Interrupciones en el Simulador MSX88

Cátedra de Arquitectura de Computadoras,
Facultad de Informática, Universidad Nacional de La Plata

4 de enero de 2017

Índice

1. Introducción

En este apunte se introducen los conceptos básicos para escribir programas que utilizan interrupciones por hardware o software en el simulador MSX88.

1.1. Interrupciones: una solución para dos problemas

Las interrupciones son un mecanismo para pausar brevemente la ejecución de un programa, ejecutar una subrutina especial, y luego continuar ejecutando el programa.

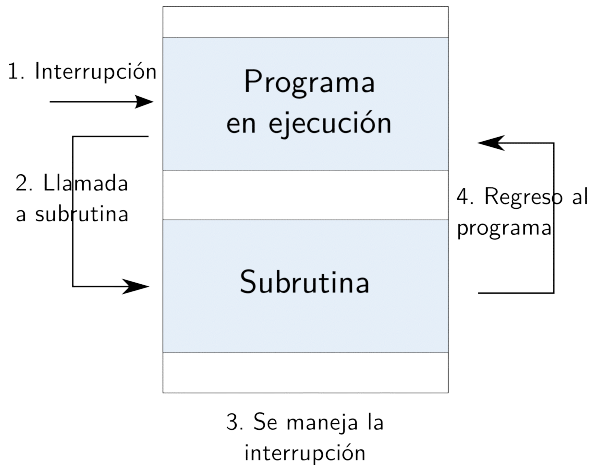

Figura 1: Etapas generales de una interrupción. Diagrama simplificado.

Las mismas sirven como solución a dos problemas distintos, que se presentan a continuación.

1.2. Problema 1: Dispositivos lentos → interrupciones por hardware

Para interactuar con una computadora, necesitamos algunos dispositivos de entrada/salida (ES), como teclados, mouses, monitores, impresoras, touchpads, etc. Los programas necesitan comunicarse con estos dispositivos para enviar o recibir información, para lo cual la CPU generalmente tiene instrucciones especiales de ES.

El problema es que, en teoría, una CPU de 1 GHz puede realizar 1 000 000 000 operaciones por segundo. En ese segundo (y también en teoría):

- Un disco rígido tradicional puede realizar alrededor de 100 operaciones
- Un disco de estado sólido (SSD) realiza de 10 000 a 50 000 operaciones
- Una placa de red gigabit recibe hasta 6 000 000 paquetes
- Una impresora láser imprime hasta 50 caracteres
- Un monitor LCD cambia su imagen 60 veces
- Una persona escribe hasta 5 letras en un teclado

Si bien estos números pueden variar en la práctica y en casos concretos, vemos una tendencia importante: la CPU es significativamente más rápida que cualquier otro dispositivo de ES.

Además, los dispositivos de ES también son inconstantes, es decir, es difícil predecir cuando van a estar disponibles o cuánto van a tardar en realizar una tarea. Por ejemplo, no podemos saber cuándo una persona va a presionar una tecla.

Si la CPU tiene que quedarse sin hacer nada mientras le pide a un dispositivo que realice una tarea o mientras espera a que esté disponible, el rendimiento de la computadora baja significativamente. Por ende, necesitamos un mecanismo para que los dispositivos le avisen a la CPU que requieren su atención, de modo que la CPU pueda ejecutar los programas sin tener que continuamente verificar el estado de los dispositivos. Es decir, necesitamos una forma en que los dispositivos puedan *interrumpir* el funcionamiento normal de la CPU para ser *atendidos*.

Como estas interrupciones vienen desde los dispositivos (hardware!), decimos que necesitamos un mecanismo de **interrupciones por hardware** para atender sus solicitudes. ¿Qué significa *atender* una interrupción de un dispositivo? Simplemente ejecutar una subrutina, con instrucciones que dependerán del tipo de dispositivo. Estas subrutinas se conocen como **manejadores de interrupciones** (*interrupt handlers*, en inglés).

Por ejemplo, si una impresora avisa que está lista para imprimir un nuevo carácter mediante una interrupción, la CPU pasará el control al manejador de interrupciones correspondiente. El manejador de interrupciones es simplemente una subrutina con instrucciones para, por ejemplo, enviar un nuevo carácter a la impresora.

La ventaja de este esquema es que la CPU nunca está ociosa esperando a un dispositivo, sino que estos mismos le avisan a la CPU cuando requieren atención.

En el simulador MSX88, el programador debe escribir los manejadores de interrupciones. O sea, tiene que escribir subrutinas que manejen las interrupciones generadas por los dispositivos. Estas subrutinas se ubican en direcciones de memoria como cualquier otra subrutina. Ahora bien, sería muy inconveniente que los dispositivos tengan que conocer de antemano la dirección de memoria de las subrutinas que los atienden. Por ende, necesitamos un mecanismo que nos permita hacer esta asociación.

Para ello, las interrupciones se identifican por un número, entre 0 y 255, llamado *identificador de interrupción*. El mecanismo de interrupciones debe permitir hacer entonces una asociación entre los identificadores de interrupción y las subrutinas (en realidad, entre los identificadores y las *direcciones* de comienzo de las subrutinas). Dicho mecanismo se describe con más detalle en la subsección 1.4, y se termina de explicar para las interrupciones por hardware en la sección 3, donde se introduce el **Controlador de Interrupciones Programable** (PIC, por Programmable Interrupt Controller).

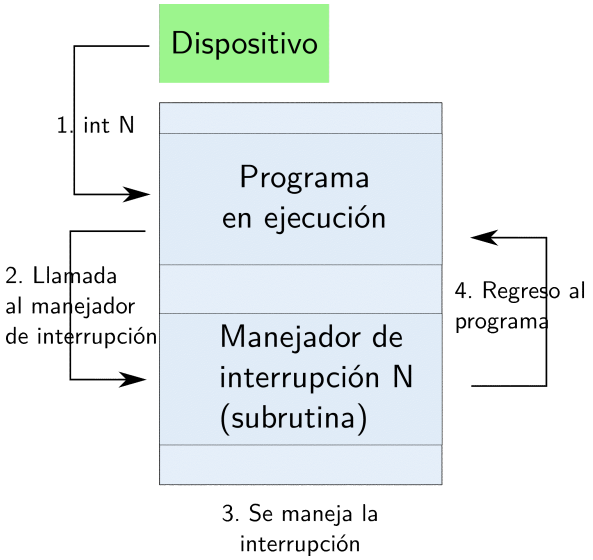

Figura 2: Etapas de una interrupción por hardware.

1.3. Problema 2: Llamadas al sistema operativo → interrupciones por software

Los sistemas operativos (SOs) centralizan el manejo de algunos de los recursos de la computadora, haciendo de intermediarios entre los programas y los recursos. Por ejemplo, si un programa quiere mostrar un texto en el monitor debe pedirle al SO que lo muestre. De esta forma, si varios programas quieren modificar el contenido de la pantalla, el SO puede mediar para que lo hagan de manera ordenada y consistente de modo que la imagen final que se muestra tenga sentido.

Por ende, los programas necesitan comunicarse con el sistema operativo. Para ello, es necesario realizar una **llamada de sistema** (o *syscall*), en donde se interrumpa la ejecución del programa y se pase temporariamente el control a una **subrutina del SO**. El SO satisface el pedido del programa (ejecutando las instrucciones de la subrutina del SO), y luego continúa con la ejecución del mismo.

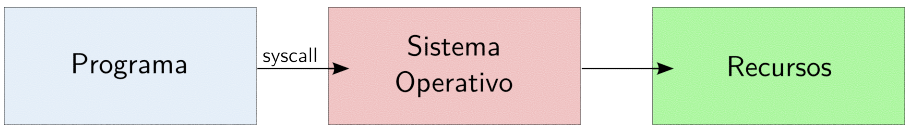

Figura 3: El SO administra los dispositivos de ES y otros recursos del sistema. Por ende, el los programas no interactuan directamente con los mismos, sino que utilizan al SO como intermediario.

El mecanismo para realizar una syscall (que resultaría en la ejecución de una subrutina del SO) es muy similar a llamar a una subrutina o procedimiento; debemos conocer la dirección de memoria donde se encuentra la misma, y poner en el registro IP dicha dirección (y apilar la dirección de retorno, para que cuando termine la subrutina del SO se pueda retornar la ejecución al punto correcto del programa!).

Sin embargo, si aplicamos el mismo esquema que funciona para las subrutinas normales para las syscalls, el programa necesitaría conocer las direcciones de las subrutinas del SO. Esto no es deseable, debido a que le quita flexibilidad al SO para ubicar sus subrutinas de sistema, lo cual podría cambiar entre versiones del SO o entre tipos de SO [1].

Por esa razón, las subrutinas del SO se suelen llamar utilizando **interrupciones por software**. Las interrupciones por software, al igual que las de hardware, se manejan identificando una subrutina mediante un número, sin tener que conocer su dirección de memoria. La diferencia con las interrupciones de hardware es que

[1] En realidad, en los SO actuales implementan muchas syscalls con llamadas a subrutinas normales, debido a que el mecanismo de interrupciones por software es más lento que llamar a una subrutina. No obstante, en el simulador MSX88 todas las syscalls se realizan mediante interrupciones por software.

es el programa quien inicia la interrupción para solicitar un servicio, mediante la instrucción `int N`. De esta manera, el programa se interrumpe a si mismo para darle el control de la CPU al SO, quién se encargará de ejecutar ese servicio y luego retornar al programa.

