Breve pausa

En el capítulo anterior estudiamos los mecanismos básicos de E/S en serie. Esta vez analizaremos las E/S a través de interfaces paralelas

Comenzaremos con una mirada a un ejemplo de salida en serie con una subrutina llamada mensaje. Esta rutina tiene la dirección del mensaje a ella pasado mediante el registro de dirección A3. Así, por ejemplo:

```
LEA TEXT,A3      establece un puntero hacia el
                 mensaje TEX
JSR MENSAJE      salida del mensaje
```

dará salida al mensaje TEXT almacenado así:

```
TEXT:DC.B        'Mi Computer',$00
```

donde DC.B es una directiva del ensamblador para declarar el espacio de memoria que albergará el texto 'Mi Computer'. El byte nulo $00 representa el fin del mensaje, e informa a la subrutina mensaje que ya no hay más texto.

La subrutina para dar salida a todos los bytes en la tabla TEXT usará la subrutina OUTCH, que ya conocemos de un capítulo anterior. En el ejemplo, OUTCH empleaba el bit "preparado" para dar salida a los caracteres uno a uno tan pronto como estuviera lista la ACIA para enviar el carácter siguiente. La subrutina mensaje tendrá, pues, el siguiente aspecto:

```
MENSAJE:  MOVE.B   (A3)+D0   toma el siguiente
                             byte de mensaje
          BEQ      DONE      comprueba el final
                             del mensaje
          JSR      OUTCH     salida de
                             caracteres
          BRA      MENSAJE   bucle hasta llegar
                             al final del
                             mensaje
DONE:     RTS
```

Cada byte del mensaje se copia en D0 (el registro de datos empleado como parámetro de valor para OUTCH), empleando A3 como puntero con direccionamiento indirecto posincremental. Si el byte es cero, se abandona la rutina mediante la etiqueta DONE; de lo contrario se da salida a un carácter a través de OUTCH.

Si precisamos dar salida a datos en el modo paralelo, en el que todos los bits de los datos aparecerán simultáneamente, el chip PIA nos proporciona las facilidades. Sin embargo, dado que éste es un chip interface para uso general, debemos configurarlo en la forma adecuada a la configuración particular de nuestro hardware. Esto se parece al chip ACIA, donde debemos establecer la velocidad de transmisión y los formatos de los bytes.

Para el PIA, habremos de configurar las ocho líneas paralelas de *ambas* mitades del chip (A y B) para que sean o bien líneas de salida o bien de entrada (es decir, debemos configurar la dirección de los datos). Para ello escribiremos en el bit 2 del registro de control (CRA o CRB) para indicar que deseamos establecer la dirección de los datos en un registro de dirección de datos (DDRA o DDRB). Por ejemplo:

```
CONFIGPIA: CLR.B     PIACRB        pone a cero
                                   el bit de
                                   registro de
                                   control
           MOVE.B    #$FF,PIADDRB  establece la
                                   dirección de
                                   los datos en
                                   salida
           BSET      #2,PIACRB     retorna al
                                   registro
                                   normal de
                                   datos
           CLR.B     PIACRA        da para A
                                   la dirección
                                   del sentido
                                   de los datos
           CLR.B     PIADDRA       establece el
                                   sentido de los
                                   datos en
                                   salida
           BSET      #2,PIACRA     vuelve el
                                   registro
                                   normal de
                                   datos
```

Este fragmento establecerá todo el lado A para entrada, y el lado B para salida. La transmisión de los datos a los dispositivos paralelos se ejecutará con:

```
MOVE.B   D0,PIADRA   salida del contenido de D0
```

y para la entrada tendremos lo siguiente:

```
MOVE.B   PIADRA,D1   lee la entrada en D1
```

Obsérvese que todas las direcciones PIA a las que nos referimos deben establecerse inicialmente, como por ejemplo:

```
PIADRA   EQU   $30051
PIACRA   EQU   $30053
```

Obsérvese también que el significado de los datos leídos o escritos hacia el PIA dependen del tipo de dispositivo que se conecta eléctricamente a los canales digitales del chip (p. ej., podríamos tener una visualización de siete segmentos conectada, como se muestra en el diagrama donde puede verse que todo dígito decimal puede construirse mediante el segmento adecuado). Para visualizar el número 3, pongamos por caso, deberemos encender los segmentos 1, 4, 2, 5 y 3.

