interrupt Vectors.

INT (Hex)	IRQ	Common Uses
00 - 01	Exception Handlers	
02	Non-Maskable IRQ	Non-Maskable IRQ (Parity Errors)
03 - 07	Exception Handlers	
08	Hardware IRQ0	System Timer
09	Hardware IRQ1	Keyboard
0A	Hardware IRQ2	Redirected
0B	Hardware IRQ3	Serial Comms. COM2/COM4
0C	Hardware IRQ4	Serial Comms. COM1/COM3
0D	Hardware IRQ5	Reserved/Sound Card

0E	Hardware IRQ6	Floppy Disk Controller
0F	Hardware IRQ7	Parallel Comms.
10 - 6F	Software Interrupts	
70	Hardware IRQ8	Real Time Clock
71	Hardware IRQ9	Redirected IRQ2
72	Hardware IRQ10	Reserved
73	Hardware IRQ11	Reserved
74	Hardware IRQ12	PS/2 Mouse
75	Hardware IRQ13	Math's Co-Processor
76	Hardware IRQ14	Hard Disk Drive
77	Hardware IRQ15	Reserved
78 - FF	Software Interrupts	

Table 1 : x86 Interrupt Vectors

The average PC, only has 15 Hardware IRQ's plus one Non-Maskable IRQ. The rest of the interrupt vectors are used for software interrupts and exception handlers. Exception handlers are routines like ISR's which get called or *interrupted* when an error results. Such an example is the first Interrupt Vector which holds the address of the Divide By Zero, Exception handler. When a divide by zero occurs the Microprocessor fetches the address at 0000:0000 and starts executing the code at this Address.

## *Hardware Interrupts*

---

The Programmable Interrupt Controller (PIC) handles hardware interrupts. Most PC's will have two of them located at different addresses. One handles IRQ's 0 to 7 and the other, IRQ's 8 to 15, giving a total of 15 individual IRQ lines, as the second PIC is cascaded into the first, using IRQ2.

Most of the PIC's initialization is done by BIOS, thus we only have to worry about two instructions. The PIC has a facility available where we can *mask* individual IRQ's so that these requests will not reach the Processor. Thus the first instruction is to the Operation Control Word (OCW1) to set which IRQ's to mask and which IRQ's not too.

As there are two PIC's located at different addresses, we must first determine which PIC we need to use. The first PIC, located at Base Address 0x20h controls IRQ 0 to IRQ 7. The bit format of PIC1's Operation Control Word 1 is shown on the next page in table 2.

Bit	Disable IRQ	Function
7	IRQ7	Parallel Port
6	IRQ6	Floppy Disk Controller
5	IRQ5	Reserved/Sound Card
4	IRQ4	Serial Port
3	IRQ3	Serial Port
2	IRQ2	PIC2
1	IRQ1	Keyboard
0	IRQ0	System Timer

Table 2 : PIC1 Operation Control Word 1 (0x21)

*Note that IRQ 2 is connected to PIC2, thus if you mask this IRQ, then you will be disabling IRQ's 8 to 15.*

The second PIC located at a base address of 0xA0h controls IRQs 8 to 15. Below is the individual bits required to make up it's Operation Control Word.

Bit	Disable IRQ	Function
7	IRQ15	Reserved
6	IRQ14	Hard Disk Drive
5	IRQ13	Maths Co-Processor
4	IRQ12	PS/2 Mouse
3	IRQ11	Reserved
2	IRQ10	Reserved
1	IRQ9	Redirected IRQ2
0	IRQ8	Real Time Clock

Table 3 : PIC2 Operation Control Word 1 (0xA1)

As the above table shows the bits required to disable an IRQ, we must invert them should we want to enable an IRQ. For example, if we want to enable IRQ 3 then we would send the byte 0xF7 as OCW1 to PIC1. But what happens if one of these IRQs are already enabled and then we come along and disable it?

