LENGUAJE MAQUINA DEL COMMODORE 64

I. SINCLAIR

GG

Indice

Prólogo

Multitud de usuarios de ordenadores se limitan a programar en BASIC durante toda su vida. Algunos otros desean aprender más sobre programación y sobre su ordenador, sin querer limitarse a trabajar sólo en BASIC. Sin embargo, muy pocos saben algo sobre el empleo del código máquina, código que permite tan notable control sobre las funciones del ordenador. La razón de eso estriba, según mi parecer, en que la mayoría de los libros que tratan el tema de la programación en código máquina parecen empezar suponiendo al lector familiarizado ya con las ideas y el vocabulario involucrados en este tipo de programación. Además, muchos de estos libros tratan el tema de la programación en código máquina como tema de estudio en sí mismo, proporcionando al lector muy pocas indicaciones de cómo utilizar el código máquina en su ordenador.

Este libro tiene dos propósitos principales. Uno de ellos consiste en introducir al propietario de un Commodore 64 en algunos de los detalles de cómo funciona este ordenador. El segundo propósito estriba en presentar de una forma sencilla los métodos empleados en la programación en código máquina. Quiero hacer hincapié en la palabra *presentar*. Ningún libro puede explicarle todo acerca del código máquina de un determinado ordenador. Todo lo que puedo pretender es iniciarle en el tema. Iniciarle en el tema significa que usted sea capaz de escribir pequeños programas en código máquina, comprender rutinas que encuentre en revistas especializadas y, en general, sacar mayor provecho de su Commodore 64. También significa que sea capaz de utilizar los libros más avanzados sobre el tema y pueda adquirir mayores conocimientos en este fascinante campo.

Comprender el sistema operativo de su Commodore 64 y tener la habilidad suficiente para trabajar en código máquina puede abrirle todo un mundo nuevo por lo que a su ordenador se refiere. Esta es la razón por la que la mayoría de los más espectaculares programas de juegos están escritos en código máquina. También se encontrará con que muchos programas que están escritos fundamentalmente en BASIC, incorporan fragmentos de código máquina con el fin de aprovechar su mayor velocidad y su mejor control del ordenador.

Estoy profundamente agradecido a una serie de personas que han hecho posible la aparición de este libro. Entre ellas, Richard Mi-

les,de Granada Publishing, que encargó su edición y consiguió hacerse con un Commodore 64 y una impresora. El y Sue Moore, también de Granada Pub., llevaron una vez más a cabo sus milagros de meticuloso esfuerzo en mi manuscrito. Estoy muy agradecido también a Milton Bathurst, a quien no conozco personalmente, pero cuyo libro *Inside The Commodore 64* constituyó para mí un elemento muy valioso de consulta.

Ian Sinclair

1. ROM, RAM, bytes y bits

Uno de los factores que más desaniman a los usuarios de ordenadores a la hora de un intento por ir más allá del BASIC es la gran cantidad de palabras nuevas con las que se encuentran. Los autores de muchos libros sobre ordenadores, especialmente los que tratan de la programación en código máquina, suponen que el lector posee unos conocimientos previos de electrónica y será capaz, pues, de comprender todos los términos que aparezcan. Nosotros no vamos a suponer que usted tiene estos conocimientos. Todo lo que daremos por sentado será que posee un Commodore 64 y que tiene ya alguna experiencia programando dicha máquina en BASIC. Eso equivale a empezar desde donde hay que hacerlo: desde el principio. En este libro no queremos interrumpir explicaciones importantes con detalles técnicos o matemáticos. Estos se desarrollarán en los Apéndices. De esta forma, si usted lo desea, podrá leer la exposición completa de algunos puntos o bien olvidarse por completo de ellos.

Para empezar, vamos a estudiar la memoria. Una unidad de memoria es, de cara al ordenador y por lo que a nosotros se refiere tan sólo un circuito eléctrico que actúa como un interruptor. Cuando usted entra en una habitación y enciende una luz, rara vez se detendrá a pensar en lo notable del hecho de que las luces permanezcan encendidas hasta que las apaga. No creo que usted vaya a contar a sus amigos que el circuito de luz contiene una memoria. Sin embargo, cada unidad de memoria de un ordenador es sólo una especie de interruptor en miniatura que puede estar abierto o cerrado. Lo que hace de él una memoria es el hecho de que permanezca en el estado en el que se encuentra (sea abierto o cerrado) hasta que este estado se vea alterado. En un ordenador, una unidad de memoria tal como ésta se denomina *bit,* abreviatura de *binary digit* (dígito binario) con lo que se quiere dar a entender que este elemento puede encontrarse en dos estados posibles.

Continuaremos con el ejemplo del interruptor ya que se demuestra muy útil. Supongamos que queremos elaborar una serie de señales mediante circuitos eléctricos e interruptores. Podríamos utilizar un circuito tal como el de la figura 1.1. Cuando el interruptor está cerrado, la luz se enciende, lo cual puede interpretarse como SI. Cuando el interruptor está abierto, la luz se apaga, lo cual podemos interpretarlo

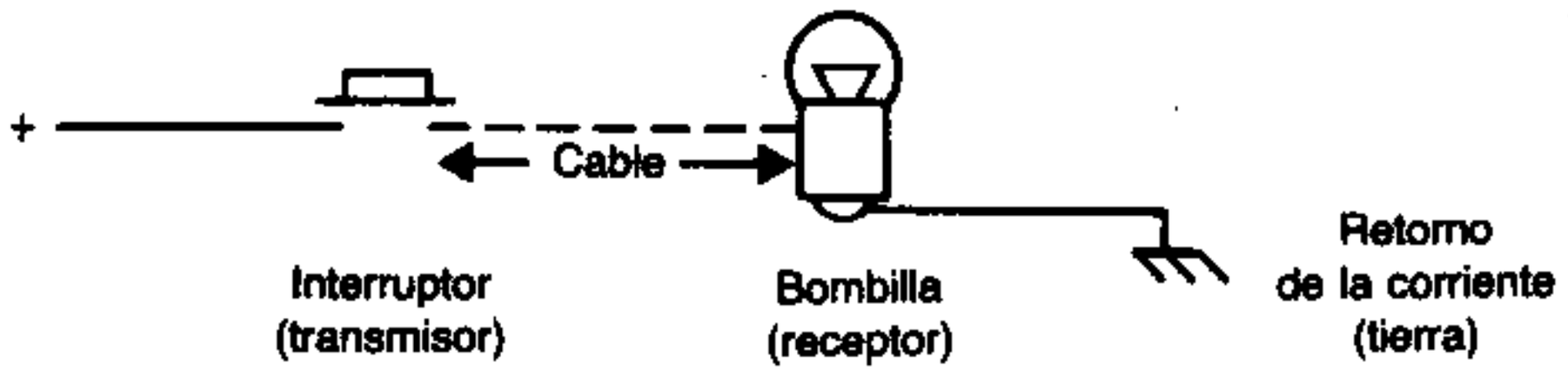
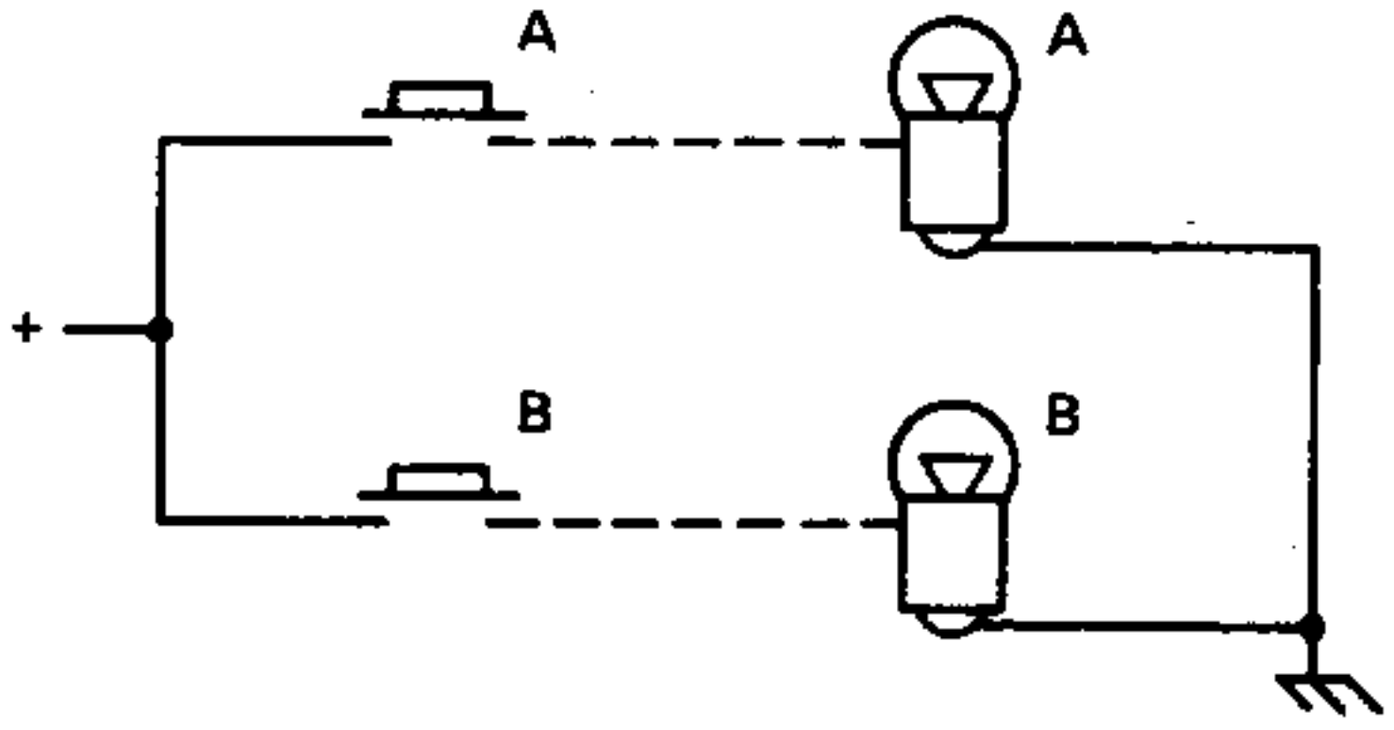


Fig. 1.1 Montaje consistente en un cable con un interruptor y una bombilla indicadora.



A	B
No (apagado)	No (apagado)
No (apagado)	Si (encendido)
Si (encendido)	No (apagado)
Si (encendido)	Si (encendido)

Fig. 1.2 Señalización a través de dos líneas. Se pueden mandar cuatro señales diferentes.

como un NO. Podrían asociarse dos significados cualesquiera a estas dos condiciones (llamadas «estados») de la luz, mientras haya sólo dos. Las cosas cambian cuando se utilizan dos interruptores y dos luces, tal como en la figura 1.2. Ahora pueden darse cuatro combinaciones posibles distintas: *a)* ambas luces apagadas; *b)* A encendida y B apagada; *c)* A apagada y B encendida, y *d)* ambas encendidas. Este conjunto de cuatro posibilidades lleva asociado el hecho de que se puedan señalar cuatro circunstancias distintas. Mediante una línea se pueden obtener dos códigos. Dos líneas permiten obtener cuatro. Siguiendo por este camino se ve que mediante tres líneas se pueden generar ocho códigos distintos. Un momento de reflexión sugiere que como 4 es 2x2 y 8 es 2x2x2 cuatro líneas permitirán generar 2x2x2x2 (igual a 16) códigos. Esto es cierto y como normalmente escribimos 2x2x2x2 como 2^4 (dos elevado a la cuarta potencia), podemos encontrar fácilmente cuántos códigos se pueden transmitir mediante el empleo de cualquier número de líneas. Según lo anterior, ocho líneas, por

ejemplo, serán capaces de generar 2^8 (equivalente a 256) códigos. A un conjunto de ocho interruptores se le puede asociar, pues, 256 significados distintos. Sin embargo, ya es hora de decidir el modo en que queremos emplear estas señales.

Una forma particularmente útil la constituye lo que se llama el *código binario.* El código binario consiste en un procedimiento para representar números utilizando sólo dos dígitos: el Ø y el 1. Podemos asociar al Ø la idea de «interruptor abierto» y al 1 la idea de «interruptor cerrado», con lo que utilizando ocho interruptores se podrán codificar 256 números distintos. Este grupo de ocho bits se denomina *byte* y es la unidad que se utiliza para especificar el tamaño de la memoria de nuestros ordenadores. Esta es la razón de porqué aparecen tan frecuentemente los números 8 y 256 en el campo de la programación en código máquina.

La forma en que se distribuyen los ocho bits dentro de un byte con el fin de representar cierta cantidad se corresponde con la manera en que indicamos, normalmente, un número. Cuando escribimos una cifra tal como 256, el 6 representa 6 unidades, el 5 escrito a su izquierda representa 5 decenas y el 2 que aparece un lugar más a la izquierda significa dos centenas. El 6 del 256 se denomina el «dígito menos significativo» y el 2 es el «dígito más significativo». Cambiando el 6 por un 7 o un 5, cambiamos el número sólo en una unidad. Cambiando el 2 por un 1 o un 3, cambiamos el número en una centena, o sea mucho más que antes.

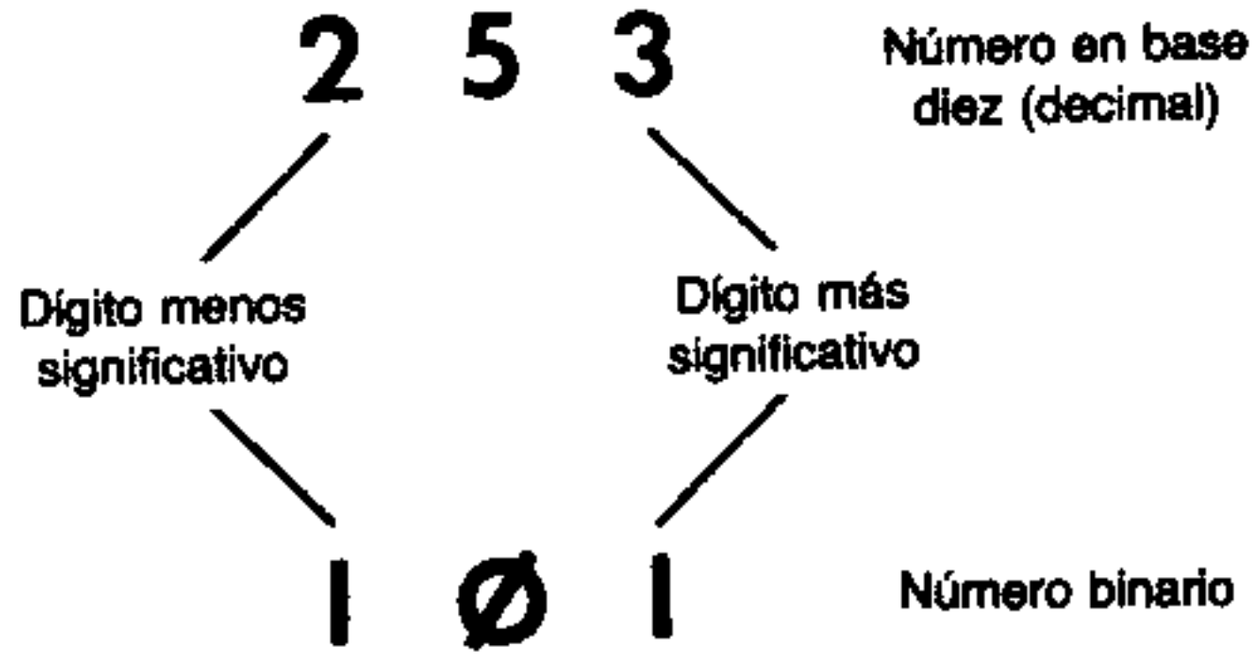


Fig. 1.3 Peso relativo de los diferentes dígitos. Nuestro sistema de numeración utiliza la posición de un dígito dentro de un número para indicar su importancia relativa.

Una vez visto lo que son los bits y los bytes, es hora de volver a la idea de la memoria como un conjunto de interruptores. De hecho, en un ordenador son necesarios dos tipos de memoria. Uno de ellos debe ser de tipo permanente, tal como los interruptores mecánicos o bien las conexiones fijas ya que debe emplearse para almacenar las instrucciones codificadas numéricamente con las que trabaja el ordenador. Este tipo de memoria se denomina ROM (del inglés Read Only

Memory, o sea memoria de sólo lectura) y lleva asociado el hecho de que pueda ver y copiar los datos que se encuentran almacenados en ella, pero puede borrarlos o bien modificarlos. La ROM es la parte más importante de su ordenador ya que contiene todas las instrucciones que hacen que el ordenador lleve a cabo las distintas acciones del BASIC. Cuando usted escribe un programa, el ordenador lo almacena en una parte de la memoria que puede utilizarse una y otra vez en forma de otro conjunto de instrucciones codificadas también numéricamente. Este es otro tipo de memoria que tanto puede ser «escrita» como puede ser «leída». De ser congruentes con lo que hemos visto anteriormente, bautizaríamos este tipo de memoria con el nombre de RWM (del inglés Read Write Memory, o sea memoria de lectura y escritura). Desgraciadamente no somos todo lo congruentes que deberíamos, conociéndose este tipo de memoria con el nombre de RAM (del inglés Random Access Memory, o sea memoria de acceso aleatorio). Este es un nombre que se empleó en los primeros tiempos de la informática para distinguir este tipo de memoria de otra que funcionaba de un modo distinto. Estamos condenados pues a utilizar por ahora, y posiblemente para siempre, el nombre de RAM.

El carnaval de los códigos numéricos

Podemos volver ahora a los bytes. Vimos anteriormente que con un byte, constituido por un grupo de ocho bits se podían adoptar 256 configuraciones distintas, por lo que a estos bits se refiere. La distribución más útil es, sin embargo, la correspondiente al denominado *código binario*. Las distintas configuraciones de bits en el código binario representan números que se escriben habitualmente del Ø al 255 (no del 1 al 256 ya que es necesario un código para el cero). Cada byte de los 38929 de los que dispone el Commodore 64 en su RAM puede almacenar, pues, un número comprendido dentro del rango Ø-255.

Los números por sí mismos no son de demasiada utilidad y no consideraríamos a un ordenador particularmente útil si pudiese trabajar sólo con números comprendidos entre Ø y 255. Lo que se hace, pues, es utilizar estos números como códigos. De hecho, cada código numérico puede ser utilizado para significar cosas distintas. Si ya ha trabajado usted en BASIC con los códigos ASCII, sabrá que cada letra del alfabeto, cada uno de los dígitos del Ø al 9 y cada signo de puntuación, se codifican en ASCII como un número comprendido entre 32 (código correspondiente al espacio en blanco) y 127 (código correspondiente a la flecha hacia la izquierda). Esta selección deja libres gran cantidad de códigos numéricos ASCII que se pueden emplear para otros propósitos, tal como los caracteres gráficos. Sin embargo, el código ASCII no es el único. El Commodore 64 utiliza sus propios criterios de interpretación para los números comprendidos en el rango

Ø-255. Así, por ejemplo, cuando se teclea la palabra PRINT en una línea de programa, lo que se coloca en la memoria del Commodore 64 (después de pulsar ENTER) no es la secuencia de códigos ASCII ligados a PRINT. Estos serían 8Ø, 82, 73, 78, y 84: un byte para cada letra. Lo que se guarda en la memoria, de hecho, es sólo un byte: el correspondiente a la forma binaria del 153. Este byte es lo que se llama un *token* y puede ser empleado de dos formas distintas por el ordenador. Una es para localizar los códigos ASCII asociados a la palabra PRINT. Estos códigos están almacenados en ROM, de forma que cuando se imprime (LIST) un programa, se ve aparecer la palabra PRINT y no un carácter cuyo código ASCII sea el 153. La otra forma de utilización de estos «tokens», precisamente la más importante, es la de localizar un conjunto de instrucciones almacenadas también en ROM en forma de códigos numéricos. Estas instrucciones harán que aparezcan en pantalla unos determinados caracteres. Los números asociados a estos códigos constituyen lo que se llama el *código máquina* ya que controlan directamente lo que hace la «máquina», o sea el ordenador. Este control de tipo directo constituye la principal razón para que queramos utilizar este código máquina. Cuando utilizamos el BASIC, las únicas instrucciones que podemos emplear son aquellas que tienen asociado un determinado «token». A través del código máquina podremos construir nuestras propias instrucciones y hacer todo lo que queramos con el ordenador.

Hay que mencionar la circunstancia de que el hecho de que PRINT genere un «token» es la causa de que se pueda utilizar ? en lugar de dicha instrucción. El Commodore 64 ha sido diseñado de tal forma que un ? que no esté entre comillas provoca que se almacene un 153 en memoria, lo mismo que PRINT.

Hágalo usted mismo

Para digerir mejor toda esta información, pruebe a ejecutar un pequeño programa, concretamente el de la figura 1.4. Está pensado para poner al descubierto todas las palabras clave almacenadas en la ROM, a través de la instrucción PEEK del BASIC.

```
10 PRINT41118;" ";:FOR N=41118 TO 41373
20 K=PEEK(N)
30 IF K<128THENPRINTCHR$(K);
40 IFK>=128THENPRINTCHR$(K-128):PRINTN+1
;" ";
50 NEXT
```

Fig. 1.4 Programa que pone de manifiesto las palabras clave del BASIC del Commodore 64.

Debe seguir a PEEK un número o bien una variable numérica, ambos entre paréntesis. El significado de esta instrucción es el de «ver cual es el byte almacenado en esta dirección numérica». Todos los bytes de la memoria del Commodore 64 están numerados de cero para arriba, con un número asociado a cada byte. Debido al gran parecido entre este sistema y la numeración de las casas en las calles, haremos referencia a estos números con el nombre de *direcciones*. La acción ligada, pues, a PEEK es la de ver qué número (comprendido entre Ø y 255) está almacenado en una dirección de memoria en particular. El Commodore 64 convierte automáticamente estos números del formato binario bajo el cual están almacenados al formato habitual (el decimal) correspondiente a los números que normalmente utilizamos. Empleando CHR$ en nuestro programa, podemos imprimir los caracteres cuyos códigos ASCII correspondan a los números obtenidos a través de PEEK. El programa utiliza la variable N como valor de dirección, comprobándose que PEEK(N) proporciona un valor menor que 128 (o sea un número que se puede asociar a un código ASCII). Si es así, se imprime el carácter.

La razón de todo ello estriba en que el último carácter de cada grupo de palabras (o bien de cada palabra) se codifica de una forma distinta. El número asociado a este último carácter tiene un 128 sumado al código ASCII correspondiente. Así, por ejemplo, las tres primeras direcciones de memoria que, a través de PEEK, lee el programa, contienen los números 69, 78 y 196. El 69 es el código ASCII de la E, el 78 es el código de la N y, finalmente, 196-128=68 corresponde al código ASCII de la D. Así, pues, es como se almacena la palabra END. La razón por la que se trata la última letra de forma distinta es la de ahorrar memoria. De esta forma, no se produce desperdicio alguno ya que la última letra de un grupo siempre tiene un código numérico mayor que 128, con lo que el ordenador la puede identificar fácilmente. Se ha seguido el mismo esquema en el programa BASIC de la figura 1.4, utilizando la línea 4Ø para imprimir la letra correcta, así como para empezar una nueva línea y escribir la dirección correspondiente. Existe también otro conjunto de números almacenados en memoria. Corresponden a las direcciones de la subrutinas que llevan a cabo las acciones del BASIC y que se encuentran almacenadas en el mismo orden que las palabras clave anteriores.

Un vistazo al Commodore 64

Echemos una ojeada al diagrama del Commodore 64 de la figura 1.5. Es un esquema más bien simple ya que se han omitido en él todos los detalles, aunque nos bastará para tener una idea de lo que ocurre en el interior de la máquina. Este tipo de diagrama es lo que se

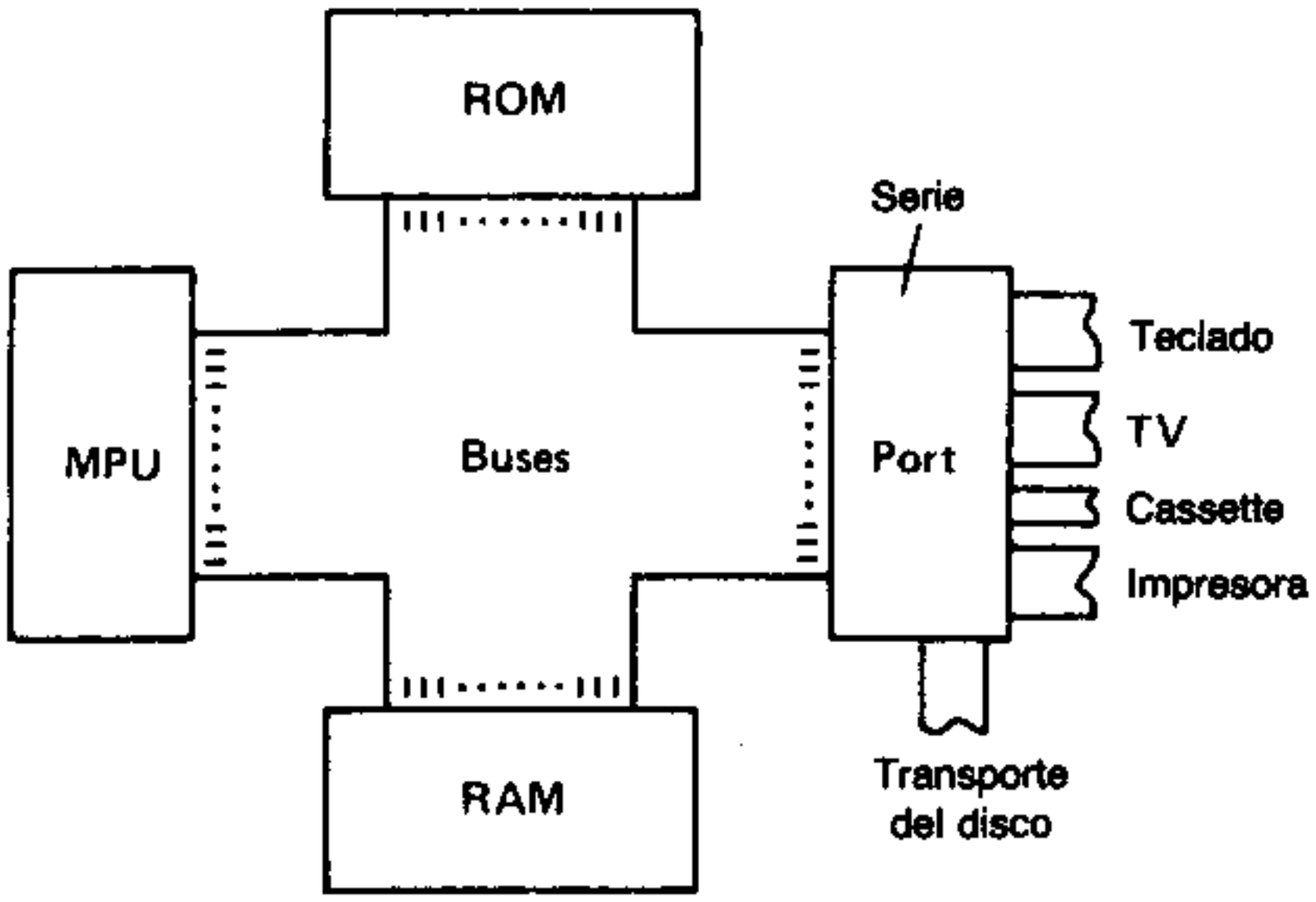


Fig. 1.5 Diagrama de bloques del Commodore 64. Las conexiones marcadas como «buses» consisten en un gran número de líneas que enlazan todas las unidades del sistema.

denomina un *diagrama de bloques*, ya que cada elemento se representa como un bloque, sin especificar lo que pueda estar contenido en él. Los diagramas de bloques son como mapas a gran escala que muestran las carreteras principales entre ciudades, pero no señalan las carreteras comarcales o los caminos vecinales. Un diagrama de bloques es suficiente para que puedan verse las principales vías que siguen las señales eléctricas en el ordenador.

Los nombres de dos de los bloques deberían serle ya familiares (ROM y RAM), aunque no ocurre lo mismo con los otros dos. El bloque marcado como MPU es particularmente importante. MPU significa Unidad del Microprocesador (del inglés Microprocessor Unit), aunque algunos diagramas de bloques emplean para designarlo las siglas CPU, o sea Unidad Central de Proceso (del inglés Central Processor Unit). La MPU es el elemento principal del sistema y constituye, de hecho, una sola unidad. La MPU es un circuito integrado de silicio (de los que ya habrá oído hablar) encapsulado en plástico negro y que posee cuarenta patas de conexión, distribuidas en dos filas de veinte patas cada una (véase la fig. 1.6)

Hay varios tipos de MPU, elaborados por diferentes fabricantes. La que se encuentra en el Commodore 64 es la denominada 65Ø2 (o 65Ø2A).

¿Qué es lo que hace la MPU? La respuesta es: casi todo; y ello a pesar del hecho de que las acciones que puede llevar a cabo son muy pocas y por lo demás muy simples. La MPU puede *leer* un byte, lo cual significa que puede copiar un byte almacenado en la memoria sobre otro tipo de unidad de almacenamiento propia de la MPU. La MPU

también puede *almacenar* un byte, lo cual significa que puede copiar un byte guardado en la unidad de almacenamiento propia de la MPU sobre cualquier dirección de la memoria del ordenador.

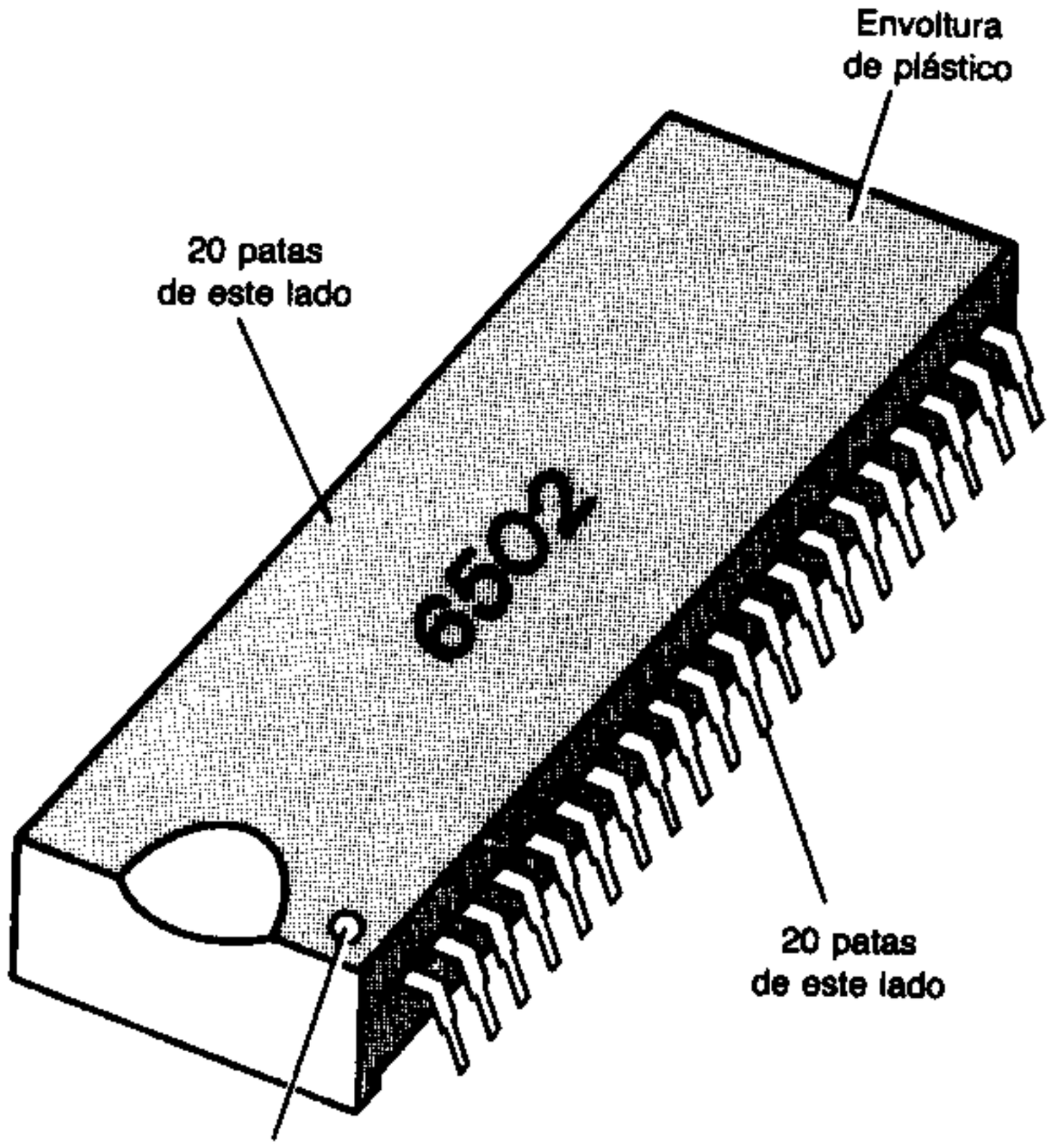


Fig. 1.6 La MPU del 65Ø2. La parte realmente activa es menor, en tamaño, que la uña de un dedo. El envoltorio o cápsula de plástico (de 52 × 14 mm) hace que sea más fácil manipularla.

Estas dos acciones (véase la fig. 1.7) son aquellas en las que la MPU emplea la mayor parte de su tiempo. Combinándolas, puede copiar un byte de una dirección a otra dirección cualquiera de memoria. ¿No opina usted que esto es muy útil? Esta acción de copiar es justamente la que se desencadena cuando se pulsa la letra H en el teclado y se la ve aparecer a continuación en la pantalla. La MPU considera el teclado como un fragmento de memoria y la pantalla como otro, transfiriendo caracteres del uno al otro a medida que los va pulsando. Esto constituye una notable simplificación de la realidad, pero nos sirve, aunque sólo sea para ver lo importante que es la acción ligada a la transferencia de datos.

Lectura y almacenamiento son dos acciones muy importantes de la MPU. Sin embargo, hay algunas otras. Un segundo conjunto de este tipo de acciones lo constituye el *conjunto de acciones aritméticas.* Para la gran mayoría de los MPU, estas acciones se reducen a la

suma y la resta empleando, además, números de un solo byte. Estando comprendido un número de un byte entre Ø y 255, ¿cómo se las arregla, pues, el ordenador para llevar a cabo acciones tales como la multiplicación de grandes cantidades, divisiones, elevaciones a potencias, logaritmos, senos, etc.? La respuesta la tienen los programas en código máquina almacenados en la ROM. Si estos programas no estuviesen allí, usted tendría que escribir los suyos propios. No es probable que hubiese muchos usuarios a los que gustase llevar a cabo una tarea de este tipo.

Fig. 1.7 Los procesos de lectura y almacenamiento. «Leer» significa mandar información desde la memoria a la MPU, de forma que se copie un byte en ésta última. «Almacenar» es el proceso inverso.

Existe también un *conjunto de acciones lógicas.* La lógica de la MPU es, como todas las acciones que lleva a cabo, extremadamente sencilla y sujeta a reglas muy estrictas. Las acciones lógicas comparan los bits de dos bytes y generan una «respuesta» que es función tanto de los bits involucrados como de la regla lógica que se esté empleando. Las tres principales reglas lógicas se denominan AND, OR y XOR. La figura 1.8 muestra cómo se aplica cada una de ellas.

Otro conjunto de acciones es el llamado *conjunto de acciones de bifurcación.* Una bifurcación representa el cambio de una dirección, tal como el GOTO del BASIC. Combinando una prueba de algún tipo con

Y

El resultado de hacer la Y lógica de dos bits es 1 si ambos bits valen 1 y Ø, en caso contrario:

$$1\ Y\ 1 = 1 \quad \begin{cases} 1\ Y\ Ø = Ø \\ Ø\ Y\ 1 = Ø \end{cases} \quad Ø\ Y\ Ø = Ø$$

Para dos bytes, se hace la Y lógica de los bits correspondientes

Y

```
  1Ø11Ø111
  ØØØØ1111
  ØØØØ111
```

sólo estos bits están a 1 en *ambos* bytes

O

El resultado de hace la O lógica de dos bits es 1 si alguno de los dos bits vale 1 y Ø, en caso contrario:

$$1\ O\ 1 = 1 \quad \begin{cases} 1\ O\ Ø = 1 \\ Ø\ O\ 1 = 1 \end{cases} \quad Ø\ O\ Ø = Ø$$

Para dos bytes, se hace la O lógica de los bits correspondientes

O

```
  1Ø11Ø111
  ØØØØ1111
  1Ø111111
```

único bit que está a cero en ambos bytes

XOR (O exclusiva)

Como la O lógica, pero el resultado es cero si ambos bits son idénticos

$$1\ XOR\ 1 = Ø \quad \begin{cases} 1\ XOR\ Ø = 1 \\ Ø\ XOR\ 1 = 1 \end{cases} \quad Ø\ XOR\ Ø = Ø$$

XOR

```
  1Ø11Ø111
  ØØØØ1111
  1Ø111ØØØ
```

si los dos bits son idénticos, el resultado es cero

Fig. 1.8 Reglas para los tres operadores lógicos: Y , O y O exclusiva.

una de estas bifurcaciones es cómo lleva a cabo la MPU sus decisiones. De la misma forma que en BASIC se puede escribir:

```
IF A=36 THEN GOTO 1Ø5Ø
```

puede arreglárselas la MPU para llevar a cabo una instrucción que esté almacenada en una dirección que no sea la dirección siguiente habitual. La MPU es un sistema programado, lo cual significa que lleva a cabo cada una de sus acciones como resultado de serle siministrado un byte con una determinada instrucción, byte que se había previamente almacenado en memoria. Normalmente, cuando se suministra a la MPU una instrucción procedente de una dirección cualquiera (normalmente en ROM), la MPU la ejecuta y lee a continuación el byte con la instrucción almacenada en la siguiente posición de memoria. Una instrucción de bifurcación impide que esto ocurra y en lugar de ir a leer la dirección siguiente la MPU busca su próxima instrucción en otra dirección: la que se le suministra junto con la instrucción de bifurcación. Este salto puede llevarse a cabo dependiendo del resultado de una prueba. Esta prueba se ejecuta normalmente sobre el resultado de la acción anterior (considerando, por ejemplo, si dio un resultado nulo, positivo o bien negativo).

No tenemos, pues, una lista de acciones ni muy larga ni muy espectacular. Sin embargo, las acciones que se han omitido, o no son importantes en el estadio en el que nos encontramos actualmente, o bien no difieren demasiado de las que se han indicado. Lo que queremos resaltar con todo ello es el hecho de que el mágico microprocesador no es un elemento demasiado inteligente. Lo que lo hace vital para el ordenador es que puede ser programado y puede ejecutar a gran velocidad las acciones que se le indiquen. Es asimismo fundamental el hecho de que se pueda programar el microprocesador a través de señales eléctricas.

Estas señales se mandan a ocho líneas de la MPU, las llamadas *líneas de datos.* No es demasiado difícil adivinar que estas ocho líneas corresponden a los ocho bits de un byte. Así pues, cada byte de la memoria puede afectar a la MPU a través de señales eléctricas ligadas al estado de sus diferentes bits. Como es un proceso que se realiza a distancia, lo denominaremos «lectura». Un proceso de lectura equivale, pues, a conectar un byte de memoria a la MPU a través de ocho líneas, de forma que cada bit que esté a cero provoque una señal adecuada en una línea de datos. De la misma forma en que leer un periódico o escuchar una grabación no destruye lo que está escrito o grabado, la lectura de la memoria no afecta para nada a su contenido. El proceso opuesto, el de escritura, sí altera, en cambio, este contenido, análogamente a lo que ocurre cuando se graba una cinta y desaparece lo que estaba grabado en ella. Cuando la MPU escribe un byte en una determinada dirección de memoria, desaparece la información

que estuviese almacenada en ella, quedando sustituida por el nuevo byte. Esto explica porqué es tan fácil escribir nuevas instrucciones BASIC en sustitución de otras antiguas que tuviesen asignado el mismo número de línea.

¿Programas en BASIC o en lenguaje máquina?

¿Quiere realmente escribir programas en BASIC? Esto puede parecer una pregunta absurda, pero no lo es. El trabajo que lleva a cabo un programa se efectúa mediante instrucciones codificadas convenientemente que se suministran a la MPU. Si se escribe sólo en BASIC, no se tiene un control directo sobre estas instrucciones. Todo lo que se hace es elegir dentro de un determinado menú de posibilidades (representadas cada una de ellas por una palabra clave de BASIC) y ordenar las alternativas elegidas en la forma que se espera que conduzca a los resultados adecuados. La elección, pues, queda limitada a las palabras clave contenidas en la ROM. No se puede alterar dicha memoria y si se quiere llevar a cabo una acción que no tenga asociada ninguna palabra clave, habrá que combinar cierto número de estas palabras clave (dando lugar a un programa en BASIC) o bien operar directamente sobre la MPU a través de determinados códigos numéricos (el código máquina). Si se opta por la primera alternativa y se quiere llevar a cabo una acción combinando un determinado número de instrucciones BASIC, el resultado puede no ser demasiado eficiente, especialmente si cada instrucción se compone de varias otras de menor nivel. La otra acción, la de la acción directa, es rápida, pero puede demostrarse difícil. La acción directa a la cual nos referimos concierne al empleo del código máquina. Gran parte de este libro está dedicada a comprender este «lenguaje», lenguaje cuya principal dificultad reside en su gran sencillez.

Veamos una situación que ilustre esta paradoja. Supongamos que dispone de un robot que sólo puede llevar a cabo acciones muy sencillas, aunque puede hacerlo a una gran velocidad. No le puede decir «levanta una pared» ya que esta orden está mucho más allá de su capacidad de comprensión. Usted tiene que decirle lo que tiene que hacer suministrándole todos los detalles. Así, por ejemplo, debería empezar ordenándole: «Traza una línea a partir de un punto que esté a 20 m al sur de la esquina de la casa correspondiente a la cocina hasta otro punto que se encuentre a 20 m al sur de la esquina de la casa correspondiente a la sala de estar. Cava una zanja de 30 cm de profundidad y 20 cm de anchura a lo largo de esta línea. Mezcla tres sacos de arena y dos de cemento junto con cuatro carretones de grava por espacio de 3 minutos. Añade agua a la mezcla hasta que un cubo lleno con la pasta resultante tarde diez segundos en vaciarse al

darle la vuelta. Llena la zanja con la mezcla obtenida, etc.» Las instrucciones son muy detalladas (se supone que son para un robot sin pizca de imaginación), pero serán ejecutadas rápidamente y sin ningún error. Si ha olvidado especificar alguna acción, por muy evidente que sea, ésta no se ejecutará. Olvide de especificar cuánto, dónde y cómo hay que poner el mortero, y se colocarán los ladrillos sin él. Olvide de especificar la altura de la pared y el robot continuará apilando más y más ladrillos hasta que alguien estornude y se venga todo abajo.

La analogía con la programación es notable. Una palabra clave de BASIC es como una instrucciones del tipo «levante una pared» para el robot constructor. Hará que se realice mucho trabajo, ejecutándose gran cantidad de instrucciones que usted no ha suministrado, aunque quizá no se haga la obra con la rapidez que esperaba. Si se puede superar el engorro que representa el hecho de suministrar todos los detalles, se verá que el código máquina es mucho más rápido, ya que con él se suministran de forma indirecta instrucciones a un elemento increíblemente rápido aunque también enormemente estúpido: el microprocesador. Podemos seguir aún con la anterior analogía. Si se ordena al robot «repara el coche», será incapaz de hacerlo. Sin embargo, mediante una adecuada serie de instrucciones elementales, podrá hacerse que se realice el trabajo. El código máquina puede utilizarse para lograr que el ordenador lleve a cabo acciones que no están previstas en el BASIC, aunque es justo decir que muchos ordenadores modernos permiten una mayor gama de instrucciones que los primeros modelos, no siendo ya este aspecto del código máquina tan importante como era anteriormente.

Es necesario echar otra ojeada al diagrama de bloques antes de empezar a trabajar con el Commodore 64 a nivel interno. El bloque que está marcado con la palabra «port» incluye más de un circuito integrado. En el lenguaje de la programación se conoce con el nombre de port a un elemento que se emplea para transferir información byte a byte hacia adentro o bien hacia afuera del resto del sistema que constituye el ordenador: MPU, ROM y RAM. La razón de tener un bloque separado para representar esta acción estriba en que la entrada y la salida de datos son operaciones muy importantes, pero muy lentas. Mediante el empleo de un port puede hacerse que el microprocesador elija el momento en que quiere leer una determinada entrada o bien sacar algo por una salida. Además, pueden independizarse ambas acciones del trabajo habitual de la MPU. Esta es la explicación de porqué no aparece nada en pantalla durante la ejecución de un programa BASIC, excepto cuando se ejecuta una instrucción PRINT. El port oculta la acción del ordenador hasta que éste no tenga que efectuar una entrada o bien una salida de datos.

Se han estudiado ya las partes principales del Commodore 64.

Aunque se han utilizado algunos términos de una forma un tanto relajada (algunos puristas pondrán serias objeciones a la definición de «port» que se ha dado, por ejemplo), la visión de conjunto que se ha presentado es correcta. Lo que hay que hacer ahora es ver como está organizado el ordenador, y ello con el fin de utilizar la MPU, la ROM, la RAM y los ports de forma que puedan ser programados en BASIC y pueda ejecutarse un programa en este lenguaje. Este parece ser un buen punto para empezar un nuevo capítulo.

2. Profundizando en el Commodore 64

¡Con el título de este capítulo no se pretende que empiece a desmontar su ordenador! Lo que se pretende hacer es ver como se las compone el Commodore 64 para cargar y ejecutar programas en BASIC. Empezaremos con una visión simplificada del funcionamiento de todo el conjunto, dejando los detalles para más tarde.

