Tecnicas de Programación Avanzada con AMSTRAD

KEITH HOOK

ra-ma

Técnicas de Programación Avanzada para el AMSTRAD CPC464

Keith Hook

Título de la obra original

ADVANCED PROGRAMMING TECHNIQUES
ON THE AMSTRAD CPC464

Publicado en Gran Bretaña en 1985
por Phoenix Publishing Associates Ltd.
14, Vernon Road, Bushey, Herts, Wd2 2JI

ISBN 0-9465-7632-7

Edición en Español

TECNICAS DE PROGRAMACION AVANZADA
PARA EL AMSTRAD CPC464

Editado por RAMA
Chinquinquira, 28 (COCUY)
28033 MADRID
ESPAÑA

I. S. B. N.: 84 - 86381 - 06 - 1
Depósito Legal: M. 35772 - 1985

Consultas referentes al libro
RAMA, Chinquinquira, 28. Teléfono: 764.50.95 - 28033 Madrid

Traducción y Composición: CONORG, S.A.

CONTENIDO

INTRODUCCION

El Amstrad CPC464 ha sido diseñado para permitir al programador tener acceso a un excelente y potente lenguaje de programación: **Locomotive Basic.**

Este libro lo he escrito con la esperanza de que sea útil a los programadores que quieren conseguir el máximo provecho de su ordenador. En esencia, este libro viene a ser un recetario de ideas desarrolladas en torno al **interpretador** Basic del Amstrad.

Este libro no está diseñado para enseñar al novato todos los aspectos de la programación en Basic, sino más bien para suplementar un nivel supuesto de conocimiento espigado del manual de instrucciones.

Si tienes como mínimo una somera familiaridad con Basic, este libro te permitirá progresar a través de numerosos ejemplos hasta un nivel avanzado de programación, mostrándote cómo incorporar sencillas rutinas en código máquina a tus programas en Basic. No te preocupes si no comprendes todas las instrucciones que se describen **-penetra hasta el extremo más profundo.** La mejor manera de aprender es practicando los ejemplos; la comprensión vendrá posteriormente.

En las páginas de este libro encontrarás una cantidad considerable de material de consulta y referencia ante el que puedes decidir si estudiarlo ahora o en una fecha ulterior; y no necesitas graduarte en informática para aplicar los nuevos conocimientos que vayas adquiriendo.

El material de este libro te mostrará cómo atrapar las potentes rutinas que descansan en lo profundo de la **'mollera'** del Amstrad. La mayoría de dichas rutinas puede ser aprovechada desde el Basic, y te permitirá producir programas que son una mixtura de código máquina y Basic. Esta manera híbrida de programar el Amstrad te permitirá realizar tareas desde Basic que pudieran parecerte imposibles **-Cargar un programa en código máquina en una Tabla y ejecutarlo; o poner bajo control del CPC464 hasta 8 figuras animadas.** Este libro te mostrará cómo lograrlo.

No hay nada difícil ni misterioso sobre los métodos sugeridos. Desde luego, hay reglas, y algunos de vosotros todavía no habéis intentado dar el paso de gigante para pasar más allá del Basic, pero con un poco de aplicación todas las cosas son posibles. Lo reitero, lánzate a fondo a las entrañas del CPC464.

Aunque no es la intención de este libro enseñarte **lenguaje de Ensamblaje,** sí te proporcionará una buena perspectiva, y espero que un buen empujón que te anime a investigar un mundo completamente nuevo en la programación.

Con el propósito de aprovechar las ventajas del **sistema operativo** del ordenador, debes poseer un buen conocimiento de cómo opera la máquina, y por lo tanto puede que algunos de vosotros piensen que los capítulos primeros son un poco elementales. Sin embargo, te insto a que los leas; nunca se sabe, pero puede que incluso descubras que no lo conocías en absoluto.

Capítulo Uno

Lo Básico del BASIC

No voy a insultar tu inteligencia comenzando este libro con largas explicaciones sobre cómo insertar los enchufes, o cómo conectar el monitor -si te has comprado el ordenador, no tengo ninguna duda de que ya lo estás usando. Sin embargo, para que realmente eches tus garras sobre todo lo que vamos a tratar, necesitas comprender **Basic**, sus frases, y cómo trabaja el **Sistema Operativo.**

Los **programas en Basic** se componen de **Líneas Basic,** y las **Líneas Basic** constan de **Frases Basic.** El CPC464 te permite programar con **Líneas Basic multi-frase.** Una **frase** (una 'sentencia') es una oración sintáctica **imperativa** que manda al ordenador llevar a cabo alguna acción específica. El Amstrad contiene en su mollera (y circuitalmente es una **Memoria de únicamente lectura**) previamente escrita por su fabricante, un programa **interpretador Basic** que escucha, interpreta y ejecuta cada una de las frases Basic que le comunicas y cursa al microprocesador las señales necesarias en código máquina para que lleve a cabo las acciones requeridas. El microprocesador **Z80** usado por el Amstrad CPC464 puede entender y ejecutar esas señales en código máquina en millonésimas de segundo, lo que significa que tus mandatos dados en Basic son cumplimentados muy rápidamente.

Sin embargo, Basic nunca puede alcanzar las velocidades de ejecución que se logran con **programas en código máquina.** Por otro lado, la confección y la **depuración** (el quitarle las pifias) a un programa en código máquina no es fácil. Incluso el programa más corto puede necesitar horas para estar completamente depurado, y además el error más liviano puede hacer que tu programa discurra por cualquier parte y hacia ningún sitio, y te deje simplemente pasmado ante una pantalla en blanco. Para depurar eficazmente rutinas en **lenguaje máquina** es necesario algunas veces que el propio programador "juegue a ser computadora" tratando de ejecutar su código, y comprobando cada acción con pluma y papel hasta que logre localizar dónde está el error.

Con la mayoría de los programas, podemos adherirnos al feliz término medio. El grueso del programa puede escribirse en **Basic** y puede apelarse a **rutinas en código máquina** encargadas de aquellas gestiones y funciones que necesitan velocidad extra. Por ejemplo, la rapidez de una rutina para **ordenamiento de datos** puede verse incrementada por un factor de 1000, si se escribe la rutina en código máquina. Este último es sólo un ejemplo; los gráficos, la animación de figuras y el sonido también pueden verse enormemente mejorados por este método de programación. También puedes llevar a cabo muchas más **tareas** escribiendo el programa que las realiza como **párrafos** en código máquina para ser incorporados y citados desde tu programa en Basic -el Amstrad aparentemente no admite las populares figuras animadas- que no tienen nada de espíritus, pero para cuando llegues al final de este libro verás cómo estás usándolos con tu ordenador.

Revisión del Locomotive Basic

Puedes inscribir las frases imperativas del Basic desde el teclado hasta las entrañas de la máquina, de dos maneras: **Modo Programa** y **Modo Directo.**

En el **Modo Directo** tus frases imperativas son interpretadas como auténticos **comandos** por lo que inmediatamente que digas **vale** (pulsando la tecla de 'adentro' marcada **ENTER**) será inmediatamente obedecida.

```
CLS: LET X=3: LET Y=X+1: PRINT Y (ENTER)
```

Si has tecleado la línea anterior correctamente, y has dicho que 'adentro', en la pantalla se estará ahora exponiendo la respuesta **4.** Ejemplos de otras palabras clave que primordialmente se usan como **comandos** son:

RUN, NEW, AUTO

El **Modo de Programa** difiere en que las frases imperativas que inscribes por teclado, llevan como prefijo un **número de línea.** El convenio es que con eso no es un comando a ejecutar inmediatamente, sino una **instrucción** que el ordenador debe 'aprender' en su memoria; por lo que al decir que **vale** (pulsando la tecla de 'adentro' marcada **ENTER**), la introducirá en el lugar correcto del programa que le estás 'enseñando', sin que en ese momento cumplimente las instrucciones recibidas.

```
AUTO    10,10     [Comando directo]

10 CLS
20 LET X=3: LET Y=X+1
30 PRINT Y
```

Si ahora mandas que **ejecute** el programa que ha aprendido (y tiene registrado en su memoria), para lo cual basta cursarle el comando **RUN** verás que el resultado final obtenido es exactamente el mismo que cuando diste esas frases imperativas en el **Modo Directo,** como auténticos **comandos.**

Por tanto, cuando tecleaste el comando directo **RUN,** automáticamente el interpretador Basic recorrió una a una las líneas de programa efectuando cada una de las acciones mandadas a medida que examinaba e interpretaba cada línea. El Basic siempre ejecuta un programa línea a línea, de acuerdo con el orden ascendente de los números de línea, excepto cuando se encuentra una instrucción que le manda **vaya a,** o **vaya y venga** al acabar a, la línea cuyo número se le indica, sea superior o sea inferior al que está tratando en ese momento. Estos saltos (GOTO) y desvíos (GOSUB) permiten cambiar el **curso** seguido por el programa. El interpretador Basic continuará cumplimentando una a una las instrucciones, o bien hasta que se le acaben, o bien hasta que encuentre una instrucción que le mande que **fin**alice, que tendrá por palabra clave **END.**

```
 10  CLS
 20  INPUT " DIME TU NOMBRE ";N$
 30  PRINT "HOLA ";N$
 40  INPUT "¿TE GUSTA TU AMSTRAD?";RE$
 50  IF RE$="SI" THEN GO TO 80
 60  PRINT "¡OH CARI\O! LO SIENTO MUCHO, ";N$
 70  GOSUB 120: GOTO 100
 80  PRINT "¡OH CARI\O! ME ALEGRO MUCHO, ";N$
 90  PRINT "ADIOS ";N$: GOSUB 120
100  CLS
110  END
119  REM BUCLE DE RETARDO PARA PERCATARSE MEJOR
120  FOR I=1 TO 500
130  NEXT
140  RETURN
```

Este breve programa ilustra el **curso** o 'flujo' del programa dentro de un ordenador. **LOS PROGRAMAS FLUYEN DESDE EL COMIENZO HASTA EL FINAL SIGUIENDO LAS LINEAS DE PROGRAMA NUMERADAS EN ORDEN ASCENDENTE.** Es posible alterar el curso del programa como hemos hecho en la línea 50 comprobando si la respuesta (RE$) a la pregunta efectuada fue **"SI"**. Si la respuesta fue afirmativa, el curso del programa cambió su dirección dando un salto hasta la línea 80, y evitando la ejecución de las líneas 60 y 70.

En el programa anterior hemos usado líneas numeradas en múltiplos de 10. ¿Por qué 10? Muy bien, podríamos haber usado cualquier numeración de líneas, v.g. **1, 2, 3, 4** o bien **1, 3, 5, 7**; sin embargo, es aconsejable mantener la numeración de las líneas sencilla y no con números consecutivos. Si usamos múltiplos de 10 siempre podremos agregar posteriormente líneas donde sean necesarias. Después de añadir esas líneas a tu programa, los números de línea no estarán según incrementos uniformes; pero en tu Amstrad siempre puedes dejar la numeración de líneas totalmente 'pulcra' si cursas el comando que le obliga a que **renumere**, que tiene por clave **RENUM**. Inténtalo por ti mismo. Teclea el comando:

RENUM 100,10,5

y dile que ya **vale** pulsando la tecla **ENTER**.

Habrás observado que cuando ahora pides que **liste** el programa, los números de línea han quedado cambiados a 100, 105, 110, ..., y se incrementan de cinco en cinco, y el programa comienza a partir de la línea numerada 100.

Usarás dentro de tus programas dos modalidades de datos: **CONSTANTES** y **VARIABLES**.

Los datos **constantes** son valores prefijados que no sufren ningún cambio durante el curso de la ejecución. El valor del número **Pi** es una constante universal. En un programa hay dos clases de constantes **numéricas: enteras** o cantidades sin parte fraccionaria como 1, 12, 100, 1111, y **reales** que son cantidades con parte fraccionaria, tal como 3.333, 1.00789, etc. (recuerda que la separación entre la parte entera y la parte fraccionaria de un número se efectúa mediante el signo **punto** y no mediante la **coma** habitual en los países hispanoparlantes.

Los datos **variables** son exactamente lo que su nombre implica: los que **varían,** o pueden **variar,** durante el curso del programa. Las variables son sencillamente **nombres** que el programa utiliza para señalar determinados sitios o **lugares** de la memoria donde almacena constantes numerales o literales. El Amstrad te permite especificar el nombre de una variable mediante una serie de 40 caracteres como máximo, todos ellos significativos. Ejemplos de nombres de variables son **X, Y, A5, BA, JAULA, NOMBRE.** Locomotive Basic también permite al programador especificar la clase o **índole** de la variable al comienzo del programa, designándola como **entera, real** o **literal,** añadiendo el signo **%, !** ó **$** al final del nombre.

Las variables y constantes enteras contienen números en la banda de -32768 a +32767, y pueden designarse las variables enteras mencionándolas en la instrucción de clave **DEFINT,** o colocando como sufijo el signo % como en **X%.**

Los datos numerales reales, constantes o variables, son los que tienen parte fraccionaria y pueden contener números con valor absoluto comprendido en la banda $1.35E^{+38}$ ($1.35*10^{+38}$) y $6.35E^{-20}$ ($6.35*10^{-20}$). Esta banda de números es lo suficientemente grande como para satisfacer al más apasionado manipulador de números ($1.7E^{+38}$ descendiendo hasta $2.9E^{-39}$). Las variables reales se designan como tales usando el comando de clave **DEFREAL** o colocándoles como sufijo el signo **!.**

Los números reales ocupan **cinco octetos** consecutivos en la memoria. Esta observación es importante. Lo **preceptuado para casos de omisión para todas las variables** es que se consideren como **REALES.** Deberías tomar la costumbre de definir tus variables al principio de cada programa, y a no ser que vayas a usar variables que exijan un mayor grado de precisión, las definirás normalmente como enteras mediante el comando de clave **DEFINT.** E.g.

```
DEFINT A,C,E-N,P-Z
```

El ejemplo anterior hará que se **traten** como enteras todas las variables cuyo nombre tengan como letra inicial las mencionadas en el comando; lo que quiere decir todas exceptuando las que comiencen por **B, D** u **O.** Recuerda que tu ordenador opera muchísimo más rápido sobre datos enteros, y que por tanto tus programas se **ejecutarán** a mayor velocidad.

Otra clase de variables, de índole totalmente diferente a la numeral, son las **variables literales.** En esencia son sencillamente cadenas o retahílas de caracteres alfabéticos, numéricos o símbolos especiales, que no tienen ningún significado para el ordenador aunque deben pertenecer a su **repertorio** de símbolos; pero sí lo tienen para el usuario. En inglés se usa la palabra **string** para destacar que son sartas, cuerdas o reatas de caracteres adosados uno tras otro. El carácter de literal se da a una variable colocando como sufijo de su nombre el signo dólar ($). Así por ejemplo:

```
LET E$="YO":LETN$="1234"&LET:X$=CHR$(13)
```

El Amstrad es capaz de operar sobre datos literales de muchas maneras y su capacidad para segregar y alterar trozos de literales constituye una de las más potentes facilidades del ordenador.

Otra clase de variable usada dentro de Basic es la denominada **variable subindicada.** Puede ser literal o numeral, y en realidad es un elemento de un conjunto homogéneo de variables que tienen un nombre común y genérico. Es una variable muy potente y muy importante que te permitirá mantener listas ordenadas o depositar y recuperar datos de forma completamente **discrecional:** basta dar la **referencia** del elemento según el orden que ocupa en el conjunto. A medida que te hagas más eficiente como programador te darás cuenta que estás usando esta clase de variable más y más veces.

Un ejemplo típico de una **variable subindicada** sería **X(1),** siendo **X** el nombre común de las variables del conjunto, e indicando ese subíndice la variable que tiene asignado como número de orden dentro del conjunto el **'uno'.** Otro ejemplo sería **X(6),** que suele leerse **'X afectada de 6',** o bien **X sub 6.** Todas las variables subindicadas que pertenecen al mismo conjunto homogéneo, y que por tanto llevan el mismo nombre común, v.g. **A(1),A(2)....A(20)** constituyen una **tabla.** En inglés se usa el término 'array' para destacar que son **ringlas,** o sea, conjuntos homogéneos y colocados según una **regla** dada.

Las tablas han de ser declaradas al interpretador Basic para que ocupe el espacio en memoria necesario para ellas, y de acuerdo con las **DIMensiones** que se den a la tabla (estrictamente hablando no sería necesario cuando el subíndice mayor no sobrepasa de 10, pero es una buena práctica en programación especificarlas al principio del programa) -e.g. **DIM X(12)** haría que se ocupara el espacio para una tabla de una sola dimensión, cuyo subíndice iría desde el 0 hasta el 12 (lo que supone trece elementos en la ristra).

Este asunto de las tablas es demasiado importante como para despacharlo en tan pocas palabras, así que lo comentaremos completamente en un capítulo ulterior.

El locomotive Basic incluye un elenco completo de **funciones** incorporadas en el lenguaje. Estas funciones son en realidad **nombres** que al ser citados efectúan unas ciertas operaciones y **entregan** un resultado convenido. Así por ejemplo, la función de nombre clave **RND** permite obtener el siguiente número de una secuencia interna generada al **azar**, que conjuntamente con el comando de clave **RANDOM** para que aleatorice el comienzo de dicha secuencia, constituyen un elemento esencial para juegos y simulaciones matemáticas. Otras funciones incorporadas en el lenguaje Basic son las de clave **FIX** para obtener un número entero mediante truncamiento de la parte fraccionaria de uno dado; **CINT** para convertir a entero cualquier dato numeral dado; **INT** que redondea un dato dado a un número entero aproximándolo hacia menos infinito; **SIN** que da el seno de un ángulo dado; **CREAL** que convierte a número real un dato dado, etc.

El tratamiento de los errores producido durante el curso de un programa es objetivo de instrucciones como las de clave **ERROR, ON ERROR GOTO** y **RESUME** (para que **reanude** la ejecución después de resarcirse del error), y como funciones tales como **ERR** que indica la clase de error y **ERL** que da la línea donde se produjo el error. Usados adecuadamente estos comandos y funciones, te permiten comprobar tus rutinas para **caza de errores** durante la ejecución, y establecer tus rutinas para resarcimiento de errores, así como simular la aparición de un error mediante el comando de clave **ERROR**; mientras que el comando **ON ERROR GOTO...** establece el número de línea del programa al que ha de irse cuando se detecte el error.

El interpretador Basic incluye una sección con un programa Editador de Línea, que entra en acción cuando se le reclama mediante el comando **EDIT.** Este programa editador permite hacer que se **revise** un determinado número de línea, y coloca el cursor dentro de dicha línea, con lo que pueden suprimirse, corregirse o insertarse caracteres en ella. La existencia de la tecla **COPY** y el uso de las teclas de cursor conjuntamente con la tecla de turno marcada **SHIFT,** hace que pueda aprovecharse para utilizar un auténtico **editador de plana** desplazando el cursor de copiar hasta la parte de la pantalla donde se desea tomar un texto y que automáticamente quede insertado en la línea en que se está revisando, a partir de la posición señalada por el cursor de edición principal.

Los comandos de clave **TRON** y **TROFF** permiten, respectivamente, que se ponga o quite un mecanismo **rastreador** del curso seguido por un programa en ejecución. Cuando está puesto ese rastreador, aparece en pantalla cada número de línea a medida que es cumplimentada, permitiéndote así examinar el curso seguido por el programa. Puedes comprobar esto por ti mismo dando el comando directo **TRON**, y pulsando **ENTER** para que pase a ejecutarlo. Ahora ejecuta el programa número 2, contesta a su pregunta con "Sí"; luego vuelve a ejecutarlo de nuevo pero esta vez responde con "No".

¿Observaste cómo cambió el flujo del programa cuando lo ejecutaste por segunda vez? El comando **TRON** es una ayuda a la **depuración de programas** (quitarle las 'pifias') muy valiosa, sobre todo cuando tienes un programa largo en memoria y no funciona de acuerdo con lo que esperabas.

El comando **STOP** hace que **pare** la ejecución del programa en curso, con lo que puedes aprovecharlo para insertar un **punto de ruptura** y hacer que se detenga la ejecución en ese punto. Cuando está parado, solemos aprovechar para examinar el contenido de las variables, tablas, etc.; y luego con el comando de clave **CONT** haremos que **siga** la ejecución del programa en curso desde el mismo punto en que se detuvo (siempre y cuando en el interregno no hayamos alterado las instrucciones del programa).

El comando de clave **CLEAR** hace que el interpretador Basic **anule** todas las variables, poniendo las numerales a cero y las literales a **vacío** (""). También se libera el espacio ocupado por las tablas, y se olvidan todos los bucles que estuvieran pendientes de ejecución.

El comando de clave **NEW** hace que el interpretador Basic **quite** el programa que hubiera registrado en la memoria en ese momento, lo que suele hacerse cuando se desea comenzar a confeccionar un **nuevo** programa en Basic.

Capítulo Dos

Designando Lugares en Memoria

En este capítulo vamos a comentar el asunto de cómo reservar e identificar **sitios** en la memoria, y de cómo expresar cantidades según el sistema de numeración **decimal** (en base 10), **binaria** (en base 2) y **decimohexal** (en base 16), y cómo los números correspondientes son reflejados en el ordenador Amstrad. Por favor, **no te saltes** esta sección. El tema es mucho más fácil de comprender de lo que pudieras pensar, y el poco tiempo empleado en aprender estas reglas fundamentales te reditará el mil por uno en tus ulteriores intentos de programación.

NUMEROS BINARIOS

Cada lugar unitario de la memoria, y si la asocias a un 'panal' hablaríamos de cada **celdilla** de la memoria, tiene una **dirección** -y dicha dirección puede ser cualquier entero desde 0 hasta 65535. En las celdillas de memoria se depositan los datos y los comandos que son usados por el ordenador para ejecutar tu programa. Sin embargo, incluso aunque podemos hablar de números en base 10 (y muchos prefieren decir sistema **denario,** en lugar de decimal para evitar las confusiones con las fracciones), el ordenador sólo puede operar con **números binarios,** por lo que todos los números denarios y hexadecimales son convertidos -internamente- en números binarios antes de ser registrados en la memoria del ordenador.

Estos números binarios son representaciones de lo que se conoce como **instrucciones en código máquina.** Cada una de estas instrucciones es realmente un **grupo de bits** dispuestos en determinado orden, que representan uno de los dos tipos posibles de señales eléctricas: la de **puesto,** en inglés **ON** y asignada al valor binario **1,** y la de **quitado,** en inglés **OFF** y designado por el valor binario **0.** El ordenador reconoce las diversas combinaciones de señales **puesto/quitado** y actúa de acuerdo con cada combinación diferente.

Si comprendes cómo se combinan e interactúan los números representados en notación decimal, decimohexal y binaria, serás capaz de escribir programas de mayor eficacia y de efectuar operaciones con los datos usando posibilidades del interpretador Basic que no están claramente mencionadas en el manual. Por ejemplo, la comprobación de si un determinado bit está puesto o quitado, o la de expresar tus datos de una forma más compacta, por mencionar únicamente dos facilidades posibles.

Cada celdilla de memoria puede almacenar ocho bits. Un bit es uno de los dos símbolos admisibles en un sistema de numeración binaria, y además ocho bits constituyen un **octeto.** En inglés se les suele llamar **byte** porque les recuerda el 'mordisco' que el microprocesador da cada vez que efectúa una operación con la memoria.

```
Una CELDILLA de memoria = 8 BITS = 1 BYTE = 1 OCTETO
```

Al contar cantidades según el sistema de numeración en base 10 usamos los **10 dígitos** simbolizados del 0 al 9. Para continuar contando después del 9 debemos volver al 0 y **llevarnos una** a la columna de las decenas, y así sucesivamente. Sabemos que **152 = 1 * 100 + 5 * 10 + 2 * 1,** y podemos así escribir en su 'forma desarrollada' expresándolo como:

1*10^2+5*10^1+2*10^0

y volviendo a nuestros días escolares, podemos recordar que cualquier número elevado a la potencia de 0 (10^0) es igual a la unidad.

El sistema de numeración binaria, al ser en base 2, utiliza únicamente **dos cifras** o símbolos, que son habitualmente el 0 y el 1. Cuando contamos cantidades según el sistema binario aplicamos las mismas reglas que en el sistema de base 10, por lo que ahora al llegar contando al 2, debemos volver al 0 y **llevarnos una unidad** a la columna siguiente a la izquierda. El número binario 0111 puede escribirse en forma desarrollada como:

```
0 *2^3 + 1 *2^2 + 1 *2^1 +1 *2^0 = 7  'en base 10
```

En binario, cada **posición** tiene un valor relativo igual a la correspondiente **potencia de 2,** en lugar de ser una potencia de 10 como en el sistema denario. Así:

```
1011 = 1 *2^3 + 0 *2^2 + 1 *2^1 + 1 *2^0
     =      8 +      0 +      2 +      1  = 11 en base 10
```

Los **ocho bits** de un octeto están numerados del 0 al 7 yendo **desde la derecha hasta la izquierda.** Si nosotros examinamos ahora la notación utilizada por un octeto, veremos lo fácil que es calcular el número equivalente en base 10, usando esta **notación posicional:**

```
Notación posicional ==)  2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
                         ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
Número binario =======) ¦ 1 ¦ 1 ¦ 0 ¦ 0 ¦ 0 ¦ 1 ¦ 1 ¦ 1 ¦
                         ¦   ¦   ¦   ¦   ¦   ¦   ¦   ¦
Número ordinal de bit )   7   6   5   4   3   2   1   0
```

Expresando en base 10 el anterior número binario sería:

```
1*2^7 +1*2^6 +0*2^5 +0*2^4 +0*2^3 +1*2^2 +1*2^1 +1*2^0 =
= 1 *2*2*2*2*2*2*2 + 1 *2*2*2*2*2*2 + 1 *2*2 +1 *2 =
=   128 +  64   +   4 +   2  +   0 = 199  'en base 10
```

Puedes ver bastante claramente a partir del ejemplo anterior, que a medida que pasas a la siguiente posición de bit **más a la izquierda,** el **peso** -o valor que corresponde a ese bit según la posición- se dobla:

```
Valor posicional (PESO) ==)  128 64 32 16  8  4  2  1
Número ordinal de BIT ====)    7  6  5  4  3  2  1  0
```

Si todos los valores posicionales se suman, veremos que un octeto puede contener un valor máximo de 255 expresado en base 10 o de 11111111 expresado según el sistema binario.

La notación posicional puede ampliarse cuando se trata de números en binario que ocupan dos o más octetos. Cuando se usan dos octetos (y tendremos un doble octeto con 16 bits, por algunos llamado dexeto, por aquello del decimohexeto) entonces pueden representarse números que corresponden a cantidades mayores (0 a 65535).

Tabla 2.1

TABLA DE VALORES PARA NUMEROS BINARIOS QUE OCUPAN DOS OCTETOS

DECIMAL	BINARY															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
254	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0
256	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1024	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
24576	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
24577	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1
32000	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0
32767	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Observarás en la tabla 2.1 que solamente usamos números de **15 bits** lo que nos lleva a un valor máximo de 32767, para verse reflejado según **2 octetos,** porque en el que ocupa la posición decimoquinta sólo empleamos el valor binario cero. Este método de almacenar números en dos octetos es el denominado según **formato entero.** Con este formato no está permitido usar números enteros positivos mayores de 32767. Compruébalo por ti mismo tecleando el siguiente programa.

```
10  CLS
20  DEFINT I 'TRATARA A I COMO VARIABLE ENTERA
30  INPUT "TECLEA CUALQUIER NUMERO ";
40  PRINT I
50  GOTO 30
```

Ahora, con el programa **trabajando** intenta responder a la pregunta con diversos valores de I. Ensaya por ejemplo **IN**poniendo a I un valor mayor de 32767. ¿Qué ha sucedido? El ordenador respondió con un mensaje de **O/V ERROR** indicando que se ha producido un **rebase** del valor permitido, ya que se ha **sobrepasado** lo máximo permitido para enteros positivos.

Tabla 2.2

POTENCIAS DE 2

2^0 = 1	2^8 = 256
2^1 = 2	2^9 = 512
2^2 = 4	2^10 = 1024
2^3 = 8	2^11 = 2048
2^4 = 16	2^12 = 4096
2^5 = 32	2^13 = 8192
2^6 = 64	2^14 = 16384
2^7 =128	2^15 = 32768

Terminología:

Estuvo una vez de moda comparar los **bits** con bombillas eléctricas o con **conmutadores** de dos estados, en que el **1** se representaba por el conmutador estando **puesto** o activo, y el **0** representado por el conmutador estando **quitado** o desactivado. Esta terminología todavía se usa hoy: si un bit tiene el valor binario 1, decimos que está puesto o "alzado", y si tiene el valor 0 decimos que está quitado o "bajado".

Vamos a tratar ahora con los tres sistemas de numeración: **Binario,** en base 2; **Decimal o denario,** en base 10; y **Decimohexal** (aunque también se llama hexadecimal y hexidecimal) en base 16. Para distinguir entre estos tres sistemas de numeración, usaremos los siguientes subíndices: binario$_B$, decimal$_D$, y decimohexal$_H$. A partir de ahora, si ves **32456$_D$** sabrás que el número está expresado en base 10, y en cambio **AC00$_H$** será un número en hexadecimal.

ARITMETICA BINARIA

La adición y la multiplicación binaria son operaciones muy fáciles de comprender. Dado que en el sistema binario sólo hay **dos cifras** (símbolos 0 y 1) para tratar estas operaciones sólo tienes **cuatro permutaciones** para aprender las reglas de la suma y otras cuatro para las reglas del producto.

```
0 + 0 = 0 : 0 + 1 = 1 : 1 + 0 = 1 : 1 + 1 = 10
0 x 0 = 0 : 0 x 1 = 0 : 1 x 0 = 0 : 1 x 1 = 1
```

De lo anterior se puede inferir que la única regla a recordar es que **1+1 = 0 y nos llevamos 1.**

Adición

```
Línea de llevada...    1 1 1 0
                  15 =   1 1 1 1
                   6 =   0 1 1 0
                  ==   =========
Resultado ..... 21 = 1 0 1 0 1
```

Método: 0 + 1 = 1, y el uno pasa al resultado. 1 + 1 = 10, ponemos 0 en la línea de resultado y un 1 en la línea de llevada. Ahora 1 + 1 = 10 + llevado 1 = 11, ponemos 1 en la línea de resultado y otro 1 en la línea de llevada. Ahora 1 + 0 = 1 + llevado = 10; ponemos 0 en el resultado y 1 en la de llevada. Ahora pasamos lo llevado a la línea de resultado.

Tabla de adición binaria

```
==================
: + : 0       1  :
------------------
: 0 : 0       1  :
:   :            :
: 1 : 1      10  :
==================
```

Multiplicación

La multiplicación sólo consiste en añadir un valor a él mismo un determinado número de veces:
4 x 3 = 4 + 4 + 4 = 12; 6 x 5 = 6 + 6 + 6 + 6 + 6 = 30

Echemos una ojeada a la multiplicación binaria y veremos que surgen un par de hechos interesantes.

Tabla de multiplicación binaria

x	0	1
0	0	0
1	0	1

```
Multiplicando ..... 1010 = 10 'en base 10

Multiplicador ..... 0110 =  6 'en base 10
                    -----
                     0000   ... producto parcial

                    1010    ... producto parcial

                   1010     ... producto parcial

                  0000      ... producto parcial
                  -------

Producto final .....  0110100 = 60 'en base 10
```

Hecho 1: Siempre que aparece un 1 en el multiplicador, se copia el multiplicando en la línea de resultado parcial. Si aparece un 0 en el multiplicador, el multiplicando no se copia.

Hecho 2: En cada paso el resultado parcial se desplaza un lugar hacia la izquierda, incluso aunque el multiplicador contenga un cero en esa posición.

Este hecho 2 nos proporciona un método rápido de multiplicar números binarios por potencias de 2 (2, 4, 8, 16, etc.).

```
2 * 2 = 4  ==) 2 = 00000010
y desplazando iZquierda  un lugar   = 00000100 = 4

2 * 4 = 8  ==) 2 = 00000010
y desplazando iZquierda dos lugares = 00001000 = 8
```

División binaria

La división binaria es la propia simplicidad como puedes apreciar con una simple mirada a cómo un número se dividirá por otro. La división también puede efectuarse por sustracciones sucesivas hasta que se encuentre un resto negativo. **La división es la inversa de la multiplicación,** de manera que una forma fácil de dividir por múltiplos de 2 es usar lo recíproco del **Hecho 2** anterior, i.e. **desplazar un lugar a la derecha.**

```
8 / 4 = 2 .. 4 = 2 x 2   asi que desplacemos 2
                         posiciones a la drcha

                                  0000 1000 = 8

  desplaza drcha primera vez  = 0000 0100 = 4
  desplaza drcha segunda vez  = 0000 0010 = 2
```

OPERACIONES LOGICALES

Una peculiaridad a menudo despreciada del ordenador es su capacidad para efectuar operaciones denominadas **lógicas.** El repertorio de instrucciones del Amstrad Basic es muy potente y está provisto de un conjunto completo de operaciones logicales a las que puede recurrirse desde Basic.

En el **Algebra Booleana** puede haber únicamente dos respuestas o **estados:** el estado de **CIERTO** con valor numeral distinto de cero, y el estado de **FALSO** cuyo valor numeral es el cero. Si el resultado de una operación logical es cierta, el ordenador coloca algunos bits del octeto al estado 1. Si el resultado es falso los 8 bits tienen el valor 0.

Las operaciones logicales no son difíciles de comprender; después de todo usamos la partícula conectiva **Y SI** en instrucciones condicionales bastante frecuentemente dentro de nuestros programas Basic:

```
¡ SI es X=3 Y SI es Y=1
     ENTONCES VAYA a la 30   !
     ENDEMAS  VAYA a la 100
```

Siempre que tú programas al ordenador con una instrucción condicional **IF...THEN...ELSE...,** estás usando realmente una premisa o expresión logical.

En el ejemplo anterior, si X = 3 **AND** Y = 1, la condición resulta ser cierta y el curso del programa saltará a la línea 30. Cualquier otro valor en X o en Y dará como resultado el valor lógico falso y por ende que el curso del programa salte a la línea numerada con 100.

Expresiones logicales en la forma: **IF G THEN GOTO 100** también están permitidas en el Amstrad Basic. Y siempre que el valor de G sea distinto de 0, se considerará como cierto y el programa continuará su curso en la línea 100. Expresiones como **IF G AND 16** también están permitidas: constituyen una aprovechable forma taquigráfica de ahorrar un montón de tecleado.

El uso de las partículas conectivas **AND (Y SI...), OR (O SI...)** y **NOT** no está restringido a las expresiones de relación como las mencionadas anteriormente. También pueden usarse como auténticos **operadores** sobre números según su equivalente binario, para tratamiento y comparaciones individuales de los bits que los componen.

