ETC II, Práctica 2.

1 OBJETIVOS:

- Construir el dispositivo de adquisición de datos. El dispositivo se maneja a través del puerto paralelo del PC, LPT1 (que utiliza la interrupción IRQ7). El esquema eléctrico está documentado en el apartado del hardware.

  Con fin de escribir el *driver*, el dispositivo se puede sustituir con lo que se denomina *"NULL device*", que consiste en un conector DB25 (el del puerto paralelo) con un puente entre las señales ACK y RD/WR (pines 10 y 17 respectivamente). La descripción del hardware está en las paginas WEB correspondientes.

- Escribir el programa *driver* de manejo del dispositivo. El programa está compuesto de dos partes -- una que sirve como interfaz del usuario y otra que sirve para el manejo de las interrupciones del dispositivo. El programa comunica con la aplicación del usuario utilizando la INT 61h. La funcionalidad es la misma que en el *driver* de la práctica 1.

Recursos de hardware utilizados:

- LTP1: Puerto paralelo, permite conectar el dispositivo con el ordenador.
- RTC: Reloj de Tiempo Real del ordenador. Utilizado para pedir los datos al ADC de manera periódica, en intervalos determinados por el usuario.

Recursos de software:

- INT 61h: Programa residente en memoria, servicio de atención a la interrupción de usuario 61h

2 FUNCIONAMIENTO DEL *DRIVER*:

1. La parte de instalación instala el *driver*. Las acciones que debe de realizar esta parte se describen en este manual. Después de la instalación correcta del *driver*:

   - El reloj genera interrupciones con una frecuencia de 1024 Hz y el PC recibe la INT 70h con la misma frecuencia.

   - Al inicializar una transformación analógico digital, si el dispositivo esta instalado correctamente, el PC recibe IRQ7 después de que ACK sube de nivel.

2. La parte que recibe las interrupciones del reloj cuenta un intervalo determinado por el usuario y al terminar dicho intervalo se pide el dato del primer canal activo, después de que el *driver* sale de la interrupción. En este punto se puede levantar una bandera que significa que el *driver* esta pidiendo datos del ADC.
3. Al recibir la interrupción del LPT1 (es decir del ADC), el *driver* guarda el dato recibido en el buffer correspondiente al canal (el numero de canal se recibe a través del LPT1), se elige el siguiente canal activo y se pide de nuevo dato del dispositivo, después de que se sale del manejo de la interrupción. De esta manera el punto 3 se repite tantas veces como canales activos hay. Al no tener mas canales activos, el *driver* baja la bandera que señala que se están pidiendo datos.
4. Si el *timeout* del *driver* expira y la bandera está levantada está claro que el dispositivo no funciona, o que el intervalo de tiempo elegido es más corto que el tiempo de conversión del ADC y el *driver* funciona erróneamente, Esta información se proporciona el usuario a la llamada siguiente. También hay que señalar al usuario si existe relleno del buffer de algún canal.

3 CRONOGRAMA DE LAS INTERRUPCIONES DEL DRIVER. SUPONIENDO QUE LOS CANALES ACTIVOS SON 0,1,3.

✍ int timer decremento del contador del timeout. Limpiar interrupción. Salir de interrupción.
✍ int timer decremento del contador del timeout. Limpiar interrupción. Salir de interrupción.
…
✍ int timer decremento del contador del timeout. Limpiar interrupción. Salir de interrupción.
✍ int timer (timeout). Poner valor inicial del contador de timeout. Limpiar interrupción. Iniciar petición de datos. Set canal 0 (en el hardware). Pedir datos. Salir del driver.
✍ int lpt1: -- los datos están disponibles. Leer dato. Determinar número canal. Guardar dato (canal 0). Limpiar la interrupción. Determinar el siguiente canal activo. Set canal (1). Pedir datos. Salir de la interrupción.
✍ int lpt1: -- los datos están disponibles. Leer dato. Determinar numero canal. Guardar dato (canal 1). Limpiar la interrupción. Determinar el siguiente canal activo. Set canal (3). Pedir datos. Salir de la interrupción.
✍ int lpt1: -- los datos están disponibles. Leer dato. Determinar numero canal. Guardar dato (canal 3). Limpiar la interrupción. Determinar el siguiente canal activo. No existe. Salir de la interrupción.
✍ int timer decremento del contador del timeout. Limpiar interrupción. Salir de interrupción.
….

4 POSIBLE PLAN DE DESARROLLO:

✍✍ Escribir la parte de manejo del reloj. Como evidencia del funcionamiento cada vez que se recibe una interrupción incrementar la posición 0B800:0 y cada vez que hay timeout la posición 0B800:2.