La ROM de su Commodore 64 —que empieza en la dirección 4Ø96Ø—, está compuesta por gran cantidad de pequeños programas (las subrutinas), escritos en código máquina, junto con varios grupos de datos (tablas), del estilo de las tablas de palabras clave. Existe al menos una subrutina en código máquina para cada palabra clave en BASIC, pudiendo algunas de ellas llevar asociadas varias subrutinas. Cuando usted conecta su Commodore 64, el fragmento de programa que se ejecuta se denomina «rutina de inicialización». Es un programa bastante largo, pero debido al hecho de ser tan rápido el código máquina (ejecutándose varios miles de instrucciones por segundo), es poco probable que pueda detectar que se lleva a cabo. Todo lo que podrá apreciar es el pequeño intervalo de tiempo que transcurre entre que se conecta la máquina y se ve aparecer en pantalla el letrero de MICROSOFT. Sin embargo, en este breve lapso se ha tenido tiempo para comprobar el correcto funcionamiento de toda la RAM, cargar una zona de ella con información que se utilizará más tarde y preparar el resto de memoria de forma que quede dispuesta para su uso. El que se haya preparado la memoria para su utilización no significa que no haya nada almacenado en ella. Cuando desconecta el ordenador, la RAM pierde toda la información que tuviese almacenada. Sin embargo, cuando se reconecta no se quedan todas las posiciones de memoria a cero. En cada byte algunos de los bits aparecerán a 1 y los restantes a Ø. Estos unos y estos ceros aparecen al azar de forma que si se pudiese examinar el contenido de cada byte después de la conexión del ordenador, se vería que lo que hay es un conjunto de números sin ningún tipo de significado. Estos números no sirven para nada. Nadie los ha colocado ahí de forma deliberada no constituyendo, pues, instrucciones ni datos de utilidad alguna. Así pues, la primera tarea del ordenador es la de limpieza. En lugar de estos números aleatorios, el ordenador genera un patrón mucho más homogéneo constituido por 32 bytes con todos los bits a 1 (255) seguidos de otros 32

bytes con todos los bits a cero. Pruebe de entrar la instrucción siguiente (dando tensión al ordenador y tecleándola sin anteponerle ningún número de línea):

```
FOR N=2Ø49 TO 2249 :? PEEK(N) ; " " ;:NEXT
```

y a continuación pulse RETURN. La zona de memoria que se ha utilizado es la del principio del área asignada al BASIC. Si se ejecuta la línea anterior justo después de conectar el ordenador (y no se ha utilizado número de línea alguno para dicha instrucción), se verá que no hay nada almacenado en la memoria excepto el patrón de los 255 y Ø comentado anteriormente. Tal como verá por la pantalla, el patrón que aparece es el que hemos descrito. Sin embargo, se verá también que al principio, y diseminados a lo largo de la zona estudiada, aparecen otros números. Los primeros se encuentran ahí debido a la existencia de la instrucción que hemos utilizado y los otros como consecuencia de su ejecución.

Sin embargo, el programa de inicialización tiene mucho más que hacer. La zona inicial de la RAM que va de la dirección 8 a la 818 se reserva para «uso del sistema». Esto es así debido al hecho de que las subrutinas de código máquina que llevan a cabo las acciones del BASIC necesitan almacenar algunos valores en memoria a medida que van realizando su función. Las posiciones de memoria 43 y 44, por ejemplo, contienen la dirección del primer byte de un programa BASIC. Una zona mucho mayor de memoria se reserva para el almacenamiento de los valores numéricos que posibilitan que aparezcan por pantalla textos y gráficos. Hay que reservar también otra zona de RAM para almacenar los valores que se crean al ejecutar un programa. Este es un punto que va a desarrollarse con mayor detalle en el apartado siguiente.

Variables en tablas

Los programas en BASIC hacen uso intensivo de variables, o sea de la utilización de letras para representar cifras y palabras. Cada vez que se «declara una variable» a través del empleo de una línea tal como:

```
N = 2Ø o  A$ = "SMITH"
```

el ordenador tiene que usar espacio de memoria para guardar el nombre de la variable (N , A$ o cualquier otro que se utilizase) así como el valor que se le ha asignado (tal como 2Ø o SMITH). El fragmento de memoria que se emplea para llevar una relación de las diferentes variables se conoce con el nombre de *tabla de la lista de variables* (VLT, del inglés variable list table). No ocupa una zona fija de memoria sino

que se almacena en el espacio que queda libre justo por encima de su programa. Si se añade una línea al programa en cuestión, habrá que reubicar la VLT hacia una zona más alta de memoria. Por el contrario, si se borra una línea del programa, la VLT se trasladará hacia zonas más bajas de memoria de forma que se mantenga siempre justo a continuación de la última línea de BASIC.

Como la tabla de la lista de variables puede moverse —y de hecho se mueve— a lo largo de la memoria como resultado de las modificaciones que se introduzcan a un programa, es necesario que el ordenador conozca en todo momento el origen de dicha tabla. Eso es posible gracias a dos bytes pertenecientes a la zona de memoria reservada para uso del sistema: los correspondientes a las direcciones 45 y 46. Quizá se pregunte porqué se emplean dos bytes. La razón estriba en que un byte puede almacenar números que no rebasen el valor 255. Sin embargo, si se utilizan dos bytes, uno de ellos puede usarse para guardar el número de los 256 que quepan en la dirección y almacenar el resto del número en el otro byte. Un número tal como 257, por ejemplo, es igual a 256 más un resto de 1. Podría, pues, codificarse como 1,1 , o sea un 1 almacenado en el byte reservado para los 256 y un 1 en el byte reservado a las unidades. El orden de almacenamiento de estos números es el de primero el byte bajo y a continuación el byte alto. Para reconstruir el número almacenado, basta con multiplicar el segundo byte por 256 y sumarle el contenido del primero. Así, por ejemplo, un 3 y un 8 almacenados en dos posiciones de memoria consecutivas equivaldrían, según este criterio a:

8 * 256 + 3 = 2Ø51

El mayor número que puede almacenarse de esta forma con dos bytes es 255, 255 que equivale a 255 * 256 + 255 = 65535. Esta es la razón de porqué no se pueden utilizar números muy grandes (tal como 7ØØØØ) como números de líneas en el Commodore 64 ya que el sistema operativo utiliza sólo dos bytes para su almacenamiento. De hecho (aunque por otras razones), el máximo número que puede utilizarse para este fin es 63999.

De todo ello se deduce que podemos conocer el número almacenado en las direcciones 45 y 46 a través de la expresión:

```
? PEEK(46) * 256 + PEEK(45)
```

Si emplea la anterior instrucción justo después de conectar su Commodore 64 verá que el número que obtiene es el 2Ø51. Esta es, pues, la dirección de memoria donde se almacenaría el primer byte de BASIC. Para ver todo esto en acción, teclee la línea:

```
1Ø N = 2Ø
```

y entre ?PEEK(46)*256+PEEK(45) otra vez. Si entró la primera línea

tal como se le indicó (con un espacio entre el número de línea y la N), la dirección que obtendrá será 2Ø6Ø. La tabla de la lista de variables se ha desplazado, pues, 9 bytes hacia arriba de la memoria. Este es un valor superior al número de caracteres que usted pulsó, tal como se puede apreciar fácilmente. La explicación de todo ello la obtendremos más adelante.

Muchas de las direcciones importantes que emplea el ordenador se *ubican dinámicamente* de esta forma. La «ubicación dinámica» representa un proceso por el cual el ordenador varía el lugar donde se almacenan determinados grupos de bytes conociendo en todo momento el lugar donde se encuentran a través de un par de bytes, como en el ejemplo que hemos visto. Esto tiene importantes consecuencias de cara a la forma de utilizar su ordenador. Así, por ejemplo, si se desplaza la VLT asignando nuevos valores a las posiciones 45 y 46, el ordenador no podrá encontrar el valor de sus variables. Pruebe lo siguiente: después de encontrar la dirección de la VLT y sin ejecutar el programa constituido por una sola línea (1Ø N=2Ø), pulse ?N. La respuesta será cero. ¿Por qué? Porque el programa aún no se ha ejecutado. La dirección 2Ø6Ø es donde empezará la VLT, pero no habrá ninguna VLT creada hasta que no se ejecute el programa. Esto hace que en este estado de cosas sea fácil añadir o quitar líneas de programa. Todo lo que habrá que alterar serán los números contenidos en las posiciones 45 y 46. Los valores de la VLT se colocan en su lugar sólo cuando se ejecuta el programa, creándose una nueva tabla cada vez que se teclea RUN y se pulsa RETURN a continuación. Cada vez que se crea un nueva tabla, se coloca un nuevo par de bytes en las direcciones 45 y 46. Esto explica porqué no se puede proseguir la ejecución de un programa después de editar. En lugar de ello hay que teclear RUN seguido de RETURN otra vez para crear así una nueva VLT en una nueva zona de memoria. Si se ejecuta ahora el programa indicado y se teclea ?N se obtendrá la respuesta esperada: 2Ø. Teclee ahora (sin emplear ningún número de línea) POKE 46,9 y pulse RETURN. Se ha alterado así la dirección de la VLT. La nueva dirección apunta a un lugar donde no se halla VLT alguna. Pruebe otra vez la instrucción ?N y vea el resultado que obtiene. En mi C 64 obtuve un número muy grande ya que el valor correcto de N no podía ser hallado por el ordenador. Si su Commodore se queda paralizado en el transcurso de una prueba de este tipo, desconéctelo y dele otra vez tensión. Recobrará así otra vez el control. Esto ocurre raras veces, pero recuerde que si desconecta el ordenador se pierde todo lo que hubiese almacenado en su memoria. Tome nota, de paso, del empleo de POKE para asignar un nuevo valor a una posición de memoria. El formato correcto de la instrucción es POKE A,D, donde A representa una dirección en el rango Ø-4Ø959 (el rango disponible de direcciones de RAM) para el Commodore 64 y D es el valor que se quiere colocar

en la dirección anterior, debiendo estar comprendido entre Ø y 255. Si se intenta cargar a través de POKE un valor superior a 255, se obtendrá el mensaje de error «FC ERROR».

Una ojeada a la tabla

Es hora ya de hacer algo más constructivo y echar un vistazo a lo que hay almacenado en la VLT. Cuando vayamos a efectuar nuestras averiguaciones es importante asegurarse de que el ordenador no esté afectado por el resultado de operaciones anteriores. Así pues, es aconsejable desconectar y volver a conectar el ordenador antes de llevar a cabo cada una de las pruebas. Pulsando tan sólo la tecla RESTORE no se modifican en absoluto los valores que hayan sido cargados en memoria (a través de instrucciones POKE). Es enojoso, pero es así.

Vamos, pues, a empezar a trabajar. Después de reconectar el ordenador, introduzca la línea:

```
1Ø N = 2Ø
```

otra vez y halle la dirección de la VLT a través de:

```
? PEEK (46) * 256 + PEEK (45)
```

obteniéndose la dirección 2060. Pulse ahora RUN, de forma que se carguen los valores correspondientes sobre la VLT y vea lo que se ha almacenado. Para ello usted puede utilizar la instrucción:

```
FOR X = 2Ø6Ø TO 2Ø7Ø : ? X ; " " ; PEEK(X) : NEXT
```

pulsando RETURN a continuación. Así se obtiene la lista que muestra la figura 2.1. ¿Reconoce algo en ella? Debería identificar el primer byte –igual a 78– ya que corresponde al código ASCII de la N.

```
2060    78
2061    Ø
2062    133
2063    32
2064    Ø
2065    Ø
2066    Ø
2067    88
2068    Ø
2069    140
2070    1
```

Fig. 2.1 Campo, dentro de la lista de la tabla de variables, asociado a una variable numérica real.

El siguiente byte es un cero ya que nuestra variable se llama N (y no N1 o NG, o cualquier otro nombre compuesto de dos letras). Si hubiésemos utilizado un nombre de dos letras, ambas direcciones, la 2Ø6Ø y la 2Ø61, estarían ocupadas. Los siguientes cinco bytes deben corresponder, pues, a la forma en que se codifica el número 2Ø. Llegados a este punto, no se preocupe de cómo se emplean estos bytes para representar el 2Ø. Haga un acto de fe y crea que así sucede. Pero ¿cómo sabemos que estos bytes representan el 2Ø? Es fácil: el byte de la dirección 2Ø67 es un 88, número que corresponde al código ASCII de la X y que justamente es el nombre de la variable que hemos utilizado para obtener la tabla de valores. El Commodore 64 emplea siempre estos cinco bytes para representar el valor de cualquier variable numérica, no importa que se trate de un valor pequeño como 2Ø o de uno mucho mayor, tal como 1427Ø68315, bien de una fracción, o bien de un número negativo. Ello hace posible que el almacenamiento de variables numéricas sea sencillo y posibilita a la vez que el ordenador localice sus variables. Si, por ejemplo, busca el valor de una variable llamada Y, cuando encuentra la «N» (codificada en ASCII como un 78), no necesita perder tiempo con los seis bytes siguientes (uno para la segunda letra y cinco más para el valor), sino que puede saltar al lugar donde se encuentra almacenado el siguiente nombre de variable. Si usted es curioso y no se le dan mal las matemáticas, puede encontrar en el Apéndice A el procedimiento que se emplea para convertir números cualesquiera a su expresión mediante cinco bytes. Sin embargo, para seguir este libro no necesita comprender cómo se realiza esta codificación. Sólo tiene que tener presente cómo se almacena el código en cuestión y los bytes que son necesarios.

Almacenamiento de cadenas alfanuméricas

Vamos a ver ahora cómo se almacena una cadena alfanumérica. Apague y conecte otra vez su ordenador y teclee a continuación la línea siguiente:

```
1Ø AB$ = "ESTO ES UNA CADENA"
```

Ejecute a continuación este breve programa y halle la dirección de la VLT a través (tal como antes) de las direcciones 45 y 46 de memoria. Yo obtuve el valor 2Ø78 para dicha dirección. Utilice ahora:

```
FOR X 2Ø78 TO 2Ø88 : ? X ; " " ; PEEK(X) : NEXT
```

para ver qué es lo que hay en la VLT. La figura 2.2 muestra qué es lo que aparece esta vez en pantalla. El primer valor de la tabla es un 65,

```
2078    65
2079    194
2080    17
2081    10
2082    8
2083    0
2084    0
2085    88
2086    0
2087    140
2088    2
```

Fig. 2.2 Campo de la VLT asociado a una variable de cadena.

que corresponde al valor ASCII de la B una vez se le ha sumado 128. Esta es la forma en que el Commodore 64 reconoce que se trata de una variable de cadena. Si hubiese empleado el nombre de variable A$ en lugar de AB$, el segundo número (que ocupa la dirección 2679) habría sido un 128 y no un cero. Cuando se utiliza una variable numérica, el segundo código ASCII del nombre de la misma será un Ø o bien uno cualquiera de los códigos numéricos ASCII, pero nunca será mayor que 127. Esta es una buena idea que se les ocurrió a los diseñadores.

Vamos a analizar el resto de trozo de tabla asociado a esta cadena. No parece que se corresponda con el código ASCII de las letras que componen dicha cadena ¿no es así? De hecho se utilizan sólo siete bytes en total, igual que para las variables numéricas. La clave de lo que ocurre aparece cuando echamos una ojeada a los números. El que sigue al código de la B (igual a 194, ya que se ha sumado 128 al código ASCII correspondiente), es un 17. Pero 17 es igual al número de caracteres de la cadena. Si usted cuenta las letras y los espacios, verá que, efectivamente, es así. Los dos bytes siguientes son un 1Ø y un 8. Combinándolos de la forma que ya hemos visto, obtenemos una dirección: 8 * 256 + 1Ø = 2Ø58. El punto siguiente consiste, lógicamente, en ver a través de PEEK (2Ø58) qué es lo que hay almacenado en esta dirección. Obtenemos, evidentemente, un 45 que es código ASCII de la «E». 2Ø58 es, pues, la dirección del primer byte de la cadena.

Combinemos ahora todo esto. El Commodore reserva en su VLT un campo de siete bytes para cada cadena. De estos siete bytes, los dos primeros corresponden al nombre de la variable de cadena, siendo el valor del segundo superior o igual siempre a 128. Cuando se utiliza un nombre de dos caracteres, se suma 128 al código ASCII de la segunda letra. Ello permite al ordenador distinguir una variable nu-

mérica de una variable de cadena. Los cinco bytes siguientes contienen la longitud de la cadena y la dirección de su primer byte. Así, sólo son necesarios tres bytes para saber donde se encuentra una cadena: uno para su longitud (ninguna cadena tendrá más de 255 caracteres. De hecho, no pueden tener más de 24Ø). A continuación se utilizan dos bytes para la dirección, con lo que quedan dos bytes del campo de siete que no se emplean, como no sea de separadores. La conveniencia de asignar campos de igual longitud en la VLT para cadenas y para valores numéricos compensa con creces la inutilización de los dos últimos bytes del campo asignado a las cadenas.

En este ejemplo, la cadena se ha almacenado en una dirección más baja que la correspondiente a la VLT: corresponde a la zona de memoria reservada para el texto en BASIC. Esta es la parte de la memoria que contiene el programa y como ahí es donde se coloca el código ASCII de los caracteres de la cadena al entrar la fuente, no parece descabellado aprovecharlos allí mismo donde se encuentran. Los números deben transferirse a la VLT ya que no se almacenan en forma de códigos ASCII. La cuestión que cabe plantearse ahora es: ¿qué ocurre con las cadenas que no se encuentran en el programa? Apague y conecte de nuevo su ordenador y teclee a continuación:

```
1Ø A$ = "AB" : B$ = "CD" : C$ = A$ + B$
```

Ejecute a continuación este programa y verá como la VLT es más larga de lo que lo era anteriormente. Esta vez tendrá que ir a mirar en las direcciones de memoria comprendidas entre la 2Ø8Ø y la 21ØØ. Encontrará ahí las áreas correspondientes a A$ y B$ tal como usted esperaba, proporcionado unas direcciones ubicadas en la zona de memoria de programa, tal como muestra la figura 2.3. Sin embargo, la variable tiene en su campo de direcciones los bytes 252 y 129. Corresponderá, pues, a esta cadena la dirección 4Ø956 (recuérdese que es 159 * 256 + 151). Echemos un vistazo a estas direcciones. Si pulsa:

```
FOR X = 4Ø956 TO 4Ø959 : ?X ; " "; PEEK(X) ; " " ; CHR$(PEEK(X)) : NEXT
```

verá lo que esconden. Los códigos ASCII de las letras ABCD están almacenados ahí: el empleo de CHR$ los identifica claramente.

Los enteros

Hemos visto cómo se almacenaban valores numéricos y cadenas, pero no debemos olvidar que el Commodore 64 permite utilizar dos tipos de números. Uno de ellos corresponde al grupo de los que se denominan «números reales». El segundo grupo es el de los «enteros». Una variable asociada a un valor real utiliza una letra, un par de letras o bien una letra y un dígito para su representación (tal como en

```
2080    65
2081    128
2082    2
2083    9
2084    8
2085    0
2086    0
2087    66
2088    128
2089    2
2090    17
2091    8
2092    0
2093    0
2094    67
2095    128
2096    4
2097    252
2098    159
2099    0
2100    0
```

Fig. 2.3 Campo de la VLT asociado a una cadena alfanumérica no contenida en la zona de memoria donde se encuentra el programa.

A, AB o A2). Un número real se codifica en la forma que hemos visto, utilizándose un total de siete bytes. De ellos dos son para el nombre y cinco para el valor. Un número real puede ser positivo, negativo o bien fraccionario y su valor puede oscilar de $1\emptyset^{38}$ a $1\emptyset^{-39}$, aproximadamente. Los números enteros, por su parte, no posee parte fraccionaria alguna y pueden adoptar los valores comprendidos entre −32768 y +32767. Es hora ya de ver como se guardan estos números en la VLT.

Empiece, tal como es habitual, desconectando y conectando otra vez el ordenador con el fin de inicializar la memoria. Entre ahora la línea:

```
1Ø A% = 15 : B% = 3ØØ
```

y a continuación, ejecútela. Halle la ubicación de la VLT, tal como es habitual, mirando en las direcciones 45 y 46 a través de PEEK. En mi Commodore 64 obtuve la dirección 2Ø68. Ahora pulse:

```
FOR X = 2Ø68 TO 2Ø85 : PRINT X ; " " ; PEEK(X) : NEXT
```

seguido de RETURN. Obtendrá así la secuencia que se muestra en la figura 2.4.

```
2068    193
2069    128
2070    0
2071    15
2072    0
2073    0
2074    0
2075    194
2076    128
2077    1
2078    44
2079    0
2080    0
2081    0
2082    88
2083    0
2084    140
```

Fig. 2.4 Campo de la VLT asociado a dos números enteros.

No es la misma que obtuvimos con los números reales en la figura 2.1. Para empezar, algo ha ocurrido con el primer byte, el que representa el nombre de variable A%. Este byte ostenta el valor 193 que corresponde a 65 + 128: es el código de la A al que se le ha sumado 128. El segundo byte es un 128. Todo ello sugiere que cuando se crea una variable numérica entera, la máquina suma 128 a los dos caracteres que componen su nombre. ¿Puede ver la regla que se ha seguido hasta ahora? Para un número real, las letras del nombre de la variable corresponden a códigos ASCII. Para cadenas alfanuméricas, el segundo código del nombre es un 128 o bien un código ASCII al que se le ha sumado dicho valor. Para los números enteros se le suma el 128 también al primer carácter. Es así como distingue la máquina los nombres de los diferentes tipos de variables. Los símbolos $ y % *no* se almacenan en memoria.

Ahora debemos ver cómo se almacenan los valores enteros. Los bytes que siguen a los dos ligados al «nombre de variable» tienen los valores Ø, 15, Ø, Ø y Ø, lo que sugiere que el número 15 se ha almacenado inalterado. Es un criterio distinto al adoptado para llevar a cabo el almacenamiento de un valor real. Sin embargo, el número 3ØØ se almacena como 1, 44. ¿Corresponde a un sistema de almacenamiento a base de dos bytes? Efectivamente: 256 * 1 + 44 = 3ØØ. El número se ha almacenado en forma de dos bytes con el más significativo primero, al revés de lo que ocurre con los números de línea o bien con los números correspondientes a las direcciones de memoria. El espa-

cio reservado para las variables enteras en la VLT sigue siendo de siete bytes, no utilizándose en este caso tres de ellos.

¿Cómo explica esto las características de los enteros? En primer lugar, puede justificarse el rango que pueden adoptar estos números. Si utilizamos sólo dos bytes para su almacenamiento, no puede guardarse un valor mayor que el que corresponde a tener un 255 en cada uno de ellos. Este valor sería 255 * 256 + 255, lo que equivale a 65536. Sabemos que el rango que pueden adoptar los enteros va de −32768 a +32767 y sumando ambos números en valor absoluto obtenemos 65535. Así pues, el Commodore 64, en lugar de utilizar el rango Ø ÷ 65535, emplea un intervalo más adecuado como es el que va de −32768 a +32767. Las razones para ello se exponen con mayor detalle en el Apéndice A. El empleo de números enteros tiene varios efectos secundarios. Uno de ellos es que los enteros tienen una precisión perfecta. Cuando declara A% = 17412, éste es exactamente el número que se almacena y no 17411.9999999. Un número «real» constituye casi siempre una aproximación y cuando se emplean números de este tipo hay que llevar a cabo una serie de «redondeos». Probablemente se ha encontrado con calculadoras que sacan como solución a un determinado cálculo 3.9999999 en lugar de 4.Ø. Ello es debido a las aproximaciones que se realizan a la hora de almacenar números reales. Algunas calculadoras redondearán el número a 4 y otras no. Si trabaja con enteros en el Commodore 64 no tiene porqué preocuparse por estos problemas. La otra ventaja de utilizar enteros es la velocidad. Como un número entero se almacena en forma de dos bytes, el ordenador puede operar con él mucho más rápidamente que con un número «real» que requiere cinco bytes y una decodificación mucho más elaborada. Si quiere velocidad en las operaciones, utilice números enteros (probablemente ya conocía usted este hecho).

Almacenamiento de los programas

Es hora de ver cómo se almacena un programa en la memoria de su Commodore 64. Tal como hemos venido haciendo hasta ahora, emplearemos PEEK sobre distintas zonas de memoria para ver qué es lo que se encuentra almacenado allí. Sin embargo, lo primero que hemos de conocer es donde están guardados los dos bytes que forman la dirección del lugar donde está almacenado el programa. La dirección de estos dos bytes es la 43 y la 44.

Podemos ver, pues, cómo está almacenado el programa en memoria. Teclee el programa de la figura 2.5, pero no lo ejecute. Pulse ahora:

```
? PEEK(44) * 256 + PEEK(43)
```

```
10 A=10
20 PRINT A
30 C$="CBM 64"
```

Fig. 2.5 Sencillo programa en BASIC.

1Ø	8	1Ø	Ø	65	178	49	48	Ø	18	8
2Ø	Ø	153	32	65	Ø	34	8	3Ø	Ø	67
36	178	34	67	77	66	32	54	52	34	Ø

Fig. 2.6 . Lista de los bytes que representan el programa en memoria.

y obtendrá la dirección de inicio del programa. Con este ejemplo, obtuve en mi Commodore 64 la dirección 2Ø49. Ahora, si utiliza el bucle de siempre para escribir los valores obtenidos a través de PEEK en las posiciones que se encuentran a partir de esta dirección, obtendrá la lista de la figura 2.6. A primera vista parece una ristra de números sin ningún sentido, pero si los analiza con mayor detenimiento, podrá apreciar en ellos un cierto patrón. Como ya es costumbre, los códigos ASCII nos proporcionan alguna pista. En la dirección 2Ø53, por ejemplo, puede verse el número 65 que es el código ASCII de la «A». Como sabemos que la primera línea era «A=1Ø», seguimos buscando el resto de la instrucción. El 1Ø puede reconocerse como un 49 (código ASCII del «1») seguido de un 48 (código ASCII del «Ø», de forma que el 178 debe representar al signo «=». Pero éste *no* es el código ASCII del «=», sino que es uno de los «tokens» que se mencionaron en el capítulo 1. Es un token debido a que el ordenador debe ejecutar una determinada acción al llegar a él, no sólo almacenarlo. El Ø en la posición 2Ø57 marca el final de esta línea.

Debemos analizar ahora los cuatro primeros bytes. Los dos primeros son, tal como quizá ya sospecha, una dirección. El 1Ø, 8 constituye la dirección 8 * 256 + 1Ø, lo que equivale a 2Ø58. ¿Qué es esta dirección? Es la dirección del primer byte de la línea siguiente. Esta es la forma en que el sistema operativo del Commodore 64 puede ir llegando a cada línea del programa en la secuencia correcta, sin importar el orden que se haya seguido a la hora de entrarlas. El misterio final se resuelve fácilmente. Los bytes tercero y cuarto de cada línea siguen la secuencia 1Ø, 2Ø, 3Ø y corresponden, pues, a los números de línea. Hay dos bytes reservados para estos números de línea ya que se quiere trabajar con números mayores que 255. Para números

de línea menores que 256, el segundo de estos bytes (el más significativo) no se utiliza.

Vamos a estudiar las otras líneas, tal como están almacenadas en memoria. Ya nos habíamos encontrado anteriormente el token de PRINT (el 153). El resto de la línea debería serle ya familiar. La única novedad se encuentra al final del programa. La última línea termina con un Ø, como es habitual, pero a continuación, donde debería estar la dirección de la línea siguiente, hay otro par de ceros. Esta es la marca que utiliza el ordenador como señal de fin de programa.

Podemos llevar a cabo algunos cambios interesantes en un programa como éste. Supongamos, por ejemplo, que modificamos mediante la instrucción POKE las posiciones de memoria que contienen los números de línea. Si teclea:

```
POKE 2Ø6Ø,1Ø : POKE 2Ø68,1Ø
```

seguido de RETURN, colocará un 1Ø como número de las líneas 2Ø y 3Ø. Teclee ahora LIST y vea el resultado. Aparece un programa con todos los números de línea igual a 1Ø. Al contrario de lo que pueda suponer, este programa puede ser ejecutado normalmente. Piense que el hecho de poderlo ejecutar depende de que la dirección de la «próxima línea» sea la correcta, no de cómo estén etiquetadas estas líneas. Sin embargo, un programa que haya sido modificado de esta forma no es, desde luego, normal. Intente, por ejemplo, cambiar, a través del editor, la A de la segunda línea por una B, de forma que dicha línea se lea PRINT B. Ejecute el programa y haga un listado. Verá que la primera línea ha desaparecido y que ahora la primera línea es PRINT A y PRINT B la segunda. A pesar de todo eso, usted puede grabar un programa que se haya visto alterado de esta forma y recuperarlo después normalmente. Ahí está el germen de un procedimiento mediante el cual puede hacerse que un programa sea francamente difícil de alterar.

Ejecución de un programa

Ahora que hemos visto la forma en que se codifica y almacena un programa en la memoria del Commodore 64, vamos a dar una idea de cómo es ejecutado. Esta acción la lleva a cabo la parte más complicada del sistema operativo a la que, como es lógico, se suministra una dirección. Como adivinará, esta dirección irá a buscarla dicho programa a las posiciones 43 y 44 de memoria. Vamos a analizar, prescindiendo de los detalles, las acciones que se van llevando a cabo. En las primeras posiciones de la zona reservada al BASIC, el programa ligado al RUN leerá los dos primeros bytes y los almacenará temporalmente. Estos bytes serán empleados en lugar de la «dirección de ini-

cio del BASIC» cuando se vaya a ejecutar la línea siguiente. También los bytes asociados a los números de línea son leídos y almacenados. ¿Por qué? Para que en el caso de producirse un error sintáctico en una determinada línea, el ordenador pueda sacar el mensaje: 'SN ERROR IN 1Ø'(del inglés error sintáctico en la línea 1Ø) en lugar del mensaje: 'SN ERROR SOMEWHERE' (del inglés error sintáctico en algún lugar). El byte siguiente corresponde a un código ASCII, considerándolo el ordenador como perteneciente a un nombre de variable. Antiguamente había que utilizar la palabra LET para «declarar una variable». Había un token asociado a LET, empleándose, sin embargo, aún dicho token en los ordenadores modernos. La diferencia estriba en que ahora el token se genera automáticamente cuando sigue una letra a un número de línea o bien a un par de puntos. Si se emplea LET, se genera el mismo token.

Después del token de asignación, el token del símbolo «=» hace que se lleve a cabo una determinada subrutina. Esta crea un campo en la lista de la tabla de variables en el primer lugar que encuentre disponible, colocando en dicho campo el código de la A. La siguiente dirección de la VLT contendrá un byte en blanco ya que el nombre de la variable contiene sólo un carácter. Se lee a continuación el número 1Ø, valor que se pasa a un formato especial binario, tal como se describe en el Apéndice A. Se coloca también este conjunto de bytes en el campo de la VLT reservado a la variable A. Se lee a continuación el siguiente byte del programa. Es un Ø, con lo que la dirección de la línea siguiente (que se leyó en primer lugar) se carga en el microprocesador. El tipo de acciones que se han analizado en detalle para la línea 1Ø se repiten para la línea 2Ø. Sin embargo, en este caso hay más trabajo que realizar una vez que se ha leído el token de acción. Como es el asociado a PRINT, hay que llamar a la subrutina ligada a dicha instrucción. Dicha subrutina identificará en primer lugar la dirección de la primera posición libre de pantalla. Esto se lleva a cabo manteniendo dicha dirección sobre un par de bytes de RAM, con lo que sólo hay que leer estos bytes para obtener la dirección deseada. A continuación se obtiene el valor de la A en la VLT, convirtiéndolo otra vez a su expresión ASCII. Estos caracteres se colocan, uno a uno, en la zona de memoria ligada a la pantalla. Eso hace que aparezcan estos caracteres en pantalla, por acción de otra subrutina. Una vez más, el cero de final de líneas hace que se emplee la dirección de la línea siguiente. Sin embargo, al final de la tercera línea, el «número de la próxima línea» es un cero, con lo cual el programa finaliza. El ordenador vuelve entonces a su estado de espera, preparado para realizar lo que usted le indique.

Todo esto no es tan sencillo como puede parecer por la descripción que hemos hecho, aunque sí han salido a la luz los puntos más importantes. Lo principal es ver que hay gran cantidad de acciones

que llevar a cabo y hay que ir haciéndolo todo paso a paso y de forma secuencial. Lo que hace lento el BASIC es que cada token fuerza la ejecución de una subrutina a la que hay que identificar previamente. Así, por ejemplo, si se tiene un programa que consista en un bucle del estilo de:

```
1Ø FOR N = 1 TO 5Ø
2Ø PRINT N
3Ø NEXT
```

la acción de leer el token asociado a PRINT (un 153) e identificar la subrutina apropiada, se llevará a cabo cincuenta veces. No hay forma de hacer que se localice a la subrutina una sola vez y se la utilice cincuenta veces. El tipo de BASIC del que dispone su Commodore 64 es un BASIC *interpretado.* Esto significa que cada instrucción debe ser analizada cuando el ordenador llega a ella. Si eso representa encontrar cincuenta veces la subrutina asociada a PRINT, tanto da. La alternativa a este esquema lo constituye la *compilación,* proceso en el cual se convierte a código máquina el programa entero antes de ser ejecutado. La compilación es llevada a cabo por otro programa denominado *compilador.* La utilización de un compilador para un programa BASIC incrementa notablemente la velocidad de este programa, aunque hace que sea más difícil de editar ya que lo convierte a un formato distinto. ¡No se puede tener todo a la vez!

3. El microprocesador

En este capítulo empezaremos a estudiar el microprocesador 65Ø2 que lleva incorporado el Commodore 64. El microprocesador —o MPU— es, como bien recordará, la parte «activa» del ordenador, en contraposición al sistema de almacenamiento (memoria) o bien al sistema de entrada/salida (ports). Así pues, lo que haga el microprocesador decidirá el comportamiento del ordenador en su conjunto.

La MPU consiste en un conjunto de unidades de memoria para almacenar números, aunque posee una organización notablemente completa y compleja. A través de circuitos a los que se denomina muy propiamente *puertas*, se controla la forma en que se transfieren los bytes entre las distintas zonas de la memoria propia de la MPU así como las acciones que se llevan a cabo sobre estos bytes tal como pueden ser sumas, restas, operaciones lógicas, etc. Cada una de estas acciones está programada. No ocurre nada a menos que se encuentre presente un byte de instrucción en forma de señales de unos o ceros en cada una de las 8 líneas del bus de datos, utilizándose estos bytes de instrucción para controlar las puertas que se encuentran en el interior de la MPU. Lo que hace tan útil al conjunto es que debido al hecho de estar las instrucciones del programa en forma de señales eléctricas sobre estas ocho líneas, se las puede cambiar muy rápidamente. La velocidad viene marcada por otro circuito, denominado «reloj generador de impulso» o bien, de forma más breve, «reloj». La velocidad que se ha elegido para el reloj del Commodore 64 es muy rápida, de forma que pueden ejecutarse del orden de un millón de operaciones por segundo.

El código máquina

El programa de la MPU, tal como hemos visto, está formado por códigos numéricos constituidos cada uno de ellos por un número entre Ø y 255 (asociados, pues, a bytes). Algunos de estos números pueden ser bytes de instrucciones que provoquen alguna acción por parte de la MPU. Otros pueden ser bytes de datos, tal como los números a sumar, almacenar, decalar..., o pueden ser también códigos ASCII asociados a cualquier carácter. La MPU no puede decir qué es cada

uno de estos bytes. Hace sólo lo que se le indica. Es tarea del programador ordenar los números y colocarlos en el orden correcto.

El orden correcto, por lo que respecta a la MPU, es muy sencillo. El primer byte que se le proporciona después de conectar el ordenador o bien después de completar una instrucción, es interpretado como un byte de instrucción. Muchas de las instrucciones del 65Ø2 consisten en un solo byte y no necesitan datos. Otras, por el contrario, deben ir acompañadas por uno o dos bytes de datos. Cuando la MPU lee un byte de instrucción, lo analiza para ver si la instrucción a la que representa debe ir acompañada por uno o dos de estos bytes de datos. Si, por ejemplo, el byte de instrucción leído es uno que deba ir acompañado por dos bytes de datos, la MPU tratará a los dos bytes que sigan al de instrucción como datos. Este proceder de la MPU es completamente automático y está implementado en la misma MPU. El problema es que el código máquina debe seguir estas mismas reglas para que un programa sea correcto, y debe hacerlo sin ningún tipo de error. Si se proporciona a la MPU un byte de instrucción cuando espera uno de datos o bien un byte de datos cuando esperaba uno de instrucción, es evidente que surgirán problemas. Y casi siempre eso significa acabar en un bucle sin fin y con la pantalla y el teclado inactivos. Incluso la combinación de las teclas STOP y RESTORE puede demostrarse inútil a la hora de sacar al Commodore 64 de un bucle de este tipo, siendo necesario, entonces, quitar y volver a dar tensión al ordenador. Se pierde entonces cualquier programa que se tuviese almacenado en la memoria de forma que, antes de ejecutarlo, es de vital importancia guardar en cinta cualquier programa en código máquina o bien cualquier programa BASIC en general que desencadene cualquier acción de código máquina.

Lo que quiero poner de manifiesto es que la programación en código máquina es aburrida. No es necesariamente difícil (se trata sólo de definir una serie de sencillas instrucciones para una máquina que en sí es muy simple), aunque sí es duro, a menudo, recordar todos los detalles necesarios. Cuando programamos en BASIC, los mensajes de error de la máquina se nos demuestran de gran utilidad y nos ayudan de forma notable a detectar los errores que se produzcan. Cuando se emplea el código máquina, hay que fiarse sólo de uno mismo y cada cual debe arreglárselas para detectar y corregir los errores que cometa. Un programa denominado *assembler* (ensamblador en castellano), a pesar de ser de las pocas ayudas de que se dispone en esta área, es de considerable utilidad. Volveremos más tarde sobre este punto. Mientras tanto, la mejor forma de aprender algo sobre el código máquina es escribir y programar en él, utilizarlo y cometer nuestros propios errores. Empecemos, pues, a ver como hacer todo esto. Para ello comenzaremos por ver las formas de escribir los números que constituyen los bytes de un programa de código máquina.

El código binario, decimal y hexadecimal

Un programa en código máquina consiste en un conjunto de códigos numéricos. Como cada uno de estos códigos no es más que la forma de representar los unos y los ceros de un byte, corresponderá a un número comprendido entre Ø y 255 si utilizamos la base diez habitual (o sea la escala decimal) para expresarlo. El programa es totalmente inútil si no se le carga en la memoria del Commodore 64 ya que la MPU es un elemento extremadamente rápido y la única forma de suministrarle los bytes a la velocidad que los lee es almacenándolos en memoria y dejando libre a la MPU para que los coja a su propio ritmo. Es imposible que usted teclee números a la velocidad suficiente para satisfacer a la MPU. Incluso procedimientos tales como la cinta o el disco magnéticos resultan demasiado lentos.

Cargar bytes en memoria es, pues, parte esencial dentro del proceso de hacer que funcione un programa en código máquina. Veremos más tarde con mayor detalle procedimientos para llevar a cabo esta tarea. Podrían cargarse en memoria programas muy cortos y sencillos a través del procedimiento más rudimentario: el empleo de ocho interruptores. Cada interruptor podría hacerse que diera un cero o un uno eléctricos a la salida, utilizándose un pulsador para hacer que se almacene en memoria el número que representase el conjunto, para pasar a seleccionar a continuación la dirección siguiente de memoria. Programar así sería terriblemente tedioso. Por otra parte, trabajando con números binarios pronto se vuelve uno bizco. Ahora que se dispone de ordenadores parece razonable utilizar el propio ordenador para cargar los números en memoria, para lo cual hay que ir a buscar una escala de numeración adecuada.

Qué es lo que se entiende por una escala de numeración adecuada es algo que depende de cómo se entran estos números y de cuánta programación en código máquina vaya a realizarse. El Commodore 64 contiene subrutinas que convierten los números binarios contenidos en la memoria al formato de números decimales que aparece en pantalla y puede, también, efectuar la tarea inversa. Cuando usted utiliza PEEK, la dirección a la cual quiere acceder, puede expresarse en base diez, obteniéndose como resultado de dicha instrucción otro número decimal comprendido, esta vez entre Ø y 255. Análogamente, cuando utiliza POKE, puede expresar tanto la dirección como el valor a cargar en base diez.

Sin embargo, los programadores serios de código máquina no encuentran conveniente el empleo del código decimal. Un número en base diez tanto puede tener un dígito (tal como el 4), como dos (el 17) o tres (el 143). Un código mucho más adecuado es el llamado código *hexa* (abreviación de hexadecimal). Todos los números de un solo

Hexa	Decimal	Hexa	Decimal
ø	ø	C	12
1	1	D	13
2	2	E	14
3	3	F	15
4	4	luego	
5	5	1ø	16
6	6	11	17
7	7	hasta	
8	8	2ø	32
9	9	21	33
A	1ø	22	34
B	11	etc.	

Fig. 3.1 Los dígitos hexa junto con los valores decimales asociados.

byte pueden representarse a través de dos dígitos hexa. Además, los programadores de código máquina escriben sus programas en lo que se denomina *lenguaje assembler* (lenguaje ensamblador, en castellano). Éste emplea palabras que son una versión abreviada de los nombres de las instrucciones de la MPU. Son los programas denominados *assemblers* (programas ensambladores, en castellano) quienes convierten estas palabras a los códigos binarios correspondientes. Prácticamente todos los assemblers muestran estos códigos en pantalla en formato hexa, en lugar del decimal. Además, cuando se entran los números asociados a los datos hay que emplear también este código hexadecimal. «Hexadecimal» significa base dieciséis y la razón de utilizar tan extensamente el código que lleva este nombre es que es el que se adecua de forma más natural a la representación de bytes binarios. Cuatro bits (medio byte) representan a los números comprendidos en el rango ø - 15 de nuestra escala normal de numeración. Este es el rango asociado a un dígito hexa (véase fig. 3.1). Como no disponemos de símbolos para los dígitos mayores que 9, hay que recurrir a las letras A, B, C, D, E y F para complementar a los dígitos ø .. 9 en la escala hexa. La ventaja de la utilización de este código estriba en que con él se puede representar un byte con un número de dos dígitos y una dirección completa con un número de sólo cuatro dígitos. Los códigos numéricos que se emplean como instrucciones han sido diseñados en código hexa, de forma que pueden apreciarse mejor las similitudes existentes entre instrucciones parecidas. Por ejemplo, puede verse que un conjunto de instrucciones de función semejante, empiezan todas con el mismo dígito cuando se representa su código de instrucción en hexa. En decimal no podría apreciarse

esta relación. Además, es mucho más fácil escribir el número binario que emplea el ordenador a partir de la versión hexa del mismo. El empleo del assembler del Commodore 64 y de los programas del monitor (tal como el excelente MIKRO assembler) exige una cierta familiaridad con el código hexa. Además, los libros de información acerca del microprocesador 65Ø2, están escritos suponiendo que el lector conoce este sistema de numeración. Así pues, parece que hay que dominarlo todo lo bien que se pueda.

La escala hexadecimal

La escala hexadecimal consta de dieciséis dígitos. Empieza, tal como es habitual, con el cero y va subiendo hasta el 9. Sin embargo, la siguiente cifra no es el 1Ø ya que ello significaría uno por dieciséis más cero unidades. Como no disponemos de dígitos para los valores superiores a 9, se utilizan las letras de la A a la F. El número que escribimos como 1Ø (diez) en el sistema decimal, se escribe como ØA en hexa, el once como ØB, el doce como ØC, y así hasta el quince, al que corresponde el ØF. El cero no es necesario aunque los programadores tienen la costumbre de escribir un byte de datos con dos dígitos y una dirección con cuatro, incluso en el caso de no ser necesarios tantos. El número que sigue al ØF es el 1Ø (dieciséis en el sistema decimal) repitiéndose la escala que hemos visto hasta el 1F (treinta y uno), al que sigue el 2Ø. El valor máximo almacenable en un byte (255 en base diez), se escribe FF en hexa. Cuando se escriben valores hexadecimales es costumbre marcarlos de alguna manera de forma que no se confundan con números pertenecientes al sistema decimal. No hay muchas posibilidades de confundir un número tal como el 3E con un número decimal, aunque un 26 tanto puede representar un número hexa como uno en base diez. El convenio que siguen los programadores del 65Ø2 es el de utilizar el signo del dólar ($) para identificar un número hexa, colocándolo inmediatamente antes del número. Así, por ejemplo, el número $47, significa 47 hexa y 47 a secas equivale al 47 decimal. A la hora de escribir números hexa para un programa de 65Ø2 es recomendable seguir este convenio.

La gran ventaja del código hexa estriba en lo íntimamente ligado que se encuentra al código binario. Si mira a la tabla de conversión hexadecimal-binario de la figura 3.2, verá que $9 equivale a 1ØØ1 en binario y que $F es 1111. Así pues, el número hexa $9F equivale al 1ØØ11111 binario: basta con escribir los dígitos binarios que correspondan a los diferentes dígitos hexadecimales. La conversión inversa es igual de fácil: sólo hay que ir agrupando los dígitos binarios de cuatro en cuatro, empezando por el menos significativo (el que se encuentra más a la derecha del número) y convertir entonces cada

Hexa	Binario	Hexa	Binario
ø	øøøø	8	1øøø
1	øøø1	9	1øø1
2	øø1ø	A	1ø1ø
3	øø11	B	1ø11
4	ø1øø	C	11øø
5	ø1ø1	D	11ø1
6	ø11ø	E	111ø
7	ø111	F	1111

Fig. 3.2 Los dígitos hexa junto con los valores binarios asociados.

uno de los grupos en su correspondiente dígito hexa. La figura 3.3 muestra ejemplos de conversiones en ambos sentidos, para que vea lo fácil que es.

El Commodore 64 no tiene ningún programa incorporado para realizar la conversión entre números decimales y números hexa, siendo un programa en BASIC la forma más adecuada de efectuar dicha conversión. La figura 3.4 muestra un programa de conversión de números decimales a hexadecimales que construye el número hexa sobre la cadena alfanumérica H$. La conversión se inicia dividiendo el número a convertir por 4ø96 y cogiendo la parte entera del resultado. Si el número en base diez es menor que 4ø96, el resultado será ø, lo que correspondrá al primer dígito hexa. La conversión de los códigos hexa se realiza a través del valor de los códigos ASCII ya que existe una sencilla relación entre los códigos ASCII correspondientes a los dígitos del ø al 9 y el valor que representan los propios dígitos. Si suma 48 a un dígito, obtendrá el código ASCII correspondiente (así, 48 + 1 = 49 que es el código del «1»). Este código ASCII puede ser, entonces, incorporado a una cadena alfanumérica. Surge una pequeña complicación cuando llegamos al diez que corresponde en hexa al dígito A, siendo 65 el código ASCII de la A (igual, pues, a 1ø más 55). El programa de conversión debe sumar, pues, 48 a los dígitos menores o iguales que 9 y 55 a los dígitos pertenecientes al rango 1ø ÷ 15 para efectuar correctamente la conversión. No cuesta mucho hacer todo esto en un programa BASIC. El paso siguiente consiste en tomar el resto de la división por 4ø96 y dividirlo a su vez por 256 (que es 4ø96/16). Se pasa, entonces, la parte entera del resultado a su expresión en forma de cadena alfanumérica y se va repitiendo el proceso hasta que no quede nada. Si se quiere hacer esta subrutina supereficiente, utilícense variables enteras para todos los números enteros que aparezcan.