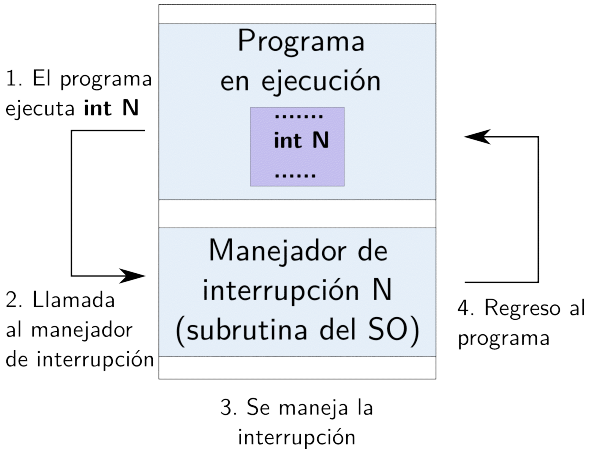

Figura 4: Etapas de una interrupción por software.

1.4. Mecanismo de interrupción: el vector de interrupciones

Hemos visto que existen dos tipos de interrupciones, por hardware y por software, que son necesarios por razones distintas. En las interrupciones por hardware, el que interrumpe es un dispositivo y la CPU pasa el control a una subrutina especial llamada manejador de interrupciones para atenderlo. En las que son por software, el que interrumpe es el mismo programa, y la CPU pasa el control al sistema operativo que lo atiende ejecutando una subrutina de sistema.

Cuadro 1: Tipos de interrupción y propiedades

Tipo de interrupción	Inicia	Atiende
Software	El programa mediante la instrucción `int N`	Una subrutina del sistema operativo (código oculto)
Hardware	El dispositivo a través del PIC	Una subrutina del programa (debe implementarla el programador)

No obstante, ambos tipos de interrupciones tienen en común que la interrupción se identifica con un número, el identificador de interrupción, y se requiere ejecutar una subrutina asociada a ese número.

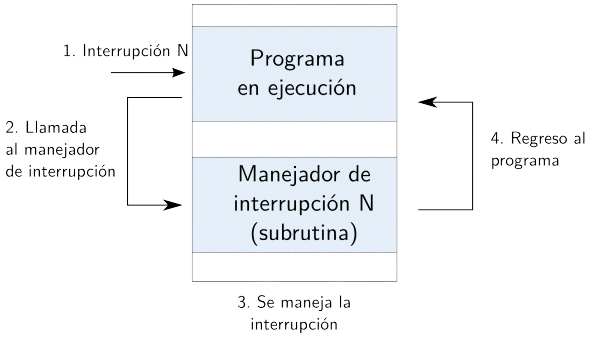

Figura 5: Etapas del mecanismo de interrupción, incluyendo el ID de la interrupción.

¿Cómo se logra esto? Con un vector de 255 direcciones de memoria, una por cada posible subrutina, llamado **vector de interrupciones**. Al recibir una interrupción con identificador N (en adelante ID N), la CPU accede a la posición N de dicho vector, obtiene la dirección de comienzo de la subrutina, y ejecuta un `call` a la misma.

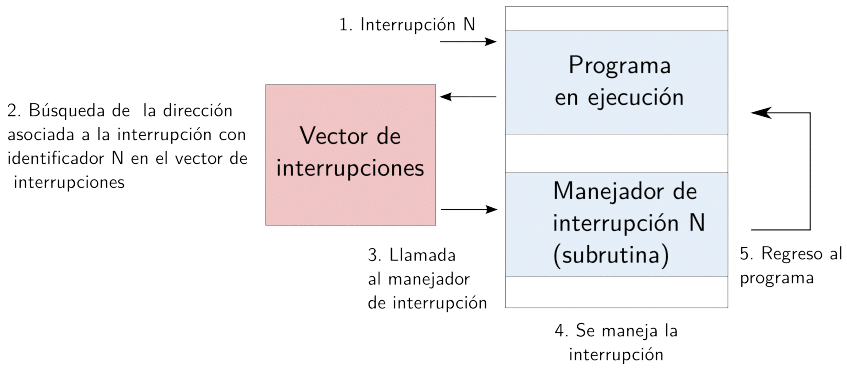

Figura 6: Etapas del mecanismo de interrupción, incluyendo el vector de interrupciones.

¿Qué contiene el vector de interrupciones? En cada elemento del vector de interrupciones se debe guardar una dirección. Esta dirección es la de comienzo del manejador de la interrupción del identificador correspondiente a dicho elemento.

Como en el simulador MSX88 las direcciones son de 2 bytes, *en teoría* todo el vector debería ocupar $256 * 2\,\text{bytes} = 512\,\text{bytes}$. No obstante, por razones históricas, en el MSX88 cada elemento ocupa 4 bytes: los primeros 2 tienen la dirección, y los otros 2 no se usan. Entonces, el tamaño total del vector es de $256 * 4\,\text{bytes} = 1024\,\text{bytes}$ o 1 kB [2].

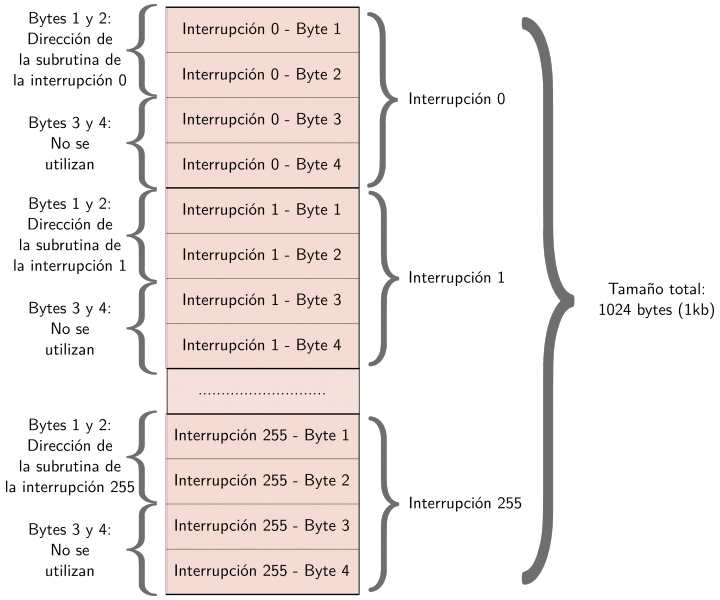

Figura 7: Elementos del vector de interrupciones. Cada elemento ocupa 4 bytes. En esos 4 bytes, los primeros 2 se utilizan para almacenar la dirección de la subrutina asociada. Los otros 2 bytes no se utilizan. En total, el vector ocupa 1024 bytes, ya que hay 256 elementos de 4 bytes.

¿Dónde se encuentra físicamente el vector de interrupciones? Como la pila, el vector de interrupciones es simplemente un área de la memoria principal. En este caso, el vector de interrupciones ocupa los primeros $256 * 4\,\text{bytes} = 1024\,\text{bytes}$ de la memoria.

Por ende, el vector ocupa los bytes cuyas direcciones van del 0 al 1023 (en decimal), lo que corresponde al rango de direcciones entre `0000h` y `03FFh` (`0400h` es 1024 en decimal). A modo de ejemplo el cuadro 2 lista las direcciones de algunos elementos del vector.

[2] En la actualidad este modelo ha cambiado y de hecho si bien sigue habiendo 256 interrupciones, cada elemento del vector ocupa 8 bytes o 64 bits, el tamaño de una dirección en un procesador moderno de 64 bits, y el tamaño total del vector es de 2048 bytes o 2 kB

Cuadro 2: Direcciones de los elementos del vector de interrupciones

Identificador de Interrupción	Dirección de comienzo (decimal)	Dirección de comienzo (hexa)
0	0	`0000h`
1	4	`0004h`
2	8	`0008h`
3	12	`000Ch`
4	16	`0010h`
...	...	...
254	1016	`03F8h`
255	1020	`03FCh`

Entonces, por ejemplo, si el procesador es interrumpido por la interrupción con ID 9, deberá ir a buscar el elemento 9 del vector de interrupciones que está en la dirección de memoria $9 * 4 = 36$, o `24h`. En general, el elemento N del vector de interrupciones se encontrará en la dirección $4 * N$.

Finalmente, si el vector de interrupciones ocupa la primera parte de la memoria, deberíamos actualizar nuestra visión de la misma, que queda entonces así:

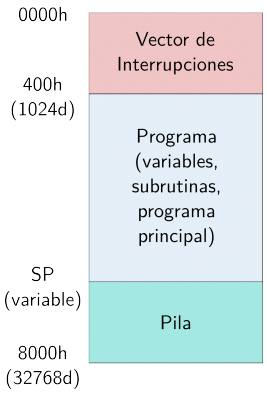

Figura 8: Áreas de la memoria principal.

El vector de interrupciones es necesario tanto para el *funcionamiento* de ambos tipos de interrupciones. No obstante, no será necesario considerarlo a la hora de *programar* con interrupciones por software en el simulador MSX88, y por ende no haremos mención al mismo en la sección 2. Volveremos a ver el vector de interrupciones al utilizar interrupciones por hardware en la sección 3, donde será un elemento esencial para programar.