- **NOT** _____ (Neg -ación)

La operación **NOT** forma el **complemento** del número sobre el que se aplica invirtiendo el valor de cada bit que lo compone.

```
NOT 12   =  -13
NOT -2   =   1
NOT  0   =  -1

0000 0000   Negación de 0 ====)    1111 1111 = -1
```

- **AND** _____ (Y L I -ación)

La operación Y-lógico es usada para segregar determinados bits de un octeto; aplicando correlativamente la operación a cada uno de los bits de los dos operandos:

```
12 YLIado con  4  da  4

                        0000 1100 = 12
                  AND   0000 0100 =  4
                        ---------
                        0000 0100 =  4

25 YLIado con  12  da  8

                        0001 1001 = 25
                   AND  0000 1100 = 12
                        ---------
                        0000 1000 =  8

 4 YLIado con  2 da  0

                        0000 0100 =  4
                  AND   0000 0010 =  2
                        ---------
                        0000 0000 =  0
```

Puedes comprobar el resultado anterior por ti mismo pidiendo al interpretador Basic que exponga el resultado en la forma:

```
PRINT 4 AND 2          <ENTER>
```

- **OR** _____ (O L I -ación)

La operación O-lógico inclusivo se usa frecuentemente para hacer que determinados bits tomen el valor 1 sin afectar al valor de los otros bits; al efectuar correlativamente la operación sobre cada bit de los operandos:

```
 4 OLIado con  2  da  6

                      0000 0100 =  4
                OR    0000 0010 =  2
                      ---------
                      0000 0110 =  6

-1 OLIado con  -2 da  -1

                      1111 1111 = -1
                OR    1111 1110 = -2
                      ---------
                      1111 1111 = -1
```

- **XOR** _____ (O L E -ación)

La operación O-lógico excluviso se usa a menudo para detectar coincidencias de bits:

```
 3 OLEado con  3  da   0
 4 OLEado con  2  da   6

                      0000 0000 =  4
               XOR    0000 0010 =  2
                      ---------
                      0000 0110 =  6
```

Tabla 2.3
OPERACIONES LOGICAS SOBRE BITS INDIVIDUALES

NOT	0	1
	1	0

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

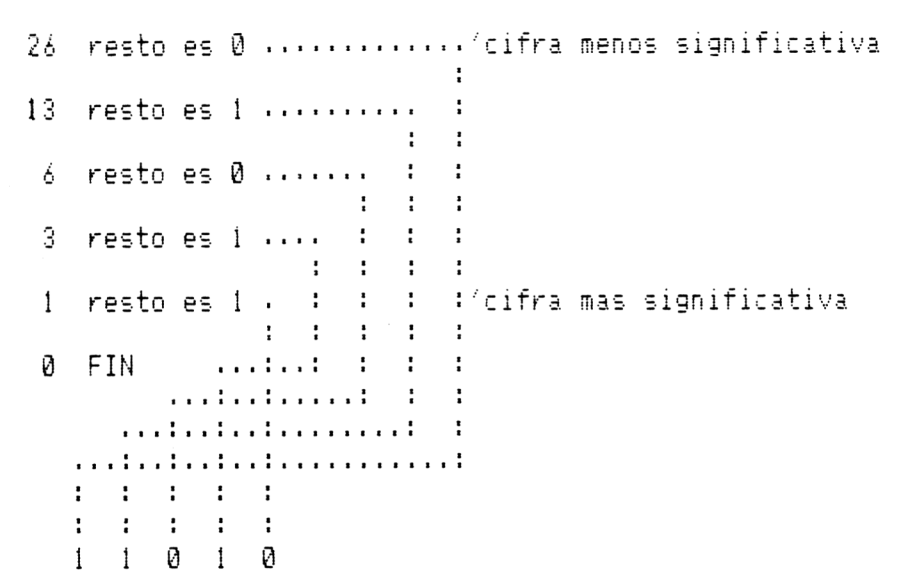
Tabla 2.4
OPERACIONES LOGICALES

```
        NOT TRUE  = FALSE

        NOT FALSE = TRUE
TRUE  AND TRUE  = TRUE

TRUE  AND FALSE = FALSE

FALSE AND TRUE  = FALSE

FALSE ANS FALSE = FALSE

TRUE  OR  TRUE  = TRUE

TRUE  OR  FALSE = TRUE

FALSE OR  TRUE  = TRUE

FALSE OR  FALSE = FALSE

TRUE  XOR TRUE  = FALSE

TRUE  XOR FALSE = TRUE

FALSE XOR TRUE  = TRUE

FALSE OR FALSE  = FALSE
```

DE DECIMAL A BINARIO

Es algunas veces necesario convertir un número expresado en notación decimal al equivalente expresado en notación binaria. El proceso consiste simplemente en dividir reiteradamente por 2 y almacenar el resto en la columna correspondiente.

Convertir 26 a binario



La división se continúa hasta que se obtiene un 0 como cociente final. El número binario es el conseguido colocando el resto final en la columna más a la izquierda seguido de los otros restos tomados de abajo a arriba. ¿Es fácil?

Si no estás familiarizado con el sistema binario, el Programa Uno te ayudará a ver cómo los números en binario están relacionados con los números en el sistema decimal.

LISTADO UNO:

```
 1 MODE 1:PAPER 0:CLS  'CONVIERTE A BINARIO
10 WINDOW #1,8,30,3,3
20 WINDOW #2,8,34,5,5
30 WINDOW #3,8,34,7,7
```

```
 40 WINDOW #4,8,34,8,8
 50 LOCATE #1,1,1
 60 CLS #1:INPUT#1,"INGRESA TU NUMERO";A
 70 LOCATE #2,1,1
 80 PEN #2,2:INK #2,5:PRINT #2,"TU NUMERO= ";A
 90 LOCATE #3,1,1:PEN #3,5:INK 5,24
100 PRINT #3,"BIT 0 1 2 3 4 5 6 7"
110 LOCATE #4,6,1:PEN #4,6
120 FOR I=0 TO 7
130 IF A AND 2^I THEN PRINT #4,"1 ";
                  ELSE PRINT #4,"0 ";
140 NEXT
150 GOTO 50
```

NUMEROS DECIMOHEXALES

Una vez has aprendido los rudimentos del sistema binario, los números del sistema **decimohexal** no son difíciles. Obviamente al ser un sistema de base 16, son necesarias **dieciséis** cifras, que formarán la base del sistema de numeración. La notación convencional emplea para estos **dexitos** (y así llaman algunos a estos 16 símbolos) utiliza los 10 dígitos del 0 al 9 y las primeras seis letras del alfabeto: A B C D E F. En esta notación, el número decimal 13_D se convierte en D_H.

Los números decimohexales nos permiten manipular números binarios de una manera más cómoda. Los números binarios largos se parecen mucho unos a otros y apreciar la diferencia entre 11110101 00101110 y 1110101 00101010 en una lista de números binarios no es fácil. Considera las direcciones adoptadas inicialmente para la pantalla del Amstrad. En el sistema decimal esa dirección se expresa como 49152_D y en el sistema binario como 1100 0000 0000 0000. En el sistema decimohexal esa dirección se expresará como $\mathbf{C000_H}$ que es más breve y cómoda.

Dos cifras decimohexales (y se abrevia normalmente por hex) son el equivalente en binario de un octeto (8 bits); y una cifra hex es por tanto un **cuarteto** (4 bits). (Estos cuartetos se denominan frecuentemente en inglés un 'nibble', que a muchos les recuerda el 'pezón' y dos de esos 'nibbles' constituyen un byte que les recuerda al 'mordisco' como ya hemos mencionado). Dividiendo pues los números binarios en grupos de 4, es relativamente sencillo pasarlo a notación decimohexal.

Tabla 2.5

COMPARACION DE CIFRAS DE CADA SISTEMA

BINARIO	DECIMAL	DE-XAL
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Vemos ahora que, por ejemplo, el número binario
1110 1100 0100 1101 = 60493_D,
observando la tabla anterior puede convertirse a su valor en decimohexal buscando para cada grupo de 4 bits la equivalente cifra hex.

```
  E    C      4     D

1110  1100  0100  1101

= 16^3 x 14 + 16^2 x 12 + 16^1 x 4 + 13 = 60493
```

Adición en el sistema decimohexal

La suma con los números hex es muy similar a la suma de números decimales, excepto que contamos hasta 16 antes de **llevarnos una** a la siguiente columna a la izquierda:

```
100   .....Linea de llevada..... 1110
AC01                              DACB
1FC3                              2BB0
----                              ----
CBC4  ......Resultado.............. 1067B
```

Para ayudarte con la suma decimohexal, echa una mirada a la tabla 2.6. Si quieres sumar C_H a B_H busca la C en la columna de la izquierda y pasa por la columna de cifras hasta que encuentres la intersección con la B en la columna superior; y verás que el resultado es 17_H. Si realmente estuvieras sumando estas dos cifras hex, pondrías el 7 en la línea de respuesta y el 1 en la línea de llevada (o acarreo como se suele llamar).

Tabla 2.6
ADICION DECIMOHEXAL

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Vamos ahora a considerar cómo el Amstrad almacena realmente los números enteros en su memoria.

Todos los ordenadores basados en el microprocesador Z80 almacenan sus números enteros en **dos octetos** consecutivos, que se denominan el octeto menos significativo (que en inglés se dice **Least Significant Byte: LSB**), y el octeto más significativo (que en inglés es **Most Significant Byte: MSB**). En términos simples, cada número entero tiene como equivalente dos octetos: el VOZ, que es el valor del octeto izquierdo, y el VOD que es el valor del octeto derecho; lo que ocurre que en memoria se guardan en sentido inverso. La razón de esto no es obvia, pero puedes aceptar mi palabra (o no aceptarla hasta que lo comprendas) que usando este método con números muy largos da como resultado un ahorro en memoria de más del 60%.

Si queremos conservar el número decimal 60493 en la celdilla de memoria 65450, y en la **siguiente**, debemos primero calcular los dos valores de los octetos correspondientes: **MSB y LSB.** Una vez que estos valores del octeto izquierdo y derecho hayan sido calculados, observa que pondremos el octeto derecho en la celdilla inferior -la 65450- y el octeto izquierdo en la celdilla superior -la 65451. Para calcular el valor del octeto izquierdo y el valor del octeto derecho, usamos la división entera, fácil de aprender, siguiente:

```
ValorOctetoIzquierdo 'VOZ' = INT (Numero/256)
ValorOctetoDerecho   'VOD' = Numero - 256 * VOZ
```

Usando la fórmula anterior podemos ahora convertir nuestro número decimal 60493 a su equivalente en dos octetos:

```
'VOZ' = COCiente de (60493/256)  = 236  =EC
'VOD' = RESiduo  de (60493/256)  =  77  =4D

y puedes usar los operadores Basic  '\' y 'MOD'
```

Si estuvieramos trabajando con Basic, puedes aprovechar las operaciones de **división entera** para el cálculo, y luego hacer que **meta** esos valores en la memoria usando los comandos:

```
POKE 65450,VOD
POKE 65451,VOZ
```

Es importante comprender esta regla del octeto izquierdo en la celdilla superior y el octeto derecho en la celdilla inferior. Posteriormente, cuando comencemos a meter instrucciones en código máquina en diversas celdillas de la memoria, sabremos cómo depositar los valores y recuperarlos correctamente.

Cómo se deposita un número de dos octetos:

```
Posicion de BIT        7   6   5   4   3   2   1   0

Dirección  65450    ! 0 ! 1 ! 0 ! 0 ! 1 ! 1 ! 0 ! 1 ! 'vod
Dirección  65451    ! 1 ! 1 ! 1 ! 0 ! 1 ! 1 ! 0 ! 0 ! 'voz
```

Anteriormente, dijimos que las variables numerales **reales** usaban **cinco octetos** consecutivos de la memoria; esta clase de datos se registra en la memoria de manera parecida a como lo hacemos con los datos enteros, pero obviamente, el sistema empleado ocupa cinco octetos: el valor del número normalizado en cuatro octetos y el exponente en el quinto octeto.

Registro en memoria de los números reales

```
                        ---------------
Dirección de            !  Oct 4 iZq  !   BYTE 0
memoria más baja        ---------------
                        !  Oct 3 iZq  !   BYTE 1
                        ---------------
                        !  Oct 2 iZq  !   BYTE 2
                        ---------------
                        !  Oct 1 iZq  !   BYTE 3
                        ---------------
Dirección de            !  EXPONENTE  !   BYTE 4
memoria más alta        ---------------
```

La porción del **exponente** se expresa como una potencia de 2, permitiendo la banda de 10^{-39} a 10^{+38}.

VALORES NEGATIVOS

Hasta este momento hemos estado comentando enteros positivos de 8 y 16 bits (octetos y dexetos). Son los que se denominan **enteros sin-signo.** Sin embargo, hay un montón de cosas más sobre el cómputo de otras clases de datos numéricos: ¿qué pasa con los valores negativos? Un ordenador no tendría mucha ventaja si no pudiera efectuar las operaciones matemáticas normales. Claramente, para trabajar sobre valores positivos y negativos, necesitamos alguna manera de decirle qué entero es positivo y cuál es negativo. Para hacer eso tenemos que convenir una serie de reglas que son las usadas en la mayoría de los ordenadores digitales existentes hoy en el mercado.

En un **entero de 8 bits con signo,** el bit situado más a la izquierda (el bit más significativo), que tiene asignado como ordinal el 7, es el considerado como **bit de signo.** Si ese bit 7 está alzado -valor 1- el número es negativo. Si el bit 7 está bajado -valor 0- el número es un número positivo, o como mínimo igual a cero.

```
0100 1111  = 79          1100 1111  = -79
```

Siempre que usamos enteros sin signo, el número máximo que puede representarse con un octeto es de 255_D, o bien $1111\ 1111_B$ o bien FF_H; y el más pequeño es el cero. El número de mayor valor que puede representarse con un entero de 8 bits **con signo** es el 127_D o bien $0111\ 1111_B$, o bien $7F_H$; y el valor máximo negativo es el -127_D, o bien $1111\ 1111_B$.

Usando este convenio para representar los valores de 8 bits, que se denomina **binario natural o con signo,** surge pronto un problema: no podemos usar las reglas normales para sumar dos números con signo. Estudia el siguiente ejemplo:

```
    0100 1111   =  79
+   1000 1101   = -13
   ----------
    1101 1100   = -92
```

La respuesta (-92_D) obviamente es incorrecta. ¡Debiera ser $+66_D$!

Representación por Complemento a Doses

Para solventar este problema, algún brillante cerebro en el pasado lejano, descubrió la regla del **complemento a doses.** No hay nada misterioso sobre esta representación de números en el sistema binario y es bastante sencilla de calcular:

Regla a: el complemento a doses de un número binario positivo de 8 bits es el propio número.

Regla b: el complemento a doses de un número binario negativo de 8 bits se calcula obteniendo su **complemento a unos** del valor absoluto del número, y añadiéndole una unidad.

Calcular el complemento a unos de un número, es simplemente **invertirlo,** i.e. cambiar todos sus unos a ceros y todos sus ceros a unos.

Así, calcular el complemento a doses de 16_D será:

```
16 = 0001 0000

     1110 1111  ............Complemento a Unos
             1  ............Suma Uno
    ----------
     1111 0000  ............Complemento a doses.
```

Encontrar el complemento a doses de 102_D será:

```
102 = 0110 0110

      1001 1001  ............Complemento a Unos
              1  ............Suma Uno
     ----------
      1001 1010  ............Complemento a doses.
```

Si usamos el operador logical **NOT** podemos variar la regla b para que diga:

1) NOT número
2) Agregar 1

Vamos a usar ahora nuestro ejemplo original, y sumar $+79_D$ a -13_D, para ver cómo este método funciona correctamente.

```
    Negado de  13  = -14 .  -14  = 1111 0010

1111 0010 + 1 = 1111 0011   Complemento a DOSes

        0100 1111 = +79
      + 1111 0011 = -13 'en complemento a DOSes
      -----------------
        0100 0011 = +66
```

¡y +66 es la respuesta correcta!

Añadir (-3) a (-2)

```
                          3 = 0000 0011    2  = 0000 0010

Complemento a UNOs ..... 1111 1100        1111 1101
 Suma Uno          .....          1              1
                          ---------      ---------
Complemento a DOSes..... 1111 1101        1111 1110

    -2 = 1111 1110 'en complemento a DOSes

+   -3 = 1111 1101 'en complemento a DOSes
------   ---------
     ?   1111 1011  y nos llevamos UNA al final
```

El valor que nos llevamos al sumar los bits 7 se ignora completamente, y nuestro resultado será 1111 1011 que debiera ser el complemento a doses del entero negativo -5. Podemos calcular fácilmente trabajando al revés.

```
    Negado de 1111 1011 =    0000 0100
                                    1 .......Suma Uno
                             ---------
                             0000 0101  = 5
```

Por lo tanto, 1111 1011 es el resultado correcto de sumar dos números negativos.

La banda de números enteros que puede representarse con 8 bits según la notación de complemento a doses es desde $+127_D$ bajando hasta -128_D (véase tabla 2.7).

Tabla 2.7 REPRESENTACION POR COMPLEMENTO A DOSES

+	2's complement	-	2's complement
127	0111 1111	-128	1000 0000
126	0111 1110	-127	1000 0001
64	0100 0000	-64	1100 0000
63	0011 1111	-63	1100 0001
32	0010 0000	-32	1110 0000
31	0001 1111	-31	1110 0001
16	0001 0000	-16	1111 0000
15	0000 1111	-15	1111 0001
14	0000 1110	-14	1111 0010
9	0000 1001	-9	1111 0111
8	0000 1000	-8	1111 1000
7	0000 0111	-7	1111 1001
6	0000 0110	-6	1111 1010
3	0000 0011	-3	1111 1101
2	0000 0010	-2	1111 1110
1	0000 0001	-1	1111 1111
0	0000 0000		

Aunque hemos comentado sólo los enteros de 8 bits, todo lo que hemos dicho también se aplica a los enteros de 16 bits. La banda de números para enteros sin signo de 16 bits es de 0 a 65535_D; y para enteros de 16 bits según representación de complemento a doses es de -32768_D hasta $+32767_D$. Debemos constatar ahora por qué dejamos sin usar el bit 15 en la tabla 2.1: fue para permitir colocar el bit de signo.

Hemos tratado bastantes fundamentos en este capítulo, así que no te preocupes si no has comprendido completamente lo que hemos comentado. Puedes también volver a consultar este capítulo en una etapa posterior, y un montón de lo tratado automáticamente lo comprenderás mejor a medida que progreses a lo largo del libro.

Capítulo Tres

La Cadena de Cosas

En el capítulo anterior hemos comentado la utilización y la representación de variables numéricas. Bien, en este capítulo vamos a hablar de otro tipo de variables: las **VARIABLES LITERALES.**

El Basic Amstrad no solamente puede tratar con valores numéricos, sino que también es un maestro en el juego con letras y textos. Los **literales** son meras **cadenas de caracteres,** (en inglés se usa la palabra 'string'=cuerda, retahíla, etc.). Dichas series de caracteres pueden estar formadas por cifras, letras, signos y cualquier carácter del repertorio admitido en el odenador.

Normalmente, esos literales están formados por caracteres **ASCII.** Las siglas ASCII son un acrónimo de **A**mericano **S**tandard **C**ódigo para **I**nformación **I**ntercambio. Es una norma estándar que ha sido adoptada por la mayoría de la industria informática, para hacer exactamente lo que dice: permitir a los microordenadores intercambiar caracteres simbólicos según un formato estándar.

Los caracteres que siguen el convenio ASCII se **codifican** utilizando **siete bits** (numerados del 0 al 6) y ya hemos visto en el capítulo anterior que con 7 bits se puede representar cualquier número en la banda de 0 a 127_D o bien $0000\ 0000_B$ a $0111\ 1111_B$; o bien 00_H a $7F_H$. Eso no sugiere que de los 256 valores posibles de un octeto, podemos elegir 128 valores concretos para formar el **repertorio de caracteres estándar ASCII.** Los otros caracteres o símbolos gráficos están admitidos en el Amstrad Basic dando al bit 7 del octeto el valor 1, lo que nos permite otros 128 caracteres correspondientes a los valores en la banda 128_D a 255_D, o bien $1000\ 0000_B$ a $1111\ 1111_B$, o bien 80_H a FF_H.

Tabla 3.1

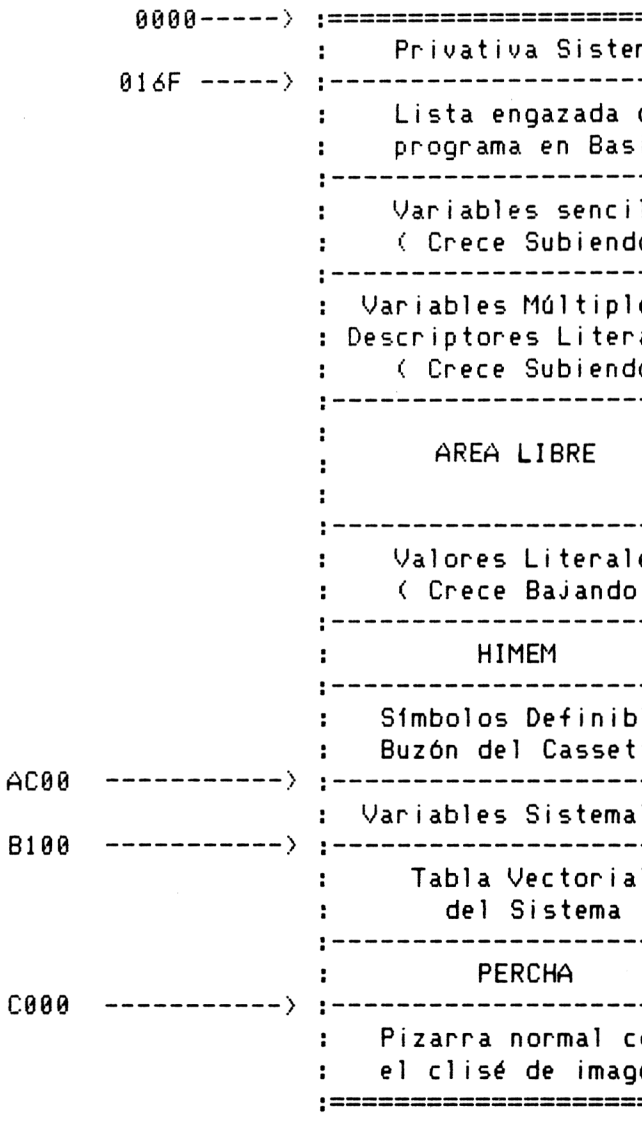
LOS CODIGOS Y CARACTERES ASCII

CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER
0		43	+	86	V
1		44	,	87	W
2	Cursor off	45	-	88	X
3	Cursor on	46	.	89	Y
4		47	/	90	Z
5		48	0	91	[
6		49	1	92	\
7	Sou Bell	50	2	93	]
8	Ba space/Erase	51	3	94	
9	C sor ==>	52	4	95	
10	ursor Down	53	5	96	`
11	Cursor Up	54	6	97	a
12	Home Cursor\Cls	55	7	98	b
13	Carriage Return	56	8	99	c
14		57	9	100	d
15		58	:	101	e
16		59	;	102	f
17		60	<	103	g
18	Erase to E.O.L	61	=	104	h
19		62	>	105	i
20	Erase to E.O.W	63	?	106	j
21		64	@	107	k
22		65	A	108	l
23		66	B	109	m
24		67	C	110	n
25		68	D	111	o
26		69	E	112	p
27	ESC	70	F	113	q
28		71	G	114	r
29		72	H	115	s
30	Home	73	I	116	t
31		74	J	117	u
32	Space	75	K	118	v
33	!	76	L	119	w
34	"	77	M	120	x
35	#	78	N	121	y
36	$	79	O	122	z
37	%	80	P	123	{
38	&	81	Q	124	|
39	'	82	R	125	}
40	(	83	S	126	
41	)	84	T	127	
42	*	85	U		

Aunque hemos dicho que los caracteres con códigos ASCII del 0 al 127 forman los códigos estándar, eso no es estrictamente verdad. Con certeza, los códigos **para los caracteres visivos** que van desde el 32 (espacio) hasta el 122 (la z) sí que son estándar; pero los códigos desde el 0 al 31, que se conocen como **caracteres de control,** y no tienen un símbolo visivo representativo, varían de ordenador a ordenador. Véase la tabla 3.1.

Diagrama 3.1

DISTRIBUCION DE LA MEMORIA



Las variables literales conservan sus valores en la propia línea del programa Basic en que son mencionadas, hasta tanto ese valor no haya cambiado de alguna manera después de su primera definición; en cuyo caso son registrados en el **área de trabajo para valores literales,** al mismo tiempo que se actualiza el **puntero** que señala la dirección de comienzo de cada literal, y que es conservado por el Basic en el bloque de **descriptores de literal,** en la manera mostrada en el diagrama 3.2. La dirección preceptuada para casos de omisión, como dirección inicial del programa en Basic del usuario es la $016F_H$, a partir de la cual quedan registradas en forma de lista 'engazada' las líneas de programa que introduces por el teclado. El número de octetos utilizado por un literal es función directa de la longitud del literal, es decir, del número de caracteres que forman esa serie; y de la longitud del propio nombre del literal: cada carácter en un literal utiliza un octeto al igual que ocurre para cada carácter en el nombre del literal.

Diagrama 3.2

Descriptor de Literal

```
:------:----:----------:---------------:---------------:
:Nombre: 02 : Longitud : Dirección VOZ : Dirección VOD :
:------:----:----------:---------------:---------------:

Por tanto   A$="TEST" se registrará así:

:------:----:----------:---------------:---------------:
:  C1   : 02 :    04    : Dirección VOZ : Dirección VOD :
:------:----:----------:---------------:---------------:
   :
   : porque C1 = 41(código de 'A') + 80(bit 7 alzado)
```

No se puede operar sobre las variables literales de la misma manera que las variables numerales: después de todo, quién quisiera usar expresiones del tipo Y$="TIO":X$="SOBRINO":IF Y$/X$="PRIMO" THEN GOTO..... Las instrucciones de esta clase no tienen ningún significado cuando se aplican a datos literales.

Los literales pueden, sin embargo, usar el signo **más** (+) dos o más literales pueden ser **empalmados** uno tras otro usando el signo +, y esta operación se suele denominar **CONCATENACION.**

```
10 LET W$= "Hola. " : LET X$="Soy tu nue"
20 LET Y$= "vo ": LET Z$="AMSTRAD "
30 LET PR$= W$+X$+Y$+Z$
40 PRINT PR$          ====>Hola. Soy tu nuevo AMSTRAD
```

También se pueden comparar datos literales usando los mismos operadores de relación que con las variables numerales, teniendo en cuenta el orden convenido para los caracteres que lo forman. Los operadores son pues:

```
<   Menor que
>   Mayor que
<>  Distinto de
<=  Menor o Igual que
>=  Mayor o Igual que
```

Cómo se comparan los datos literales unos con otros, está en dependencia de los **códigos ASCII** de los caracteres. Si ensayas con el siguiente programa podrás apreciar de lo que se trata.

```
10 CLS
20 LET X$=INKEY$
30 IF X$="" THEN GOTO 20
40 LOCATE 1,1:PRINT X$;" = ";ASC(X$);" CODIGO ASCII"
50 GOTO 20
```

- **ASC**

La función de nombre **ASC** entrega el valor ASCII correspondiente al carácter **inicial** del literal que percibe como argumento. E.g.

```
10 LET X$= "A":PRINT ASC(X$)        EXpondrá  65

10 PRINT ASC("z")                   EXpondrá  122
```

Al comparar dos o más literales, el interpretador Basic compara realmente el código ASCII de cada uno de los caracteres que forman los literales. Por ejemplo: "AB" será mayor que "AA", y "AC" será menor que "AX". Observa también que "a" es mayor que "A". (Véase la tabla 3.1).

Cuando dos literales son de distinta longitud y coinciden en los primeros caracteres, obviamente el más corto es **menor que** el más largo: "JUAN" "JUANITO".

Anteriormente, vimos cómo podían adosarse literales uno tras otro usando el signo + para la concatenación; sin embargo, no se permite que 'trunquemos' literales usando el signo -. Para poder **segregar** un subliteral de un literal dado, usamos las funciones de nombre **LEFT$, RIGHT$** o **MID$.** Estas funciones nos permiten extraer un trozo de un literal, tomando los caracteres que lo componen y están situados a la **izquierda** (los delanteros), a la **derecha** (los últimos) y en **medio** (los medianeros). La función de nombre **INSTR** permite cotejar dos literales para ver si uno **pertenece** al otro, y el cotejo puede comenzarse a partir del carácter que ocupa una posición dada.

- **LEFT$**

Al nombrar esta función, el Basic nos entrega los primeros **n caracteres** comenzando por los situados a la izquierda del dato literal dado como argumento.

```
10 LET A$="AMSTRAD"
20 LET B$=LEFT$(A$,3): 'con lo que segrega los 3
                        caracteres ANTeriores del
                        literal A$ para formar B$
30 PRINT B$
```

```
por lo que EXpondrá  AMS  'valor del literal B$
```

- **RIGHT$**

Al nombrar la función **RIGHT$** el Basic entrega los **últimos n caracteres** comenzando por el situado en el extremo derecho del dato literal dado.

```
10 LET A$="AMSTRAD"
20 LET B$=RIGHT$(A$,4):  'con lo que segrega los 4
                          caracteres ULTeriores del
                          literal A$ para formar B$
30 PRINT B$
```

```
por lo que EXpondrá  TRAD  'valor del literal B$
```

- **MID$**

Cuando al Basic se nombra como función la clave **MID$,** entregará la parte de un literal que tenga una longitud de **n caracteres,** comenzando a partir del que ocupa la **posición p** del literal dado como argumento de la función.

```
10 LET A$="ESTO DEMUESTRA LA FUNCION MID$"
20 LET B$=MID$(A$,6,9):  'con lo que segrega los 9
                         caracteres MEDianeros del
                         literal A$, empezando desde
                         el 6, para formar el B$

30 PRINT B$

por lo que EXpondrá  DEMUESTRA  'valor dado a B$
```

La clave MID$ puede usarse también como un **comando** colocándola como **miembro izquierdo** de una igualdad, que en informática recuerda es un comando de **asignación,** y mediante el cual se señala esa porción del subliteral y se le altera por el nuevo valor dado.

```
10 LET  A$="HOLA HERMOSAS PERSONAS"
20 MID$(A$,6,8)="QUERIDAS": 'con lo que reemplaza esos 8
                            caracteres MEDianeros del
                            literal A$ -ya declarado-
                            contados desde el 6, con los
                            dados ahora en el comando.
30 PRINT A$

por lo que EXpondrá  HOLA QUERIDAS PERSONAS
                      valor del literal A$ ahora
```

- **INSTR**

El nombre de esta función es abreviatura de **in**cluido en literal (string) y al nombrarla el interpretador Basic escruta el primer literal dado para ver si en él está contenido como subliteral el segundo dado. Si el subliteral sí pertenece al literal, la función devuelve como resultado la **posición** a partir de la cual coinciden. En los demás casos devuelve el valor cero.

Al nombrar esta función deberá observarse que el subliteral debe pertenecer por completo al literal escrutado.

```
Si es A$="ABCDEFG"
  la función  INSTR(A$,"EFG") entregará el valor 5
  la función  INSTR(A$,"FGH") entregará el valor 0
  la función  INSTR(2,A$,"ABCD") entregará el valor 0
  la función  INSTR(2,A$,"DEF") entregará el valor 4
```

En el último ejemplo de cotejo de literales, el subliteral "DEF" se compara con el literal a partir de la **segunda posición** del literal escrutado, y devuelve un valor que nos informa que dicho subliteral "DEF" pertenece al primero a partir de la posición 4 de éste.

- **CHR$**

En una parte anterior de este capítulo usamos la función **ASC** para averiguar el código ASCII equivalente a un carácter literal dado. La función de nombre **CHR$** permite hacer la equivalencia en la dirección contraria: nos entrega el carácter literal que corresponde a un código numeral ASCII dado.

La función **CHR$** nos permite incluir los **caracteres no visivos** dentro de nuestro programa, y es una función extremadamente potente.

¿Por qué querremos exponer lo no visivo? me parece que te oigo preguntar.

¿No tienes ningún gusto por la aventura?

Si examinas la tabla 3.1 verás que algunos de los códigos se usan para mover el cursor en diferentes direcciones de la pantalla. O bien, ¿cómo harías que sonara el timbre? Mandando que expusiera CHR$(7) lo conseguirías. Sigue y ensaya.

```
10 CLS:LOCATE 1,24:PRINT "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
20 PRINT CHR$(7)
30 FOR I=1 TO 100:NEXT
40 GOTO 20
```

¿Ves lo que quiero decir? Simplemente has pedido al Basic que exponga un carácter no visivo, y al nombrar la función CHR$ interpreta que ha de efectuar el envío de ese pitido. La línea 10 la he incluido para mostrarte que incluso aunque está sonando el timbre, también está actualmente exponiendo datos en pantalla.

La función CHR$(n) también puede aprovecharse para comparar literales de un único carácter:

```
10 INPUT A$
20 IF A$=CHR$(65) THEN PRINT "!SI!" ELSE PRINT "!NO!"
30 GOTO 10
```

Puedes incluso usarla en 'tandem' con otra función, haciendo lo que los matemáticos denominamos **función de función.** Ensaya con el siguiente programa:

```
10 LET X$="A"
20 INPUT A$
30 IF A$=CHR$(ASC(X$)) THEN PRINT "SI"
                       ELSE PRINT "NO"
40 GOTO 40
```

Hace exactamente lo mismo que el programa anterior.

- **LEN**

Al nombrar la función LEN con un argumento literal, el Basic entregará la cantidad de caracteres de ese literal. Es una función muy útil cuando estamos tratando con datos literales.

```
10 LET X=0:LINE INPUT A$
20 FOR I=1 TO LEN(A$):LET X=X+1
30 NEXT
40 PRINT "A$ tiene ";X;" caraacteres (LONgitud de A$)"
50 GOTO 10
```

- **LOWER$**

Al nombrar la función LOWER$ (de lower case=caja baja) se convertirán todos los caracteres que sean letras mayúsculas a las correspondientes letras minúsculas.

```
10 LINE INPUT X$
20 FOR I=1 TO LEN(X$)
30 IF MID$(X$,I,1)=CHR$(30) OR MID$(X$,I,1)=32 OR I=1
      THEN GOTO 50
40 MID$(X$,I,1)=LOWER$(MID$(X$,I,1))
50 NEXT I
60 GOTO 10
```

- **UPPER$**

Esta función opera de la misma manera que LOWER$, pero convirtiendo de minúsculas a mayúsculas.

HEX$** y **BIN$ son funciones muy útiles para pasar de un sistema de numeración a otro, y pueden usarse dentro de un programa que convierta números según diferentes **bases** de numeración.

```
10 INPUT X
20 PRINT "EN BINario  ES ";BIN$(X)
30 PRINT "EN DEcimoheXal ":HEX(X)
40 GOTO 10
```

Dos funciones que todavía no hemos mencionado son las que tienen por nombre **SPACE$** (de espacio en blanco) y **STRING$** (serie de caracteres). Ambas funciones son ayudas valiosísimas cuando se están rellenando literales, o cubriendo una parte de pantalla con caracteres repetidos.

Al nombrar la función STRING$, se forma un literal de una determinada longitud, mediante la repetición del carácter dado como argumento.

```
10 LET X$=CHR$(129)+CHR$(132)
20 PRINT STRING$(40,X$)

10 FOR I=1 TO 24
20 PRINT STRING$(40,140)
30 NEXT
```

¿Has observado lo rápidamente que se rellena la pantalla con caracteres?

Finalmente, tenemos dos funciones realmente potentes para convertir la **índole** de los datos, de literal a numeral mediante la función **VAL**, y de numeral a literal mediante la función **STR$**.