Conversión: Hexa a Binario

Ejemplo: 2 CH ..... 2H es ØØ1Ø binario
CH es 11ØØ binario

Luego 2CH es ØØ1Ø11ØØ en binario (byte a convertir)

Ejemplo: 4A7FH ..... 4H es Ø1ØØ binario
AH es 1Ø1Ø binario
7H es Ø111 binario
FH es 1111 binario

Luego 4A7FH es Ø1ØØ1Ø1ØØ1111111 en binario (una dirección)

Conversión: Binario a Hexa

Ejemplo: Ø11Ø1Ø11 ..... Ø11Ø es 6H
1Ø11 es BH

Luego Ø11Ø1Ø11 es 6BH

Ejemplo: 1Ø11Ø1ØØ1ØØ1Ø ..... vea que no constituye un número entero de bytes

Agrupándolos de cuatro en cuatro, empezando por el LSB:

ØØ1Ø es 2H
1ØØ1 es 9H
11Ø1 es DH y
el resto 1Ø es 2, lo que hace 2D92H

Fig. 3.3 Conversión entre valores hexa y binarios.

La conversión de hexa a decimal puede realizarse de una forma totalmente parecida, mostrándose en la figura 3.5 un programa que lleva a cabo esta función. Este programa se basa en convertir el código ASCII de cada dígito a su valor correspondiente, multiplicándolo a continuación por el factor adecuado (los números 1, 16, 256 y 4Ø96

```
10 PRINT"■":PRINT"ENTRE, POR FAVOR, UN N
UMERO DECIMAL":INPUT D
20 IFD>65535THENPRINT"ES DEMASIADO GRAND
E - EL LIMITE ES 65535":GOSUB300:GOTO10
30 IFD<1THENPRINT"NO VALEN LOS NUMEROS M
ENORES QUE 1":GOSUB300:GOTO10
40 IFD<>INT(D)THENPRINT"NO VALEN LAS FRA
CCIONES":GOSUB300:GOTO10
50 F=4096:H$=""
60 Y=INT(D/F)
70 GOSUB200
80 D=D-Y*F:F=INT(F/16)
90 IF F<1THEN110
100 GOTO60
110 IF LEFT$(H$,2)="00"THEN H$=RIGHT$(H$
,2)
120 H$="$"+H$
130 PRINT"EL EQUIVALENTE HEXA ES ";H$
140 END
200 IF Y<=9THEN H$=H$+CHR$(Y+48)
210 IF Y>9THEN H$=H$+CHR$(Y+55)
220 RETURN
300 FORN=1TO1000:NEXT:RETURN
```

Fig. 3.4 Programa de conversión de valores decimales a hexa.

```
10 PRINT"■":Y=1:D=0
20 PRINT"ENTRE,POR FAVOR,UN NUMERO HEXA"
25 INPUTH$
30 L=LEN(H$):FORN=0TOL-1
40 GOSUB200:NEXT
50 PRINT"EL EQUIVALENTE DECIMAL ES ";D
100 END
200 P$=MID$(H$,L-N,1):A=ASC(P$)
210 IFA<48ORA>102THENGOSUB300:GOTO280
220 IFA<65ANDA>57THENGOSUB300:GOTO280
230 IFA<=97ANDA>70THENGOSUB300:GOTO280
240 IFA<=57 THEN Q=A-48
250 IF A>=65 THEN Q=A-55
260 IF A>=97 THEN Q=A-87
270 D=D+Q*Y:Y=Y*16
280 RETURN
300 PRINT"VALOR HEXA INCORRECTO,";
310 PRINT"INTENTELO DE NUEVO"
500 FORX=1TO1000:NEXT:GOTO10:RETURN
```

Fig. 3.5 Programa de conversión de valores hexa a decimales.

para números hexa de hasta 4 dígitos). Los números obtenidos de esta forma se van sumando hasta obtener el total D. Una vez más, el programa BASIC correspondiente es francamente sencillo. Esta es la razón de porqué la sección de «pagamos por cada programa publicado» de las revistas especializadas está llena de programas de conversión decimal-hexa para cada nuevo ordenador que vaya apareciendo en el mercado. Si quiere realizar estas conversiones cuando no tenga a mano un Commodore 64, en el Apéndice B encontrará una detallada descripción de los procedimientos de conversión descritos en líneas anteriores.

Suponiendo, de forma razonable, que no quiera abordar el diseño de un programa ensamblador (assembler en inglés), ¿qué es lo que hay que hacer para construir programas en código máquina? La respuesta es: diseñar el programa en lenguaje ensamblador (que es la forma más fácil de realizar programas en código máquina) y convertirlo posteriormente a código hexa. Esta conversión implica la consulta de una serie de tablas (denominadas el *repertorio de instrucciones)* en busca del número hexa que represente a cada instrucción. Todos los fabricantes de microprocesadores proporcionan los repertorios de instrucciones de los micros que fabrican. Así, Mostek, que diseñó el 65Ø2, proporciona el que corresponde a este microprocesador. Para ayudarle, en el Apéndice C se ha incluido una breve guía del repertorio de instrucciones del 65Ø2. Por el momento, olvídese de él. Lo utilizaremos más adelante.

Los números negativos

A pesar de lo útiles que son estos programas de conversión para el Commodore 64, presentan un grave inconveniente: no pueden utilizarse con ellos números negativos. Aunque es un inconveniente notable, es comprensible. Los números negativos son muy importantes en los programas de código máquina, especialmente si se trabaja sin un ensamblador. La razón estriba en que, a veces, se quiere que la MPU realice el equivalente a un GOTO, yendo a una instrucción que se encuentre 3Ø bytes por delante de la dirección actual. Esto se programa normalmente suministrando el número que corresponda a la cantidad de bytes que se quiere saltar. Sin embargo, si se quiere ir a una instrucción anterior, habrá que utilizar un número negativo para este byte de datos. Esto es muy frecuente ya que es la forma en que se programan bucles en código máquina. Así pues, es necesario saber cómo se escribe un número negativo en hexa.

Lo que hace que este tema sea un tanto complicado es que en la aritmética hexadecimal no existe el signo negativo. Lo mismo ocurre con el sistema binario. La transformación de un número en su negati-

Binario: (8 bits) ØØ11Ø1Ø1 (53 en decimal)

Primer paso: cambiar todos los 1 en Ø y
todos los Ø en 1 11ØØ1Ø1Ø

Segundo paso: sumar 1 +1

El resultado es la versión
negativa del número 11ØØ1Ø11 (decimal 2Ø3)

Decimal

Primer paso: restar de 256 53
2Ø3

Esta es la versión negativa (en decimal)

Hexa

Primer paso: restar de $FF $35
FF
35
$CA

Recuerde que F representa el 15 decimal y que 15 − 5 = 1Ø, que es A en hexa

Segundo paso: sumar 1 +1
$CB

Esto es más fácil que una resta de $1ØØ que involucra la utilización de acarreos en hexa

El resultado es la versión hexa negativa del número

Constituye una alternativa realizar la conversión decimal y convertir el resultado a hexa

Fig. 3.6 El complemento a dos (o forma negativa) de un número binario.

vo se realiza a través de un procedimiento denominado *complementación,* que se ilustra en la figura 3.6. Después de un primer vistazo (y muy a menudo después de un segundo, un tercero y un cuarto), parece un sistema completamente disparatado. Así, por ejemplo, trabajando con números de un solo byte, la forma decimal del −1 es el 255. Se utiliza, pues, un número positivo alto para representar un número negativo bajo. La cosa empieza a tener un poco más de sentido cuando se estudian los números en binario. Los números que deben ser considerados como negativos empiezan todos con un 1 y los positivos

con un Ø. La MPU puede identificar, pues, el signo de un número analizando tan sólo el bit más significativo.

Es un procedimiento sencillo que puede ser utilizado eficientemente por la máquina, pero que presenta diversos inconvenientes para los humanos. Uno de ellos estriba en que los dígitos de un número negativo no son los mismos que los del mismo número positivo. En base diez, por ejemplo, −4Ø presenta los mismos dígitos que +4Ø. En hexa, sin embargo, −40 equivale a $D8 mientras que + 4Ø equivale a $28. En base diez el número −85 equivale a $AB y +85 a $55. El segundo inconveniente estriba en que los humanos no podemos distinguir entre un número de un solo byte que sea negativo y otro que sea mayor que 127. Así, por ejemplo, ¿$9F significa 159 o −97? La respuesta es: no debemos preocuparnos por ello. El microprocesador sabrá usar correctamente el número. No importa la confusión que pueda crearnos a nosotros. El problema sigue siendo, sin embargo, que tenemos que saber cuál es el empleo correcto en cada caso. A lo largo de este libro, así como en otros textos que versen sobre la programación en código máquina, usted verá cómo se emplean frecuentemente los términos «con signo» y «sin signo». Un número con signo es aquel que puede ser negativo o bien positivo. Para números de un solo byte, los valores de Ø a $7F son positivos y los valores de $8Ø a $FF son negativos. Ello corresponde a los números decimales Ø a 127 para los valores positivos y 128 a 255 para los negativos. Los números sin signo son considerados siempre como positivos. Si se encuentra con el número $9C descrito como con signo, sobrá que se trata de un número negativo (ya que es mayor de $8Ø). Si está descrito como sin signo, sabrá entonces que es positivo y que su valor se obtiene por conversión directa. ¿Cómo podemos convertir un número hexa con signo y de un solo byte a su expresión decimal utilizando nuestros programas? Es sencillo: si el número es mayor que $7F, restamos 256 de su valor decimal. Si, por ejemplo, obtiene 24Ø como resultado, entonces 24Ø − 256 = −16, y éste será el valor con signo en base diez.

Un breve descanso

Aunque sólo sea para descansar un poco de toda esta aritmética, vamos a echar un vistazo a las salidas en pantalla del Commodore 64. Cada zona de la pantalla puede controlarse a través del valor almacenado en una posición de memoria, memoria de la cual existen dos modalidades. La más sencilla desde el punto de vista de operación es la denominada *memoria de texto de pantalla.* Ocupa las posiciones que van de la $4ØØ a la $7E7, lo que equivale a las direcciones 1Ø24 a 2Ø23 en base diez. Lo que se quiere dar a entender con el nombre de «memoria de texto de pantalla» es que se trata de una zona de memo-

ria que requiere un tratamiento especial. Cualquier cosa que se almacene en ella será empleada para la visualización de texto en pantalla. Ello significa que cualquier código numérico que se almacene en una determinada posición de esta zona de memoria hará que aparezca el carácter alfanumérico o gráfico correspondiente en pantalla. Sin embargo, no todo es tan sencillo para el Commodore 64. Cuando se coloca un determinado valor en alguna posición de esta zona de memoria, su efecto no es visible a menos que se den dos condiciones. La primera es que no hubiese con anterioridad ningún carácter en esta posición de la pantalla. La segunda es que hay que utilizar un nuevo color de fondo. Los lectores de mi libro de consulta básico para el Commodore 64, *Commodore 64 Computing*, estarán al tanto ya de todas estas restricciones. Intente lo siguiente: pulse la tecla CLEAR de pantalla para borrar su contenido y pulse a continuación:

```
POKE53281,3:POKE1524,1
```

seguido de RETURN. El resultado consiste en la aparición de la letra A en medio de la pantalla. El ordenador ha convertido el código «interno» del 1 almacenado en la posición 1524 en un conjunto de números que harán que aparezca la letra «A» en el centro de la pantalla. Puede programar este efecto dentro de un bucle, tal como ilustra la figura 3.7. Antes tiene que inicializar la pantalla a través de la tecla STOP/RESTORE.

```
10 POKE53281,3
20 FORN=1024TO2023
30 POKEN,1:NEXT
```

Fig. 3.7 Programa que llena la pantalla de letras A.

El resultado de este bucle es el de llenar la pantalla de Aes. El proceso no es demasiado rápido ya que el bucle está hecho en BASIC. Veremos más adelante un proceso similar, aunque mucho más rápido, realizado en código máquina. Por el momento vea el efecto que tiene sobre el programa sustituir el 1 por otro número que corresponda al código de uno de los caracteres gráficos, tal como el 65. Es francamente útil. Si se quiere evitar el «agujero» que aparece en la pantalla debido al mensaje de OK junto al cursor, añadir un bucle sin fin en el programa, tal como:

```
40 GOTO 40
```

El problema estriba ahora en el empleo de los códigos «internos». El Commodore 64 utiliza el conjunto estándar de códigos numéricos ASCII en su BASIC. Cuando emplee las instrucciones ASC y CHR$ del BASIC, utilizará con ellas códigos numéricos ASCII. Para su uso

interno, sin embargo, el Commodore 64 convierte estos códigos ASCII en otros números. Estos números son los códigos numéricos «internos» que se describen en el Apéndice E del manual del Commodore 64. Cuando guardamos un número en la parte de memoria reservada para la pantalla, el carácter que aparecerá será el que corresponda según esta lista «interna», en lugar del que sería lógico esperar según el código ASCII correspondiente.

4. Detalles del 65Ø2

Los registros, el PC y el acumulador

Un microprocesador consiste en una serie de unidades de memoria, denominadas *registros*. Estas memorias son distintas de las que constituyen la ROM o la RAM. Los registros están conectados los unos con los otros y a la línea del núcleo de la MPU a través de unos circuitos denominados *puertas*. Estudiaremos en este capítulo algunos de los registros más importantes del 65Ø2, así como la forma en que son utilizados. Un buen punto de partida lo constituye el registro denominado *PC* (abreviación del inglés Program Counter, o sea contador de programa).

Su misión no es la de contar programas, sino la de contar los diferentes pasos de un programa. El PC es un registro de dieciséis bits (dos bytes) capaz de almacenar una dirección de memoria completa, hasta el valor $FFFF (65535 en base diez). Su finalidad es la de contar los números de dirección, incrementándose en 1 el contenido del PC cada vez que se completa una instrucción o bien cuando es necesario otro byte. Así, por ejemplo, si el PC alberga la dirección $1F3A (7994 en base diez) y esta dirección contiene un byte de instrucción, se incrementará el valor del PC al valor $1F3B (7995 en base diez), con lo que la MPU estará en disposición de aceptar otro byte. El byte siguiente se leerá de esta nueva dirección.

Lo que hace tan importante al PC es el hecho de que constituye el procedimiento a través del cual se utiliza, de forma automática, la memoria. Cuando el PC contiene un determinado número de dirección, las señales eléctricas asociadas a los unos y los otros de la misma aparecen en un conjunto de líneas denominadas *bus de direcciones* que son quienes enlazan la MPU con la totalidad de la memoria, sea RAM o ROM. El número almacenado en el PC seleccionará un byte de la memoria: el que se encuentre almacenado en la dirección contenida en el PC. Al principio de una operación de lectura, la MPU mandará una señal denominada de lectura sobre otra línea, lo que hará que la memoria conecte el byte en cuestión a otro conjunto de líneas: el bus de datos. Las señales del bus se corresponderán, así, con el patrón de ceros y unos almacenados en el byte de memoria seleccionado por la dirección del PC. Cada vez que cambia el número almacenado en el

PC, se selecciona otro byte de memoria, siendo ésta la forma a través de la cual la MPU va accediendo a los distintos bytes. Cuando la MPU está preparada para aceptar otro byte, se incrementa el PC y se manda otra señal de lectura.

Hay otras formas a través de las cuales se puede alterar el contenido del PC aunque prescindiremos, por el momento, de ellas y estudiaremos otro registro: el *acumulador.* El acumulador de un microprocesador es el registro en el que «más cosas» se hacen. Con ello queremos decir que se le utiliza para almacenar un número que se quiera guardar en otro sitio, así como para realizar sumas o, en general, cualquier operación que se tercie. El nombre de «acumulador» viene de la forma en que opera este registro. Si se tiene un número almacenado en él y se le suma otro número, el resultado queda almacenado en el mismo acumulador. El equivalente más cercano en BASIC consiste en utilizar una variable A y escribir la línea:

```
A = A + N
```

donde N representa una variable numérica. El resultado de esta línea de BASIC consiste en sumar N al valor de A y hacer A igual al nuevo valor calculado. De esta forma el valor primitivo de A se pierde. El acumulador actúa de la misma forma, con la diferencia de que éste no puede almacenar números mayores que 255 (en base diez).

El 65Ø2 tiene un registro acumulador denominado muy a menudo con el nombre de registro A. Su importancia estriba en que es mucho más utilizado que el resto de registros ya que hay muchas acciones que se llevan a cabo mucho más rápidamente o de forma mucho más adecuada sobre él, o incluso que sólo pueden ser realizadas en él. Cuando se lee un byte de la memoria acostumbra guardarse en el acumuldor. Cuando se lleva a cabo una acción aritmética o lógica, acostumbra realizarse en el acumulador quedando almacenado el resultado, una vez más, en el mismo acumulador.

Métodos de direccionamiento

Cuando programamos en BASIC no tenemos porqué preocuparnos de direcciones de memoria a menos que utilicemos las instrucciones PEEK o POKE. La tarea de encontrar el lugar donde están almacenados los diferentes bytes la realiza el sistema operativo de la máquina. Cuando en un programa BASIC se hace referencia a un valor tal como, por ejemplo, en la línea:

```
1Ø N = 12
```

no tenemos porqué preocuparnos de dónde se almacena el número

12 ni de la forma en que se almacena dicho valor. De forma similar, si añadimos la línea:

```
2Ø K = N
```

tampoco tenemos porqué preocuparnos de dónde estaba almacenado el valor de N ni de dónde se guardará el valor de K. Recordando el ejemplo que propusimos referente a la construcción de una pared, es de esperar que cuando programemos en código máquina tendremos que especificar cada uno de los números que utilicemos o bien la dirección donde se puede encontrar dicho número. La forma en la que obtenemos un número o bien encontramos el sitio donde almacenarlo es lo que se denomina el *método de direccionamiento*. Lo que hace tan importante su elección es el hecho de tener que utilizarse en cada instrucción un código numérico distinto según el método de direccionamiento que se utilice. Esto significa que cada instrucción existe en varias versiones distintas, con un código numérico diferente según el método de direccionamiento que se emplee en ella. Una lista exhaustiva de todos los métodos de direccionamiento que existen en el 65Ø2 podría crear confusión, una vez llegados a este punto. Esta es la razón por la que se ha reservado el Apéndice D para su comentario. Lo que haremos en las líneas que siguen es ver varios ejemplos de los métodos de direccionamiento junto con la forma en que se describen en lenguaje ensamblador (assembler en inglés).

El lenguaje ensamblador

Intentar escribir código máquina directamente en formato numérico es un proceso difícil sujeto a errores desde el principio al final. La forma más conveniente de empezar a escribir un programa de este tipo es hacerlo en varias etapas, empezando con lo que se denomina *lenguaje ensamblador* (assembly o assembler language, en inglés). Este está constituido por una serie de abreviaciones de palabras —denominadas mnemónicos— junto con números que constituyen los datos y las direcciones numéricas empleadas. Los números tanto pueden estar en hexadecimal como en base diez, con tal de que adopten el formato adecuado. Cada línea de un programa ensamblador señala una acción del microprocesador, siendo este conjunto de instrucciones «ensamblado» (o sea, traducido) posteriormente a código máquina (de ahí el nombre de «ensamblador»).

El propósito de cada línea de un programa ensamblador es el de ilustrar tanto una acción como los datos o direcciones involucrados en ella. De la misma forma, cuando utilizamos la instrucción TAB en BASIC hemos de completarla con un número. La parte de lenguaje ensamblador que indica qué es lo que hay que hacer, se conoce con

el nombre de *operador* y la parte que especifica dónde hay que realizar dicha acción se conoce con el nombre de *operando.* Algunas instrucciones no necesitan operandos. Las estudiaremos más adelante.

Un ejemplo aclarará todo esto. Supongamos que tenemos una línea de lenguaje ensamblador tal como:

```
LDA # $12
```

El operador es LDA, una versión abreviada de LOAD A, que significa cargar con un determinado byte el registro acumulador A. El operando es #$12, el $12 del cual significa que se trata de un 12 hexadecimal, en lugar de un 12 en base diez. El otro símbolo, el #, se utiliza para especificar el método de direccionamiento a emplear, concretamente el denominado «direccionamiento inmediato».

La línea entera debería tener el efecto, pues, de colocar el número $12 en el registro acumulador A. Es el equivalente, en términos de código máquina, de la instrucción BASIC:

```
A = 18 (recuérdese que $12 es igual a 18 en base diez)
```

Imagine, según ella, que la memoria a la que se asigna este número está en el interior del microprocesador en lugar de formar parte de la RAM y que ostenta la etiqueta «A».

Una instrucción como LDA #$12 se dice que emplea direccionamiento immediato debido a que el byte que hay que guardar en el acumulador debe estar colocado en el byte de memoria que se encuentra justo después del byte de instrucción. Es como si dejase una nota a su lechero en la que pusiese «el dinero está en el sobre de la puerta siguiente». El código numérico para la parte de la instrucción correspondiente a «LDA #» es el $A9, con lo que la secuencia A9 12 en memoria representará la instrucción completa. Es mucho más fácil recordar qué es lo que significa LDA #$12 que interpretar A9 12: ésta es la razón de utilizar el lenguaje ensamblador siempre que se pueda.

El direccionamiento immediato, tal como hemos visto, puede demostrarse útil, pero obliga a utilizar un valor en concreto y una dirección de memoria fija. Es como programar en BASIC:

```
N = 4 * 12 + 3
```

en lugar de:

```
N = A * B + C
```

En el primer ejemplo N nunca puede adoptar un valor que no sea el 51, pudiéndose haber escrito perfectamente: N = 51. El segundo ejemplo es mucho más flexible, dependiendo el valor de N de los valores elegidos para las variables A, B y C. Cuando se guarda un programa de código máquina en RAM, los números utilizados a través de este método de direccionamiento inmediato pueden ser alterados en

caso de necesidad. Sin embargo, cuando se guarda el programa en ROM, es imposible modificarlos. Por esta razón son necesarios otros métodos de direccionamiento. Uno de ellos es el llamado «direccionamiento absoluto».

El direccionamiento absoluto utiliza una dirección completa de dos bytes como operando. Eso origina un trabajo suplementario para el 65Ø2, y es así debido a que tiene que leer el código del operador y a continuación los dos bytes de datos con el fin de poder construir la dirección de memoria donde encontrar el dato real. Entonces deberá colocarse esta dirección en el PC, leer el byte de dato, efectuar la operación y volver a colocar la dirección correcta en el PC. La figura 4.1 muestra en forma de diagrama lo que debe llevarse a cabo. Como consecuencia de todo ello, una operación que se efectúe con direccionamiento absoluto es mucho más lenta de ejecución que una que se efectúe con direccionamiento inmediato, pero como puede guardarse cualquier byte en la dirección especificada, es muy fácil modificar los datos.

Supongamos, por ejemplo, que tenemos la instrucción:

```
LDA $7FFE
```

En este fragmento de lenguaje ensamblador, el operador LDA (cargar el acumulador A) y el operando es la dirección $7FFE. Lo que

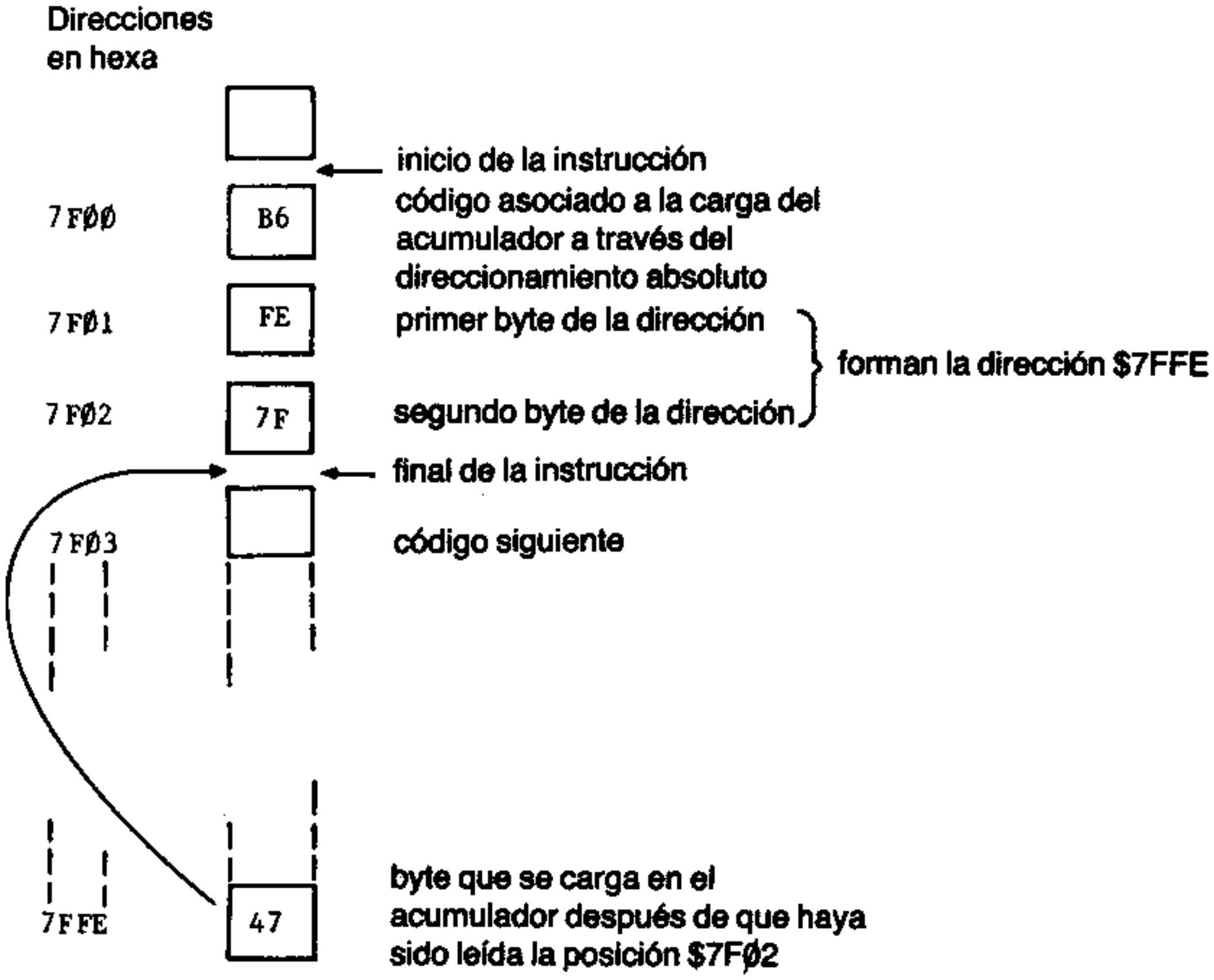


Fig. 4.1 Forma en que actúa el método de direccionamiento absoluto.

hay que recordar es que lo que se carga en el registro A no es 7FFE, que es una dirección de dos bytes, sino el byte almacenado en esta dirección de memoria. El efecto de la instrucción completa es, pues, colocar una copia del byte almacenado en $7FFE en el acumulador A del 65Ø2. Cuando se haya ejecutado esta instrucción, la dirección $7FFE continuará albergando su propia copia del byte ya que leer una memoria no altera en forma alguna su contenido.

Se puede utilizar también el método de direccionamiento absoluto en una instrucción cuya misión sea la de almacenar un byte en memoria. La instrucción:

```
STA $7FFF
```

significa que el byte guardado en el acumulador A será almacenado en la dirección de memoria $7FFF. Esta instrucción sí altera el contenido de esta posición de memoria, pero no la del acumulador A que seguirá conteniendo el mismo byte después de que se haya ejecutado dicha instrucción.

Direccionamiento de página cero

El direccionamiento de página cero es un método que permite especificar una dirección de memoria utilizando un solo byte. El secreto estriba en considerar como igual a cero el byte alto de la dirección. De ahí viene el nombre de página cero para este tipo de direccionamiento. Supongamos, por ejemplo, que utilizamos:

```
LDA $3F
```

No aparece por ningún lado el signo #, de forma que la anterior instrucción no puede significar cargar el número $3F en el acumulador. Lo que significa es que hay que cargar el acumulador con el byte almacenado en la posición $ØØ3F de memoria. El ØØ representa la página cero y el 3F es el trozo de dirección que se ha especificado en la instrucción anterior. El rango de instrucciones a las que se puede acceder con este método va solamente de $ØØØØ a $ØØFF. Esto representa, tan sólo, 256 (en base diez) bytes, aunque se trata de unos 256 bytes muy importantes. En el 65Ø2 el direccionamiento de página cero permite tener acceso a cualquiera de estas direcciones de una forma muy rápida, utilizando un solo byte de la dirección (el más bajo). Esto hace que estas direcciones sean especialmente útiles para un programador que quiera llevar a cabo lecturas y almacenamientos rápidos. Por esta razón, las direcciones de RAM de la página cero de un 65Ø2 son las favoritas para el almacenamiento de valores importantes. Ya hemos visto como eran utilizadas para almacenar valores tal como la dirección de inicio del programa BASIC, así como la dirección de la

tabla de la lista de variables. Las mismas direcciones se emplean para multitud de otros importantes valores. A medida que avancemos en este libro, veremos algunos de ellos con mayor detalle. Como el Commodore 64 emplea muchas de las direcciones de la página cero para sus propios fines, tendremos que tener mucho cuidado sobre cómo utilizamos esta zona de memoria en nuestros propios programas. Si intentamos utilizar una posición de memoria que deba emplear el ordenador para su propio uso, podemos paralizar todo el sistema. Esta es la razón de porqué es tan importante grabar un programa en código máquina antes de probarlo.

Direccionamiento indexado

El direccionamiento indexado es un método que es particularmente útil en el 65Ø2. Su principio de funcionamiento se basa en utilizar un registro de ocho bits para almacenar un cierto byte. Cuando se utiliza el direccionamiento indexado, se suma este byte a lo que se denomina una dirección de base. Dirección de base significa una dirección a la cual se puede sumar un cierto número antes de ser utilizada. Así, por ejemplo, supongamos que tenemos el número $4C almacenado en el registro de índice. Si especificamos, entonces, que queremos leer la dirección $7ØØØ con indexado, sucede que se suma a la dirección $7ØØØ el valor almacenado en el registro de índice. Se llega así a la dirección $7Ø4C, siendo éste el valor que se utilizará como dirección de memoria. El resultado final es que se cargará en el acumulador el byte almacenado en la posición $7Ø4C de memoria.

Hay dos registros en el 65Ø2 que pueden ser utilizados de esta forma: los registros X e Y. Ambos son registros de ocho bits, los dos muy parecidos, aunque presentan varias características diferenciales que se demostrarán, más tarde, muy importantes. Una de las diferencias a la que podemos hacer referencia afecta al direccionamiento indexado de la página cero. Cuando cargamos el registro X con un byte tal como $4A y utilizamos una instrucción (en lenguaje ensamblador) tal como:

LDA $7ØØØ,X

queremos significar con ello que la dirección que hay que utilizar es la $7ØØØ más el contenido del registro X. El resultado es que se utiliza la dirección $7Ø4A. Esto es lo que se denomina «direccionamiento indexado absoluto». «Absoluto» significa en este contexto que estamos utilizando para la «dirección de base» $7ØØØ el método de direccionamiento absoluto y que sumamos a esta dirección el número de «índice» 4A contenido en el registro de índice X. Podemos utilizar el regis-

tro de índice y del 65Ø2 exactamente de la misma forma, con una instrucción tal como:

```
LDA $7ØØØ,Y
```

en lenguaje ensamblador. La utilización del registro de índice X permite, sin embargo, usar el direccionamiento de página cero. Podemos, por ejemplo, utilizar una instrucción de lenguaje ensamblador tal como:

```
LDA ØC,X
```

Esto significa que la dirección de base a utilizar es la $ØØØC a la que hay que añadirle el número almacenado en el registro X antes de utilizarla. Si el número almacenado en el registro X fuese $23, la dirección a emplear vendría dada por $ØØØC + $23, lo que equivale a la dirección $ØØ2F. Entonces se cargaría el acumulador con el contenido de la posición $ØØ2F de memoria. El direccionamiento de página cero no puede utilizarse con el registro de índice Y, de forma que una instrucción del estilo de:

```
LDA $1F,Y
```

es imposible: no tiene ningún código de instrucción asociado.

Por ahora, el empleo del direccionamiento indexado puede no parecer demasiado útil. Después de todo, lo único que se hace es sumar un número a una dirección. Lo que hace tan útil a este método es el hecho de poderse alterar el valor almacenado en los registros X e Y. Más concretamente, puede incrementarse o decrementarse el valor contenido en cualquiera de los dos registros de índice. La instrucción INX significa «incrementar X». Su efecto consiste en sumar un uno al valor almacenado en el registro X. Pero como el valor del registro X es el que se suma a la «dirección de base», utilizando el direccionamiento indexado, incrementar X equivale a incrementar la dirección utilizada en una operación de lectura o escritura indexada.

Supongamos, por ejemplo, que se quieren almacenar diez bytes en diez posiciones consecutivas de memoria. Es un problema muy frecuente ya que es lo que hay que hacer, por ejemplo, para sacar diez caracteres por pantalla. Mediante la utilización del indexado puede construirse un bucle que almacene un byte (mediante indexado), incremente el registro de índice y repita la operación de escritura. Llevando a cabo esta operación diez veces se completa la acción que se quería llevar a cabo. Quizás es un poco prematuro hablar de bucles, pero volveremos en breve sobre ello. El ejemplo es ilustrativo ya que muestra uno de los empleos más frecuentes del direccionamiento indexado.

Además de incrementar los registros X e Y (mnemónicos INX e INY), se puede también decrementarlos. La instrucción de ensambla-

dor DEX significa decrementar X (restar un uno al valor almacenado en X) y DEY significa decrementar Y. Como disponemos de dos registros de índice, incluso es posible leer un byte de una dirección de base X-indexada e incrementar a continuación X. El byte leído puede almacenarse, acto seguido, a través de un proceso de indexado (esta vez a través de Y) con otra dirección de base. Puede decrementarse, entonces, el registro Y. Si se realiza todo ello dentro de un bucle, el resultado será que se irán copiando bytes de un conjunto de posiciones de memoria, empezando por una de ellas y yendo hacia arriba, hacia otro conjunto de posiciones de memoria, empezando por una de ellas y yendo esta vez hacia abajo. ¿Le parece complicado? Puede parecerlo expresándolo así en palabras, pero de hecho es un método muy sencillo para ir moviendo bytes de una zona de memoria hacia otra, lo que constituye una operación fundamental dentro del campo de la programación.

Direccionamiento indirecto

«Direccionamiento indirecto» significa buscar en una posición de memoria la dirección del lugar donde se encuentra un cierto byte. Es como ir donde está una oficina de turismo para encontrar la dirección de un cierto hotel. El 65Ø2 permite utilizar dos tipos fundamentales de direccionamiento indirecto. Ambos son relativamente complicados y haremos escaso uso de ellos en este libro, ya que éste trata de ser un libro de introducción, no una enciclopedia. Lo que sí podemos hacer, sin embargo, es ver los principios en los que se basan ya que ambos procedimientos son muy similares en varios aspectos.

El principio fundamental es el de utilizar las posiciones de la página cero de memoria de dos en dos. En cualquiera de estos pares puede almacenarse una dirección de dos bytes. El orden empleado para ello debería serle, ya, familiar: primero el byte bajo y después el byte alto. Uno de los métodos de direccionamiento indirecto utiliza uno de estos pares de posiciones de la página cero. El efecto de una instrucción que utilice direccionamiento indirecto es, así pues, el de colocar el primero de estos bytes en la parte inferior del registro asociado al contador de programa (PC). A continuación se lee el siguiente byte de la página cero y se coloca en la parte alta del mencionado contador de programa. De esta forma, este contador de programa pasa a contener una dirección completa que es la que se utilizará en el proceso de lectura o escritura, dependiendo de la operación que se especifique. Así, por ejemplo, si la posición $1Ø contiene $3D y la posición $11 contiene $7F, el resultado de una lectura indirecta a través de $1Ø sería el de colocar la dirección $7F3D en el contador de programa

para pasar a cargar el acumulador con el byte almacenado en la posición $7F3D. Una vez más, parece todo muy complicado e innecesario a menos que se piense en las especiales ventajas que supone un método tal como éste. La principal ventaja es que se puede cambiar la dirección utilizada modificando los números contenidos en la página cero de la RAM. Los métodos de direccionamiento indirecto del 65Ø2 son, de hecho, bastante más complicados de lo que se ha apuntado aquí, ya que pueden utilizarse también los registros de índice. Uno de los métodos de direccionamiento indirecto se ilustra, a través de ejemplos, en los capítulos 7 y 9 de este libro.

Direccionamiento relativo

El direccionamiento relativo fue uno de los primeros métodos de direccionamiento utilizados, aunque no es demasiado empleado hoy en día. El concepto de direccionamiento relativo lleva asociada la idea de que el operando de una instrucción pueda ser de uno o dos bytes y que la dirección que se utilice se calcule sumando este número (al que se denomina «offset») a la «dirección actual», que corresponde al valor almacenado en el contador de programa. Es algo parecido a los viejos mapas que aparecen en las historias de la Isla del Tesoro que especificaban «un paso a la izquierda, dos hacia adelante, tres a la derecha, ...» y así sucesivamente. No se sabe donde conduce todo ello a menos que se sepa el lugar de donde partir. En un microprocesador, cuando se utiliza el direccionamiento relativo, el lugar de partida acostumbra ser la dirección almacenada en el PC. El 65Ø2 utiliza el direccionamiento relativo sólo en las instrucciones de bifurcación, siendo el offset un byte que se trata como un número con signo. La utilización de un solo byte con signo significa que se puede saltar a una dirección que se encuentre, como máximo a 127 pasos de memoria hacia adelante o bien a 128 pasos hacia atrás con respecto a la posición de la actual instrucción del programa. Estas bifurcaciones son los equivalentes en código máquina del GOTO, con la diferencia de que se las puede hacer depender de una condición, tal como que el acumulador esté a cero. Es como si se tratase de una instrucción BASIC del estilo de:

IF A = Ø THEN GOTO ....

Estudiaremos más adelante con mayor detenimiento estas instrucciones de bifurcación.

Los otros registros

El registro S es uno de los otros registros de ocho bits que existen en el 65Ø2 cuyo estudio dejaremos, por el momento, para más tarde. Es un registro del tipo de los que se denominan «puntero de pila» y que se utilizan para localizar los bytes que haya almacenado, de forma temporal, la MPU. Si se altera el contenido del registro S, puede provocarse un colapso del sistema operativo del ordenador. Otro registro de notable importancia para nosotros es el denominado registro de estado del ordenador (registro P) que vamos a estudiar con mayor detalle en el apartado que sigue.

El registro P

El registro de estado del procesador, también llamado registro de indicadores, no es realmente un registro como los otros. No puede hacerse nada con los bits de este registro ya que no constituyen número alguno. Para lo que se utiliza el registro de estado es como una especie de indicador electrónico. Siete de los bits del registro (hay ocho en total) se utilizan para memorizar lo que ocurrió en el anterior paso de programa. Si se trataba de una resta que dejó el acumulador a cero, uno de los bits del registro de estado pasará de Ø a 1 para indicar esta circunstancia a la MPU. Si se suma un número al contenido del acumulador y el resultado tiene nueve bits en lugar de los ocho habituales (fig. 4.2), otro de los bits del registro de estado se pondrá a 1 (o sea, pasará de Ø a 1). Si el bit más significativo de un registro cualquiera se pone a 1 (lo que significa que alberga un número negativo), otro de los bits de estado se pondrá también a 1. Así pues, cada bit se utiliza para memorizar una determinada circunstancia que se

Número en el acumulador	1Ø11Ø11Ø
Número que se le suma	11ØØØ1Ø1
Resultado	1Ø1111Ø11

Consiste en 9 bits, cuando el acumulador puede contener sólo ocho.
El bit más significativo va a parar al indicador de acarreo del registro de estado.

El acumulador contiene ahora Ø1111Ø11
El bit de acarreo se queda a 1

Fig. 4.2 Por qué es necesario el bit de acarreo.

haya producido. Lo que hace que sea tan importante este registro es el hecho de que se puedan establecer instrucciones de bifurcación en función del estado de alguno de los bits que lo componen, según que esté a 1 o a Ø.

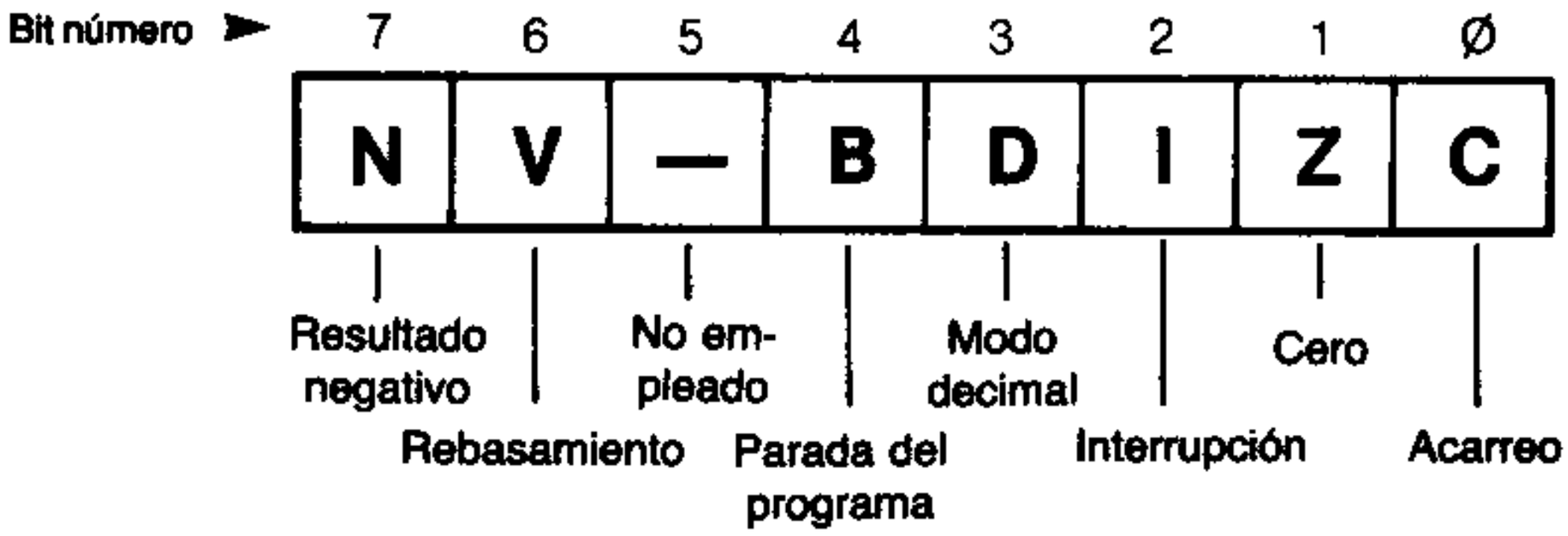


Fig. 4.3 Los bits del registro de estado del procesador. Sólo tres de ellos se utilizan de forma intensiva en la mayoría de los programas (el N, el Z y el C).

La figura 4.3 muestra la distribución de bits del registro de estado del 65Ø2. De ellos, los bits Ø, 1 y 7 son los que tienen más posibilidades de ser utilizados por un principiante de la programación en código máquina. Los otros son de una utilización bastante más especializada y de uso más restringido. El bit Ø es el bit de Acarreo (o indicador de Acarreo). Se pone a 1 si una determinada suma resulta en el acarreo del bit más significativo de un registro. Si no se produce desbordamiento, el bit permanece a Ø. Cuando se lleva a cabo una resta (o una operación parecida, tal como una comparación) se utiliza este bit, también, para indicar el hecho de «llevarnos una» al realizar la operación. Para algunas aplicaciones puede utilizarse como el noveno bit del acumulador, especialmente por lo que respecta a las operaciones de decalaje y rotación en las que los bits de un byte se corren todos un lugar (fig. 4.4).

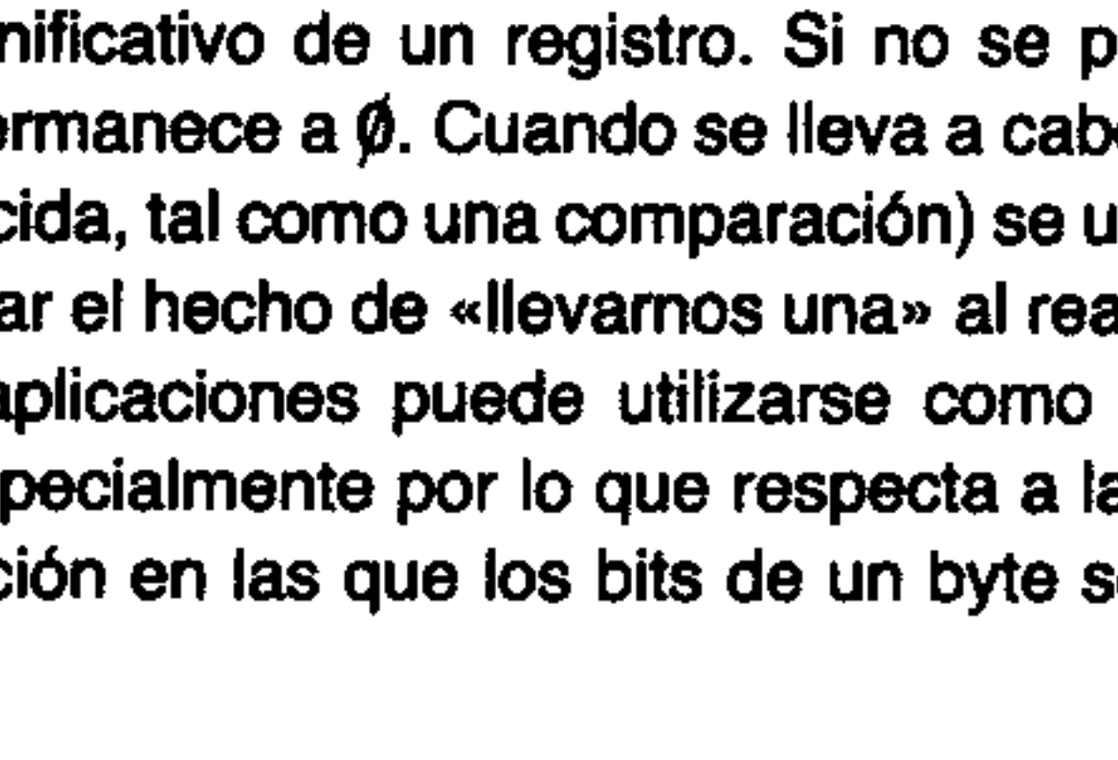


Fig. 4.4 Utilización del bit de acarreo en una operación de decalaje en la cual todos los bits de un byte se corren un lugar hacia la izquierda.