Therefore we must first get the mask and use the AND function to output the byte back to the register with our changes so to cause the least upset to the other IRQs. Going back to our IRQ3 example, we could use `outportb(0x21, (inportb(0x21) & 0xF7));` to enable IRQ3. Take note that the OCW1 goes to the register at Base + 1.

The same procedure must be used to mask (disable) an IRQ once we are finished with it. However this time we must OR the byte 0x08 to the contents of OCW1. Such an example of code is `outportb(0x21, (inportb(0x21) | 0x08));`

The other PIC instruction we have to worry about is the End of Interrupt (EOI). This is sent to the PIC at the end of the Interrupt Service Routine so that the PIC can reset the In Service Register. See *The Programmable Interrupt Controller* (Page 7) for more information. An EOI can be sent using `outportb(0x20, 0x20);` for PIC1 or `outportb(0xA0, 0x20);` for PIC2

## *Implementing the Interrupt Service Routine (ISR)*

---

In C you can implement your ISR using `void interrupt yourisr()` where `yourisr` is a far pointer, pointing to the address that your Interrupt Service Routine will reside in memory. This is later placed in the Interrupt Vector Table so that, it will be called when interrupted.

The following code is a basic implementation of an ISR.

```
void interrupt yourisr()    /* Interrupt Service Routine (ISR) */
{
    disable();

    /* Body of ISR goes here */

    oldhandler();

    outportb(0x20, 0x20);    /* Send EOI to PIC1 */
    enable();
}
```

`void interrupt yourisr()` defines this function as an Interrupt Service Routine. `disable();` clears the interrupt flag, so that no other hardware interrupts, except a NMI (Non-Maskable Interrupt) can occur. Otherwise, an interrupt with a higher priority than this one can interrupt the execution of this ISR. However this is not really a problem in many cases, thus is optional.

The body of your ISR will include code which you want to execute upon this interrupt request being activated. Most Ports/UARTs may interrupt the processor for a range of reasons, eg byte received, time-outs, FIFO buffer empty, overruns etc, thus the nature of the interrupt has to be determined. This is normally achieved by reading the status registers of the port you are using. Once it has been established, you can service its requests.

If you read any data from a port, it is normally common practice to place it in a buffer, rather than immediately writing it to the screen, inhibiting further interrupts to be processed. Most Ports these days will have FIFO buffers which can contain more than one byte, thus repeat your read routine, until the FIFO is empty, then exit your ISR.

In some cases it may be appropriate to chain the old ISR to this one. Such an example would be the Clock Interrupt. Other TSR or resident programs may also be using it, thus if you intercept the interrupt and keep it all for yourself, the other ISR's can no longer function possibly causing some side effects. However for Serial/Parallel Ports this is not a problem. To chain the old ISR to your new ISR, you can call it using `oldhandler()`; where `oldhandler` points to your old ISR.

Before we can return from the interrupt, we must tell the Programmable Interrupt Controller, that we are ending the interrupt by sending an EOI (End of Interrupt 0x10) to it. As there are two PIC's you must first establish which one to send it to. Use `outportb(0x20,0x20)`; for PIC 1 (IRQ 0 - 7) or `outportb(0xA0,0x20)`; for PIC 2 (IRQ 8 - 15).

*Note: If using PIC2, then an EOI has to be sent to both PIC1 and PIC2.*

## Using your new Interrupt Service Routine

Now that we have written our new interrupt service routine, we can start looking at how to implement it. The following code segment shows the basic usage of your new ISR. For this example we have chosen to use IRQ 3.

```
#include <dos.h>

#define INTNO 0x0B                                /* Interrupt Number - See Table 1 */

void main(void)
{
    oldhandler = getvect(INTNO);                  /* Save Old Interrupt Vector */
    setvect(INTNO, yourisr);                      /* Set New Interrupt Vector Entry */
    outportb(0x21,(inportb(0x21) & 0xF7)); /* Un-Mask (Enable) IRQ3 */

    /* Set Card - Port to Generate Interrupts */

    /* Body of Program Goes Here */

    /* Reset Card - Port as to Stop Generating Interrupts */

    outportb(0x21,(inportb(0x21) | 0x08)); /* Mask (Disable) IRQ3 */
    setvect(INTNO, oldhandler);                /* Restore old Interrupt Vector Before
                                                Exit */
}
```