Al nombrar la función **VAL** (de value=valor) con un argumento literal, convertiremos dicho argumento en un valor numeral que esté representado por los mismos caracteres -cifras- que haya en el argumento literal. De hecho, podemos decir que **eval**uamos ese argumento literal, o que lo convertimos a índole numeral.

Obviamente, el literal argumento debe ser un numeral, y si es un número real debe contener el **punto** decimal o la indicación de exponente (E).

```
LET B$="100.60"      :PRINT VAL(B$) ==> EXpondrá 100.60
LET B$="999999999"   :PRINT VAL(B$) ==> EXpondrá999999999
LET B$="9999999999":PRINT VAL(B$) ==> EXpondrá1E10
LET B$="ABC45"       :PRINT VAL(B$) ==> EXpondrá 0
LET B$="45ABC64"     :PRINT VAL(B$) ==> EXpondrá 45
LET B$="1E-3"        :PRINT VAL(B$) ==> EXpondrá 0.001
```

Observa en los ejemplos anteriores que la función **VAL** sólo convierte a índole numeral los caracteres **delanteros** del literal dado como argumento, que sean realmente cifras. La evaluación de un literal termina por tanto con el primer carácter no numérico (y la **E** en determinadas circunstancias lo es) que encuentre. Cuando el literal consta sólo de letras o signos, o las letras preceden a las cifras, la función VAL entrega como resultado el valor cero.

```
10 LET A$=INKEY$
20 IF A$="" THEN GOTO 10
30 LET X=VAL(A$)
40 IF X<1 OR X>9 THEN GOTO 10
50 PRINT "PULSASTE UN NUMERO ENTRE 1 Y 9"
60 GOTO 10
```

La función de nombre **STR$** es un recurso muy potente del lenguaje Basic que convierte un dato numeral a otro equivalente pero de índole literal. Esta función es valiosísima en el tratamiento de texto y en la recogida de datos: los numerales pueden ser pasados a literales, la información puede ser revisada y adornada y vuelta de nuevo a su índole numeral mediante la función VAL, o bien puede ser expuesta en pantalla o por impresora, y hacerlo en un formato predeterminado.

Un punto a tener en cuenta sin embargo, es que cuando un dato numeral vuelve a convertirse a un dato literal, se deja en el literal un espacio en blanco por **delante** para permitir que se incluya el signo cuando proceda. Además, PRINT A expondrá ese valor numérico con un espacio en blanco **por detrás**, y PRINT STR$(A) expondrá el valor sin dicho espacio en blanco tras él.

```
LET A =  1650 :PRINT LEN(STR$(A))    EXpondrá 5
LET A = -1650 :PRINT LEN(STR$(A))    EXpondrá 5
```

```
10 LET X=120.62 :LET Y=-120.62
20 PRINT STR$(X); LEN(STR$(X))
30 PRINT STR$(Y); LEN(STR$(Y))
40 PRINT STR$(X) + STR$(Y)
50 PRINT STR$(X+Y)
60 PRINT STR$(Y) + STR$(X)
```

Los caracteres de control, que no son visivos, son aprovechados para establecer el formato de la imagen o para desplazar el cursor hasta otra zona de exposición sin usar el comando de **situarlo,** de clave **LOCATE.** Por ejemplo, ¿cuántas veces has visto o usado por ti mismo, un trozo en Basic similar al siguiente?:

```
10 LOCATE 3,5:INPUT "ELIGE UN NUMERO DEL 1 AL 10";X
20 IF X>10 OR X<1 THEN LOCATE 3,5:PRINT "       ";
30 LOCATE 3,12 :PRINT "INposicio ILEGAL  ";
40 FOR I=1 TO 100:NEXT
50 LOCATE 3,5  :PRINT "                    ";
60 GOTO 10
```

La misma rutina puede escribirse de forma diferente usando caracteres de control:

```
10 LOCATE 3,5:INPUT "ELIGE UN NUMERO DEL 1 AL 10";X
20 IF X<1 OR X>10 THEN PRINT CHR$(11);CHR$(18);
      ELSE GOTO 10
30 LOCATE 3,12 :PRINT "INposicio ILEGAL  ";
40 FOR I=1 TO 100:NEXT
50 PRINT CHR$(11);CHR$(18);:GOTO 10
100 PRINT CHR$(11);CHR$(8);:LOCATE 3,5:PRINT "O.K"
110 GOTO 10
```

El programa anterior pide que se inscriba un dato en la banda de 1 a 10. Si el dato ingresado no es correcto, se deja en claro esa línea usando CHR$(11) que mueve el cursor hacia arriba una línea (para contrarrestar el retorno de carro correspondiente a cuando pulsaste la tecla ENTER), y luego CHR$(18) que borra hasta el final de la línea antes de exponer el siguiente mensaje.

Si estudias la tabla 3.1 verás que los caracteres de control permiten exhibir/ocultar el cursor, y que puedes colocarlo en cualquier parte dentro de una ventana, simplemente usando otros caracteres de control. Los caracteres de control también pueden incrustarse dentro de literales para añadir algunos efectos especiales a la exposición de caracteres o símbolos gráficos. Eso es especialmente útil cuando necesitas mover un carácter que ocupa dos posiciones consecutivas verticales. Ensaya con esto:

```
10 LET X$=CHR$(240)+CHR$(10)+CHR$(8)+CHR$(8)
         +CHR$(242)+CHR$(9)+CHR$(243)+CHR$(10)
         +CHR$(8)+CHR$(8)+CHR$(241)
20 LOCATE 20,20:PRINT X$
```

¿QUIERES SABER UN SECRETO?

Antes que terminemos con el tema de los literales y de su utilidad cuando se usan gráficos, te voy a mostrar algo muy diferente que no es probable encuentres en libros y manuales. Si eres un novato, no te preocupes de los comandos que todavía no hemos comentado; ensaya el programa y comprenderás cómo funciona antes de finalizar el libro.

ENSAYA ESTE PROGRAMA

```
 10 CLS
 20 LET X$=STRING$(40," ")
 30 LET PK=@X$
 40 LET AD=PEEK(PK+2)*256+PEEK(PK+1)
 50 FOR I=AD TO LEN(X$)+AD
 60 POKE I,238
 70 NEXT
 80 FOR I=1 TO 24
 90 PRINT X$;
100 NEXT
```

El programa anterior combina diversos puntos importantes de los comentados en el capítulo anterior, y usa algunas de las funciones con literales que hemos visto en este capítulo.

Las implicaciones del programa anterior puede que no sean obvias para ti en este momento, pero te puedo asegurar que es una manera muy elegante de incluir tus símbolos gráficos dentro de literales.

El programa comienza prefijando un literal **postizo** (X$) formado por 40 espacios en blanco. Puedes preparar cualquier literal postizo en tanto y en cuanto uses uno con una longitud igual al número de caracteres que tiene el que has preparado. La línea 30 obtiene la dirección real de ese literal. La expresión **"@nombre de variable"** hará que se entregue como resultado la dirección de esa variable siempre y cuando le haya sido asignado un valor inicial por el interpretador Basic. Cuando se usa con variables literales, en realidad entrega la dirección del **descriptor del literal** que comienza con la dirección que contiene la **longitud del literal** (véase Diagrama 3.3). La línea 40 permite investigarlo examinando esa dirección más dos, que es donde está el octeto izquierdo de la dirección donde el valor literal está registrado. La dirección correcta de dicho dato literal se calcula multiplicando el octeto izquierdo por 256 y sumándole el octeto derecho (¿recuerdas el Capítulo Tres?). La línea 50 averigua la longitud de X$ y la línea 60 rellena las celdillas de memoria ocupadas por ese literal, con el símbolo gráfico de código 238, metiendo directamente ese valor **ahí** -en la celdilla. Las líneas 80 y 90 se han añadido justamente para comprobar que todo funciona.

En el ejemplo anterior formamos un literal con símbolos gráficos y caracteres de control, pero ¿no era lo suficientemente largo como para definir el literal? Usando el método anterior puedes preparar tus caracteres gráficos y de control en instrucciones DATA, luego hacer que sean **tomadas** como valores de una variable y luego hacer que se **metan** dichos valores en las celdillas ocupadas por el literal. Puedes incluso reemplazar un carácter dentro del literal, o el literal completo, si lo deseas. Estos literales también pueden exponerse en pantalla muy rápidamente, y son una de las ventajas para un programa en Basic cuando la velocidad es esencial.

Ahora, no sigas adelante porque tenemos unos cuantos trucos más que descubrir; pero antes de comenzar el siguiente capítulo permitiré que te sientes y digieras un último programa.

Ya sabrás, si realmente piensas sobre ello, que no hay nada que impida que una subrutina en código máquina sustituya una retahíla de caracteres gráficos, y luego puede apelarse a esa rutina mencionando su dirección, tal y como hacemos con la variable AD calculada en la línea 40.

Usando este método no tenemos que definir inicialmente ninguna dirección de memoria de usuario. Desde luego, cualquier rutina en código máquina tendría en este caso que escribirse de manera que pudiera caber en 255 octetos consecutivos; pero un montón de rutinas útiles pueden escribirse en ese espacio.

SOLO PARA ABRIRTE EL APETITO, ENSAYA ESTO

```
 10 CLS
 20 LET X$=STRING$(22," ")
 30 DATA &21,&01,&01,&CD,&75,&BB,&01,&EB,&03
 40 DATA &C5,&3E,&EE,&CD,&5D,&BB,&C1,&0B,&79
 50 DATA &B0,&20,&F4,&C9

 60 LET PK=@X$
 70 LET AD=PEEK(PK+2)*256+PEEK(PK+1)
 80 FOR I=AD TO AD+LEN(X$)-1
 90 READ T
100 POKE I,T
110 NEXT I
120 CALL AD
```

No te preocupes si no comprendes este programa; se ha incluido para demostrar que todas las cosas (más bien casi todas) son posibles con el ordenador Amstrad.

El censo de datos reflejado en las líneas 30, 40 y 50 corresponden a una breve rutina en código máquina que rellenará la pantalla con un símbolo gráfico. Los datos representan realmente las instrucciones del código máquina. La variable AD tiene como valor la dirección de comienzo de la serie real de datos literales, de manera que cuando **apelas a esa dirección,** mediante la instrucción **CALL AD,** estás citando la rutina en código máquina. Esta rutina puede guardarse junto con el programa Basic, y no requiere prefijación del valor de la cima de la memoria, y Basic nunca escribirá encima de ella.

Capítulo Cuatro

Tablas: Conjuntos Homogéneos Arreglados

Durante el curso de este capítulo estaremos hablando de las instrucciones de clave **READ, DATA** y **RESTORE** y de la instrucción **DIM;** y de cómo se organizan los **conjuntos** de datos con el Amstrad Basic. Al final del capítulo veremos una forma más potente y provechosa de utilizar **tablas de datos.**

Uno de los usos más comunes del ordenador es para el **proceso de datos,** y la forma más simple de datos estructurados disponibles en el Basic es la de una **lista de datos.** Las **listas** son estructuras con las que todos estamos familiarizados: una lista de artículos a comprar, una lista de nombres en el colegio, una lista de programas a ejecutar, etc.

Son pues, meros conjuntos de datos, colocados **uno detrás de otro** formando por tanto una **estructura lineal,** cuyo acceso a cada elemento es consecutivo o **secuencial.**

Para crear una lista en un programa en Basic se usa la instrucción de clave **DATA.** En dicha instrucción podemos incluir tantos elementos o **items** como deseemos y de la índole que queramos ('heterogéneos'), con la única restricción de la capacidad de memoria disponible.

```
10 DATA ENERO,FEBRERO,MARZO,ABRIL
20 DATA VERANO,OTOÑO,INVIERNO,PRIMAVERA,X
50 DATA 3,456,255,8,1064,34,&B10111000,-1,"COSAS"
```

Esta serie de instrucciones **DATA** constituyen el **censo de constantes** usados en el programa, y podemos hacer que el Basic **tome** cada uno de esos valores constantes y lo apunte como valor de una variable dada, usando la instrucción de clave **READ** (lectura). Estas dos instrucciones para **censar un valor constante,** de clave **DATA** y **tomar** esa constante como valor de una variable, de clave **READ,** se usan siempre combinadamente en el programa.

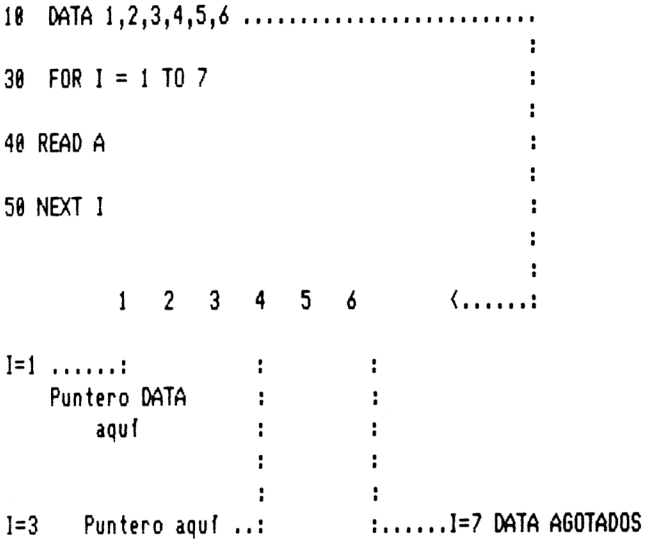
Si añadimos las siguientes líneas al programa anterior, podemos obtener una buena idea de cómo funcionan estas dos instrucciones.

```
 50 READ A$
 60 IF A$= "X" THEN GOTO 90
 70 PRINT A$
 80 GOTO 50
 90 READ A
100 IF A=-1 THEN GOTO 130
110 PRINT A
120 GOTO 90
130 READ A$
140 PRINT A$
150 END
```

En la línea 50 instruimos al Basic para que **tome** como valor de la variable **A$** el correspondiente elemento del censo de constantes DATA; como es la **primera** toma le corresponde el **primer** elemento. De hecho, el Basic prepara un **puntero** para ir marcando sucesivamente cada valor que ha tomado de la lista DATA.

En la línea 60 el programa comprueba si el valor de A$ es el literal "X", que hemos incluido en el censo de constantes DATA para indicar el final de una primera tanda de constantes. Si no es ese valor, exponemos el que tiene la variable A$ y hacemos que el Basic **vaya** de nuevo a tomar otra constante del censo, avanzando acordemente su puntero marcador. Las constantes posteriores al valor "X" están consideradas como valores numéricos, y por tanto, al tomar el valor en la instrucción 90 mencionamos una variable numeral; y de esta manera nos aseguramos siempre de saber cuál es el dato que corresponde a cada variable. Mientras que A$ no haya tomado el valor "X", exponemos el valor tomado por A$ y el proceso se repite. En el momento en que A$ toma el valor "X", el curso del programa es cambiado de dirección hasta la línea 90 donde se repite el proceso de toma de constantes, pero esta vez para dar valor a la variable numeral **A.** Para acabar esta tanda de datos se comprueba el valor tomado para la variable A; y si es igual a -1 se efectúa un salto en el curso del programa para que tome el siguiente elemento -que es un dato literal- y lo apunte como valor de la variable A$; en los demás casos se reitera el proceso.

Hemos mencionado que el interpretador Basic usa un **puntero marcador** para ir 'punteando' los datos de la lista DATA que ya ha tomado y apuntado como valor de una variable; señala por tanto a la primera de las constantes del censo que todavía no ha sido tomada. Cada vez que se efectúa una toma (una lectura) el puntero es avanzado por el Basic para señalar al siguiente item de la lista.



Este programa anterior, al ser ejecutado, da un mensaje de error **"Data Exhausted in 10"** para indicar que en el programa se intenta tomar un elemento más de los que están censados en la lista DATA, y lo indica diciendo que **se ha quedado sin datos en la 10.** Si añadiéramos como línea 60 la instrucción para que **repunte** dicho puntero marcador hacia el principio de la línea 10, incluyendo la instrucción **RESTORE 10** y volviéramos a ejecutar el programa, no surgiría esta condición de error. La instrucción **RESTORE** (restaure, restablezca, etc.) le indica al Amstrad que **repunte** el puntero marcador hasta el primer elemento de la parte de la lista DATA que comienza en la línea mencionada. **RESTORE n** hará que el puntero repunte al comienzo de una instrucción DATA numerada con **n,** y este comando es útil cuando se necesita accesar los datos de la lista censada en el programa más de una vez.

RESTORE 10

El comando anterior restablecería el puntero marcador para que señalara al primer elemento de la línea 10, con lo que la subsecuente toma de datos, mediante la instrucción **READ A** haría que A tomara el valor 1.

Podemos tener cualquier número de elementos en una lista DATA, y cada uno está separado del siguiente por el signo **coma.** Como ya hemos visto y mencionado, la índole de los datos puede ser cualquiera (literales y numerales), pero ha de corresponderse con la variable a la que se apunta como dato.

Se puede tomar más de un dato del censo de constantes mediante una sola instrucción **READ,** mencionando tantas variables como valores se deseen tomar, y no hay otra restricción sobre el número de elementos siempre y cuando haya suficientes datos en la lista. Las instrucciones DATA pueden colocarse en cualquier parte del programa, y el Basic las considerará colocadas **una tras otra** para formar el censo continuo de constantes con el que trabaja.

```
10 CLS
20 DATA A,B,C,D,E
30 DATA O,E,",",F
40 READ X,Y,Z
50 END
```

Un punto que hay que recordar cuando se usa más de una variable en una instrucción **READ** es que el **puntero** sigue avanzando una posición en la lista para cada variable que se mencione en la instrucción.

```
10 DATA MARIA,91,2692254,ALBERTO,965,253215,
       RAFAEL,91,4158716
20 DATA ALVARO,91,6388927,GONZALO,928,248070,
       DOCTOR,93,5797788
30 CLS
40 FOR I=1 TO 6
50  READ N$,C,T
60  PRINT N$; "TELEFONO (";C;") ";T
70  NEXT
```

Se debe poner cuidado en asegurar que para las variables se toman datos de la índole adecuada, de manera que no se genere un error por **discordancia de tipo** (type mismatch).

Estamos ahora preparados para pasar a un área del cómputo que la mayoría de los programadores novatos encuentran difícil de comprender... las **ringlas,** o colecciones de variables homogéneas colocadas según una estructura **regular.** (Aunque el nombre más usado es el de **tablas,** mucha gente las llama ringlas, ringleras, arreglos, raglas, etc., para destacar el hecho de que los elementos responden a una misma **regla;** que en realidad es de donde proviene el nombre inglés 'array':=acorde con una regla). Y en la vida normal las 'tablas' pueden ser de elementos no homogéneos, aunque sí están dispuestos regularmente.

Las **tablas** son las estructuras de datos más potentes al programar en Basic. Una tabla es **básicamente** (y perdón por lo de básico) una colección de datos variables ordenada regularmente, en que cada elemento de la colección posee una **referencia numérica** que le permite ser accesible **directamente** (y no como en el caso de las listas, secuencialmente). Son pues, en realidad, estructuras **planares** con un solo plano ('ristras'), dos planos o tres planos. Cada elemento de la tabla viene indicado por un número ordinal que señala la posición que tiene en cada plano, de manera que con esa **referencia** puede ser fácilmente localizado. El número de planos o dimensiones de la tabla establece el número de valores que ha de tener el **subíndice** que señala la posición de ese elemento.

Se denomina también a cada variable que compone la tabla, una variable **subindicada,** y puede ser de cualquiera de las categorías de variables que ya hemos mencionado: literales, numerales reales o numerales enteros. Todos los elementos usan un **mismo nombre** que señala a la colección, pero ha de ir seguido del correspondiente subíndice o subíndices, encerrado entre paréntesis, para señalar un elemento concreto.

A (9)

Nombre del colectivo...: ▲ ▲ :...Subíndice de la variable

El número entre paréntesis, es el identificativo de **un elemento** perteneciente a la tabla, y los paréntesis indican al Basic que la variable es una variable subindicada, perteneciente a ese colectivo.

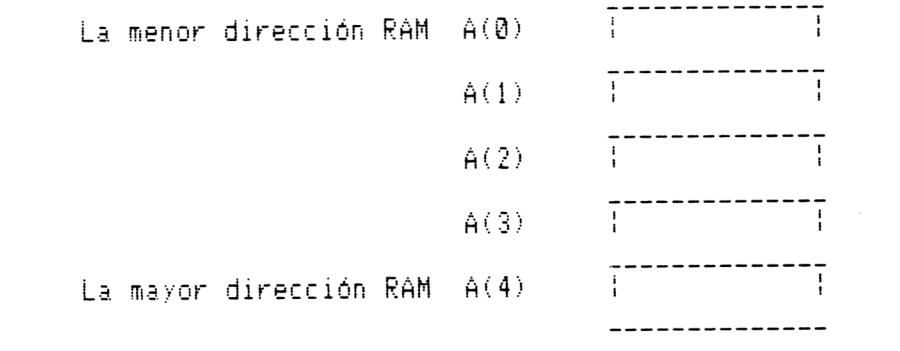
De esta manera se pueden depositar y recuperar el dato correspondiente a un elemento específico de la tabla, indicando al ordenador cuál es su **referencia** (de ahí que se hable de acceso **refrencial).**

A(0),A(1),A(2).....A(10) son todos elementos de la RINGLA A

Un ejemplo típico de una tabla monodimensional ('ristra') es una colección de programas. Si escribes todos los títulos de los programas que has comprado para tu CPC464, y haces que cada programa sea accesible directamente mencionando su **referencia,** habrás creado una tabla de una sola dimensión. Recuerda que hemos dicho casi lo mismo sobre las listas DATA; pero hay diferencias importantísimas: con la instrucción **READ** solamente podemos tomar **constantes** pertenecientes a ese censo, de una manera **secuencial;** y además los valores reflejados en las instrucciones DATA no pueden modificarse a no ser que se revise y altere el programa Basic. Por el contrario, una **ristra** como la mencionada, tiene todos sus elementos accesibles sin más que mencionar la **referencia** asociada a cada uno, y además son **variables,** lo que significa que su valc puede ser alterado cómodamente.

El ordenador almacena los valores de las variables agrupadas en una tabla, de acuerdo con la **regla** establecida para ordenar los elementos, y así para una monodimensional -ristra- lo hará en la forma presentada en el Diagrama 4.1

Diagrama 4.1



La tabla monodimensional -la ristra- **A** queda reflejada en la memoria manteniendo exactamente para sus elementos el mismo orden que indican los subíndices; con A(0) ocupando la dirección más baja de la tabla, y A(4) ocupando la más alta.

```
10 CLS
20 FOR I=0 TO 3
30  LET A(I)=I
40  NEXT
50 FOR I=1 TO 3
60  PRINT " ESTE ES A(";A(I);")"
70  NEXT
```

Este programa expondrá cada elemento de la ristra A con su correspondiente valor de subíndice, tal y como están depositados en memoria.

Otra manera de dar un valor a un elemento variable de una tabla es aprovechar las instrucciones **READ** y **DATA.**

```
10 CLS
20 FOR I=0 TO 10
30 READ A(I)
40 NEXT
50 INPUT X
60 PRINT A(X)
70 GOTO 60
80 DATA 365,400,1,67,32,56,899,3,33,40000
```

Este programa anterior demuestra cómo es posible **accesar** los datos almacenados en forma de tabla, de una manera **refrencial.** Una vez que se han tomado los datos y apuntados como valores a las variables que componen la tabla, ya no es necesario usar la instrucción **RESTORE** ni ninguna otra correspondiente al censo de constantes. Los datos pueden recuperarse de cualquier elemento de la tabla, tantas veces como sea necesario y en el orden que la **discreción** del usuario aconseje.

En el ejemplo anterior, observa que hemos aumentado el número máximo del subíndice al valor 10. Ese es el subíndice máximo que está permitido sin que previamente haya que **ocupar** espacio para almacenar los valores de la tabla.

Intentando apuntar un valor en, digamos, A(11) daría como resultado un error de **"Bad Subscript"**, que nos indica que es un **subíndice malo,** y es la manera del Amstrad de indicarnos que hemos señalado un elemento de una tabla que no existe.

Para que ocupe el espacio necesario para una tabla de más de 11 elementos, debemos usar la instrucción de clave **DIM,** con lo que reservará espacio de acuerdo con las **dimensiones** de la tabla. Así por ejemplo, **DIM A(20)** indica al ordenador que reserve en **21** sitios (numerados del 0 al 20) para almacenar los valores que corresponden a los elementos de la **ristra A.**

Accesar datos reflejados en forma de tabla puede ser asimilado a un **archivo con index.** Usando dos o más tablas, se pueden construir colecciones de datos con estructuras complejas e interrelacionadas. Digamos por ejemplo que queremos crear un listín con los nombres de nuestros amigos, y almacenar la información en el ordenador; podemos hacerlo bastante fácilmente generando una **tabla** cuyos valores serán tomados del correspondiente censo de datos constantes.

```
 10 DIM FR$(100)
 20 READ B
 30 FOR I=0 TO B
 40  READ FR$(I)
 50  NEXT

500 DATA 12,MARIA,JULIO,RAFAEL,LEOPOLDO,RUBEN,LUIS
510 DATA    ANA,JUAN,LOURDES,PAULA,ROSA,JAVIER
```

El programa anterior nos provee una manera de crear un listín con los nombres de nuestros amigos. Colocando un valor numérico al principio del censo de datos constantes, podemos fácilmente agregar datos sin tener que revisar la parte principal del programa: cada vez que añadamos el nombre de un nuevo amigo, solamente necesitaremos aumentar ese valor que refleja la cantidad de constantes. Este programa está bien, pero todo lo que hace es exponer en pantalla el nombre de nuestros amigos. La información puede ampliarse para generar más **tablas monodimensionales** -ristras- e instrucciones extra con los datos constantes.

```
 10 DIM NM$(100),DR$(100),CD$(100)
 20 READ B
 30 FOR I=0 TO B
 40  READ NM$(I),DR$(I),CD$(I)

500  4,MARIA,Juan Bravo 60,Madrid
510    JULIO,Gran Via 32,Zamora
520    LEOPOLDO,Clara del Rey 120,Madrid
530    RAFAEL,Paseo de la Habana 50,Madrid
```

Si queremos encontrar las señas de uno de nuestros amigos, simplemente escrutaremos la **ristra** (y con esta denominación podremos así hablar de una **tabla** formada por tres ristras) que contiene los nombres y eso nos dará un **subíndice** que nos permitirá sacar la dirección y la ciudad de las ristras homólogas.

```
 60 INPUT "DIME EL NOMBRE ";X$
 70 READ B
 80 FOR I=0 TO B
 90  IF X$=NM$(I) THEN GOTO 130
100  NEXT I
110 PRINT "NO ENCUENTRO ESE NOMBRE"
120 GOTO 60
130 PRINT NM$(I);"SUS SE\AS SON ";DR$(I);CD$(I)
140 LET I=B
150 END
```

En la línea 140 mandamos que **haga** I = B porque nos hemos salido del bucle sin haber terminado de dar las rondas marcadas. Aunque no es necesario hacer esto, siempre es una buena práctica hacer que la variable contadora de un bucle quede con el valor final.

De la demostración anterior puedes ver que también pueden tratarse **tablas literales** de la misma manera que las tablas numerales. Este programa de demostración es muy sencillo, y podrían diseñarse estructuras de datos más complejas para contener más información -manteniendo siempre las **ristras** con variables homogéneas- pero el principio seguiría siendo el mismo.

Encuentros en la segunda fase

Las tablas monodimensionales -las ristras- son fáciles de comprender porque podemos compararlas como ya hemos hecho, con algo concreto como una lista de nombres, teniendo en cuenta que ahora el **acceso es refrencial,** y no secuencial como en las listas. Las tablas bidimensionales -las ringlas- son también fáciles de visualizar dado que forman una estructura **planar,** y podemos asociarlas a un rectángulo de dos dimensiones, en que los dos subíndices representan filas y columnas, tal y como en un tablero de ajedrez.

Cada elemento de una tabla de dos dimensiones estará identificado mediante **dos subíndices:**

```
Un Elemento de la Tabla A = A( 1 , 8 )
                              :   :
   Primer Subíndice ..........:   :
                                  :
   Segundo Subíndice..............:
```

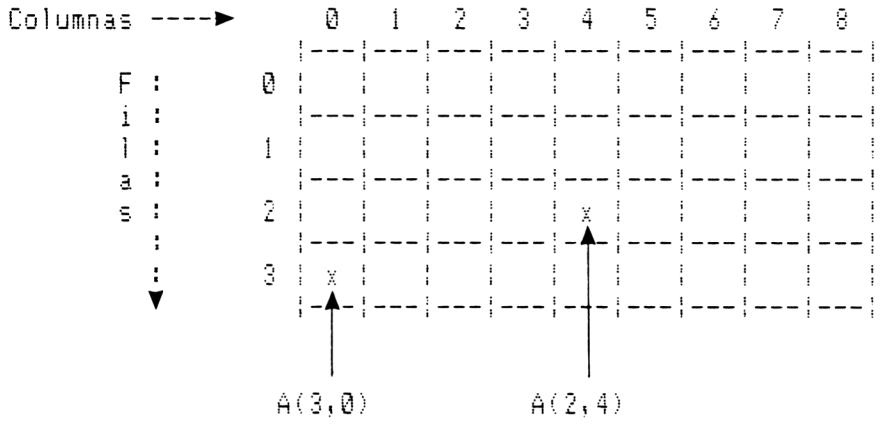
Decíamos que una tabla bidimensional puede representarse como un cuadro compuesto de **filas y columnas.** En general, el primer subíndice se asocia con las filas o líneas horizontales, y el segundo subíndice con las columnas o líneas verticales.

```
          A( 1 , 8 )
             :   :
FILA 1...:   :.....COLUMNA 8
```

Con una instrucción como **DIM A(3,8)** indicamos al ordenador que **ocupe** suficiente espacio en memoria como para almacenar 36 números (4 filas por 9 columnas), dispuestos como una ringla bidimensional.

```
10 DIM A(3,8)
20 FOR FILA = 0 TO 3
30  FOR COLU = 0 TO 8
40   LET A(FILA,COLU) = FILA*9 + COLU
50   NEXT COLU
60  NEXT FILA
```

Diagrama 4.2



Una ringla de dos dimensiones

Del diagrama 4.2 puede deducirse que siempre que queremos señalar un lugar concreto dentro del espacio ocupado en memoria por la tabla, podemos tomar como **referencia** los números de **fila** y **columna.**

Para tomar valores y apuntarlos en los elementos de una tabla bidimensional, podemos usar dos **bucles anidados,** o sea, incluidos completamente uno del otro, y haciendo que uno de ellos corresponda a la cantidad de filas, y el otro a la cantidad de columnas.

El programa anterior al diagrama 4.2 es un ejemplo de dos bucles anidados. En el ejemplo, la variable **fila** es válida **para** valores entre 0 y 3, mientras que la variable **colu** es válida **para** valores de 0 a 8. Por lo tanto, al dar la primera ronda del bucle combinado, la variable subindicada A(0,0) se le da el valor resultante de multiplicar la columna 0 por la fila 0, y luego se incrementa el valor de la variable colu en una unidad, y se pasa a dar valor en la siguiente ronda a la variable subindicada A(0,1). Cuando el contador **colu** llega a ser igual a 8, el contador **fila** se incrementa en una unidad, y los siguientes elementos de la tabla vuelven a rellenarse **para** valores de **colu** desde 0 hasta 8. Se repite este proceso hasta que el contador de **fila** alcanza el valor 3, con el valor de colu en el valor 8. Vamos a comprobar todo esto, reescribiendo el programa anterior para que exponga los valores en cada lugar de la tabla, después de rellenarla.

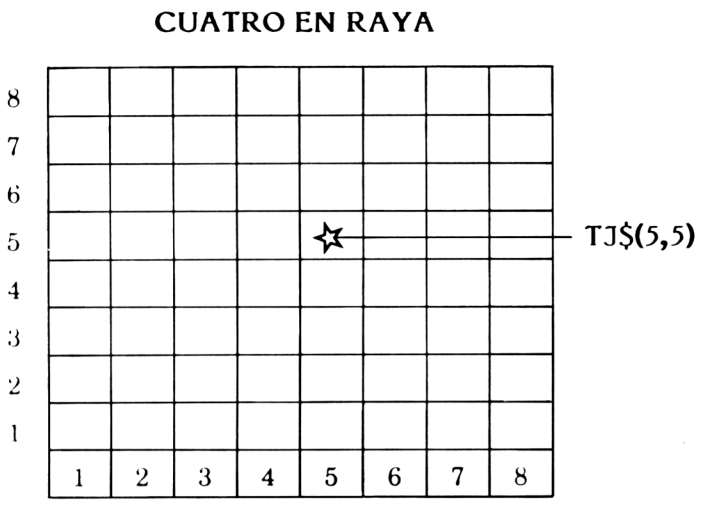
LISTADO DOS

```
10 CLS
20 DIM A(3,8)
30 FOR FILA = 0 TO 3
40  FOR COLU = 0 TO 8
50   LET A(FILA,COLU) = FILA*9 + COLU
60   NEXT COLU
70  NEXT FILA

75 PRINT SPC(10);
76 FOR COLU = 0 TO 8
77  PRINT USING "###";COLU;
78  NEXT COLU:PRINT

80 FOR FILA = 0 TO 3:PRINT "FILA ";FILA;"= "
90  FOR COLU = 0 TO 8
100  PRINT USING " ###"; A(FILA,COLU);
110  NEXT COLU
120 PRINT
130 NEXT FILA
```

Este listado nos proporciona un buen ejemplo de cómo una **ringla bidimensional de datos** puede relacionarse con un objeto específico que tenga una estructura regular cuadriculada. Estoy seguro que la mayoría de vosotros habéis jugado a **"Cuatro en Raya"**, o al menos estáis familiarizados con las reglas generales. El **Listado dos** que te damos más adelante, sigue el juego original muy estrechamente, y en el cual el Amstrad asume el papel de oponente: ¡y juega bastante bien!

Para jugar, se te pide que elijas un número de columna entre 1 y 8. Si tu elección es legal, el ordenador colocará tu pieza en la columna escogida, y a la altura de fila que le corresponda. Luego el ordenador analizará el tablero antes de efectuar su elección. El juego continúa hasta que uno de los jugadores se las apañe para colocar cuatro piezas consecutivas según una raya vertical, horizontal o diagonal, o hasta que no haya más espacio disponible dentro del tablero de juego.

Diagrama 4.3



Del Diagrama 4.3 resulta patente que el tablero de juego queda muy bien representado por una ringla monodimensional, que denominaremos TJ$(8,8) (y aquí lo de tabla sólo tiene que ver con lo de tablero, por eso usamos ringla). Observarás que dentro del programa no hemos usado las posiciones del tablero en que la fila o la columna es 0. Aunque sí ocupan espacio en memoria, pero queda a nuestra discreción no usarlas cuando nos es más cómodo considerar como subíndices del 1 al 8.