El indicador de Cero es el bit 1. Se pone a 1 cuando el resultado de la anterior operación haya sido un cero exacto y se pone a Ø en caso contrario. Es una manera útil de detectar la igualdad de dos bytes: restar el uno del otro y si el indicador de cero se pone a 1, los dos bytes son iguales. El indicador Negativo se pone a 1 si el valor que se obtiene como resultado de una operación sobre un cierto registro tiene su bit más significativo a 1. Este el tipo de número que se considerará negativo si se trabaja con números con signo. Así pues, este bit se utiliza de forma extensiva cuando se trabaja con números con signo.

Pueden alterarse de forma selectiva algunos de los bits del registro de estado, aunque no es tarea para principiantes. Para la mayoría de las aplicaciones no cargaremos nunca nada sobre este registro ni memorizaremos para nada su contenido. Se le utiliza casi exclusivamente como forma de tener constancia de lo que se ha hecho y esto es lo que estudiaremos en este libro. Otras aplicaciones pueden esperar hasta que sea un experto.

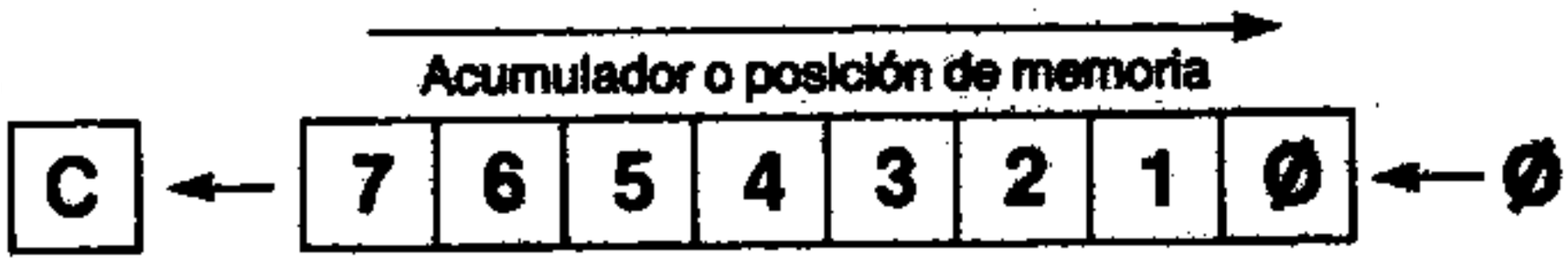
5. Acciones ligadas a los registros

Acciones sobre el acumulador

Como el acumulador es el principal registro de un solo byte, vamos a enumerar en detalle y a describir las acciones que se llevan a cabo sobre él. De todas las acciones que involucran al acumulador, la más importante es la de transferencia de información. No puede llevarse a cabo, por ejemplo, ningún tipo de aritmética sobre códigos numéricos ASCII, de forma que las principales operaciones que se llevan a cabo sobre estos bytes son las de lectura y almacenamiento. Cargamos el acumulador con un byte leído de una determinada posición de memoria y lo almacenamos, a continuación, en otra. Muy pocos ordenadores permiten transferir de forma directa un byte de una dirección de memoria a otra, de manera que se emplea de forma casi exclusiva este procedimiento un tanto peculiar para leer de una dirección y almacenar en otra un dato cualquiera.

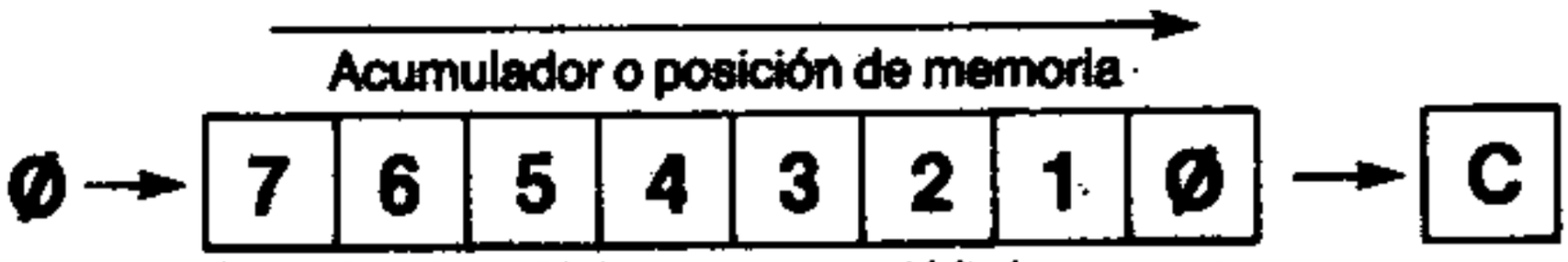
El grupo de acciones que sigue en importancia es el correspondiente el capítulo de operaciones aritméticas y lógicas, capítulo en el que se encuentran incluidas la suma, la resta y la Y y la O lógicas. Pueden incluirse también en este grupo las operaciones de DECALAJE y ROTACION que ya se comentaron, aunque de una forma muy breve, en el capítulo anterior. El efecto de las instrucciones de decalaje y rotación del 65Ø2 se ilustran, junto con los mnemónicos correspondientes, en la figura 5.1. Un decalaje implica siempre la pérdida de uno de los bits almacenados, más concretamente, el que se encuentra en el extremo del byte hacia el cual se desplaza éste. Ambos tipos de decalajes hacen que aparezca un cero en el extremo opuesto y ambos utilizan el bit de acarreo como el noveno bit del acumulador. La operación de decalaje puede llevarse a cabo tanto sobre el registro A (el acumulador) como sobre cualquier byte que se encuentre almacenado en memoria. El efecto de un decalaje sobre un número binario almacenado en un registro es el de multiplicar este número por dos si el decalaje es hacia la izquierda, o bien de dividirlo por dos si es hacia la derecha (fig. 5.2). Una rotación, por el contrario, mantiene los mismos bytes en el acumulador, pero altera su posición. El 65Ø2 tiene dos instrucciones de rotación: una para rotar hacia la izquierda y otra para hacerlo hacia la derecha. Una vez más, se utiliza el bit de acarreo

ASL



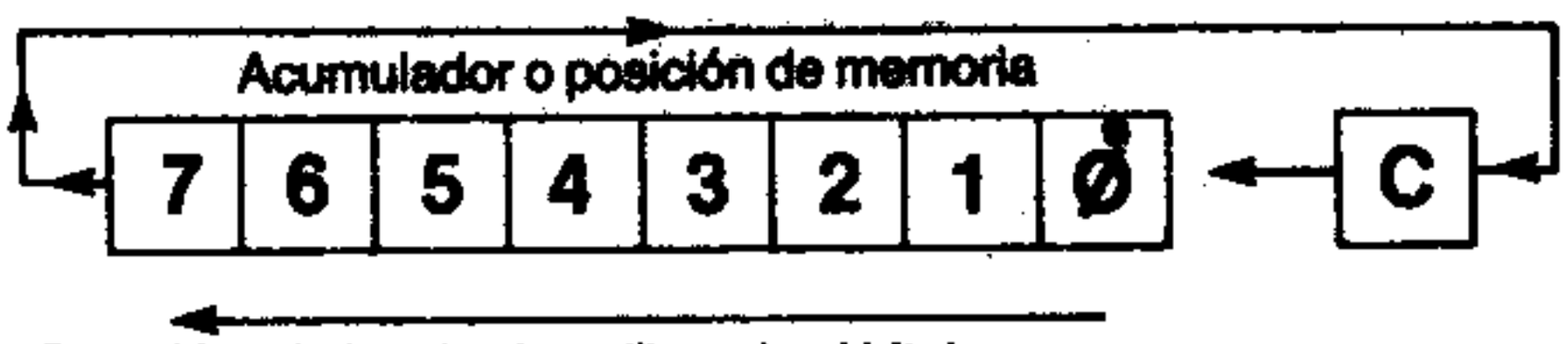
Se coloca un Ø en el lsb y el msb va a parar al bit de acarreo

LSR



Se coloca un Ø en el msb y el lsb va a parar al bit de acarreo

ROL



ROR

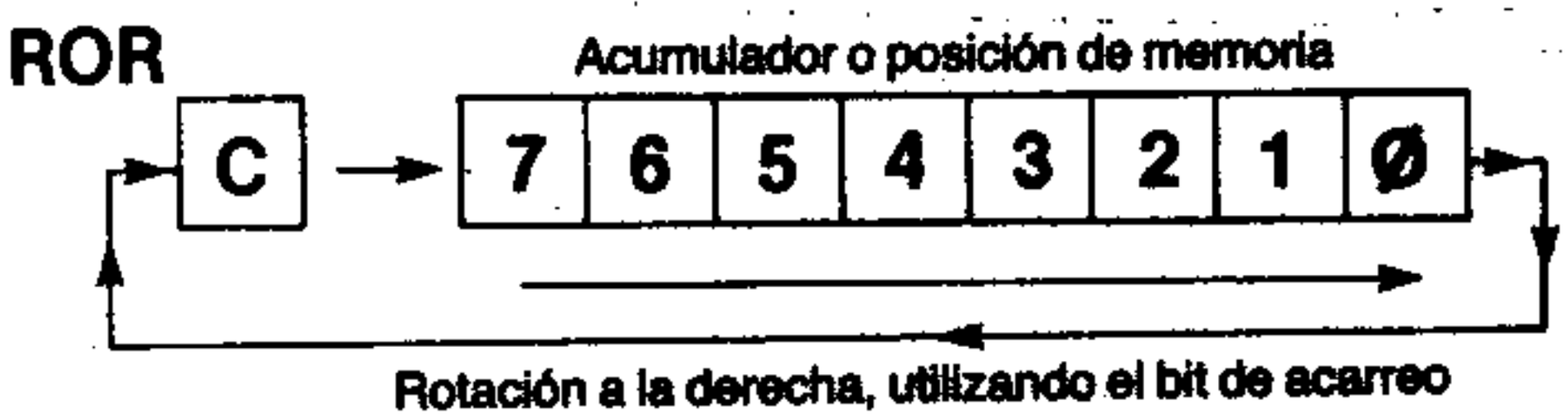


Fig. 5.1 Las instrucciones de decalaje y rotación del 65Ø2: ASL (decalaje a la izquierda), LSR (decalaje a la derecha), ROL (rotación a la izquierda) y ROR (rotación a la derecha).

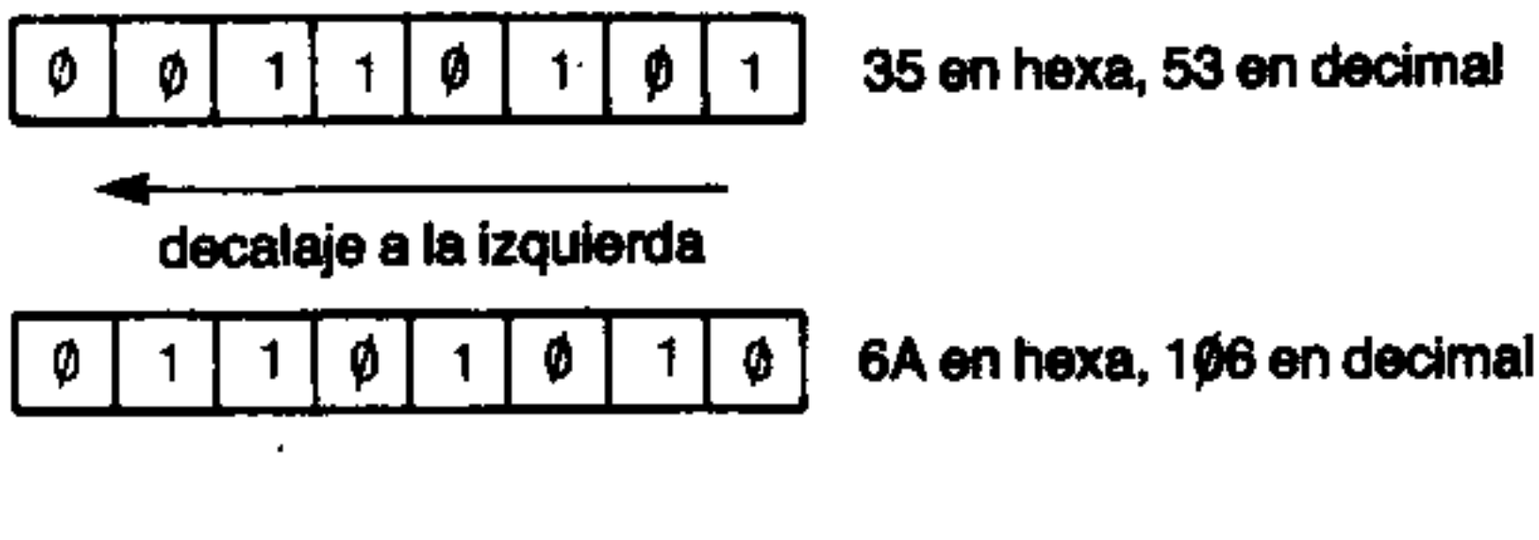


Fig. 5.2 Efecto de un decalaje sobre un número.

como noveno bit del acumulador. Tanto puede llevarse a cabo una rotación sobre el acumulador A como sobre un byte almacenado en memoria.

Al contrario de lo que ocurre con la gran mayoría de los microprocesadores, el 65Ø2 no permite utilizar el acumulador para llevar a cabo acciones de incremento o decremento. Incrementar significa sumar 1 y decrementar significa restar también 1. Ambas acciones pueden realizarse sobre bytes almacenados en la memoria, de forma que el programador del 65Ø2 debe incrementar o decrementar los bytes antes de utilizarlos. Esto puede suponer que haya que devolver un byte a la memoria para incrementarlo o bien decrementarlo. Es igual de sencillo, sin embargo, utilizar un ADD inmediato para efectuar un incremento. No es tan fácil emplear una resta inmediata ya que debe ponerse a cero el bit de acarreo del registro P (el registro de estado del procesador) antes de llevar a cabo la resta. Si el bit de acarreo no está a cero, el resultado de restar 1 será el de restar 2. A pesar de todo ello, puede emplearse, sin embargo, el acumulador para llevar a cabo las importantes operaciones de comparación.

La instrucción CMP (CoMParar) es particularmente útil. CMP constituye el mnemónico y debe emplearse con él uno de los procedimientos estándar de direccionamiento de memoria. El efecto de CMP es el de comparar el byte leído de la memoria con el que se encuentre almacenado en el acumulador A. En este contexto, «comparar» significa que el byte leído de la memoria es restado del byte almacenado en el acumulador. La diferencia entre esta instrucción y una verdadera resta estriba en que no se almacena el resultado de la operación en ningún lugar. El resultado de la resta se utiliza sólo para actualizar los indicadores del registro P, pero para nada más ya que el byte del acumulador permanece inalterado. Así, supongamos que el acumulador contenga el byte $4F y que se encuentre el mismo valor almacenado en la dirección $327F. Si utilizamos la instrucción:

```
CMP $327F
```

el indicador de cero del registro P se pondrá a 1 (se activará), aunque el byte almacenado en el acumulador seguirá siendo el $4F y el byte almacenado en la memoria seguirá siendo, asimismo, un $4F. Una resta habría dejado el contenido del acumulador a cero.

¿Por qué es tan importante todo esto? Bien, supongamos que se quiere que un programa haga algo si se pulsa la tecla «Y» y que haga algo diferente si se pulsa la tecla «N». Si nos las arreglamos para que el programa de código máquina almacene en el acumulador el código ASCII de la tecla pulsada, puede utilizarse este valor para efectuar una comparación. Comparando el contenido del acumulador con $4E (el código ASCII de la «N»), podremos ver si fue la tecla «N» la que se pulsó. Si fue así, el indicador de cero se pondrá a 1 al efectuar la

comparación. Si no es así, hay que seguir comparando. Comparando el acumulador con $59 podemos, una vez más, ver si se pulsó la tecla «Y» a través del valor del indicador de cero. Si ninguna de estas comparaciones hace que se ponga este bit a 1, sabremos que no se pulsó ni la «Y» ni la «N», y tendremos que seguir probando. Si le parece que todo lo anterior se parece mucho a la acción que se programa a través de un bucle GET$ en BASIC, no va por mal camino. Así es.

Finalmente, tenemos las instrucciones de bifurcación condicional. Estas, tal como sugiere su nombre, permiten que se compruebe el valor de los indicadores del P y que el programa salte a una nueva dirección si se detecta que el indicador considerado está a 1. Pero ¿cuál es el indicador que se utiliza? Depende de cual sea la instrucción de bifurcación condicional que se utilice ya que existe una diferente para cada uno de los indicadores principales, así como para cada uno de los dos estados que se pueden considerar. Considérese por ejemplo, las dos pruebas cuyos mnemónicos son BEQ y BNE. BEQ significa «saltar si es igual a cero» (del inglés Branch if EQual zero). Tal como su nombre sugiere, esta instrucción hará que se lleve

---

BCC (Branch on Carry Clear): Salta a una nueva dirección si el indicador de acarreo está a cero.
BCS (Branch on Carry Set): Salta a una nueva dirección si el indicador de acarreo está a uno.
BEQ (Branch if EQual to zero): Salta a una nueva dirección si el indicador de cero está a uno.
BNE (Branch if Not Equal to zero): Salta a una nueva dirección si el indicador de cero está a cero.
BMI (Branch on result MInus): Salta a una nueva dirección si el resultado de la operación anterior fue un número negativo (indicador N a uno).
BPL (Branch on Positive resuLt): Salta a una nueva dirección si el resultado de la operación anterior fue un número positivo o nulo (indicador N a cero).
BVC (Branch on oVerflow Clear): Salta a una nueva dirección si el indicador de rebasamiento está a cero.
BVS (Branch if oVerflow Set): Salta a una nueva dirección si el indicador de rebasamiento está a uno. Ello ocurrirá cuando una acción de sumar o bien de restar provoque que el bit de signo (el más significativo) cambie de forma incorrecta.

Nota: Todas las instrucciones anteriores utilizan el método de direccionamiento relativo al PC. El byte de instrucción debe ir seguido por un solo byte, denominado de desplazamiento, que se suma a la dirección contenida en el Contador de Programa para obtener la nueva dirección.

JMP (JuMP to new address): Salta a la nueva dirección definida por los dos bytes que siguen al byte de instrucción.

Fig. 5.3 Lista completa de las instrucciones de bifurcación del 65Ø2.

a cabo un salto en el programa si el resultado de una resta, o bien de una comparación, es cero. En otras palabras, hace que se lleve a cabo la bifurcación si el indicador de cero está a 1. Su «homónimo opuesto», BNE, significa «saltar si no es igual a cero» (del inglés Branch if Not Equal to zero). Su acción es la de llevar a cabo una bifurcación si el indicador de cero no está a 1. Algo parecido a todo esto ocurre con varios de los otros indicadores. Existe también un tipo distinto de instrucción de bifurcación, cuyo mnemónico es JMP, que no lleva a cabo ningún tipo de comprobación, actuando tal como lo haría un GOTO sin ningún IF que lo precediese.

La lista completa de todas las instrucciones de bifurcación se muestra en la figura 5.3. Probablemente nunca utilizará muchas de estas instrucciones, siendo las más importantes las que involucren los indicadores de cero, de acarreo y de valor negativo. Todas ellas, a excepción de JMP, utilizan el direccionamiento relativo. Ello significa que un solo byte debe seguir al de instrucción, correspondiente al código de la bifurcación. Se trata a este byte como un número con signo (o sea, en otras palabras, si es mayor que $7F, 127 en base diez, se le considera negativo) que se suma a la dirección que se encuentra en el PC en el momento de ejecutarse la bifurcación. El resultado de esta suma constituye la dirección donde se transfiere el flujo del programa, de forma que la próxima instrucción que se ejecute será la que se encuentre en esta dirección. Este tipo de bifurcación, que utiliza un número de «desplazamiento» de un solo byte, permite un salto de 127 (en base diez) pasos hacia adelante o 128 hacia atrás. Esto es así debido al hecho de que estos valores no pueden exceder un byte con signo.

Empezando a trabajar con el Commodore 64

Es hora ya de empezar a hacer algunas prácticas de programación en código máquina con su Commodore 64. Ello no se reduce al simple problema de entrar las líneas de lenguaje ensamblador como si fueran líneas de BASIC. A menos que tenga conectado un cartucho con un programa ensamblador, el Commodore 64 sacará el mensaje de error «SN ERROR» cuando intente ejecutar estos programas. Como queremos ir poco a poco nos olvidaremos, por el momento, de ensambladores y realizaremos su función «a mano». Esto significa que obtendremos los bytes de código máquina que correspondan a las instrucciones de lenguaje ensamblador a través de la consulta de una tabla. Tendremos que convertir entonces los códigos hexa y los números asociados a los datos en valores decimales. Cargaremos entonces estos números en la memoria del Commodore 64 a través de la instrucción POKE, colocaremos la dirección del primer byte en el

PC del 65Ø2 y veremos lo que pasa. Parece sencillo, pero hay bastantes cosas a tener en cuenta y muchas precauciones a tomar. Para empezar, el Commodore 64 utiliza, tal como hemos visto, parte de su RAM para sus propios fines. Si cargamos un cierto número de bytes en la memoria sin tener en cuenta la parte de la misma que utilizamos, hay muchas posibilidades de que modifiquemos bytes que el Commodore necesita para su funcionamiento o bien que los bytes de nuestro programa se vean afectados por la acción del propio Commodore 64. Lo que necesitamos, pues, es una zona de memoria que esté debidamente protegida para utilizarla para nuestro propio uso.

Esto puede conseguirse aprovechando el hecho de que el Commodore 64 puede desplazar sus programas BASIC de la misma forma en que puede desplazar la tabla de la lista de variables. El desplazamiento más fácil de llevar a cabo es el del final de la memoria. El último byte de RAM del que un programa nuestro puede hacer uso es el 4Ø96Ø (en decimal). Las posiciones de memoria que se encuentran a partir de ésta se utilizan normalmente para almacenar cadenas que no se encuentran en las zonas más bajas de memoria. Ya habíamos constatado este hecho en el capítulo 2. Sin embargo, el Commodore 64 no está programado para no sobrepasar este byte. Lo que se hace es almacenar esta dirección de «fin-de-RAM», en forma de dos bytes, en las posiciones de memoria 55 (byte bajo) y 56 (byte alto). Estas son direcciones de la página cero, comprobándose de forma continua durante la ejecución de un programa BASIC que no se intente utilizar posiciones de memoria más altas que el número almacenado en ellas.

Supongamos, entonces, que modificamos este número. No tenemos por qué alterar ambos bytes ya que si disminuimos en una unidad el byte alto habremos reservado 256 bytes de memoria para nuestras propias aplicaciones. Esto es así debido al hecho de que un número almacenado en forma de dos bytes es igual a 256 veces el valor del byte más significativo más el valor del byte menos significativo. Cuando conectamos el Commodore 64, el número almacenado en la posición 56 es el 16Ø. Así pues, si empleamos POKE 56,159 habremos reservado 256 bytes desde la posición 4Ø7Ø5 a la 4Ø96Ø. Recuerde que van incluidos los dos extremos del intervalo en cuestión. Una vez hecho esto, nos habremos asegurado de que cualquier programa BASIC que ejecutemos no utilizará las posiciones de memoria por encima de la 4Ø7Ø4 y no podrá interferir con nuestros programas de código máquina.

El otro problema que queda pendiente es el de colocar la dirección de inicio del programa en el contador de programa del 65Ø2. Afortunadamente, los diseñadores del Commodore 64 lo han previsto todo. Existe una instrucción BASIC (SYS) que lo hará por usted. SYS debe ir acompañada de un número que es el que se colocará en el PC. Es ésta la dirección que será utilizada, pues, como dirección del

primer byte de su programa. Puede considerarse éste como el «byte inicial». Es posible, sin embargo, escribir programas en los que los primeros bytes sean datos, de forma que el programa empiece, por ejemplo, en el décimo byte. Ello no representa ningún tipo de problema ya que basta, entonces, utilizar para la instrucción SYS la dirección del byte inicial del programa.

Por último (al menos por el momento), hay que asegurarse de que el programa en código máquina finalice de forma apropiada. Nada de lo que hemos visto hasta ahora indicará al 65Ø2 del Commodore 64 dónde termina su programa. Como consecuencia de ello, el 65Ø2 podría continuar leyendo bytes después del final del programa hasta que encontrase alguno que fuese capaz de colapsar la máquina. Este podría ser, por ejemplo, un byte que provocase un bucle sin fin. Algunos programadores dudan de que en estas circunstancias no aparezca ninguna configuración que provoque un bucle sin fin de este tipo. Para volver de forma correcta al sistema operativo del Commodore 64, hay que hacer que cada programa de código máquina termine con una instrucción de «vuelta de subrutina», instrucción cuyo mnemónico es RTS y cuyo código de instrucción es $6Ø.

Existe otro problema del cual no hemos de preocuparnos por el momento. Cuando se ejecuta un programa de código máquina en el Commodore 64 desde un programa BASIC, se está utilizando el mismo microprocesador para ambas tareas. No pueden ejecutarse las dos a la vez, de forma que primero se ejecuta una y después la otra. Si se utilizan los registros del 65Ø2 en el programa de código máquina tal como ocurre en la mayoría de los casos, hay que asegurarse de que no se destruya información que sea necesaria después para el programa BASIC. Así, por ejemplo, si en el momento de empezar a ejecutarse el programa en código máquina los registros del 65Ø2 contenían la dirección de una palabra clave en ROM, tendrá que haber esta misma dirección en los mismos registros cuando finalice el mencionado programa de código máquina. Cuando se ejecuta un programa de código máquina a través de la instrucción SYS, ello se realiza de forma automática. El contenido de los registros del 65Ø2 se guarda en una zona de la memoria RAM denominada «pila». Esto constituye, dicho sea de paso, una buena razón para vigilar en qué parte de la memoria se coloca el código máquina. Si se modifica el contenido de la «pila», puede que el Commodore 64 se enoje. La pila está situada en la zona de memoria comprendida entre la posición 256 y la 511. Cuando se encuentra la instrucción RTS el final del código máquina, los bytes que se almacenaron en la pila se restituyen a los registros originales, continuando la ejecución normal del programa. Si se ejecuta un programa de código máquina por cualquier otro procedimiento que no involucre a la instrucción SYS, habrá que incluir uno mismo esta operación de almacenamiento y restitución del contenido de los registros como

parte del programa en código máquina. Ello implica la utilización de las instrucciones PUSH y PULL que comentaremos más adelante.

Por fin, un ejemplo práctico de programa

Una vez vistos todos estos preliminares podemos empezar a estudiar algunos programas muy simples, pero que están pensados para que se familiarice con la forma en que se guardan los diferentes programas en la memoria del Commodore 64. Adquirirá asimismo alguna experiencia en la utilización del lenguaje ensamblador y del código máquina, así como en la forma en que puede ejecutarse un programa de código máquina.

Empezaremos con el ejemplo más sencillo posible: un programa que lo único que haga sea colocar un byte en memoria. En lenguaje ensamblador adoptaría la forma siguiente:

```
ORG 4Ø7Ø5     ; empezar a colocar los bytes en esta dirección
LDA #$55      ; colocar un 55 hexa en el acumulador
STA $9FC4     ; guardarlo en la posición 9FC4
RTS           ; volver al BASIC
```

La primera línea contiene un mnemónico (ORG) que no habíamos visto hasta ahora. No es ninguna de las instrucciones del 65Ø2 sino que constituye una instrucción para el programa ensamblador que, en este caso, es usted. ORG es una abreviación de ORiGen y constituye una indicación (o recordatorio) de que ésta es la primera dirección de memoria que deja suficiente espacio para programas más extensos que iremos descubriendo a lo largo de este libro, aunque podíamos haber elegido un número mayor. De todas formas, éste nos servirá como cualquier otro, con la ventaja de poder utilizarla para programas más largos. Cuando se programa utilizando un ensamblador, puede entrarse esta línea para que éste vaya colocando automáticamente los bytes del programa a partir de la dirección indicada. De la forma en que trabajamos, con el ensamblado hecho «a mano», esta instrucción sólo constituye un recordatorio de las direcciones que se van a utilizar. Tome nota de los comentarios que siguen a los puntos y comas. El punto y coma se utiliza en lenguaje ensamblador de la misma forma que REM en el BASIC. Todo lo que le sigue constituye sólo un comentario que el ensamblador ignora, pero que puede ser útil para el programador.

Vamos a ver ahora lo que hace el programa. La primera verdadera instrucción es la de cargar el número $55 en el acumulador «A». Utiliza direccionamiento inmediato, de forma que deberá colocarse el $55 inmediatamente después de la instrucción. El símbolo «#» se em-

plea en lenguaje ensamblador para indicar que se está usando direccionamiento inmediato. La línea siguiente establece que se cargue el byte contenido en el acumulador (el $55) en la posición de memoria $9FC4. En decimal, esto es 4Ø9ØØ. Es una dirección bastante por encima de las que utilizaremos para el programa. Evidentemente, no es nada recomendable usar una dirección que vaya a ser utilizada, también, por el programa. Esta instrucción utiliza direccionamiento absoluto. Finalmente, el programa termina con la instrucción RTS que es esencial para asegurarnos que el Commodore 64 continuará correctamente después de que finalice nuestro programa.

La siguiente etapa dentro del proceso de programación consiste en escribir los códigos en hexa. Cada uno de ellos debe obtenerse de una tabla, asegurándonos de escoger el código adecuado al tipo de direccionamiento utilizado. El código para el LDA inmediato es $A9, de forma que éste será el primer byte del programa que se almacenará, pues, en la dirección 4Ø7Ø5 (en base diez). Podemos empezar una tabla que relacione posiciones de memoria con contenidos con la línea:

4Ø7Ø5 $A9

El byte que se quiere cargar es el $55. Este es, pues, el byte que hay que colocar en la siguiente dirección de memoria, ya que es así como trabaja el direccionamiento inmediato. La tabla tendrá ahora el aspecto siguiente:

4Ø7Ø5 $A9

4Ø7Ø6 $55

El siguiente byte que necesitamos es el código de instrucción de STA con direccionamiento absoluto. Este byte es el $8D que debe ir seguido por los dos bytes de la dirección donde quiere almacenarse el contenido del acumulador. La dirección 4Ø9ØØ adopta en hexa la expresión $9FC4, de forma que hay que colocar los bytes $C4 y $9F después de la instrucción STA. Recuérdese que hay que colocar estos bytes en el orden de primero el menos significativo y luego el más significativo. El último código debe ser el correspondiente a la instrucción RTS, que es el $6Ø, de forma que nuestra tabla presentará el aspecto que se muestra en la figura 5.4. Esta tabla emplea las posiciones de memoria que van de la 4Ø7Ø5 a la 4Ø71Ø, o sea seis bytes en total. El resultado del programa será el de cargar un byte en la dirección 4Ø9ØØ (en base diez). Lo que hay que hacer ahora es colocar esta tabla en memoria y ejecutar el programa que contiene.

Se requiere para ello un programa en BASIC que inicialice la memoria y cargue los bytes uno a uno. Antes de escribir este programa, hay que convertir cada byte hexa a base diez ya que la instrucción

4Ø7Ø5	A9
4Ø7Ø6	55
4Ø7Ø7	8D
4Ø7Ø8	C4
4Ø7Ø9	9F
4Ø71Ø	6Ø

Fig. 5.4 Codificación del programa empleando direcciones decimales y bytes de datos en hexa.

POKE de Commodore 64 trabaja sólo con números decimales. Puede realizarse esta conversión mediante el programa que se mostró en el capítulo 3 o bien con la ayuda de una calculadora. El programa BASIC de carga se ilustra en la figura 5.5. A través de POKE 56,159 nos aseguramos de que todas las posiciones de memoria por encima de la 4Ø7Ø4 no serán utilizadas por el Commodore 64. Se declara la variable A como 4Ø7Ø4 para que pueda ser utilizada en las instrucciones POKE. Las líneas 2Ø a 4Ø cargan los códigos numéricos en las posiciones de memoria que se encuentran a partir de la dirección 4Ø7Ø5. ¿Por qué la 4Ø7Ø5? Como hemos utilizado POKE A+N, con A = 4Ø7Ø4 y N = 1, la primera dirección será la 4Ø7Ø5. Todos los códigos deben ponerse en formato decimal para poder ser empleados en el programa BASIC de carga. El único problema que surge es el de tener una dirección decimal y tener que convertirla en dos números de un solo byte en base diez. El método para hacerlo se muestra en el Apéndice B, después del apartado de conversiones de decimal a hexa. Sin embargo, en muchos aspectos es mejor trabajar todo lo que se pueda en hexa y pasar a base diez sólo cuando sea estrictamente necesario. Como hay que emplear el sistema hexa cuando se trabaja con el ensamblador MIKRO (o cualquier otro ensamblador), es aconsejable empezar a familiarizarse con los principios del trabajo en dicho sistema.

```
10 POKE56,159:A=40704
20 FORN=1TO6:READ D%
30 POKEA+N,D%:NEXT
40 SYS40705
100 DATA169,85,141,196,159,96
```

Fig. 5.5 Programa BASIC que carga los bytes en su lugar. Vea como se utiliza el entero %D para los datos y como se ejecuta el programa a través de SYS 4Ø7Ø5.

La última línea del programa, la línea 4Ø, contiene SYS 4Ø7Ø5. Esta es la instrucción BASIC que hará que se ejecute el código máquina a partir de la dirección inicial que se especifique. La línea 1ØØ contiene los seis bytes de datos que obtuvimos anteriormente. Cuando ejecute este programa, no se produce ningún efecto aparente. Esto es así debido al hecho de que no puede verse lo que hay en la posición 4Ø9ØØ de memoria. Si usted pulsa:

```
? PEEK(4Ø9ØØ)
```

debería obtener el valor 85 que constituye la versión decimal del $55, valor que almacenó ahí el programa. Pruebe ahora de entrar POKE 4Ø9ØØ, 255 seguido de RETURN y borre la línea 4Ø de su programa, esto es la línea correspondiente a la instrucción SYS. Ejecute otra vez el programa y entre ?PEEK(4Ø9ØØ) para ver lo que hay almacenado ahí. Debería haber un 255. Pulse ahora SYS 4Ø7Ø5 seguido de RETURN. A través de ?PEEK(4Ø9ØØ) debería obtener esta vez un 85. Esto es así debido a que cargando sólo los bytes del programa en memoria no se provoca la ejecución del mismo. De ello se encarga la instrucción SYS. Así pues, pueden cargarse valores en memoria en las primeras etapas de un programa BASIC y hacer uso de ellos más tarde, cuando usted quiera, a través de la mencionada instrucción SYS.

Este programa no es, desde luego, nada ambicioso. No hace más que la función que realizaría la instrucción POKE 4Ø9ØØ,85 en BASIC, pero algo es algo. Llegados a este punto, lo principal es haberse habituado a la forma en que opera el código máquina, en cómo cargarlo en memoria y en cómo ejecutarlo. Otro aspecto a considerar es el hecho de que el código máquina esté a salvo en la memoria. Si usted entra NEW(RETURN), desaparecerá el programa BASIC, pero el código máquina permanecerá. Si carga un 255 en la posición de memoria utilizada a través de POKE 4Ø9ØØ,255 y comprueba el valor ahí almacenado a través de ?PEEK(4Ø9ØØ), verá que puede alterarse el contenido de esta dirección de memoria a través de SYS 4Ø7Ø5. Estos bytes continuarán ahí hasta que usted haga algo para hacerlos desaparecer o bien desconecte el ordenador. Si se quiere, puede guardarse en cinta el programa de código máquina, aunque ésta es una técnica que estudiaremos más adelante. ¡Poco a poco, por favor! Otro punto que dejaremos para más tarde es el método alternativo de llamada de un programa a través de la instrucción USR.

Vamos a intentar algo más ambicioso por lo que a la utilización del código máquina se refiere, a pesar de seguir en la línea de los ejemplos sencillos. La figura 5.6 muestra la versión del programa propuesto en lenguaje ensamblador. Lo que se quiere hacer es cargar un byte en el acumulador, decalarlo un lugar hacia la izquierda y colocarlo otra vez en la memoria, esta vez en la dirección inmediatamente

```
LDX #$Ø          A2 ØØ
LDA $9FC$,X      BD C4 9F
ASL A            ØA
INX              E8
STA $9FC4,X      9D C4 9F
RTS              6Ø
```

Fig. 5.6 Programa en ensamblador para «multiplicar por dos». El listado muestra el ensamblador en la izquierda y los códigos hexa a la derecha.

superior a aquella de donde se obtuvo el byte original. Vamos a utilizar el direccionamiento indexado, de forma que empezaremos colocando un cero en el registro X mediante la instrucción LDX #$ØØ. Tal como vimos anteriormente, el signo # identifica el empleo del direccionamiento inmediato. La línea siguiente, LDA $9FC4,X indica que hay que cargar el acumulador con el contenido de la dirección $9FC4 después de sumarle el valor del registro X. Eso hace que se efectúe la lectura en la posición $9FC4 (4Ø9ØØ en base diez). El tercer paso lo constituye ASL A, un decalaje aritmético hacia la izquierda del byte contenido en el acumulador, con lo que los bits del mencionado registro se desplazan un lugar hacia la izquierda. A continuación, y en cuarto lugar, incrementamos el contenido del registro X a través de INX. Con ello se transforma el Ø que contenía en un 1. Acto seguido, el byte del acumulador se almacena en la dirección $9FC5 a través de STA $9FC4,X. Esta vez, como se había incrementado el contenido del registro X, el byte queda almacenado en la posición $9FC5 de memoria. Terminamos, como siempre, con la instrucción RTS.

Podemos pasar ya todo esto al formato de los códigos numéricos. Es tan fácil como vimos en el ejemplo anterior, a pesar del empleo del indexado. La instrucción LDX requiere el código de carga inmediata consistente en un $A2, al que debe seguir el byte del dato, el $ØØ. El LDA con direccionamiento indexado se codifica como $BD que debe ser seguido, a su vez, por los dos bytes de la dirección $9FC4, en el orden de menos a más significativo. El código de ASL es ØA, llevándose a cabo el incremento del registro X con INX, cuyo código es el $E8. Entonces, devolvemos el byte del acumulador a la memoria a través de STA $9FC4,X. El código ligado a STA indexado es $9D al que siguen los ya familiares dos bytes de la dirección. Finalmente, corresponde el código $6Ø a la instrucción RTS.

Hay que codificar ahora todo esto en BASIC. Si elegimos un valor reducido para el contenido de la posición de memoria $9FC4, el efecto del decalaje a la izquierda consistirá en doblar su valor, de forma que puede utilizarse este programa como base de una aritmética un tanto

```
10 POKE 56,159:A=40704
20 FORN=1TO11:READ D%
30 POKEA+N,D%:NEXT
40 POKE40900,7:SYS40705
50 PRINT"DOS VECES";PEEK(40900);
55 PRINT"ES IGUAL A";PEEK(40901)
100 DATA162,0,189,196,159,10
105 DATA232,157,196,159,96
```

Fig. 5.7 Programa BASIC que carga los bytes en su lugar y utiliza a continuación el programa de código máquina.

especial. El programa en BASIC se muestra en la figura 5.7. Se empieza, como es habitual, inicializando el espacio de memoria. No hay que preocuparse de si había anteriormente un programa en esta zona de memoria. El nuevo programa lo sustituirá completamente y con tal de que el nuevo programa termine con el byte correspondiente a la instrucción RTS, los bytes ligados al programa antiguo no interferirán con los del nuevo. Los valores se van cargando en el lugar correspondiente en la forma habitual en las líneas 2Ø y 3Ø. Sin embargo, en la línea 4Ø colocamos un número (7 en base diez) en la dirección $9FC4 (4Ø9ØØ en base diez). Esta es, pues, la dirección que será utilizada por el programa, colocándose en ella el valor 7 (ØØØØ Ø111 en binario). En la segunda parte de la línea 4Ø se ejecuta el código máquina a través de la instrucción SYS 4Ø7Ø5, lo que debería provocar que se desplazase un lugar hacia la izquierda el byte en cuestión, obteniéndose el valor ØØØØ 111Ø. En base diez, ello equivale al valor 14, o sea dos veces 7. La línea 5Ø escribe el resultado y la línea 1ØØ contiene los bytes de datos. Todo ello es bastante sencillo, pero si no supiese nada de código máquina seguramente se preguntaría cómo demonios queda el valor original multiplicado por dos. Una vez más, el programa no hace nada que no pudiese hacerse más fácilmente e igual de rápido en BASIC. Lo importante, desde nuestro punto de vista, es que haya utilizado el direccionamiento indexado, así como la instrucción de decalaje, y que haya adquirido un poco más de experiencia colocando un programa de código máquina en la memoria del Commodore 64 a través del más rudimentario de los métodos que existen para ello. Hay que hacer notar, dicho sea de paso, que si se cometieron errores, especialmente con los bytes de datos, es poco probable que el Commodore 64 detecte esta circunstancia y se niegue a continuar. Cuando haya entrado un programa BASIC de este tipo (que va cargando bytes en memoria) es aconsejable grabar el programa en cuestión antes de ejecutarlo. De esta forma, si el efecto de un byte incorrecto es inutilizar la mitad de la RAM, puede desconectarse y conectarse otra vez el ordenador y cargar otra vez a continuación el progra-

ma. Si no lo grabó, tendrá que entrarlo todo a mano otra vez. Es un trabajo duro, pero la vida es así. Otra consideración final afecta al número almacenado en la posición 4Ø9ØØ. Si se trata de un valor pequeño, el programa funcionará correctamente. Sin embargo, no se puede guardar un número mayor que 255 en un solo byte. Además, si el número que se utiliza es mayor que 127, el resultado del programa podrá parecer sorprendente. Ello es así debido a que un número mayor que 127 es considerado como un número negativo. Las rutinas qué multiplican números en su Commodore 64 son bastante más sofisticadas que este sencillo ejemplo.

6. Diseño de programas más complejos

Los sencillos programas que vimos en el capítulo 5 no son demasiado útiles aunque sí son prácticos para ejercitarse en la forma en que se escriben los programas en código máquina. El practicar en la utilización del lenguaje ensamblador así como en su conversión a código máquina es fundamental, una vez llegados a este estadio, ya que es mucho más fácil ver si se está cometiendo algún error cuando se trata de programas tan sencillos. No es fácil detectar un error en un programa de código máquina que sea largo, especialmente si se está luchando aún por aprender el lenguaje.

La mayoría de las dificultades de los principiantes provienen, sin embargo, del hecho de que sea tan sencillo el código máquina más que por el hecho de que sea difícil de aprender. Debido a la circunstancia de ser tan sencillo el código máquina, es necesario una gran cantidad de instrucciones para conseguir hacer algo útil y cuando un programa contiene gran cantidad de instrucciones, es más difícil de diseñar. La parte más difícil de este diseño consiste en descomponer el programa en una serie de elementos que agrupen, desde el punto de vista lógico, varias instrucciones de ensamblador. Para esta etapa del diseño, los diagramas de flujo son, como mucho, la forma más fácil de seguir el buen camino. No puede afirmarse que los diagramas de flujo sean idóneos para el diseño de programas en BASIC, pero sí demuestran todo lo que se espera de ellos cuando se utilizan para planificar programas en código máquina.

Los diagramas de flujo

Los diagramas de flujo son a los programas lo que los diagramas de bloques son a la electrónica: muestran lo que hay que hacer sin descender a un detalle mayor que el que sea estrictamente necesario. Un diagrama de flujo consiste en una serie de figuras, cada una de ellas con una forma que simboliza un determinado tipo de acción. La figura 6.1 muestra algunos de los bloques de los diagramas de flujo que se demostrarán más útiles para nuestros propósitos (tomados de entre el conjunto estándar de formas aceptadas para los diagramas de flujo). Estos elementos son el bloque terminal (inicio o final), el de

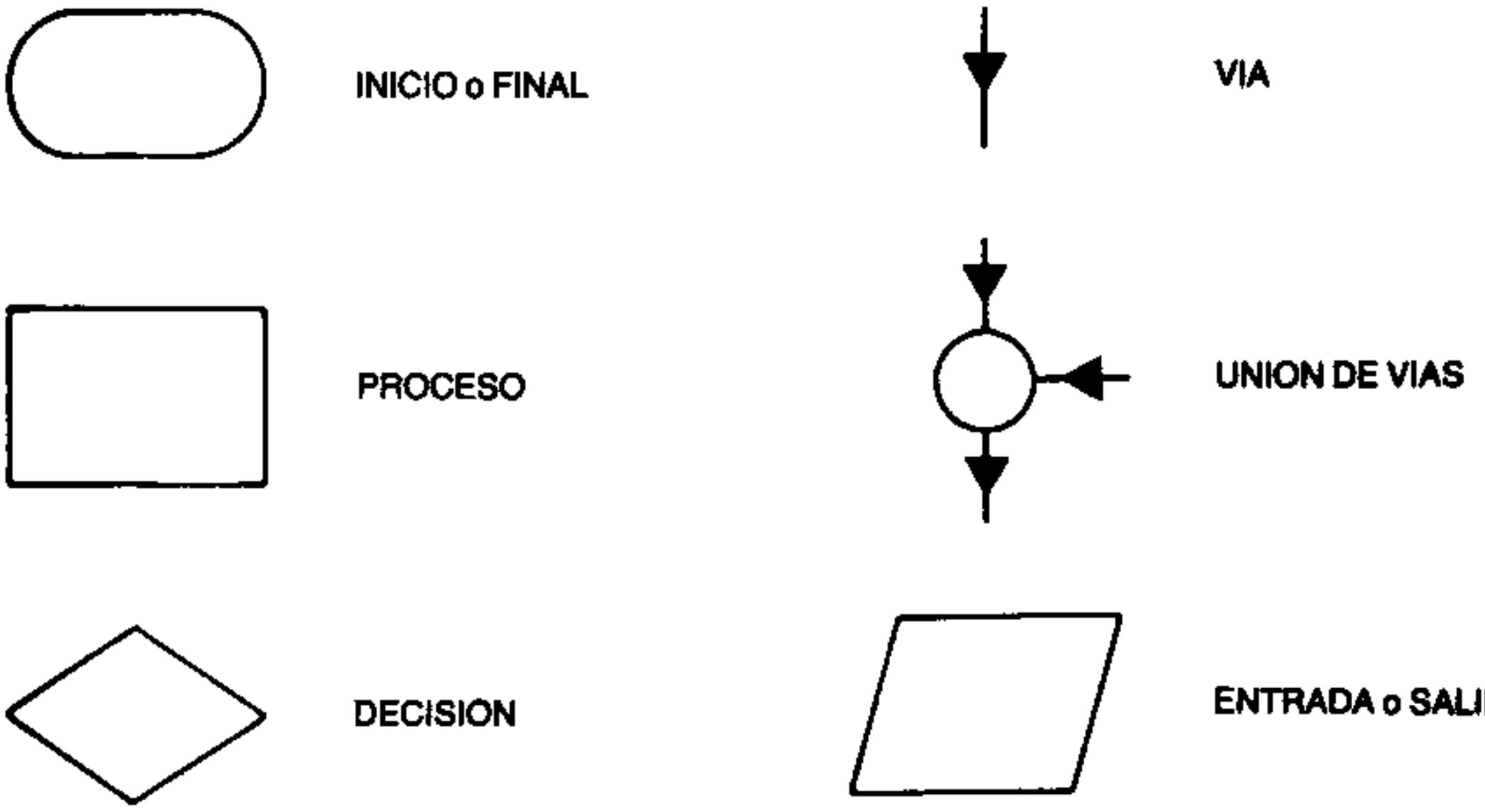


Fig. 6.1 Principales bloques de un diagrama de flujo.