Before you can place the address of your new ISR in the interrupt vector table, you must first save the old interrupt vector, so that you can restore it once you exit your program. This is done using `oldhandler = getvect(INTNO)`; where `INTNO` is the number of the interrupt vector you wish to return. Before `oldhandler` can be used, you must first declare it using `void interrupt (*oldhandler)();`

Once the old interrupt vector is stored, we can now install your new ISR into the interrupt vector table. This is done using the line `setvect(INTNO, yourisr)`; where `yourisr` points to your interrupt service routine.

The IRQ which you are using must now be unmasked. We have already discussed this earlier. See *Hardware Interrupts* (Page 3.)

Most Ports/UARTs will need some initialization to be able to generate interrupts. For Example, The Standard Parallel Port (SPP) will require Bit 4 of the Control Port, *Enable IRQ Via ACK Line* to be set at Base + 2. The Serial Port will require the appropriate setting of Bits 0 to 4 of the Interrupt Enable Register (IER) located at Base + 1.

Your body of the program normally consists of a few housekeeping tasks depending upon your application. Here you look for new keys pressed, menus being selected, updating clocks, checking for incoming data in buffers etc, knowing that any data from your Ports will be automatically read and processed, by the ISR.

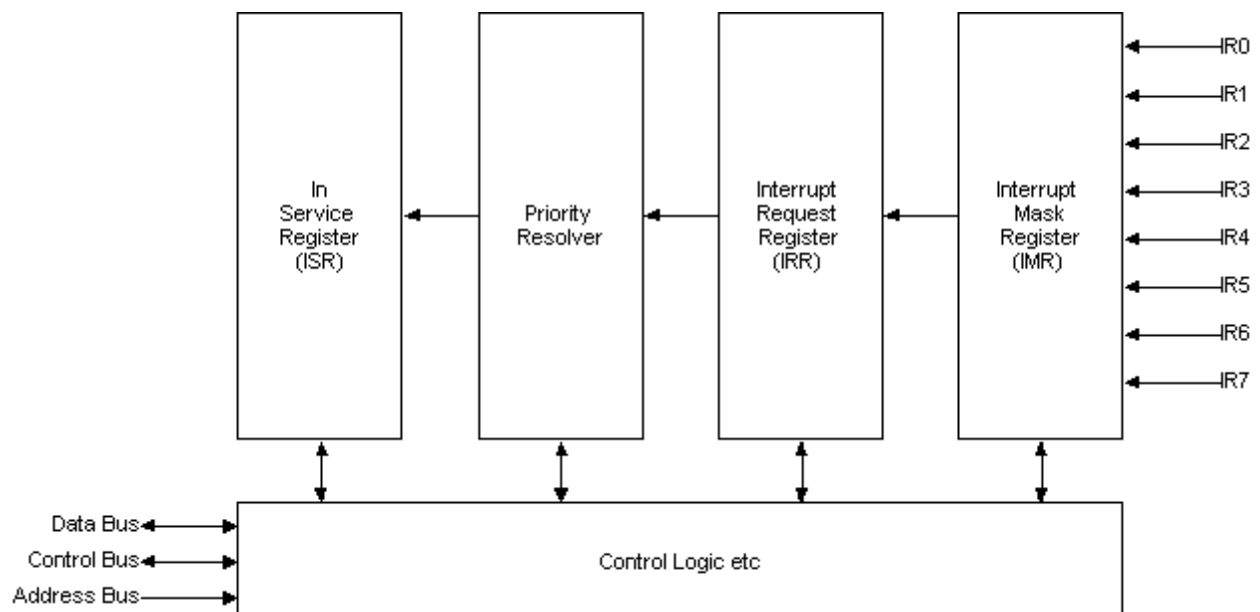
If you like implementing ISR's so much, you can attach your own ISR to the Keyboard Interrupt, so any keys being pressed will be automatically handled by another ISR, and even one to the clock. Upon every 18.2 ticks you can update the seconds on your display! The possibilities of ISR's are endless.