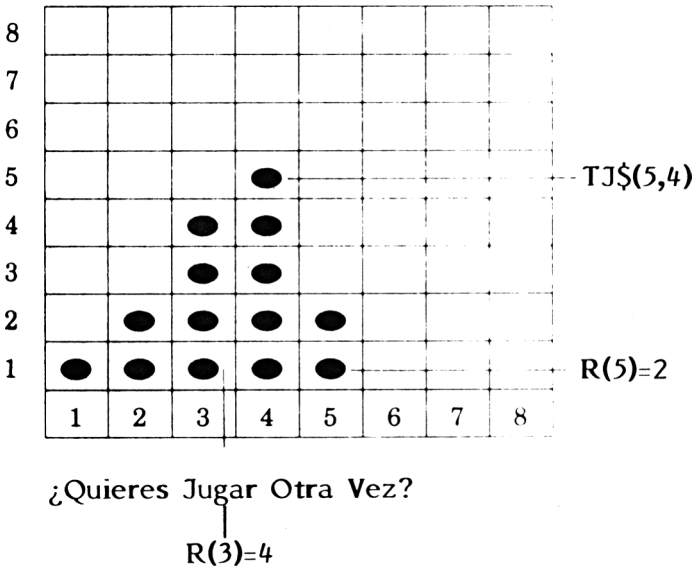
En el juego del Cuatro en Raya, las piezas o 'chinos' se colocan en el tablero de abajo hacia arriba, apilándolas; esa es la razón de que hayamos numerado las filas horizontales de 8 al 1 (Diagrama 4.4). La posición vertical de cada fila está reseñada en la ristra DN(8) con la posición Y más alta en primer lugar, de manera que DN(1) contiene la posición Y para la 20-ésima línea horizontal de pantalla, y DN(2) contiene la posición Y para la línea 18 de pantalla, etc. La posición según el eje X se calcula multiplicando la columna elegida por el jugador por el número 3 y añadiéndole 7 para que el Basic **sitúe** el cursor en la posición correcta de exposición del 'chino'.

La ringla TJ$(8,8) se usa para mantener en memoria una copia de la imagen en pantalla -un **cliché**- y siempre que se coloca una pieza en el tablero su posición queda registrada en esta ringla. El ordenador escruta esta ringla bidimensional para comprobar si un movimiento es legal, y también para decidir dónde coloca su 'chino' cuando le toca el turno.

La ristra **PR(8)** se emplea para reseñar cuántas piezas hay en cada una de las columnas del tablero: es por tanto el puntero o 'boya' que indica la altura alcanzada en cada columna.

Diagrama 4.4

CUATRO EN RAYA



La ristra denominada G(16) contiene los **parámetros de evaluación** que son usados por el programa cuando debe decidir cuál es el movimiento que le conviene hacer al mecano. Puedes experimentar con ellos ensayando diversos valores y observando cómo los cambios afectan a la táctica del juego.

Si realmente quieres analizar el programa, ahora es una buena oportunidad de ensayar el comando para que **ponga rastreador** de clave **TRON.** Ponlo en marcha y sigue el programa mirando al mismo tiempo en el listado y comprobando el curso del programa a medida que los números de línea son expuestos en pantalla. Puedes incluso hacer que se interrumpa la ejecución del programa pulsando sucesivamente ESCape ESCape, y aprovechando para corroborar algunos valores de variables.

```
  5 DEFINT A-C,D,H,I,J-T,V-Z
 10 DIM TJ$(8,8),A(4),R(8),K(4),J(4),G(16),DN(8),TH$(8)
 30 REM  "4 EN RAYA" PARA COMPUTADORA Y UN JUGADOR
 40 MODE 1:GOSUB 390:GOSUB 1130:
    BORDER 0:INK 0,0:INK1,26:INK 2,21:INK 3,6
 50 LET TH$=STRING$(8,CHR$(230))
 60 MODE 1:GOSUB 470:GOSUB 280
 70 GOTO 540

 80 REM ********** CALCULADOR PRINCIPAL **********
 90 SOUND 1,100,8
100 LET O$=H$:IF X$=H$ THEN LET O$=C$
110 LET Y=1:LET YY=0:LET S=0:GOSUB 160
120 LET Y=1:LET YY=1:GOSUB 160
130 LET Y=0:LET YY=1:GOSUB 160
140 LET Y=-1:LET YY=1:GOSUB 160
150 RETURN

160 LET XX=1:LET A=1:LET Q=0:LET S=S+1
170 LET M=0
180 FOR I=1 TO 3:
     LET H=X+I*YY:LET N=R+I*Y
190  IF H<1 OR N<1 OR H>8 OR N>8 THEN GOTO 250
200  LET G$=TJ$(N,H):IF M=0 THEN GOTO 230
210  IF G$=O$ THEN LET I=3:GOTO 260
220  LET Q=Q+1:GOTO 250
230  IF G$=X$ THEN LET A=A+1:GOTO 250
240  LET M=1:GOTO 210
250  NEXT I
260 IF XX=0 THEN LET A(S)=A:LET K(S)=Q:RETURN
270 LET XX=0:LET YY=-YY:LET Y=-Y:GOTO 170

280 REM ********* DIBUJADOR TABLERO *********
290 PEN 2:PAPER 3:LET J=49:
    FOR I=10 TO 32 STEP 3:
     LOCATE I,22:PRINT CHR$(J)+" ":LET J=J+1:
     NEXT
300 PAPER 0
310 FOR I=136 TO 520 STEP 48:PLOT I,40:DRAW I,328:NEXT
320 FOR I=40 TO 328 STEP 32:PLOT 136,I:DRAW 520,I:NEXT
330 LOCATE 15,2:PEN 3:PRINT"4 EN RAYA":PEN 1:RETURN

340 REM ********* INGRESADOR DE DATOS *********
350 LOCATE 10,25:PRINT SPACE$(30);:PEN 2:
    LOCATE 10,25:PRINT MES$
360 LET IN$=INKEY$:IF IN$="" THEN GOTO 360
370 RETURN
```

```
380 REM ********* DEFINIDOR CARACTERES *********
390 SYMBOL AFTER 90:
    FOR I=93 TO 96:
     READ N1,N2,N3,N4,N5,N6,N7,N8
400  SYMBOL I,N1,N2,N3,N4,N5,N6,N7,N8:
     NEXT
410 DATA   7,31,57,127,126,59,28,7,
         224,248,156,254,126,220,56,224
420 DATA   3,15,153,185,255,191,152,7,
         192,240,153,157,255,253,25,224
430 LET H$=CHR$(93)+CHR$(94):'FICHA HUMANO
440 LET C$=CHR$(95)+CHR$(96):'FICHA MECANO
450 RETURN

460 REM ********* PREPARADOR DE VARIABLES *****
470 DATA 20,18,16,14,12,10,8,6
480 FOR I=1 TO 8:READ DN(I):NEXT

490 REM ******** PARAMETROS DE EVALUACION ********
500 REM *** EXPERIMENTA CAMBIANDOLOS Y OBSERVA ***
510 DATA 1,120,505,1E22,1,880,3000,1E30,
         1,80,1000,1E16,1,475,3050,1E14
520 FOR I=1 TO 16:READ G(I):NEXT
530 RETURN

540 SOUND 1,100,15:SOUND 0,50,7:
    LET MES$="] Quieres ser el primero ?":GOSUB 350
550 IF IN$="S" OR IN$="s" THEN GOTO 580
560 IF IN$<>"N" AND IN$<>"n" THEN GOTO 540
       ELSE LET X=INT(RND*8)+1:GOTO 990

570 REM ******** JUGADOR HUMANO *******
580 PRINT CHR$(7):
    LET MES$="Escoge un numero (1 al 8)":GOSUB 350
590 LET X=INT(VAL(IN$)):IF X=0 THEN GOTO 580
600 IF X>=1 AND X<=8 THEN GOTO 620
610 GOTO 580
620 LET R=R(X):IF R>7 THEN GOTO 580
630 LET R(X)=R+1:LET R=R+1:LET E=X*3+7:
    LET F=DN(R):LET TJ$(R,X)=H$
640 LOCATE E,F:PEN 2:PRINT H$;
650 LET X$=H$:SOUND 1,100,7:GOSUB 90
660 FOR I=1 TO 4:
    IF A(I)<4 THEN NEXT:GOTO 710
670 LET I=4
680 INK 1,6,0:PEN 1:LOCATE 10,25:PRINT SPACE$(30):
    LOCATE 12,25:PRINT"<<< ! HUMANO GANAS! >>>"
690 FOR BB=1 TO 4000:NEXT::INK 1,26:GOTO 1120
```

```
700 REM ******** PENSADOR MECANO *******
710 LET P6=0:MES$="Pensando....":
    LOCATE 10,25:PRINT SPACE$(30)
720 LOCATE 10,25:PRINT MES$;:LET Z1=23:LET Z2=25:
    LOCATE Z1,Z2:PEN 2:PRINT TH$;
730 LET U=0:LET J=1:
    FOR P=1 TO 8:LET R=R(P)+1
740  IF R>8 THEN GOTO 940
750  LET E=1:LET X$=C$:LET F=0:LET X=P
760  GOSUB 90
770  FOR L=1 TO 4:LET J(L)=0:NEXT
780  FOR I=1 TO 4:LET A=A(I):IF A-F>3
                                  THEN LET I=4:GOTO 990
790   LET Q=A+K(I):IF Q<4 THEN GOTO 810
800   LET E=E+4:LET J(A)=J(A)+1
810   NEXT I
820  FOR I=1 TO 4:LET W=J(I)-1:IF W=-1 THEN GOTO 840
830   LET Z=8*F+4*SGN(W)+I:LET E=E+G(Z)+W*G(8*F+I)
840   NEXT I
850  IF F=1 THEN GOTO 870
860  LET F=1:LET X$=H$:GOTO 760
870  LET R=R+1:IF R>8 THEN GOTO 900
880  GOSUB 90
890  FOR I=1 TO 4:IF A>3 THEN LET E=2 ELSE NEXT I
900  IF E<U THEN GOTO 940
910  IF E>U THEN LET O=1:GOTO 930
920  LET O=O+1:IF RND>1/O THEN GOTO 940
930  LET U=E:LET P6=P
940  LOCATE Z1,Z2:PEN 3:PRINT CHR$(231);:LET Z1=Z1+1:
     SOUND 1,500,14:
     NEXT P
950 IF P6<>0
       THEN GOTO 980
       ELSE LOCATE 10,22:PEN 1:PRINT CHR$(5);
960 LOCATE 5,25:PRINT "**** Es un EMPATE ****"
970 FOR a=1 TO 1000:NEXT:GOTO 1120
980 LET X=P6
990 LOCATE 10,25:PEN 1:
    PRINT"Pongo en la columna";X;"         ";
1000 LET R(X)=R(X)+1:LET R=R(X)
1010 LET TJ$(R,X)=C$
1020 LET X$=C$
1030 LET E=X*3+7:LET F=DN(R):SOUND 0,100,23,15
1040 FOR I=1 TO 3:LOCATE E,F:PRINT "  ";:
      FOR BB=1 TO 100:NEXT BB:LOCATE E,F
1050  PEN 3:PRINT C$;:FOR BB=1 TO 100:NEXT BB:
      NEXT I:SOUND 1,17,23,8
1060 GOSUB 90
```

```
1070 FOR I=1 TO 4:IF A(I)<4 THEN NEXT I:GOTO 580
1080  LET I=4
1090  LOCATE 5,25
1100  FOR I=1 TO 8:PEN 2:LOCATE 10,25:
       PRINT "<<<< MECANO GANA >>>>";:
       FOR BB=1 TO 500:NEXT BB
1110   LOCATE 10,25:PEN 3:PRINT" !JA! !JA! !JA! ":
       FOR BB=1 TO 500:NEXT BB:
       NEXT I
1120 FOR a=1 TO 100:NEXT:RUN
1130 BORDER 13:INK 0,13:INK 1,0:INK 2,24:INK 3,26:CLS
1140 CLS:PEN 1:LOCATE 14,1:PRINT "INSTRUCCIONES":
     LOCATE 14,2:PEN 3:PRINT STRING$(12,CHR$(208))
1150 PRINT:PEN 2:PRINT "4 EN RAYA";:PEN 1:PRINT ":"

1160 REM  El 1 es control de IN/EX_GRESO
1170 PRINT:PRINT"   El juego consiste en  colocar
                 tus fichas sobre  un tablero con la
                 intención de conseguir cuatro de
                    tus fichas sobre una linea."
1180 PRINT:PRINT"   Bien sea :
                    Diagonalmente
                    Verticalmente
                    Horizontalmente"
1190 PRINT:PRINT" La computadora intentará ganarte
                  con su ingenio; asi que ! EN GUARDIA !"
1200 PRINT:PRINT" Ahora pulsa " +CHR$(34)+"ENTER"+CHR$(34)+
                       " para comenzar."
1210 IF INKEY$=CHR$(13) THEN RETURN ELSE GOTO 1210
```

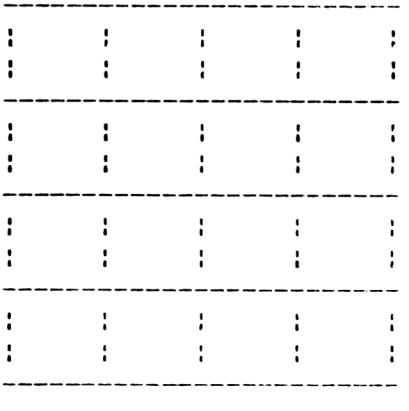
MAS ALLA DE LA TERCERA FASE

Hemos alcanzado el punto en que los débiles de corazón cejan y se colocan a un lado: ellos desaprovechan su capacidad -que sin duda tienen- para copar con los términos adicionales de las dimensiones extra. No seas uno de esa mediocre mayoría; emplea algo de tiempo en comprender los **conjuntos ordenados regularmente,** y pronto te percatarás que las **ringlas** multidimensionales son instrumentos de programación muy útiles que pueden aprovecharse con ventaja, y con los que no es difícil trabajar, tanto como mucha gente piensa.

Como un ejemplo, considera el caso de un amigo mío llamado Sancho Sánchez Sanchis. Sancho es un comerciante en ferretería y dirige un negocio detallista de mucho éxito.

Como puedes imaginar, tiene cientos de pequeños y diferentes artículos almacenados en su tienda de **hardware,** pero el problema más grande de Sancho es el de llevar la cuenta de cuántos tornillos tiene en inventario. El almacena un surtido de tornillos que van de 1/16-avo de pulgada hasta 8 pulgadas, en incrementos de 1/16-avo de pulgada. Sancho es una persona minuciosa y ordenada, y guarda los tornillos en gavetas compartimentadas en cajetines similares a la del diagrama 4.5.

Diagrama 4.5



Sancho también es una persona con **razonamiento lógico,** y no tardó mucho en comprarse un ordenador Amstrad y darse cuenta que era capaz de escribir un programa para tener registrados todos sus tornillos de inventario usando simplemente una **ringla** donde reseñar todos los datos. Sus tornillos estaban guardados en dos **armarios,** cada uno con cuatro **cajetines** y cada gaveta contenía suficientes cajetines rectangularmente dispuestos como para almacenar tornillos con un incremento total en diámetro de una pulgada; así que en la gaveta 1 tenía tornillos desde 1/16-avo de pulgada hasta 1 pulgada; en la gaveta 2 tenía los tornillos desde 1 1/16-avo de pulgada hasta 2 pulgadas, y así sucesivamente hasta que en la gaveta 8 tenía los tornillos desde 7 1/16-avo de pulgada hasta 8 pulgadas inclusive.

Sancho nunca había usado anteriormente tablas multidimensionales, pero así fue como comenzó su programa.

```
TORNI (4, 4, 4, 2)
       :  :  :  :
       :  :  :  := Armarios
       :  :  := Gavetas
       :  := Columnas
       := Filas
```

```
 10  DIM TORNI(4,4,8,2)
 20  FOR ARMA = 1 TO 2
 30   FOR GAVE = 1 TO 8
 40    FOR FILA = 1 TO 4
 50     FOR COLU = 1 TO 4
 60      READ QTD
 70      TORNI(FILA,COLU,GAVE,ARMA)=QTD
 80      NEXT COLU
 90     NEXT FILA
100    NEXT GAVE
110   NEXT ARMA
```

```
500  DATA 3600,2000,4000,12000,1200,4560,2389,1907
510  DATA 1200,1324,5279,567,1290,4317,2345,6732
520  DATA ...........
```

Una vez que había rellenado la tabla con las cifras de la cantidad de tornillos en inventario, Sancho sabía que podría encontrar la cantidad de un tornillo dado si examinaba el valor del elemento homólogo de dicha tabla.

```
Elemento TORNI(1,1,1,1) refleja los que hay
                        de   1/16 de pulgada
Elemento TORNI(1,1,2,1) refleja los que hay
                        de 1 1/16 de pulgada
Elemento TORNI(1,1,4,2) refleja los que hay
                        de 7 1/16 de pulgada
```

OTRO SECRETO DEL AMSTRAD

En el último capítulo comentamos, y desde luego demostramos, que un programa en código máquina puede ser reflejado como un dato literal, y luego puede apelarse a ese programa desde Basic. Un inconveniente de este método es que los parámetros y valores percibidos por el programa en código máquina, tenían que cedérsele definiendo sitios específicos de la memoria de manera que el programa en código máquina pudiera recoger esos valores, e.g. es decir, **metiendo** el parámetro en un sitio de la memoria y observando lo que había **ahí** guardado. Tú has corroborado lo fácil que es usar ringlas de datos una vez que las reglas fundamentales iniciales han sido comprendidas; imagínate entonces lo fácil que sería **ceder** valores de parámetros y argumentos a una subrutina en código máquina simplemente cargando un elemento de la ringla con un valor, v.g. A(1,3) = &FF.

Teclea el siguiente programa y ejecútalo.

```
500 DATA &3E00,&0000,&FEF5,&3E07
510 DATA &0000,&2000,&E602,&5F3F
520 DATA &FEF1,&300E,&4B04,&34CD
530 DATA &C9BD


 10 DEFINT A-Z:DIM A(12)
    :' Solamente trabajando con variables enteras
 20 FOR I = 0 TO 12
 30  READ A(I)
 40  NEXT
```

```
 50 LET A(1) = 8
 60 LET A(4) = 16
 70 GOSUB 1000
 80 LET A(1) = 7
 90 LET A(4) = 34
100 GOSUB 1000
110 LET A(1) = 13
120 LET A(4) = 8
130 GOSUB 1000
140 LET A(1) = 12
150 LET A(4) =  9
160 END

→ 1000 CALL @A(0):RETURN
```

Después de hacer que el programa anterior **trabaje,** si eres un novato te habrás quedado asombrado. Los programadores expertos se habrán dado cuenta que esta manera de establecer las subrutinas en código máquina abre un panorama completamente nuevo.

Sin embargo, comencemos desde el principio. Las instrucciones DATA son donde está reflejada la rutina en código máquina que esquiva los comandos de sonido del Basic y nos permite accesar directamente los registros del generador de sonido. No te preocupes en este momento de cómo funciona; trataremos del generador de sonido con mayor detalle en un capítulo ulterior. Las líneas en Basic de la 50 a la 150 son las encargadas de traspasar los parámetros a la rutina en código máquina, y la línea 1000 localiza el comienzo de la tabla en memoria y luego cede la ejecución a la sección en código máquina que ha sido almacenada en forma de ringla. La ventaja de usar ringlas en lugar de literales (simples retahílas de caracteres con acceso cuasi secuencial) para conservar y ejecutar rutinas en código máquina son dobles: el programa no está limitado a 255 octetos, lo que significa que se pueden emplear secciones en código máquina más grandes; los parámetros pueden pasarse desde Basic bastante fácilmente cargando valores en los elementos correspondientes de la ringla. Además, si se requiere una gran cantidad de información para la rutina en código máquina, se puede preparar otra tabla de datos y la dirección de esta nueva tabla puede traspasarse al programa en código máquina usando la expresión **@nombre de variable.**

Dado que este método es innovador, hay ciertas reglas que deben obedecerse cuando se prepara la ringla o tabla para aceptar el código máquina. Para el lector impaciente, explicaré ahora cómo funciona este método.

El programador en Basic que desee regresar a este tema posteriormente, puede saltar hasta el siguiente capítulo.

Preparación de la ringla con código máquina:

El Listado tres es el listado real en **lenguaje para ensamblar** usado cuando se escribió la rutina anterior.

LISTADO TRES

```
;************************************************
;  Demostración del uso de código maquina
;  dentro de una TABLA Basic
;  y luego CITAndo la dirección sede, o base.
;  Ensamblado usando KUMA'S ZEN EDITOR ASSEMBLER.
;************************************************
;
;
;
1   00              NOP             ;para alinear
2   3E00            LD A,00
3   00              NOP             ;para alinear
4   F5              PUSH AF
5   FE07            CP 07
6   3E00            LD A,00
7   00              NOP             ;para alinear
8   00              NOP             ;para alinear
9   2002            JR NZ,SALTO
10  E63F            AND 3FH
11  5F      SALTO:  LD E,A
12  F1              POP AF
13  FE0E            CP 0EH
14  3004            JR NC,RETRO
15  4B              LD C,E
16  CD34BD          CALL 0BD34H     ;CARGAR REGISTRO
17  C9      RETRO:  RET
18  00              END
```

Cuando se usa este método, la rutina escrita para ensamblar **debe ser reubicable.** La razón para eso es simple: las ringlas de datos en Basic quedan conservadas en memoria inmediatamente delante del censo de variables sencillas usadas en el programa; y por tanto, siempre que se cambian variables o se añaden, las ringlas se desplazan en memoria; lo que significa que una **apelación** (call=cita, llamada) a otra sección de código dentro de la rutina, no cederá la ejecución a la dirección correcta. Los saltos absolutos (JP) también deben evitarse.

Después de haber ensamblado tu código fuente, tienes que determinar dónde es necesario insertar los valores de los parámetros cedidos, y colocar ahí instrucciones de **no operación** (NOP). Cada elemento de la ringla ocupará dos octetos (o sea, un doble-octeto, o **dexeto**). Intenta disponer las celdillas que recibirán los valores cedidos desde Basic para que caigan sobre celdillas de **numeración par**, dentro de la rutina acabada. Además, si la longitud del código máquina no es divisible por 2, inserta instrucciones de no operación (NOP) hasta que lo sea.

Código objeto para el Listado Tres

```
0000   00      NOP   <-- ;para alinear el inicio
0001   3E00    LD A,00
0003   00      NOP   <-- ;para percibir aquí un dato
0004   F5      PUSH AF
0005   FE07    CP 07
0007   3E00    LD A,00
0009   00      NOP   <-- ;para percibir aquíun dato
000A   00      NOP   <-- ;para lograr longitud par
000B   2002
000D   E63F
000F   5F
0010   F1
0011   FE0E
0013   3004
0015   4B
0016   CD34BD
0019   C9
```

Si no hubiéramos incluido las instrucciones de no operación en los octetos con dirección relativa 0,3,9,A, los datos no hubieran quedado alineados después de haber sido reseñados en la ringla. No olvides que el **código objeto** se coloca en instrucciones DATA invirtiendo los convenios normales del microprocesador Z80, y poniendo el octeto izquierdo primero (el más significativo), y el octeto derecho último (menos significativo). Como consecuencia, el primer valor para el censo de constantes DATA del Basic será &3E00: el microprocesador Z80 invertirá estos dos octetos para obtener el orden correcto cuando el Basic los cargue en las celdillas de memoria ocupadas por la tabla. Usando el código objeto anterior, la instrucción DATA se calcula con:

&3E00,&0000,&FEF5 .....&C9BD.

Puedes comprobar que el código está correctamente alineado si examinas cada dato de la tabla y luego mandas en modo directo que exponga el valor decimohexal del elemento, dando el comando **PRINT HEX$(A(0)).** Si estás usando el programa ensamblador-editador ZEN de la compañía Kuma, puedes hacer entrar a la sección supervisora si apelas a la dirección 16384, y luego efectuando el desensamblaje a partir de la celdilla de memoria cuyo valor fue el entregado por PRINT HEX$(A(0)). El código máquina deberá concordar exactamente con el código objeto original.

Una observación adicional, es que todas las ringlas **deben ser de numerales enteros;** en los demás casos, el código no quedará alineado tal y como se espera. Una vez que hayas dominado esta técnica, todos los otros métodos de relacionarte con código máquina te parecerán anticuados, e infinidad de posibilidades se abren al programador ingenioso: una rutina que pueda **ordenar** una ringla de datos en Basic según orden ascendente -¡1000 octetos en 9,6 segundos!

Capítulo Cinco

'Hincando' Datos en Memoria

Ya es hora de que echemos una mirada **dentro** del ordenador. Pero no, no es necesario usar destornilladores, estoy hablando en metáfora. Echaremos la mirada dentro del ordenador con la ayuda de dos palabras clave del Basic que puede te hayan llamado la atención en un capítulo anterior: el comando para que **meta** ahí, en un lugar de la memoria, un dato, de clave **POKE** ('embolsar'), y la función que le manda nos diga lo que **hay** metido en una determinada celdilla, de clave **PEEK** ('mirar').

Con la ayuda de estas claves podemos accesar directamente sitios de la memoria, y podemos usar con grandes ventajas este comando y esta función en nuestra programación.

Aunque el siguiente ejemplo sea de poco -o de ningún- uso para nosotros, esta es la manera de usar un comando **POKE**, decirle al Amstrad que **hinque** un determinado valor en la memoria.

```
10 DEFINT A-Z
20 CLS
30 INPUT "ELIGE UN NUMERO   ";X
40 PRINT "TU NUMERO HA SIDO ";X
50 LET Y = RND(50*2)+1
60 PRINT "VOY A SUMAR ";Y; "A TU NUMERO ";X
70 POKE @X,RND(100*2)+1
80 PRINT "EL RESULTADO ES ";X
```

Antes de comenzar a usarlos es necesario comprender la diferencia entre la **memoria de únicamente escritura,** o **memoria del sistema;** y la **memoria de escritura-lectura,** o **memoria de usuario.** Además, para aprovechar estas dos claves debemos ser conscientes de lo que sucede dentro del ordenador.

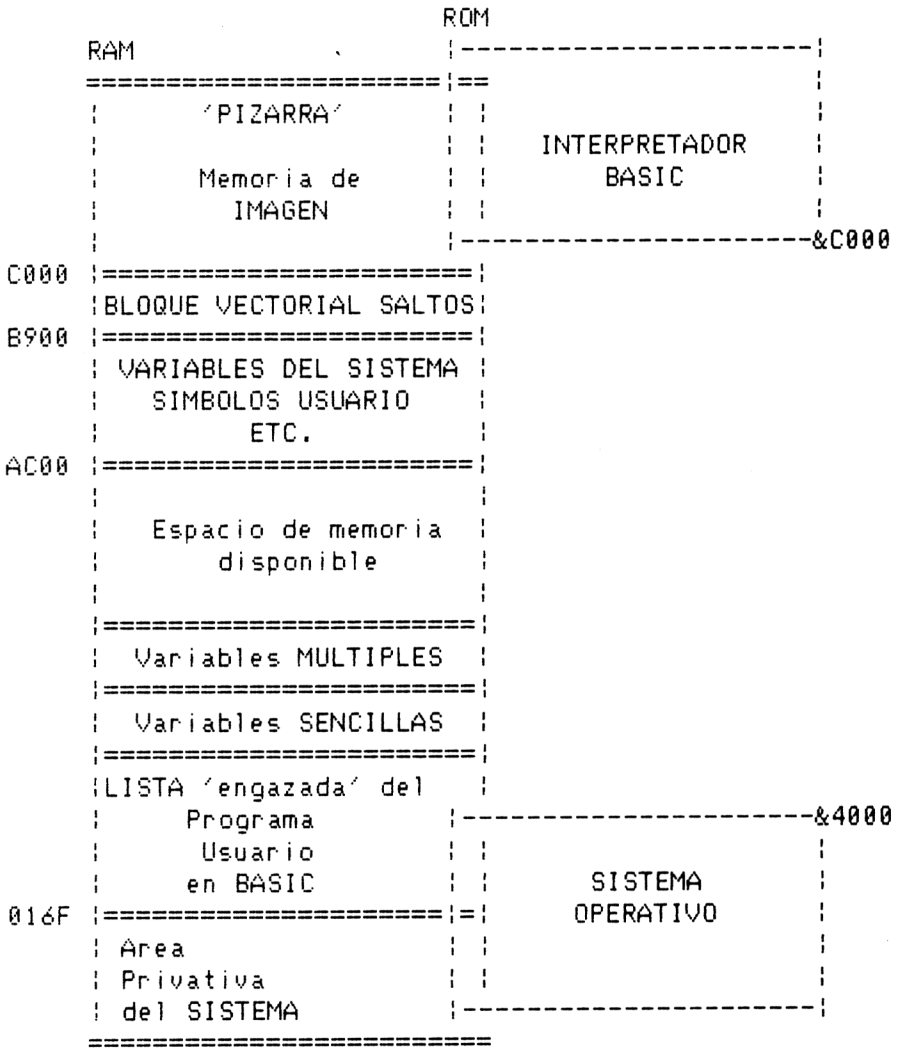
ROM y RAM

ROM que es abreviatura de **"Read Only Memory"**, es esa parte de la memoria del ordenador que el usuario únicamente puede ver lo que hay ahí, que ha sido grabado en ella por el fabricante, de manera indeleble y que por tanto no puede ser alterada por el usuario. El comando POKE no permite meter ahí ningún valor. La memoria llamada **RAM** que es abreviatura de **¡Read-Alterable Memory!** es por otro lado, donde el usuario puede meter cualquier valor (POKE) y puede examinar lo que **hay** metido (PEEK). Originalmente, se decía que el nombre correspondía a "Random Access Memory", que venía a ser algo así como Memoria con Acceso Fortuito, pero en cuanto al acceso, ambas memorias son de la misma clase, por lo que modernamente ya no se considera que el nombre RAM provenga de ahí. En efecto, podemos accesar directamente **cualquier** celdilla de memoria, sabiendo la **referencia** numérica que corresponde a esa celdilla. Esta clase de acceso llamada **refrencial** o directa, se contrapone al acceso secuencial de aquellos dispositivos que sólo nos permiten comenzar por la primera de las celdillas de almacenamiento y sucesivamente y una tras otra ir examinando cada una de las siguientes: 0,1,2,.....100, etc.

Cuando echamos una mirada dentro de la memoria, la situación se complica además por el hecho de ser imposible predecir si estamos mirando lo que hay en la ROM o en la RAM, y se hace por tanto esencial comprender qué parte de la memoria está implementada en ROM y qué parte lo está en RAM. También debemos ser cuidadosos con **qué** y **dónde** metemos algún valor: algunos sitios de la memoria son usados por el sistema y si nosotros hincamos ahí algún valor sin el preciso cuidado, es muy fácil provocar que el programa interpretador **se la pegue.** Aunque no podemos dañar el ordenador por muchas 'puyas' que le metamos, si el sistema se queda colgado, nuestro único recurso es apagar y volver a encender otra vez. Esta clase de acción no es satisfactoria en absoluto; especialmente si hemos acabado de teclear un largo programa en la memoria. Así que recuerda: siempre que uses el comando **POKE** dentro de tu programa, asegúrate primero de que lo has **guardado** en cinta antes de intentar ejecutarlo. Si el ordenador se la pega entonces, tendrás por lo menos la comodidad de saber que puedes volver a **traer** el programa de la cinta, sin tener que volver a teclear un montón.

Diagrama 5.1

Organización de la Memoria



La organización de la memoria es bastante complicada por el hecho de que en el ordenador hay 64 Kilo-octetos de memoria de escritura/lectura (RAM) y 32 Kilo-octetos de memoria de únicamente lectura (ROM). Sin embargo, como el microprocesador Z80 sólo es capaz de tratar con **direcciones** pertenecientes a un espacio de 64 Kilo-octetos de memoria, eso implica que debemos emplear algún método para **conmutar** las diversas "bancadas" de memoria. El Amstrad CPC464 soluciona este problema dividiendo la **memoria de sistema** (ROM) en dos secciones iguales:

El bloque de memoria ROM que contiene el **sistema operativo** (el programa 'regentador' de la máquina) está ubicado desde la dirección 0000_H hasta la $3FFF_H$, en la parte inferior, por tanto, del espacio de memoria manejado por el Z80, y es el responsable de realizar operaciones tales como gobernar el generador programable de sonido, o las transferencias de datos al cassette. El bloque de memoria **ROM** con el **interpretador Basic** está ubicado en la parte superior de la memoria desde la dirección $C000_H$ hasta la $FFFF_H$ (direcciones 49152_D a 65535_D). Eso significa que siempre que el ordenador necesita acudir al interpretador Basic, tiene que 'quitar de en medio' de la zona de memoria RAM que contiene la imagen proyectada en pantalla, y cuyas direcciones se solapan con esa parte de la memoria ROM. Sin embargo, esta conmutación de poner y quitar secciones de memoria no afecta a la imagen en pantalla, sino que más bien la escritura posterior en la 'pizarra' reflejo de la pantalla no puede tener lugar hasta que ha sido **facultada** otra vez la zona RAM dedicada a la pantalla.

Las dos secciones de memoria ROM pueden, por separado, hacerse activas y desactivarse. Ya hemos dicho que no podemos intrínsecamente meter ningún dato en una memoria de únicamente lectura, de manera que no importa si están facultadas o canceladas las operaciones con la ROM, en la dirección en que queremos meter algún dato; y por tanto no podemos hacer ningún daño al sistema por mucho comando **POKE** que usemos para 'embolsar' datos ahí.

Variables sistemales

Dado que una memoria de únicamente lectura (ROM) nunca puede cambiar lo que hay escrito en ella, y en un entorno tan cambiante como es un programa en Basic, el ordenador debe tener algún método de conservar diversos datos e informaciones, e.g. la longitud del programa en Basic, la posición corriente del cursor, la cima de la memoria usable, etc.; el Amstrad mantiene registrados esta clase de datos reservándose un bloque de memoria de escritura-lectura que comienza en la dirección $AC00_H$ (44032_D). Este área de memoria es muy crítica y debe tenerse extrema precaución cuando se quiere 'hincar' algún valor dentro de esa región.

La lista de las instrucciones del programa

Esta lista de instrucciones contiene las frases originales -el listado **fuente**- del programa real que, o bien has inscrito directamente a través del teclado, o bien has traído desde el almacenamiento en cinta. La dirección de comienzo de esta lista está prefijada en la dirección $016F_H$ (367_D), pero es obvio que la dirección final variará según el tamaño del programa existente en memoria. A medida que se añaden o se suprimen líneas de programa, el final de esta zona de la memoria quedará alterado acordemente.

A medida que se inscribe cada línea de un programa en Basic desde el teclado, es analizada en busca de las **palabras reservadas** y siempre que se encuentra una, se sustituyen las letras que lo forman por un **solo código** interno para ocupar menos espacio, y es ese código (por algunos llamado "glifo" y en inglés se usa la palabra 'token':=ficha o moneda de jugar, entre otras cosas) el que se registra. Esta manera de ahorrar memoria es usada por muchos ordenadores, ya que es mucho más económica en cuanto a espacio: almacenar el código $8E_H$ en lugar de las seis letras del comando Basic **DEFINT**, por ejemplo. Un programa en Basic tiene muchas líneas y la mayoría de las líneas, varias instrucciones con palabras clave, de manera que el ahorro de memoria puede ser enorme; tanto como un 20%. Desafortunadamente, la **tabla de glifos** está grabada en la ROM con el interpretador Basic, de manera que en el Amstrad no puede examinarse lo que **hay** usando la función **PEEK.** Para tu conocimiento hemos escrito dicha tabla de manera que puedes ver cómo cada **glifo** está en estrecha correlación con cada palabra **clave** del Basic.