1.5. Resumen

Las interrupciones son un mecanismo para pausar temporariamente la ejecución del programa actual, atender algún evento, y luego volver al programa. Hay 2 tipos:

- Por hardware: Sirven para atender eventos de los dispositivos. Los dispara el dispositivo, y causan que se ejecute una subrutina del programa (debe implementarla el programador).
- Por software: Sirven para hacer llamadas al sistema. Las inicia el programa mediante la instrucción `int N`, y hacen que se ejecute una subrutina del sistema operativo (cuyo código está oculto al programador).

En ambos casos, el mecanismo mediante el cual la CPU produce la ejecución de una subrutina en base al identificador de interrupción es el mismo: se busca en el vector de interrupciones la dirección de la subrutina. El vector de interrupciones ocupa los primeros 1024 bytes de la memoria, y cada elemento del mismo ocupa 4 bytes. Por ende, la dirección de la subrutina correspondiente a la interrupción con identificador N se encuentra en la dirección $4 * N$ de la memoria.

2. Interrupciones por software

En la sección anterior establecimos que las interrupciones por software las genera el programa, y causan que se ejecute una subrutina del sistema operativo. Entonces, podemos pensarlas simplemente como subrutinas que se llaman de una manera distinta.

Hay 4 tipos de interrupción por software en el simulador MSX88:

- Interrupción 0: Terminar el programa.
- Interrupción 3: Poner un punto de parada (*breakpoint*).
- Interrupción 6: Lee un carácter de teclado.
- Interrupción 7: Escribe un string en pantalla.

Las interrupciones por software se generan con la instrucción `int N`, donde `N` es el número de interrupción. Entonces `int 0` invoca a la interrupción 0, `int 6` la interrupción 6, etc.

La interrupción por software con ID 0 sirve para que la CPU deje de ejecutar el programa. Por ende, ejecutar `int 0` es equivalente a ejecutar `hlt`, sólo que involucra al SO, el cual puede así terminar la ejecución del programa de forma segura. De ahora en adelante, utilizaremos `int 0` en lugar de `hlt` para terminar el programa.

La instrucción `int 3` establece un punto de parada o breakpoint, de forma de que cuando se ejecuta esa instrucción se inicia el modo de **debug** o inspección del programa (pero no es necesaria para programar).

Las interrupciones más importantes que vamos a utilizar son las que restan: `int 6` e `int 7`[3]. La mejor manera de pensar estas interrupciones es como si fueran los procedimientos `read()` y `write()` del lenguaje de programación Pascal.

2.1. int 6: Leer un carácter

La instrucción `int 6` sólo lee de a un carácter por vez, y sólo lee caracteres (no números). Como parámetros, recibe en el registro `bx` la dirección de memoria donde se va a guardar el carácter que se lee. Para leer un carácter y guardarlo en una variable llamada `letra`, podemos escribir el siguiente programa:

[3]En un sistema operativo real hay varias interrupciones más que en el simulador, pero cómo el mismo tiene como objetivo el aprendizaje sólo implementa estas 4.

Figura 9: Lectura de un carácter en la variable `letra` utilizando `int 6`. Equivale a ejecutar `read(letra)` en el lenguaje Pascal.

```
org 1000H
letra db ?

org 2000h
; poner la direccion de 'letra' en bx
mov bx,offset letra
; leer el caracter
int 6
; el caracter leido ya esta en 'letra'
int 0
end
```

2.2. int 7: Escribir un string

La `int 7` escribe un string o cadena de caracteres en la pantalla. Como parámetros, recibe en el registro `bx` la dirección de memoria donde comienza el string, y la cantidad de caracteres del string en el registro `al`. La impresión es, a diferencia de la lectura, de varios caracteres con una sola instrucción. Para escribir el string `"Hola"`, declarado con la etiqueta `mensaje`, podemos escribir el siguiente programa:

Figura 10: Escritura del string `"Hola"` en la pantalla utilizando `int 7`. Equivale a hacer `write(mensaje)` en el lenguaje Pascal.

```
org 1000H
mensaje db "Hola"

org 2000h
mov bx,offset mensaje
mov al,4 ; porque el string tiene 4 caracteres
int 7
int 0
end
```

2.3. Lectura de strings

La interrupción 6 lee de a un carácter. ¿Qué sucede si necesitamos leer un string? Tenemos que llamar varias veces a `int 6`, tantas como caracteres tenga el string. El programa de la figura 11 lee un string de 10 caracteres y lo vuelve a mostrar por pantalla. Para ello, implementa una subrutina que nos permite leer strings. La subrutina recibe el tamaño del string a leer por parámetro en el registro `al`, y un puntero a la dirección donde se deben almacenar los caracteres leídos en el registro `bx`.

Figura 11: Lectura de un string utilizando `int 6` de forma repetida.

```
org 1000H
longitud db 10
mensaje db ?

org 3000h
; parametros:
```

```
; bx: direccion de memoria donde guardar el string
; al: longitud del string a leer
leer_string: cmp al,0
             jz fin  ; no hacer nada si piden leer un string de longitud 0
             push bx;
             push ax; preserva bx y ax (y al)

      loop:  int 6 ; leer los caracteres
             inc bx; apuntar al siguiente byte con bx
             dec al
             jnz loop; hasta que al quede en 0

             pop ax
             pop bx; restaura bx y ax (y al)
      fin:   ret

org 2000h; programa principal

mov bx,offset mensaje
mov al,longitud
call leer_string; lee el string de longitud al y lo guarda a partir de bx
; Como justo tenemos en bx la direccion y
; en al su longitud por haber llamado a leer_string
; directamente llamamos a int 7 para imprimirlo.
int 7
int 0
end
```

Es importante notar que el programa en ningún momento *reserva* memoria para el string; simplemente guardamos los caracteres a partir de la direcciòn `1001H`, hasta la `100AH` inclusive, sobreescribiendo lo que sea que haya en esas posiciones de memoria. Por eso es que primero se declaro la etiqueta `longitud`, y luego `mensaje`.

2.4. Representación de caracteres: el código ASCII

El código ASCII es una forma de representar símbolos con números.

Si bien nosotros escribimos caracteres como `"a"`, `"B"`, `"3"`, `"?"` o `"\#"`, en la memoria se almacena su código ASCII, codificado en BSS. Esto sucede tanto cuando definimos una variable como cuando leemos de teclado.

Consideremos este programa simple, que solo declara una etiqueta:

```
org 1000H
mensaje db "Hola"
end
```

Cuando compila el programa, el simulador lee las declaraciones de las etiquetas que se ubican a partir de la dirección de memoria `1000h`, y examina cada declaración. Para la variable `mensaje`, reserva 4 bytes, uno por cada carácter. Se obtiene el código ASCII de cada carácter, y esos códigos se guardan e n la memoria. Entonces, en las celdas de memoria con direcciones `1000h`, `1001h`, `1002h` y `1003h` el simulador pone los valores `72`, `111`, `108` y `97` (en hexadecimal serían `48h`, `6Fh`, `6Ch` y `61h`), correspondientes a los caracteres `"H"`, `"o"`, `"l"` y `"a"`.

Luego de esta operación no hay ningún registro de que esos valores correspondían a caracteres. Es decir, el programa anterior hace exactamente lo mismo que el siguiente:

```
org 1000H
mensaje db 72, 111, 108, 97
end
```

Algo parecido sucede cuando leemos de teclado. Volvamos a considerar el programa que lee de teclado un carácter:

Figura 12: Lectura de un carácter en la variable `letra` utilizando `int 6`. El programa es idéntico al de la figura 9.

```
org 1000H
letra db ?

org 2000h
mov bx,offset letra
int 6
int 0
end
```

Es importante entender que luego de `int 6`, si el usuario ingresa la letra `"H"`, en la variable `letra` quedará guardado el valor `72`. Ahora, si el usuario ingresa el carácter `"3"`, en letra queda guardado el valor `52` que es el código ASCII de dicho dígito.

Entonces, ¿cómo podemos hacer para leer un dígito y que quede guardado su valor numérico, no su código ASCII? ¿Y si queremos leer un número como `12` o `513`?

2.5. Lectura de dígitos

Para leer dígitos decimales (números entre 0 y 9) y obtener su valor numérico, simplemente debemos restarle al número el código ASCII del dígito `"0"`, o sea, restarle `48` (o `30h`, en hexadecimal). ¿Por qué funciona esto? Porque en el código ASCII los dígitos están ordenados consecutivamente a partir del `48` (`30h`) como se muestra en el cuadro 3:

Cuadro 3: Código ASCII de los caracteres que representan a los dígitos `"0"` a `"9"`. Dichos caracteres tienen códigos del 48 al 58 (en decimal) o del `30h` al `39h` (en hexadecimal).