Before you exit your program always restore the old interrupt vector, so that your computer doesn't become unstable. This is done using `setvect(INTNO, oldhandler);`, where `oldhandler` points to the old interrupt service routine, which we stored using `oldhandler = getvect(INTNO);`

## *The Programmable Interrupt Controller*

---

As we have all ready discussed, the Interrupt ReQuests (IRQ's) of a PC is handled by two 8259 Programmable Interrupt Controllers. On the old XT's/AT's these were two 28 Pin DIP IC's, but as you can imagine, things have changed dramatically since then. While the operational principal is still the same, the PIC is now integrated somewhere into your chipset, along with many other devices.



The basic block diagram of the PIC is shown above. The 8 individual interrupt request lines are first passed through the Interrupt Mask Register (IMR) to see if they have been masked or not. If they are masked, then the request isn't processed any further. However if they are not masked, they will register their request with the Interrupt Request Register (IRR).

The Interrupt Request Register will hold all the requested IRQ's until they have been dealt with appropriately. If required, this register can be read by setting certain bits of the Operation Control Word 3. The Priority Resolver simply selects the IRQ of highest priority. The higher priority interrupts are the lower numbered ones. For Example IRQ 0 has the highest priority followed by IRQ 1 etc.

Now that the PIC has determined which IRQ to process, it is now time to tell the processor, so that it can call your ISR for you. This process is done by sending a INT to the processor, i.e. the INT line on the processor is asserted. The processor will then finish the current instruction it's processing and acknowledge your INT request with a INTA (Interrupt Acknowledge) pulse.

Upon receiving the processor's INTA, the IRQ which the PIC is processing at the time is stored in the In Service Register (ISR) which as the name suggests, shows which IRQ is currently in service. The IRQ's bit is also reset in the Interrupt Request Register, as it is no longer requesting service but actually getting service.

Another INTA pulse will be sent by the processor, to tell the PIC to place a 8 bit pointer on the data bus, corresponding to the IRQ number. If an IRQ serviced by PIC2 is requesting the service, then PIC2 will send the pointer to the processor. The Master (PIC1) at this stage, will select PIC2 to send the pointer, by placing PIC2's Slave ID on the Cascade lines, which is a 3 wire bus between all the PIC's in a system.

The 5 most significant bits of this pointer is set using the Initialization Command Word 2 (ICW2). This will be 00001 for PIC1 and 01110 for PIC2. The three least significant bits, will be sent due to which IRQ is being serviced. For example, if IRQ3 is requesting service then the 8 bit pointer will be made up with 00001 for the 5 most significant bits and 011 (IR3) for the least significant bits. Put this together and you get 00001011 or 0x0B which just happens to be IRQ3's interrupt vector.