✍✍ Utilizando el "*NULL device*", pedir datos de un canal, siempre activo cuando expira el timeout. Primero hacer un programa aparte que pide un dato, espera la señal de que el dato está listo y leer el dato. Después incorporar solo la parte que pide dato (sin recibir ni esperar) en la interrupción del reloj. Al pedir el dato incrementar 0B800:4. La petición del dato va a generar la IRQ07. Capturar esta interrupción. Leer el dato del LPT1, poner el dispositivo para que puede generar mas interrupciones , incrementar 0B800:6 y escribir el dato en 0B800:8 antes de salir.

✍✍ Guardar el dato en un buffer cíclico. Poner cuatro canales.

5 ESTRUCTURA DEL *DRIVER* PARA MANEJAR UN DISPOSITIVO QUE GENERA INTERRUPCIONES DE HARDWARE:

Cada *driver* consta, al menos, de las siguientes partes:

✍✍ Parte de instalación del *driver*,
✍✍ Parte baja, que maneja las interrupciones de hardware generadas por el dispositivo.
✍✍ Parte alta, que comunica la aplicación del usuario con la parte baja y el dispositivo.
✍✍ A veces existe parte de desinstalación del *driver*.

5.1 La parte de instalación del *driver*:

1. Comprueba si el dispositivo está instalado. Si no, termina el programa sin instalar.
2. Genera el espacio de datos del *driver*.
3. Prohíbe interrupciones.
4. Pone el dispositivo en estado que no puede generar interrupciones.
5. Lo programa en su modo operativo.
6. Instala el vector de interrupciones, guardando el antiguo.
7. Programa el PIC.
8. Habilita las interrupciones.
9. Da señal al SO que instale el *driver*. En caso de DOS, queda como TSR.

Esta parte puede utilizar todos los recursos del SO en entorno DOS.

5.2 La parte de manejo de interrupciones generadas por el hardware.

1. Comprueba si la interrupción ha sido pedida por este dispositivo. Si no, está generada por otro dispositivo y va a la dirección antigua de manejo.
2. Guarda TODOS los registros utilizados en la pila.
3. Si la interrupción está generada por este dispositivo, el programa lee los datos del dispositivo (o escribe los datos si se trata de salida).
4. Libera la señal de hardware que genera la interrupción, es decir, asegura que la señal eléctrica de la interrupción está en estado inactivo.
5. Maneja los datos del *driver*.
6. Envía EOI según los requerimientos del SO.
7. Recupera los registros.
8. Sale del manejo de la interrupción.

Comunicación con otras partes: comunica sólo con la parte alta del *driver*, la memoria del *driver* y los registros del dispositivo.

Notar:

Si el tiempo de manejo de una interrupción es corto y limitado no se recomienda habilitar las interrupciones. Al tenerlas prohibidas hay seguridad que el trabajo de esta parte no se interrumpe por otras interrupciones de hardware.

No es recomendable manejar el PIC o cualquier otra parte del sistema que genéricamente esta dedicada a procesar interrupciones.

Al iniciar esta parte el único registro de segmento con valor determinado es CS.

5.3 La parte alta del driver:

Es la parte visible para la aplicación. La aplicación llama a esta parte, utilizando interrupciones del software. Esta parte consiste en un conjunto de servicios proporcionados por el *driver*. En algunos sistemas operativos, como por ejemplo UNIX, algunos de los servicios son definidos e iguales para todos los *drivers*. En todos los casos hay servicios específicos para cada dispositivo o tipo de dispositivo que sirven para controlarlo y que son dependientes del dispositivo.

Funcionalidad:

Diversa, dependiente del dispositivo. Casi siempre inicializa la petición de datos para activar la parte baja del *driver*.

El *driver* que se va a desarrollar en esta práctica se hace de manera parecida a los servicios del BIOS del PC, es decir: la parte alta se implementa como interrupción de software y la parte baja como programas que manejan las interrupciones de hardware que utiliza este dispositivo (del RT clock) y del LPT1.

La interfaz del usuario debe de tener los mismos servicios como el driver-simulador de la práctica 1.

Comunica con:

El proceso del usuario: con interrupciones de software.

La parte baja del driver: la memoria del driver y los registros del dispositivo.

SO: raras veces. Se puede utilizar, porque es una parte de la aplicación. Hay que permitir las interrupciones previamente.

Recursos que se pueden utilizar.

Dependiente del SO. De todos modos esta parte tiene acceso a los datos del usuario, a la memoria del *driver* que comunica con la parte baja del *driver* y los registros de E/S del dispositivo. En DOS esta parte del *driver* puede utilizar todos los recursos del sistema.

6 EJEMPLO DE *DRIVER* DE ESTE TIPO -- EL DRIVER DEL TECLADO DEL BIOS.

Accesible utilizando int 16h. Utiliza interrupción de hardware 09. Advertencia -- este código se puede utilizar solo con fines educativos.