Carácter	Código ASCII (en decimal)	Código ASCII (en hexa)	Obtención del valor numérico
`"0"`	48	`30h`	$48 - 48 = 0$
`"1"`	49	`31h`	$49 - 48 = 1$
`"2"`	50	`32h`	$50 - 48 = 2$
`"3"`	51	`33h`	$51 - 48 = 3$
`"4"`	52	`34h`	$52 - 48 = 4$
`"5"`	53	`35h`	$53 - 48 = 5$
`"6"`	54	`36h`	$54 - 48 = 6$
`"7"`	55	`37h`	$55 - 48 = 7$
`"8"`	56	`38h`	$56 - 48 = 8$
`"9"`	57	`39h`	$57 - 48 = 9$

En la figura 13 mostramos un programa que lee un caracter y, asumiendo que usuario ingresa un dígito, le resta 48 (el código ASCII de `"0"`) para que el valor almacenado finalmente en `letra` sea el que representa y

no su código ASCII. [4]

Figura 13: Lectura de un carácter en la variable `letra` utilizando `int 6`. Se asume que el carácter leído es un dígito. Al carácter se le resta el código ASCII del `"0"` para obtener el valor numérico correspondiente.

```
org 1000H
letra db ?

org 2000h
mov bx,offset letra
int 6
sub letra,48; 48 es el codigo ASCII del "0"
; seria lo mismo escribir
; sub letra,30h
; ya que 30h es 48 en decimal
int 0
end
```

Si bien esto nos permite leer dígitos individuales, la lectura (y escritura) de números de *varios dígitos* requiere un poco más de trabajo. Para el lector interesado, en el anexo 4 hay un programa de ejemplo que realiza esta tarea.

2.6. Ejemplo combinado

Para cerrar estas ideas, vamos a resolver un ejemplo que combina lecturas y escrituras en un mismo programa. El objetivo del mismo será leer caracteres y volver a imprimirlos inmediatamente. Para los dígitos `"0"` al `"9"`, en lugar de imprimirlos se debe mostrar el carácter `"#"`. El programa termina cuando se ingresa el carácter `"z"`. La figura 14 muestra una implementación de la solución.

Figura 14: Leer e imprimir caracteres, reemplazando los dígitos por `"#"`, hasta leer el carácter `"z"`.

```
org 1000H
c_leer db ?
c_mostrar db ?
zeta db "z" ;definimos una variable con el codigo de la "z"
numeral db "#" ;definimos una variable con el codigo del "#"
org 2000h
          mov al,1; siempre imprimimos de a un caracter
          ; leer un caracter
    loop: mov bx,offset c_leer
          int 6
          mov ah,zeta; ponemos el ASCII de "z" en ah para poder usar el cmp
          cmp c_leer,ah
          jz fin
          ; en principio el caracter a mostrar
          ; es el mismo que el leido
          mov cl,c_leer
          mov c_mostrar,cl
          ; Si el codigo es menor que "0" o mayor que "9"
          ; imprimir lo que se leyo
          cmp c_leer,48
          js imprimir
          ; ponemos 57, el codigo del "9" en ah porque
          ; cmp no soporta que su primer operando sea inmediato
          mov ah,57
```

[4] El código ASCII de los dígitos puede recordarse considerando que el `"0"` vale 48 en decimal, o quizás es más fácil considerar que el `"0"` vale `30h` en hexadecimal, y por ende el `"1"` vale `31h`, y así sucesivamente hasta el `"0"` que vale `"39h"`.

```
25          cmp ah,c_leer
26          js imprimir
27          ; si el codigo era de un digito
28          ; cambiar el caracter a mostrar por "#"
29          mov ah,numeral
30          mov c_mostrar,ah
31  imprimir: mov bx,offset c_mostrar
32          int 7
33          jmp loop
34      fin: int 0
35          end
```

2.7. Resumen

Las interrupciones por software son una forma de llamar subrutinas del sistema operativo, y se invocan mediante la instrucción `int N`. En el simulador MSX88 hay 4 interrupciones por software, y las más útiles son:

- `int 0` para finalizar un programa (equivalente a `hlt`).
- `int 6` lee un solo carácter, y lo guarda en la dirección almacenada en `bx`.
- `int 7` imprime un string o vector de caracteres, cuya dirección de comienzo se especifica en `bx`, y la cantidad de caracteres a partir de esa dirección debe especificarse en `al`.

3. Interrupciones por hardware

Como se mencionó en la sección 1.2, las interrupciones por hardware son necesarias para lidiar con la diferencia de velocidad entre la CPU y los dispositivos de ES. Como la CPU y los dispositivos trabajan cada uno a su propia velocidad, cuando un dispositivo está listo para interactuar con la CPU, emite una solicitud de interrupción. La CPU interrumpe la ejecución normal del programa, ejecuta una subrutina que atiende la solicitud del dispositivo, y luego reanuda la ejecución del programa. La dirección de la subrutina a ejecutar se encuentra en el vector de interrupciones, al que se accede mediante un identificador (ID), de manera que los dispositivos no requieren conocer dichas direcciones, sino estar asociados a un ID.

A diferencia de las interrupciones por software, tanto la subrutina que atiende el dispositivo como la configuración previa para lograr que dicha subrutina se ejecute cuando corresponda serán partes del programa que nosotros deberemos de escribir; de esto tratará principalmente esta sección.

Primero, veremos como funciona el PIC, un dispositivo que permite conectar varios dispositivos a la CPU para realizar interrupciones. Luego veremos como se configura el PIC mediante sus registros internos y dos nuevas instrucciones, `in` y `out`. Escribiremos un programa que detecte las pulsaciones de la tecla F10, uno de los dispositivos del MSX88, mediante interrupciones. Analizando este programa, resumiremos los pasos a realizar para programar con interrupciones. Finalmente, aplicaremos estos pasos para programar con el Timer, otro dispositivo del MSX88, y también haremos un ejemplo combinado que utiliza el Timer y la tecla F10.

3.1. El PIC

La CPU del MSX88, por diseño, tiene una sola **línea de entrada** de interrupciones. Cuando por esta línea se indica que hay una interrupción, la CPU espera que el ID de la interrupción se especifique a través del bus de direcciones. Una línea de entrada es básicamente un 'enfuche', entonces en principio sólo se podría conectar un dispositivo a la CPU para que trabaje con interrupciones.

Eso nos pone en un problema, ya que en general uno quiere conectar varios dispositivos en la PC: mouse, teclado, disco duro y pantalla, como mínimo, cada uno con posibilidades de interrumpir a la CPU. Tenemos muchos 'cables' pero un solo enchufe. Por ende, necesitamos algún dispositivo que haga de intermediario entre la CPU y las interrupciones de los dispositivos, que de alguna manera permita conectar los 'cables' de varios dispositivos a la CPU de forma indirecta.

Ese dispositivo es el **Controlador de Interrupciones Programable** (PIC, por *Programmable Interrupt Controller*). El PIC es un dispositivo interno con 4 líneas de interrupción por hardware, donde se conectan dispositivos que pueden interrumpir a la CPU. De esta forma el PIC permite utilizar varios dispositivos, **multiplexando** los pedidos de todos ellos a la única línea de interrupción del procesador [5].

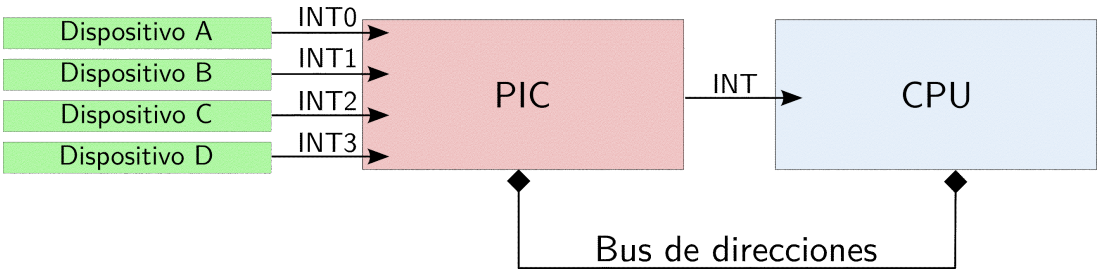

Figura 15: Conexiones del PIC, que actúa como intermediario entre los dispositivos y la CPU.

Los dispositivos se conectan a una de las 4 líneas de interrupción, llamadas INT0, INT1, ..., INT3 (estos nombres no se refieren a IDs de interrupción, sino a las líneas de entrada del PIC).

El MSX88 conecta cuatro dispositivos al PIC:

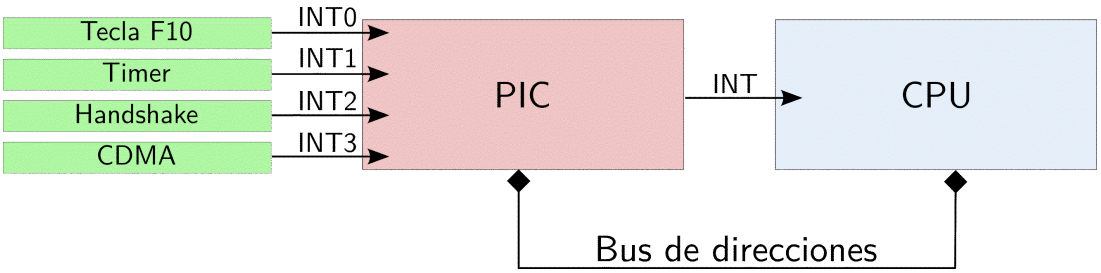

Figura 16: Conexiones de los dispositivos (F10, Timer, Handshake, CDMA) al PIC en el MSX88-

De estos dispositivos, sólo utilizaremos la tecla F10 y el Timer (un reloj) en este apunte.