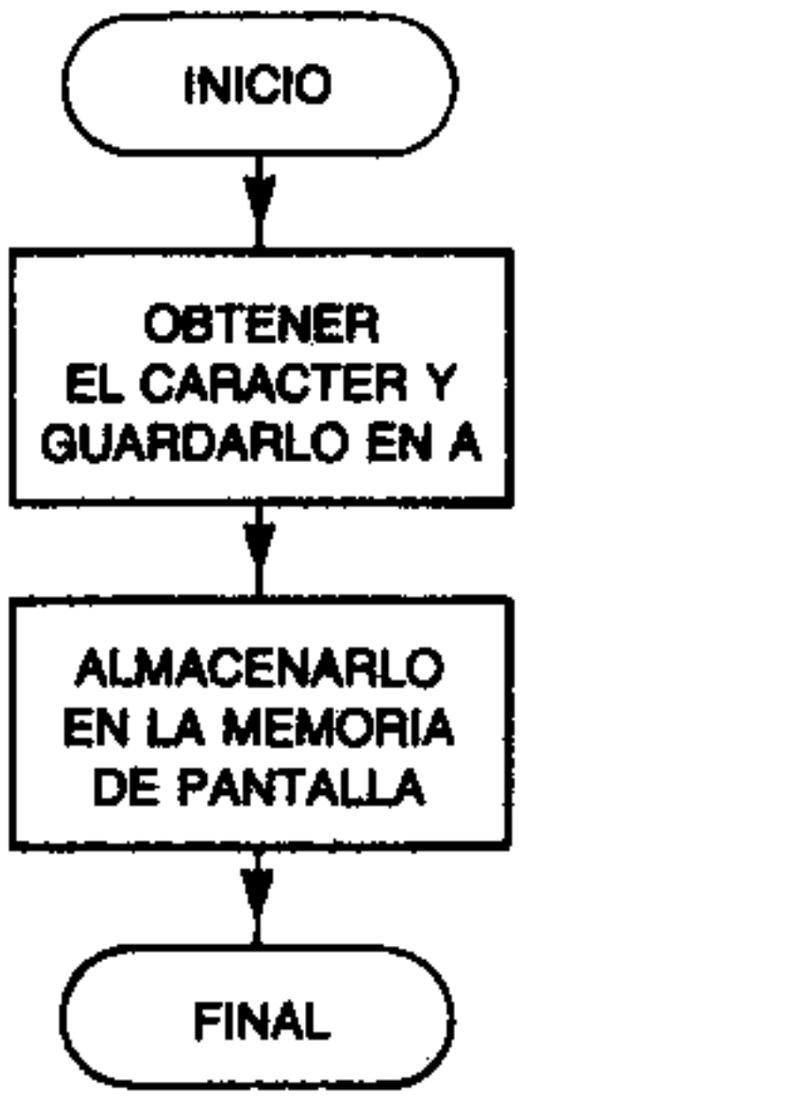


Fig. 6.2 Diagrama de flujo de un programa para escribir un carácter.

proceso (o acción), el de entrada/salida y el de toma de decisiones. Dentro de las mencionadas figuras podemos escribir breves anotaciones acerca de la acción a la que nos queremos referir, aunque sin entrar en detalles excesivos. Un ejemplo constituye siempre la mejor forma de ver como se emplea un diagrama de flujo. Supongamos que se quiere hacer un programa en código máquina que adquiera el código ASCII de una tecla que se pulse y saque por pantalla el carácter asociado a dicha tecla. En la figura 6.2 se muestra un diagrama de flujo que refleja este proceso. El primer bloque terminal es «INICIO», ya que cualquier programa o fragmento de programa debe empezar, lógicamente, en algún sitio. De él parte una flecha que conduce al

primer bloque «de acción» que ostenta la etiqueta «obtener el carácter y guardarlo en A». Se describe así lo que se quiere hacer: almacenar el código numérico del carácter en el acumulador. No se sabe, por el momento, lo que se hará con él. Ello vendrá más tarde. Después de hacernos con el carácter, la flecha apunta a la acción siguiente: guardar el carácter en la memoria de pantalla. Así es como se lleva a cabo la acción de «escribir»: es algo que ya habíamos visto con anterioridad. El bloque terminal FINAL nos recuerda que éste es el fin del programa y que no se trata de un bucle sin fin.

Este es un diagrama de flujo muy sencillo, aunque es suficiente para ilustrar lo que queremos dar a entender. Hay que hacer notar que las descripciones de las acciones son muy generales (no se coloca una sola instrucción de lenguaje ensamblador dentro de ninguno de los bloques utilizados). Hablando de forma estricta, no deberíamos habernos referido al acumulador A en el bloque de «obtener el carácter y guardarlo en A», aunque podemos justificarnos por la necesidad de recordar donde debe almacenarse el código en cuestión. Un diagrama de flujo debe bastar para que el que lo vea pueda hacerse una idea de lo que se quiere hacer. Nunca debe ser algo que sólo el diseñador del programa pueda comprender y utilizar y que confunda a los demás. Un buen diagrama de flujo es, de hecho, aquel que puede ser utilizado por cualquier programador para escribir el programa en cualquier tipo de código máquina (o en cualquier «lenguaje» tal como BASIC, FORTH, PASCAL, etc.). Sin embargo, muchos diagramas de flujo se escriben después de haber escrito el programa (normalmente después de muchas pruebas y errores), con la esperanza de que ayuden a clarificar su acción. No lo consiguen y probablemente usted no intentaría hacer algo semejante ¿no es así?

Una vez que se tiene un diagrama de flujo, puede comprobarse que haga lo que se espera de él estudiándolo con detenimiento. En el ejemplo utilizado, las acciones de «obtener el carácter» y «guardarlo en la memoria de pantalla» serán llevadas a cabo a través de código máquina, de forma que nos centraremos en ellas. Obtener el código ASCII de un carácter puede parecer, en un principio, complicado. Sin embargo, muchos ordenadores colocan el código ASCII del último carácter empleado en una determinada posición de memoria. Es ahí donde se revela de gran utilidad un buen conocimiento de cómo el Commodore 64 emplea su memoria. El Apéndice E muestra alguna de estas importantes direcciones. Una de particular interés es la posición 512 (decimal), o sea $2Ø0. Esta es la dirección de inicio del «buffer del teclado». Un buffer es una zona de RAM que utiliza el ordenador para guardar información de forma temporal. Tal como sugiere su nombre, el buffer del teclado es utilizado para almacenar en él los códigos numéricos asociados a las teclas pulsadas. Estos códigos permanecen en dicho buffer hasta que se pulse la tecla RETURN. Pulsando esta

tecla se transfiere todo el buffer del teclado a otra zona de memoria. La importancia de esta dirección estriba en que puede ser utilizada para ver el código numérico de la última tecla pulsada. Si cargamos el acumulador con el contenido de esta dirección podemos utilizar esta información para llevar a cabo lo que queríamos. El primer paso parece ser, pues, cargar el acumulador, mediante el método de direccionamiento adecuado, de forma que pueda utilizarse la información contenida en la posición 512 de memoria.

El segundo punto a llevar a cabo, el almacenar el byte en la memoria de pantalla, es inmediato. La memoria que utilizamos es la denominada «memoria de texto» que ocupa las posiciones de memoria comprendidas entre la dirección 1Ø24 y la 2Ø23 (en base diez). ¿Qué le parece colocar el carácter en el centro de la pantalla, en la posición 1524? ¿Desea saber cómo obtuvimos este valor? Pues bien, si consideramos el rango 1Ø24 a 2Ø23, vemos que comprende 1ØØØ posiciones, incluyendo la primera y la última. La mitad son 5ØØ, de forma que si sumamos 5ØØ a 1Ø24 obtenemos 1524, que debe corresponder, pues, a la posición asociada al punto medio de la línea central de la pantalla. Así una instrucción del tipo «almacenar el acumulador en la posición 1524 de memoria» debería llevar a cabo lo que queríamos.

Podemos pasar ya al diseño de la versión en lenguaje ensamblador del código que vamos a utilizar. Siguiendo el camino que apuntábamos ya anteriormente, podemos hacer que el código máquina en cuestión empiece en la posición 4Ø7Ø5 (en base diez). Con ello, dicho código adoptará el aspecto siguiente:

```
ORG 4Ø7Ø5
LDA $Ø2ØØ
STA $Ø5F4
RTS
```

Ahora podemos ensamblar a mano las anteriores instrucciones, determinando los códigos numéricos necesarios para que funcione el anterior programa. Estos códigos —todos ellos en hexa— son:

```
AD ØØ Ø2
8D F4 Ø5
96
```

Si los convertimos a base diez tendremos, por fin, los valores que podremos utilizar para formar ya nuestro programa de «escritura en pantalla». Podemos ponerlos en forma de un programa BASIC que cargue estos códigos en memoria y ejecute a continuación el programa de código máquina que representan. Tendrá el aspecto de la figura 6.3, con lo que podemos entrar ya dicho programa y ejecutarlo. Otro pequeño avance para el usuario del Commodore 64.

Bien, la cosa funciona, pero no de la forma que esperábamos. El

```
10 POKE56,159:A=40704
20 FORN=1TO7:READ D%
30 POKEA+N,D%:NEXT
40 GET A$:IF A$=""THEN 40
50 SYS40705
60 POKE53281,243
100 DATA173,0,2,141,244,5,96
```

Fig. 6.3 Programa BASIC de carga para el programa de escribir un carácter.

mismo programa nos indica lo que hay que añadir. Para empezar, necesitamos algún procedimiento para obtener un carácter y almacenarlo en la primera dirección del buffer (la 512, o sea, $2ØØ). El programa utiliza para ello el bucle GET A$ de la línea 4Ø. El segundo punto a considerar es el hecho de que no aparece nada en pantalla hasta que se han seleccionado los colores, lo que se lleva a cabo en la línea 6Ø. Si entra este programa y lo ejecuta, funcionará. Para hacer que funcione una segunda vez, habrá que restaurar los registros asociados al color pulsando simultáneamente las teclas STOP/RESTORE. Si no lo hace así, el programa seguirá funcionando correctamente, pero no podrán apreciarse sus efectos en pantalla.

En nuestro estado actual de conocimientos, realizamos el bucle de espera del carácter en BASIC, llamando al programa de código máquina tan pronto como detecta el BASIC que se ha pulsado una tecla. En lugar de dedicar más tiempo a este procedimiento mixto, vamos a intentar hacerlo todo en código máquina, aún con el riesgo de cometer errores.

Bucles en ensamblador

Como nos enfrentamos a un programa sencillo, parece ser ésta una buena oportunidad para introducirnos en los bucles. Si ha hecho algún programa que involucre algo más que las instrucciones más elementales del BASIC, sabrá lo que significa un bucle. Existe un bucle cuando puede hacerse que se repita un fragmento de programa una y otra vez hasta que se dé una condición predeterminada. En BASIC puede provocarse un bucle mediante una instrucción del tipo:

```
2ØØ IF A = Ø THEN GOTO 1ØØ
```

En ella existe una comprobación (es A = Ø ?) y si se cumple la condición que lleva implícita (o sea, si, efectivamente, es A igual a Ø), el programa retrocede a la línea 1ØØ y repite todas las instrucciones

que existan entre esta línea y la línea 2ØØ. Este tipo de bucle en BASIC se corresponde bastante exactamente con los bucles que pueden crearse en código máquina. Sin embargo, en lugar de utilizarse números de línea, en este caso se utilizan direcciones de memoria. En lugar de comprobarse el valor de una variable «A» se comprueba el valor del contenido de un registro que puede ser en este caso el registro «A».

Vamos a empezar en la forma correcta mediante un diagrama de flujo. La figura 6.4 muestra el aspecto que puede adoptar. El primer paso es el mismo que vimos en el caso anterior: obtener el código del carácter y cargarlo en el acumulador A. Sin embargo, el punto siguiente es un bloque de decisión. La pregunta es «¿se trata de un Ø?». En los diagramas de flujo hay que establecer todos estos bloques de decisión de forma que admitan sólo dos respuestas posibles: si o no. Eso se refleja en el hecho de partir dos flechas del mencionado bloque. Una de ellas lleva la etiqueta «SI». Conduce al principio del programa, al lugar donde se carga el byte de memoria en el acumulador. ¿Por qué? Porque si nos encontramos con que tenemos un cero en el acumulador, significa que no se ha pulsado tecla alguna y hay que volver atrás e intentarlo otra vez. El otro camino, el que lleva la etiqueta «NO», conduce a la siguiente acción: guardar el contenido del acumulador en la memoria de pantalla.

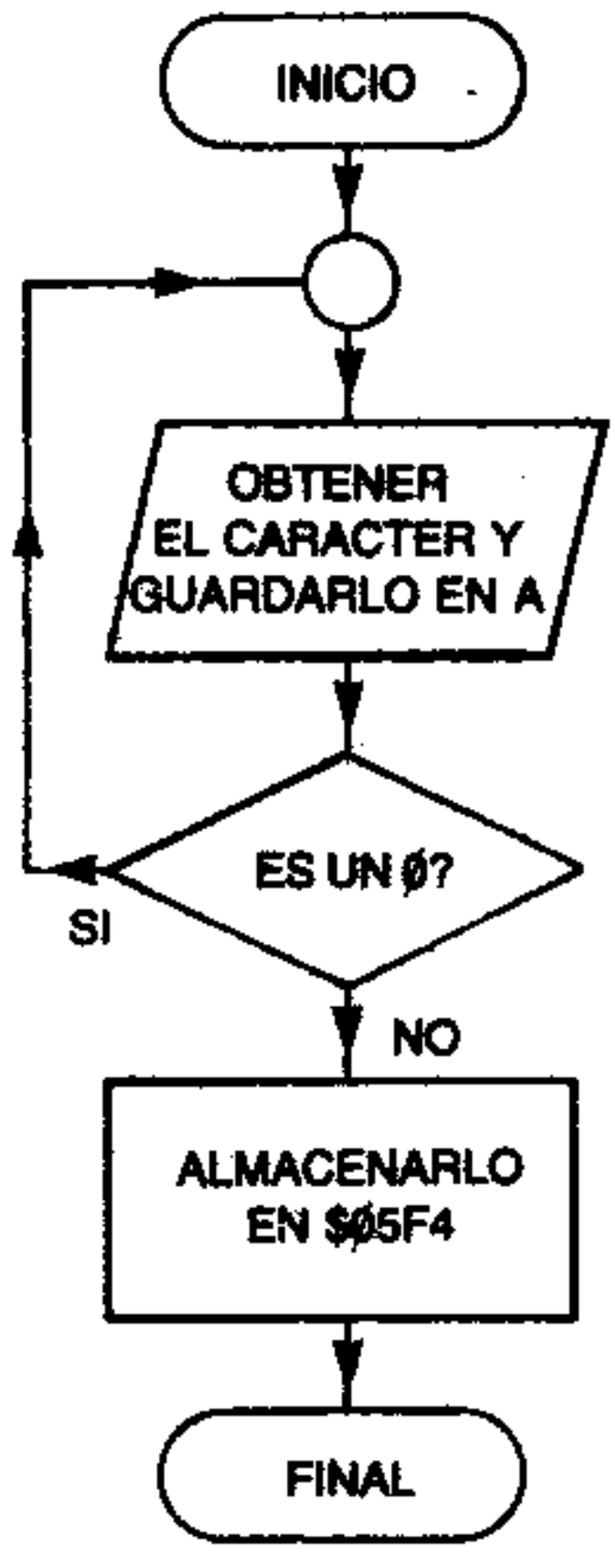


Fig. 6.4 Otro diagrama de flujo : éste tiene un bucle que rechaza el carácter nulo.

Así pues, la acción consistirá en cargar el acumulador con el contenido de la posición 512 y ver si es un cero. Si lo es, se repite la carga del byte. Si no es cero (lo que significa que se pulsó una tecla), se almacena el byte en la memoria de pantalla. Hemos de pasar ahora todo esto a lenguaje ensamblador, lo que hará que nos enfrentemos a nuevos problemas.

```
BUCLE:  LDA $Ø2ØØ
        BEQ BUCLE
        STA $Ø5F4
        RTS
```

Fig. 6.5 Programa en lenguaje ensamblador correspondiente al diagrama de flujo.

La figura 6.5 muestra un programa en lenguaje ensamblador que debería hacer lo que señala el diagrama de flujo que hemos visto. Con respecto al último programa que vimos, apreciamos dos diferencias: se ha añadido una nueva instrucción y se ha modificado otra. La nueva instrucción es la «BEQ BUCLE», y la que se ha modificado es la primera, a la que se le ha añadido al principio BUCLE: Esta palabra BUCLE es una «etiqueta». Se la utiliza aquí en lugar de una dirección y representa la dirección en que empieza la instrucción. Con la palabra BUCLE: colocada delante de la instrucción LDA 512, esta palabra pasa a adoptar el valor de la dirección de memoria donde está almacenado el byte de instrucción de LDA. De esta forma, empleando palabras, evitamos tener que pensar en términos de direcciones numéricas hasta que escribamos el código máquina. Si utilizamos un ensamblador, lo más normal es que no tengamos que preocuparnos en absoluto de direcciones numéricas: el ensamblador colocará de forma automática las direcciones que correspondan en lugar de las etiquetas. La misma etiqueta se utiliza en la línea siguiente. BEQ significa «saltar si el registro es igual a cero» (del inglés Branch if the register is EQual to zero), de forma que el efecto de BEQ BUCLE es que el programa retroceda a la dirección de la instrucción LDA si el acumulador contiene un cero. Es como utilizar una especie de BASIC que permite usar variables en lugar de números de línea (tal como ocurre, de hecho, con algunos BASIC).

En lenguaje ensamblador, todo esto parece claro y directo. Si utilizásemos un ensamblador, sí que sería directo, pero si ensamblamos a mano no es tan sencillo. La razón estriba en que hay que acompañar a la instrucción BEQ de un byte que refleja la posición de la instrucción LDA. Se trata de una forma de direccionamiento relativa al PC, de

Destino: Dirección a la que se quiere saltar (que en la instrucción de ensamblador, tiene una etiqueta precediéndola).

Origen: Dirección desde donde se quiere saltar (dirección del código de bifurcación: en ensamblador, es la instrucción que ostenta el nombre de la etiqueta *después* de ella).

Desplazamiento: Destino menos origen menos 2, expresado en formato hexa.

Fig. 6.6 Fórmula para encontrar el valor de un byte de desplazamiento.

forma que tenemos que utilizar un byte con signo que se sumará a la dirección del contador de programa para obtener la dirección de la instrucción LDA. La fórmula se muestra en la figura 6.6. Todo lo que hay que hacer es hallar la dirección a donde se quiere saltar y la dirección de la instrucción que llevará a cabo el salto. Halle la diferencia entre las dos y a continuación reste 2 del resultado. Lo que obtendrá es el valor del «byte de desplazamiento» que debe acompañar a la instrucción de bifurcación. Como es un número negativo, habrá que pasarlo al formato de byte con signo, utilizando el procedimiento que vimos anteriormente.

```
4Ø7Ø5    173
4Ø7Ø6    Ø
4Ø7Ø7    2
4Ø7Ø8    24Ø
4Ø7Ø9    byte de desplazamiento
```

La dirección de «origen» es la 4Ø7Ø8, donde está el byte correspondiente a BEQ.
La dirección de «destino» es la 4Ø7Ø5, la correspondiente a la instrucción LDA. El procedimiento es:

Dirección de destino − dirección de origen = 4Ø7Ø5 − 4Ø7Ø8 = −3
Restar ahora otro 2, de forma que −3 −2 = −5

En base diez, el byte equivalente para −5 es 256 − 5 = 241. Este es el byte que habrá que colocar en la dirección 4Ø7Ø9.

Fig. 6.7 Un ejemplo de cómo calcular un byte de desplazamiento.

Si todo esto le parece un poco complicado, eche una ojeada al ejemplo práctico de la figura 6.7. Suponiendo que queremos colocar el primer byte del programa en la dirección 4Ø7Ø5, la dirección de la instrucción BEQ será la 4Ø7Ø8. El número 4Ø7Ø8 corresponderá,

pues, a la dirección de origen, o sea de donde vinimos, y el número 4Ø7Ø5 corresponderá a la dirección de destino, o sea a donde vamos. Restemos la dirección de origen de la dirección de destino. Obtenemos un −3. Restemos un 2 a este valor y obtenemos un −5, que en hexa es FB. Este es, pues, el byte de desplazamiento que hay que colocar después del código de instrucción del BEQ.

```
10 POKE56,159:A=40704
20 FORN=1TO9:READ D%
30 POKEA+N,D%:NEXT
40 GET A$:IF A$=""THEN 40
50 SYS40705
60 POKE53281,243
100 DATA173,0,2,240,251,141,244,5,96
```

Fig. 6.8 Programa BASIC para el código de ensamblador de la figura 6.5.

Cuando pruebe el programa de la figura 6.8, verá que, efectivamente, funciona, lo cual nos proporciona un dato importante acerca del comportamiento del Commodore 64. Muchos ordenadores no podrían ejecutar un programa como éste. Si ha adivinado el porqué, ¡enhorabuena!, lo que ocurre es que el programa en cuestión permanece en un bucle mientras no se coloque un código numérico en la posición 512, lo cual no sucederá hasta que no se pulse una tecla. Sin embargo, muchos ordenadores, si dedican el microprocesador a ir iterando dentro del bucle, no pueden, simultáneamente, estar vigilando el teclado, a la espera de que se pulse una tecla. Hay un solo microprocesador en el Commodore 64, y es él quien debe hacerlo todo. Sin embargo, lo que sucede es que no emplea todo su tiempo en el programa que ejecuta. Periódicamente interrumpe lo que está haciendo para ir a comprobar el teclado. Si se ha pulsado una tecla, coloca en el buffer el código numérico que corresponda. Este tipo de acción se conoce, muy acertadamente, con el nombre de interrupción, siendo un proceso muy útil. Su utilización hace mucho más difícil que el Commodore 64 quede colapsado como consecuencia de un programa que tenga un bucle incorrecto. Además, el microprocesador no debe ocuparse de mantener la información en pantalla. Si ocurriese así se vería como desaparecía el contenido de la pantalla cuando se ejecutase este programa.

¿Existe otra alternativa? Efectivamente: podemos escribir un trozo de programa que atienda la lectura del teclado y colocarlo dentro de nuestro bucle. Ello es posible, pero consume mucho tiempo y requiere un profundo conocimiento del Commodore 64. Parece absurda,

de todos modos, esta posibilidad si consideramos que debe haber una rutina en la ROM del Commodore 64 que haga esta misma función. Efectivamente, existe, y podemos encontrar su dirección en el Apéndice E, junto con las direcciones asociadas a otras importantes rutinas. Así, por ejemplo, la que empieza en la dirección $E112, estudia si se ha pulsado alguna tecla. Si no es así, pone un cero en el acumulador. Si se ha pulsado una tecla, el número que deja en el acumulador es el código numérico asociado a la tecla en cuestión. Con esta rutina, no hace falta que utilicemos ninguna posición de memoria, tal como hemos venido haciendo con la 512.

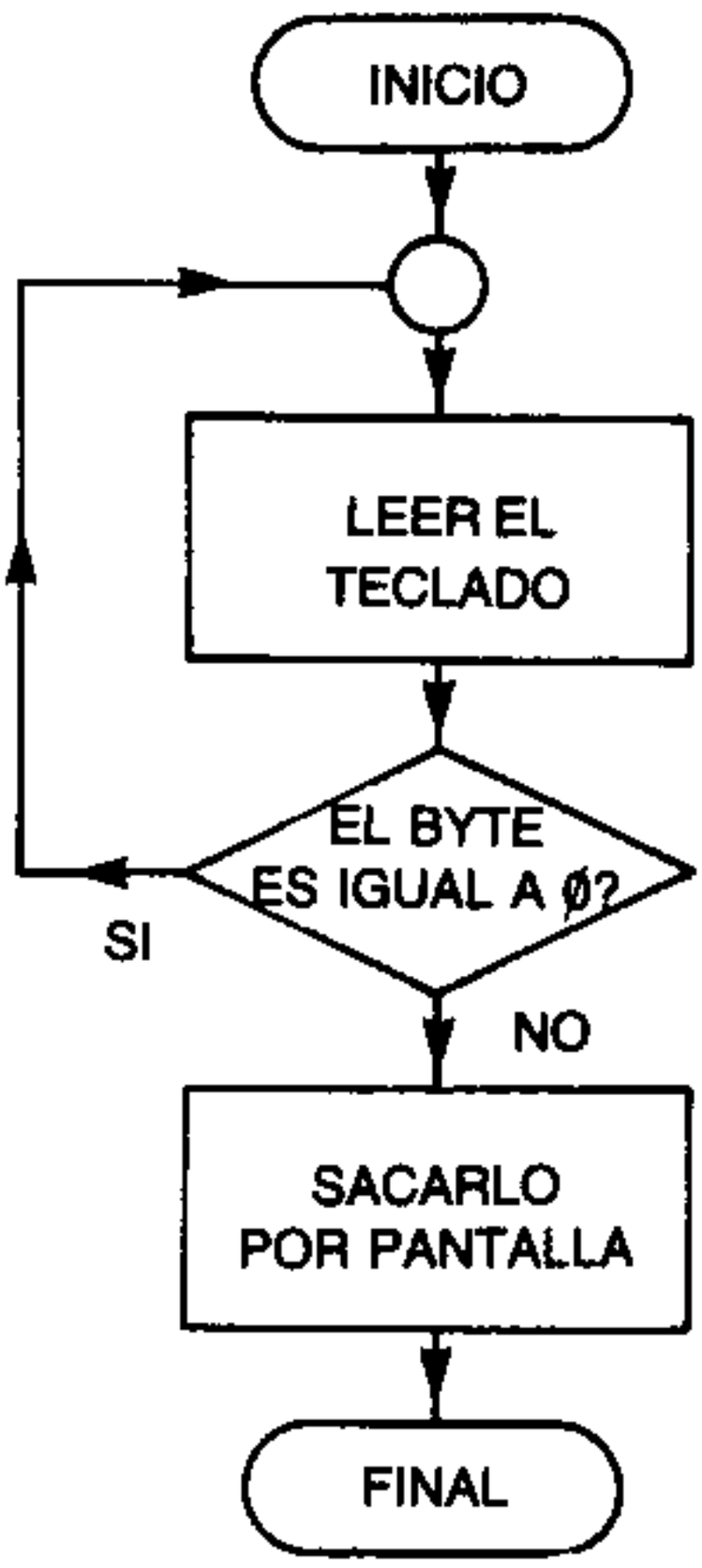


Fig. 6.9 Diagrama de flujo para un programa de escritura de caracteres.

El siguiente paso consiste, pues, en ver como podemos utilizar la rutina que empieza en la dirección $E112. La instrucción que necesitamos se denomina «saltar a la subrutina», y su mnemónico es JSR (del inglés Jump to SubRoutine). Cada subrutina que se encuentra en la zona de ROM, termina con el código asociado a RTS, lo que hace que se vuelva al programa que la llamó. Así, por ejemplo, si empleamos JSR seguido de la dirección $E112, se ejecutará la subrutina, y acto seguido volveremos a nuestro programa. La figura 6.9 muestra un diagrama de flujo de lo que intentamos hacer. Llamaremos a la subrutina para colocar el byte en el acumulador y comprobar después su valor. Si el byte es cero, volveremos al bloque de «leer el teclado».

```
BUCLE:    JSR $E112 ; teclado
          BEQ BUCLE
          JSR $E10C ; pantalla
```

Fig. 6.10 Versión de lenguaje ensamblador del programa.

```
10 POKE56,159:A=40704
20 FORN=1TO9:READ D%
30 POKE A+N,D%:NEXT
40 SYS40705
100 DATA32,18,225,240,251,32,12,225,96
```

Fig. 6.11 Programa BASIC de carga.

Si no lo es, usaremos otra subrutina: la encargada de sacar por pantalla el carácter ligado al código almacenado en el acumulador. Hasta ahora no hay ningún problema. La figura 6.10 muestra la versión en ensamblador del proceso descrito en el diagrama de flujo, donde se utiliza una vez más la palabra BUCLE para representar la dirección donde debe saltar el programa si el byte contenido en el acumulador es cero. La figura 6.11 muestra el mismo programa en forma de un conjunto de instrucciones POKE de BASIC. Cuando se ejecuta, al pulsar una tecla cualquiera, se verá aparecer en pantalla, en el lugar ocupado por el cursor, una letra o bien otro carácter cualquiera. No parece que esto sea muy distinto de lo que ocurre habitualmente cuando se teclea un carácter cualquiera, con lo que podemos preguntarnos ¿cómo sabemos que el programa funciona? Es fácil: basta con ejecutarlo y pulsar una tecla. Acto seguido pulse RETURN. Verá aparecer otra vez en pantalla el mismo carácter que antes, junto con el mensaje de READY en la línea siguiente. No aparece ningún mensaje de error del tipo ?SYNTAX ERROR tal como ocurriría si en una situación normal se pulsara una tecla seguida de RETURN. ¿Por qué aparece repetido el carácter? Porque la rutina de lectura coloca también el código de la letra en el buffer del teclado, con el que tiene mucho que ver la acción de pulsar RETURN. No aparece el mensaje ?SYNTAX ERROR debido a que hemos cortocircuitado la rutina. Hemos incorporado muchos conceptos a este breve programa, de forma que quizás es un buen momento para volver con mayor detenimiento sobre todo lo que hemos visto, antes de enfrascarnos con otros temas.

Más acerca de los bucles

El bucle que hemos utilizado era uno muy sencillo, de un tipo que se conoce con el nombre de «bucle de espera». Su misión es la de hacer que el programa repita una determinada acción hasta que se dé una circunstancia en concreto. Es hora ya de echar un vistazo a otro tipo de bucle, el denominado *bucle de cuenta.* Su importancia es doble: constituye la forma en que pueden programarse retardos en código máquina y proporciona a la vez una oportunidad excelente para demostrar lo rápido que puede llegar a ser este código máquina.

El tipo de bucle que se utiliza más en BASIC es el correspondiente a la instrucción FOR...NEXT. Esta instrucción utiliza una variable como «contador» para mantener un registro de las veces que se ha ejecutado el bucle comparando a la vez esta variable con el valor límite que se ha establecido cada vez que termina el mencionado bucle. Sin embargo, el efecto del bucle FOR... NEXT puede simularse en BASIC sin utilizar las palabras clave FOR o NEXT. El método se muestra en la figura 6.12.

```
10 C=0:ND=10
20 PRINT"ACCION!  ";C
30 C=C+1
40 IF C<=ND THEN 20
50 PRINT"FIN"
```

Fig. 6.12 Sencillo bucle en BASIC que proporciona la acción ligada al FOR ... NEXT.

La variable contador es C, y el límite que se establece para ella es ND. Al final del programa el valor de C será 11, igual que en el caso de emplearse el bucle FOR N=1 TO 1Ø. Así, la próxima cosa que hay que hacer es echar un vistazo al diagrama de flujo de este tipo de programas, lo que podemos hacer si miramos la figura 6.13.

Este es el método que se utiliza en código máquina para establecer un bucle de cuenta. Podemos escribir unas líneas de lenguaje ensamblador que hagan lo mismo, pero, tal como es habitual, tenemos que reflexionar un poco más en cómo llevar a cabo dicha traducción. Hay que considerar que no tenemos variables en código máquina. Tenemos que decidir, pues, dónde almacenar un determinado valor y qué registro utilizar para decrementarlo. El bloque de decisión es más fácil: podemos utilizar, en este caso, una prueba de tipo BNE para mantener el programa dentro del bucle, hasta que el contenido del registro sobre el que se lleva a cabo la comprobación sea cero. En el caso de que se esté preguntando cuál es el registro que se com-

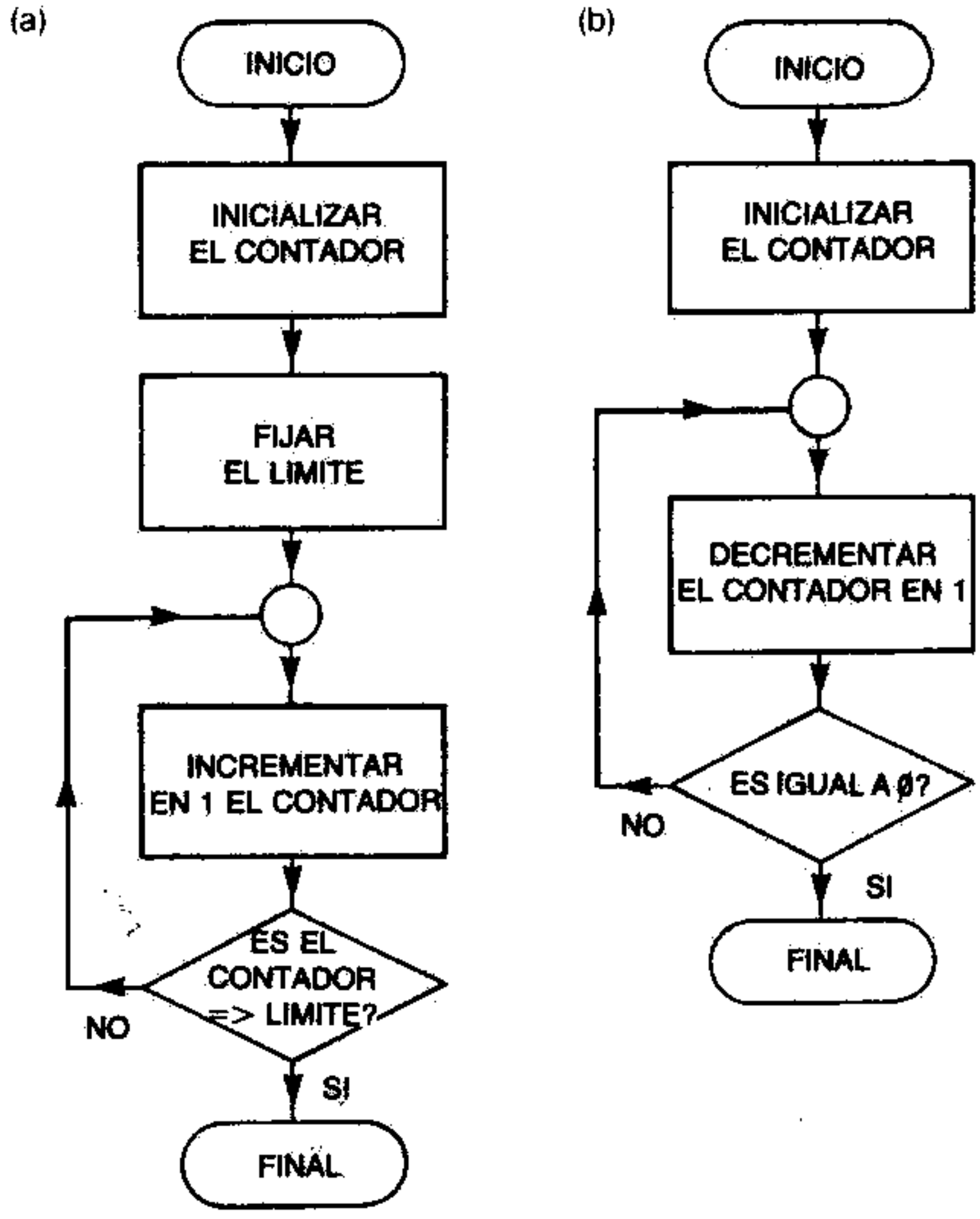


Fig. 6.13 Diagramas de flujo de bucles. (a) Por incremento. (b) Por decremento, que es más sencillo.

(a)

```
                LDX #$FF    A2 FF
BUCLE:          DEX         CA
                BNE BUCLE   DØ FD
                RTS
```

(b)

```
10 POKE56,159:A=40704
20 FORN=1TO6:READ D%
30 POKE A+N,D%:NEXT
40 PRINT"INICIO"
50 SYS40705
60 PRINT"FINAL"
100 DATA162,255,202,208,253,96
```

Fig. 6.14 (a) Bucle de cuenta en lenguaje ensamblador. (b) Programa BASIC de carga.

prueba si está o no a cero, la respuesta es: el que se utilizó justo antes de la prueba BNE (o cualquier otra prueba de este tipo).

La figura 6.14(a) muestra el aspecto que presenta el programa en lenguaje ensamblador. El registro que utilizamos es el registro X en lugar del acumulador. Esto es debido a que el registro X, junto con el Y, es el que mejor se adapta para llevar a cabo las operaciones de cuenta. La razón de esto estriba en la existencia de las instrucciones para incrementar y decrementar dichos registros. INX y DEX son las instrucciones que llevan a cabo, de forma respectiva, estas operaciones en el registro X. Las instrucciones INY y DEY realizan la misma función en el registro Y. No hay ninguna instrucción que haga lo mismo con el acumulador, debiendo recurrir a un procedimiento distinto si queremos llevar a cabo una operación de cuenta empleando el registro A. En este ejemplo se carga en un registro de ocho bits. Una vez inicializado el registro X, lo decrementamos y marcamos el lugar correspondiente a esta instrucción con el nombre de «BUCLE». Esta será la instrucción a la que querremos volver si el contenido del registro no es cero. La prueba se lleva a cabo mediante la instrucción BNE (saltar si no es igual a cero), ya que queremos que se repita la acción de decrementar hasta que el contenido del registro X llegue a cero. El programa en BASIC que carga los bytes en memoria y ejecuta a continuación el programa es el de la figura 6.14(b). Cuando lo ejecutemos, no seremos capaces de apreciar el intervalo de tiempo que transcurre entre que aparecen las palabras «INICIO» y «FINAL» en pantalla. Esto no es debido a que no haya ocurrido nada, sino a lo extremadamente rápido que se ejecuta el programa de cuenta en código máquina. Si prueba a ejecutar un programa en BASIC del estilo de:

```
1Ø A= 255:? "INICIO"
2Ø A= A−1
3Ø IF A<>Ø THEN 2Ø
4Ø ? "FINAL"
```

verá que se produce una pausa notable. Sin embargo, la diferencia no refleja las velocidades relativas de ambos lenguajes, ya que gran parte del tiempo invertido por ambos programas se emplea en la parte de escritura en BASIC. Para ver lo grande que es esta diferencia, por lo que a tiempo de ejecución se refiere, entre el código máquina y el BASIC, tenemos que utilizar valores más grandes. Hay varias formas de hacerlo, pero la que vamos a emplear conlleva la utilización de dos bucles. Probablemente ya ha usado bucles anidados en BASIC. La idea se basa en la existencia de un bucle interior y otro exterior. En cada pasada del bucle exterior, se lleva a cabo en su totalidad el bucle interior. Eso nos permite llevar a cabo retardos mucho mayores por el procedimiento de ejecutar una cuenta dentro de otra.

Supongamos que tenemos en BASIC las líneas:

```
1Ø X= 1ØØ:? "INICIO"
2Ø X= X-1
3Ø Y= 255
4Ø Y= Y-1
5Ø IF Y<>Ø THEN 4Ø
6Ø IF X<>Ø THEN 2Ø
7Ø ? "FINAL"
```

Con ellas podemos llevar a cabo un proceso de decremento de Y desde 255 hasta Ø cada vez que se decrementa el valor de X. Ejecute el programa anterior y mida el tiempo que invierte en ejecutarse. No es necesario un cronómetro. Cualquier reloj que tenga minutero le servirá.

(a)

```
                LDX #$FF        A2 FF
BUCLE 2:        LDY  #$64       AØ 64
BUCLE 1:        DEY             88
                BNE BUCLE 1     DØ FD
                DEX             CA
                BNE BUCLE 2     DØ F8
                RTS
```

(b)

```
10 POKE56,159:A=40704
20 FORN=1TO11:READ D%
30 POKE A+N,D%:NEXT
40 PRINT"INICIO"
50 SYS40705
60 PRINT"FINAL"
100 DATA162,255,160,100,136,208
105 DATA253,202,208,248,96
```

Fig. 6.15 (a) Programa en ensamblador para una cuenta con dos bucles. (b) Programa BASIC de carga.

Vamos a ver ahora cómo pueden manejarse los mismos valores con un programa de decremento en código máquina. La figura 6.15(a) muestra la versión del programa en ensamblador. Se carga el registro X con $FF y el registro Y con $64 (1ØØ decimal). A esta segunda instrucción se le ha asignado la etiqueta «BUCLE2». Luego viene la instrucción DEY que decrementa el registro Y y que lleva asociada la etiqueta «BUCLE1». La prueba BNE devuelve el programa a esta últi-

ma instrucción mientras no llegue el registro Y a cero. Cuando ello ocurre se decrementa el registro X y se comprueba su valor. Vea como el orden no es el mismo que en la versión BASIC del mismo programa. En una operación de decremento y prueba en código máquina, hay que efectuar la operación justo antes de la comprobación, ya que de otra forma podría darse la circunstancia de que el registro que se comprobara no fuera el correcto. Mientras el registro X no llegue a cero, el programa vuelve para atrás, esta vez a BUCLE2, para volver a cargar el registro Y y efectuar otra vez el bucle interior.

Cuando ejecute este programa, verá como es difícil apreciar el tiempo que transcurre entre su inicio y su final. Es un buen ejemplo de la ventaja, en cuanto a velocidad, del código máquina frente al BASIC. Si no está convencido de que se haya ejecutado la cuenta, modifique el número del bucle exterior y sustituya el $64 por un $FF (o sea, cambie el 1ØØ por un 255 en la línea 1ØØ del programa BASIC). Esto hace que el retardo sea un poco más apreciable.

INC y DEC en el acumulador

Señalábamos anteriormente que no existen instrucciones del tipo INC y DEC que actúen sobre el acumulador. Sin embargo, esto no significa que no podamos utilizar el acumulador para operaciones de cuenta, sólo porque no está tan bien equipado como los registros X e Y. Suponiendo que debamos llevar a cabo una cuenta en el acumulador, podemos utilizar la instrucción SBC # 1, instrucción que significa restar un 1 del contenido del registro A. Ello nos conduciría a un programa de cuenta del tipo del que se muestra en la figura 6.16(a). Empezamos con CLC, la instrucción que pone el indicador de acarreo a cero. Es necesario hacerlo, ya que si estuviese a 1, la primera instrucción SBC restaría 2 en lugar de 1. El resto de las instrucciones SBC serán normales. De todas formas, no habría afectado mucho al retardo efectuado el que se hubiese llevado a cabo un decremento de más. En algún tipo de operaciones de decremento, tal como en el caso de contar bytes, el restar un 2 en lugar de 1 podría representar un desastre. Por esta razón, es una buena costumbre la de poner a cero el bit de acarreo antes de llevar a cabo una resta de este tipo. La línea de decremento corresponde a SBC # 1, siendo muy similar el bucle resultante al bucle ligado al registro X que vimos anteriormente. El programa BASIC de carga se muestra en la figura 6.16(b).

Sin embargo, el 65Ø2 le permite decrementar también valores que no se encuentren almacenados en registros. La instrucción DEC puede llevarse a cabo sobre cualquier posición de memoria, de forma que pueden escribirse programas de cuenta que utilicen todos los by-

(a)

```
                CLC          18
                LDA #$FF     A9  FF
BUCLE:          SBC  #1      E9  Ø1
                BNE FC       DØ  FC
                RTS          6Ø
```

(b)

```
10 POKE56,159:A=40704
20 FORN=1TO8:READ D%
30 POKE A+N,D%
40 PRINT"INICIO"
50 SYS40705
60 PRINT"FINAL"
100 DATA24,169,255,233,1,208,252,96
```

Fig. 6.16 Cómo llevar a cabo una cuenta atrás utilizando un byte en el acumulador. (a) Versión en ensamblador. (b) Programa BASIC de carga.