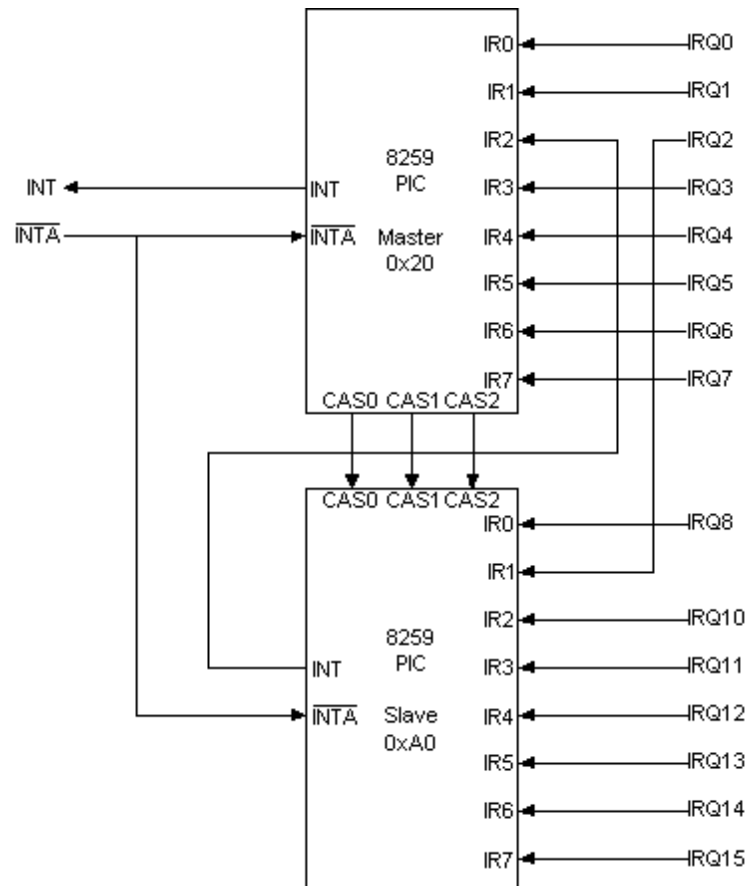
For PIC2, the same principal is applied. If IRQ10 is requesting service, then 01110010 will be sent, which just happens to represent Interrupt 72h. IRQ10 happens to be connected to IR2 on the Second PIC, thus 010 is used as the least significant bits.

Once your ISR has done everything it needs, it sends an End of Interrupt (EOI) to the PIC, which resets the In-Service Register. If the request came from PIC2, then EOI's are required to be sent to both PICs. The PIC will then determine the next highest priority interrupt and repeat the same process. If no Interrupt Requests are present, then the PIC waits for the next request before interrupting the processor.



## IRQ2/IRQ9 Redirection

The redirection of IRQ2 causes quite some confusion, and thus is discussed here. In the original XT's there were only one PIC, thus only eight IRQ's. However users soon out grew these resources, thus an additional 7 IRQ's were added to the PC. This involved attaching another PIC to the existing one already in the XT. Compatibility always causes problems as the new configuration still had to be compatible with old hardware and software. The "new" configuration is shown below.



The CPU only has one interrupt line, thus the second controller had to be connected to the first controller, in a master/slave configuration. IRQ2 was selected for this. By using IRQ2 for the second controller, no other devices could use IRQ2, so what happened to all these devices using IRQ2? Nothing, the interrupt request line found on the bus, was simply diverted into the IRQ 9 input. As no devices yet used the second PIC or IRQ9, this could be done.

The next problem was that a hardware device using IRQ2 would install it's ISR at INT 0x0A. Therefore an ISR routine was used at INT 71h, which sent a EOI to PIC2 and then called the ISR at INT 0x0A. If you dis-assemble the ISR for IRQ9, it will go a little like,

```
MOV AL,20
OUT A0,AL    ; Send EOI to PIC2
INT 0A      ; Call ISR for IRQ2
IRET
```

The routine only has to send a EOI to PIC2, as it is expected that a ISR routine written for IRQ2 will send a EOI to PIC1. This example destroys the contents of Register AL, thus this must be placed on the stack first (Not shown in example). As PIC2 is initialized with a Slave on IRQ2, any request using PIC2 will not call the ISR routine for IRQ2. The 8 bit pointer will come from PIC2.

## *Programmable Interrupt Controller's Addresses*

---

The two PIC's found in an IBM compatible system are initialized via BIOS thus you don't have to worry about all of their registers. However for some people, who have inquisitive minds the following information may come in some use or maybe you want to (re)program a BIOS? Below is a table of all the command words of the 8259 and compatible Programmable Interrupt Controller. The Top Table shows the Addresses for the PIC1, while the bottom table shows addresses for PIC2.