El PIC es un pequeño procesador, con un poco de memoria interna, y permite:

1. Configurar el ID de interrupción (un número entre 0 y 255) asociado a cada línea de interrupción.
2. Deshabilitar las interrupciones de ciertos dispositivos.
3. Establecer prioridades entre dispositivos (por si varios quieren interrumpir al mismo tiempo).
4. Consultar qué dispositivos se encuentran solicitando interrumpir.
5. Consultar qué dispositivos están siendo atendidos.

Entonces, cuando un dispositivo quiere interrumpir a través de la línea N, el PIC realiza las siguientes acciones:

1. Verifica que la línea N esté habilitada (que *no este enmascarada*, en la jerga del PIC).
2. Verifica que no haya otros dispositivos con mayor prioridad que también quieran interrumpir.
3. Si las dos verificaciones previas fueron exitosas, manda la señal de interrupción a la CPU, al mismo tiempo que pone en el bus de direcciones el identificador de interrupción correspondiente a la línea N.
4. La CPU responde buscando la dirección de la subrutina asociada al identificador de interrupciones en el vector de interrupciones, y ejecutándola.

A continuación se examinarán los **registros internos** del PIC, sus funciones y posibles configuraciones. De esa forma podremos entender cómo se configura el PIC para que pueda realizar correctamente estas acciones.

[5] En realidad el PIC del procesador SX88 tiene 8 líneas de interrupción, llamadas INT0... INT7, pero el simulador MSX88 sólo implementa cuatro (INT0... INT3).

3.2. Registros internos del PIC

Para comunicarnos con los dispositivos, necesitamos intercambiar datos entre los mismos y la CPU. Para ello, cada dispositivo tiene algunos **registros internos**. Estos registros pueden ser de entrada (los datos van del dispositivo a la CPU), de salida (los datos van de la CPU al dispositivo) o ambos.

En el caso del PIC, tenemos los siguientes registros:

Cuadro 4: Registros internos del PIC. Cada uno ocupa un byte (8 bits). La última columna indica si se utilizan mayormente como de entrada (E) o de salida (S).

Dirección	Registro	Nombre	Propósito	E/S
20h	`EOI`	Fin de interrupción	Avisa al PIC que se terminó una interrupción	S
21h	`IMR`	Máscara de interrupciones	Sus bits indican qué líneas de interrupción están habilitadas. Si el bit N vale 1, las interrupciones del dispositivo conectado a la línea INTN serán ignoradas. Si vale 0, las interrupciones del dispositivo serán atendidas en algún momento. Sólo importan los 4 bits menos significativos.	S
22h	`IRR`	Interrupciones pedidas	Sus bits indican qué dispositivos están solicitando una interrupción. Si el bit N vale 1, entonces el dispositivo conectado a la línea INTN está haciendo una solicitud. Sólo importan los 4 bits menos significativos.	E
23h	`ISR`	Interrupción en servicio	Sus bits indican si se está atendiendo la interrupción de algún dispositivo. Si el bit N vale 1, entonces el dispositivo conectado a la línea INTN está siendo atendido. Cómo en el MSX88 sólo se puede atender un dispositivo por vez, nunca habrá más de un bit del registro con el valor 1. Sólo importan los 4 bits menos significativos.	E
24h	`INT0`	ID de Línea INT0	Almacena el ID de la interrupción asociada al dispositivo F10 para buscar en el vector de interrupciones la dirección de comienzo de la subrutina que lo atiende.	S
25h	`INT1`	ID de Línea INT1	Almacena el ID de la interrupción asociada al dispositivo Timer para buscar en el vector de interrupciones la dirección de comienzo de la subrutina que lo atiende.	S
26h	`INT2`	ID de Línea INT2	Almacena el ID de la interrupción asociada al dispositivo Handshake para buscar en el vector de interrupciones la dirección de comienzo de la subrutina que lo atiende.	S
27h	`INT3`	ID de Línea INT3	Almacena el ID de la interrupción asociada al dispositivo CDMA para buscar en el vector de interrupciones la dirección de comienzo de la subrutina que lo atiende.	S

Llamamos a estos *registros*, pero no son registros de la CPU como `AX` o `BX`. En cambio, se utilizan como celdas de una memoria especial llamada **memoria de ES**, diferente a la memoria principal. Este método, en donde los registros de los dispositivos se acceden como celdas de una memoria separada de la principal responde al paradigma llamado **Entrada/Salida aislada**[6].

Entonces, para referenciar un registro de un dispositivo, utilizaremos su *dirección*, no su nombre. Por ejemplo, no podemos usar la palabra *IMR* para referenciar este registro. En cambio, tenemos que acceder a la posición 21H de la memoria de ES, ya que en la tabla se indica que 21H es la dirección del registro IMR.

[6]La otra alternativa, llamada **Entrada/Salida mapeada en memoria** se verá en la segunda parte de la materia con el simulador WinMIPS64.

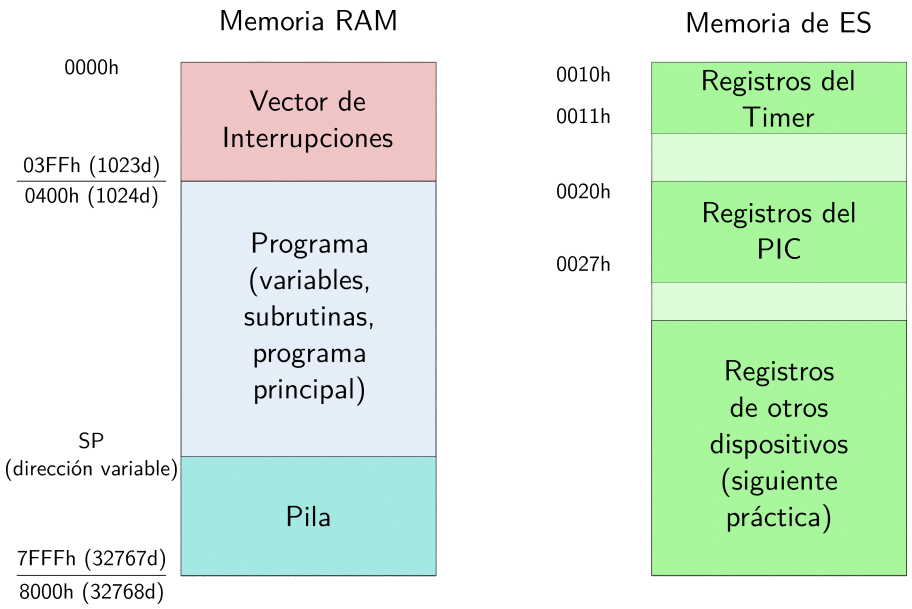

Figura 17: Memoria principal y memoria de ES (registros de internos de los dispositivos).

Por ende, es mejor pensar en estos registros como posiciones especiales de la memoria paralela de ES. A continuación veremos como interactuar con los mismos.

3.3. Instrucciones `in` y `out`

Para leer y modificar los registros internos de los dispositivos, tenemos que utilizar dos nuevas instrucciones, `in` y `out`. Estas instrucciones son parecidas a la instrucción `mov`, pero en lugar de intercambiar datos entre la CPU y la memoria principal, sirven para mover datos desde la CPU hacia los dispositivos (`out`) o traer datos desde los mismos a la CPU (`in`).

La instrucción `out` tiene el formato `out DIRECCION, al`, y la instrucción `in`, el formato `in al, DIRECCION` [7]. ¿Cómo se usan estas instrucciones? Por ejemplo, si para la línea INT0 queremos asociar el ID 20, debemos hacer enviar el valor `20` al registro con dirección `24h`:

Figura 18: Envío del valor `20` al registro INT0 del PIC ubicado en la dirección de ES `24h`.

```
1 mov al, 20
2 out 24h, al
```

Notemos dos cosas importantes:

1. Para referirnos al registro `INT0`, utilizamos su direccion, `24h`.
2. Para pasar el valor 20, primero lo pusimos en el registro `al`, y luego lo escribimos en `INT0` utilizando `out` con `al`.

El registro `al` es el único que puede utilizarse con las instrucciones `in` y `out`, y *siempre* debe utilizarse para pasar o recibir valores; no se permite utilizar otros registros ni otros tipos de direccionamiento (inmediato, directo, indirecto) [8].

¿Qué tal si ahora queremos leer el valor del registro `INT0`, y guardarlo en `cl`? Simplemente utilizamos `in`:

Figura 19: Copia del valor del registro INT0 del PIC (ubicado en la dirección de ES `24h`) al registro `cl`.

[7] Si bien este es el formato que usaremos en las prácticas, el simulador permite otros. Para más información consultar el set de instrucciones de referencia en el sitio web de la cátedra `http://weblidi.info.unlp.edu.ar/catedras/arquitecturaP2003`

[8] Como mencionamos antes, en realidad hay otros formatos y en algunos se puede utilizar `in` o `out` con `AX` o `DX` u otros modos de direccionamiento, pero no los usaremos en la práctica.

```
in al, 24h
mov cl,al
```

Una vez más, fue necesario utilizar `al` con el `in`, y luego copiar ese valor a `cl` con un `mov`.

Hasta aquí, hemos visto cómo manipular registros internos, el diseño del PIC y sus registros, el vector de interrupciones, y el mecanismo de interrupciones. Ahora vamos a combinar estos conceptos para escribir un programa que responda a las interrupciones generadas por las pulsaciones de la tecla F10. Luego analizaremos el programa escrito para extraer las partes generales de la solución que se aplican a cualquier dispositivos que use interrupciones.