TABLA 5.2

Glifos (Códigos internos de las claves BASIC)

KEYWORD	TOKEN	KEYWORD	TOKEN
ABS	FF00	AFTER	80
ASC	FF01	ATN	FF02
AUTO	81	BIN$	FF71
BORDER	82	CALL	83

CAT	84	CHAIN	85
CHAIN MERGE	85 AB	CHR$	FF03
CINT	FF04	CLEAR	86
CLG	87	CLOSEIN	88
CLOSEOUT	89	CLS	8A
CONT	8B	COS	FF05
CREAL	FF06	DATA	8C
DEF FN	8D	DEFINT	8E
DEFREAL	8F	DEFSTR	90
DEG	91	DELETE	92
DIM	93	DRAW	94
DRAWR	95	EDIT	96
ELSE	97	END	98
ENT	99	ENV	9A
EOF	FF4D	ERASE	9B
ERR	FF41	ERL	E3
ERROR	9C	EVERY	9D
EXP	FF07	FIX	FF08
FOR	9E	FRE	FF09
GOSUB	9F	GOTO	A0
HEX$	FF73	HIMEM	FF42
IF	A1	INK	A2
INKEY	FF0A	INKEY$	FF43
INP	FF0B	INPUT	A3
INSTR	FF74	INT	FF0C
JOY	FF0D	KEY	A4
KEY DEF	FF75	LEFT$	DB
LEN	FF0E	LET	A5
LINE INPUT	A6 A3	LIST	A7
LOCATE	A9	LOG	FF0F
LOG10	FF10	LOWER$	FF11
MAX	FF76	MEMORY	AA
MERGE	AB	MID$	AC
MIN	FF77	MODE	AD
MOVE	AE	MOVER	AF
NEW	B1	NEXT	B0
ON GOSUB	B220	ON GOTO	B2A0
ON BREAK GOSUB	B39F	ON BREAK STOP	B3CE
ON ERROR GOTO	B29C	ON SQ GOSUB	B59F
OPENIN	B6	OPENOUT	B7
ORIGIN	B8	OUT	B9
PAPER	BA	PEEK	FF12
PEN	BB	PI	FF44
PLOT	BC	PLOTR	BD
POKE	BE	POS	FF78
PRINT	BF	RAD	C1
RANDOMIZE	C2	READ	C3
RELEASE	C4	REM	C5

RENUM	C6	RESTORE	C7
RESUME	C8	RETURN	C9
RIGHT$	FF79	RND	FF45
ROUND	FF7A	RUN	CA
SAVE	CB	SGN	FF14
SIN	FF15	SOUND	CC
SPACE$	FF16	SPEED INK	CD A2
SPEED KEY	CD A4	SPEED WRITE	CD D9
SQ	FF17	SQR	FF18
STOP	CE	STR$	FF19
STRING$	FF73	SYMBOL	CF
SYMBOL AFTER	CF 80	TAG	D0
TAGOFF	D1	TAN	FF1A
TEST	FF7C	TESTSR	FF7D
TIME	FF46	TROFF	D2
TRON	D3	UNT	FF1B
UPPER$	FF1C	VAL	FF1D
VPOS	FF7F	WAIT	D4
WEND	D5	WHILE	D6
WIDTH	D7	WINDOW	D8
WINDOW SWOP	D8 E7	WRITE	D9
XPOS	FF47	YPOS	FF48
ZONE	DA	=	EF
<	F2	+	F4
-	F5	*	F6
/	F7		

==========

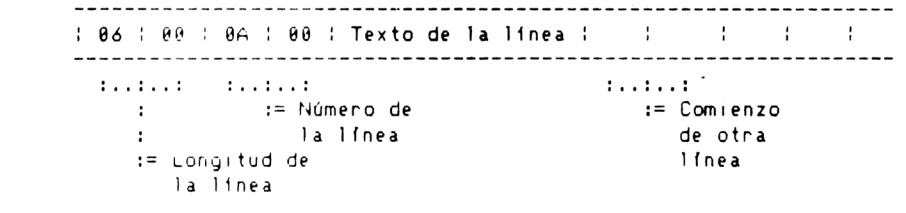
En cuanto tu programa pasa a ser ejecutado, el control es cedido a su programa **gestor de ejecución** que analiza cada línea de programa en busca de dichos códigos internos, y cuando encuentra uno, el gestor de ejecución cede el control a la rutina que trata con ese comando o función. Cuando se detecta un glifo, e.g. $A0_H$:=GOTO, sabe inmediatamente cuál es la rutina a la que ha de apelar, y cuál es la dirección en memoria donde comienza. El secreto descansa en la forma en que dichos glifos están adjudicados a las diversas palabras clave del Basic.

La ROM del Basic contiene una **lista de verbos de acción** en la que se reflejan las direcciones de todas las rutinas encargadas de cumplimentar los diversos **comandos** del Basic. Para encontrar la dirección correcta, el ordenador tiene que calcular el **desplazamiento** desde el principio de la lista, de acuerdo con el valor del **glifo** en cuestión. El primero de estos glifos es el código 80_H (AFTER), y todos los otros son superiores a ese código.

Por ejemplo, el glifo del comando CLOSEOUT tiene el valor 89_H, y para encontrar la dirección de la rutina encargada del **cierre**, el ordenador resta el valor 80_H del glifo dado (89_H) y suma este resultado (9_H) a la dirección donde empieza la lista de verbos de acción. El gestor de ejecución extrae entonces la dirección contenida en ese elemento de la lista, y efectúa un salto hasta esa dirección que es donde comienza la rutina real que cumplimentará todo comando de clave **CLOSEOUT.**

Detrás de la lista de instrucciones del programa en Basic, están registradas las variables empleadas por el programa, dispuestas también en forma de **lista** o 'censo'. Esta lista contiene los nombres y valores de todas las variables utilizadas en el programa en Basic. Está subdividida en tres secciones: nombres de las variables sencillas, punteros descriptores de los datos literales y colección de variables subindicadas, esta lista también altera el tamaño a medida que se suprimen o presentan nuevas variables.

Estructura de una línea de programa en Basic



La tabla de **clases de variables** está reflejada como una ringla de longitud constante que comienza en la dirección $AE0C_H$ (44556_D). Esta tabla ocupa 26 octetos, y cada elemento corresponde a una letra del alfabeto (de la A a la Z, sin la Ñ). Al poner en marcha el ordenador, las condiciones iniciales de esta tabla son estipuladas por el Basic como el valor 05_H para cada una de las 26 celdillas. Este marcador 05_H denota valores **reales** que es lo preceptuado para casos de omisión en Basic. Si un programa en Basic comienza con una instrucción para hacer que trate las variables como de categoría entera, mediante la clave **DEFINT,** cualquier variable mencionada en la instrucción tendrá la letra inicial correspondiente cambiada al valor de marcador 02_H.

Una instrucción en Basic para qué haga **X!=33.3333** todavía será admitida como variable real debido al marcador de clase que va detrás del nombre de la variable. Sin embargo, si el interpretador Basic se encuentra la instrucción para que haga **Z=765.8993** examinará la tabla para ver si el elemento correspondiente a la Z contiene un marcador de la categoría real (05_H).

Ahora es un buen momento para que intentemos ver lo que **hay** en ciertas posiciones de la memoria del Amstrad, y averigüemos así algunos de sus secretos. El siguiente programa te permitirá examinar la tabla de categorías de variables, y verás cómo cada instrucción altera el elemento de la tabla correspondiente a la inicial del nombre de la variable.

```
 10 CLS
 20 GOSUB 150
 30 CLS
 40 DEFINT A-Z
 50 GOSUB 150
 60 CLS
 70 DEFSTR
 80 GOSUB 150
 90 CLS
100 DEFREAL
110 GOSUB 150
120 END

150 FOR I% = &AE0C TO &AE0C+25
160  PRINT PEEK(I%);
170  NEXT
180 INPUT "Pulsa 'Enter' para otra OJEADA";
190 RETURN
```

Podemos hacer que **embolse** nuestras rutinas en código máquina dentro de las celdillas de la memoria, usando el comando de clave **POKE**. Desde luego, ya hemos usado este método en un capítulo anterior donde colocamos una breve rutina como un valor literal. También podemos usar el comando **POKE** para ceder parámetros a una rutina en lenguaje máquina, desde nuestro programa Basic. Por ejemplo, si tuviéramos un programa en código máquina que moviera una bola alrededor de la pantalla, usando una celdilla concreta de la memoria para señalar la posición corriente de la bolita, podríamos observar lo que **hay** en esa posición, en nuestro programa Basic; o cambiar la posición desde Basic **hincando** el nuevo valor en esa celdilla de memoria concreta.

Ejemplo de cómo **meter** valores en direcciones sucesivas de la memoria, operación secuencial:

```
10  LET X = 'DIREcción_COMienzo'
20  LET Y = 'DIREcción_FINalizo'

30  FOR I = X TO Y
40   READ A
50   POKE I,A
60   NEXT
70  END

80 DATA 34,56,&45,&65,12 .......
```

Rutina para meter datos en direcciones elegidas de la memoria, operación **refrencial:**

```
10  READ DIRE,VALOR
20  IF DIRE = -1 AND VALOR = -1 THEN  END
30  POKE DIRE,VALOR
40  GOTO 10

50 DATA DIR1,00,DIR2,&CD,DIR3,&F0,DIR4,&45,-1,-1
```

Donde DIR1,DIR2, etc, representan las celdillas de memoria elegidas. La comprobación de si DIRE y VALOR = -1 es para asegurarnos que se ha alcanzado el final del censo de constantes DATA. La razón de que también compruebe que VALOR = -1 es simplemente porque es posible que una dirección en memoria valga -1, como muy pronto descubriremos.

Dos reglas para observar y alterar lo que HAY

En el Capítulo 3 descubrimos el método de cómo el Amstrad almacena los números de 2 octetos: primero el octeto derecho o menos significativo (LSB); segundo el octeto izquierdo o más significativo (MSB). Siempre que efectuemos la lectura de un valor binario de 16 bits (doble octeto), tendremos que aplicar la fórmula correctora para paliar esta anomalía:

REGLA 1

VALOR = VOZ*256 + VOD

O BIEN

VALOR = PEEK(dire) + 256*PEEK (dire+1)

REGLA 2

Si estamos usando números **en base 10** para representar las direcciones en memoria, deberemos aplicar la siguiente regla para todas las celdillas de memoria cuya dirección sobrepase la 32767_D:

DIRECCION ACTUAL = DIRECCION FORMAL - 65536

La razón de esta regla guarda relación con los **enteros sin-signo** de 16 bits y la notación según **complemento a doses** de un número decimal (véase Capítulo 3). En el sistema de base 10 formal, las direcciones se expresan en la banda de 0 a 65535. Sin embargo, los enteros con signo usados por el ordenador, sólo están en la banda de -32768 a +32767; pero el entero con signo -1 se expresa como $FFFF_H$, mientras que $FFFF_H$ corresponde como **entero sin-signo** al número 65535_D; y de ahí es de donde surge la confusión. Echa una mirada a la siguiente lista.

```
 32767 = 7FFF
 32768 = 8000
 32769 = 8001
  ...     ...
 65535 = FFFF

-32768 = 8000
-32769 = 8001
  ...     ...
-65535 = FFFF
```

De lo anterior es patente que todos los enteros positivos superiores al 32767 se representan como equivalentes a sus números opuestos negativos. Como **X%** es una variable entera, sólo aceptará números en la banda de -32768 a +32767, por lo que para señalar una celdilla de memoria cuya dirección esté por encima de ese límite, deberemos usar el número equivalente negativo. Si no comprendes esta regla vuelve atrás a revisar el Capítulo 3.

El programa del **Listado 4** te permitirá observar el valor que **hay** en cualquier celdilla de la memoria de escritura/lectura. Mostrará la dirección de la celdilla seguida del valor que haya contenido en ella. Si el valor observado tiene un carácter equivalente en el código ASCII, dicho carácter será expuesto en pantalla. En los demás casos, el programa simplemente expone el valor real en dicha celdilla. Este programa **no es un desensamblador,** pero sí te permite investigar unos cuantos puntos interesantes del Basic Locomotive.

Después de teclear el programa, observa lo que hay en las celdillas que van desde la dirección $AC00_H$ (44032_D) hasta la $C000_H$. Ese es un área de la memoria que comentaremos un poco más adelante. También observa lo que hay a partir de la dirección $016F_H$ (367_D), y verás algunos rasgos interesantes. Te percatarás que son las propias líneas del programa que estás ejecutando.

Desafortunadamente, dada la manera en que está organizada la memoria en el Amstrad, no puedes observar lo que hay en la memoria de únicamente lectura; y además, para poder interpretarlo adecuadamente, necesitarás lo que se llama **programa desensamblador.** Esta clase de programa es uno que automáticamente traduce las instrucciones en código máquina del Z80, en los correspondientes **nemónimos** (nombres nemotécnicos) usados para programar en lenguaje máquina. Desde luego, para comprender los listados obtenidos tienes que tener un conocimiento somero del código máquina. El **Listado 5** es un ejemplo de lo que puede conseguirse al desensamblar. Dicho listado fue producido por el programa editador/ensamblador **ZEN** de la compañía Kuma. Los primeros números que aparecen son las direcciones de las celdillas de memoria, y los otros valores son las instrucciones de código máquina **en decimohexal.** La última columna son las instrucciones de código máquina utilizando los **nemónimos,** llamados así porque son **nombres** (nimos) que son más fáciles de **recordar** (nemo-técnicos).

LISTADO 4

```
 10 DEFINT A-Z
 20 INPUT "DAME Dirección de COMienzo ";SA
 30 INPUT "DAME Dirección de FINalizo ";EA

 40 CLS:LET FL=0
 50 FOR L=SA TO EA
 60  LET HAY = PEEK(L)
 70  GOSUB 200
 80  NEXT

 90 PRINT:INPUT "PULSA 'S' PARA OTRA RONDA ";A$
100 IF A$="S" THEN GOTO 40 ELSE GOTO 20
200 IF HAY < 33 THEN GOTO 250
210 IF HAY >122 THEN GOTO 250
220 IF FL = 0 THEN PRINT L;" ";CHR$(HAY);:GOTO 240
230 PRINT CHR$(HAY);
240 LET FL=1:RETURN

250 GOSUB 300:PRINT L;" ";HAY
260 LET FL=0
270 RETURN

300 IF FL=1 THEN PRINT:RETURN
310 RETURN
```

A lo largo del siguiente capítulo, a medida que utilicemos algunos de los comandos incorporados en el Amstrad, emplearemos programas en código máquina para ilustrar el texto. Para sacar ventaja de los rasgos innovadores del CPC464, es esencial un buen conocimiento del lenguaje de **ensamblaje.** La enseñanza de este lenguaje de programación se sale fuera del ámbito de este libro, pero sí es prudente emplear unos cuantos párrafos explicando los rudimentos del lenguaje.

```
DIREcción de memoria
¦
¦       Instrucción Código Máquina
¦       ¦
ACFE    211010
AD01    CD75BB
AD04    01E803
```

La comprensión del programa anterior es bastante difícil. Para ver lo que el programa hace, sería necesario tener un index o tabla de todos los diferentes **códigos de operación** admitidos por el microprocesador Z80, y con la explicación de lo que se efectúa en cada operación. Echa ahora una mirada a ese mismo programa escrito mediante la ayuda de un programa **editador-ensamblador.**

```
1        ORG  0ACFEH
2  LD HL,1010H
3  CALL 0BB75H
4  LD BC,1000H
```

Esto es lo que se denomina un **listado fuente** (y lo de fuente es por que realmente el programador lo escribe **originalmente** así) y obviamente si supieras lo que las diferentes **claves nemotécnicas** significan, serías capaz de investigar lo que el programa hace.

Una vez que el programa fuente ha sido escrito, podemos hacer que un programa ensamblador lo examine y lo **ensamble,** obteniendo las correspondientes instrucciones en **verdadero código máquina,** expresadas normalmente según la notación **hexadecimal,** y después de unos pocos segundos de trabajo del programa ensamblador, obtendríamos un listado similar al siguiente:

LISTADO 5

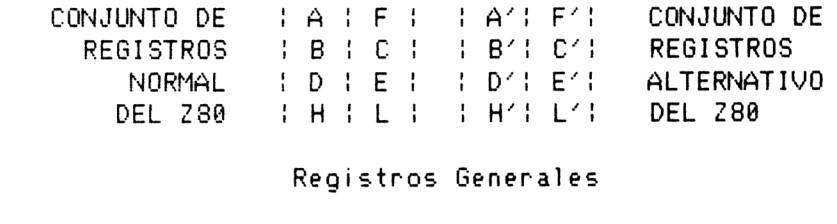
```
1                ORG   0ACFEH
2  ACFE  211010        LD HL,1010H
3  AD01  CD75BB        CALL 0BB75H
4  AD04  01E803        LD BC,1000H
```

Observa por favor, que el programa ensamblador utilizado para escribir las rutinas de este libro es el denominado ZEN de la casa Kuma. Los programas ensambladores trabajan de diferentes maneras, así que si tú estás usando otro producto puede que tus listados varíen ligeramente con respecto a los ejemplos aquí mostrados.

Usamos un programa ensamblador para que nos traduzca los **nemónimos** del Z80 que corresponden a nuestros programas, que son más fáciles de comprender al ser palabras claves con cierta ortografía similar al inglés. El programa ensamblador traduce este programa -denominado programa **fuente**- en verdaderas instrucciones en código máquina que el ordenador directamente comprende. El resultado es otro programa, expresado en **hexadecimal**, que se denomina habitualmente **código objeto.**

CODIGO FUENTE ===> ENSAMBLAJE ===> CODIGO OBJETO

El microprocesador Z80 contiene dos grupos de registros internos generales, y seis registros de propósito especial. La notación habitual para designarlos es la siguiente:



Registros Generales

| IX | IY | SP | PC | I | R |

Registros de propósito especial

Los designados como **A, B, C, D, E, F, H y L,** son los registros generales normales, mientras que los designados con esas mismas letras seguidas de apóstrofe (') son el grupo de registros generales **alternativos,** que solamente pueden ser accesados por dos instrucciones que **intercambian** (exchange) o canjean el contenido del grupo principal con el contenido del grupo alternativo, y que tienen por claves nemónicas **EX AF,AF'** para ese par de registros, y **EXX** para todo el grupo. El Z80 únicamente puede trabajar con uno u otro grupo de registros en un momento dado. Además de estos dos grupos de registros de **ocho bits,** hay cuatro registros especiales de **dieciséis bits** denominados generalmente **IX, IY** (registros de indexación X e Y) **SP** (puntero de la 'percha', o del 'stack':=pila) y **PC** (contador de programa o **Registro Seguidor).**

Otros dos registros denominados **I** (para el vector de Interrupción) y **R** (para el sistema de Refresco de memorias dinámicas) son raramente usados en la mayoría de las aplicaciones de programación normales.

El registro más utilizado, el denominado **A**, se conoce habitualmente como el **acumulador** dado que todas las instrucciones aritméticas y la mayoría de las otras instrucciones usan el contenido de este registro **A** tanto como operando como para 'acumular' el resultado obtenido de la operación. De hecho, es desde donde la mayoría de las transferencias de datos tienen lugar.

El registro denominado **F** (inicial de Flag:=banderín) se denomina también registro de **testigos**, ya que cada uno de sus ocho bits indica internamente una condición de **cierto** o de **falso** para atestiguar algunas características del resultado de una operación; y nunca se usa como operando.

El contador de programa, mejor llamado **registro seguidor** y designado como **PC**, es un registro de 16 bits que señala a la **corriente** celdilla de memoria, donde está almacenada la instrucción que será ejecutada a continuación. Otro registro de 16 bits muy importante es el designado por **SP** (de Stack Pointer) que se utiliza para indicar la dirección corriente de la memoria de escritura/lectura que actúa como **'percha'** donde se van apilando o colgando sucesivamente, y del que se pueden ir recogiendo o descolgando el contenido de los otros registros o datos cualesquiera. Otros dos registros de 16 bits que ofrecen potentes posibilidades de programación son los denominados **registros de indexación**, donde puede colocarse la dirección **base** de una tabla o ringla, mientras que en algunos de los otros registros se señala la **cota**, o desplazamiento, de un elemento concreto de la tabla.

Cada uno de los registros de 8 bits puede usarse por separado o 'solidarizarse' según parejas predeterminadas (BC, DE, HL, AF) y tratarse como auténticos registros de 16 bits.

Los programas ensambladores tienen todos sus propios conjuntos de reglas, pero no suelen ser difíciles de aprender:

```
DB  ó  DEFB  := Define Byte     ... DB &FA,"T","UNO"
DW  ó  DEFW  := Define Word     ... DW &4007 ó etiqueta
DS  ó  DEFS  := Define Space    ... DS 245 reserva
DM  ó  DEFM  := Define Message  ... DM "Pensando ..."
```

Todas estas 'instrucciones' anteriores se conocen con el nombre de **pseudo-operaciones** o bien como **directrices,** y son directamente interpretadas por el ensamblador para que lleve a cabo funciones predeterminadas, pero no se traducen en auténticas instrucciones en código máquina para el microprocesador.

Los rótulos o etiquetas (en inglés 'labels') se usan para hacer referencia a una instrucción desde dentro de otra. Por ejemplo: **JP Z,OTRA** corresponderá a una instrucción para que **salte** (JumP), si el testigo de Zero está alzado, a la dirección correspondiente a la etiqueta **otra.** Una etiqueta puede ser comparable por tanto a un número de línea en un programa en Basic; e.g. **SI A = 0 ENTONCES VAYA A LA 100.**

El signo **punto-y-coma** (;) se usa de la misma manera que las instrucciones de **recordatorio** en el Basic, y el programa ensamblador ignorará todo el texto que vaya detrás de ese signo. Es una buena práctica de programación acostumbrarse a documentar cada programa. Debes creerme que cuando uno mismo examina un programa después de unos cuantos meses, le es difícil comprender lo que tenía en su mente en el momento en que escribió el programa.

El convenio usado para la mayoría de los ensambladores es como sigue:

```
Etiqueta  Cod.Oper.  Operando  Comentarios

COMIN:    LD A,      (DE)      ;Cargue en el 'A'cumulador
                                 el contenido de la celda
                                 se\alada por la dirección
                                 registrada en 'HL
```

Una operación muy común del microprocesador es por ejemplo la de **comparar** dos datos, cuyo nemónimo en el Z80 es **CP,** y que funciona de una manera similar a la instrucción en Basic:

```
10 IF A = 10 OR A>10 OR A<10 THEN GOTO 100
```

Te permite tomar decisiones que afectan al curso del programa haciendo que se salga de una subrutina o que salte hasta otra parte del programa. El resultado de una **comparación** se constata comprobando el estado de ciertos bits del registro de testigos **F.**

Número del BIT:	7	6	5	4	3	2	1	0
	S	Z	-	H	-	P/V	N	C

```
C := Llevada ('Carry') e.g: 7+4 =1 y me LLEVO una
H := Semi-llevada ('Half carry')  N := suma/resta
(ambos H y N con decimales codificados en binario)
  Z := Zero(resultado = 0)          S := Signo
    P/V := Parity/oVerflow (Paridad/Rebase)
```

Los bits 3 y 5 del registro de testigos, no se usan. Los testigos de semi-llevada (H) y de sustracción (N) sólo se usan para operandos decimales codificados en binario, y no son de interés en este momento. El testigo de llevada, (también llamado de 'acarreo') indica cuando está alzado que se ha producido esa condición durante una operación de suma o una operación de resta (del cuatro al uno son siete y **me llevo** uno). Este importantísimo testigo se ve afectado directamente por las operaciones matemáticas, además de otras operaciones como las de rotación. Deberá comprenderse que todas las operaciones de **comparación** efectúan una 'resta ficticia' entre el valor contenido en el **registro A** y el valor reflejado por el operando mencionado en la instrucción, que puede ser el valor en otro registro cualquiera, o un valor absoluto y directamente dado:

```
CP C        ;compare valor en 'A con valor en 'C

CP &32      ;compare valor en 'A con ese número
```

Lo que realmente está sucediendo durante una operación de comparación es que el valor del operando mencionado se 'resta' del valor contenido en el registro A, pero el resultado obtenido en la resta no se acumula en el registro A.

```
LD A,40H  ;cargue en 'A ese valor( 64 en base 10)

CP L      ;compare 'A con 'L (resta ficticia A-L)
```

Puedes ver por lo anterior que una operación de comparación es esencialmente una operación aritmética con el registro A, y que por tanto el resultado sí afectará al testigo de **llevada.**

El testigo de Zero está alzado (valor 1) siempre que el resultado de una operación aritmética arroja como resultado el valor **cero.** Si ambos testigos de llevada y de cero se usan combinadamente, se puede comprobar cualquier posibilidad de comparación. Considera las siguientes instrucciones en Basic:

```
10 LET A = VALOR
20 IF A = 10 THEN GOTO 40
30 IF A > 10 THEN GOTO 50
40 GOTO 40
```

Expresando este programa en nemónimos del Z80, sería:

```
        LD A,VALOR      ;cargue en 'A'cumulador
        CP 10           ;compare lo de 'A' con 10
        JR Z,IGUAL      ;si dif. =0 pues salte allí
        JR NC,MAYOR     ;si dif. >0 pues salte allá
PAUSA:  JR PAUSA
```

Esta situación de **no aCarreo** surgirá si A = 10 o si A > 10, y por lo tanto siempre es sensato comparar el registro A con un valor inferior en una unidad al valor que realmente se desea comparar:

```
CP 10
JR NC,MAYOR
```

Si el testigo de llevada no está alzado, entonces A es estrictamente mayor que 9 pero podría ser igual a 10.

Esa es la razón de comprobar el testigo de **Zero** antes de comprobar el testigo de llevada o acarreo en el ejemplo anterior.

Las cuatro situaciones pueden resumirse como sigue:

```
NC --- valor en 'A'cumulador >= valor comparando
 C --- valor en 'A'cumulador <  valor comparando
 Z --- valor en 'A'cumulador  = valor comparando
NZ --- valor en 'A'cumulador <> valor comparando
```

Observa también que el valor en el registro A **no se ve afectado** por la comparación, ya que la sustracción es meramente ficticia.

El testigo de signo se utiliza con números según notación de complemento a doses, en la que si el séptimo bit 7 está alzado (valor 1) entonces el número es **negativo;** y si el bit 7 está bajado (valor 0) entonces el número es **positivo.** Por tanto, el testigo de signo simplemente reproduce el valor de este bit 7 en la representación del número. Los otros testigos serán comentados a medida que surja la situación, pero los tres ya explicados son los más utilizados.

SEÑALAMIENTO DE DIRECCIONES

Cualquier revisión detallada de un microprocesador siempre ha de mencionar los llamados modos de señalar direcciones de los operandos, o 'modos de direccionamiento'. Ahí es precisamente donde el microprocesador Z80 obtiene sus ventajas: la amplia variedad de maneras de indicar la dirección disponible en este microprocesador hace realmente la vida fácil para el programador. No creo que ningún modo de dar la dirección de los operandos te provoque un problema serio. Pronto estarás familiarizado con los más utilizados y para ayudarte te presento ahora las formas más comunes de señalar direcciones con el microprocesador Z80.

Operando inmediato:

En Basic una instrucción similar sería: **HAGA A = 3**

LD A,03 o **LD HL,5007** (en que el operando está ampliado a un doble octeto) serían los nemónimos para expresar la operación de **cargue** el registro A con el valor 03 o la pareja de registros HL con el valor 5007, respectivamente.

Operando en registro:

Es exactamente lo que indica: el valor del operando está 'registrado' en un registro dado.

LD A,C ;Cargue en A el valor que hay en C

Operando dado indirectamente por registro:

En este modo de dar la dirección del operando, el **sitio** de la memoria donde está contenido el valor del operando, viene dado precisamente por el valor contenido en una de las parejas de registros: **BC, DE o HL.** Como a cada sitio le corresponde una dirección, se necesita la pareja de registros para dar el doble octeto correspondiente. Una traducción al Basic podría ser:

```
10 LET BC = 14390
20 LET A = PEEK(BC)
```

En nemónimos para un programa ensamblador sería:

```
LD A,(BC)       ;   cargue en el registro A el valor que hay
                ;   en la celdilla de memoria señalada por
                ;   el contenido del par de registros BC.
.....
.....
LD HL,14390     ;   cargue en HL la dirección 14390
LD A,(HL)       ;   cargue en A el contenido de la celdilla
                ;   señalada por HL.
LD DE,56789     ;   la pareja DE señala a la celdilla con
                ;   dirección 56789
LD(DE),A        ;   mete en la celdilla de memoria señalada
                ;   por el contenido de la pareja DE, el valor
                ;   registrado en A.
```

Señalamiento indexado de la dirección:

Este es un modo realmente potente de indicar la dirección de un operando. Te permite depositar o recuperar datos organizados en forma de tablas o ringlas, rápidamente. Podemos hacer que los registros **IX** o **IY** contengan la dirección que señala la **base** o comienzo de la tabla; y luego sumarles una **cota,** o desplazamiento mediante un número positivo o negativo, dentro de la banda de **-128** a **+127.**

Si el registro **IX** apunta a la dirección $3C00_H$ de la memoria, podemos mandar **LD A,(IX+15)** para que cargue en el registro A el valor que hay en la celdilla de memoria cuya dirección es $C0F_H$, que es el resultado de efectuar esa suma indicada.

Igualmente, LD A,(IX+00) cargaría el acumulador con el valor que hay en la celdilla de memoria cuya dirección es $3C00_H$, que es el valor contenido en el registro doble IX.

Señalamiento implícito de la dirección:

Con este modo de indicar la dirección queremos significar que no se menciona en el nemónimo la dirección del operando, pero que de alguna forma está implícitamente mencionada dentro del propio nemónimo.

```
ADD E       ;Agregue al contenido del registro A'
             el contenido del registro 'E
SCF         ;Alce el testigo de llevada (aCarreo)

CONVENIO:

LD A,(HL) ;cargue en el A'cumulador el contenido de
           la celdilla SE\ALADA por el contenido de 'HL
Flujo de los datos:  A <=== celdilla(HL)
siempre hasta la izquierda desde la derecha, v.g: LD A,36
```

- **DJNZ**

Esta instrucción de nemónimo, tan poco nemotécnico (corresponde a la acción 'Decrement and Jump if No Zero') es una de las más utilizadas y trabaja combinadamente con el contenido del registro B como contador, para usarlo de manera similar a un bucle preconfinado en Basic.

```
        LD B,100  ;cargue en B el valor 100
RONDA:  DEC HL    ;'HL = "HL - 1
        LD (HL),A ;cargue en la celdilla se\alada por 'HL
                   el contenido del A'cumulador (lleve a ...)
        DJNZ RONDA ========== FOR i=1 TO 100
                               POKE dirección,valor
                               NEXT i

DJNZ := Decrement 'b' and Jump if Not Zero
     := HAGA B=B-1: SI B<>0 ENTONCES VAYA('salte') a RONDA
```

Capítulo Seis

Un Comentario sobre Elecciones

El Capítulo Dos nos encontró tratando con los números binarios y las potencias de dos. Comprender el sistema de numeración binario y su equivalente, el decimohexal (ya que la base de éste -16- es igual a 2 elevado a 4), puede ayudarnos con nuestra programación en toda forma y manera, y ciertamente hace la vida considerablemente fácil al tratar con gráficos.

Cada **ubicación** o posición donde se expone un símbolo en la pantalla de texto, consta de una **gradilla** de 8 por 8 puntos de imagen. En el **Modo 1** disponemos de 40 posiciones, columnas, por cada una de las 25 filas, lo que hace un total de 40*25=1000 posiciones usadas para exponer caracteres de texto. Cuando se expone un carácter en la pantalla, ocupa una de esas posiciones, un **cuadratín de ocho por ocho motas,** de las cuales hay algunas iluminadas con un color, el de la 'pluma', y otras iluminadas con otro color, el del 'papel', lo que da la forma o tipo característico de ese símbolo.

Todos los símbolos, caracteres ASCII o gráficos, siguen este mismo **patrón** de estar construidos sobre una red de 8 por 8 mallas. El carácter puede tener una sola de esas mallas iluminada en el color de tinta (por ejemplo, el punto decimal) o bien las 64 mallas pueden estar iluminadas con el color de tinta, lo que daría una mancha sólida y cuadrada en la pantalla. Este método de exponer símbolos, guarda una relación muy estrecha con el sistema de numeración en binario, al estar constituido por la iluminación de cada malla usando **uno de dos** colores. Si tratamos cada malla individual iluminada en un color como un **uno** (bit alzado) y cada malla individual iluminada en el otro color como **cero** (bit bajado), podemos convertir cada dibujo patrón en un grupo de datos que puede inscribirse en la memoria del ordenador mediante el comando **SYMBOL.**

Al convertir en datos las motas iluminadas en uno de dos colores para dar la forma patrón característica con que se expone en pantalla, el ordenador utiliza la misma **notación posicional** que se usa para expresar una cantidad según los ocho bits de un octeto.

Formato del OCTETO:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	0	1	0	1	1

PATRON CARACTERISTICO DE LA X

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^1
0	*****	*****				*****	*****	
1		*****				*****		
2			*****		*****			
3				*****				
4			*****		*****			
5		*****				*****		
6	*****	*****				*****	*****	
7								

Para convertir el dibujo patrón de un carácter en datos característicos, simplemente sumamos todos los valores posicionales de los unos (motas iluminadas con el color de la pluma) en cada línea horizontal, y el resultado es el número usado para reflejar esa raya moteada en pantalla:

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^1	
0	*****	*****				*****	*****		= 128 +64 + 4 + 2 = 198
1		*****				*****			= 64 + 4 = 68
2			*****		*****				= 32 + 8 = 40

Parte del dibujo de la "X"

Luego, con el comando **SYMBOL 128, 198,68,40.....** crearíamos las tres rayas superiores de esa X y asignaríamos ese patrón característico al símbolo de código interno 128.

Se pueden generar dibujos patrones característicos usando dos o más gradillas de 8 por 8 adosadas de manera que constituyan una gradilla de 16 por 16 mallas: de hecho puedes usar los 255 caracteres y crear así un **repertorio de símbolos** de diseño único y exclusivamente tuyo.

```
 10 CLS
 20 DATA 175,215,175,213,172,212,172,212
 30 DATA 252,0,80,248,112,32,0,0
 40 DATA 192,248,254,254,30,28,60,56
 50 DATA 56,112,112,112,126,49,65,126
 60 LET COD=128
 70 FOR I = 0 TO 3
 80  READ A,B,C,D,E,F,G,H
 90  SYMBOL COD,A,B,C,D,E,F,G,H
100  NEXT

120 LET X$ = CHR$(128)+CHR$(130)
130 LET Y$ = CHR$(129)+CHR$(131)
140 LOCATE 12,12
150 PRINT X$;CHR$(8);CHR$(8);CHR$(10);Y$
160 GOTO 160
```

La rutina anterior usa dos caracteres de control **CHR$(8)** y **CHR$(10)** para hacer respectivamente, que se retrase el cursor una posición y avance una línea de manera que el valor literal de **Y$** será expuesto en pantalla inmediatamente por debajo del valor de **X$.** Eso da el efecto de proyectar un único carácter que ocupe un cuadratín de 16 por 16 **motas.**

La sensación de movimiento puede conseguirse en la forma más simple, si primero se expone el carácter, luego se expone encima de él un espacio en blanco para borrarlo, luego la posición se cambia de lugar y se vuelve a repetir la misma serie de acciones. Debe usarse alguna clase de **bucle de demora** para aminorar la velocidad de manera que el movimiento aparente ser fluido.

```
EXponga caracter      EXponga caracter      EXponga caracter
Pause un rato         Pause un rato         Pause un rato
EXponga "         "   EXponga "         "   EXponga "         "
Pause otro rato  ──►  Pause otro rato  ──►  Pause otro rato
```

```
10 CLS: LET Y=1
20 FOR X = 1 TO 39
30  LOCATE X,Y
40  PRINT "A";
50  FOR J = 1 TO 80: NEXT
60  PRINT CHR$(8);" ";:' Para REtroCEDA una POSICION
70  NEXT
```

La animación de figuras puede simularse diseñando dos por separado que tengan forma diferente y proyectándolas alternativamente en la misma posición de la pantalla. También aquí tendrá que usarse un bucle de demora para dar animación realista y suave a la figura. Si además se emplea el método anterior para el desplazamiento de la figura, se puede fácilmente conseguir la apariencia de un hombre paseando o de un pájaro volando.