7 ERRORES FRECUENTES:

- Utilizar registros sin recuperar en el manejo de interrupción de hardware. Consecuencias -- lo mas probable es que el PC se queda "colgado".
- Utilizar registros de segmentos sin inicializar en el manejo de interrupción de hardware. Consecuencias: Los más probable es que se destruyen los datos del usuario escribiendo es su segmentos de datos.
- Utilizar variables que cambian en la parte de manejo de interrupciones de hardware. El *driver* funciona probabilisticamente de manera extraña.

Ejemplo típico:

```
Timeint proc
        Inc tiempo
        Cmp max_tiempo_usuario
        Jne ok_exit
        Call timeoout
Ok_exit:
        ... EOI ...
        IRET
Timeint endp
```

Donde max_tiempo_usuario puede cambiarlo el usuario en la parte alta y se encuentra en el segmento de datos del programa. Si max_tiempo_usario cambia en el ultimo ciclo antes del timeout se produce error.

- No liberar el dispositivo y no ponerlo en estado sin interrupción. Consecuencias: Se genera solo una interrupción. Si hay dispositivos encadenados que utilizan la misma interrupción se produce colapso de estos dispositivos.
- Repetir código (cut and paste). Esto en la mayoría de los casos en ensamblador produce errores al no corregir sincrono las partes afectadas. Lo típico es repetir 4 veces algún procedimiento para los cuatro canales. Mejor utilizar un registro de índice.
- Esperar en bucle mientras dura la interrupción del dispositivo. El peligro es que el dispositivo puede no estar instalado o defectuoso y se bloquea la máquina.

8 ESTRUCTURAS DE DATOS Y PSEUDOCODIGO:

Al escribir en ensamblador a veces es mas fácil utilizar como punto de partida pseudocódigo de alto nivel. Aquí viene un ejemplo:

Suponemos que tenemos que programar una cola de words de tamaño MAX_QUE que tiene dos funciones -- añadir y leer dato.

El pseudocodigo puede ser:

```
int add_que(int *queue,int head,int tail,int data) {
          queue[head]=data;
          head++;
          if (head==MAX_QUE) head=0;
          if (head==tail) {
          tail++;
          if (tail==MAX_QUE) {
              tail=0;
              overrun=1;
            }
            }
}

int get_que(int *que,int head,int tail) {
    int a;
    if (head==tail) return NO_DATA;
    a=queue[tail];
    tail++;
   if (tail==MAX_QUE) tail=0;
   return a;
}
```

Pregunta: ¿porqué se trata de pseudocódigo y no de código?

Pasos para convertir este programa en programa en ensamblador.

1. Decidir qué variable va a qué registro.
2. Decidir como se van a comunicar los parámetros y los resultados de las funciones.
3. "Compilar" el programa a mano.

Queue tiene que ir a una variable o un registro que podamos utilizar como dirección, por ejemplo BX.

Head y tail tiene que ir a registros que se pueden utilizar como índices, por ejemplo SI y DI.

Data puede ir a AX y las condiciones NO_DATA y overrun pueden ir como el flag CY.

Los resultados pueden ir a AX y los registros de tail y head:

```
; int get_que(int *queue,int head,int tail) {
; que en BX, head en SI, tail en DI
GET_QUE PROC
;    int a; -- AX
;    if (head==tail) return NO_DATA;
          CMP     SI,DI
          JNE     OK
          STC
          RET
OK:
;
 ; a=queue[tail];
          MOV     AX,[BX+DI]
 ;   tail++;
          ADD     DI,2
;   if (tail==MAX_QUE) tail=0;
          CMP     DI,MAX_BUF
          JNE     OK1
          XOR DI,DI
OK1:
; return a;
          CLC
          RET
```

; }
GET_QUE ENDP
Ejercicio: escribir add_que compilado.

9 ESTRUCTURA GENERAL DE MANEJO DE DISPOSITIVOS CON INTERRUPCIONES DE HARDWARE

El manejo de los dispositivos se escribe en niveles, según su grado de acceso a los recursos del sistema. Los niveles 2 y 3 se llaman servicio.

Niveles:

9.1 Nivel de aplicación de usuario:

Propósito – la aplicación del usuario.

Tipo de acceso al dispositivo – a través de interrupción del software (nivel 2).

Ejemplo: MOV AH, 2 // INT 16h

Tipo de programas – todos los tipos de aplicaciones, escritos en bajo o alto nivel, que utilizan este dispositivo o esta clase de dispositivos si se trata de un servicio unificado (por ejemplo todos interfaces serie se utilizan de la misma manera del parte del software, independiente de la versión del BIOS o el tipo del chip UART.