(a)

```
                LDA   #$FF
                STA   $6A
BUCLE 1:        STA   $6B
BUCLE 2:        STA   $6C
BUCLE 3:        DEC   $6C
                BNE   BUCLE 3
                DEC   $6B
                BNE   BUCLE 2
                DEC   $6A
                BNE   BUCLE 1
                RTS
```

(b)

```
10 POKE56,159:A=40704
20 FORN=1TO21:READ D%
30 POKE A+N,D%:NEXT
40 PRINT"INICIO"
50 SYS40705
60 PRINT"FINAL"
100 DATA169,255,133,150,133,151,133,152
110 DATA198,152,208,252,198,151,208,246
120 DATA198,150,208,240,96
```

Fig. 6.17 Una cuenta mucho más larga utilizando direcciones de la página cero para almacenamiento de valores. (a) Versión en lenguaje ensamblador. (b) Programa BASIC de carga.

tes que quiera. Las direcciones más adecuadas para guardar bytes de cuenta son las correspondientes a la página cero (de la Ø a la 255), pero cuando se utiliza el 65Ø2 dentro del Commodore 64, hay que ir con cuidado. Eso es debido a que el Commodore 64 utiliza muchas de estas posiciones de memoria para sus propios fines, con lo que cargar valores en algunas de ellas puede provocar un colapso en la máquina. Una rápida ojeada a la página cero de memoria pone de relieve que las posiciones 15Ø a 177 son relativamente poco usadas en el transcurso de programas no demasiado complejos, de forma que quizá podríamos utilizarlas para llevar a cabo una cuenta realmente larga. La figura 6.17(a) muestra la versión en ensamblador del programa. Las tres direcciones utilizadas son las $96, $97 y $98 (15Ø, 151 y 152, respectivamente, en base diez). El programa empieza cargando $FF en el acumulador y guardando este valor en las tres mencionadas posiciones de la página Ø de memoria. Como pertenecen a esta página de memoria, podemos utilizar el direccionamiento de página cero. No vuelve a utilizarse el acumulador, con lo que el valor $FF permanecerá en él hasta el final del programa. Esto nos permite ir almacenando su contenido en cada una de las tres direcciones de memoria sin necesidad de tener que reinicializarlo cada vez.

La técnica utilizada para la cuenta atrás es muy parecida a la que vimos anteriormente, con la diferencia de que ahora tenemos tres bucles. Pruebe a dibujar el diagrama de flujo correspondiente para ver como funciona el conjunto. El resultado final es que se va contando hacia abajo a partir del valor 16.777.215 (en base diez) hasta cero. Como puede imaginar, en este proceso se emplea mucho más tiempo que en el que involucraba sólo a dos registros. En este caso tendremos que esperar varios minutos. La versión en BASIC del programa se encuentra en la figura 6.17(b). Si utiliza un cronómetro para medir el tiempo transcurrido entre la aparición de las palabras INICIO y FINAL, y lo divide por el número citado anteriormente, obtendrá un valor aproximado del tiempo que se emplea en ejecutar una sola vez todo el bucle. No intente trasladar este método de cuenta atrás a un programa BASIC. El tiempo empleado en ejecutarlo puede superar todo lo que usted puede imaginarse.

7. Entradas, salidas y otros temas

Bucles de video

De todos los programas de bucles que podemos realizar en código máquina, los que involucran a las direcciones ligadas a la salida de video, están dentro del grupo de los que podemos considerar como más útiles. Ya hemos visto anteriormente cómo podíamos actuar sobre el contenido de la pantalla a través de instrucciones POKE de BASIC. Las técnicas utilizadas entonces pueden ser empleadas también desde el código máquina. La única diferencia estriba en que este último es mucho más rápido, pero sin embargo requiere mayor trabajo por su parte.

Para empezar, eche una ojeada a la figura 7.1. Muestra el diagrama de flujo de un programa encargado de llenar parte de la pantalla

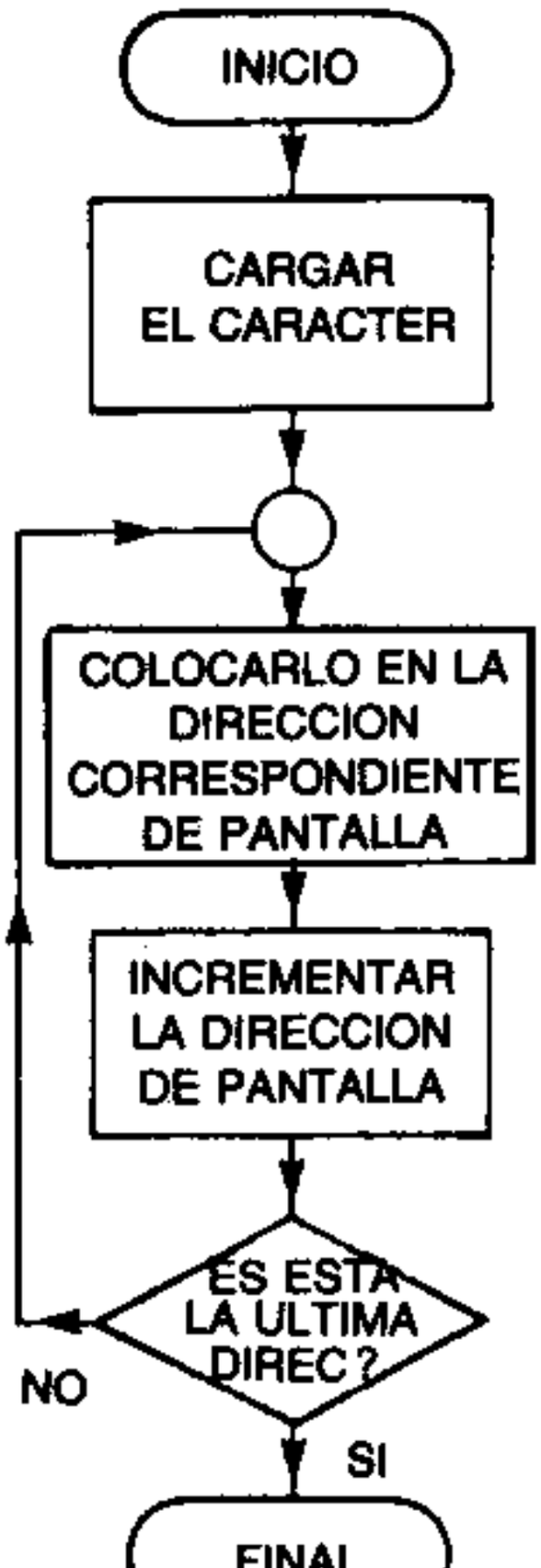


Fig. 7.1 Diagrama de flujo de un programa para llenar la pantalla con un determinado carácter.

con un determinado carácter. La idea consiste en cargar un registro (normalmente el acumulador es el más adecuado) con el código numérico asociado a un determinado carácter y guardar este valor en la primera posición de la memoria de pantalla, que es la $4ØØ. Debemos luego incrementar esta dirección, guardar de nuevo el contenido del acumulador y repetir el proceso hasta que lleguemos a la dirección final. El diagrama de flujo muestra lo que debemos hacer, aunque tenemos que ver cómo traducirlo para que pueda ser ejecutado por el 65Ø2. Este es el tipo de programa en el que puede utilizarse el direccionamiento indexado, de forma que vamos a empezar por ver lo que ello involucra.

Existen dos registros de índice (el X y el Y) cada uno de los cuales puede contener un número de un solo byte. Cuando llevamos a cabo una lectura o una carga indexada (o cualquier otra operación en la que se emplee un byte de memoria), se suma a la dirección que haya sido especificada el número contenido en el registro de índice. Se forma así una nueva dirección que es la que se emplea en el proceso de carga o lectura. Si por ejemplo tenemos un 2 guardado en el registro X, y especificamos una acción de almacenamiento a través de:

```
STA 1Ø24, X
```

(utilizando números decimales), significamos con ello que la dirección que se utilizará es la 1Ø24+2= 1Ø26 y que el byte contenido en el acumulador será almacenado en la posición 1Ø26 de memoria. Uno de los aspectos que hace que todo ello sea tan útil es el hecho de que dispongamos de las instrucciones INX y DEX para incrementar o decrementar, respectivamente, el contenido del registro de índice X. Existen unas instrucciones análogas (INY y DEY) para el registro Y.

Vamos a ver la versión del programa en lenguaje ensamblador, que se muestra, junto con el programa BASIC de carga, en la figura 7.2. El primer paso es directo: poner un 124 en el acumulador. Este es el código numérico ASCII correspondiente a un carácter gráfico, más concretamente el que consiste en medio cuadrado, aunque, evidentemente, podía haberse elegido otro cualquiera. Cargamos a continuación el registro X con un cero. El siguiente elemento es el bucle. Empieza almacenando el byte contenido en el acumulador en la posición de memoria 1Ø24 (en base diez) más el contenido del registro X, incrementando a continuación el registro X. En lenguaje ensamblador, esto se escribe como:

```
STA 1Ø24, X
INX
```

El siguiente paso consiste en comparar el contenido del registro X con cero, a través de BNE. Como el registro X empezó con el valor cero y se le acaba de incrementar, la comparación no pondrá a 1 el

(a)

```
          LDA #124        ;124 en decimal
          LDX #Ø          ;X empieza en cero
BUCLE:    STA 1Ø24,X      ;colocarlo en 1Ø24 + X
          INX             ;incrementar X
          BNE BUCLE       ;hasta el final de la cuenta
          RTS             ;volver al BASIC
```

(b)

```
10 POKE56,159:A=40704
20 FORN=1TO11:READ D%
30 POKE A+N,D%:NEXT
35 POKE53281,3
40 SYS40705
100 DATA169,124,162,0,157,0
110 DATA4,232,208,250,96
```

Fig. 7.2 (a) El programa en ensamblador. (b) El programa BASIC de carga. Sin embargo, esta versión llena sólo el primer cuarto de pantalla.

indicador de cero, con lo que BNE hará que el programa retroceda para ir a guardar el carácter en otra dirección. Cuando el contenido del registro X sea igual a 255 (decimal), la siguiente operación de incremento hará que el valor almacenado en dicho registro pase a ser cero otra vez. Eso es debido a que el registro puede contener un solo byte. En este punto, la prueba de la instrucción BNE da resultado negativo con lo que el programa sale del bucle y vuelve al BASIC.

Logramos así parte de lo que pretendíamos, aunque no todo. Colocaremos el mismo carácter en 256 posiciones consecutivas de la pantalla, pero ésta consiste en un millar de posiciones. Nos hemos visto limitados en este caso por el tamaño del registro X : un byte. Ello hace que una lectura o una carga indexadas, llevadas a cabo dentro de un bucle, no puedan involucrar a más de 256 bytes. Partiendo de la base de que poco (un cuarto de la pantalla), es mejor que nada, vamos a dejarlo. Más tarde veremos cómo podemos superar esta limitación.

Traducir a mano este código ensamblador a código máquina es relativamente directo. Una vez que se han seleccionado los bytes de código máquina (vea el byte de desplazamiento que sigue a la instrucción BNE), puede escribirse el programa BASIC que los cargue en memoria [figura 7.2(b)]. No debería tomarnos demasiado tiempo, ya que aparte del valor de N y la línea DATA, es casi idéntico a los que ya hemos visto hasta ahora. Sin embargo, debemos incluir la instrucción POKE 53281,3 para asegurarnos de que el resultado del programa

sea visible. Al final de cada ejecución, debemos emplear las teclas STOP y RESTORE para devolver la pantalla a las condiciones normales. Cuando ejecutemos este programa, veremos cómo se producirá un pequeño retardo mientras que el BASIC carga, con su habitual lentitud, los bytes en la memoria, aunque a continuación, el código máquina lleva a cabo su tarea a la gran velocidad que lo caracteriza.

El resto del camino

Escribir un programa que llene toda la pantalla no es fácil, debido a las limitaciones impuestas por el registro de índice. Hay varias formas de llevar a cabo la operación que pretendemos, pero la más directa es a través del empleo del llamado *direccionamiento indirecto.* Este método de direccionamiento se comentó brevemente en el capítulo 4, aunque vamos a volver ahora sobre él y estudiaremos con un poco más de detalle los dos métodos que puede utilizar el 65Ø2. Estos dos métodos son conocidos a menudo con los nombres de «indexado indirecto» e «indirecto indexado». Son nombres más bien confusos, de forma que a lo largo de este libro nos referiremos a ellos con los nombres más sencillos de X-indirecto e Y-indirecto. Se debe al hecho de que el primero de ellos utiliza el registro de índice X, y el registro de índice Y el otro. El que necesitamos para el programa que queremos hacer es el método Y-indirecto.

La forma en que opera el direccionamiento Y-indirecto es la siguiente. La primera dirección que se quiere utilizar (tal como $Ø4ØØ en nuestro caso) se almacena en forma de dos bytes en dos posiciones de memoria consecutivas. Como es habitual, los bytes deben almacenarse en el orden de primero el menos significativo y luego el más significativo. Deben estar, además, en la página cero de memoria. También se coloca un número en el registro de índice Y. Al llevar a cabo una operación de memoria tal como una lectura o un almacenamiento, se utilizará la forma Y-indirecta. En lenguaje ensamblador, una operación de almacenamiento de este tipo podría escribirse en la forma:

STA dirección,Y

Su efecto es, a primera vista, bastante complicado. Se lee de la memoria el byte bajo de la dirección y se le suma el contenido del registro Y. Si se produce acarreo en esta operación, se suma 1 al byte alto. Los dos bytes de la nueva dirección se utilizan para la operación de almacenamiento (en este caso). Supongamos, por ejemplo, que almacenamos la dirección del primer byte de la memoria de pantalla ($Ø4ØØ) en la página cero de memoria, en las posiciones 15Ø y 151 (en base diez). El orden de almacenamiento empleado obliga a guar-

dar $ØØ en la posición 15Ø y $Ø4 en la posición 151. Suponiendo que tuviéramos almacenado en el registro Y el número $26 (hexa) el resultado de:

```
STA (15Ø),Y
```

sería el de sumar $26 al número almacenado en 15Ø (que era cero), constituyéndose así la dirección $Ø426. Esta es la dirección donde se cargará el byte del acumulador.

La figura 7.3(a) muestra la versión completa del programa en ensamblador. Las seis primeras líneas colocan los valores adecuados en los registros y en la memoria. Ello no se ha realizado de la forma más eficiente posible, sino de una forma que sea fácil de seguir. Donde las cosas empiezan a complicarse es en la línea 7, al principio del bucle.

(a)

```
        LDA   #124
        LDX   #Ø
        STX   15Ø
        LDX   #4
        STX   151
        LDY   #Ø
BUCLE:  STA   (15Ø),Y
        INY
        BNE   BUCLE
        INC   151
        LDX   151
        CPX   #8
        BNE   BUCLE
        RTS
```

(b)

```
10 POKE56,159:A=40704
20 FORN=1TO26:READ D%
30 POKE A+N,D%:NEXT
35 POKE53281,3
40 SYS40705
50 GOTO 50
100 DATA169,124,162,0,134,150,162,4,134
110 DATA151,160,0,145,150,200,208,251
120 DATA230,151,166,151,224,8,208,243,96
```

Fig. 7.3 Llenado de la pantalla entera. También existe un problema en este programa pero que no tiene efecto alguno en este caso. (a) Versión en lenguaje ensamblador. (b) Versión de carga en BASIC.

Lo que origina el problema es la necesidad de componer un millar de veces la dirección asociada a la memoria de pantalla. Vamos a estudiar con detalle esta parte del programa.

Al principio del bucle, el valor del registro Y es Ø y la «dirección de base» para la instrucción STA se encuentra almacenada en las direcciones 15Ø y 151 (decimal) de la página cero. La posición 15Ø contiene un Ø y la posición 151 un $Ø4, de forma que ambas componen la dirección de la primera posición de la memoria de pantalla $Ø4ØØ (1Ø24 en decimal). Cuando se utiliza la instrucción STA (15Ø),Y por primera vez se almacenará el byte del acumulador en la posición $Ø4ØØ. La instrucción INY incrementa, entonces, el valor del registro Y que pasa de Ø a 1. Cuando se llega a la instrucción BNE BUCLE, el hecho de que el registro Y contenga un 1 (y sea, pues, diferente de cero), provoca que el programa vuelva hacia atrás. Este bucle hará que se vayan utilizando direcciones consecutivas de la memoria de pantalla. Una vez más, el bucle seguirá así hasta que el registro Y, al ser incrementado, pase de 255 (decimal) a cero.

Con ello el programa sale del bucle. Como inicializamos la dirección 15Ø con un cero, no se producirá suma automática alguna sobre el contenido de la posición 151, de forma que tendremos que ocuparnos nosotros de ello. Para esto he utilizado un atajo. Debe comprender en qué consiste y cómo utilizarlo. Al final del primer bucle, incrementamos el número almacenado en la posición 151. Con ello, si repetimos el primer bucle, pasamos automáticamente al siguiente cuarto de pantalla. Sin embargo, hemos de poder detener el programa si no queremos cargar el byte 124 en todas y cada una de las posiciones de la memoria, lo que pronto daría al traste con nuestro programa y con un montón de cosas más. Para hacerlo, se copia el contenido de la posición 151 en el registro X y se compara con el número 8. Si no es cierta la igualdad, la segunda instrucción BNE devuelve el programa al primer bucle para ir a llenar otra zona de memoria. Cuando el contenido de la posición 151 alcanza el valor 8, el programa se detiene. La figura 7.3(b) muestra el programa BASIC que carga los bytes correspondientes en memoria y ejecuta el código máquina.

El programa llena correctamente toda la pantalla, pero no siempre puede utilizarse. La razón estriba en que las posiciones de memoria ligadas a la pantalla van de la $Ø4ØØ (1Ø24 en base diez) a la $Ø7E7 (2Ø23 en base diez). En lugar de ello, hemos utilizado las posiciones que van de la $Ø4ØØ a la $Ø7FF. La zona que hemos llenado de más, comprendida entre $Ø7E7 y $Ø7FF es una zona de memoria reservada a los «punteros de patrones». Si su programa no utiliza patrones, todo va bien. Si se van a utilizar patrones y se necesita, pues, esta zona de memoria más tarde, podemos recurrir a cargarla posteriormente. Pero si se quiere guardar información referente a patrones en esta zona de la RAM y ejecutar a continuación el programa

de código máquina, habrá que detener el llenado de la pantalla justo después de la posición $Ø7E8. Para ello es necesario algo más que la simple instrucción CPX 8 que utilizamos en el ejemplo. Habría que utilizar CPX 7 en lugar de CPX 8 y después del bucle BNE, habría que colocar LDX 15Ø seguido de CPX $E8 para comprobar el valor del byte bajo. Les seguiría otra instrucción BNE BUCLE. El resultado de todo ello consistiría en detener el programa cuando se hubiera alcanzado la dirección $Ø7E8, salvando así a sus patrones de algo peor que la muerte.

Y seguimos experimentando

Podemos modificar el programa de la figura 7.3 de una forma interesante. Supongamos que hubiéramos empezado con el acumula-

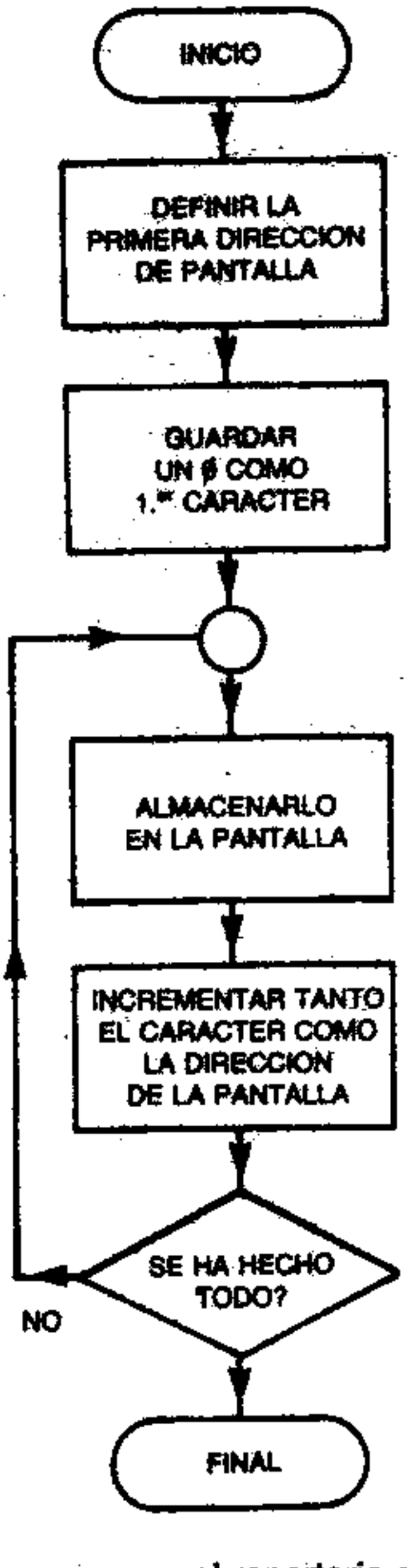


Fig. 7.4 Diagrama de flujo para sacar el repertorio completo de caracteres.

(a)

```
          CLC
          LDA   #Ø
          TAX
          STX   15Ø
          LDX   #4
          STX   151
          TAY
BUCLE:    STA   (15Ø),Y
          ADC   #1
          INY
          BNE   BUCLE
          INC   151
          LDX   151
          CPX   #8
          BNE   BUCLE
          RTS
```

(b)

```
10 POKE56,159:A=40704
20 FORN=1TO27:READ D%
30 POKE A+N,D%:NEXT
35 POKE53281,3
40 SYS40705
50 GOTO 50
100 DATA24,169,0,170,134,150,162,4,134
110 DATA151,168,145,150,105,1,200,208
120 DATA249,230,151,166,151,224,8,208
130 DATA241,96
```

Fig. 7.5 (a) Versión en lenguaje ensamblador. (b) Listado BASIC correspondiente al diagrama de flujo de la figura 7.4.

dor igual a cero y que incrementáramos su valor en cada pasada del bucle. Ello equivaldría a sacar por pantalla todos los caracteres asociados a los números del Ø al 255 (decimal). ¿Qué pasaría entonces? Bien, como el acumulador puede almacenar sólo 8 bits y 255 es el mayor número que puede codificarse con estos ocho bits, una vez llegados a 255, al incrementar el acumulador, éste volvería a cero. La figura 7.4 muestra el diagrama de flujo de todo este proceso. La figura 7.5, por su parte, muestra su traducción a ensamblador así como el programa BASIC de carga. No hay nada en todo ello que pueda provocarle quebradero de cabeza alguno, ya que la única diferencia entre este programa y el de la figura 7.3 estriba en el hecho de incrementar el acumulador. Como antes, tenemos que poner a cero el bit de aca-

rreo al principio del programa. La instrucción ADC#1 del bucle, irá sumando 1 al acumulador en cada iteración. Pruebe el programa, y verá aparecer en pantalla el conjunto completo de caracteres. Este es un buen ejemplo de cómo puede ampliarse un sencillo programa para que haga bastante más que la versión original. Este es un punto importante, ya que lo mismo ocurre con gran parte de la programación en código máquina. Si toma nota de todos los programas en ensamblador que haya utilizado, así como de lo que hacen, verá que esta «librería» es un elemento de capital importancia. Muy a menudo se encontrará que cualquier nuevo programa que quiera escribir puede elaborarse modificando o combinando (o ambas cosas a la vez) viejas rutinas con las que ya esté familiarizado. Otra gran ventaja de todo esto es que estas viejas rutinas son muy fiables, mientras que una nueva requiere cierto período de pruebas antes de que pueda confiar en ella.

Antes de que se me olvide: ¿ha pensado en lo que ocurre cuando hay un 255 en el acumulador y se le suma un 1? ¿Qué pasa con el bit de acarreo? ¿Tenemos que utilizar una instrucción CLC en algún sitio del bucle para contrarrestar su efecto? Compruebe si puede prever el fenómeno que se produce, así como si puede planear la forma de evitarlo.

Almacene sus programas

Una vez llegados a este punto, cuando los programas son cada vez más largos y hacen cosas cada vez más interesantes, es hora ya de estudiar el tema del amacenamiento de los programas de código máquina en cinta magnética. Sin embargo, no está obligado a ello. Nuestros programas en código máquina han adoptado hasta el momento la forma de un programa BASIC encargado de cargar números en memoria. Evidentemente usted puede almacenar este programa BASIC, que es el que creará el código máquina siempre que usted quiera. Cuando se está utilizando una mezcla de BASIC y código máquina, ésta es la forma ideal de guardar y recuperar los bytes de código máquina. Sin embargo, hay veces en que se necesita de toda la memoria que sea posible y un programa BASIC es tan bienvenido como un elefante a una cápsula espacial. También es posible que se quiera guardar en cinta un programa que sea una mezcla de BASIC y código máquina para dificultar la copia del mismo por parte de terceros. De una u otra forma, puede querer que se graben de forma directa los bytes almacenados en memoria. Las instrucciones normales del BASIC pueden ocuparse de ello aunque tal como vamos a ver, de una forma bastante tortuosa.

Al revés que la gran mayoría de las demás máquinas, el Commo-

dore 64 no tiene instrucción alguna en BASIC para guardar o recuperar un programa de código máquina. Cuando utilice un programa monitor de código máquina escrito para el Commodore 64, verá que ya incluye rutinas para el almacenamiento y la recuperación de programas en código máquina. Esto es estupendo si se está utilizando de forma continua el monitor, pero sería más apropiada otra forma de guardar y recuperar programas.

Afortunadamente, las instrucciones BASIC ordinarias de SAVE y LOAD pueden prestarnos un gran servicio. Cuando se utiliza la instrucción SAVE, el Commodore 64 guarda un programa BASIC. Esto significa que se guardarán todos los bytes desde la dirección 2Ø49 hasta el último byte de BASIC. Los dos primeros bytes que se almacenan son, de hecho, los dos que señalan el final del programa BASIC. ¿De dónde saca la máquina estas direcciones? De su página cero de memoria. El principio del programa BASIC tal como usted ya sabe, está almacenado en las posiciones 43 y 44 de memoria. El final del BASIC, que es lo mismo que el principio de la tabla de la lista de variables, está almacenado en las direcciones 45 y 46. Si modificamos los valores de estas posiciones de memoria, podemos controlar el almacenamiento o la recuperación de cualquier parte de la memoria.

```
10 POKE56,159:A=40704
20 FOR N=1TO100
30 POKEA+N,N:NEXT
40 POKE43,255:POKE44,158
50 POKE45,101:POKE46,159
60 SAVE"MC"
70 POKE43,1:POKE44,8
80 POKE45,3:POKE46,8
```

Fig. 7.6 Programa que carga una serie de números en memoria y los guarda a continuación en cinta. Esto ilustra como se pueden almacenar y recuperar programas en código máquina.

Vamos a probarlo. La figura 7.6 muestra un programa BASIC que carga unos números en memoria y a continuación los graba sobre cinta magnética. Desde luego, podíamos haber utilizado código máquina auténtico, pero así la secuencia de números es más fácil de reconocer. Como es habitual, la zona de memoria empleada ha sido protegida de forma que los cien números no corren peligro de ser alterados por acción de la máquina. Las líneas 4Ø y 5Ø llevan a cabo las secuencias POKE necesarias para inicializar los valores de la página cero. En lugar de la dirección habitual de inicio del BASIC que se coloca en las posiciones 43 y 44 de memoria, colocaremos una direc-

ción dos bytes por debajo del inicio de nuestro conjunto de números. Esta diferencia de dos bytes es importante ya que si utilizásemos la verdadera dirección inicial, el sistema operativo sustituiría los dos primeros números por los dos bytes de una dirección: la dirección del final del programa. Colocamos a continuación la última dirección de nuestro programa en las posiciones 45 y 46 de memoria. Después de esto, podemos utilizar una instrucción SAVE normal y corriente. Ello hará que aparezca en pantalla el mensaje habitual de cinta. En las líneas 7Ø y 8Ø se devuelven los valores habituales a las posiciones de la página cero.

Ahora hemos de comprobar que todo ello funcione. Ejecute (RUN) el programa con una cinta virgen en el transporte. Lleve a cabo las habituales acciones ligadas a PRESS RECORD AND PLAY cuando la máquina se lo requiera, y deje que el programa finalice. Rebobine la cinta que debería contener ahora ya los números que se cargaron en memoria. Desconecte el Commodore 64. Cuando lo conecte otra vez, se habrán borrado todas las direcciones y los números habrán desaparecido de la memoria. Para reproducir el contenido de la cinta, habrá que volver a inicializar los parámetros de la página cero. Ello no puede llevarse a cabo en un programa ya que la operación LOAD no funciona si existen líneas de programa después de la instrucción correspondiente.

Para recuperar el programa, así pues, hay que empezar por hacer la reserva de memoria a través de POKE 56,159 (RETURN). A continuación, hay que cambiar los valores almacenados en las posiciones 43 y 44 de memoria a través de:

```
POKE 43,255 : POKE 44,158
```

tal como hicimos anteriormente. Entre ahora LOAD "MC" para poner los bytes en su lugar. Después de completarse la carga, teclee:

```
POKE 43,1 : POKE 44,8 : POKE 45,3 : POKE 46,8
```

Con esto restauraremos los valores habituales de los parámetros de la página cero que hemos utilizado. Puede usted comprobar entonces, los valores que se han leído de la cinta. Para ello pulse:

```
FOR N=1 TO 1ØØ : ?PEEK (4Ø7Ø4+N); " "; : NEXT
```

seguido de RETURN. Con ello, debería ver como aparecen los números en la pantalla. Queda así demostrado que un programa de código máquina puede guardarse y ser recuperado utilizando tan sólo las instrucciones BASIC habituales.

No crea usted, dicho sea de paso, que todo lo anterior es aplicable sólo a programas de código máquina. Cualquier zona de memoria puede guardarse en cinta y ser leída posteriormente a través de este procedimiento. Como el contenido de la pantalla está controlado por

los bytes de memoria que van de la posición 1Ø24 a la 2Ø23 (en base diez) pueden emplearse las instrucciones SAVE y LOAD de esta forma, para guardar y recuperar contenidos de pantalla. Estos contenidos de pantalla, pueden verse estropeados por los mensajes que aparecen durante la fase de lectura, de forma que para obtener mejores resultados debería usted desconectar la pantalla de la salida del teclado. Ello puede hacerse a través de la instrucción POKE 154,4. Puede reconectarse la pantalla a través de POKE 154,3. Mientras esté desconectada la pantalla no verá usted nada en ella cuando teclee POKE 154,3. El efecto será más evidente, para sacar por pantalla un programa.

Algo referente a los mensajes

Después de este breve intermedio referente al almacenamiento y lectura de código máquina, vamos a volver a los programas. Si se acuerda habíamos dejado el tema en la figura 7.5., con el programa para escribir caracteres por la pantalla. Ya es hora de que veamos la forma de que algo más interesante aparezca por la salida de video y las letras parecen ser un buen principio para este tipo de programación. ¿Qué es lo que hemos de hacer? Bien, para empezar queremos almacenar los códigos ASCII de algunas letras en alguna parte de la memoria, sin hacerlo a través de una cadena alfanumérica tal como lo haríamos en BASIC. Debemos saber la dirección donde se guardará la primera letra y cuantas letras se han almacenado a partir de ella. Después de esto, deberemos ser capaces de construir un bucle que vaya cogiendo bytes de la «zona de texto» (o sea donde hemos almacenado los códigos de las letras) y los vaya colocando en la memoria de pantalla. Ya hemos visto el tipo de instrucciones que podíamos utilizar para ello: la lectura o la escritura autoincrementada.

Empezamos como siempre con un diagrama de flujo. Esta vez la cosa no es tan fácil ya que necesitamos una forma diferente de acabar el bucle. Podríamos recurrir a contar las letras que queremos colocar en la pantalla, pero quiero utilizar esta vez una técnica distinta: la basada en un *elemento terminal*. Probablemente ya esté usted familiarizado con esta idea a través de los programas de BASIC. Un «elemento terminal» es un byte que el programa puede reconocer como un carácter especial: uno que no forme, por ejemplo, parte del mensaje. Un elemento terminal adecuado para muchas aplicaciones es el Ø, de forma que éste es el que vamos a utilizar. Las dificultades aparecen cuando no queremos que aparezca este elemento terminal en pantalla. Debido a la forma en que el Commodore 64 utiliza sus códigos numéricos, el número Ø, tal como están las cosas, daría lugar a un carácter impreso: el @. No queremos que esto suceda, de forma que

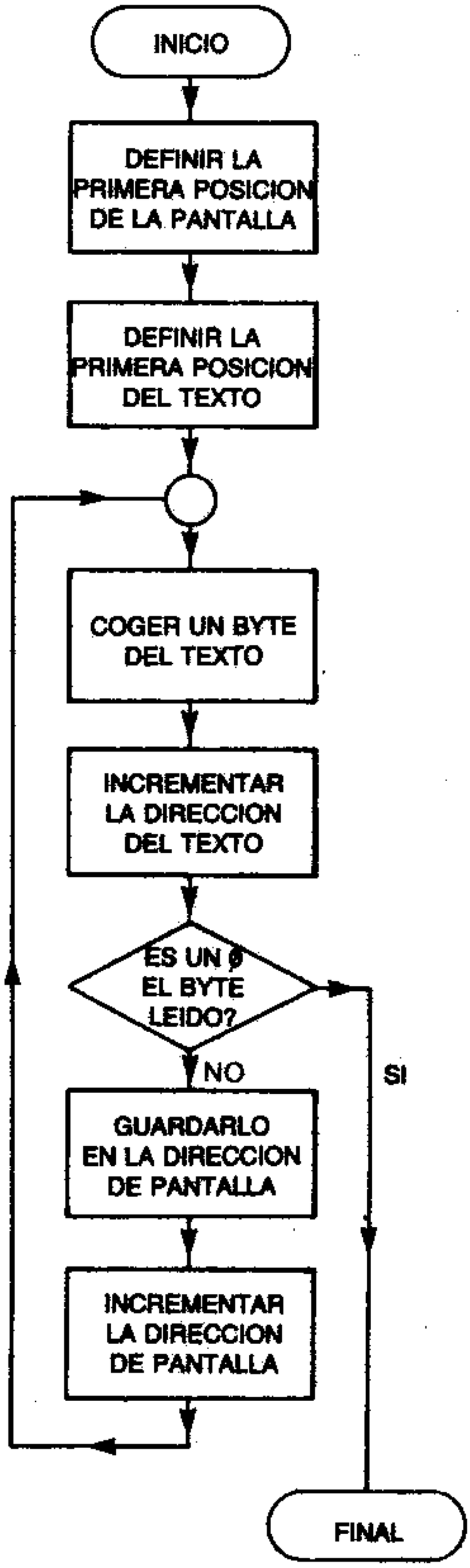


Fig. 7.7 Diagrama de flujo para sacar un mensaje por pantalla.

hemos de comprobar el contenido del acumulador después de leer el byte de memoria y antes de colocarlo en la memoria de pantalla. Ello, tal como puede usted imaginar, complicará un poco los bucles.

El diagrama de flujo que necesitamos lo podemos ver en la figura 7.7. Lo que hemos de hacer es almacenar dos direcciones. Una de ellas le será familiar: forma parte de la memoria de pantalla. Corres-

ponderá a la dirección del primer byte que queremos colocar en la pantalla. La otra direción, debe ser una dirección de almacenamiento: el inicio de la cadena de bytes que habremos utilizado para guardar los códigos ASCII y que no habrá sido utilizada para nada más. Podemos simplificar las cosas asignando a estas dos direcciones sendos nombres simbólicos. He elegido PANT para la memoria de pantalla y TXT para el lugar donde almacenamos los códigos. Lo que hacemos entonces, una vez asignadas estas direcciones es leer un código de TXT e incrementar acto seguido la dirección de TXT. Comprobamos luego el carácter, con el fin de ver si es nuestro elemento terminal Ø. Si es así, queremos que el programa finalice. Si no se trata del elemento terminal, almacenamos el byte de PANT, incrementamos la dirección de PANT y vamos a por otro carácter. Ello nos conduce a un diagrama de flujo con dos saltos. Uno de ellos es el correspondiente a la parte iterativa y el otro el que corresponde al final del programa. ¿Qué forma adoptará todo ello en lenguaje ensamblador?

La respuesta aparece en la figura 7.8. Se corresponde exactamente con el diagrama de flujo, siendo las partes que merecen un

(a)

```
            LDX #Ø
BUCLE:      LDA TXT,X   ;coger un carácter
            CMP #Ø      ;es un Ø?
            BEQ FUERA   ;si lo es, acabar
            STA PANT,X  ;sacarlo por pantalla
            INX         ;incrementar el índice
            BNE BUCLE   ;otra vez!
FUERA:      RTS         ;volver al BASIC
```

(b)

```
10 POKE56,159:A=40704:B=40863
20 FOR N=1 TO 16:READ D%
30 POKE A+N,D%:NEXT
40 REM CARGAR LOS CODIGOS DE LAS LETRAS
50 FORJ=1TO13:READ D%:POKE B+J,D%:NEXT
60 PRINT"J"
70 POKE53281,3
80 SYS40705
100 DATA162,0,189,160,159,201,0,240,6
110 DATA157,224,5,232,208,243,96,3,15
120 DATA13,13,15,4,15,18,5,32,54,52,0
```

Fig. 7.8 (a) Versión en ensamblador de la rutina de salida del mensaje. (b) Listado del programa BASIC de carga.

análisis más detallado las instrucciones BEQ y BNE. En la instrucción BEQ se ha cargado el acumulador con el byte procedente de la zona de memoria TXT, cuya dirección inicial es $9FAØ. Esta dirección se traduce en base diez en forma de los dos bytes 16Ø y 159, lo que corresponde a la dirección decimal 4Ø864. He elegido una dirección que esté bien lejos de la zona de memoria que contiene el programa. El acumulador se carga utilizando el direccionamiento X-indexado, habiéndose almacenado un cero en el registro X. Como resultado, el primer byte cargado en el acumulador se leerá de la dirección 4Ø864. Este será el byte 3, que corresponde al código numérico interno del Commodore 64 para la letra «C», (al utilizar este método de sacar texto por pantalla). La instrucción CMP Ø se coloca después de la lectura, de forma que se puede detectar el byte Ø al final del mensaje. BEQ significa «saltar si es igual a cero» (del inglés Branch if EQual to zero). El desplazamiento que sigue a este byte de instrucción llevará el programa a la instrucción RTS saltando por encima de las instrucciones comprendidas entre BEQ y RTS. Si, sin embargo el byte no es cero, se le almacena en la dirección PANT, utilizando otra vez el registro X para el indexado. Hay que ir entonces a por otro carácter. Como para ello es necesario un salto, y además un salto sin condiciones, podemos utilizar la instrucción JMP. Sin embargo, JMP debe ir seguida por una dirección completa de dos bytes, lo que es mucho más fácil que utilizar BNE. Una vez llegados a este punto del programa, el byte del registro X nunca podrá ser cero (a menos que se esté intentando sacar demasiadas letras por pantalla). A consecuencia de ello, BNE devolverá el programa a la posición BUCLE. Se continuará así hasta que se cargue un cero en el acumulador y la instrucción BEQ FUERA fuerce al programa a volver al BASIC.

Una vez explicado todo esto, podemos pasarlo todo al formato de un programa BASIC y probarlo. Colocaremos los bytes en memoria de una forma más sencilla : cargándolos a partir de una línea DATA. Ello se lleva a cabo en la línea 5Ø. Las líneas 2Ø y 3Ø habrán colocado previamente en su lugar el código máquina y la línea 8Ø será la que lo ejecute. Al hacerlo, se ve aparecer el mensaje en pantalla.

Quizá ya podemos abordar un punto un tanto enojoso al sacar texto por pantalla. En todos los programas precedentes que sacaban caracteres por pantalla, tuvimos que cargar el color de fondo adecuado para hacer visible el texto. En un programa iterativo de este tipo, podemos hacerlo mucho mejor. Lo que vamos a hacer será colocar el número 1 en la posición asociada a cada letra en la zona del color. Si recuerda el BASIC de su Commodore 64, recordará que sumando 54272 decimal (que es $D4ØØ) a la dirección de una posición de la memoria de pantalla, se obtiene la correspondiente dirección de color. Cargando un número entre 1 y 15 en esta dirección, podemos obtener distintos «colores» para los caracteres que queremos sacar. Este pro-

```
            LDX   #Ø
BUCLE:      LDA   TXT,X
            CMP   #Ø
            BEQ   FUERA
            STA   PANT,X
            LDA   #1
            STA   COLR,X
            INX
            BNE   BUCLE
FUERA:      RTS
```

Fig. 7.9 Modificación de la rutina de forma que se carguen los valores adecuados en la memoria asociada al color. Se asegura así que se pueda ver el texto sin tener que cambiar el color de fondo.

cedimiento requiere sólo unos reducidos cambios en el programa de código máquina. La figura 7.9 muestra la versión del mismo en lenguaje ensamblador. Después de la instrucción STA PANT, X que coloca el código del carácter en la memoria de texto, tenemos las dos instrucciones LDA#1 y STA COLR, X. Son las encargadas de cargar el número 1 en la posición de la memoria de color adecuada, suponiendo que partamos de la dirección de base correcta. Esta dirección de base es la $Ø5EØ para el texto, a partir de la cual, si se le suma el valor $D4ØØ, se obtiene la dirección $D9EØ. En base diez ello equivale a 55776. Utilizando este método para colocar caracteres en la pantalla, evitamos tener que cargar ningún valor en la posición 53281, así como el tener que pulsar las teclas STOP y RESTORE al final del programa. Este es otro pequeño paso hacia adelante.

Salidas por el port

Cuando describíamos en el capítulo 1 el funcionamiento de conjunto del sistema de ordenador, apareció el concepto de «port». Por lo que concierne al ordenador, un port es un circuito integrado (o colección de circuitos integrados) que efectúa la tarea de enviar o recibir bytes. Sucede, sin embargo, que la distribución de ports en el Commodore 64 es más bien complicada, estando organizados, además, de una forma bastante compleja. Afortunadamente, a menos que empecemos a utilizar gráficos realmente complejos, o bien que querramos efectuar grabaciones no estándar sobre cinta, no necesitaremos trabajar con estos ports. El sistema de sonido, así como los efectos especiales de video (tal como, por ejemplo, los patrones) utilizan unos circuitos integrados especiales.

Tal como es habitual, el llevar a cabo este tipo de acciones en código máquina es particularmente sencillo ya que no hace falta para ello comprender el sistema de ports. El Commodore 64 utiliza instrucciones POKE de BASIC para sus programas de sonido. Ello significa que podemos emplear métodos idénticos en código máquina, con la diferencia de tener que utilizar la instrucción LDA(inmediato) para obtener un byte y la instrucción STA para cargarlo en memoria. Todo lo que puede hacerse en BASIC utilizando POKE puede realizarse en código máquina a través de esta combinación LDA-STA. Así pues, no tenemos por qué preocuparnos de ello ya que podemos hacer en lenguaje máquina todo lo que es posible hacer en BASIC. De todas formas, no tiene sentido utilizar el código máquina para efectos de sonido ya que la velocidad de la máquina no representa en este caso ventaja alguna debido al hecho de tener que utilizarse bucles de retardo para obtener una duración adecuada del sonido. Es sólo cuando se quiere trabajar con efectos de video, imposibles de realizar con los patrones gráficos habituales, cuando hay que trabajar directamente con el circuito integrado de video del Commodore 64. Ello no constituye en modo alguno tarea para principiantes en el campo del código máquina, así que nos olvidaremos, por el momento, de ello.

8. Depuración y prueba de programas con MIKRO

Los encantos de la depuración

Ahora que ya ha experimentado algunas de las delicias de la programación en código máquina, parece justo mencionar algunos de los inconvenientes de este sistema de programación. Uno de ellos es la depuración de programas. Un error es un fallo de un programa y la depuración consiste en el proceso de encontrar estos errores y eliminarlos. Parece un proceso sencillo, pero es mucho más complicado de lo que parece.

Supongo que es mucho más fácil decirlo que hacerlo, pero lo primero que hay que hacer es prevenir estos errores. Compruebe cuidadosamente su diagrama de flujo para asegurarse de que realmente hace lo que espera de él. Cuando haya repasado bien el diagrama de flujo, pase el programa ensamblador y asegúrese de que lleva a cabo las instrucciones del diagrama de flujo. Una vez que haya comprobado este punto, asegúrese de que los bytes que se cargan en memoria se corresponden con las instrucciones de lenguaje ensamblador. Un punto al que hay que prestar especial atención es el de que se utilice el código adecuado al método de direccionamiento que se esté empleando. Si comprueba de esta forma cada parte de un programa podrá eliminar muchos errores antes de que se pongan de manifiesto. No se desanime si aún así el programa no funciona. A pesar de lo sencillo que es el código máquina hay muchas posibilidades de que esconda errores de uno u otro tipo. Esto le ocurre a todo el mundo y sólo a través de la experiencia puede llegarse a un estadio en el que los errores sean escasos y fáciles de localizar.

Si utiliza un ensamblador, una de las posibles fuentes de errores desaparece por completo. La flaqueza humana fuerza que el proceso de convertir las instrucciones de ensamblador en bytes de código máquina sea muy susceptible de errores. Ello es debido a que implica la consulta de tablas y todo lo que signifique pasear la vista de un papel a otro presenta muchas posibilidades de provocar errores. Más tarde dentro de este capítulo describiremos brevemente la acción que lleva a cabo el ensamblador MIKRO. En el momento de escribir este libro, había varios ensambladores disponibles para el Commodore 64, aunque MIKRO ofrecía varias ventajas que no ofrecían los otros, como por

ejemplo el hecho de estar disponible en un cartucho junto con un programa monitor denominado TIM. Hablaremos también más tarde sobre monitores. Si el código máquina le ha cautivado y quiere abordar tareas más complejas que las que se puedan presentar en el reducido ámbito de este libro, es esencial que se provea de un buen ensamblador y de un buen monitor. Tendrá que hacerse a la idea de pagar bastante por programas de este tipo. Si, por el contrario, no aspira a salir de la categoría de aficionado, los métodos que ya hemos visto de carga en memoria a través de la instrucción POKE del BASIC son más que suficientes.

El utilizar estos métodos significa, sin embargo, la existencia de errores acechando en cada rincón del código. La principal causa de estos errores es la fatiga. Convertir un programa en ensamblador a su expresión en bytes hexa y escribir estos bytes en forma de una línea DATA para un programa de carga en BASIC es un trabajo aburrido y todos los trabajos aburridos conllevan errores. La selección incorrecta del método de direccionamiento o bien simplemente equivocarse de código son los dos errores más frecuentes. Una importante fuente de problemas la constituyen también las instrucciones de bifurcación. Pueden cometerse errores al restar las dos direcciones así como al convertir el resultado a hexa (especialmente si se trata de un valor negativo). Surgen también problemas cuando se modifica un programa y se añade código entre instrucciones de salto y las instrucciones de destino correspondientes. Es muy frecuente, al hacerlo, olvidarse de modificar el valor del byte de desplazamiento. Este es un problema que no se da cuando se utiliza un ensamblador. Un salto incorrecto casi siempre conduce a un colapso del ordenador. A menudo puede recuperarse el control a través de las teclas STOP y RESTORE, aunque no siempre ocurre así, con lo que a veces se perderá el programa (¡esperemos que para entonces lo tuviese grabado!). Otra forma de bifurcación incorrecta se produce cuando se hace lo opuesto a lo que se quiere, o sea cuando, por ejemplo, se utiliza BEQ en lugar de BNE o bien se comete cualquier otro error por el estilo. Un análisis detenido de lo que hará la instrucción de salto para diferentes valores de los bytes involucrados debería eliminar este tipo de errores.