3.4. La tecla F10

Vamos a escribir un programa que cuente la cantidad veces se presionó la tecla F10. En el mismo, a las interrupciones de la tecla F10 les vamos a asignar el ID 11 (lo elegimos de forma arbitraria). Para que el programa pueda responder a las pulsaciones de esta tecla debemos:

1. Escribir una subrutina, `contar_pulsaciones`, que incremente un contador cada vez que se ejecute. Esta subrutina es el manejador para la interrupción que se produce cada vez que se presiona la tecla.
2. Configurar el PIC para que se ejecute `contar_pulsaciones` cuando se pulsa F10. Esto requiere de tres pasos:
   - 2.1. Poner la dirección de `contar_pulsaciones` en el elemento 11 del vector de interrupciones, que se encuentra en la dirección $11 * 4 = 44$.
   - 2.2. Habilitar las interrupciones de la línea INT0 (donde está conectado el F10) poniendo el valor `1111 1110` en el registro `IMR`.
   - 2.3. Poner el número `11` como ID para la línea INT0, escribiendo en el registro `INT0` del PIC

Figura 20: Programa que cuenta la cantidad de pulsaciones de la tecla F10.

```
org 1000H
pulsaciones db 0

; Paso 1: escribir la subrutina contar_pulsaciones
org 3000h
contar_pulsaciones: inc pulsaciones
                    ; enviar el valor 20h al registro EOI para indicarle al
                    ; PIC que finalizo el manejo de la interrupcion
                    mov al,20h
                    out 20h,al
                    iret

; paso 2.1: poner la direccion de contar_pulsaciones en la
; posicion 11 del vector de interrupciones (direccion 44, en decimal)
org 44
 dw contar_pulsaciones; no es necesario ponerle una etiqueta

org 2000h
cli; deshabilitar TODAS las interrupciones
;paso 2.2: Poner el ID 11 en el registro INT0
mov al,11
out 24h,al
;paso 2.3: Habilitar las interrupciones de INT0
mov al,11111110B
out 21h,al
sti; habilitar todas las interrupciones que no áestn enmascaradas por el IMR

```

```
28 ; loop infinito para que el programa siga
29 ; ejecutandose, esperando las interrupciones
30 loop_infinito: jmp loop_infinito
31
32 end
```

En este programa se pueden observar algunas cosas que agregamos y no fueron explicadas anteriormente. Todas ellas son importantes a la hora de manejar interrupciones, por lo que siempre se aplican en los programas interrupciones:

- La subrutina `contar_pulsaciones` utiliza la instrucción `iret` en lugar de `ret` para terminar. Esto se debe a que cuando se produce la interrupción y se llama a `contar_pulsaciones` no sólo se apila la dirección de retorno (como en una llamada normal a una subrutina), sino que además se apila el estado de los flags [9]. La instrucción `iret` desapila tanto la dirección de retorno como los flags.

- Además, antes de volver de la subrutina, se ejecutan dos instrucciones, `mov al,20h` y `out 20h,al` para indicar al PIC que la atención de la interrupción ha terminado. Lo que hacen es mover el valor `20h` al registro `EOI` del PIC (que justamente tiene la dirección `20h`, lo cual hace que las cosas sean un poco confusas, debido a que el `20h` de la primera instrucción representa un *valor*, y el `20h` de la segunda se refiere a una *dirección*). De esta forma la CPU estará preparada para procesar un nuevo pedido de interrupción de los que pudieran estar pendientes (bits en 1 en el IRR).

- En el programa, el código de los pasos 2.2 y 2.3 se encuentra rodeado por las instrucciones `sti` y `cli`. Estas instrucciones respectivamente deshabilitan y vuelven a habilitar todas las interrupciones; por ende, en las líneas de código entre `cli` y `sti`, las interrupciones están deshabilitadas. Esta es una buena práctica, ya que en general no queremos atender una interrupción si todavía no terminamos de configurar qué hacer cuando eso sucede[10].

- La línea 42 del programa establece un lazo o loop infinito, de modo que el programa nunca termina. Si no estuviera este lazo, el programa terminaría antes de que se llegue a oprimir la tecla F10 y el manejador de interrupciones `contar_pulsaciones` nunca llegaría a ejecutarse aunque se presione la tecla F10.

- Para poner la dirección de `contar_pulsaciones` en la dirección `44` del vector de interrupciones se utilizaron las instrucciones `org 44` y `dw contar_pulsaciones`. Otra forma de lograrlo es poniendo las instrucciones `mov bx,44` y `mov [bx],contar_pulsaciones` al comienzo del programa inmediatamente luego de `org 2000h`. Ambas formas son equivalentes, pero en general preferimos la primera por ser más simple.

3.5. Uso de constantes para mejorar la legibilidad

En el programa de la subsección anterior aparecen muchos números especiales. Por ejemplo, `11`, el ID de interrupción, o `21h` y `24h`, las direcciones de los registros `IMR` e `INT0` del PIC. Esto hace que la lectura del programa sea más difícil.

, y luego utilizar los nombres de las constantes en el programa.

Para mejorarlo, vamos a definir constantes con estos valores, y luego utilizar los nombres de las constantes en el programa. Si los nombres se eligen adecuadamente (por ejemplo, IMR para el registro IMR del PIC), la lectura del código se vuelve mucho más clara.

Las constantes se definen con la instrucción `nombre_constante EQU valor_constante`. Luego las usamos con su nombre. Al compilar el programa, las constantes se reemplazan directamente por su valor. Veamos cómo queda el programa anterior utilizando esta técnica:

Figura 21: Reescritura del programa que cuenta la cantidad de pulsaciones de la tecla F10 utilizando constantes para mejorar su legibilidad.

[9] Es necesario guardar los flags debido a que no se sabe cuando se va a interrumpir al programa. Si no guardaramos los flags, podría suceder que el manejador de interrupciones los cambie, y el programa, que dependía de los valores anteriores, deje de funcionar correctamente.

[10] Cómo entre esas instrucciones las interrupciones no están habilitadas, no importa el orden en que se realizan los pasos 2.2 y 2.3.

```
; registros del PIC
IMR  EQU 21h
EOI  EQU 20h
INT0 EQU 24h

;valores especiales
MASCARA_F10            EQU 11111110B
VALOR_FIN_INTERRUPCION EQU 20h

;id de interrupcion
ID_F10     EQU 11

org 1000H
pulsaciones db 0

; Paso 1: escribir la subrutina contar_pulsaciones
org 3000h
contar_pulsaciones: inc pulsaciones
                    ; enviar el valor 20h al registro EOI para indicarle al
                    ; PIC que finalizo el manejo de la interrupcion
                    mov al, VALOR_FIN_INTERRUPCION
                    out EOI,al % EOI ,al
                    iret

; paso 2.1: poner la direccion de contar_pulsaciones en la
; posicion 11 del vector de interrupciones (direccion 44, en decimal)
org 44
 dw contar_pulsaciones; no es necesario ponerle una etiqueta

org 2000h
cli
;paso 2.2: Poner el ID 11 en el registro INT0
mov al, ID_F10
out INT0,al  INT0 ,al
;paso 2.3: Habilitar las interrupciones de INT0
mov al, MASCARA_F10
out IMR, al % IMR ,al
sti

; loop infinito para que el programa siga
; ejecutandose, esperando las interrupciones
loop_infinito: jmp loop_infinito

end
```

3.6. Esquema para programar con interrupciones

En esta sección se muestran de forma genérica los pasos requeridos para escribir un programa que interactúe con un dispositivo mediante interrupciones:

1. Escribir el manejador de interrupción, que consiste en una subrutina que se ejecuta cuando ocurre una interrupción para atenderla. La subrutina debe escribir el valor `20h` en el registro `EOI` del PIC antes de finalizar. Además, debe terminar con `iret` en lugar de `ret` para restaurar los flags del programa interrumpido, así quedan en el mismo estado en que estaban previo a la interrupción.

2. Según el dispositivo que querramos utilizar, buscar en la tabla 16 cuál es la línea de interrupción del PIC a la cual se conecta el mismo. Llamaremos INTN a esta línea (por facilidad del lector, recordamos aquí que la tecla F10 se conecta a la línea INT0, el Timer a la INT1, el Handshake a la INT2 y el DMAC a la INT3).

3. Elegir un ID de interrupción que no esté utilizado (no elegir ni 0, 3, 6 o 7, correspondiente a las interrupciones por software). Llamaremos X a este número.

4. Configurar el PIC para que se ejecute la subrutina

   4.1. Poner la dirección del manejador de interrupción en el elemento X del vector de interrupciones, que se encuentra en la dirección $X * 4$.

   4.2. Habilitar las interrupciones de la línea INTN poniendo una máscara adecuada en el registro `IMR`. La máscara tiene 8 bits (el bit 7 es el de más a la izquierda, el 0 el de más a la derecha), donde cada bit corresponde a una línea. En este caso, el bit de la línea N debe valor 0, y el resto 1.

   4.3. Poner el número X como ID para la línea INTN, escribiendo en el registro `INTN` del PIC

Estos pasos también nos definen un esquema que podemos seguir para escribir el programa, como se muestra en en el cuadro 22:

Figura 22: Esquema general de un programa que utiliza interrupciones.