Prioridades de programación – según la aplicación, pero en general:

- Versatilidad.
- Buena estructura.
- Fácil manejo de los fuentes.

Acceso de los recursos del SO

- Total (salvo en caso de *driver* del SO).

Dificultades de programación y depuración – pocas.

Subniveles:

- *Drivers* del SO (a veces se mezcla con nivel 2).
- Librerías estándar del uso de dispositivos (*stdio*).

En el caso de practica 2 – uso directo del nivel 2.

9.2 Nivel de interrupción del software:

Propósito –

- Aislar el nivel del usuario del manejo del dispositivo.
- Manejar el dispositivo en manera independiente del hardware
- En sistemas multitareas – garantizar la serialización del uso del dispositivo (no permitir que dos programas clientes utilizan el dispositivo en el mismo tiempo en manera asíncrona).
- Manejar un dispositivo unificado en manera común para este grupo de dispositivos.

Tipo del acceso al dispositivo –

- A través de DATA POOL (DP) – método principal.
- Uso directo de los recursos del hardware del dispositivo.

Tipo de programas

- En general programas de bajo nivel (ensamblador), orientados a un arquitectura del ordenador (IBM PC) y un sistema operativo (MS DOS).

Acceso de los recursos del SO

- Poco restringido. En el caso del DOS no restringido.

Prioridades de programación:

- Interfaz unificado y con funcionalidad completa.
- Facilidad de verificar
- Velocidad
- Uso modesto de recursos del sistema.
- En caso del SO multitarea – uso no monopolista de los recursos (si es posible).

Dificultades de programación y depuración

– medio - baja.

Subniveles:

2.1. Rutinas de mantenimiento,

- Modos de funcionamiento del servicio.
- Instalación/desinstalación del servicio.
- Diagnosticas del servicio

2.2. Rutinas que manejan solo es data pool

- Transferir datos de/a dispositivo.
- Control del timing del dispositivo.
- Esperas de terminación de operaciones
- Sincronización y seralizacion de tareas (si no se hace del SO).

2.2. Rutinas que manejan el dispositivo (instrucciones in-out).

- Arrancan el autómata finito del dispositivo, para manejar una operación de entrada - salida.
- Otras manipulaciones necesarias del hardware.
- En caso de manejo muy simple del hardware – sustituye la parte 3.2.

9.3 Data Pool (DP)

Estrictamente no es ningún nivel del software, tal como se trata de estructuras de datos en la memoria del sistema. Si es un nivel del servicio, tal como este es el modo de comunicación entre el nivel de la interrupción del software y las interrupciones del hardware.

El DP esta compuesto por estructuras de datos que manejan es software en estado coherente con el estado del dispositivo, es estado del propio servicio y el estado del SO.

Las estructuras más utilizadas en el DP son:

- Banderas *Flags* (mirar la instrucción TSET).
- Contadores
- Enumeradores del estado de autómatas
- *Buffers* cíclicos
- Colas
- Buffer Pool

9.4 Nivel de manejo de interrupciones del hardware:

Subniveles:

Por su carácter las interrupciones son

4.1. Interrupciones propios (del dispositivo) y
4.2. Interrupciones del sistema (por ejemplo del timer - 1Ch, del DMA etc.).

Propósito

- Manejar el dispositivo con instrucciones in-out obedeciendo las limitaciones de su hardware.
- Manejar los timeouts, el timing del dispositivo etc.

Tipo del acceso al dispositivo –

- Uso directo de los recursos del hardware del dispositivo según los datos en el DP.

Tipo de programas

- De bajo nivel (ensamblador), orientados a un arquitectura del ordenador (IBM PC) y el dispositivo. Pocas veces interactúan con el SO.
- Funcionamiento asíncrono en respecto con otras programas (incluso SO).

Acceso de los recursos del SO

- Muy restringido. Casi no existente en la mayoría de los SO (se utilizan bibliotecas muy limitadas y pequeñas para acceder a unos de los servicios de SO).
- En DOS - hay manera de acceder a los servicios de DOS si el momento de la interrupción es oportuno, es decir si no hay otros programas funcionando en las secciones criticas de DOS. Esto hace los programas que utilizan DOS en este nivel del tipo funcionando con probabilidad cerca de 1. Es decir – por suerte.

Prioridades de programación:

- Simplicidad
- Uso no monopolista de los recursos del sistema. Incluso en los casos del sistema de un usuario y una tarea las interrupciones son otra tarea y pueden bloquear el sistema.
- Facilidad de verificar
- Velocidad

Dificultades de programación y depuración

– Alta - muy alta, por lo tanto se hace lo mínimo posible en este nivel y en la manera mas simple posible. La técnica mas utilizada es de autómatas finitos.

9.5 Manejo de dispositivos