Address	Read/Write	Function
20h	Write	Initialization Command Word 1 (ICW1)
	Write	Operation Command Word 2 (OCW2)
	Write	Operation Command Word 3 (OCW3)
	Read	Interrupt Request Register (IRR)
	Read	In-Service Register (ISR)
21h	Write	Initialization Command Word 2 (ICW2)
	Write	Initialization Command Word 3 (ICW3)
	Write	Initialization Command Word 4 (ICW4)
	Read/Write	Interrupt Mask Register (IMR)

Table 4 : Addresses/Registers for PIC1

PIC2 Addresses . . .

Address	Read/Write	Function
A0h	Write	Initialization Command Word 1 (ICW1)
	Write	Operation Command Word 2 (OCW2)
	Write	Operation Command Word 3 (OCW3)
	Read	Interrupt Request Register (IRR)
	Read	In-Service Register (ISR)
A1h	Write	Initialization Command Word 2 (ICW2)
	Write	Initialization Command Word 3 (ICW3)
	Write	Initialization Command Word 4 (ICW4)
	Read/Write	Interrupt Mask Register (IMR)

Table 5 : Addresses/Registers for PIC2

## *Initialization Command Word 1 (ICW1)*

---

If the PIC has been reset, it must be initialized with 2 to 4 Initialization Command Words (ICW) before it will accept and process Interrupt Requests. The following selection outlines the four possible Initialization Command Words.

Bit(s)	Function	
7: 5	Interrupt Vector Addresses for MCS-80/85 Mode.	
4	Must be set to 1 for ICW1	
3	1	Level Triggered Interrupts
	0	Edge Triggered Interrupts
2	1	Call Address Interval of 4
	0	Call Address Interval of 8
1	1	Single PIC
	0	Cascaded PICs
0	1	Will be Sending ICW4
	0	Don't need ICW4

Table 6 : Initialization Command Word 1 (ICW1)

The 8259 Programmable Interrupt Controller, offers many other features which are not used in the PC. It also offers support for MCS-80/85 microprocessors. All we have to be aware of being PC uses, is if the system is running in single mode (One PIC) or if in Cascaded Mode (More than one PIC) and if the Initialization Command Word 4 is needed. If no ICW4 is used, then all of it's bits will be set to 0. As we are using it in 8086 mode, we must send a ICW4.

## *Initialization Command Word 2 (ICW2)*

---

Bit	8086/8080 Mode	MCS 80/85 Mode
7	I7	A15
6	I6	A14
5	I5	A13
4	I4	A12
3	I3	A11
2	-	A10
1	-	A9
0	-	A8

Table 7 : Initialization Command Word 2 (ICW2)

Initialization Command Word 2 (ICW2) selects which vector information is released onto the bus, during the 2nd INTA Pulse. Using the 8086 mode, only bits 7:3 need to be used. This will be 00001000 (0x08) for PIC1 and 01110000 (0x70) for PIC2. If you wish to relocate the IRQ Vector Table, then you can use this register.

## *Initialization Command Word 3 (ICW3)*

---

There are two different Initialization Command Word 3's. One is used, if the PIC is a master, while the other is used for slaves. The top table shows the ICW3 for the master.

Bit	Function
7	IR7 is connected to a Slave
6	IR6 is connected to a Slave
5	IR5 is connected to a Slave
4	IR4 is connected to a Slave
3	IR3 is connected to a Slave
2	IR2 is connected to a Slave
1	IR1 is connected to a Slave
0	IR0 is connected to a Slave

Table 8 : Initialization Command Word 3 for Master PIC (ICW3)

And for the slave device, the ICW3 below is used.

Bit(s)	Function	
7	Reserved. Set to 0	
6	Reserved. Set to 0	
5	Reserved. Set to 0	
4	Reserved. Set to 0	
3	Reserved. Set to 0	
2 : 0	<i>Slave ID</i>	
	000	Slave 0
	001	Slave 1
	010	Slave 2
	011	Slave 3
	100	Slave 4
	101	Slave 5
	110	Slave 6
	111	Slave 7

Table 9 : Initialization Command Word 3 for Slaves (ICW3)