Como hemos visto, muchos problemas pueden eliminarse mediante un análisis riguroso, debiéndose prestar especial atención a las direcciones involucradas en las bifurcaciones así como al contenido inicial de los registros. Un error muy frecuente consiste en usar los registros en la suposición de que están a cero al principio del programa. Nunca puede estar uno seguro de ello. Es más aconsejable, de hecho, suponer que cada registro contendrá el valor que más nos incordie de cara a la normal ejecución del programa. Una vez dicho todo esto y con todo el esfuerzo y la buena voluntad del mundo, ¿qué hacer si, a pesar de todo, el programa sigue sin funcionar?

No existe una respuesta sencillo a esta pregunta. Puede ser que el diagrama de flujo no haga lo que supone (y si no dibujó un diagrama de flujo, ¡ya tiene lo que se merece!). Puede ser también que intente utilizar una subrutina de la ROM del Commodore 64 y que ésta no funcione en la forma en que imagina. Cuando tenga más experiencia en el seguimiento de programas más elaborados, podrá recurrir a un desensamblado de la ROM. En el momento de escribir estas líneas, existe en el mercado un libro muy útil titulado *Inside The Commodore 64*, cuyo autor es Milton Bathurst y que está publicado por DataCap, en Bélgica. Puede encontrarse en las librerías especializadas. Este libro constituye un listado completo (con comentarios) de la ROM del Commodore 64. Versa también sobre el empleo de la RAM, en particular de la página cero de memoria. Una vez más, todo esto es para aquellos que quieran dedicarse en serio a trabajar con el código máquina del Commodore 64. Por lo demás, todo lo que puedo hacer aquí es dar unas ideas generales sobre cómo eliminar los errores de un programa que parece estar bien construido, pero que no funciona en la forma que nosotros esperábamos.

La primera regla de oro es la de no probar nunca nada nuevo en medio de un programa extenso. Lo ideal sería que su programa de código máquina estuviese compuesto por subrutinas almacenadas en cinta, cada una de las cuales hubiera sido exhaustivamente comprobada antes de integrarla en un programa de mayor extensión. En la vida real ello no es fácil, especialmente si las subrutinas existen sólo como líneas DATA de un programa BASIC de carga. Como siempre, los que utilicen un ensamblador llevan las de ganar ya que pueden guardar las instrucciones de ensamblador igual que si se tratase de un programa BASIC, pudiendo combinarlas y editarlas.

Otra cosa que es recomendable hacer cuando se tiene una librería de subrutinas en cinta es documentarlas ampliamente. Además de las rutinas propias pueden guardarse junto con la documentación correspondiente subrutinas que se haya visto en revistas. *Personal Computer World* tiene una sección titulada SUBSET que ya me gustaría a mí que se reimprimiese en forma de libro. Cada mes publican varias rutinas de propósito general en código máquina. La mayoría son para los dos microprocesadores más ampliamente utilizados: el Z8Ø y el 65Ø2. Incluso si no utiliza estas rutinas, la forma en que están documentadas debería proporcionarle ideas sobre cómo documentar sus propias subrutinas. Sólo por ello ya vale la pena comprar esta revista. Si quiere utilizar una nueva rutina en un programa parece razonable asegurarse primero de qué es lo que hay que colocar en cada registro antes de llamar a la subrutina, así como conocer el estado en que quedan estos registros después de ejecutarse el subprograma en cuestión. Estudie los ejemplos que aparecen en SUBSET y vea como se presenta esta información.

Una planificación previa de este tipo debería eliminar muchos errores, pero si aún así se enfrenta a un programa que no funciona y que no quiere dejar de lado, tendrá que recurrir a los denominados *breakpoints*. Un breakpoint, por lo que concierne al sistema operativo del Commodore 64, es el byte $6Ø. Este es el byte que corresponde a la instrucción RTS, siendo su efecto el de volver al BASIC. Una vez en la BASIC, puede examinarse el contenido de la memoria a través de la instrucción PEEK. La idea consiste en elegir un lugar en el programa en el que se cargue algún valor en memoria. Si se coloca un $6Ø inmediatamente después, cuando se ejecute el programa, éste volverá al BASIC justo después de utilizarse esta posición de memoria. A través de PEEK podrá comprobarse entonces que el valor almacenado en memoria es el que esperaba. Si no es así, debería tratar de encontrar el lugar donde se produce el error. Si por el contrario una vez llegados a este punto todo es correcto, sustituya el byte original que había en lugar del $6Ø y coloque el $6Ø en la dirección siguiente después de otra instrucción de almacenamiento. Este tipo de proceso, sin embargo, se lleva a cabo mucho más fácilmente con la ayuda de un buen programa monitor. Hablaremos de ello más tarde.

El error más difícil de detectar por este o por cualquier otro procedimiento es el correspondiente a un bucle incorrecto. Un bucle incorrecto provoca siempre un colapso de la máquina. Aunque con las teclas STOP y RESTORE acostumbra resolverse la situación, no siempre ocurre así. Es posible, por ejemplo, que un programa que incurra en un error de este tipo modifique uno de los bytes que controlan el uso del teclado, de forma que no puedan utilizarse para nada ya las teclas del mismo. Puede inhibirse, por ejemplo, la acción de las teclas STOP y RESTORE escribiendo en la posición 8Ø8 (decimal) de memoria. La principal causa de este tipo de cosas es un salto a un lugar incorrecto. Así, por ejemplo, si tuviésemos un programa que contuviese las instrucciones :

```
        LDX, #$FF
BUCLE : DEX
        BNE BUCLE
```

podríamos encontrarnos con problemas. Supongamos que realizásemos el ensamblado a mano y que efectuásemos por error la bifurcación a la instrucción LDX en lugar de a la instrucción DEX. Ello provocaría que el registro X estuviese siempre a su máximo valor y por consiguiente nunca fuese decrementado hasta cero, con lo que el bucle no finalizaría nunca. Un error de este tipo es fácilmente localizable en lenguaje ensamblador ya que es fácil comprobar la situación del nombre de la etiqueta. Es mucho más difícil de detectar cuando se trabajo sólo con bytes. Como siempre, el poner todos los sentidos en la elaboración de estos bucles es la única solución que hay, siendo el

método descrito en este libro (el calcular y comprobar los bytes de desplazamiento) una buena medida de precaución.

El monitor TIM

Ya mencionamos los monitores al principio de este capítulo. A lo que con este nombre vamos a referirnos ahora no tiene nada que ver con un monitor de TV, que no es más que una pantalla de TV con una mejor calidad de imagen con relación a los receptores habituales, por lo que a la salida de información se refiere. En el ámbito del software, un monitor es un programa que supervisa la ejecución de otro programa de código máquina. Un monitor es (o deberia ser) un programa de código máquina que pueda cargarse en una zona de memoria que no sea susceptible de ser utilizada por nadie más. Una vez allí, permite visualizar (en hexa) el contenido de cualquier zona de memoria, alterar el valor de cualquier posición de RAM e inspeccionar o incluso modificar el contenido de los registros del 65Ø2. Estas son las acciones más elementales de un monitor aunque sería muy práctico, además, poder ejecutar un trozo de programa, colocar breakpoints e inspeccionar los registros durante la ejecución de un programa. El monitor ideal sería aquel que ejecutase las instrucciones de un programa de código máquina una a una, sacando cada vez por pantalla el contenido de los registros y de la memoria.

Este tipo de monitor estaba disponible para el viejo TRS-8Ø MK.1 y sería un elemento muy bienvenido por los programadores de código máquina de otros microprocesadores.

El monitor MIKRO

El cartucho con el MIKRO ensamblador/monitor contiene, entre otras utilidades, un monitor excelente muy cercano al TIM que estaba disponible en los ordenadores primitivos «PET». Cuando se conecta el cartucho, el Commodore 64 debe encontrarse apagado aunque una vez que esté el cartucho en su lugar puede dejarse ahí hasta que deba cambiarse por otro. El colocar el cartucho en su lugar no provoca que se ejecute el monitor. Habrá que llamarlo tecleando TIM (seguido de RETURN). Ello hace que aparezca en pantalla la información que se ilustra en la figura 8.1. En ella puede apreciarse la dirección de inicio de TIM y debajo de ella el contenido de los principales registros del 65Ø2, junto con otras dos importantes direcciones. Dejando de lado por el momento estas direcciones (a este nivel de estudio del código máquina no son relevantes), podemos pasar a estudiar la información referente a los registros. Aparece el contenido de los cinco registros

```
CALL @    814B
   ADDR   IRQ   SR   AC   XR   YR   SP
 .;814B   EA31  4D   ØØ   ØØ   ØØ   FB
```

Fig. 8.1 Mensaje del monitor TIM en pantalla. Aparecerá cada vez que se entre en el monitor pulsando TIM seguido de RETURN.

principales: registro de estado (que aparece como SR), acumulador, registro de índice X, registro de índice Y y puntero de la pila. Si acaba de poner en marcha su Commodore 64, el acumulador y los registros de índice estarán a cero. El registro de estado normalmente presentará el valor 4D (varios indicadores, incluido el bit de acarreo estarán a 1), siendo poco probable que el registro SP contenga su valor inicial ($FF).

Todo esto puede que no nos sea de mucha utilidad por ahora, pero a medida que vayamos progresando más y más en el trabajo en código máquina veremos que la visualización de los registros puede ser muy útil. Puede hacerse que vuelva a aparecer toda esta información en pantalla siempre que se quiera. Basta con pulsar la letra R (seguida de RETURN). Todas las acciones del monitor pueden ser llamadas a ejecución utilizando la combinación de un punto y una letra, tal como en .R. Cuando el monitor está a la espera de otra instrucción saca un punto por pantalla, de forma que basta con que pulse usted la letra de su elección, seguida de RETURN. La figura 8.2.

M — Visualiza el contenido de la memoria en hexa. Deben seguir a M una dirección inicial y otra final, cada una de ellas expresada como 4 dígitos hexa.

S — Guarda un programa de código máquina en cinta o disco.

L — Carga de cinta o disco un programa de código máquina.

G — Ejecuta un programa. Debe seguir a G la dirección inicial del programa, en forma de cuatro dígitos hexa.

H — Busca un grupo de bytes en memoria. Deben seguir a H una dirección inicial, una dirección final y los bytes que hay que buscar, todos ellos en hexa.

T — Transfiere un bloque de memoria. Deben seguir a T la dirección inicial del bloque, la dirección final y la nueva dirección inicial.

D — Desensamblar código. Deben seguir a D la dirección inicial y la final de la zona de memoria a desensamblar. La pantalla se llena con la información generada y no sale más información hasta que se pulse alguna tecla.

X — Volver al BASIC.

Fig. 8.2 Menú asociado a TIM dentro del paqueta MIKRO. Todo son funciones típicas de un monitor.

muestra las opciones que hay disponibles a partir de estas instrucciones de una sola letra.

No hace falta que emplee necesariamente todas las opciones. Algunas de ellas puede que nunca sean de interés para usted. En lugar de analizarlas todas, una detrás de otra, escoja las que son más útiles para el principiante del código máquina. Entre ellas, el comando «.M» (inspección de memoria) es el más importante. Cuando se teclea .M (o sólo M por la razón antes expuesta) hay que pulsar a continuación un espacio. Acto seguido hay que entrar la dirección inicial de la zona de memoria en la que se esté interesado. Ello debe hacerse en forma de un número hexa de cuatro dígitos. Así por ejemplo, la dirección 2B debe entrarse como ØØ2B. Hay que entrar entonces otro espacio seguido de la dirección final de la zona de memoria que se quiere visualizar. Una vez más, esto debe hacerse en forma de un número hexa de cuatro dígitos. No ocurre nada hasta que se pulse RETURN, de forma que aún puede uno reconsiderar lo que está haciendo. Una vez que se haya pulsado este tecla, se verá aparecer en pantalla el área de memoria solicitada. La columna de la izquierda contiene un punto, un punto seguido de dos puntos y la dirección inicial de cada fila de números. Hay ocho números en cada una de estas líneas. Entonces aparecen los valores almacenados en las direcciones indicadas. Supongamos por ejemplo que estemos considerando la línea cuyo primer número (a la izquierda) sea ØØ4Ø. El primer byte de esta línea será el byte almacenado en la posición Ø4ØØ, el siguiente byte será el almacenado en la posición ØØ41 y así sucesivamente.

Cuando de esta forma aparece en pantalla el contenido de la memoria, puede modificarse el valor de los bytes correspondientes a cualquiera de las posiciones de memoria visualizadas. Ello se realiza de una forma totalmente similar a como se edita un programa en BASIC. Se coloca el cursor encima del primer dígito del byte a través de las teclas correspondientes. Se entra entonces el dígito que se quiere poner allí, seguido del segundo dígito del byte correspondiente. Evidentemente, si sólo se quiere modificar un dígito, puede hacerse por el mismo procedimiento. Cuando se pulse RETURN, el valor modificado se cargará en memoria. Si entra una letra sin sentido, no se lleva a cabo ningún cambio. Una característica importante de esta posibilidad consiste en poder modificar el código máquina a fin de igualar cualquier byte del mismo a cero. Este valor corresponde a la instrucción BRK (Break), cuyo efecto consiste en devolver el control al monitor TIM. No es lo mismo que RTS, que normalmente devuelve el control al BASIC. Cuando se emplea la instrucción BRK para volver al monitor, puede utilizarse el comando .R para inspeccionar los registros del 65Ø2 o bien el .M para ver lo que ha sucedido en la memoria. Colocar el código $ØØ en un programa de código máquina es lo que se denomina «poner un breakpoint» y constituye una forma particular-

mente útil de ver lo que hace un programa en un punto determinado de su ejecución.

Así pues, el siguiente comando en importancia, por lo menos en lo que por ahora nos concierne, es el .R que mencionamos anteriormente. Pulsando la tecla R seguida de RETURN se provoca la aparición en pantalla del contenido de los registros del 65Ø2. A menos que se interrumpa a la máquina en medio de un programa, no habrá mucho que ver ahí, aunque cuando utilice las posibilidades más avanzadas del TIM, tal como el empleo de breakpoints, se demostrará muy útil este comando. Aparece pues en pantalla el contenido de todos los registros, siendo los que más a menudo se consultan, los registros A,B,X e Y. Esto significa pues, que cuando se ejecuta un programa en código máquina con un breakpoint en su interior, el programa se detendrá al llegar al mismo y podrá ver usted el contenido de los registros si utiliza el comando «.R». Muy a menudo esto es todo lo necesario para ver donde falla un programa. El monitor TIM permite establecer todos los breakpoints que se quiera, aunque sólo pueden colocarse los breakpoints en lugar de un código de instrucción. Si por ejemplo, se tiene un fragmento de código máquina con la instrucción LDA $4Ø5Ø, consistente como usted ya sabe en el código de instrucción $AD seguido de los bytes de la dirección $5Ø y $4Ø, y se quiere establecer un breakpoint en ella, sólo puede sustituirse el byte $AD por un $ØØ, no los bytes de dirección. Sólo un código de instrucción representa para el 65Ø2 una instrucción. Cuando el programa se detiene en un breakpoint, puede inspeccionarse el contenido de los registros, utilizar el comando .M para examinar la memoria y puede reemprenderse entonces la ejecución del programa con el comando .R. Incluso puede modificarse el contenido de los registros antes de proseguir la ejecución del programa, aunque no es recomendable que lo haga hasta que tenga más experiencia con el código máquina. Sin embargo, es útil si se está analizando la acción de un bucle y se quiere ver qué es lo que sucede cuando el contenido de un registro alcanza un valor determinado, tal como $ FF o $ ØØ. En lugar de efectuar el bucle docenas de veces, puede ejecutarse una vez para ver que funciona y modificar el contenido del registro con el fin de detener el bucle (y comprobar entonces qué es lo que ha hecho). Se puede alterar el contenido de los registros del 65Ø2 después de usar el comando .R para visualizar su contenido. Todo lo que hay que hacer es colocar el cursor encima del dígito que se quiera modificar y entrar el nuevo valor para el mismo seguido de RETURN. El valor se cargará en el registro cuando se reemprenda la ejecución del programa. .G debe ir seguido por un espacio en blanco y una dirección compuesta de cuatro dígitos que corresponda a la siguiente instrucción que se quiere ejecutar. Dondequiera que ponga breakpoints, debe anularlos después de utilizados empleando .M seguido de las operaciones de edición.

Todos estos son métodos muy potentes de depuración de un programa aunque constituyen sólo una pequeña parte de las posibilidades que ofrece este útil monitor. La ausencia de las instrucciones SAVE y LOAD para programas en código máquina queda solventada en el monitor TIM. El comando .S almacena en cinta un programa de código máquina, para lo cual hay que especificar un nombre de fichero, el tipo de almacenamiento utilizado (cinta o disco), la dirección inicial del código y la dirección final del mismo más 1. Todos los números deben expresarse en hexa, utilizándose cuatro dígitos para las direcciones y dos para los bytes. Supongamos, por ejemplo, que quiere guardarse en cinta un programa que empiece en la posición $9F6Ø y termine en la $9FE2. El comando SAVE debería entrarse, entonces, tal como sigue :

```
.S"MCPROG",Ø1,9F6Ø,9FE3
```

Dentro de esta línea podemos distinguir el .S que es el identificador del comando y MCPROG que es el nombre del fichero. El Ø1 indica que se está utilizando un cassette (un Ø8 indicaría el empleo de un disco). Recuerde que hay que emplear Ø1 o Ø8, no 1 ni 8. A continuación se encuentra la dirección inicial, así como la dirección final a la que se le ha añadido un 1. Las comas se usan para separar los números. Leer el programa del cassette es mucho más fácil. Basta con entrar:

```
.L"MCPROG"
```

o incluso tan sólo .L si lo que se quiere hacer es leer el primer programa de la cinta.

Hay que destacar también la posibilidad que ofrece el monitor TIM para buscar bytes en memoria, transferirlos y efectuar tareas de desensamblado. Supongamos que se quiere ver si el byte 2D se encuentra entre la dirección 9FØØ y la 9FFF. Si se entra .H 9FØØ 9FFF 2D seguido de RETURN, se verán aparecer en pantalla las direcciones del intervalo señalado que contengan un byte igual al indicado. Puede usarse también esta posibilidad para buscar grupos de bytes, lo cual es mucho más útil. Así, por ejemplo, si se busca en la ROM el lugar en el que esté una instrucción de lectura de la posición 7A (cuyo código será, pues, 85 7A) y piensa que esta instrucción puede encontrarse en la zona comprendida entre la dirección $AØØØ y la $AFFF, a través de .H AØØØ AFFF 85 7A obtendrá las diez direcciones de memoria en las cuales aparece esta combinación de bytes. Si se quiere trasladar un fragmento de código de la ROM al propio programa sin tener que recurrir al procedimiento de copiarlo byte a byte, puede utilizarse el comando .T. Deben acompañar a esta instrucción la dirección inicial del bloque de memoria que se quiere transferir así como la dirección final del mismo y la dirección de destino. Si se quiere, por

ejemplo, transferir el código que se encuentra entre $AB7B y $ABA4 a la zona de memoria que se encuentra a partir de $9FØØ, la directiva que hay que utilizar es:

```
.T AB7B ABA4 9FØØ
```

Vamos a comentar finalmente el comando .D cuya misión es la de permitir el desensamblado de memoria. Es particularmente útil para la ROM si no se dispone del desensamblado de la misma. Deben acompañar a la directiva .D la dirección inicial y la dirección final de la zona de memoria a desensamblar, ambas expresadas en la forma de cuatro dígitos hexa. Al pulsar RETURN aparecerá en pantalla el resultado de la operación de desensamblado en forma de 25 líneas de texto (a menos que se haya indicado una zona muy reducida de código). Cada línea contendrá un número de referencia, una dirección (correspondiente al código de instrucción), el código en hexa y su versión en lenguaje ensamblador. Algunas zonas de ROM sólo contienen bytes de datos (y no códigos de instrucción), de forma que aquellos bytes que no puedan ser interpretados como códigos de instrucción aparecerán acompañados de la palabra BYT. Si el producto del desensamblado ocupa más de 25 líneas, pulsando cualquier tecla se hará que aparezca en pantalla el siguiente bloque de 25 líneas y así sucesivamente hasta que se haya desensamblado la totalidad de la zona de memoria señalada. El empleo del desensamblador se demuestra particularmente útil si se quiere utilizar las rutinas de la ROM.

Cuando haya terminado de utilizar el monitor TIM, puede pulsar la tecla X para volver al BASIC. En conjunto, se trata de un paquete de software realmente satisfactorio, tanto más si consideramos que forma parte de un bloque que incorpora el ensamblador MIKRO, programa que vamos a comentar en el apartado siguiente.

Utilización del ensamblador MIKRO

En el momento de escribir este libro existían varios programas ensambladores para el Commodore 64, aunque, según mi opinión, el mejor de todos ellos era, con mucho, el ensamblador MIKRO. Aunque este es un libro de introducción pensado para lectores que nunca tendrán la necesidad de utilizar un ensamblador por lo sencillo de los programas de código máquina con los que deberán enfrentarse, se impone una descripción del MIKRO. El omitir la descripción de este ensamblador sería como escribir un libro sobre la historia de la aviación y omitir los nombres de Alcock y Brown (aunque muy pocos lectores estén al corriente de su vida).

Cualquier ensamblador que merezca el nombre de tal debe estar escrito en código máquina. MIKRO da un paso más en este sentido al

presentarse en forma de cartucho. Ello constituye una gran ventaja ya que el tener que cargar un ensamblador desde cinta puede representar un engorro. La razón estriba en que, de forma inevitable, cuando se está desarrollando un programa en código máquina con la ayuda de un ensamblador, en un estadio u otro del proceso de pruebas se dará la circunstancia de que el programa se pierda. Cuando sucede esto es muy frecuente que afecte al contenido de la memoria, pudiendo alterar al ensamblador en la misma forma en que puede alterar cualquier cosa almacenada en RAM. Con el ensamblador residente en la ROM del cartucho, el código del mismo está a salvo, pudiéndose volver a él siempre que se quiera.

Todos los ensambladores son distintos y es posible que la descripción que hagamos de la forma en que opera el MIKRO no se corresponda exactamente con la forma en que actúe cualquier otro ensamblador. Los principios, sin embargo, son los mismos, tratándose sólo de una cuestión de aprender una secuencia distinta de instrucciones. Las diferencias entre ensambladores son como las diferencias entre ordenadores: si se conocen los respectivos lenguajes, las diferencias pierden importancia. El lenguaje, en este caso, es el lenguaje ensamblador del 65Ø2. Lo que hay que aprender es a entrar un programa de forma que el ensamblador MIKRO pueda operar con él, traducirlo a código máquina y almacenar el código resultante para que pueda ejecutarse posteriormente.

Trabajando con el MIKRO

Antes de utilizar el programa MIKRO, es conveniente que conozca la forma en que aborda la tarea de ensamblar sus instrucciones. El principio se basa en que entre su programa en lenguaje ensamblador utilizando líneas numeras, tal como haría para escribir un programa en BASIC. Esto no es una coincidencia ya que puede escribir su programa aunque no esté instalado el cartucho del MIKRO. Simplemente se utilizan las facilidades del Commodore 64 por lo que concierne a la escritura de líneas de instrucción. Con tal de que no intente ejecutar su programa, no tiene por qué estar el MIKRO presente, al menos mientras no quiera ensamblarlo. Estas líneas de lenguaje ensamblador pueden guardarse en cinta exactamente de la misma forma en que se procedería para cualquier programa BASIC. Lo que distingue este programa de otro escrito en BASIC es el hecho de que utiliza instrucciones de lenguaje ensamblador así como directivas para el programa encargado del ensamblado. Sin embargo, es conveniente tener conectado el cartucho del MIKRO cuando se entra el programa. Una razón es la de poderse utilizar entonces la directiva AUTO (seguida de RETURN) para proceder a renumerar de forma automática las líneas

de su programa. Cuando se entra AUTO seguido de RETURN aparece el número 1ØØ en pantalla. Este es el número de línea inicial. Cada vez que pulse RETURN después de haber entrado una línea de BASIC o de lenguaje ensamblador, aparecerá un nuevo número de línea. Los valores irán incrementándose en la forma habitual, de forma que se verá aparecer la secuencia 1ØØ, 11Ø, 12Ø, ... y así sucesivamente. Esta posibilidad es la razón de que tenga siempre conectado el cartucho de MIKRO cuando trabajo en BASIC.

Puede modificarse la directiva AUTO para cambiar el número de líneas inicial así como el incremento utilizado. Así, por ejemplo, entrando AUTO 1Ø,5 se obtendrá un número de línea inicial de 1Ø que se irá incrementando de cinco en cinco. Puede inhibirse la numeración automática de líneas pulsando la tecla RETURN.

Otra directiva de edición muy útil que ofrece el MIKRO es DELETE, lamentablemente omitida en el Commodore 64. DELETE puede ir seguido de dos números separados por un guión. Su efecto es el de borrar las líneas con los números iguales o comprendidos entre los especificados. Así, por ejemplo, DELETE 1ØØ-2ØØ borrará las líneas 1ØØ,11Ø,12Ø ... y así sucesivamente hasta llegar a la 2ØØ. Puede omitirse uno de los números de la directiva, de forma que DELETE 2ØØ– borrará la línea 2ØØ y todas las demás líneas que ostenten un número mayor. DELETE –3ØØ, por su parte, borrará la línea 3ØØ así como todas aquellas que tengan un número de línea menor.

Además de estas directivas de edición que son igual de útiles cuando se trabaja en BASIC, el MIKRO posee dos comandos específicos para la programación en lenguaje ensamblador. En primer lugar está FORMAT, cuya misión es la de organizar la salida de información de forma que se muestren ordenadas en columnas la dirección, la etiqueta, el código de instrucción, el operando y el comentario separados, sin importar el orden en el que se hubiesen entrado. Esto es especialmente útil para programas en lenguaje ensamblador que pueden ser difíciles de comprobar, a menos que se puedan visualizar en un formato claro y ordenado. La segunda directiva que mencionábamos anteriormente es FIND. FIND debe ir seguida de un nombre, tal como el nombre de una etiqueta, de forma que saldrán por pantalla (o bien por la impresora) todas las líneas en las que aparezca el nombre en cuestión. FIND funciona también con programas BASIC y es muy útil para tener constancia de dónde se utiliza un determinado nombre de variable. Así, por ejemplo, FIND X mostrará cada una de las líneas en las que se haya utilizado X. Estas líneas no aparecen en el formato habitual del BASIC, sino en el formato correspondiente a un programa en lenguaje ensamblador. Puede restringirse el campo de actuación de la directiva a un conjunto de líneas añadiendo dos números de línea después de una coma y separados por un guión. Así, por ejemplo, FIND BUCLE,15Ø-3ØØ visualizará todas las líneas comprendidas

entre la 15Ø y la 3ØØ que contengan la palabra BUCLE. Pueden utilizarse las mismas variaciones para esta directiva que para DELETE y LIST, de forma que son perfectamente aceptables expresiones tal como FIND BUCLE,4ØØ– o bien FIND BUCLE, –2ØØØ.

Hay otras dos directivas bastante útiles. Una de ellas es NUMBER, cuya misión es la de efectuar conversiones numéricas para librarle así de esta tarea. Para cualquier número (dentro del rango habitual) que siga a NUMBER se obtendrá su expresión en hexa, decimal, octal (base ocho, no muy utilizada hoy en día) y binario. Los diferentes tipos de valores vienen identificados con el prefijo $ para el hexa, @ para el octal y % para el binario. Si entra, por ejemplo, NUMBER 123, verá aparecer en pantalla:

```
HEX     = $ØØ7B
DECIMAL = 123
OCTAL   = @ØØØ1Ø5
BINARY  = %ØØØØØØØØ1ØØØ1Ø1
```

Pueden entrarse valores en cualquiera de estas bases, de forma que tanto NUMBER $45 como NUMBER @215312, como NUMBER %11Ø111Ø111 harán que aparezca el número correspondiente en pantalla expresado en los cuatro sistemas de numeración. También existe la directiva DISASSEMBLE que hace lo mismo que el comando .D del monitor TIM.

El lenguaje ensamblador empleado por el MIKRO es muy parecido al lenguaje estándar que estableció Mostek (empresa fabricante del 65Ø2). No es exactamente idéntico, aunque las diferencias son muy pequeñas. Una que se aprecia en seguida es el hecho de que los comentarios se separen mediante un punto de exclamación en lugar de con un punto y coma. Veremos sólo algunas de estas diferencias ya que hay otras que sólo tienen interés para aquellos que tengan mucha experiencia programando en código máquina. Otro aspecto en que difiere este programa con respecto al BASIC es en lo que hace referencia a las directivas al ensamblador. No basta con suministrar un conjunto de instrucciones en lenguaje ensamblador. Hay que especificar también, por ejemplo, en qué direcciones de memoria se quiere tener ensamblado el código. De todas formas, tal como ocurre muy a menudo, verá que una pequeña parte de estas instrucciones basta para la gran mayoría de aplicaciones. El ensamblador MIKRO no colocará el código en las direcciones que hemos estado utilizando hasta ahora para los ejemplos. Eso se debe a que emplea estas direcciones para su propio uso. Podemos utilizar 4K de memoria (a partir de la dirección $CØØØ) para nuestros programas. Alternativamente puede ensamblarse el código en la dirección definida cargando un 127 en la posición 56 de memoria. Corresponde a la dirección $7FØØ. Para empezar el ensamblado de código a partir de la siguiente dirección, la

$7FØ1, habría que poner al principio del programa ensamblado la línea:

```
1ØØ *=$7FØ1
```

Normalmente empezaremos siempre el programa de forma parecida con el fin de ubicar la memoria a utilizar. Sin embargo, debe asegurarse antes de que se haya reservado la memoria que va a emplear. Eso puede llevarse a cabo utilizando POKE 56,127 en BASIC, o bien con los correspondientes LDA #127 y STA 56 al principio del código máquina. Si se está escribiendo un programa sólo en código máquina (sin nada de BASIC), se puede disponer de toda la RAM. En este caso parece lógico empezar cargando el código en la dirección habitual de inicio del BASIC. De esta forma, si se quiere, pueden emplearse las directivas LOAD y SAVE. Si no se utiliza *=$7FØ1 o un comando inicial parecido, se ensamblará el código a partir de la dirección $Ø33C. Esta es una zona de RAM que se conoce con el nombre de «buffer del cassette». Se utiliza sólo durante las operaciones ligadas a LOAD y STORE. terminando en la posición $Ø3FC. No es exactamente un lugar ideal para colocar el código si tiene intención de emplear las mencionadas directivas LOAD y STORE.

Después de definir la dirección inicial pueden definirse a continuación nombres de etiquetas. Puede hacerse asignándolas a direcciones o bien a bytes. Así, por ejemplo, podemos tener:

```
11Ø SCRNST = $Ø4ØØ
12Ø CARRET = $ØD
```

en las dos líneas siguientes. No provocan generación alguna de código al ensamblarlas. Lo que hacen es asegurar que dondequiera que aparezcan estas etiquetas se colocarán los números correctos en su lugar. Dondequiera que aparezca la palabra SCRNST (SCReeN STart, o sea inicio de pantalla) se colocará el número $Ø4ØØ. Dondequiera que aparezca la palabra CARRET (CARriage RETurn) se colocará en su lugar el código $ØD. Utilizando etiquetas de esta forma se logra que el programa sea mucho más fácil de seguir. Si además define estas etiquetas el principio del código, podrá ver el valor a que equivalen sin tener que buscar a lo largo de todo el programa. Los nombres de las etiquetas deben tener seis letras como máximo. Si se omite la definición de alguna de estas etiquetas no puede tener lugar el ensamblado y en el caso de intentar llevarlo a cabo, se obtendrá un mensaje de error. No hace falta que una etiqueta se defina explícitamente de esta forma, puede definirse también implícitamente en el programa, tal como por ejemplo con:

```
22Ø BUCLE : LDA #24
```

Iremos escribiendo, pues, nuestro programa en ensamblador tal como haríamos con un programa en BASIC, aunque con una sola instrucción por línea. Si terminamos el programa con una instrucción RTS nos aseguramos de que se devuelva correctamente el control al BASIC. Las líneas de ensamblador no pueden ir nunca seguidas por líneas de BASIC. Hay que colocar, sin embargo, una línea END al final de la sección de lenguaje ensamblador. No basta con ensamblar e ir automáticamente al BASIC. Si no está la sentencia END, el MIKRO intentará leer las líneas de BASIC lo que originará un mensaje de error. La principal diferencia estriba, de hecho, en que el MIKRO es mucho más exigente sobre la forma en que se entran las sentencias de ensamblador. Hay que dejar, por ejemplo, al menos un espacio entre cada uno de los campos que componen una línea. «Campo», en este contexto, equivale a sección, siendo las diferentes secciones la etiqueta, el código de la instrucción, el operando y el comentario. No se puede entrar, por ejemplo, la línea:

```
BUCLELDA#$45!EMPEZAMOS
```

y confiar en que el ensamblador pueda interpretarla de la misma forma en que actuaría normalmente el BASIC del Commodore 64 para interpretar instrucciones compactadas de este tipo. A lo que se llega habitualmente cuando se intenta ensamblar una línea de las características de la anterior es al mensaje de error NO OPCODE que significa que el ensamblador es incapaz de separar las distintas partes de la instrucción. Si por el contrario entra la línea anterior en la forma:

```
BUCLE   LDA   #$45   !EMPEZAMOS
```

o sea separando los diferentes campos, el ensamblador podrá traducirla correctamente.

Vamos a ir viendo progresivos refinamientos y útiles amplicaciones de nuestro repertorio de comandos, aunque los más importantes de cara a hacer un uso adecuado del MIKRO son las que ya hemos visto. Si cada programa que escriba empieza con un *= para ubicar la memoria y define a continuación todas las etiquetas que le hagan falta, ya habrá dado un primer paso hacia la consecución de un resultado satisfactorio del programa. Si utiliza un número que no vaya precedido de uno de los símbolos de identificación que hemos visto, tal como $, será interpretado como un número decimal. Empleando el símbolo $ como prefijo identificamos el número como hexa. Cuando comentamos la directiva NUMBER ya vimos también el significado de los prefijos @ y %. El asterisco (*) se considera como equivalente a la «dirección actual», o sea la dirección inicial de la instrucción en la que aparece el símbolo en cuestión. Así pues, el asterisco constituye una forma muy útil de indicar una dirección sin conocer el valor a que corresponde. El signo de exclamación se emplea para indicar el inicio de

una línea correspondiente a un comentario, tal como REM en BASIC.

Cualquier ensamblador admite también lo que se denomina «directivas de ensamblado» o bien «pseudoinstrucciones» y que no son más que instrucciones para el mismo programa ensamblador. El MIKRO no constituye una excepción. Vale la pena comentar algunas de estas directivas ya que se las utiliza ampliamente (con una u otra versión) en ensambladores para otros microprocesadores. Ya hemos visto el * que representa la dirección actual. Otras tres pseudoinstrucciones son WOR, BYT y TXT. Las tres sirven para colocar datos que no sean instrucciones en memoria. WOR se encarga de colocar una «palabra» de dos bytes en memoria. Separa el número en byte alto y byte bajo y los almacena en el orden habitual de menos a más significativo. WOR $7F12, por ejemplo, colocará en memoria los bytes $12 y $7F, en este orden. Serán almacenados a partir de la «dirección actual». Supongamos, por ejemplo, que se quisiera hacer un programa que utilizase una tabla de valores que empezase en la dirección $9FFØ. Podría incluir para ello en su programa de ensamblador las líneas:

```
2ØØ *=$7FF1
21Ø WOR $2E14,$115B,$513A
```

con lo que almacenaría, a partir de la dirección $7FF1 la secuencia siguiente de bytes: 14,2E,5B,11,3A y 51. Vea que pueden utilizarse varias direcciones después de WOR mientras vayan separadas por comas.

La pseudoinstrucción BYT realiza la misma función aunque opera sobre bytes aislados, de forma que lo que hará BYT $ØD será colocar el código del RETURN en la posición de memoria señalada por la dirección actual. BYT puede utilizarse también para almacenar los códigos ASCII de caracteres a través del símbolo ' . Si en su programa usa BYT'A, hará que se guarde el código ASCII de la A en la dirección actual. Sin embargo, TXT constituye una forma más adecuada para almacenar una cadena de caracteres. Después de TXT hay que colocar una cadena alfanumérica entre comillas. Los códigos ASCII de todas las letras, incluidos los espacios y signos de puntuación, serán colocados en memoria a partir de la dirección actual. Así, por ejemplo, a través de TXT "PULSE CUALQUIER TECLA" se almacenarán los códigos ASCII de todas las letras y espacios que se encuentren entre las comillas de la instrucción.

Hágalo funcionar ¡ya!

Cuando se ha entrado un programa en lenguaje ensamblador, puede grabarse igual que si se tratase de un programa BASIC. Este

punto es importante ya que se trata de almacenar su «código fuente». El código fuente es el conjunto de instrucciones que se ensamblarán a código máquina, las cuales, si se tiene una versión grabada de las mismas, se pueden volver a cargar y editarlas y reensamblarlas siempre que lo desee. Como el código fuente puede grabarse como si fuese BASIC, hay que ser cuidadoso a la hora de poner etiquetas en sus cintas o discos con el fin de no confundirlos. El código fuente no se ejecutará cuando pulse RUN. Para ensamblar su código fuente tiene que teclear ASSEMBLE seguido de RETURN. El ensamblador analizará entonces en tres pasadas sucesivas su código fuente. Debe hacerse así ya que puede que algunas de las etiquetas empleadas no hayan sido definidas al principio del programa. Todos los errores que se encuentren serán detectados en la primera o en la segunda «pasada» sobre el código fuente, deteniéndose, en este caso, el proceso de ensamblado con un mensaje de error. Tal como era de esperar, el mensaje de error indicará en qué línea se produjo el error así como el tipo de error detectado. Si no se encuentran errores, es de esperar que vea aparecer en la pantalla algo parecido a:

```
ASSEMBLE
* PASS(1) *
* PASS(2) *
* PASS(3) *
*** ASSEMBLY COMPLETE ***
START ADDRESS —$9FØ1
END ADDRESS   —$9F43
```

Estas direcciones son útiles si se tiene intención de utilizar la directiva SAVE del monitor TIM para guardar el código máquina en cinta o disco. La dirección inicial es también la que se deberá utilizar cuando se quiera ejecutar este código a través de SYS. Vea que el ensamblador MIKRO, por sí mismo, no inicia la ejecución del programa, a menos que la última línea del código fuente sea SYS seguida de la dirección inicial adecuada.

En este capítulo no pretendo ofrecer una descripción completa del ensamblador MIKRO. Mi propósito es el de proporcionar una visión de cómo puede utilizarse un ensamblador. Si está preparado para usar el MIKRO, podrá interpretar sin ningún tipo de problemas el manual de instrucciones (más bien breve) que lo acompaña, y lo que es más importante: podrá utilizar otros ensambladores e incluso ensambladores para otros microprocesadores, en el caso de que deba cambiar algún día de máquina.

9. Unos últimos detalles

Uno de los principales problemas que surgen cuando se escribe un libro sobre código máquina para principiantes es saber dónde debe uno detenerse. Podrían escribirse volúmenes y más volúmenes sobre la programación en código máquina del Commodore 64, dejándonos aún así siempre cosas en el tintero, debiendo elegir otra vez un punto final arbitrario. Mi propósito ha sido el de introducir el tema y situarle a un nivel a partir del cual pueda empezar a hacer progresos por su propia cuenta. Una vez que haya alcanzado este estadio, podrá utilizar otros libros que se encuentran disponibles para abordar el tema del código máquina a un nivel más avanzado. Este capítulo está dedicado a atar los cabos que hayan podido quedar sueltos, presentando algunas nuevas instrucciones e ilustrando cómo sacar partido de algunas de las características del Commodore 64.

La pila

No puede adentrarse uno mucho en el campo de la programación en código máquina sin tropezar con la palabra *pila.* Una pila es una zona de memoria cuya finalidad primordial es la de guardar los valores almacenados en los registros. No hay ningún conjunto especial de circuitos integrados que se utilicen como pila, aunque sí se reserva una zona especial de la RAM para esta finalidad. Para el microprocesador 65Ø2 hay que utilizar la memoria comprendida entre la posición $1ØØ y la $1FF. Lo que es posible que encuentre difícil de entender por el momento es por qué hemos de utilizar la memoria de esta forma.

Consideremos un sencillo ejemplo. Supongamos que tenemos un programa en el que se utiliza el registro A para guardar el código ASCII de un determinado carácter. Supongamos ahora que en medio del programa queremos generar un retardo a través de un contador en el mismo registro A. Tan pronto como carguemos el valor inicial del contador en el acumulador perderemos el código numérico ASCII que teníamos almacenado allí y si queremos utilizar otra vez el registro A durante el resto del programa, deberemos volver a colocar en él el valor perdido. Ahí está la razón de ser de la pila. A través de una instrucción de un solo byte podemos almacenar el contenido del acu-

mulador o bien del registro de estado en la memoria de la pila. De la misma forma, a través de una instrucción similar podemos devolver estos valores a los registros adecuados. La acción de almacenar un valor en la pila se conoce (en inglés) con el nombre de «pushing» y la acción de recuperar estos valores se conoce (también en inglés) con el nombre de «polling».

Veremos en breve un ejemplo de un programa que utilice la pila aunque, por ahora, vamos a volver a cuestiones más sencillas. El resto de este capítulo está dedicado, de hecho, a mostrar ejemplos de programas que pueden constituir la base para que desarrolle rutinas realmente útiles para el Commodore 64. Me gustaría poner de relieve que todavía no lo sabe todo sobre el código máquina. De ahora en adelante, lo que necesita es práctica, así como sacar provecho de toda la información sobre el tema que llegue a sus manos. Estudie atentamente, por ejemplo, todos los programas para Commodore 64 que contengan código máquina. Incluso si éstos están en forma de bytes que se cargan en memoria, puede desensamblarlos y ver lo que hacen. Procediendo así podrá descubrir direcciones que serán muy útiles en sus propios programas. De ahora en adelante, al menos en potencia, todo es útil para usted.

La rutina KEYBEEP

Vamos a ver una rutina que ilustra muchos de los puntos que ya hemos mencionado. Nos servirá también para introducirnos en una programación de tipo más avanzado. Esta vez queremos diseñar un programa que provoque un breve sonido cada vez que se pulse una tecla. Este programa deberá actuar cuando trabajemos en BASIC, de forma que lo que necesitamos es una forma de insertar un fragmento de código máquina dentro del BASIC. Es un problema bastante distinto del de hacer un programa en código máquina que se ejecute y vuelva a continuación al BASIC, de forma que tendremos que aprender algo más del Commodore 64 para llevar a cabo esta tarea.

Para empezar, tenemos el problema de cómo «entrar» en las rutinas que utiliza el BASIC. Afortunadamente, el Commodore 64 utiliza un «bloque de conexión» extremadamente manejable. Lo que quiero significar por bloque de conexión es un fragmento de código almacenado en RAM en lugar de ROM. Cualquier código almacenado en RAM puede ser alterado, al contrario de lo que ocurre con la ROM. La finalidad de todo eso es, de hecho, permitir que se realicen cambios por «parcheo». En este contexto parchear significa insertar un fragmento de nuestro programa en una rutina utilizada por el sistema operativo del Commodore 64.

En una operación de este tipo, lo más difícil es encontrar el lugar

donde efectuar el mencionado parcheo. Existe una rutina ubicada en $Ø324 y $Ø325 que podemos utilizar de esta forma, aunque sólo se ejecuta cuando se pulsa RETURN. Esto no es lo que queríamos ya que pretendemos que el programa KEYBEEP se ejecute cada vez que se pulse una tecla cualquiera. Es casi imposible descubrir por nuestros propios medios unas direcciones adecuadas en un tiempo razonable, así que recurriremos al desensamblado de estas rutinas. Un cuidadoso análisis del mismo nos revela un par interesante de direcciones en las posiciones $Ø28F y $Ø29Ø. Contienen otra dirección, la $EB48, que corresponde a la rutina de decodificación del teclado, lo cual las hace sumamente interesantes. Ocurre que cuando se pulsa una tecla se provoca la llegada de unas señales eléctricas a un determinado port. El 65Ø2 debe leer a continuación estas señales y convertirlas en un número de un solo byte, utilizando un código distinto para cada tecla. Evidentemente, el Commodore 64 lleva a cabo todo ello por partes, comprobando primero si se ha pulsado una tecla y llevando a cabo posteriormente la decodificación. Una ojeada al desensamblado de la rutina del teclado (que empieza en $EA87) revela que se utiliza esta dirección (Ø28F) de forma indirecta. Esto significa que el Commodore 64 lee los bytes $Ø28F y $Ø29Ø cada vez que se pulsa una tecla y salta, entonces, a la dirección formada por el contenido de estos dos bytes. Esto se parece bastante a lo que estábamos buscando. Es aquí donde colocamos nuestro parche.

Lo que haremos es lo siguiente: en la dirección $Ø28F sustituiremos el byte original ($48) por el byte bajo de una nueva dirección y en la posición $Ø29Ø colocaremos el byte alto de la misma. Podremos, entonces, escribir una rutina propia que empezará en la nueva dirección que habremos definido. Al final de esta rutina tendremos una instrucción JMP. La dirección que acompañará a JMP será la correspondiente a la rutina original de decodificación del teclado: $EB48. De esta forma, cada vez que se pulse una tecla, se ejecutará nuestra rutina antes de que la máquina proceda a decodificar la tecla pulsada. La rutina que vamos a parchear es una muy sencilla. Cargará los valores adecuados en las direcciones del circuito integrado de sonido, tal como haría un programa de BASIC. Tendrá que incluir también una rutina de retardo a fin de que el sonido dure lo suficiente como para que sea oído y finalizará volviendo a la dirección $EB48. Parece todo muy sencillo al leerlo, pero se van a introducir bastantes conceptos nuevos. Algunos harán referencia a nuevas instrucciones y otros serán fruto de nuevos conocimientos acerca del Commodore 64. Una vez llegados a este punto, todo es útil.