3.7. El Timer

El Timer es otro dispositivo de ES como el F10. Se utiliza como un reloj despertador para la CPU. Se configura para contar una cantidad determinada de segundos y cuando finaliza la cuenta emite una interrupción. El Timer tiene dos registros, `CONT` y `COMP`, con direcciones de la memoria de ES `10h` y `11h`, respectivamente.

Cuadro 5: Registros de E/S del Timer.

Dirección	Registro	Nombre Largo	Propósito
10h	`CONT`	Contador	Se incrementa automáticamente una vez por segundo, para contar el tiempo transcurrido.
11h	`COMP`	Comparador	Contiene el tiempo límite del Timer. Cuando CONT vale igual que COMP, se dispara la interrupción.

Escribamos un programa que imprime el mensaje `"Hola"` una vez cada diez segundos. Una gran parte del programa será similar al que escribimos para la tecla F10; la diferencia principal estará en la implementación del manejador de interrupción.

Figura 23: Programa que imprime el mensaje `"Hola"` cada 10 segundos, utilizando el Timer.

```
1 ; registros del PIC
2 IMR EQU 21h
3 EOI EQU 20h
4 INT1 EQU 25h
```

```
5
6  ; registros del Timer
7  CONT EQU 10h
8  COMP EQU 11h
9
10 ;valores especiales
11 MASCARA_TIMER EQU 11111101B
12 VALOR_FIN_INTERRUPCION EQU 20h
13
14 ;id de interrupcion y direccion asociada
15 ID_TIMER EQU 13
16
17 org 1000H
18 mensaje db "Hola"
19
20 ; Paso 1: escribir la subrutina imprimir_hola
21 org 3000h
22 imprimir_hola:mov bx, offset mensaje
23               mov al,4 ; cantidad de caracteres del mensaje
24               int 7
25               ; Volver a poner el CONT en 0, para que vuelva a
26               ; contar desde 0. Si no incluimos este codigo, no
27               ; obtendremos una interrupcion cada COMP segundos. 6
28               mov al,0
29               out CONT,al
30               ; enviar el valor 20h al registro EOI para indicarle al
31               ; PIC que finalizo el manejo de la interrupcion
32               mov al,VALOR_FIN_INTERRUPCION
33               out EOI,al
34               iret
35
36 ; Poner la direccion de imprimir_hola en la posicion 13
37 ; del vector de interrupciones (direccion 52, en decimal)
38 org 52
39   dw imprimir_hola; no es necesario ponerle una etiqueta
40
41                 ;Programa principal
42                 org 2000h
43                 cli
44                 ; Poner el ID 13 en el registro INT1
45                 mov al,ID_TIMER
46                 out INT1,al
47                 ;Habilitar la interrupcion de INT1
48                 mov al,MASCARA_TIMER
49                 out IMR,al
50                 ;Configurar el COMP del Timer en 10 segundos
51                 mov al, 10
52                 out COMP, al
53                 ; Poner el CONT en 0, para que comience a contar desde 0
54                 mov al,0
55                 out CONT,al
56                 sti
57                 ; loop infinito para que el programa siga
58                 ; ejecutandose, esperando las interrupciones
59 loop_infinito:  jmp loop_infinito
60                 end
```

Notemos tres cosas importantes:

- El registro `COMP` fue inicializado con el valor 10, para que las interrupciones se generen una vez cada 10 segundos.
- El valor de `CONT` se incrementa una vez por segundo; cuando `CONT` es igual a `COMP`, se genera la interrupción.

- El registro `CONT` no vuelve automáticamente a 0 cuando se produce una interrupción. Por eso, debe ser re-inicializado con el valor 0 cada vez que ocurre una interrupción. Para lograrlo, agregamos dos líneas que hacen eso en la subrutina `imprimir_hola`. Si `CONT` no se pusiera en 0, seguiría incrementándose, superando el valor 10, por lo que el programa no funcionaría adecuadamente.

3.8. Ejemplo combinado

Ahora vamos a ver un ejemplo en el que se utilizan al mismo tiempo la interrupción del Timer y la de la tecla F10. Para ello, vamos a programar un juego sencillo con la siguiente lógica:

- El jugador debe presionar la tecla F10 para comenzar el juego, y luego presionar la tecla F10 exactamente 60 segundos después de que la presionó por primera vez.
- Si lo logra, el programa muestra el mensaje `"Muy bien!"`. Si toca la tecla antes de los 60 segundos, el programa muestra el mensaje `"Muy rapido!"`, y si lo toca después, `"Muy lento!"`.
- Luego, haya acertado o no, el juego vuelve a empezar.

Al comienzo del juego, siempre se muestra un mensaje explicando como jugar: `"El objetivo del juego es ver si podes llevar la cuenta del tiempo correctamente sin un reloj. Pulsa F10 para comenzar, y vuelve a pulsar F10 cuando creas que transcurrieron exactamente 60 segundos desde la primera pulsación"`.

Para este programa necesitaremos configurar el PIC para las interrupciones del Timer y de la tecla F10. Además, utilizaremos una variable que indique en que estado estamos (esperando la primera pulsación del F10, o la segunda). Cuando se presiona la tecla F10 se cambia de estado; si se estaba esperando la primera pulsación del F10, se debe prender el Timer, si se está esperando la segunda, se debe verificar si el jugador ganó, mostrar el mensaje con el resultado, y resetear el juego al estado inicial. Por otro lado, cuando el Timer está prendido, se debe llevar la cuenta de los segundos transcurridos. Por ende, el Timer debe configurarse para interrumpir cada un segundo (no cada 60!) de modo de poder llevar la cuenta segundo a segundo. He aquí una posible implementación de la solución:

Figura 24: Implementación de un juego simple, cuyo objetivo es ver si el jugador puede contar los segundos que pasan sin un reloj.

```
;direcciones de los registros del PIC
IMR EQU 21h
EOI EQU 20h
INT0 EQU 24h
INT1 EQU 25h
;direcciones de los registros del Timer
CONT EQU 10h
COMP EQU 11h
;valores especiales
MASC_F10 EQU 11111110B
MASC_TIMER_F10 EQU 11111100B
FIN_INT EQU 20h
;estados
ESTADO_INICIAL EQU 0
ESTADO_ADIVINANDO EQU 1
SEGUNDOS_ADIVINAR EQU 60

ID_F10 EQU 4
ID_TIMER EQU 5

org 16
  dw cambiar_estado

org 20
  dw actualizar_tiempo

org 1000h
  estado db ESTADO_INICIAL
  segundos db 0
```

```
  mje_ganador db "Muy␣bien!"
  mje_lento db "Muy␣lento!"
  mje_rapido db "Muy␣rapido!"
  mje_guia db "Objetivo:␣contabilizar␣tiempo␣transcurrido.␣"
        db "Pulsa␣F10␣para␣comenzar,␣y␣vuelve␣a␣pulsar␣F10␣"
        db "cuando␣te␣parezca␣que␣pasaron␣exactamente␣60␣seg."
  fin_msj   db ?

org 3000h
actualizar_tiempo:inc segundos
                  mov al,0
                  out CONT,al ; reiniciar CONT
                  mov al,FIN_INT
                  out EOI,al ; avisar al PIC
                  iret

org 3200h
ver_inst: mov bx, offset mje_guia
          mov al, offset fin_msj - offset mje_guia
          int 7
          ret

org 3300h
inicializar: mov segundos,0; reiniciar contador de segundos
             cli
             mov al,MASC_F10
             out IMR,al; habilitar solo el F10
             sti
             ret

org 3400h
ver_res:  cmp segundos,SEGUNDOS_ADIVINAR
          jz ganador
          js rapido
          ;si no salta, fue lento
          mov bx, offset mje_lento
          mov al,offset mje_rapido - offset mje_lento
          jmp imprimir
ganador:  mov bx, offset mje_ganador
          mov al,offset mje_lento - offset mje_ganador
          jmp imprimir
rapido:   mov bx, offset mje_lento
          mov al,offset mje_guia - offset mje_rapido
imprimir: int 7
          ret

org 3500h
cambiar_estado: cmp estado,ESTADO_INICIAL
                jnz evaluar_ganador
                ;estamos en ESTADO_INICIAL
                mov al,0
                out CONT,al; reinicio el timer
                mov al,MASC_TIMER_F10
                out IMR,al; activamos el timer
                mov estado,ESTADO_ADIVINANDO
                jmp fin
                ;estamos en ESTADO_ADIVINANDO
evaluar_ganador: call ver_res
                 call inicializar
                 call ver_inst
                 mov estado,ESTADO_INICIAL
            fin: mov al,FIN_INT
                 out EOI,al; avisar al PIC
                 iret
;programa principal
org 2000h
```

```
96                cli
97                mov al,ID_F10
98                out INT0,al; id de interrupcion del F10
99                mov al,ID_TIMER
100               out INT1,al; id de interrupcion del Timer
101               mov al, 1
102               out COMP, al; el timer interrumpe cada 1 segundo
103               sti
104               call inicializar
105               call ver_inst
106               ; loop infinito para que el programa siga
107               ; ejecutandose, esperando las interrupciones
108   loop_infinito: jmp loop_infinito
109   end
```

3.9. Resumen

En esta sección vimos cómo escribir programas que interactúan con dispositivos mediante interrupciones. Utilizamos los elementos principales involucrados en las interrupciones: el vector de interrupciones, el identificador de interrupción, los dispositivos (PIC, Timer, F10), las líneas de interrupción del PIC, los registros de E/S y las instrucciones nuevas (`in`, `out`, `cli`, `sti`, `iret`). Si bien la relación entre estos elementos es compleja, utilizando la tecla F10 como caso de estudio desarrollamos un esquema sistemático para programar con interrupciones, que luego aplicamos a programar con el Timer. Este esquema será util para programar otros dispositivos disponibles en el MSX88, como el Handshake y el CDMA.