## *Initialization Command Word 4 (ICW4)*

---

Bit(s)	Function	
7	Reserved. Set to 0	
6	Reserved. Set to 0	
5	Reserved. Set to 0	
4	1	Special Fully Nested Mode
	0	Not Special Fully Nested Mode
3 : 2	0x	Non - Buffered Mode
	10	Buffered Mode - Slave
	11	Buffered Mode - Master
1	1	Auto EOI
	0	Normal EOI
0	1	8086/8080 Mode
	0	MCS-80/85

Table 10 : Initialization Command Word 4 (ICW4)

Once again, many of these are special functions not used with the 8259 PIC in a PC. We don't use, Special Fully Nested Mode, thus this bit is set to 0. Likewise we use non-buffered mode and Normal EOI's thus all these corresponding bits are set to 0. The only thing we must set is 8086/8080 Mode which is done using Bit 0.

## *Operation Control Word 1 (OCW1)*

---

Once all the required Initialization Command Words have been sent to the PIC, then you can send Operation Control Words, in any order and at any time during the PIC's operation. The Operation Control Words are shown in the next sections.

Bit	PIC 2	PIC 1
7	Mask IRQ15	Mask IRQ7
6	Mask IRQ14	Mask IRQ6
5	Mask IRQ13	Mask IRQ5
4	Mask IRQ12	Mask IRQ4
3	Mask IRQ11	Mask IRQ3
2	Mask IRQ10	Mask IRQ2
1	Mask IRQ9	Mask IRQ1
0	Mask IRQ8	Mask IRQ0

Table 11 : Operation Control Word 1 (OCW1)

Operation Control Word 1, shown above is used to mask the inputs of the PIC. This has already been discussed, earlier in this article.

## Operation Control Word 2 (OCW2)

---

Bit(s)	Function	
7:5	000	Rotate in Auto EOI Mode (Clear)
	001	Non Specific EOI
	010	Reserved
	011	Specific EOI
	100	Rotate in Auto EOI Mode (Set)
	101	Rotate on Non-Specific EOI
	110	Set Priority Command (Use Bits 2:0)
	111	Rotate on Specific EOI (Use Bits 2:0)
4	Must be set to 0	
3	Must be set to 0	
2 : 0	000	Act on IRQ 0 or 8
	001	Act on IRQ 1 or 9
	010	Act on IRQ 2 or 10
	011	Act on IRQ 3 or 11
	100	Act on IRQ 4 or 12
	101	Act on IRQ 5 or 13
	110	Act on IRQ 6 or 14
	111	Act on IRQ 7 or 15

Table 12 : Operation Control Word 2 (OCW2)

Operation Control Word 2 selects how the End of Interrupt (EOI) procedure works. The only thing of interest to us in this register is the non-specific EOI command, which we must send at the end of our ISR's.



## Operation Control Word 3 (OCW3)

---

Bit(s)	Function	
7	Must be set to 0	
6 : 5	00	Reserved
	01	Reserved
	10	Reset Special Mask
	11	Set Special Mask
4	Must be set to 0	
3	Must be set to 1	
2	1	Poll Command
	0	No Poll Command
1 : 0	00	Reserved
	01	Reserved
	10	Next Read Returns Interrupt Request Register
	11	Next Read Returns In-Service Register

Table 13 : Operation Control Word 3 (OCW3)

Bits 0 and 1 are of the most significant to us, in Operation Control Word 3. These two bits enable us to read the status of the Interrupt Request Register (IRR) and the In-Service Register (ISR). This is done by setting the appropriate bits correctly as above, and reading the register at the Base Address.

For example if we wanted to read the In-Service Register (ISR), then we would set both bits 1 and 0 to 1. The next read to the base register, (0x20 for PIC1 or 0xA0 for PIC2) will return the status of the In-Service Register.

Craig Peacock's Interfacing the PC

---

<http://www.senet.com.au/~cpeacock>

<http://www.geocities.com/SiliconValley/Bay/8302/>

Copyright January 1998 Craig Peacock.

---

Any errors, ideas, criticisms or problems, please contact the author at [cpeacock@senet.com.au](mailto:cpeacock@senet.com.au)