La manera en que la imagen en pantalla se consigue en el Amstrad a partir de la **proyección** del contenido de la zona de memoria reservada para ello, vulgarmente llamada **pizarra,** hace que sea virtualmente imposible efectuar la lectura directamente de lo que **hay** en pantalla usando la función **PEEK.** Esta es una característica desafortunada ya que algunas veces es deseable saber cuál es el carácter expuesto en la posición corriente del cursor. El siguiente programa te proveerá con un **utensilio** para observar lo que hay en la pantalla del Amstrad en la posición corriente del cursor y te entrega el valor como contenido de la variable entera **PK%.** Esta variable debe especificarse antes de **apelar** a la subrutina, pues en los demás casos ocurrirán resultados impredecibles. El programa también te introduce además otra manera de reflejar en un programa en Basic, rutinas en código máquina.

```
  1 GOTO 10: REM**************
 10 CLS: DEFINT A-Z: LET PK=0: LET DIR=&017B

 20 DATA &DD,&6E,&00,&DD,&66,&01,&E5
 30 DATA &CD,&60,&BB,&E1,&77,&C9

 40 FOR I = DIR TO DIR+12
 50  READ DTA:POKE I,DTA
 60  NEXT
 70 LOCATE 4,12:PRINT "A"
 80 LOCATE 4,12:CALL DIR,@PK
 90 LOCATE 1,1:PRINT "EL VALOR QUE 'HAY' ES ";PK
100 GOTO 100
```

El programa recoge cualquier carácter que esté situado en la posición corriente del cursor y lo coloca como valor de la variable PK; y luego mediante el examen de lo que hay contenido en PK habrás efectivamente observado el carácter proyectado en pantalla.

Este método usa una instrucción de recordatorio, de clave **REM**, para almacenar la rutina en código máquina. Siempre que uses este método debes preparar la línea 1 exactamente como se muestra en el programa anterior, excepto en la cantidad de asteriscos que colocas después de la palabra clave REM, ya que ésta corresponde al número de octetos consecutivos ocupados por tu rutina en código máquina. Si preparas la línea exactamente, tu código máquina siempre comenzará en la dirección **&017B$_H$**, y puede apelarse a ella para que sea ejecutada citando esa dirección.

Una observación final. Asegúrate siempre que guardas tu programa antes de intentar que se ejecute, ya que puede ser imposible revisarlo posteriormente. Si intentas listar el programa después de iniciada la ejecución, puede que obtengas un "error sintáctico". Eso puede solucionarse tecleando el comando **LIST 10**, que entonces listará todas las líneas a partir de la 10.

Este es un método muy versátil de reflejar breves subrutinas en código máquina, y siempre que sigas las instrucciones dadas anteriormente, el programa estará 'autocontenido' y no causará ningún problema.

EL MAPA DE MEMORIA DEL AMSTRAD

Lin 1 **C000$_H$** ..... **C04F$_H$**
C800$_H$ ..... **C84F$_H$**
D000$_H$ ..... **D04F$_H$**
D800$_H$ ..... **D84F$_H$**
E000$_H$ ..... **E04F$_H$**
E800$_H$ ..... **E84F$_H$**
F000$_H$ ..... **F04F$_H$**
F800H ..... **F84F$_H$**

Lin 2 $C050_H$ ..... $C09F_H$
$C850_H$ ..... $C89F_H$
D050H ..... $D09F_H$
$D850_H$ ..... $D89F_H$
$E050_H$ ..... $E09F_H$
$E850_H$ ..... $E89F_H$
$F050_H$ ..... $F09F_H$

Lin 3 $C0A0_H$ ..... $C0EF_H$
: :
: :
$F8A0_H$ ..... $F8EF_H$

Lin 4 $C0F0_H$ ..... $C1BF_H$
: :
: :
$F8F0_H$ ..... $F93F_H$

Lin 5 $C140_H$ ..... $C18F_H$
: :
: :
$F940_H$ ..... $F98F_H$

: :

Lin 25 $C780_H$ ..... $C7CF_H$
$CF80_H$ ..... $CFCF_H$
$D780_H$ ..... $D7CF_H$
$DF80_H$ ..... $DFCF_H$
$E780_H$ ..... $E7CF_H$
$EF80_H$ ..... $EFCF_H$
$F780_H$ ..... $F7CF_H$
$FF80_H$ ..... $FFCF_H$

==========

Capítulo Siete

Aviso Sensato

Un rasgo realmente excitante e innovador del ordenador Amstrad es la manera en que ha sido diseñado para permitir que el programador tenga un fácil acceso para modificar el **sistema operativo.** La **entrada** en el sistema es a través de una serie de **vectores de salto,** colocados en forma de tabla o ristra en la parte alta de la memoria a partir de la dirección $B900_H$.

Este **bloque vectorial de saltos** contiene una serie consecutiva de vectores que señalan a las rutinas más útiles contenidas en la memoria de únicamente lectura donde está grabado el sistema operativo. La parte inferior de la memoria de únicamente lectura (ROM), como ya hemos presentado, es la responsable de controlar el **hardware** del sistema y las subrutinas relacionadas. Usando estos vectores, tan amablemente suministrados por los cerebros que trabajan en el **software** de Amstrad, podemos aprovechar las rutinas contenidas en la memoria de únicamente lectura, en lugar de tener que escribir centenares de líneas en código máquina por nuestros medios. Además, y dado que esos bloques vectoriales de saltos están contenidos en la **memoria de escritura-lectura,** podemos hacer fácilmente leves alteraciones, o modificar las rutinas de una manera más drástica, mediante el cambio del punto de destino señalado por los vectores, haciendo que se dirijan a direcciones distintas.

Para sacar ventaja completa de estas rutinas, es aconsejable comprar una copia del libro del Amstrad **"The Complete CPC 464 Operating System"** (Soft 158). El libro es caro pero contiene una enorme cantidad de información sobre el sistema operativo, y virtualmente es la 'biblia' del programador con Amstrad. Hay más de 189 puntos diferentes de entrada en el bloque vectorial de saltos, y desde luego cae fuera del ámbito de este libro tratar con cada elemento del bloque individualmente.

Para demostrar la utilidad de este bloque vectorial, escribamos una pequeña rutina para exponer un mensaje en pantalla. No es una actividad muy emprendedora, yo también estoy de acuerdo; pero ilustrará lo fácil que es utilizar las rutinas imbuidas en la memoria de únicamente lectura del Amstrad.

```
                ORG   0A000H
SITCUR:         EQU   0BB75H
EXPO:           EQU   0BB5DH
;
INICI:          LD    HL,0104H
                CALL  SITCUR
                LD    HL,MENSA
OTRA:           LD    A,(HL)
                CP    0FFH
                JR    Z,FINAL
                PUSH  HL
                CALL  EXPO
                INC   HL
                JR    OTRA
FINAL:          RET
MENSA:          DB    "ESTO ES DE PRUEBA
```

El programa **apela** a dos de las rutinas internas cuyo punto de entrada está reseñado en el bloque vectorial de saltos: las de direcciones $BB75_H$ y $BB5D_H$. Nada más comenzar el programa se dan etiquetas a estas dos direcciones con las direcciones actuales **igualadas** inmediatamente después de iniciarse nuestra rutina.

SITCUR $BB75_H$

La rutina citada mediante esta dirección **sitúa** el cursor en una nueva posición, tal y como esté especificada en la pareja de registros HL. El registro H debe contener la **columna** requerida, y el registro L la **fila** requerida.

EXPO $BB5D_H$

La rutina citada con esta dirección **expone** el carácter contenido en ese momento en el registro A, en la posición **corriente** del cursor.

Nuestro **programa** comienza cargando la pareja de registros HL con la posición deseada para el cursor, que es en este caso la columna 1 y la fila 4. Luego, la pareja de registros HL se carga de manera que señale a la dirección donde comienza el 'buzón' del mensaje (LD HL,MENSA), y se prepara un bucle etiquetado como OTRA para, reiteradamente, cargar el registro A con el carácter señalado en cada momento por la pareja de registros HL.

Se efectúa a continuación la comprobación de si el contenido del acumulador es igual al valor FF_H -que actúa como octeto **testigo** de final de mensaje. Hemos usado un método similar en Basic para detectar el final de un censo de datos constantes. Si el carácter no es el FF_H, entonces se apela a la rutina EXPO para proyectarlo en pantalla. El curso del programa da luego un salto al comienzo de OTRA ronda para examinar el siguiente carácter del mensaje. Una vez que se encuentra la marca FF_H la rutina termina regresando al programa **apelante.**

LISTADO 5

El programa ensamblado

```
 1                          ORG  0A000H
 2              SITCUR:     EQU  0BB75H
 3              EXPO:       EQU  0BB5DH
 4              ;
 5 A000 210401  INICI:      LD   HL,0104H
 6 A003 CD75BB              CALL SITCUR
 7 A006 2116A0              LD   HL,MENSA
 8 A009 7E      OTRA:       LD   A,(HL)
 9 A00A FEFF                CP   0FFH
10 A00C 2807                JR   Z,FINAL
11 A00E E5                  PUSH HL
12 A00F CD5DBB              CALL EXPO
13 A012 23                  INC  HL
14 A013 18F4                JR   OTRA
15 A015 C9      FINAL:      RET
16 A016 4553544F MENSA:     DB   "ESTO ES DE PRUEBA
16 A01A 20455320
16 A01E 44452050
16 A022 52554542
16 A026 41
17 A027 FF                  DB   0FFH
18                          END
```

Una vez que hemos **ensamblado** el programa, podemos **implantarlo** en memoria de diversas maneras:

a) Cargándolo en celdillas de memoria protegidas (por encima de la 'cima' de memoria usable) y **apelando** a él desde un programa en Basic.

b) Reflejando el código máquina como valor de una tabla o ringla en Basic, y haciendo que se **meta** en determinadas celdillas protegidas de la memoria, luego se apela a él desde el programa en Basic.

c) Asignándolo como valor **literal**, o sarta de caracteres en el programa, y apelando a él mencionando simplemente la dirección correspondiente a dicha variable literal.

d) Si el 'código objeto' del programa es **re-ubicable** podemos cargarlo en una tabla o ringla del Basic (tal y como se describió en el Capítulo 5), y hacer que se ejecute mencionando la dirección de esa tabla.

En esta ocasión, vamos a colocar las instrucciones en código máquina como un censo de datos constantes del programa, y vamos a hacer que se metan en celdillas de memoria; luego apelaremos a la rutina desde Basic:

```
 10 CLS
 20 FOR I = &A000 TO &A027
 30  READ DTA
 40  POKE I,DTA
 50  NEXT
 60 CALL &A000H
 70 FOR I = 1 TO 500:NEXT
 80 CLS:GOTO 60

 90 DATA &21,&04,&01,&CD,&75,&BB,&21,&16,&A0
100 DATA &7E,&FE,&FF,&28,&07,&E5,&CD,&5D,&BB
110 DATA &23,&18,&F4,&C9,&45,&53,&54,&4F,&20
120 DATA &45,&53,&20,&44,&45,&20,&50,&52,&55
130 DATA &45,&42,&41,&FF
```

Ahora que ya hemos tenido una muestra de la forma en que puede sacarse ventaja del bloque vectorial de saltos del sistema, pasemos a hacer algo realmente más útil, y al mismo tiempo confeccionar una subrutina que nos permita dar a nuestros programas mayor atractivo.

SONIDO

No es nuestra intención profundizar en las rutinas de sonido incorporadas en el sistema, dado que están adecuadamente tratadas en el **Concise Basic Manual** publicado por Amsoft. Vamos a diseñar nuestras propias rutinas de sonido e **implementar** nuevos comandos en Basic que nos permitan sacar ventaja completa de dichas subrutinas.

Podría ser bonito tener una manera menos complicada de producir sonidos en el ordenador. El programa que vamos a confeccionar nos dará esa facilidad. Cuando hayamos finalizado con él, seremos capaces de usar tres nuevos comandos:

|QSON |TOQU **y** |SON

Para escribir este programa debemos primero comprender cómo el sistema utiliza el circuito integrado (el 'chip') con el que genera el sonido, y cómo opera dicho circuito AY8912 Programmable Sound Generator. Obviamente, sin conocer esta información nos sería imposible confeccionar una rutina efectiva que nos sirviera de **nexo** con el hardware.

AY8912 Programmable Sound Generator (PSG)

Dicho PSG es un circuito con integración en gran escala (LSI) que puede producir un variado surtido de sonidos bajo control del software. Una vez que se ha enviado al circuito los rasgos de la nota sonora a emitir, el ordenador queda libre para realizar otras tareas, hasta que llegue el momento en que los sonidos o los registros de ese circuito, necesiten actualización.

El circuito usa tres canales de sonido, designados por A, B, C, y gobernables de manera independiente, que pueden emitir **tonos puros** o **ruido blanco.** La respuesta en frecuencia de dicho PSG varía desde los tonos **pre-audibles** en las frecuencias más bajas, hasta los tonos **post-audibles** en las frecuencias más altas.

Algunos sonidos puede que sobrepasen los límites de reproducción que tiene el amplificador de sonido incorporado en el Amstrad, y puede que sólo se hagan patentes cuando el sistema esté conectado a un sistema de alta fidelidad sonora externo.

Los bloques básicos de este circuito 'producidor' de imagen sonora, son como sigue:

Generador de tono Produce para cada canal las señales de frecuencia sonora pertinente.

Generador de ruido Produce ruido, es decir, una señal aleatoriamente modulada en frecuencia.

Mezcladores Estos bloques circuitales combinan las salidas de los generadores de tono con la del generador de ruido, y hay un bloque mezclador para cada canal.

Amplificador de amplitud Proporciona un nivel constante de sonido o un sonido de volumen o amplitud variable. En este caso el nivel sonoro queda controlado por el Generador de Envolvente.

Generador de Envolvente Produce un **perfil** o forma de la envolvente de volumen, que puede aprovecharse para modular en amplitud la salida de cada uno de los mezcladores.

Las diversas operaciones posibles por este circuito productor de sonido, están controladas por **14 registros** (los designados como R_0 a R_{13}, de los 16 registros de que consta el generador), y sus funciones se enuncian a continuación:

LOS REGISTROS SONICOS DEL PSG

R_0 **Valor alto para la nota del canal A**
R_1 **Valor bajo para la nota del canal A**
R_2 **Valor alto para la nota del canal B**
R_3 **Valor bajo para la nota del canal B**
R_4 **Valor alto para la nota del canal C**
R_5 **Valor bajo para la nota del canal C**

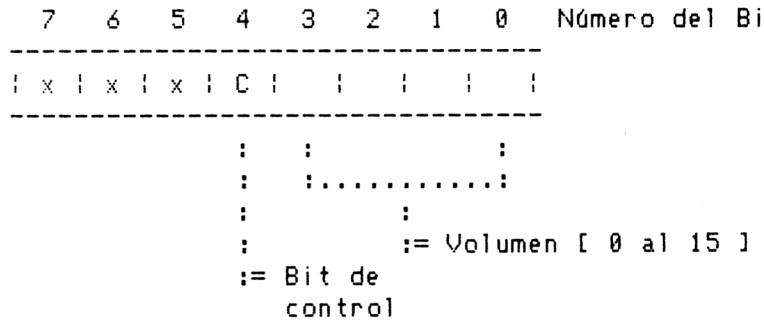
R_6 **Valor para el ruido generado**
R_7 **Canal mezclador -Tono puesto/quitado; Ruido puesto/quitado; portIN/EX**

R_8 **Volumen para canal A**
R_9 **Volumen para canal B**
R_{10} **Volumen para canal C**

R_{11} **Valor alto del período de envolvente**
R_{12} **Valor bajo del período de envolvente**
R_{13} **Forma/ciclo de envolvente.**

REGISTROS 8 - 9 - 10

Estos registros controlan el volumen del sonido emitido dependiendo de cuáles son los bits alzados (valor 1). El gobierno del volumen por cada canal es mediante sólo 5 bits (del 0 al 4) y los restantes 3 bits del octeto no son tenidos en cuenta por el PSG.



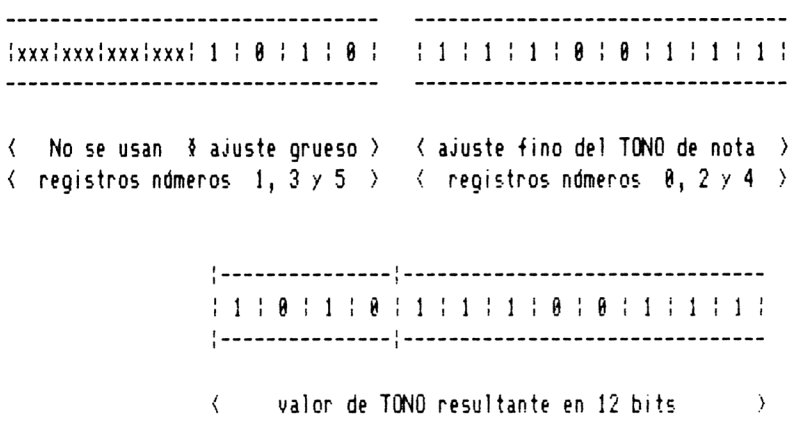
Si el **bit 4** está alzado, entonces el control se cede al generador de envolvente que dará un nivel variable de sonido de acuerdo con los diversos perfiles de envolvente.

Cuando el **bit 4** está bajado (valor 0), el control del volumen se efectúa mediante el valor reflejado en los registros 8, 9 y 10, respectivamente. Todo lo que eso significa es que puedes traspasar un valor del 0 al 15, y el nivel de volumen es constante y acorde con dicho valor: con el 0 siendo el mínimo y el 15 el máximo de amplitud. Cuando el valor en esos registros es de 16, el volumen de la nota es modulado en amplitud bajo control del registro 13, ya que el valor 16 alza el bit 4 (valor 1).

REGISTROS 0,1; 2,3; 4,5

Estos registros controlan el **tono** de la nota sonora producida. En cada pareja, el registro más bajo utiliza sus 8 bits para gobernar la 'sintonía fina' (correspondiendo su valor al **período** del tono emitido, por lo que usando sólo el registro inferior se producen las notas de alta frecuencia o **tono agudo**). El registro alto de cada pareja usa los cuatro bits numerados del 0 al 3, para controlar la **sintonía gruesa** (que corresponde a las notas de baja frecuencia o **tonos graves**).

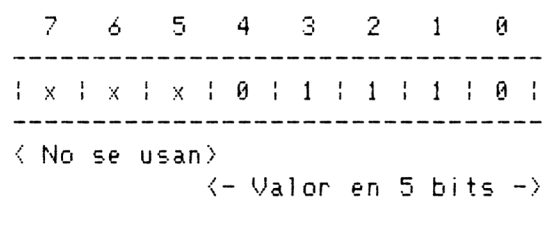
Cuando se combinan los valores de los dos registros, el resultado es un valor de 12 bits y la nota producida corresponde a las enumeradas en el Manual Basic del Amstrad para ese valor.



El valor combinado en cada pareja de registros es equivalente al valor de la nota usado en el comando Basic.

REGISTRO 6

Este registro 6 trabaja de manera similar a los registros anteriores, pero lo que hace es gobernar el ruido generado. El registro sólo usa 5 bits (del 0 al 4) y cuanto mayor sea el valor en el registro, más baja es la frecuencia media resultante del ruido blanco generado. La banda de valores que pueden inscribirse en este registro es pues de 0 a 31_D.



REGISTRO 7

En cuanto nos concierne ahora, éste es el registro más importante que nos conviene dominar, ya que este registro combina el ruido blanco y los tonos puros generados mediante los registros 8, 9 y 10 para cada uno de los tres canales respectivamente.

Este registro también está calibrado en binario, y si estudias la tabla 7.1 fácilmente comprenderás cómo los diferentes valores afectan a la nota sonora finalmente emitida por los canales A, B y C.

```
  7   6   5   4   3   2   1   0   Número del Bit
---------------------------------
| x | x | 1 | 0 | 1 | 1 | 1 | 0 |
---------------------------------

<faculta%  cancela  %  cancela  >
 accesos     ruido       tono

---------------------------------
| x | x | C | B | A | C | B | A |
---------------------------------

<--- acciones del registro 7 --->

Bit 'alzado' [1] === QuiTadA acción
Bit 'bajado' [0] === PuesTA  acción
```

TABLA 7.1

acciones	canal	bit 5 4 3	bit 2 1 0	canal	acciones
ruido PTO	A B C	0 0 0	0 0 0	A B C	PTO tono
ruido PTO	- B C	0 0 1	0 0 1	- B C	PTO tono
ruido PTO	A - C	0 1 0	0 1 0	A - C	PTO tono
ruido PTO	- - C	0 1 1	0 1 1	- - C	PTO tono
ruido PTO	A B -	1 0 0	1 0 0	A B -	PTO tono
ruido PTO	- B -	1 0 1	1 0 1	- B -	PTO tono
ruido PTO	A - -	1 1 0	1 1 0	A - -	PTO tono
ruido QTO	A B C	1 1 1	1 1 1	A B C	QTO tono

Para facultar la mezcla del ruido blanco sobre el canal B y los tonos puros sobre los canales A y C, por ejemplo, necesitarás examinar las tablas y combinar los valores necesarios, e.g. $101010_B = 42_D = 2A_H$.

Los bits 6 y 7 controlan los dos registros utilizados como portIN/EX de acceso dentro del sistema. Estos registros son usados por el ordenador para la exploración del teclado y no nos conciernen en cuanto al sonido. El valor que puede ser INgresado/EXpedido a través de esta vía de acceso está en la banda de 0 a 255_D, y recuerda que el bit debe estar bajado (valor 0) para facultar el acceso IN/EX a través de dichos registros.

REGISTRO 13

Este registro usa 4 bits, del 0 al 3. El registro sólo afecta al sonido global producido cuando se coloca el valor 16 en los registros 8, 9 ó 10 de control de amplitud. Cualquier otro valor inferior a 16 no permitirá que la amplitud del volumen sea controlada por el registro 13.

```
  7   6   5   4   3   2   1   0   Número del Bit
---------------------------------
| x | x | x | x | 1 | 1 | 1 | 0 |
---------------------------------
                  :   :   :   := Mantenga
                  :   :   := Alterne
                  :   := 'Ataque
                  := Continue
```

MANTENimiento

Cuando el bit 0 está alzado el perfil de envolvente está limitado a un ciclo, ya que se mantlene la cuenta más reciente.

ALTERNamiento

Si el bit 1 está alzado, entonces el contador de envolvente invierte la dirección de cuenta después de cada ciclo.

Ataque

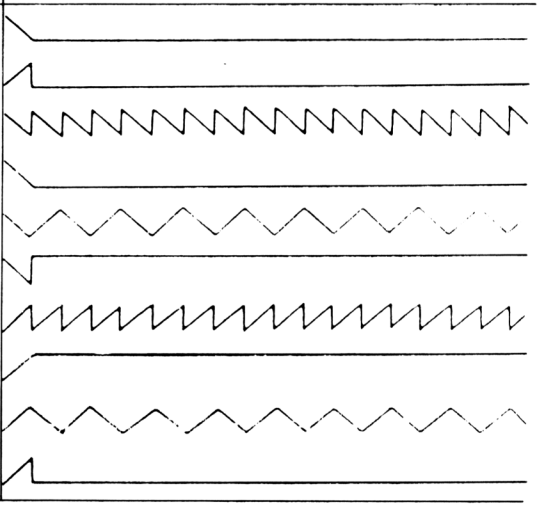
Cuando el bit 2 está alzado, la cuenta es ascendente (ataque) mientras que es descendente en los demás casos.

Continuación

Cuando el bit 3 está alzado, la pauta de ciclos es la definida por el bit de mantenimiento; mientras que si está bajado el generador de envolvente continuará al nivel más bajo.

Pero es más fácil observar el **perfil** de envolvente generado de acuerdo con esta tabla:

CONTINUE	ATTACK	ALTERNATE	HOLD
0	0	X	X
0	1	X	X
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1



REGISTROS 11 y 12

El control del **período** de la envolvente es por medio de estos dos registros. Combinadamente, los dos registros usan 16 bits (doble octeto) y pueden tener por tanto un valor en la banda 0 a 65535_D. El registro 12 controla el **valor de ajuste** (notas de alta frecuencia) y el registro 11 controla el **valor aproximado** del período de la envolvente (notas de baja frecuencia).

En síntesis, estos dos registros controlan la duración del intervalo durante el cual la nota se 'sostiene' o 'decae' cuando está bajo control del generador de envolvente. Utilizando con pericia estos dos registros, se pueden producir algunos efectos sonoros interesantes, que pueden ir desde el rebote de una pelota de ping-pong hasta el bramar de un coche de carreras.

AÑADIENDO TRES NUEVOS COMANDOS DE SONIDO

Después de que haya sido instalado en el ordenador el programa enunciado en el **Listado Uno de Ensamblaje,** el Basic será capaz de responder a los tres nuevos comandos introducidos: **|TOQU, |SON** y **|QSON.**

FORMATO SINTACTICO DE |TOQU

```
|TOQU, <canal>, <octava>, <tono>, <volumen>

        Canal   = 1 al 3
        Octava  = 1 al 8
        Tono    = 1 al 12
        Volumen = 1 al 15

    DO mayor = C media en octava 4
```

NOTAS:

0 = pausa	4 = F	8 = C agudo
1 = C	5 = G	9 = D agudo
2 = D	6 = A	10 = F agudo
3 = E	7 = B	11 = G agudo
		12 = A agudo

FORMATO SINTACTICO PARA |SON

```
|SON, <núm.ID del registro>, <valor a cargar>

        Número ID del registro = 0 al 13
              Valor a cargar = 0 al 255
```

|QSON Quita la posibilidad de emitir sonido.

```
|QSON
```

```
 1                ;*******************************
 2                ;LISTADO 1 DE ENSAMBLAJE
 3                ;*******************************
 4                ;
 5                ;
 6                          ORG  42000
 7                          LOAD 42000
 8                ENVIE:    EQU  0BD34H          ;a rgto sonoro
 9                FUERA:    EQU  0BCA7H          ;claree rgtos
10 A410 011AA4              LD   BC,TABCOM       ;'bc=sede tabla
11 A413 21BEA4              LD   HL,BUCHE        ;exigido por Basic
12 A416 CDD1BC              CALL 0BCD1H          ;a poner a$adidos
13 A419 C9                  RET                  ;reflejadas claves,vuelva
14                              ;donde vino
15 A41A 25A4      TABCOM:   DEFW VERBOS
16 A41C C331A4              JP   EMITA           ;a cargar rgto.
17 A41F C347A4              JP   TOQUE           ;a sacar nota sonora
18 A422 C3C3A4              JP   QUITE           ;a clausurar canal
19                ;*******************************
20                ;Definidor tabla de nuevos
21                ;comandos para que Basic sepa
22                ;reconocerlas desde ahora
23                ;*******************************
24                ;
25 A425 534F      VERBOS:   DEFB 'SO'
26 A427 CE                  DB   'N'+80H
27 A428 544F51              DB   'TOQ'
28 A42B D5                  DB   'U'+80H
29 A42C 51534F              DB   'QSO'
30 A42F CE                  DB   'N'+80H
31 A430 00                  DB   00
32                ;
33                ;
34                ;*******************************
35                ;EMITA hacia el generador de
36                ;sonido un valor carg ndolo
37                ;directamente en un Re6isTro.
38                ;*******************************
39                ;
40                ;
41 A431 DD7E02    EMITA:    LD   A,(IX+02H)      ;'A=num_R6To
42 A434 F5                  PUSH AF              ;apile eso en percha
43 A435 FE07                CP   07H             ;compare 'A con eso
44 A437 DD7E00              LD   A,(IX+00H)      ;'A=valor_a_emitir
45 A43A 2002                JR   NZ,NYLIA        ;si distinto de Zero
46 A43C E63F                AND  3FH             ;baje bit 6(10111111)
```

```
47 A43E 5F       NYLIA:      LD   E,A             ;'e='A(val_a_emitir
48 A43F F1                   POP  AF              ;saque eso de percha
49 A44@ FE@E                 CP   @EH             ;el 13 es el maximo
5@                         ;compare con max+1
51 A442 3@4B                 JR   NC,RESTO        ;si >=14 vaya ah!
52 A444 C39DA4               JP   EXGR2           ;a emitir segundoVALor
53               ;
54               ;
55               ;*******************************
56               ;TOCAdor de nota sonora segun
57               ;canal/octava/tono/volumen
58               ;*******************************
59               ;
6@               ;
61 A447 DD7E@6   TOQUE:      LD   A,(IX+@6H)      ;'A=numcanal
62 A44A F5                   PUSH AF              ;apile eso en percha
63 A44B DD7E@4               LD   A,(IX+@4H)      ;'A=numoctava
64 A44E F5                   PUSH AF              ;apile eso en percha
65 A44F DD7E@2               LD   A,(IX+@2H)      ;'A=num_tono
66 A452 B7                   OR   A               ;comprueba si es cero
67 A453 283A                 JR   Z,RESTO         ;si lo es salte ah!
68 A455 CB27                 SLA  A               ;multiplique por 2
69 A457 5F                   LD   E,A             ;'e='A
7@ A458 16@@                 LD   D,@@H           ;'de=tono a tocar
71 A45A 21A2A4               LD   HL,NOTAS        ;'hl=apunta a sede
72 A45D 19                   ADD  HL,DE           ;'hl=direcci'n enTABLA
73                         ;de nota a tocar
74 A45E 5E                   LD   E,(HL)          ;'e=vod nota s/ tabla
75 A45F 23                   INC  HL              ;siguiente direcci'n
76 A46@ 56                   LD   D,(HL)          ;'d=voz nota s/ tabla
77                       ;'de=valor tono exacto
78 A461 C1                   POP  BC              ;saque eso de percha
79                       ;valor de la octava
8@ A462 @E@@                 LD   C,@@H
81 A464 @5                   DEC  B
82 A465 28@6                 JR   Z,EVITE         ;si 'b=Zero salte ah!
83 A467 CB3A     JUSTE:      SRL  D               ;desplace izquierda
84                         ;de acuerdo octava
85                         ;corregida
86 A469 CB1B                 RR   E
87 A46B 1@FA                 DJNZ JUSTE           ;repita otro ajuste
88                        ;'b='b-1 y si 'b<>@
89 A46D F1       EVITE:      POP  AF              ;recupere canal
9@ A46E F5                   PUSH AF              ;deposite otra vez
91 A46F CB27                 SLA  A               ;desplace izquierda
92                     ;si canal=1 multiplique
93                     ;*2 = registro 2 para
94                     ;ajuste grueso de tono
```

```
 95 A471 3D                 DEC  A              ;corrigiendo a 1 menos
 96 A472 CD96A4             CALL EX6R1          ;valor primero
 97 A475 DD7E00             LD   A,(IX+00H)     ;'A=volumen
 98 A478 FE10               CP   10H            ;el 16 es el maximo
 99 A47A 3802               JR   C,BUENO        ;si hubo aCarreo
100 A47C 3E10               LD   A,10H          ;forzado a vol.max.
101 A47E 5F       BUENO:    LD   E,A            ;'e='A(volumen)
102 A47F D5                 PUSH DE             ;apile en percha
103 A480 1E38               LD   E,38H          ;'e=00111000 para
104 A482 3E07               LD   A,7H           ;'A=00000111 sonoro
105 A484 CD9DA4             CALL EX6R2          ;valor segundo
106 A487 D1                 POP  DE             ;recupere volumen
107 A488 F1                 POP  AF             ;recupere canal
108 A489 C607     PREPA:    ADD  A,7            ;ajuste para
109                          ;R6Tos 8/9/10
110 A48B CD9DA4             CALL EX6R2          ;valor segundo
111 A48E C9                 RET                 ;vuelva al Basic
112 A48F F1       RESTO:    POP  AF
113 A490 F1                 POP  AF
114 A491 1E00               LD   E,00
115 A493 C389A4             JP   PREPA
116 A496 6F       EX6R1:    LD   L,A
117 A497 4A                 LD   C,D
118 A498 CD34BD             CALL ENVIE
119 A49B 2D                 DEC  L
120 A49C 7D                 LD   A,L
121 A49D 4B       EX6R2:    LD   C,E
122 A49E CD34BD             CALL ENVIE
123 A4A1 C9                 RET
124               ;********* Tabla de TONOS
125 A4A2 0000     NOTAS:    DEFW 0000H
126 A4A4 EE0E               DEFW 0EEEH
127 A4A6 4D0D               DEFW 0D4DH
128 A4A8 DA0B               DEFW 0BDAH
129 A4AA 2F0B               DEFW 0B2FH
130 A4AC F709               DEFW 09F7H
131 A4AE E108               DEFW 08E1H
132 A4B0 E907               DEFW 07E9H
133 A4B2 180E               DEFW 0E18H
134 A4B4 8E0C               DEFW 0C8EH
135 A4B6 8F0A               DEFW 0A8FH
136 A4B8 6809               DEFW 0968H
137 A4BA 6108               DEFW 0861H
138 A4BC 0000               DEFW 0000H
139               ;
140               ;
141               BUCHE:    DEFS 4
```

```
142                ;
143                ;
144 A4C2 00                    NOP
145 A4C3 CDA7BC    QUITE:      CALL FUERA
146 A4C6 C9                    RET
147 A4C7 00                    NOP
148                            END
```

Si usamos el código objeto tal y como está enunciado podemos simplemente cargarlo en celdillas protegidas de la memoria, luego acudir a la dirección 42000_D, y a partir de eso usar los nuevos comandos desde dentro de nuestros programas Basic. Hay sin embargo, una desventaja usando este método: cada vez que deseemos usar los comandos con un programa en Basic, tendremos que volver a reimplantar la rutina en la parte protegida de la memoria como un programa separado, lo que no es una manera muy satisfactoria de aprovechar el programa.

El método más útil de emplear el programa es crear un **cargador** en Basic transformando todo el código máquina en constantes DATA y **metiéndolas** directamente en memoria, con el comando **POKE**. Si usamos este método, cualquier programa futuro puede sacar ventaja de este utensilio para sonido, **mezclando** su programa con el cargador Basic.