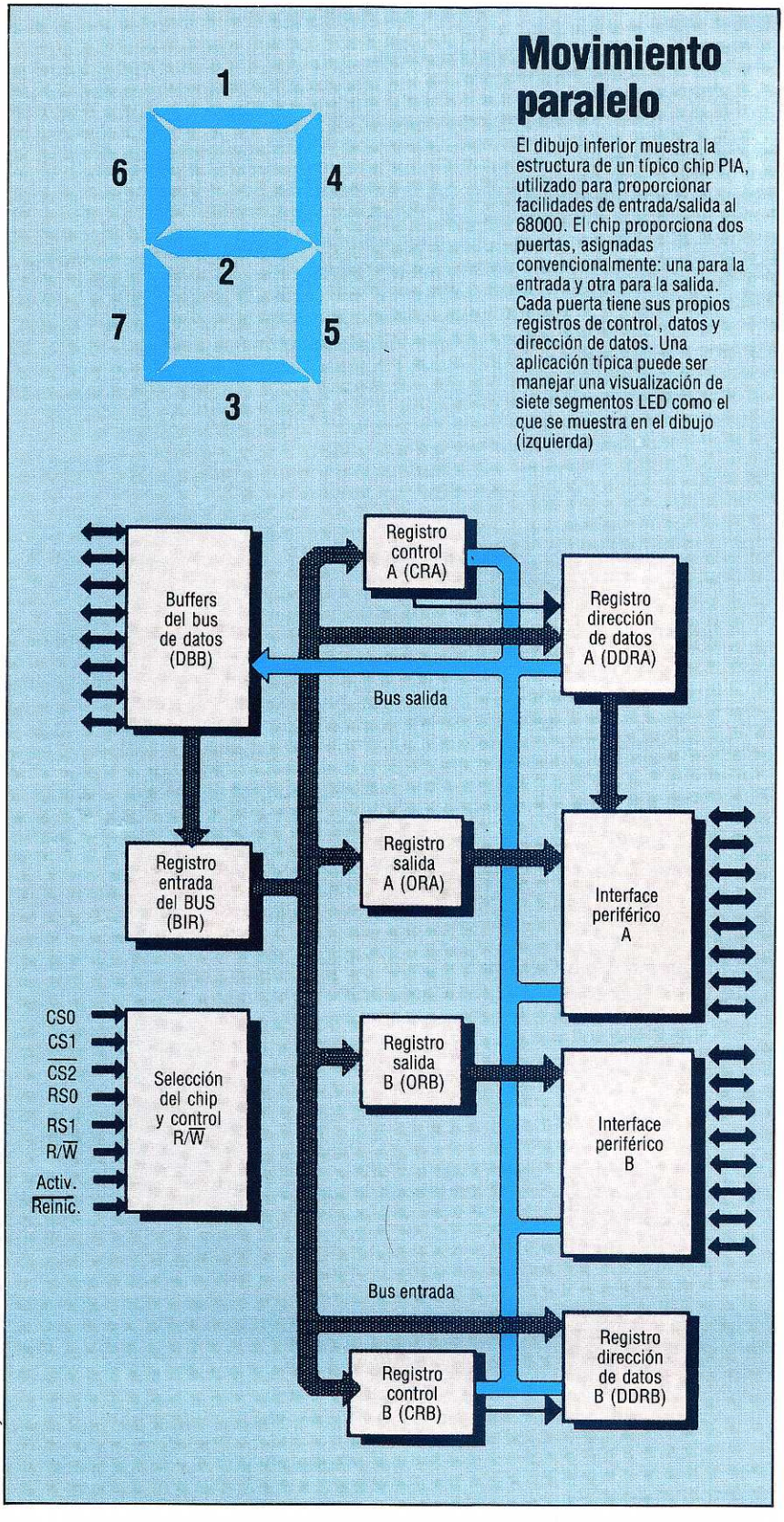
No sólo necesitaremos transmitir datos al dispositivo periférico sino que se puede necesitar el control de su función eléctrica. En el caso propuesto de una visualización con siete segmentos, puede que necesitemos fijar los datos en el dispositivo mediante algunos de los bits de reserva escritos en modo salida dentro de la palabra de datos. A menudo estas señales de control simulan el pulso de un "reloj" eléctrico, pudiendo traducirse en una activación y desactivación del bit de control. Así:

```
BSET    #CONBITNO,PIADRA
                    activa el bit de control
JSR     DELAY       espera un pequeño instante
BCLR    #CONBITNO,
        PIADRA      y lo reinicializa después
```

proporciona el "reloj" a cualquier canal digital que se asigne a CONBITNO en la dirección PIADRA.