Empezaremos con un diagrama de flujo. La figura 9.1 muestra el que está asociado a lo que queremos hacer. Como cualquier otro buen diagrama de flujo, no especifica con demasiado detalle lo que se va a hacer. Inmediatamente después del inicio se llega al bloque de

decisión «se ha pulsado alguna tecla?». Esta acción la lleva a cabo el sistema operativo, así que no nos preocuparemos más de ella. Si se ha pulsado una tecla, cargamos los valores adecuados en las direcciones de sonido. Es una tarea más aburrida que difícil (y es algo que debería guardar en cinta, después de haberlo tecleado, de forma que pueda utilizarlo en cualquier otro momento). Sigue a este proceso de carga un retardo. A continuación se provoca el final de la señal acústica cargando ceros en algunas de las posiciones ligadas al sonido. Si no lo hiciésemos así, el sonido continuaría después de que se hubiese pulsado la tecla, que no es lo que queremos. Finalmente, saltamos al BASIC, preparados ya para cuando se pulse la tecla siguiente.

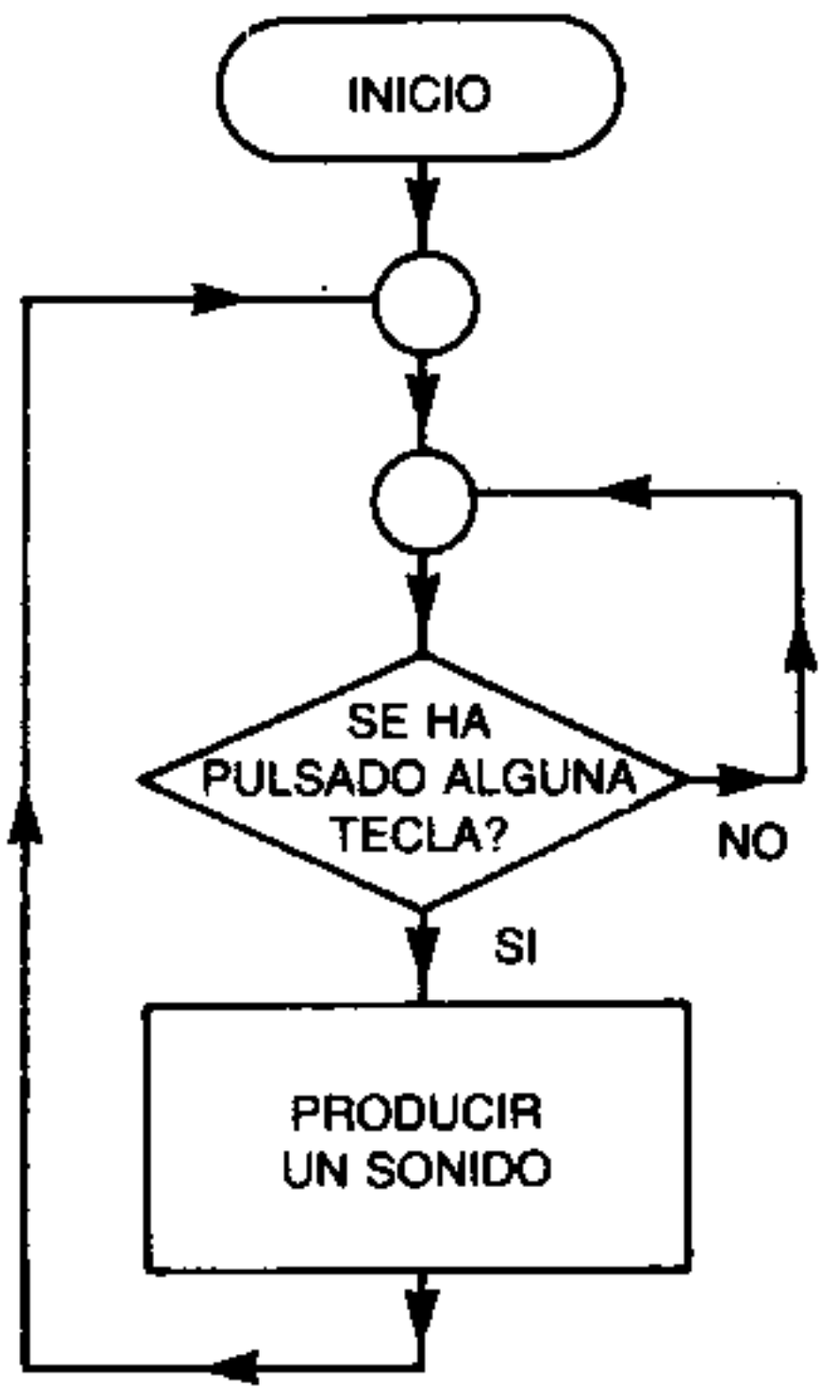


Fig. 9.1 Diagrama de flujo del programa «KEYBEEP» encargado de hacer que suene una nota determinada cada vez que se pulse una tecla.

La figura 9.2 muestra la versión en lenguaje ensamblador de todo lo que hemos mencionado. Se ha sacado en forma de un conjunto de líneas numeradas, que es el formato que utiliza el ensamblador MIKRO. Puede ensamblar este programa a mano si cree que le conviene realizar esta práctica, aunque en la figura 9.3 se muestra el listado de un proceso de ensamblado del mismo programa a través del MIKRO. Podemos ver ahí las direcciones y los códigos de los bytes así como la versión en ensamblador de las instrucciones. Recuerde que en el caso de estar instalado el cartucho del MIKRO, la dirección inicial será la $7FØØ. Si quiere, puede reescribir el programa para ubicarlo en una zona más alta de memoria, en el caso de que no esté utilizando el

```
100 *=$7F00
120 LDA #<PUNTO
130 STA $28F
140 LDA #>PUNTO
150 STA $290
160 RTS
170 PUNTO PHA
175 LDA #15
180 STA 54296
190 LDA #190
200 STA 54273
205 LDA #248
210 STA 54278
220 LDA #17
230 STA 54273
240 LDA #37
250 STA 54272
260 LDA #17
270 STA 54276
280 JSR RETARDO
290 LDA #0
300 STA 54276
310 STA 54277
320 STA 54278
330 PLA
340 JMP $EB48
350 RETARDO LDA #100
360 STA 251
370 BUCLE1 STA 252
380 BUCLE2 DEC 252
390 BNE BUCLE2
400 DEC 251
410 BNE BUCLE1
420 RTS
430 END
```

Fig. 9.2 Versión en ensamblador de la rutina KEYBEEP. Se ha escrito en forma de líneas numeradas para el ensamblador MIKRO.

MIKRO. Si se quiere usar otra dirección inicial, hay que modificar parte del código. El punto crítico es la dirección $7F3C asociada a la subrutina de retardo. Si se cambia la dirección de inicio del programa, deberá cambiarse también la dirección de la subrutina de retardo.

```
100  7F00                  *=$7F00
120  7F00                          LDA #<PUNTO
130  7F02 8D8F02                   STA $28F
140  7F05 A97F                     LDA #>PUNTO
150  7F07 8D9002                   STA $290
160  7F0A 60                       RTS
170  7F0B 48          PUNTO        PHA
175  7F0C A90F                     LDA #15
180  7F0E 8D18D4                   STA 54296
190  7F11 A9BE                     LDA #190
200  7F13 8D01D4                   STA 54273
205  7F16 A9F8                     LDA #248
210  7F18 8D06D4                   STA 54278
220  7F1B A911                     LDA #17
230  7F1D 8D01D4                   STA 54273
240  7F20 A925                     LDA #37
250  7F22 8D00D4                   STA 54272
260  7F25 A911                     LDA #17
270  7F27 8D04D4                   STA 54276
280  7F2A 203C7F                   JSR RETARDO
290  7F2D A900                     LDA #0
300  7F2F 8D04D4                   STA 54276
310  7F32 8D05D4                   STA 54277
320  7F35 8D06D4                   STA 54278
330  7F38 68                       PLA
340  7F39 4C48EB                   JMP $EB48
350  7F3C A964        RETARDO      LDA #100
360  7F3E 85FB                     STA 251
370  7F40 85FC        BUCLE1       STA 252
380  7F42 C6FC        BUCLE2       DEC 252
390  7F44 D0FC                     BNE BUCLE2
400  7F46 C6FB                     DEC 251
410  7F48 D0F6                     BNE BUCLE1
420  7F4A 60                       RTS
```

Fig. 9.3 Listado obtenido mediante el ensamblador MIKRO en el que pueden verse los códigos hexa así como las instrucciones de ensamblador.

Detalles y más detalles

Vamos a analizar en detalle el programa en lenguaje ensamblador.

Empezamos de la forma habitual, definiendo la dirección inicial $7FØØ. A continuación, se coloca una nueva dirección en las posiciones $Ø28F y $Ø29Ø. Esta dirección corresponderá a la dirección inicial de la rutina de sonido. En el momento de escribir estas líneas no se

sabe qué dirección será ésta, así que se utiliza la etiqueta PUNTO. El empleo de los símbolos < y > no es exclusivo del MIKRO. Verá utilizarlos en muchos programas en ensamblador del 65Ø2. El signo < significa «byte bajo de», mientras que el signo > significa «byte alto de». Así pues, lo que se hace en las líneas 12Ø a 15Ø es cargar los dos bytes de la dirección de PUNTO en las posiciones $Ø28F y $Ø29Ø del bloque de conexión. Una vez que se haya llevado a cabo esto, siempre que se pulse una tecla se hará que la máquina vaya a la dirección definida por PUNTO.

Lo que queremos hacer ahora es ejecutar una sola vez esta parte de «incialización». Una vez que se haya cargado el valor de estas posiciones de memoria, queremos volver al BASIC, yendo a ejecutar el resto del programa sólo cuando se pulse una tecla. La instrucción siguiente, en la línea 16Ø, será pues RTS, con lo que se devolverá el control al BASIC. El resto del programa es el trozo que se ejecuta cada vez que se pulsa una tecla.

Este trozo del programa empieza en la línea 17Ø, con la etiqueta PUNTO. Con ella aseguramos que cuando llevemos a cabo el ensamblado se cargue la dirección de la primera instrucción en el «bloque de concexión». La primera instrucción es PHA. Esto es importante. PHA significa «guardar el acumulador en la pila» (del inglés PusH the Accumulator onto the stack). Cuando la máquina quiera decodificar el resultado de pulsar una tecla, esperará el código correspondiente en el acumulador. Si destruimos este byte, no podemos esperar que la máquina funcione normalmente. A través de PHA ponemos a salvo el contenido del acumulador en la pila. Podremos recuperarlo al final de nuestra subrutina, de forma que cuando el programa salte a la rutina de decodificación el acumulador contenga el mismo valor que contendría si nuestro programa no existiese. Las líneas 175 a 27Ø cargan, a continuación, los bytes adecuados en las direcciones de sonido con el fin de producir un sonido grave. Evidentemente, usted puede experimentar con estos valores. Las direcciones son las mismas que se ilustran en el manual de BASIC del Commodore 64.

La siguiente etapa consiste tan sólo en un retardo. Se ha empleado para ello una subrutina, aunque podía haberse colocado perfectamente el código en la parte principal del programa. La subrutina consiste en un programa normal y corriente de retardo que utiliza direcciones de la página cero de memoria para llevar a cabo su función. Evite usar los registros X e Y en previsión del caso que fuesen también utilizados por la rutina de decodificación. La razón para ser cuidadosos en este punto estriba en que no existen instrucciones para guardar el contenido de los registros X o Y en la pila. La única forma de hacerlo es transfiriendo los bytes por turno al acumulador y guardando el contenido de este acumulador en la pila. Al recuperar los bytes de la misma, éstos van a parar también al acumulador (en el caso de utili-

zarse PLA para ello) desde donde pueden ser transferidos a otros registros. Todo ello es tan pesado que vale la pena intentar evitarlo. Yo lo he conseguido utilizando las direcciones de la página cero para almacenar los números de la cuenta. El almacenamiento de un valor de 1ØØ (en base diez) produce un sonido razonablemente corto.

Finalmente, el programa termina guardando un cero en tres de las direcciones asociadas al sonido, con lo que finaliza la señal acústica generada. El valor que se encontraba originariamente en el acumulador es restituido a su ubicación original a través de la instrucción PLA, finalizando la rutina con un salto a la dirección $EB48 para proceder a decodificar la tecla pulsada.

Después de ensamblar este programa y utilizar SYS 32512 para ejecutarlo, verá como se obtiene una señal acústica (y un leve retardo) cada vez que se pulsa una tecla. Si no ocurre así, quizás haya olvidado que el sonido debe provenir del altavoz del TV con lo que hay que ajustar adecuadamente el mando de volumen del mismo. Esta rutina podría iniciarle en diversos nuevos caminos. Para empezar, podría hacer un programa de ayuda a ciegos que proporcionase una nota diferente según la tecla que se pulsase. Ello representaría utilizar el código contenido en el registro A (en el momento de salvaguardar el contenido de los registros) para modificar el byte de «tono» asociado al sonido generado.

Renumeración: un ejemplo de diseño de un programa

Como último ejemplo, vamos a estudiar desde el principio hasta el final el diseño de un programa bastante más elaborado. ¿Hasta el final? Ningún programa está realmente terminado hasta que uno pueda colocar la mano sobre el corazón y jurar que no es posible mejorarlo. Por ningún concepto voy a llegar yo a este punto. De hecho, espero que descubra aspectos susceptibles de mejora de forma que este programa le sirva más como guía que como algo que sólo vaya a ensamblar y utilizar. El programa es una utilidad de la que desgraciadamente carece el Commodore 64: una rutina de renumeración. La idea consiste un poder cargar en memoria un número inicial junto con el valor de un incremento de forma que se puedan renumerar las líneas de un programa BASIC utilizando simplemente una instrucción del tipo SYS-dirección.

Como siempre, es aconsejable empezar con una etapa de planificación. La idea básica consiste en que podemos obtener la primera dirección de un programa BASIC a través del contenido de las posiciones $2B y $2C. La línea siguiente empieza en una posición cuya dirección está almacenada en los dos primeros bytes asociados a esta línea, estando el número de línea guardado en los dos bytes siguien-

1. Obtener la dirección del primer byte del programa BASIC.
2. Guardar el primer byte en LB. Este es el byte bajo la dirección de inicio de la línea siguiente.
3. Guardar el segundo byte en HB. Este es el byte alto de la dirección de inicio de la línea siguiente.
4. Colocar el byte bajo del número de línea en la tercera dirección de la línea.
5. Colocar el byte alto del número de línea en la cuarta dirección de la línea.
6. Sumar el valor del incremento al número de línea (almacenado).
7. Comprobar el valor de la dirección de la línea siguiente (almacenado).
8. Repetirlo todo, esta vez utilizando la dirección de la nueva línea, almacenada en HB y LB.

Fig. 9.4 Resumen de lo que debería hacer el programa de renumeración.

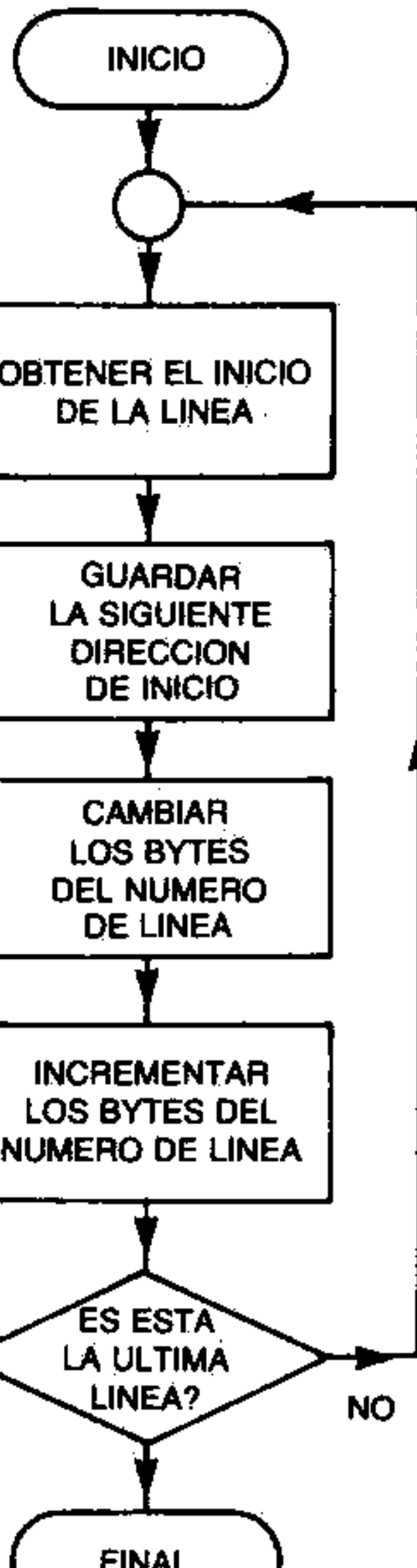


Fig. 9.5 Diagrama de flujo para el programa de renumeración.

tes. Deberíamos, así pues, diseñar un programa interativo que fuese de una línea a la siguiente, cambiando cada vez los números de línea correspondientes, hasta encontrar como dirección de la línea siguiente un ØØØØ. Esto representaría el final del programa BASIC así como el momento de detener la ejecución de la rutina de renumeración. Todo ello parece ser suficientemente razonable. La figura 9.4 describe todo el proceso que hemos explicado.

El siguiente paso, tal como debe suponer, consiste en dibujar un diagrama de flujo. La figura 9.5 muestra el que corresponde al proceso descrito. Este diagrama de flujo reseña con un poco más de detalle lo que queremos hacer. En este caso, le he añadido un poco de información suplementaria. Hemos de mantener almacenada la «dirección de la línea actual» en algún lugar, preferiblemente en dos direcciones de la página cero. Las he bautizado con los nombres AD y AD+1. Deberíamos guardar también el «número de línea actual» en otro par de posiciones de memoria a las que asociaremos los nombres LIN y LIN+1. Finalmente, necesitamos un incremento. Reservaremos para ello sólo el byte de la posición 255 de memoria, de forma que especificaremos que no se podrán renumerar los programas utilizando incrementos mayores que 255. No parece que ello constituya una limitación demasiado grave.

Quiero insistir en que se trata de un programa sencillo. Sólo renumerará las líneas en sí, sin alterar las direcciones de los GOTO o bien de los GOSUB. Es útil, sin embargo, como ejercicio de diseño de utilidades en lenguaje máquina. Es útil también como programa de utilización diaria, con tal que tenga siempre presente sus limitaciones. Es ideal para renumerar el tipo de programas que se utilizan con el ensamblador MIKRO. En cualquier caso, vamos a echarle una ojeada.

La renumeración vista de cerca

El programa se muestra en la figura 9.6, en el formato ligado a MIKRO. Es bastante más largo y elaborado de lo que hemos visto hasta ahora. Quiero analizarlo en detalle, apuntando la razón de ser de cada uno de los pasos efectuados. Raramente encontrará una descripción detallada de como funciona un programa en lenguaje máquina, así que aproveche la oportunidad que se le presenta. A menudo intentará seguir programas aparecidos en revistas y estas explicaciones le serán de gran utilidad para llevar a cabo dicha tarea. El programa empieza en la forma habitual definiendo la dirección inicial del mismo así como el valor de diversas etiquetas. Se ha elegido como dirección inicial la $7FØØ ya que utilicé el MIKRO para desarrollar este programa (cuando un programa empieza a tener las dimensiones de

```
100 *=$7F00
110 AD=251
120 INC=255
130 LIN=253
140 CLC
150 LDA 43
160 STA AD
170 LDA 44
180 STA AD+1
190 BUCLE LDY #0
200 LDA (AD),Y
210 PHA
220 INY
230 LDA (AD),Y
240 PHA
250 INY
260 LDA LIN
270 STA (AD),Y
280 INY
290 LDA LIN+1
300 STA (AD),Y
310 LDA LIN
320 ADC INC
330 STA LIN
340 BCC NXT
350 INC LIN+1
360 CLC
370 NXT PLA
380 STA AD+1
390 PLA
400 STA AD
410 BNE BUCLE
420 LDA AD+1
430 BNE BUCLE
440 RTS
450 END
```

Fig. 9.6 Listado del programa de renumeración. Este es el formato que acepta el ensamblador MIKRO.

Como número de línea inicial, 1ØØØ está compuesto de dos bytes. Divida 1ØØØ por 256: el resultado es 3.9Ø625. La parte entera es 3. Tome ahora 1ØØØ – 3*256 que es 232. 232 será, pues, el byte bajo del número de línea.

Fig. 9.7 Expresión de un número como 1ØØØ en forma de dos bytes.

```
100 7F00                *=$7F00
110 7F00                AD          = 251
120 7F00                INC         = 255
130 7F00                LIN         = 253
140 7F00 18                         CLC
150 7F01 A52B                       LDA 43
160 7F03 85FB                       STA AD
170 7F05 A52C                       LDA 44
180 7F07 85FC                       STA AD+1
190 7F09 A000           BUCLE       LDY #0
200 7F0B B1FB                       LDA (AD),Y
210 7F0D 48                         PHA
220 7F0E C8                         INY
230 7F0F B1FB                       LDA (AD),Y
240 7F11 48                         PHA
250 7F12 C8                         INY
260 TF13 A5FD                       LDA LIN
270 7F15 91FB                       STA (AD),Y
280 7F17 C8                         INY
290 7F18 A5FE                       LDA LIN+1
300 7F1A 91FB                       STA (AD),Y
310 7F1C A5FD                       LDA LIN
320 7F1E 65FF                       ADC INC
330 7F20 85FD                       STA LIN
340 7F22 9003                       BCC NXT
350 7F24 E6FE                       INC LIN+1
360 7F26 18                         CLC
370 7F27 68             NXT         PLA
380 7F28 85FC                       STA AD+1
390 7F2A 68                         PLA
400 7F2B 85FB                       STA AD
410 7F2D D0DA                       BNE BUCLE
420 7F2F A5FC                       LDA AD+1
430 7F31 D0D6                       BNE BUCLE
440 7F33 60                         RTS
```

Fig. 9.8 Listado del programa de renumeración obtenido a través del MIKRO. Ha sido renumerado por el propio programa.

éste, prefiero utilizar un ensamblador). Cualquiera que sea la dirección inicial que utilice, debe asegurarse de que corresponde a una zona de memoria protegida. Colocando el programa en memoria protegida, la carga de un programa BASIC no modificará los bytes de código máquina. Si olvida definir el límite superior de la memoria a través de un POKE a la dirección 56, aparecerán problemas en un momento u otro. La etiqueta AD corresponde a la dirección de la pági-

na cero utilizada para el byte bajo de la dirección de la línea sobre la cual está trabajando el programa. Esta dirección cambiará de línea en línea, de forma que utilizaremos bastante la etiqueta AD. INC es el byte asociado al incremento, o sea el valor que separará dos números de líneas consecutivos. Normalmente será 1Ø, pero el poder cambiar este valor hace más flexible el programa. Puede utilizarse cualquier valor de 1 a 255. LIN representa el byte bajo del número de línea para cada una de las líneas renumeradas.

Después de esta parte de inicialización, empieza el programa propiamente dicho. Se pone a cero el bit de acarreo y se carga el acumulador con el contenido de la posición 43 de memoria. Este es el byte bajo de la dirección del primer byte de BASIC y lo colocaremos en AD, la dirección asociada a la línea actual. Cargamos a continuación el acumulador con el contenido de la posición 44 en la que se encuentra el byte alto de la dirección de inicio del BASIC, colocándolo en AD+1. De esta forma, AD y AD+1 pasan a contener una copia de los contenidos de las posiciones 43 y 44 de memoria, o sea, de los bytes que componen la dirección del primer byte de nuestro programa BASIC.

El bucle empieza ahora. Se pone a cero el registro Y. LDA(AD),Y cargará el acumulador con el contenido de la dirección almacenada en AD y AD+1. Este es el primer byte del programa BASIC y corresponde al byte bajo de la dirección de la línea siguiente. Lo guardamos en la pila a través de PHA e incrementamos a continuación el registro Y. Debido a haber incrementado el registro Y, el siguiente LDA(AD),Y cargará en el acumulador el segundo byte de la línea de BASIC. Corresponde al byte alto de la dirección de la línea siguiente. También se le guarda en la pila, después del byte bajo. Se incrementa a continuación otra vez el registro Y. El siguiente paso consiste en obtener de LIN el byte bajo de nuestro primer número de línea. Se le almacena en el siguiente byte de la línea, otra vez a través del direccionamiento Y-indirecto. Se incrementa otra vez el registro Y, se carga en el acumulador el byte alto del número de línea inicial y se le almacena en la línea utilizando otra vez el direccionamiento indirecto. Con ello completamos la renumeración de la primera línea con el primero de los números de línea seleccionados.

El siguiente paso consiste en sumar el incremento al número de línea. Cargamos el acumulador con el contenido de LIN y le sumamos el incremento procedente de INC. Se coloca el número contenido en el acumulador (el byte bajo) otra vez en LIN y, en el caso de existir acarreo, se incrementa LIN+1. Se pone a continuación el bit de acarreo a cero (como medida de seguridad). Ahora hemos de colocar la dirección del siguiente número de línea en AD y AD+1 con el fin de poder repetir todo el proceso. El byte alto del siguiente número de línea fue el último que se guardó en la pila, así que es el que saldrá en primer

lugar de ella, a través de la instrucción PLA. Se le almacena en la posición AD+1. Se recupera a continuación de la pila el byte bajo y se le almacena en la posición AD. Podemos comprobar, una vez llegados a este punto, si el byte bajo de la siguiente dirección es un Ø. Si no es así, es señal de que no hemos alcanzado aún el final del programa y volvemos al principio del bucle. Sin embargo, si ambos bytes son igual a cero, es señal de que hemos alcanzado el final del programa, con lo que RTS nos devolverá al BASIC.

¿Cómo podemos utilizar este programa? Hemos de reservar la memoria y colocar el código máquina en su lugar. A continuación hemos de cargar el programa BASIC y decidir un número de línea inicial así como un incremento. Luego hemos de cargar el número de línea inicial en las posiciones 253 y 254, con el byte alto en la posición 254. Para un número de línea inicial menor que 255, tal como 1Ø o 1ØØ, sólo hay que cargar el valor correspondiente a la posición 253, aunque es más seguro cargar un Ø en la 254. Si se quiere empezar con un número de línea igual a 1ØØØ, hay que expresarlo en forma de dos bytes (la fig. 9.7 muestra cómo hacerlo). Hemos de cargar a continuación el valor del incremento (normalmente 1Ø) en la dirección 255. Si se ha ensamblado el programa en la posición $7FØØ, hay que utilizar SYS 32512 para activarlo. Su programa BASIC quedará renumerado en un abrir y cerrar de ojos. Desde luego, si se ensambló el programa en cualquier otra dirección, ésta será la dirección que habrá que utilizarse con SYS. La figura 9.8 muestra el listado del ensamblado del programa a través de MIKRO, de forma que se puedan ver los códigos numéricos involucrados. El código puede ensamblarse en cualquier posición de forma que, si quiere, puede transformarlo en un programa BASIC de carga.

Este es, pues, el final del camino. Hay poco de nuevo para aprender, excepto por lo que respecta a las sorpresas que pueda depararle el Commodore 64. A pesar de todo, a medida que vaya progresando, irá acumulando experiencia, hasta que se encuentre con que las sorpresas son más escasas y con que puede encontrar mucho más fácilmente formas de evitarlas. Cuando llegue este momento, puede considerarse un experto, un verdadero programador de código máquina.

Apéndice A.
Cómo están almacenados los números

El Commodore 64 utiliza cinco bytes de memoria para almacenar un número cualquiera. Este Apéndice describe la forma en que se almacenan estos números, aunque si las matemáticas no son su fuerte, tiene pocas posibilidades de asimilar lo que vamos a exponer.

Para empezar, los números con coma flotante (no los enteros) están almacenados en forma de *mantisa* y *exponente*. Este es un formato que se utiliza también para los números decimales. Así, por ejemplo, podemos escribir el número 216ØØØ como $2{,}16\times1Ø^{5}$, o bien el número Ø,ØØØ12 como $1{,}2\times1Ø^{-4}$. Cuando se utiliza esta forma de escribrir los números, a la potencia (de diez, en este caso) se la conoce con el nombre de *exponente* y al multiplicador (un número mayor que Ø y menor que 1) se le conoce con el nombre de *mantisa*. Los números binarios también pueden escribirse de esta forma, aunque con algunas diferencias. Para empezar, la mantisa de un número binario escrito de esta forma es siempre fraccionaria, aunque nunca se escribe el punto. Además, el exponente es una potencia de dos, en lugar de una potencia de diez. Podríamos, así pues, escribir el número binario 1Ø11ØØØØ como 1Ø11E1ØØ. Ello equivale a una mantisa de valor 1Ø11 (imagínela en la forma .1Ø11) y un exponente igual a 1ØØØ (2 elevado a 8 en decimal). No representa ventaja alguna escribir valores pequeños de esta forma, aunque sí es muy conveniente hacerlo para números grandes. Así por ejemplo, el número:

11Ø1Ø1ØØØØØØØØØØØØØØØØ

puede escribirse como 11Ø1Ø1E11ØØ (piense en ello como $.11Ø1Ø1\times2^{24}$).

Este es el esquema que se sigue en el Commodore 64 y otras máquinas que utilizan el BASIC de Microsoft. Como el dígito más significativo de la mantisa (la parte fraccionaria del número) es siempre 1, cuando se representa un número en este formato se convierte en Ø por razones de almacenamiento. Además, se suma 128 (decimal) al exponente antes de almacenarlo. Ello permite almacenar sin complicaciones números con exponentes negativos (hasta −128), ya que un exponente negativo se almacena, entonces, como un número de valor menor que 128 (decimal). El Commodore 64 utiliza cuatro bytes para almacenar la mantisa de un número y un byte para el exponente.

Como ejemplo sencillo, consideremos cómo se codificaría el número 2Ø (decimal). Convirtiéndolo en binario, se transforma en 1Ø1ØØ, lo que equivale a .1Ø1ØØØØ×2^5 en el formato de como flotante que hemos visto, utilizando ocho bits para la mantisa y con el exponente expresado en decimal. Se cambia, entonces, el MSB de la fracción por un Ø, de forma que el número almacenado será ØØ1ØØØØ. Esto se traduciría en un número con un 32 (decimal) almacenado en el byte más bajo de la mantisa. Por lo que respecta al exponente, 5 , en binario, es 1Ø1. Sumándole 128 obtenemos el número 1ØØØØ1Ø1, o sea, el valor 133 (que es 128 + 5).

Los enteros, por su parte, requieren sólo dos bytes para su almacenamiento. Deben tener un valor comprendido entre −32768 y −32767. Para convertir un entero al formato que utiliza el Commodore 64, haga lo siguiente:

1. Si el número es negativo, réstelo de 65536 y utilice el resultado.
2. Divida el número por 256 y tome la parte entera. Este es el byte más significativo.
3. Reste del número 256×(dígito más significativo). Este es el byte menos significativo.

Ejemplo: convertir 9438 al formato de almacenamiento entero.

El número es positivo, así que puede utilizarse directamente 9438/256 = 36.867187.

El MSB es, así pues, 36

36*256 = 9216 y 9438−9216 = 222

El byte bajo será, pues, 222.

Apéndice B.
Conversión hexadecimal-decimal

(a) Hexa a decimal

Para bytes aislados (dos dígitos hexa)

Multiplicar el dígito más significativo por 16 y sumar al resultado obtenido el dígito restante.

Por ejemplo :

$3D es 3*16+13 = 61 decimal

Para bytes dobles (direcciones)

Escribir el dígito menos significativo. Debajo de él escribir el valor del siguiente dígito multiplicado por 16. Debajo, escribir el siguiente dígito multiplicado por 256. Debajo, escribir el siguiente dígito multiplicado por 4Ø96. Hallar la suma total.

Por ejemplo :

$F3DB se convierte en la forma siguiente :

Escribir el primer dígito	11
El siguiente dígito por 16 es 13*16	2Ø8
El siguiente dígito por 256 es 3*256	768
El siguiente dígito por 4Ø96 es 15*4Ø96	6144Ø
Calculamos ahora la suma total que es	62427

(b) Decimal a hexa

Para bytes aislados (números menores que 256 decimal)

Dividir el número por 16. La parte entera del resultado corresponde al dígito más significativo. Multiplicando por 16 la parte fraccionaria del mismo resultado se obtiene el dígito menos significativo.

Por ejemplo :

Para convertir 155 a hexa:
153/16 = 9.6875, luego el dígito más significativo es 9

El dígito menos significativo es .6875*16, que es 11. Ello corresponde a B en hexa, de forma que el número será $9B.

Para números de dos bytes (comprendidos entre 256 y 65535 decimal)

Dividir, como antes, el número por 16. Anotar la parte entera y escribir la parte fraccionaria multiplicada por 16 como un dígito hexa. Repetir el proceso con la parte entera hasta que quede un solo dígito hexa.

Por ejemplo :

Para convertir 23815 a hexa :
23815/16 = 1488.4375. La fracción .4375 por 16 da 7. Este es el dígito menos significativo.
Tomando la parte entera anterior, 1488/16 = 93.ØØ. Como no hay parte fraccionaria, el siguiente dígito es Ø.
93/16 = 5.8125. La fracción .8125 multiplicada por 16 da 13, que es D en hexa. Este es el tercer dígito hexa. Como la parte entera es menor que 16 (es un 5), éste será el dígito más significativo. El número completo será, pues, $5DØ7.

Apéndice C.
El conjunto de instrucciones

El repertorio de instrucciones del 65Ø2 es relativamente reducido, aunque habrá varias instrucciones en la lista que sigue que nunca tendrá que utilizar a menos que progrese mucho en el campo de la programación. Una descripción completa del efecto de cada instrucción ocuparía demasiado espacio, de forma que indicaremos el efecto asociado a cada instrucción mediante abreviaciones. Para una descripción completa, consulte alguno de los libros dedicados a la programación del 65Ø2. En general, M representará un byte en una posición de memoria. A los registros se les denominará con las letras habituales de referencia. Una flecha indicará el lugar donde se almacena el resultado de una acción. Así, por ejemplo, A+M+C→A+C significa que se suma el byte de memoria (direccionado por la instrucción) al byte del acumulador y al bit de acarreo C, colocándose el resultado en el acumulador A, con el acarreo en C.

Los códigos de instrucción se muestran en columnas, por orden según el método de direccionamiento empleado. Estos métodos son el Inmediato, el de Página cero, el de Página cero X-indexado, el Absoluto, el Absoluto X-indexado, el Absoluto Y-indexado, el X-indirecto, el Y-indirecto y el Relativo al PC. Pocas instrucciones utilizan el direccionamiento Implícito. El método de direccionamiento Implícito significa que no es necesario ningún direccionamiento en especial. En el formato de lenguaje ensamblador DIR identificará una dirección de dos bytes, mientras que dir representará el único byte de dirección (el más bajo) que se utiliza en el direccionamiento de página cero. Después se utilizará para representar el desplazamiento en el direccionamiento relativo al PC. Byte representará el byte que sigue a un código de direccionamiento inmediato. Se ha utilizado S para identificar el registro de estado del procesador. Los indicadores se representan por C, N y V. Todos los códigos de instrucción aparecen en hexa.

Formato de lenguaje ensamblador	*Método de direccionamiento*	*Código*	*Acción*
ADC # Byte	Inmediato	69	A + M + C → A + C
ADC dir	Página cero	65	
ADC dir, X	Página cero, X	75	
ADC DIR	Absoluto	6D	
ADC DIR, X	Absoluto, X	7D	
ADC DIR, Y	Absoluto, Y	79	
ADC (dir, X)	Indirecto, X	61	
ADC (dir), Y	Indirecto, Y	71	
AND # Byte	Inmediato	29	A Y M → A
AND dir	Página cero	25	
AND dir, X	Página cero, X	35	
AND DIR	Absoluto	2D	
AND DIR, X	Absoluto, X	3D	
AND DIR, Y	Absoluto, Y	39	
AND (dir, X)	Indirecto, X	21	
AND (dir), Y	Indirecto, Y	31	
ASL A	Implícito	ØA	Decalaje a la izquierda
ASL dir	Página cero	Ø6	
ASL dir, X	Página cero, X	16	
ASL DIR	Absoluto	ØE	
ASL DIR, X	Absoluto, X	1E	
BCC Desp	Relativo	9Ø	Saltar si C = Ø
BCS Desp	Relativo	BØ	Saltar si C = 1
BEQ Desp	Relativo	FØ	Saltar si Z = 1
BIT dir	Página cero	24	O con M
BIT DIR	Absoluto	2C	Comprobar N y V
BMI Desp	Relativo	3Ø	Saltar si N = 1
BNE Desp	Relativo	DØ	Saltar si Z = Ø
BPL Desp	Relativo	1Ø	Saltar si N = Ø
BRK	Implícito	ØØ	Interrumpir el programa
BVC Desp	Relativo	5Ø	Saltar si V = Ø
BVS Desp	Relativo	7Ø	Saltar si V = 1
CLC	Implícito	18	Poner a cero el acarreo

Formato de lenguaje ensamblador	*Método de direccionamiento*	*Código*	*Acción*
CLD	Implícito	D8	Anular modo decimal
CLI	Implícito	58	Anular la inhibición de interrupciones
CLV	Implícito	B8	Poner a Ø el bit V
CMP # Byte	Inmediato	C9	A-M, poner indicadores
CMP dir	Página cero	C5	
CMP dir, X	Página cero, X	D5	
CMP DIR	Absoluto	CD	
CMP DIR, X	Absoluto, X	DD	
CMP DIR, Y	Absoluto, Y	D9	
CMP (dir, X)	Indirecto, X	C1	
CMP (dir), Y	Indirecto, Y	D1	
CPX # Byte	Inmediato	EØ	X−M, poner indicadores
CPX dir	Página cero	E4	
CPX DIR	Absoluto	EC	
CPY # Byte	Inmediato	CØ	Y−M, poner indicadores
CPY dir	Página cero	C4	
CPY DIR	Absoluto	CC	
DEC dir	Página cero	C6	M−1 → M
DEC dir, X	Página cero, X	D6	
DEC DIR	Absoluto	CE	
DEC DIR, X	Absoluto, X	DE	
DEX	Implícito	CA	X−1 → X
DEY	Implícito	88	Y−1 → Y
EOR # Byte	Inmediato	49	A Oexc M → A
EOR dir	Página cero	45	
EOR dir, X	Página cero, X	55	
EOR DIR	Absoluto	4D	
EOR DIR, X	Absoluto, X	5D	
EOR DIR, Y	Absoluto, Y	59	
EOR (dir, X)	Indirecto, X	41	
EOR (dir), Y	Indirecto, Y	51	
INC dir	Página cero	E6	M + 1 → M
INC dir, X	Página cero, X	F6	
INC DIR	Absoluto	EE	

Formato de lenguaje ensamblador	*Método de direccionamiento*	*Código*	*Acción*
INC DIR, X	Absoluto, X	FE	
INX	Implícito	E8	X + 1 → X
INY	Implícito	C8	Y + 1 → Y
JMP DIR	Absoluto	4C	Saltar a DIR
JMP (DIR)	Indirecto	6C	Saltar a la dirección almacenada
JSR DIR	Absoluto	2Ø	Saltar a subrutina
LDA # Byte	Inmediato	A9	M → A
LDA dir	Página cero	A5	
LDA dir, X	Página cero, X	B5	
LDA DIR	Absoluto	AD	
LDA DIR, X	Absoluto, X	BD	
LDA DIR, Y	Absoluto, Y	B9	
LDA (DIR, X)	Indirecto, X	A1	
LDA (DIR), Y	Indirecto, Y	B1	
LDX # Byte	Inmediato	A2	M → X
LDX dir	Página cero	A6	
LDX dir, Y	Página cero, Y	B6	
LDX DIR	Absoluto	AE	
LDX DIR, Y	Absoluto, Y	BE	
LDY # Byte	Inmediato	AØ	M → Y
LDY dir	Página cero	A4	
LDY dir, X	Página cero, X	B4	
LDY DIR	Absoluto	AC	
LDY DIR, X	Absoluto, X	BC	
LSR A	Implícito	4A	Decalaje a la derecha
LSR dir	Página cero	46	
LSR dir, X	Página cero, X	56	
LSR DIR	Absoluto	4E	
LSR DIR, X	Absoluto, X	5E	
NOP	Implícito	EA	Operación nula
ORA # Byte	Inmediato	Ø9	A O M → A
ORA dir	Página cero	Ø5	
ORA dir, X	Página cero, X	15	
ORA DIR	Absoluto	ØD	

Formato de lenguaje ensamblador	*Método de direccionamiento*	*Código*	*Acción*
ORA DIR, X	Absoluto, X	1D	
ORA DIR, Y	Absoluto, Y	19	
ORA (dir, X)	Indirecto, X	ø1	
ORA (dir), Y	Indirecto, Y	11	
PHA	Implícito	48	Guardar A en la pila
PHP	Implícito	ø8	Guardar S en la pila
PLA	Implícito	68	Sacar A de la pila
PLP	Implícito	28	Sacar S de la pila
ROL A	Implícito	2A	Decalar a la izquierda
ROL dir	Página cero	26	
ROL dir, X	Página cero, X	36	
ROL DIR	Absoluto	2E	
ROL DIR, X	Absoluto, X	3E	
ROR A	Implícito	6A	Decalar a la derecha
ROR dir	Página cero	66	
ROR dir, X	Página cero, X	76	
ROR DIR	Absoluto	6E	
ROR DIR, X	Absoluto, X	7E	
RTI	Implícito	4ø	Volver de interrupción
RTS	Implícito	6ø	Volver de subrutina
SBC # Byte	Inmediato	E9	A−M−C* → M
SBC dir	Página cero	E5	C* es lo que nos llevamos
SBC dir, X	Página cero, X	F5	
SBC DIR	Absoluto	ED	
SBC DIR, X	Absoluto, X	FD	
SBC DIR, Y	Absoluto, Y	F9	
SBC (dir, X)	Indirecto, X	E1	
SBC (dir), Y	Indirecto, Y	F1	
SEC	Implícito	38	Poner a 1 el bit de acarreo
SED	Implícito	F8	Establecer modo decimal
SEI	Implícito	78	Establecer la inhibición de interrupciones

Formato de lenguaje ensamblador	*Método de direccionamiento*	*Código*	*Acción*
STA dir	Página cero	85	A → M
STA dir, X	Página cero, X	95	
STA DIR	Absoluto	8D	
STA DIR, X	Absoluto, X	9D	
STA DIR, Y	Absoluto, Y	99	
STA (dir, X)	Indirecto, X	81	
STA (dir), Y	Indirecto, Y	91	
STX dir	Página cero	86	X → M
STX dir, Y	Página cero, Y	96	
STX DIR	Absoluto	8E	
STY dir	Página cero	84	Y → M
STY dir, X	Página cero, X	94	
STY DIR	Absoluto	8C	
TAX	Implícito	AA	A → X
TAY	Implícito	A8	A → Y
TYA	Implícito	98	Y → A
TSX	Implícito	BA	S → X
TXA	Implícito	8A	X → A
TXS	Implícito	9A	X → S

Apéndice D.
Métodos de direccionamiento del 65Ø2

Cada uno de los métodos de direccionamiento tiene como efecto el utilizar un byte de la memoria. La dirección donde está almacenado este byte se denomina Dirección Efectiva (DE). La finalidad de cualquier método de direccionamiento es la de utilizar una determinada dirección efectiva.

Direccionamiento Inmediato: La DE es la dirección que sigue al byte de instrucción.

Direccionamiento de Página cero: Sólo aparece en la instrucción el byte bajo de la DE. El byte alto es siempre ØØ, de ahí el nombre de Página cero. Así, por ejemplo, utilizar el direccionamiento de página cero con el byte FB resultaría en el empleo de la dirección $ØØFB.

Direccionamiento Absoluto: Acompañan a la instrucción dos bytes que forman una dirección completa. Así, por ejemplo, LDA $563F significa que hay que cargar el acumulador con el contenido de la dirección $563F.

Direccionamiento Indexado: Hay un número almacenado en el registro de índice (X o Y). La dirección efectiva consiste en este número sumado a la dirección que se especifique en la instrucción. Así, por ejemplo, LDA 25,X significa cargar el acumulador con el contenido de la dirección que resulta de sumar el valor del registro de índice X a 25.

Direccionamiento Implícito: La dirección a utilizar está implícita en la instrucción, no siendo necesaria ninguna referencia a una dirección de memoria en concreto. Así, por ejemplo, INX significa incrementar el registro X, no siendo necesaria ninguna DE.

Direccionamiento Indirecto: Existen dos modalidades del mismo, utilizando ambas direcciones de la página cero. El Indirecto X-Indexado suma el contenido del registro X al valor de la dirección de la página cero y coge el byte de la dirección resultante. Este constituye el byte bajo de la dirección efectiva. El byte alto se coge de la dirección siguiente de la página cero. El Indirecto Y-Indexado coge el byte almacenado en la dirección de la página cero y le suma el contenido del registro Y. El resultado es el byte bajo de la dirección efectiva, cogiéndose, tal como antes, el byte alto de la siguiente posición de la página cero.

Apéndice E.
Algunas direcciones de ROM y RAM

Ahora que está disponible el desensamblado de la ROM del Commodore 64, todas las direcciones de la misma están al alcance de cualquiera. A continuación se muestran algunas de las direcciones más interesantes. Junto con ellas hemos incluido algunas posiciones útiles de RAM junto con unas breves notas sobre lo que se almacena en ellas.

Direcciones de ROM

Rutina de entrada	$F157
Rutina de salida	$F1CA
Lectura del teclado	$E112
Escribir en pantalla	$E1ØC
Inicio del BASIC	$AØØØ
Mensaje READY	$A376
NEW	$A644
Instrucción de ejecución	$A7E4
Circuito Integrado de video	$DØØØ a $DØ21
Port 1	$DCØØ a $DCØF
Port 2	$DDØØ a $DDØF

Direcciones de RAM

$2B, $2C	Inicio del BASIC
$2D, $2E	Inicio de la VLT
$2F, $3Ø	Inicio de los vectores
$31, $32	Final de los vectores
$33, $34	Inicio de almacenamiento de cadenas
$37, $38	Límite de memoria
$9A	Código del periférico de salida
$C5	Última tecla pulsada
$CC	Autorización de cursor (Ø = si)
$D1, $D2	Línea de la pantalla en uso
$D3	Posición del cursor

$D6	Número de la línea del cursor
$F3, $F4	Puntero de la memoria asociada al color
$2Ø0	Buffer de entrada del BASIC
$Ø277	Buffer del teclado
$Ø28A	Repetición del teclado ($8Ø hace que todas las teclas puedan repetirse)
$Ø28F, $Ø29Ø	Dirección de la rutina de decodificación del teclado

Indice analítico