Para crear un censo de constantes DATA nos basta mirar el **código objeto** que aparece en la tercera columna del listado, y colocar **cada pareja** de cifras decimohexales como un solo dato constante:

```
10  DATA  &01,&1A,&A4,&21,&BE,&A4
```

El programa **apela** a tres subrutinas incorporadas en el sistema operativo acudiendo a las tres direcciones del bloque vectorial de saltos: **$BD34_H$, $BCA7_H$ y $BCD1_H$**. La más interesante de estas citas es la de dirección **$BCD1_H$**. El Basic permite ampliaciones a su repertorio de comandos Basic mencionándolas en la forma **Comando, <parámetro>, <parámetro> ,...**

Un nuevo comando debe llevar siempre como prefijo el símbolo barra vertical (|) que es como las dos rayitas verticales colocadas encima de la arroba (@) en el teclado. Puedes comprobarlo tecleando|**TOQU** y concluyendo con **ENTER**, con lo que aparecerá un mensaje de error sintáctico. Pero si ahora tecleas |**TOQU** y concluyes el comando con **ENTER**, el ordenador responderá con un mensaje de error que indica "comando desconocido".

Para asegurarte que el ordenador no rechaza nuestros nuevos comandos, debemos informar al sistema operativo **residente** que pretendemos usar estas nuevas **palabras clave,** y le indicamos eso **reseñando** las nuevas palabras vía la subrutina citada mediante la dirección $\mathbf{BCD1_H}$.

Para **reseñar** una nueva palabra clave, la primera parte del programa debe tener la siguiente forma:

```
LD BC,TABCOM    ;DIREccion sede de la Tabla de 'saltos'
                 a las rutinas ejecutoras de los comandos
LD HL,BUCHE     ;DIREccion del deposito intermediario
                 de 4 octetos requerido por el Sistema
CALL 0BCD1H     ;apelación al subrutina del Sistema que
                 registra los nuevos comandos declarados
```

Nuestra tabla de saltos debe prepararse como sigue:

```
TABCOM:   DEFW VERBOS     ;DIREcción sede de la Tabla
                           donde se declaran los nombres
                           clave que escoges a discreción
          JP donde comience la rutina ejecutora 1
          JP donde comience la rutina ejecutora 2
          ...
          JP donde comience la rutina ejecutora última
```

Añadiendo $\mathbf{80_H}$ al último carácter de cada palabra clave, hace que el bit más significativo (el bit 7), del carácter quede **alzado;** i.e. Ascii **Y** + $\mathbf{80_H}$ = $\mathbf{D9_H}$. Este **convenio** es requerido en el Amstrad de manera que el sistema operativo residente pueda reconocer el final de cada nueva palabra clave.

```
VERBOS:   DEFM "SO"
          DEFB "N" + 80H

          DEFM "TOQ"
          DEFB "U" + 80H

          ...
          DEFB 00H        ;marca de final de la Tabla
```

Las ampliaciones de comandos Basic también pueden **percibir** parámetros que maticen la acción que el comando lleva a cabo. Son traspasados a la rutina que cumplimenta el comando aprovechando el registro **IX** del microprocesador Z80, en la forma:

```
(IX + 00H) apunta hacia celda con último parámetro
(IX + nnH) apunta hacia celda con primer parámetro


|TOQU, <parám1>, <parám2>, <parám3>, <parám4>
              :          :          :          :
   (IX +03) =:          :          :          :
             (IX +02) =:          :          :
                       (IX +01) =:          :
                                 (IX +00) =:
```

Y eso es todo lo que hay que hacer. Puedes añadir tantos nuevos comandos como desees dentro de la capacidad de almacenamiento de la memoria.

Nota: La primera línea del programa en Basic debe tener una instrucción para establecer la **cima** de la memoria reservada al Basic, un octeto por debajo de donde comienza la rutina en código máquina. La rutina de sonido ha usado la dirección $A410_H$ (42000_D) como dirección de comienzo y eso nos permite suficiente espacio como para mover la tabla de matrices patrones desde la memoria de únicamente lectura hasta la memoria de escritura lectura, con el comando **SYMBOL AFTER.**

LISTADO 6

El listado 6 es el programa cargador en Basic y puede mezclarse con cualquier programa que desees. Una vez se haya ejecutado este programa cargador puedes usar los nuevos comandos en el **Modo Directo** para experimentar con la generación de sonido.

```
10 CLS
20 FOR I = &A410 TO &A4C7
30 READ DTA
40 POKE I,DTA
50 NEXT
60 GOTO 290
```

```
 70 DATA &01,&1A,&A4,&21,&BE,&A4,&CD,&D1,&BC
 80 DATA &C9,&25,&A4,&C3,&31,&A4,&C3,&47,&A4
 90 DATA &C3,&C3,&A4,&53,&4F,&CE,&54,&4F,&51
100 DATA &D5,&51,&53,&4F,&CE,&00,&DD,&7E,&02
110 DATA &F5,&FE,&07,&DD,&7E,&00,&20,&02,&E6
120 DATA &3F,&5F,&F1,&FE,&0E,&30,&4B,&C3,&9D,&A4
130 DATA &DD,&7E,&06,&F5,&DD,&7E,&04,&F5,&DD
140 DATA &7E,&02,&B7,&28,&3A,&CB,&27,&5F,&16
150 DATA &00,&21,&A2,&A4,&19,&5E,&23,&56,&C1
160 DATA &0E,&00,&05,&28,&06,&CB,&3A,&CB,&1B
1D0 DATA &10,&FA,&F1,&F5,&CB,&27,&3D,&CD,&96
180 DATA &A4,&DD,&7E,&00,&FE,&10,&38,&02,&3E
190 DATA &10,&5F,&D5,&1E,&38,&3E,&07,&CD,&9D
200 DATA &A4,&D1,&F1,&C6,&07,&CD,&9D,&A4,&C9
210 DATA &F1,&F1,&1E,&00,&C3,&89,&A4,&6F,&4A
220 DATA &CD,&34,&BD,&2D,&7D,&4B,&CD,&34,&BD
230 DATA &C9,&00,&00,&EE,&0E,&4D,&0D,&DA,&0B
240 DATA &2F,&0B,&F7,&09,&E1,&08,&E9,&07,&18
250 DATA &0E,&8E,&0C,&8F,&0A,&68,&09,&61,&08
260 DATA &00,&00,&00,&00,&00,&00,&00,&CD,&A7
270 DATA &BC,&C9,&00
280 '****************************************
290 'EL RESTO DEL PROGRAMA PUEDE EMPEZAR AQUI
```

PROGRAMAS DE DEMOSTRACION

Ahora que ya tienes tus tres nuevos comandos, intenta teclear los siguientes breves programas y verás rápidamente lo versátiles que son y cómo mejorarán tus programas futuros.

Demostración Uno

```
10   FOR I = 48 TO 192
20    |SON,7,254
30    |SON,8,20
40    |SON,0,I
50    NEXT
60   |SON,6,0
70   |SON,7,7
80   |SON,8,30
90   |SON,9,30
100  |SON,10,30
110  |SON,12,56
120  |SON,13,0
130  FOR I = 1 TO 1000:NEXT
140  |QSON
150  GOTO 120
```

Demostración Dos

```
10   INPUT "PULSA CUALQUIER TECLA PARA DETONACION"
20   !SON,6,0
30   !SON,7,7
40   !SON,8,16
50   !SON,9,16
60   !SON,12,56
80   !SON,13,0
90   GOTO 10
```

Demostración Tres

```
10   INPUT "PULSA CUAQUIER TECLA PARA DISPARO"
20   !SON,6,15
30   !SON,7,7
40   !SON,8,16
50   !SON,9,16
60   !SON,10,16
70   !SON,12,16
80   !SON,13,0
90   GOTO 10
```

Capítulo Ocho

Figuras Animadas en el AMSTRAD

EL AMSTRAD NO TIENE POSIBILIDAD DE TRABAJAR CON FIGURAS ANIMADAS (que aunque no tienen nada de 'espíritus' sí que se llaman también 'sprites'). En el momento en que hayas acabado con este capítulo, esta declaración que hacemos ya no será cierta. El siguiente programa te permite crear hasta 8 figuras animadas y examinar lo que hay expuesto en la pantalla del Amstrad.

Una figura animada debe ser capaz de moverse por toda la pantalla en todas las direcciones sin destruir la escena de **fondo** -moviéndose sobre el fondo en lugar de hacerlo en el fondo. Además, como es casi imposible observar mediante la memoria lo que hay expuesto en la pantalla, debemos incluir en nuestra ampliación Basic una rutina que solvente este problema de manera que puedan detectarse las colisiones entre figuras animadas.

El Listado 2 de Ensamblaje está incluido en el libro de manera que puedes ver cómo se ha diseñado el programa. Intenta seguirlo minuciosamente, y te dará algunas buenas ideas para incorporar tus propios comandos.

El método de usar el programa es como antes: teclea el **Listado 7** y guárdalo en cinta cassette. Ahora teclea el **Listado 8,** guárdalo y luego ejecuta el programa. El módulo de demostración te mostrará lo que es posible lograr con estas rutinas: no significa que sea una pieza modelo de programación, y estoy seguro que serás capaz de mejorarla; lo que constituye la verdadera razón de haberlo incluido.

LISTADO 7

```
  2 'FIGuRAs o eFIGIEs con animación (Sprites)
  3 'CARGADOR de rutinas en código máquina
  5 SYMBOL AFTER 32
 10 MEMORY &9C39
 20 FOR DIRE = &9C40 TO &9E19
```

```
 30  READ VL:POKE DIRE,VL
 40  NEXT
 50 CALL &9C40
 60 RUN "

100 DATA &01,&49,&9C,&21,&AE,&9D,&CD,&D1,&BC
110 DATA &54,&9C,&C3,&63,&9C,&C3,&5E,&9D,&C3,&8E,&9D
120 DATA &50,&55,&CA,&44,&45,&46,&49,&C7,&48,&41,&59
130 DATA &CF,&00,&00,&00,&DD,&7E,&02,&32,&B2,&9D,&CD,&4D,&9D
140 DATA &FD,&7E,&00,&FE,&FF,&C8,&DD,&7E,&00,&FE,&00
150 DATA &CA,&F4,&9D,&FE,&09,&D0,&FD,&77,&02,&CD,&78,&BB
160 DATA &22,&AB,&9D,&FD,&6E,&00,&FD,&66,&01,&E5,&CD,&75,&BB
170 DATA &FD,&7E,&04,&FE,&00,&20,&18,&3E,&FE,&FD,&77,&04
180 DATA &CD,&9F,&BB,&CD,&93,&BB,&32,&AD,&9D,&FD,&7E,&03
190 DATA &CD,&90,&BB
191 DATA              &E1,&C3,&1D,&9D,&FD,&7E,&05,&CD,&5D,&BB
200 DATA &3E,&FE,&CD,&9F,&BB,&CD,&93,&BB,&32,&AD,&9D
210 DATA &FD,&7E,&03,&CD,&90,&BB,&E1,&FD,&7E,&02,&FE,&01
220 DATA &28,&1D,&FE,&02,&28,&25,&FE,&03,&28,&1B,&FE,&04
230 DATA &28,&25,&FE,&05,&28,&10,&FE,&06,&28,&21,&FE,&07
240 DATA &28,&0E,&FE,&08,&28,&11,&C9,&25,&18,&17,&24
250 DATA &18,&14,&2D,&18,&11,&2C,&18,&0E,&2D,&25,&18,&0A
260 DATA &2C,&25,&18,&06,&2D,&24,&18,&02,&2C,&24,&7C,&FE,&01
270 DATA &38,&22,&FE,&29,&30,&1E,&7D,&FE,&01,&38,&19
280 DATA &FE,&1A,&30,&15,&FD,&75,&00,&FD,&74,&01
290 DATA &FD,&66,&01,&FD,&6E,&00,&CD,&75,&BB,&CD,&60,&BB
300 DATA &FD,&77,&05,&FD,&66,&01,&FD,&6E,&00,&CD,&75,&BB
310 DATA &FD,&7E,&06,&CD,&5D,&BB,&2A,&AB,&9D,&CD,&75,&BB
320 DATA &3E,&00,&CD,&9F,&BB,&3A,&AD,&9D,&CD,&90,&BB
330 DATA &C9,&3A,&B2,&9D,&3D,&07,&07,&07,&4F,&06,&00
340 DATA &FD,&21,&B3,&9D,&FD,&09,&C9,&DD,&7E,&08,&FE,&09
350 DATA &D0,&32,&B2,&9D,&CD,&4D,&9D,&DD,&7E,&02,&FD,&77
360 DATA &00,&DD,&7E,&04,&FD,&77,&01,&DD,&7E,&00
370 DATA &FD,&77,&03,&DD,&7E,&06,&FD,&77,&06,&3A,&B2,&9D
380 DATA &FD,&77,&07,&3E,&00,&FD,&77,&04,&C9,&DD,&7E,&00
390 DATA &6F,&DD,&7E,&02,&67,&7C,&FE,&01,&D8,&FE,&29,&D0
400 DATA &7D,&FE,&01,&D8
401 DATA                  &CD,&75,&BB,&CD,&60,&BB
410 DATA &32,&F3,&9D,&C9,&00,&00,&00,&00,&00,&00,&00
420 DATA &00,00,00,00,00,00,00,00,00
430 DATA 00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00
440 DATA 00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00
450 DATA 00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00
460 DATA 00,00,00,00,00,00,00,00,00
470 DATA &CD,&78,&BB,&22,&AB,&9D,&FD,&6E,&00,&FD,&66,&01
480 DATA &CD,&75,&BB,&FD,&7E,&05,&CD,&5D,&BB,&3E,&FF
490 DATA &FD,&77,&00,&AF,&FD,&77,&01,&2A,&AB,&9D,&CD,&75
500 DATA &BB,&C9,&00
```

LISTADO 8

```
  1 'FIGuRAs o eFIGIEs con animación (Sprites)
  2 'DEMOSTRACION de empleo
  3 'PREPARADOR DE PARAMETROS ************
 10 DEFINT A - Z: CLS:BORDER 9,9: ADD = &9DB3 :'Comienzo
    de tabla
 11 INK 3,17
 20 LET PK = &9DF3
 30 FOR I =0 TO 56 STEP 8
 40  POKE ADD+I,&FF
 41  POKE ADD+I+4,0
 42  NEXT
 43 LET BUL =0:LET INVB =0
 49 'DEFINIDOR DE CARACTERES **************
 50 SYMBOL 249,&81,&42,&3C,&5A,&66,&3C,&42,&81
 60 SYMBOL 250,&3C,&3C,&3C,&3C,&7E,&7E,&FF,&FF
 70 SYMBOL 251,&10,&10,&00,&10,&10,&00,&10,&10
 80 SYMBOL 252,&10,&10,&08,&08,&10,&10,&08,&08
 85 PEN 1: FOR I = 3 TO 23
 86  FOR J = 1 TO 3:LET X = INT(RND*39+1)
 87   LOCATE X,I
 88   PRINT "."
 89   NEXT:NEXT
 90 |DEFIG,1,249,10,2,3
100 |DEFIG,2,249,16,2,3
110 |DEFIG,3,249,23,2,3
120 |DEFIG,4,249,30,2,3
130 |DEFIG,5,250,20,24,2
140 FOR I = 1 TO 5:|PUJ,I,1:NEXT 'Mostrador de ca\ón e
    invasores
149 'Bucle del programa principal **************
160 POKE PK,0:LET FL = INT(RND*9)
165 IF FL = 2 THEN GOSUB 3000: GOTO 190
170 IF FL = 5 THEN GOSUB 4000
190 IF INKEY(8) = 0 THEN GOTO 230
200 IF INKEY(1) = 0 THEN GOTO 250
210 IF BUL <> 0 THEN GOTO 1000
220 IF INKEY(47) = 0 THEN GOTO 2000
225 GOTO 1000
230 |PUJ,5,1:GOTO 1000
250 |PUJ,5,5:GOTO 1000
1000 IF INVB <> 0 THEN GOTO 1500
1010 LET B = INT(RND*4+1)
1020 LET B=B-1:LET B=ADD+(B*8)+1 :'Ajusta para ver tabla
1030 LET A=PEEK(B) ' Posición horizontal de invasores
1040 LET A = A+1: |DEFIG,6,252,A,3,1: |PUJ,6,7
```

```
1049 'Comprueba la bala humana
1050 LET INVB = 1: GOTO 1800
1500 LET CHK=PEEK(&9DDB):LET CHK1=PEEK(&9DDC):
     IF CHK>=25 THEN |PUJ,6,0:LET INVB =0: GOTO 1800
1505 LET CHK = CHK+1
1510 |HAYO,CHK1,CHK: IF PEEK(PK)=250 THEN GOTO 1520
                        ELSE IF PEEK(PK)=251 GOTO 1515
1511 |PUJ,6,7: GOTO 1800
1515 |PUJ,6,0: |PUJ,7,0: LET INVB=0:BUL=0: GOTO 160
1520 |PUJ,6,0: |PUJ,5,0
1530 LET INVB =0:BUL=0: |PUJ,7,0
1540 LET LVE = LVE - 1: IF LVE =0 THEN GOTO 5000
1550 FOR I=1 TO 500:NEXT
1560 |DEFIG,5,250,24,20,2
1570 GOTO 160
1800 LET POSL =PEEK(&9DE3): LET POSH=PEEK(&9DE4):
     IF POSL = 1 THEN LET BUL = 0: |PUJ,7,0: GOTO 160
1805 LET POSL = POSL -1
1810 |HAYO,POSH,POSL
1820 IF PEEK(PK)=249 THEN GOTO 1830
        ELSE IF PEEK(PK)=252 THEN GOTO 1825
1821 |PUJ,7,3: GOTO 160
1825 |PUJ,7,0: |PUJ,6,0: LET BUL=0:INVB=0: GOTO 160
1830 FOR I = 0 TO 40 STEP 8
1840  IF PEEK(ADD+I+1) <> PEEK(&9DE4) THEN NEXT:GOTO 160
1850  LET TMP = ADD+I+7
1870  LET INV =PEEK(TMP)
1880  |PUJ,7,3
1900  |PUJ,7,0
1910  |PUJ,INV,0
1920  LET SCRE=SCRE+20
1930  IF SCRE=80 THEN GOTO 5000
1940  LET BUL=0: GOTO 160
2000  LET BUL =1: LET X=0
2010  LET X=0
2020  LET X=PEEK(&9DD3)-1: LET Y=PEEK(&9DD4)
2030  |DEFIG,7,251,Y,X,2: |PUJ,7,3
2040  GOTO 160
3000  FOR I= 1 TO 4
3010   |PUJ,I,5
3020   NEXT
3030  RETURN
4000  FOR I= 4 TO 1 STEP -1
4010  |PUJ,I,1
4020  NEXT
4030 RETURN
5000 STOP
```

Comentarios sobre el programa y los nuevos comandos:

|PUJ

Permitirá que puedas mover, o empujar la figura especificada como primer parámetro del comando, en cualquier dirección tal y como se mencione en el segundo parámetro:

```
|PUJ, <número ID de FIGuRA>, <rumbo>
```

El número identificativo de la figura puede ser de 1 a 8, y la dirección o 'rumbo' puede también ser de 1 a 8, tal y como se indica en la tabla 8.1. **Nota:** si se coloca un cero como valor del parámetro Dirección, hará que la figura desaparezca de la pantalla, y a continuación se inscribirá el valor &FF (marca de figura no creada) y 00 (testigo de primera entrada en la rutina), en la tabla de atributos de las figuras. Debes entonces volver a generar la figura antes de que puedas proyectarla en pantalla. Esto es de ayuda cuando se detecta una colisión con otro objeto en pantalla y deseas dejar en blanco la figura animada y la escena de fondo tal y como estaba.

|DEFIG

Para describir una figura debes usar este comando. La rutina encargada de la cumplimentación del comando almacenará la información en el elemento homólogo de la tabla de atributos de figuras. La sintaxis para este comando es:

```
|DEFIG, <núm. ID de FIGuRA>, <núm. ID símbolo>,
        <posición X>, <posición Y>, <color>
```

Este comando debe usarse para inscribir cada figura antes de poder cursar el comando **|PUJ** para proyectarla en pantalla. La figura puede ahora ser mostrada en pantalla en la **posición** especificada mediante los parámetros de este comando DEFIG, si se cursa además el comando **|PUJ, nº identificativo de figura ,1** correspondiente.

!HAYO

Este comando examina lo que **hay** expuesto en pantalla y coloca el valor correspondiente en la dirección $9DF3_H$. Usa la siguiente sintaxis cuando curses este comando:

```
!HAYO, <posición X>, <posición Y>
```

Una línea típica en Basic puede ser entonces:

```
10 !HAYO,3,5: LET PK = PEEK(&9DF3)
```

También pueden usarse variables como parámetros de los comandos, e.g.

```
!HAYO,X,Y    y también  !DEFIG,3,249,X,Y,3
```

Esta sección de código puede separarse y usarse como un utensilio de programación para entregar el valor que **haya observado** en pantalla, con cualquier otro programa en Basic.

TABLA DE ATRIBUTOS DE FIGURAS

La tabla con los rasgos característicos de cada figura tiene su **base** en la dirección $9DB3_H$. El método de hallar la **cota** o desplazamiento, dentro de la tabla para una figura dada, será como sigue:

Base de la Tabla+(Número identificativo figura - 1)*8:

Si deseas por ejemplo comprobar en qué dirección se está 'empujando' a la figura número 3, debes usar la siguiente expresión:

&9DB3+((3-1)*8+2)

para encontrar la **cota** adecuada de ese elemento en la tabla.

Los elementos de la tabla de atributos de las figuras están ordenados así:

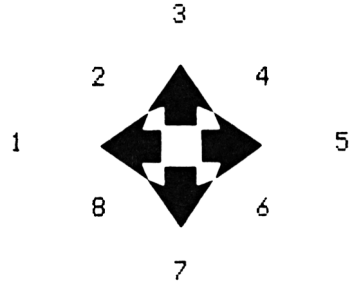
Octeto 1 **Posición Y**
Octeto 2 **Posición X**
Octeto 3 **Dirección**
Octeto 4 **Tinta**
Octeto 5 **Banderín testigo**
Octeto 6 **Carácter en Basic**
Octeto 7 **Patrón de forma de figura**
Octeto 8 **Número identificativo de figura.**

Una observación que hay que hacer es que cuando dos figuras animadas chocan ha de borrarse primero la última de las figuras usando el comando **|PUJ** seguido por la figura con la que ha ocurrido la colisión.

El **Listado 7** se preocupa de la mayoría de las situaciones que pudieran surgir en tu programación. Tecleándolo y luego estudiando cómo funciona, adquirirás una perspectiva de trabajo sobre cómo aprovechar estos comandos. El listado fuente puede usarse como una matriz o molde, con la que puedes crear tus propias rutinas, yo estaría interesado en saber de cualquier usuario que haya encontrado otros comandos para los que haya apreciado trabajos útiles.

Tabla 8.1

Movimiento de la figura según la dirección:



De manera que |PUJ,1,5 'empujará' la figura número 1 hacia la izquierda.

```
 1              ;*******************************
 2              ;LISTADO 2 ENSAMBLAJE
 3              ;*******************************
 4              ;
 5              ;
 6                             ORG  4@@@@
 7                             LOAD 4@@@@
 8 9C4@ @1499C                 LD   BC,TABCOM
 9 9C43 21B19D                 LD   HL,BUCHE
1@ 9C46 CDD1BC-                CALL @BCD1H            ;a$adir nueva clave
11             ;
12 9C49 549C    TABCOM:       DEFW CLAVES
13 9C4B C3639C                JP   PORTE             ;FIGuRA a pantalla
14 9C4E C35E9D                JP   TRACE             ;FIGuRA mota a mota
15 9C51 C38E9D                JP   VISTO             ;en pantalla
16             ;
17 9C54 5@55    CLAVES:       DEFB 'PU'
18 9C56 CA                    DEFB 'J'+8@H
19 9C57 44454649              DEFB 'DEFI'
2@ 9C5B C7                    DEFB '6'+8@H
21 9C5C 484159                DEFB 'HAY'
22 9C5F CF                    DEFB 'O'+8@H
23 9C6@ @@                    DEFB @@H
24                            DEFS @2H               ;postizo de relleno
25             ;
26 9C63 DD7E@2  PORTE:        LD   A,(IX+@2H)        ;numID FIGRA
27 9C66 32B59D                LD   (NUMID),A
28 9C69 CD4D9D                CALL DELTA             ;cota dentro de tabla
29             ;entrega en 'IX el puntero sede
3@ 9C6C FD7E@@                LD   A,(IY+@@)         ;posici'n verti
31 9C6F FEFF                  CP   @FFH
32 9C71 C8                    RET  Z                 ;regrese si figra no hech
33 9C72 DD7E@@                LD   A,(IX+@@H)        ;DIREcci'n PORTOR
34 9C75 FE@@                  CP   @@H               ;'@@ es borrar figra
35 9C77 CAF79D                JP   Z,BORRE
36 9C7A FE@9                  CP   @9H               ;el 8 es el maximo
37 9C7C D@                    RET  NC                ;regrese si hay mas
38 9C7D FD77@2                LD   (IY+@2H),A        ;endemas inscrito
39 9C8@ CD78BB                CALL @BB78H            ;cursor (d'nde?
4@ 9C83 22AE9D                LD   (VIECUR),HL       ;conservado ah!
41 9C86 FD6E@@                LD   L,(IY+@@H)        ;'l=posici'n Y
42 9C89 FD66@1                LD   H,(IY+@1H)        ;'h=posici'n Y
43 9C8C E5                    PUSH HL                ;deposite en percha
44 9C8D CD75BB                CALL @BB75H            ;situar cursor
45 9C9@ FD7E@4                LD   A,(IY+@4H)        ;'A=testigo tabla
46 9C93 FE@@                  CP   @@H
47 9C95 2@18                  JR   NZ,SALTE          ;si<>@ figra corrida
48 9C97 3EFE                  LD   A,@FEH            ;atestigua entrada RUT
49 9C99 FD77@4                LD   (IY+@4H),A        ;ina transparencia
```

```
5@ 9C9C CD9FBB            CALL @BB9FH          ;fondo no-escribible
51 9C9F CD93BB            CALL @BB93H          ;mirar tinta pluma
52 9CA2 32B@9D            LD   (TINPLU),A      ;archivado ah!
53 9CA5 FD7E@3            LD   A,(IY+@3H)      ;'A=pluma deseada
54 9CA8 CD9@BB            CALL @BB9@H          ;fije tinta pluma
55 9CAB E1                POP  HL              ;recupere pos_cursor
56 9CAC C31D9D            JP   RE_INI
57 9CAF FD7E@5   SALTE:   LD   A,(IY+@5H)      ;'A=carANTES
58 9CB2 CD5@BB            CALL @BB5@H          ;re-EXponga en panta
59 9CB5 3EFE              LD   A,@FEH          ;testigo transparencia
6@ 9CB7 CD9FBB            CALL @BB9FH          ;fondo no escribible
61 9CBA CD93BB            CALL @BB93H          ;mirar tinta pluma
62 9CBD 32B@9D            LD   (TINPLU),A      ; archivado ah!
63 9CC@ FD7E@3            LD   A,(IY+@3H)      ;tinta deseada
64 9CC3 CD9@BB            CALL @BB9@H          ;fije tinta pluma
65 9CC6 E1                POP  HL              ;recupere cursor posi.
66 9CC7 FD7E@2            LD   A,(IY+@2H)      ;direcci'n figra
67 9CCA FE@1              CP   @1H
68 9CCC 281D              JR   Z,ZURDA
69 9CCE FE@2              CP   @2H             ;izquierda y arriba?
7@ 9CD@ 2825              JR   Z,ZDIAS
71 9CD2 FE@3              CP   @3
72 9CD4 281B              JR   Z,SUBA
73 9CD6 FE@4              CP   @4H             ;derecha y arriba?
74 9CD8 2825              JR   Z,DDIAS
75 9CDA FE@5              CP   @5H             ;derecha?
76 9CDC 281@              JR   Z,DECHA
77 9CDE FE@6              CP   @6H             ;derecha y abajo?
78 9CE@ 2821              JR   Z,DDIAB
79 9CE2 FE@7              CP   @7H             ;abajo?
8@ 9CE4 28@E              JR   Z, BAJE
81 9CE6 FE@8              CP   @8H             ;izquierda y abajo?
82 9CE8 2811              JR   Z,ZDIAB
83 9CEA C9                RET                  ;regrese si no es ninguno
84 9CEB 25       ZURDA:   DEC  H
85 9CEC 1817              JR   RESTOR          ;columna menos 1
86 9CEE 24       DECHA:   INC  H
87 9CEF 1814              JR   RESTOR          ;columna mas 1
88 9CF1 2D       SUBA:    DEC  L
89 9CF2 1811              JR   RESTOR          ; fila menos 1
9@ 9CF4 2C       BAJE:    INC  L
91 9CF5 18@E              JR   RESTOR          ;fila mas 1
92 9CF7 2D       ZDIAS:   DEC  L
93 9CF8 25                DEC  H
94 9CF9 18@A              JR   RESTOR
95 9CFB 2C       ZDIAB:   INC  L
96 9CFC 25                DEC  H
97 9CFD 18@6              JR   RESTOR
```

```
 98 9CFF 2D       DDIAS:      DEC  L
 99 9D00 24                   INC  H
100 9D01 1802                 JR   RESTOR
101 9D03 2C       DDIAB:      INC  L
102 9D04 24                   INC  H
103 9D05 7C       RESTOR:     LD   A,H              ;coja posici'n X
104 9D06 FE01                 CP   1                ;primera X
105 9D08 3822                 JR   C,REINT2         ;salte si menor
106 9D0A FE29                 CP   41               ;ultima pos X + 1
107 9D0C 301E                 JR   NC,REINT2        ;salte si mayor
108 9D0E 7D                   LD   A,L              ;lo mismo para pos Y
109 9D0F FE01                 CP   1
110 9D11 3819                 JR   C,REINT2
111 9D13 FE1A                 CP   26
112 9D15 3015                 JR   NC,REINT2
113 9D17 FD7500               LD   (IY+00H),L       ;refleje nueva Y
114 9D1A FD7401               LD   (IY+01H),H       ;refleje nueva X
115 9D1D FD6E00   RE_INI:     LD   L,(IY+00H)       ;nuevacoorY
116 9D20 FD6601               LD   H,(IY+01H)       ;nuevacoorX
117 9D23 CD75BB               CALL 0BB75H           ;situar cursor
118 9D26 CD60BB               CALL 0BB60H           ;examinar caracter
119 9D29 FD7705               LD   (IY+05H),A       ;refleje en tabla
120 9D2C FD6E00   REINT2:     LD   L,(IY+00H)
121 9D2F FD6601               LD   H,(IY+01H)
122 9D32 CD75BB               CALL 0BB75H
123 9D35 FD7E06               LD   A,(IY+06H)       ;simbolo de figra
124 9D38 CD5DBB               CALL 0BB5DH           ;EXponga en pantalla
125 9D3B 2AAE9D               LD   HL,(VIECUR)      ;vieja posici'n
126 9D3E CD75BB               CALL 0BB75H
127 9D41 3E00                 LD   A,00H            ;testigo opacidad
128 9D43 CD9FBB               CALL 0BB9FH           ;fondo si-escribible
129 9D46 3AB09D               LD   A,(TINPLU)       ;tinta de la pluma
130 9D49 CD90BB               CALL 0BB90H           ;fije tinta de pluma
131 9D4C C9                   RET                   ;vuelva donde vino
132               ;
133               ;
134 9D4D 3AB59D   DELTA:      LD   A,(NUMID)        ;'A=num FIGRA
135 9D50 3D                   DEC  A                ;ajuste en menos uno
136 9D51 07                   RLCA                  ;*2 rotando izquierda
137 9D52 07                   RLCA                  ;*2=*4
138 9D53 07                   RLCA                  ;*2=*8
139 9D54 4F                   LD   C,A
140 9D55 0600                 LD   B,00H
141 9D57 FD21B69D             LD   IY,TBLFGR
142 9D5B FD09                 ADD  IY,BC            ;'IY=entrada en tabla
143 9D5D C9                   RET
144               ;
145               ;
```

```
146 9D5E DD7E08   TRACE:     LD   A,(IX+08H)      ;numID FIGRA
147 9D61 FE09                CP   09H             ;el 8 es el maximo
148 9D63 D0                  RET  NC              ;regrese si es mayor
149 9D64 32B59D              LD   (NUMID),A       ;reflejado p/ DELTA
150 9D67 CD4D9D              CALL DELTA           ;hallar desplazamto.
151 9D6A DD7E02              LD   A,(IX+02H)      ;nueva posici'n X
152 9D6D FD7700              LD   (IY+00H),A      ;refleje en tabla
153 9D70 DD7E04              LD   A,(IX+04H)      ;nueva posici'n Y
154 9D73 FD7701              LD   (IY+01H),A      ;refleje en tabla
155 9D76 DD7E00              LD   A,(IX+00H)      ;nuevo color
156 9D79 FD7703              LD   (IY+03H),A
157 9D7C DD7E06              LD   A,(IX+06H)      ;forma modeloFIGRA
158 9D7F FD7706              LD   (IY+06H),A
159 9D82 3AB59D              LD   A,(NUMID)
160 9D85 FD7707              LD   (IY+07H),A
161 9D88 3E00                LD   A,00H           ;testigo de figra
162 9D8A FD7704              LD   (IY+04),A
163 9D8D C9                  RET                  ;vuelva al principal
164               ;
165               ;
166 9D8E DD7E00   VISTO:     LD   A,(IX+00H)      ;coord X
167 9D91 6F                  LD   L,A
168 9D92 DD7E02              LD   A,(IX+02H)      ;coord Y
169 9D95 67                  LD   H,A
170 9D96 7C                  LD   A,H             ;rellenando postizos
171 9D97 FE01                CP   01H
172 9D99 D8                  RET  C
173 9D9A FE29                CP   41              ;lo hemos visto antes ya
174 9D9C D0                  RET  NC
175 9D9D 7D                  LD   A,L
176 9D9E FE01                CP   01H
177 9DA0 D8                  RET  C
178 9DA1 FE1A                CP   26
179 9DA3 D0                  RET  NC
180 9DA4 CD75BB              CALL 0BB75H          ;re-EXponga caracter
181 9DA7 CD60BB              CALL 0BB60H          ;examina pantalla
182 9DAA 32F69D              LD   (MIRIL),A       ;conserve eso ah!
183 9DAD C9                  RET                  ;y esto es todo,compadre.
184               ;
185               ;
186 9DAE 0000     VIECUR:    DEFW 0000
187 9DB0 00       TINPLU:    DEFB 00H
188               BUCHE:     DEFS 04H             ;exigido porBasic
189 9DB5 00       NUMID:     DEFB 00
190               TBLFGR:    DEFS 40H
191 9DF6 00       MIRIL:     DEFB 00H
192 9DF7 CD78BB   BORRE:     CALL 0BB78H          ;cursor(d'nde?
193 9DFA 22AE9D              LD   (VIECUR),HL     ;archivela ah!
```

```
194 9DFD FD6E00          LD   L,(IY+00H)
195 9E00 FD6601          LD   H,(IY+01H)
196 9E03 CD75BB          CALL 0BB75H           ;situar cursor
197 9E06 FD7E05          LD   A,(IY+05H)       ;caracter en Basic
198 9E09 CD5DBB          CALL 0BB5DH           ;re-EXponga en panta
199 9E0C 3EFF            LD   A,0FFH           ;testigo aun no-hecho
200 9E0E FD7700          LD   (IY+00H),A       ;incluido en tabla
201 9E11 AF              XOR  A                ;'A=00000000 y testigos=0
202 9E12 FD7704          LD   (IY+04),A  <---  ;marque en la tabla
203 9E15 2AAE9D          LD   HL,(VIECUR)
204 9E18 CD75BB          CALL 0BB75H           ;situar viejo cursor
205 9E1B C9              RET
206                      END
```

Si queremos sacar ventaja al hecho de poder crear nuevos comandos e incorporarlos al repertorio Basic, podemos añadir útiles rutinas que se adecúen a nuestra fantasía. Se pueden añadir nuevas rutinas a un **utensilio de programación** ya existente simplemente incluyendo los nuevos verbos y nombres en la **tabla de claves,** luego las direcciones en la **tabla de saltos,** y empalmando el nuevo listado de las subrutinas al final del programa ya existente.