4. Anexo: Lectura y escritura de números

Lectura de números Si queremos leer un número, no nos queda otra que leer varios dígitos, es decir, varios caracteres. Para simplificar, pongamos como objetivo leer un número de dos dígitos (o sea, entre 00 y 99). Para lograrlo, tenemos que leer dos dígitos: el primero va en el lugar de las decenas, el segundo en el de las unidades. Cada uno de ellos quedaría guardado en un byte diferente. Por ejemplo, si leemos primero el `"3"` y luego el `"7"`, obtendríamos lo siguiente:

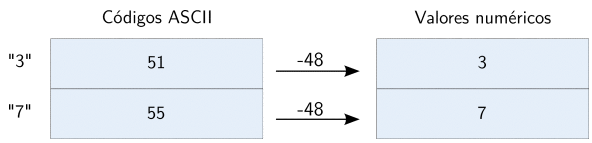

Figura 25: Conversión de dígitos ASCII a su valor numérico.

Pero con eso todavía no terminamos: si vamos a guardar el número 37, en realidad no necesitamos dos bytes, sino uno. Es decir, ahora el número está guardado en dos bytes como si fuera un Decimal Codificado con Binario desempaquetado (BCD, por Binary Coded Decimal). Lo que necesitamos es juntar esos dos bytes en uno solo que represente al número 37 en en BSS, que es el tipo de codificación que estamos acostumbrados a manejar y que usa la CPU por defecto. Para ello tenemos que multiplicar el 3 por 10, de modo que obtenga el valor que le corresponde por estar en la posición de las decenas; luego le sumamos el 7, la parte de las unidades, y obtenemos el número $3 \times 10 + 7 = 30 + 7 = 37$ como queríamos.

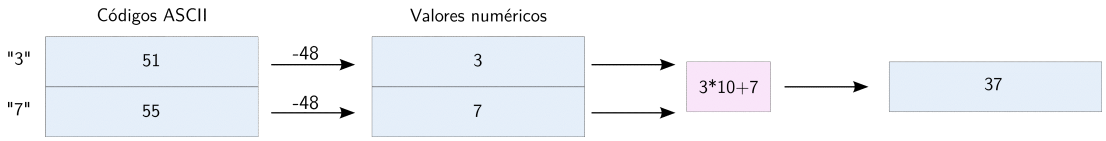

Figura 26: Conversión de dígitos ASCII a su valor numérico. y posterior combinación de ambos como un único valor, codificado en BSS.

En la figura 27 mostramos un programa que lee dos caracteres que corresponden a un número de dos dígitos, y los transforma en un sólo número codificado en BSS como muestra la figura anterior. Para ello implementamos una subrutina, `leer_numero`, que recibe como parámetro la dirección donde se debe guardar el número. A su vez, `leer_numero` utiliza otra subrutina, `mul`, para multiplicar el dígito de las decenas por 10.

Figura 27: Lectura de un número de dos dígitos

```
1  D_CERO EQU 48 ; 48 es el codigo ascii del 0
2
3  org 1000H
4  numero db ?
5
6  org 2000h; programa principal
7  mov bx, offset numero
8  call leer_numero
9  ; ahora la variable numero tiene el numero leido
10 ; codificado en BSS
11 int 0
12
13 org 3000h
14 ; leer_numero recibe una direccion de memoria en bx
15 ; lee dos caracteres de teclado y los convierte en
16 ; un numero de dos digitos codificado en BSS
17 leer_numero: push dx
18              push ax
19              int 6 ;lee el digito de las decenas
20              mov dl, [bx]
21              sub dl,D_CERO; lo guarda en dl y le resta el ascii de "0"
22              mov dh,10
23              call mul; multiplica dl por 10
24              int 6
25              mov al, [bx]
26              sub al,D_CERO; lee el digito de las unidades
27              add dl,al; lo suma a lo anterior
28              mov [bx],dl ; guarda el resultado en la direccion pasada como parametro
29              pop ax
30              pop dx
31              ret
32
33 org 3500h
34 ; mul recibe en dl y dh dos numeros y los multiplica.
35 ; el resultado queda en dl
36 mul:   push ax; preservar ax
37        mov al,dl ; pasamos los valores a AL y AH
38        mov ah,dh ; para poder poner el resultado en dl
39        mov dl,0  ; inicializamos el resultado
40   loop:cmp ah,0 ; while ah>0
41        jz fin
42        dec ah
43        add dl,al
44        jmp loop
45    fin:pop ax
46        ret
47
48 end
```

4.1. Impresión de números en pantalla (mostrando strings)

Para mostrar números codificados en BSS en la pantalla tenemos un problema el problema inverso; sólo podemos mostrar caracteres, por ende tenemos que transformar el número en BSS en un string donde cada carácter del string sea un dígito del número.

Como antes, comencemos con el caso de números de un sólo dígito. En ese caso, para convertir el dígito al carácter ASCII correspondiente debemos ahora *sumar* el valor ASCII del carácter `"0"`. El siguiente programa ilustra esta técnica:

Figura 28: Impresión de un número de un solo dígito.

```
D_CERO EQU 48 ; 48 es el codigo ascii del 0
org 1000H
numero db 4
letra db ?

org 2000h
; convertimos el digito en un caracter ascii
mov ah,numero
add ah,D_CERO
;imprimimos el caracter ascii
mov letra, ah
mov al,1
mov bx,offset letra
int 7
int 0
end
```

Ahora, ¿qué pasa si el número tiene 2 dígitos? Entonces tenemos que separar las decenas de las unidades. ¿Cómo lo logramos? Dividiendo por 10. Para seguir con el ejemplo anterior, 37 dividido 10 da 3 como cociente y 7 como resto, que son los dos dígitos que deben almacenarse en la memoria (y que luego pueden utilizarse para mostrar el número en pantalla). A continuación, cada dígito se convierte en su correspondiente código ASCII, y se guardan de forma consecutiva formando un string. El siguiente programa ilustra este procedimiento.

Figura 29: Impresión del número 37, de dos dígitos.

```
D_CERO EQU 48 ; 48 es el codigo ascii del 0

org 1000H
numero db 37
string_numero db ?,?

org 3000h
; div divide dos numeros, calculando su resto y cociente, mediante restas sucesivas
; recibe:
;    dh: divisor
;    dl: dividendo
; devuelve:
;    cl: cociente
;    ch: resto
div: push dx; preservar dx
     mov ch,dl
     mov cl,0
; "while ch>dh", o sea "while numero>10" si dividimos un numero por 10
loop:cmp ch,dh
     js fin
     sub ch,dh
     inc cl
     jmp loop
```

```
fin:pop dx
    ret


    org 2000h; programa principal
    ; calcular digitos (resto y cociente de dividir por 10)
    mov dl,numero
    mov dh,10
    call div
    ; convertir en codigos ASCII
    add cl,D_CERO
    add ch,D_CERO
    ; copiar a string_numero
    mov bx, offset string_numero
    mov [bx],cl
    inc bx
    mov [bx],ch
    dec bx
    ; imprimir el string
    mov al,2
    int 7
    int 0
    end
```

Números con más de dos dígitos No hemos detallado cómo imprimir números con más de dos dígitos. No vamos a profundizar en este tema, pero a continuación damos una idea general del algoritmo. Primero, deberíamos armar un string con los dígitos del número, es decir, un vector donde cada elemento sea el código ascii de cada uno de los dígitos, en el orden correcto. El problema principal es que el esquema de dividir por 10 nos da los dígitos del número de derecha a izquierda, y nosotros necesitamos armar un string de izquierda a derecha. Una forma de resolver esto es repetir el procedimiento anterior, haciendo divisiones por 10 hasta que el resto sea 0, para primero *contar* cuántos digitos hay. Luego, sabiendo la cantidad de dígitos, podemos nuevamente ir haciendo divisiones por 10 y guardando los restos de derecha a izquierda en bytes separados, codificados en ASCII, para generar el string.

Números negativos Tampoco hemos mencionado cómo imprimir números negativos. En el simulador, dichos números están codificados en complemento a 2 (CA2). Tampocos vamos a profundizar en este tema, pero una vez que tenemos claro cómo imprimir números en BSS es fácil imprimir números en CA2. En ese caso, si el número es positivo (primer bit en 0), entonces directamente imprimimos el número como si estuviera en BSS, lo cual ya sabemos hacer. Si el número es negativo (primer bit en 1), entonces hacemos dos cosas: 1) Imprimimos un `"-"`, 2) Obtenemos el valor absoluto del número (en BSS) y lo imprimimos. Para obtener el valor absoluto, simplemente debemos complementarlo utilizando la instrucción `neg`. Un procedimiento similar (reemplazando las operaciones de impresión por lecturas, y e invirtiendo su orden) serviría para leer números negativos.