Movimiento paralelo

El dibujo inferior muestra la estructura de un típico chip PIA, utilizado para proporcionar facilidades de entrada/salida al 68000. El chip proporciona dos puertas, asignadas convencionalmente: una para la entrada y otra para la salida. Cada puerta tiene sus propios registros de control, datos y dirección de datos. Una aplicación típica puede ser manejar una visualización de siete segmentos LED como el que se muestra en el dibujo (izquierda)



Caroline Clayton

Interrupciones

Aunque el tema de las interrupciones no siempre es bien entendido, la finalidad del uso de interrupciones en un sistema de ordenador tiene que ver con el uso eficiente de la CPU y el poder responder a eventos externos. Por ejemplo, puede que no deseemos que la CPU pierda el tiempo mientras se imprimen los caracteres, como en el ejemplo de la subrutina OUTCH. También necesitamos, para poder hacer otra cosa, saber cuándo se ha terminado de imprimir un carácter para enviar el siguiente disponible.

La situación es aún peor cuando el programa está a la espera de una entrada; digamos, por ejemplo, que del teclado. La eficiencia del sistema depende, como es obvio, de la velocidad de escritura y también de otras tareas asumidas por la CPU, tales como dar salida hacia la impresora en paralelo al tiempo que espera una entrada del teclado.

Para obtener esta operación en paralelo, debemos organizar la secuencia lógica de eventos dentro de la máquina para asegurarnos de que no perdemos el control del programa o algún dato. Veamos lo que se necesita.

- *Guardar el estado del ordenador.* Necesitamos hacer esto para poder volver al programa que hemos interrumpido sin ningún efecto notable o pérdida de datos (lo que sí perderemos inevitablemente es tiempo). Pero primero definiremos qué es lo que constituye el "estado" del ordenador.

Podemos considerarlo como el área total para programas y datos del programa del usuario con su estado de registro y contador del programa. En la práctica, y puesto que en general no se esperarán cambios en el programa del usuario desde la fuente de interrupciones, basta con guardar tan sólo el contador del programa (PC) y el registro de estado (SR). Esto nos dará una idea cabal del estado del programa interrumpido.

- *Localizar la fuente de interrupción.* Esto es necesario porque probablemente tendremos conectados al ordenador más de un dispositivo y necesitamos saber qué rutina de "servicio" se ha de ejecutar.

- *Inhibir interrupciones de otras fuentes.* Necesitamos hacerlo porque si llega otra interrupción mientras atendemos a una ya producida se pueden perder datos. Esto, sin embargo, sólo puede suceder si el tiempo invertido en atender la segunda interrupción ha sido mayor que el tiempo entre la llegada de los datos en la primera interrupción.

• *Entrar la rutina de servicio de la interrupción.* Necesitamos un mecanismo que haga esto, ya sea automáticamente o ejecutando una instrucción específica.

• *Volver al programa interrumpido.* Finalmente, después de haber atendido la interrupción, deberemos volver al estado exacto del programa inmediatamente anterior a la interrupción.

Veamos cómo se consigue esta organización lógica en el 68000. Ante todo, el estado de la máquina se guarda mediante la pila donde se almacena primero el PC del usuario (como palabra larga completa) y después el SR. Con este mecanismo, las interrupciones pueden anidarse de tal modo que se pueda interrumpir una rutina de tratamiento de interrupción.

Si la rutina de tratamiento de interrupción fuera a emplear algunos de los registros que ya hemos señalado para un propósito específico, la rutina puede salvar estos registros a la entrada y restaurarlos a la salida. La instrucción MOVEM hará esto en el 68000 con la mayor sencillez para nosotros:

```
MOVEM   D1,D3,-(SP)     apila los registros
                        de datos
                        código de servicio
                        de interrupción
MOVEM   (SP)+,D1,D3     y los restaura
```

El problema de localizar la fuente de interrupción es también fácil en el 68000, dado que a cada interrupción se le asigna generalmente una sola posición de memoria llamada *vector*, que puede considerarse que contiene un puntero hacia la dirección de la rutina de servicio. Naturalmente, pueden existir varios dispositivos en un determinado vector, en cuyo caso es necesario un *pooling* para establecer cuál es el dispositivo interruptor. Pero éste no suele ser el caso del 68000, por lo general.