Algunas versiones de Basic permiten el comando **PRINT @,nn** siendo **nn** la dirección en pantalla **en** la que queremos exponer un dato. Ese comando proporciona una manera fácil de establecer el 'formato' de exposición en pantalla cuando se están usando montones de texto. El **Listado 3 de Ensamble** provee esta facilidad para el CPC464. Este programa también incorpora otros dos útiles comandos, para observar lo que hay expuesto en pantalla y para poder meter directamente un valor en la memoria o 'pizarra', correspondiente a la pantalla.

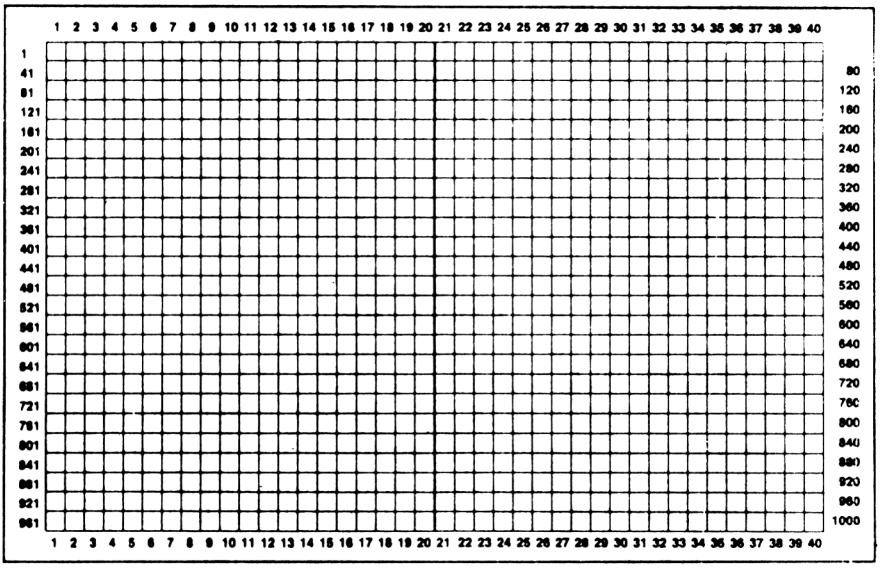
LISTADO 3 DE ENSAMBLAJE

```
1                       ORG  4000
2
3               ;
4               ;
5               ;*******************************
6               ;Rutina para a$adir al Basic
7               ;3 nuevos comandos
8               ;Sintaxis:
9               ;|VPOKE,<DIREcci'n),<OCTEto>
10              ;|VPEEK,<DIREcci'n),@VAY
11              ;|.,<POSIci'n>
12              ;'@VAY' es donde se entrega al
13              ;Basic lo que HAY en pantalla
14              ;*******************************
15              ;
16              ;
17              ;*******************************
18              ;Preparador de nuevos comandos
19              ;Registrador en tabla del Basic
20              ;*******************************
21 9C40 0145A9C             LD   BC,TBL_VCTR       ;'BC=sede tabla
22                             ;nuevas rutinas
23 9C43 21C49C              LD   HL,BUCHE          ;Exigido por Basic
24 9C46 CDD1BC              CALL #BCD1H            ;a poner a$adidos
25 9C49 C9                  RET                    ;vuelva donde vino
```

```
26                ;
27                ;
28                ;*******************************
29                ;Prepare ahora TaBLa VeCToRes
3Ø                ;*******************************
31                ;
32 9C4A 559C      TBL_VCTR:   DEFW TBL_VERB
33 9C4C C3869C                JP   WOLSE
34 9C4F C3619C                JP   WMIR
35 9C52 C3A59C                JP   EN
36                ;
37                ;*******************************
38                ;Definidor claves a$adidas
39                ;*******************************
4Ø 9C55 565Ø4F4B TBL_VERB:   DB   'VPOK','E'+8ØH
4Ø 9C59 C5
41 9C5A 565Ø4545              DB   'VPEE','E'+8ØH
41 9C5E C5
42 9C5F AE                    DB   '.'+8ØH
43 9C6Ø ØØ                    DB   ØØ               ;Marca fin de tabla
44                ;
45                ;
46                ;*******************************
47                ;Observador de lo que HAY
48                ;*******************************
49                ;
5Ø                ;
51 9C61 DD6EØØ    WMIR:       LD   L,(IX+ØØH)       ;'l=vod RESUL
52 9C64 DD66Ø1                LD   H,(IX+Ø1H)       ;'h=voz RESUL
53 9C67 E5                    PUSH HL               ;percha='h'l
54 9C68 CD78BB                CALL ØBB78H           ;cursor (d'nde?
55 9C6B 22C89C                LD   (CURSOR),HL      ;deposite v_posi
56 9C6E DD6EØ2                LD   L,(IX+Ø2)        ;'l=vod DIREmirar
57 9C71 DD66Ø3                LD   H,(IX+Ø3)        ;'h=voz DIREmirar
58 9C74 CDB29C                CALL DIVIDA           ;calcular pos(x,y)
59 9C77 CD75BB                CALL ØBB75H           ;situar cursor
6Ø 9C7A CD6ØBB                CALL ØBB6ØH           ;examinar pantalla
61 9C7D E1                    POP  HL               ;'hl=percha(DIREvisto)
62 9C7E 77                    LD   (HL),A           ;meta en ese sitio
63 9C7F 2AC89C                LD   HL,(CURSOR)      ;recupere v_posi
64 9C82 CD75BB                CALL ØBB75H           ;re-situar cursor
65 9C85 C9                    RET                   ;vuelva donde vino
66                ;
67                ;
68                ;*******************************
69                ;emBOLSAdor en Video-direcci'n
7Ø                ;*******************************
71                ;
```

```
                ;
9C86 CD78BB     WOLSE:      CALL @BB78H         ;cursor(d'nde?
9C89 22C89C                 LD   (CURSOR),HL    ;deposite v_posi
9C8C DD6E02                 LD   L,(IX+02H)     ;'l=vod DIREmeter
9C8F DD6603                 LD   H,(IX+03H)     ;'h=voz DIREmeter
9C92 CDB29C                 CALL DIVIDA         ;calcular pos(x,y)
9C95 CD75BB                 CALL @BB75H         ;situar cursor
9C98 DD7E00                 LD   A,(IX+00H)     ;'A=valor_a_meter
9C9B CD5ABB                 CALL @BB5AH         ;emBOLSE segun ('hl
9C9E 2AC89C                 LD   HL,(CURSOR)    ;recupere v_posi
9CA1 CD75BB                 CALL @BB75H         ;re-situar cursor
9CA4 C9                     RET                 ;vuelva donde vino
                ;
                ;
                ;*******************************
                ;Colocador de cursor ENpantalla
                ;*******************************
                ;
                ;
9CA5 DD6E00     EN:         LD   L,(IX+00H)     ;'l=voz posi
9CA8 DD6601                 LD   H,(IX+01H)     ;'h=vod posi
9CAB CDB29C                 CALL DIVIDA         ;calcular pos(x,y)
9CAE CD75BB                 CALL @BB75H         ;situar cursor
9CB1 C9                     RET                 ;vuelva donde vino
                ;
                ;
                ;*******************************
                ;DIVIDAdor segun anchura imagen
                ;percibe en 'hl el dividendo
                ;entrega en 'l=coorY,'h=coorX
                ;*******************************
9CB2 112800     DIVIDA:     LD   DE,40          ;numero de COLUs
9CB5 0601                   LD   B,1            ;'b=inicio eje Y
9CB7 B7         RONDA:      OR   A              ;clarea testigos
9CB8 ED52                   SBC  HL,DE          ;dividir es restar
9CBA FAC09C                 JP   M,SALGA        ;siMenos0 en cociente
9CBD 04                     INC  B              ;baje coorY una posici'n
9CBE 18F7                   JR   RONDA          ;restar OTRA vez
9CC0 19         SALGA:      ADD  HL,DE          ;corregir UN 40
9CC1 65                     LD   H,L            ;'h=coordenada X
9CC2 68                     LD   L,B            ;'l=coordenada Y
9CC3 C9                     RET                 ;vuelva donde vino
                ;
                ;
                ;
                BUCHE:      DS   4              ;exigido por Basic
9CC8 0000       CURSOR:     DEFW 0000
9CCA 00                     NOP
                            END
```

La pantalla del Amstrad dispone de 1000 cuadratines para exponer un carácter (25 filas por 40 columnas), que normalmente será señalado mediante el comando para que **sitúe** el cursor en una columna, fila dada, de clave **LOCATE.** Una vez que este nuevo programa haya sido instalado, cada cuadratín de exposición puede ser accesado simulando el comando **PRINT@.** La sintaxis para nuestro nuevo comando es: **|.,<dirección>** y una línea en Basic sería similar a esta: **10 |.,41:?"Así se usa el nuevo comando para la posición 41."** Cuando se usa este nuevo comando, la pantalla presenta la siguiente **gradilla:**



Para exponer un mensaje que comience a partir de la posición 41, la instrucción Basic sería:

10 |.,41:?"Esto comienza en la posición 41".

La dirección dada en este nuevo comando puede ser un número en la banda de 1 a 1000, o una variable numeral que te permitirá dar forma al texto de una manera pulcra. Las siguientes instrucciones son todo lo que se necesita para llevar la cuenta de la posición corriente del cursor, y controlar fácilmente las zonas de exposición en pantalla.

CR = Posición corriente del cursor.

```
SUBA CURSOR          CR = CR + 40 * (CR-40)0)
BAJE CURSOR          CR = CR - 40 * (CR+40(1000)
AVANCE CURSOR        CR = CR - (CR+1(1000)
REtroCEDA CURSOR     CR = CR + (CR-1)0)

SUBA A 'TOPE
(misma COLUmna)      CR = CR - INT(CR/40)*40
BAJE A 'BASE
(misma COLUmna)      CR = CR - INT(CR/40)*40+960

REtroCEDA CURSOR
A INICIO DE ESA FILA      CR = INT(CR/40)*40
A INICIO DE FILA PREVIA   CR = -((CR(40)*CR)
                               -(CR(=40)*(INT(CR/40)*40-40)
AVANCE CURSOR
A INICIO DE OTRA FILA     CR = -((CR)=960)*CR)
                               -(CR(960)*(INT(CR/40)*40+40)
```

```
10 CLS:LET CR = 1
20 !.,CR: PRINT "FILA  1, COLUMNA 1"
40 LET CR = CR -INT(CR/40)*40+960
50 !.,CR: PRINT "FILA 25, COLUMNA 1"
```

Una vez que hayas agarrado esta idea, el uso de este método de conformar el texto en pantalla es muy superior al que se logra usando la instrucción para situar cursor, de clave **LOCATE;** ya que siempre puedes comprobar el cursor y desplazarlo hasta la posición que desees usando las fórmulas anteriores.

|VPOKE, <DIRECCION>,Valor

El parámetro **Valor** ha de estar en la banda 0 a 255 y corresponde a los caracteres de control ASCII, a los caracteres normales y a los símbolos gráficos.

Este nuevo comando te permite **meter** directamente valores en una dirección concreta de la pantalla. Este método de establecer una correspondencia o proyección con la pantalla es exactamente como el usado en el comando anterior, por lo que las direcciones deben estar en la banda 1 a 1000. Para obtener una visión sobre las ventajas de este comando, intenta la siguiente demostración.

```
10 CLS
20 LOCATE 25,40:PRINT "K";
30 GOTO 30
```

Y AHORA ENSAYA ESTE OTRO:

```
10 CLS
20 !VPOKE,1000,122
30 GOTO 30
```

¿Observaste la diferencia? La imagen en pantalla no se **corre** (no se 'desrrolla'). Con eso se consigue una manera útil de escribir en pantalla sin hacer que la imagen se desplace hacia arriba.

```
10 CLS
20 !.,1: PRINT "TEST"
30 !VPOKE,1000,122
40 PRINT " VPOKE"
50 GOTO 50
```

Observa en esta última demostración que la posición del cursor permanecía donde se expuso por última vez, incluso aunque metiéramos el dato en esa misma dirección de pantalla.

|VPEEK,<dirección>,@PK

Este comando funciona de manera similar al comando | POKE que hemos usado en un capítulo anterior, ya que te permite observar lo que hay expuesto en pantalla, en cualquier dirección dada del 1 al 1000. El valor observado es transferido como contenido de la variable denominada **PK.**

```
10 DEFINT P:LET PK=0
20 CLS
30 !.,500: PRINT "XYZ"
40 !VPEEK,500,@PK
50 PRINT PK
60 !VPEEK,503,@PK
70 PRINT PK
```

Cuando se usa este nuevo comando, la variable **PK** debe haber sido asignada previamente, con la categoría de numeral entero, y al comienzo del programa o como mínimo antes de haber usado este nuevo comando |VPEEK.

El listado de ensamble

El programa comienza dando las condiciones iniciales de los nuevos comandos y luego poniéndose en contacto con el interpretador Basic. Una vez que esto haya sido hecho, el programa regresa de nuevo a Basic y las rutinas permanecen 'dormidas' hasta que sean citadas en un programa Basic. El comando |VPEEK usa dos parámetros, el de dirección y la variable PK. Cuando entra en acción esta sección de código, los parámetros son traspasados con el registro **IX** señalando al último de ellos, que en este caso es @**PK**. Cada parámetro usa dos octetos, de manera que (IX+00) con (IX+01) contienen el valor correspondiente del último parámetro (@PK); mientras que (IX+02) con (IX+03) contienen el valor dado al parámetro de dirección. Una observación importante a hacer con esta subrutina es que siempre que se traspasa a un programa una variable en la forma @ <nombre de variable> es la **dirección sede de la variable** lo que se traspasa, pero no el valor de la variable. Eso significa que una vez que la rutina de ampliación haya **percibido** la dirección sede de la variable puede cambiar el valor en esa dirección contenido, de acuerdo con las circunstancias; y así al regresar al Basic, la variable tendrá asignado el nuevo valor.

Una vez que la rutina disponga de la dirección en pantalla que se quiere observar, guarda la posición corriente del cursor, luego sitúa el cursor en la posición dada, recurriendo a la rutina de dividir para convertir esa dirección en posiciones X,Y en pantalla. Se efectúa la lectura de esa posición y el valor del carácter que se encuentre se coloca en la dirección de memoria correspondiente a la variable PK. Antes de que la rutina devuelva el control al Basic, se restaura la posición original del cursor.

La rutina cumplimentadora del nuevo comando |VPOKE funciona exactamente de la misma manera que la de |VPEEK, exceptuando que ahora es el carácter dado el que se estampa en pantalla. El carácter percibido por la rutina puede ser cualquier código ASCII o gráfico válido en la banda 0 a 255.

La subrutina de dividir es útil para relacionarse con otros programas y una explicación de cómo trabaja te permitirá apreciar la manera en que puedes aprovecharla dentro de tus propios programas.

Esta subrutina de dividir funciona mediante **restas sucesivas,** y puedes recordar que en el Capítulo Dos ya mencionamos que otra manera de ver la división es restar el divisor tantas veces como sea posible antes de que el resultado sea un valor negativo.

La pareja de registros DE se carga con el valor 40_D que es igual al número de columnas que hay en cada fila de la pantalla. Dado que la primera posición en pantalla es la (1,1) el registro B se carga con un desplazamiento de 1, lo que hace que se alinee la respuesta final en la posición correcta. La instrucción de OLIación **OR A** prepara el registro de testigos de manera que se pueda detectar el bit testigo **M** (menos) cuando la pareja de registros HL alcanza un valor negativo. E.g.

Si es	HL = 81_D
Entonces después de	SBC HL,DE
será	HL = 41 y el curso del programa retrocede para hacer nuevamente:
	SBC HL,DE
Si es	HL = 1 en este momento el testigo **M** no ha sido alzado de manera que el curso del programa retrocede de nuevo...
	SBC HL,DE
	HL = -39

Este valor hace que sea alzado el testigo **M** y el programa salta a la **salida** donde la instrucción ADD HL,DE restaura en la pareja HL el resto -39+40=1. Así HL contiene ahora el resto (1) que es de hecho la **posición X.** La rutina del sistema operativo espera percibir en el registro H la posición X y en el L la posición Y, de manera que se traspasa a H desde L, y a L desde B, que se incrementó cada vez que tuvo lugar una resta que no dió como resultado que se alzara el testigo M. (Cada vez que se resta 40 de la pareja HL sin que el testigo M se alce, significa que la posición en pantalla se ve incrementada en una fila de 40 columnas).

Si quieres que este utensilio de programación se ejecute desde tu programa en Basic, debes cargar cada octeto como constantes de una instrucción DATA y luego meterlos en la parte de la memoria que comienza en la dirección 40000_D. El programa queda entonces instalado y entra en acción citando esa dirección, con la clave **CALL 40000.** Luego ya pueden usarse los tres nuevos comandos directamente desde el teclado o como instrucciones dentro de un programa.

Para impedir que el interpretador Basic escriba encima de las rutinas, debes estipular que la cima de la **memoria** usable por el Basic sea la dirección 39999_D.

Para conservar el programa como un fichero separado, simplemente teclea el listado 3 de ensamble usando un editador/ensamblador, ensambla ese listado fuente y guarda el código objeto obtenido como un fichero en cinta. (Guarda también como fichero el listado fuente simplemente por si has cometido errores de tecleado o quieres hacer posteriormente modificaciones). Antes de cargar el programa en memoria teclea el comando **MEMORY** 39999_D y después de cargarlo **cita** la subrutina mediante el comando **CALL** 40000_D.

Este programa hace amplio uso del registro de indexación IX y sería sensato en este momento explicar cómo se utiliza estos registros de 16 bits. Los registros denominados IX e IY son registros de 16 bits conocidos como **registros de indexación.** Permiten al programador una manera fácil de aprovechar un **índice** para señalar sitios de la memoria ordenados en forma de **ristra** o tabla utilizando un sitio central de la tabla como **base** y señalando cada elemento mediante una **cota,** o desplazamiento, que puede variar desde -128 hasta +127.

LD IX,$3C00_H$

IX -03 apunta aquí ======⇒	2FFD = CA_H
	2FFE
	2FFF = 32_H
IX+00 apunta aquí ========⇒	3C00 = FF_H
	3C01
	3C02
	3C03
IX+04 apunta aquí =========⇒	3C04 = $3F_H$

La instrucción nemónica LD A,(IX+04) dada en este momento haría que el acumulador A contuviera el valor $3F_H$. La nemónica LD A,(IX-03) daría como resultado ahora que el acumulador contuviera el valor CA_H.

Deberías ser capaz de apreciar lo manipulables que son estos dos registros para accesar datos desde una ristra o tabla y esa es la razón de que el Amstrad use el registro IX para señalar a los datos cuando se traspasan parámetros a una subrutina desde el Basic.

Capítulo Nueve

Bits: Pizcas de Información

Ya hemos comentado las ventajas de las **operaciones logicales** en el capítulo 3. En este capítulo echaremos una mirada a la manera de utilizarlas en nuestra programación en Basic. También estudiaremos otro comando provechoso que a menudo es despreciado por los programadores en Basic, que nos permite comunicar al interpretador Basic que vamos a **usar** una función definida por nosotros mismos, mediante el comando de clave **DEF FN.**

Hay algunas operaciones en Basic que son **específicamente binarias,** ya que operan de diferentes maneras sobre operandos enteros de acuerdo con la notación binaria en que se expresa internamente ese operando. Eso se hará patente cuando escribamos un programa para darnos tres nuevos comandos que usan el generador programable de sonidos en un capítulo posterior: todos los registros de ese circuito operan de manera diferente dependiendo de la **calibración binaria** del valor registrado en ellos.

Un buen ejemplo de cómo el Basic opera con bits es la función de clave **JOY(0),** que entrega un valor entero dependiendo del movimiento que el operador haya hecho con el **mando de rumbo.** Si dicho mando ha sido movido a la izquierda, el valor entregado por esta función será el 4. Sin embargo, si el operador apretó al mismo tiempo el botón de disparo, el valor entregado por esa función será 20.

Tabla 9.1

JOY(0)

BIT		MOVIMIENTO DEL MANDO
0	ALZADO	ARRIBA
1	ALZADO	ABAJO
2	ALZADO	IZQUIERDA
3	ALZADO	DERECHA
4	ALZADO	BOTON 2 APRETADO
5	ALZADO	BOTON 1 APRETADO

De la tabla 9.1 podemos ver que el valor entregado por la función es 2 cuando el operador ha movido el mando hacia abajo ya que el bit 1 estará alzado (valor 1), y por tanto tendremos $00010_B=2_D$. Comprobando los otros valores es posible inspeccionar los diversos movimientos del mando para juegos.

20 = mando movido hacia la izquierda y botón de disparo 2 apretado.
5 = mando movido arriba y a la izquierda.

Usar las operaciones logicales para constatar el estado actual de los bits, es una manera más sencilla que la de comprobar el valor entregado por la función.

La capacidad del Amstrad para alzar o poner, bajar o quitar, y examinar cualquier bit desde Basic nos permite utilizar técnicas de programación muy sofisticadas, y una mirada a los métodos empleados para manipular individualmente los bits de un octeto es un ejercicio que merece la pena.

Para comprobar el estado de cualquier bit en un entero de dos octetos, usaríamos la siguiente fórmula:

```
IF N AND 2^A THEN GOTO 20 ELSE RETURN
```

Para constatar el bit 4, usaríamos:

```
IF N AND 2^4 THEN GOTO 1200
```

Para alzar, o poner a 1 cualquier bit en un entero de dos octetos, usaríamos la instrucción de asignación:

```
LET N = N OR 2^A
```

Y si quisiéramos alzar el bit 7, nuestra línea de programa sería:

```
10 LET N = N OR 2^7
```

Para bajar o quitar, o poner a cero, cualquier bit A en un entero de dos octetos N, usaríamos el siguiente comando de asignación:

```
LET N = N AND NOT 2^A
```

Y así para poner a cero el bit 9, sería:

```
LET N = N AND NOT 2^9
```

Este control sobre los bits individuales puede aprovecharse completamente cuando se desea calcular el resto de cualquier valor entero dividido por una potencia de 2. El resto de N/4 se calcula mediante: **N AND NOT -4.**

```
10 LET X = 345
20 LET X= X/16
30 PRINT "EL RESTO DE X/16 ES ";X AND NOT -16
```

También podemos comprobar si un número es impar o par usando la fórmula anterior.

```
10 INPUT "ESCOGE UN NUMERO";N
20 IF N AND NOT -2 THEN GOTO 40
30 PRINT "EL NUMERO ES PAR ":GOTO 10
40 PRINT "EL NUMERO ES IMPAR ": GOTO 10
```

Esta misma clase de operaciones puede usarse con grandes ventajas en datos literales, o 'sartas' de caracteres:

```
Para constatar el bit N del primer caracter de X$:

     10 IF ASC(X$) AND 2^N THEN GOTO 100

Para alzar (fijar a 1) el bit N de un literal
                       de un solo caracter X$:

     10 LET X$ = CHR$(ASC(X$)) OR 2^N


     V.G:   LET X$ = CHR$(ASC(X$)) OR 2^3


Para bajar (fijar a 0) el bit N de un literal
                       de un solo caracter X$:

     10 LET X$ = CHR$(ASC(X$)) AND NOT 2^4
```

En las operaciones con literales anteriores hemos tenido que usar la función de clave ASC para que nos entregue el valor entero asociado al carácter literal almacenado en X$. Después de efectuar una operación con un bit, hemos usado la función CHR$ para hacer que nos entregue un literal de un octeto correspondiente al valor numeral obtenido en la operación.

```
10 CLS
20 INPUT X$
30 FOR I = 0 TO 7
40 PRINT "NUMERO DEL BIT ";I;
50 IF ASC(X$) AND 2^I THEN PRINT "SI": GOTO 70
60 PRINT "NO"
70 NEXT
```

Para establecer más de ocho comprobaciones condicionales necesitamos primero conocer cuántas condiciones se requieren. Podemos luego crear un literal de la longitud calculada usando la siguiente fórmula:

Longitud de literal = INT(Número de Condiciones/8)+1

Dicho literal puede estar formado totalmente por ceros mediante:

X$ = STRING$(Longitud de Literal,0)

También es posible, desde luego, hacer que todos los bits del literal sean uno sustituyendo el 0 por el 255, o sea:

X$ = STRING$(Longitud de Literal,255)

Usando esta forma de manipular individualmente bits, se puede construir una **plantilla** de condiciones 'Sí'/'No' que concuerde con las condiciones requeridas dentro del programa. Con un literal de la máxima longitud se pueden confeccionar así 255 por 8 condiciones 'puesto/quitado'.

El uso de estas funciones para manipular individualmente bits se puede mejorar aún más incluyéndolas en una **función de usuario,** comunicada al interpretador Basic mediante la instrucción **DEF FN.** Una vez que hayas establecido la función, puedes **nombrarla** usando simplemente el nombre dado en la descripción. Eso ahorra un montón de tecleado y hace que los programas sean más fáciles de entender.

```
DEF FN TEST(ARG$,ARG) = ( ASC(MID$(ARG$,INT(ARG/8)+1))
                                    AND
                          2^(ARG-INT(ARG/8)*8) )  <>  0
```

La función anterior comprobará el valor del bit **ARG** dentro del literal **ARG$.** Siempre que quiera nombrarse esta función, una vez que haya sido previamente notificada al interpretador Basic, lo podrás hacer mencionando el nombre, e.g.

FN TEST(n$,n)

Si quisiéramos comprobar el estado del bit 6 en el literal P$ nombraríamos la función y daríamos como **argumentos actuales** los deseados, o sea:

FN TEST(P$,6)

Esta posibilidad de funciones de usuario, cuyo nombre está precedido por las siglas **FN,** es uno de los instrumentos más útiles que han puesto a nuestra disposición. Con un poco de imaginación se pueden conseguir con funciones de usuario hacer absolutamente todo, y la ventaja es que una vez que has comprobado el funcionamiento, puedes usar esa función en tantos programas como desees.

La mayoría de nosotros, en algún momento hemos creado o utilizado un programa para efectuar la conversión a hexadecimal. Considera simplemente la potencia de la siguiente función:

```
10  DEF FN D!(H$) =
      INSTR("123456789ABCDEF",MID$(A$,1,1))*4096 +
      INSTR("123456789ABCDEF",MID$(A$,2,1))* 256 +
      INSTR("123456789ABCDEF",MID$(A$,3,1))*  16 +
      INSTR("123456789ABCDEF",MID$(A$,4,1))
20  INPUT X$
30  PRINT X$;"HEX = " FN D!(X$);" DECIMAL"
40  GOTO 20
```

Apéndice Uno:

Rutinas Utiles del Sistema

$BB06_H$

Explora el teclado hasta que se detecta una pulsación. Entrega en el registro A el carácter pulsado.

```
PUSH AF          ;Preserva registros AF
CALL 0BB06H
LD HL,BUFFER     ;HL apunta al 'buzón' de ingreso y...
LD (HL),A        ;Almacena el carácter en el buzón
POP AF
```

$BB1E_H$

Comprueba si se ha pulsado una tecla.

A = NUMERO DE TECLA
NZ=TECLA PULSADA
Z = TECLA NO PULSADA

```
        LD A,171        ;Comprueba la tecla 171
SCAN:   CALL 0BB1EH
        JR Z,SCAN       ;Tecla no pulsada, así que 'explora'
                        otra vez.
```

BB5D$_H$

Expone en pantalla un carácter.

A = CARACTER A EXPONER

Debes preservar BC, DE, HL, ya que estos registros quedan con información corrupta al salir de esta rutina.

```
PUSH BC
PUSH DE
PUSH HL
LD A,32    ;envía un espacio a la posición corriente del cursor.
CALL 0BB5DH
POP HL
POP DE
POP BC
```

BB60$_H$

Observa cuál es el carácter expuesto en la posición corriente del cursor.

Entrega dicho carácter en el registro A.

```
CALL 0BB60H
CP "T"
JR NZ,NO
```

$BB75_H$

Mueve el cursor a una nueva posición en pantalla.

CONDICIONES PERCIBIDAS
H = COLUMNA
L = FILA

AF & HL CON INFORMACION CORRUPTA AL SALIR

```
PUSH AF
LD HL,0202H
CALL 0BB75H
```

$BB78_H$

Obtiene la posición corriente del cursor:

Entrega: H = Columna
L = Fila

$BB81_H$

Hace el cursor visible.

$BB84_H$

Oculta el cursor.

$BB90_H$

Cambia el color de tinta.

AL ENTRAR A = NUMERO DE TINTA
AL SALIR AF & HL CORRUPTOS

```
LD A,2 ; Tinta 2
CALL 0BB90H
```

$BB96_H$

Fija el color del papel.

AL ENTRAR A = NUMERO DE TINTA
AL SALIR A & HL CORRUPTOS

```
LD A,2 ; Tinta 2
CALL 0BB96H
```

$BBAE_H$

Da la dirección de comienzo de la tabla matricial de caracteres definidos por el usuario.

A = NUMERO DE CARACTER PARA EL PRIMER ELEMENTO DE LA TABLA.
HL = DIRECCION DEL PRIMER OCTETO EN LA TABLA.

BBC0H

Mueve el cursor de gráficos a una nueva posición.

AL ENTRAR DE = COORDENADA X
HL = COORDENADA Y

AL SALIR AF,BC,DE,HL CORRUPTOS

BBDEH

Fija pluma de gráficos al color de tinta percibido en el registro A.

```
LD A,3 ; Tinta 3
CALL 0BBDEH
```

BBE4H

Fija papel de gráficos al color de tinta percibido en el registro A.

```
LD A,1 ; Tinta 1
CALL 0BBE4H
```

BBEAH

Pinta una mota del color corriente.

AL ENTRAR DE = COORDENADA X
HL = COORDENADA Y

AL SALIR AB & BC CORRUPTOS

$BBF6_H$

Traza una raya desde la posición corriente del cursor de gráficos.

DE = COORDENADA X DEL PUNTO FINAL
HL = COORDENADA Y DEL PUNTO FINAL

$BBFC_H$

Expone un carácter en la pantalla de gráficos en la posición corriente del cursor.

AL ENTRAR A = CARACTER

AL SALIR BC,DE,HL CORRUPTOS

El carácter se expone con su esquina superior izquierda situada en la posición gráfica corriente.

$BB0E_H$

Elije un modo de pantalla.

```
LD A,1 ; Modo 1
CALL 0BB0EH
```

AL SALIR BC,DE,HL CORRUPTOS

BC14$_H$

CLS

AL SALIR AF,BC,DE,HL CORRUPTOS

BC32$_H$

Fija una tinta a un nuevo color.

AL ENTRAR A = NUMERO DE LA TINTA
B = PRIMER COLOR
C = SEGUNDO COLOR

AL SALIR AF,BC,DE,HL CORRUPTOS

BC38$_H$

Fija colores para el reborde.

AL ENTRAR:

B = PRIMER COLOR
C = SEGUNDO COLOR

AL SALIR AF,BC,DE,HL CORRUPTOS

$BC5F_H$

Traza una raya recta horizontal.

AL ENTRAR:

DE = COORDENADA X COMIENZO
BC = COORDENADA X FINAL
HL = COORDENADA Y

$BC62_H$

Traza una raya recta vertical.

AL ENTRAR:

A = NUMERO DE TINTA
DE = COORDENADA X
HL = COORDENADA Y COMIENZO
BC = COORDENADA Y FINAL

$BCD1_H$

Incorpora una nueva clave BASIC.

AL ENTRAR:

BC = TABLA DE LA NUEVA CLAVE
HL = PUNTERO A UN DEPOSITO INTERMEDIO DE 4 OCTETOS.

$BD2B_H$

EXpresa un carácter por impresora.

AL ENTRAR:

A = CARACTER A ENVIAR

Si el carácter es enviado con éxito a la impresora el testigo de llevada ('C') queda alzado.

```
        LD A,41
SEND:   CALL 0BD2BH
        JR NC, SEND  ; si no enviado, ensaya otro envío
```

$BD34_H$

Envía dato al PSG.

AL ENTRAR:

A = NUMERO DEL REGISTRO
C = DATO A ENVIAR (0 a 255).

Apéndice Dos:

Tintas y Valor de Luminancia

LUMINANCIA	TINTA	LUMINANCIA	TINTA
0	NEGRO	13	BLANCO
1	AZUL	14	AZUL PASTEL
2	AZUL BRILLANTE	15	NARANJA
3	ROJO	16	ROSA
4	MAGENTA	17	MAGENTA PASTEL
5	MALVA	18	VERDE BRILLANTE
6	ROJO BRILLANTE	19	VERDE MARINO
7	PURPURA	20	CIANO BRILLANTE
8	MAGENTA BRILLANTE	21	LIMA
9	VERDE	22	VERDE PASTEL
10	CIANO	23	CIANO PASTEL
11	AZUL CIELO	24	AMARILLO BRILLANTE
12	AMARILLO	25	AMARILLO PASTEL
		26	BLANCO BRILLANTE

Técnicas de Programación Avanzada con AMSTRAD.

Este libro está diseñado para abrir un mundo nuevo de oportunidades de programación para los usuarios de AMSTRAD.

Aquí encontrarás rutinas e instrumentos de programación que te ahorrarán montones de horas de innecesario tecleo y te ayudarán a incrementar la velocidad de ejecución de tus juegos y programas.

Se explican exhaustivamente las técnicas empleadas para crear estas ayudas de programación, útiles y poco habituales, que incluyen como reseñar programas en código máquina dentro de instrucciones REM, tablas DIMensionadas y ¡meras variables literales!.

Destacan en este singular libro los programas para:
- Creación de FIGURAS ("sprites").
- Desarrollo de un sistema de sonido completamente nuevo.
- Generación de gráficos.

Este libro te ayudará a conseguir incluso más de lo que crees posible sacar de tu AMSTRAD.

El autor *KEITH HOOK lleva más de quince años en estudios informáticos, ha escrito varios libros sobre esta materia y contribuye habitual y regularmente con artículos en la revista PERSONAL COMPUTER NEWS.*

También de RA-MA:
- AMSTRAD CPC-464 Programación Avanzada. Mark Harrison.

RA-MA
CHIQUINQUIRA, 28 (COCUY)
28033 MADRID.

ISBN 84-86381-06-1

Tecnicas de
Programación Avanzada con **AMSTRAD**

KEITH HOOK

ra-ma

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/