La necesidad de arbitrar entre dispositivos interruptores también es atendida por usted. Y esto porque el 68000 asume la prioridad hardware de la fuente interruptora, es decir, *sólo* obtendrán la atención de la CPU las interrupciones de mayor prioridad. Esto significa que las fuentes de datos de alta velocidad tendrán una relativa alta prioridad en un sistema de multiinterrupciones, de modo que no se pierda dato alguno.

Una vez que la interrupción ha atraído la atención de la CPU, se carga en el PC el contenido del vector de interrupción, y se entra en la rutina de servicio de interrupciones. El dispositivo es atendido cargando los datos de entrada en un registro y de allí a algún buffer, o evacuando los datos de un buffer llevándolos a un dispositivo de salida.

La única acción que resta por hacer ahora es volver de la rutina de servicio al programa interrumpido. Esto se consigue casi de idéntica manera que una vuelta de subrutina, empleando esta vez una instrucción RTE y no una RTS. La instrucción RTE vuelve a cargar el PC y el SR automáticamente desde la pila de sistema, por lo que todas las rutinas de servicio deben concluir con esta instrucción. Significa también que en ese momento A7 (el puntero de la pila de sistema) debe estar apuntando a los registros del programa interrumpido. Entonces si usted coloca datos en la pila de sistema, ¡asegúrese de que son sacados de allí antes de ejecutar la instrucción RTE! O quizá más apropiadamente debería usar otra pila, dado que con el 68000 es posible emplear como pila cualquiera de los registros de direcciones.

Rutina de servicio

Veamos ahora una rutina de servicio de interrupciones típica donde tendremos también que hacer *pooling* de los dispositivos para encontrar la fuente interruptora. Dentro de nuestro sistema teórico, tenemos dos posibles fuentes de interrupción en lo que se llama nivel 4 (una línea de interrupción hardware de prioridad 4). Los dos dispositivos son un teclado externo en solitario y un reloj de tiempo real. El reloj se emplea para contar segundos, que sirven a otras partes del sistema (software).

Para establecer correctamente el vector de interrupciones para el nivel 4 (en la dirección $70), necesitamos inicializarlo cuando se inicie por primera vez el programa, como en:

```
MOVE.L  #EXTDEVS,$70    establ. vector nivel 4
```

Es obvio que puede haber otros registros por inicializar, en particular la pila del usuario y la pila de sistema. Por ejemplo, podemos usar:

```
MOVE.L  #STACK,SP       establ. pila sistema
MOVE.L  #USERTACK,A0    y pila usuario
```

donde establecemos A7 con la dirección STACK y A0 con la dirección USERSTACK.

La rutina de servicio de interrupciones EXTDEVS sería, por ejemplo:

```
EXTDEVS MOVEM.L  D0-D7,(A0)  guarda los
                             regs. de datos
        BTST     #7,PIACRA   mira si está
                             el teclado
                             interrumpido
        BNE      CHAR
CKCLK   BTST     #6,PIACRB   si no, lo estará
                             el reloj
        BNE      CLOCK
        BRA      WILD        si no, entonces
                             espurio
CHAR    JSR      KEYBOARD    entra el
                             servicio
                             del teclado
        BRA      CKCLK       pero el reloj
                             puede estar
                             también
                             interrumpido!
CLOCK   JSR      SECS        entra el
                             servicio del reloj
WILD
        MOVEM.L  (SP)+,D0-   restaura los
                 D7          registros
                             de datos
        RTE
```

En esta rutina son guardados todos los registros de datos porque tanto CLOCK como CHAR los usarán. Nótese también que hacemos *pooling* de los dos dispositivos mirando los bits de estado de los registros en los registros de control del PIA. Comprobar el estado del reloj después de haber atendido al teclado, ya que puede tener a la vez dos interrupciones.

En el capítulo final de esta serie examinaremos el ensamblador y las facilidades que ofrece. Si se utilizan bien, quizá resulte la herramienta de desarrollo más poderosa de que dispone el programador.