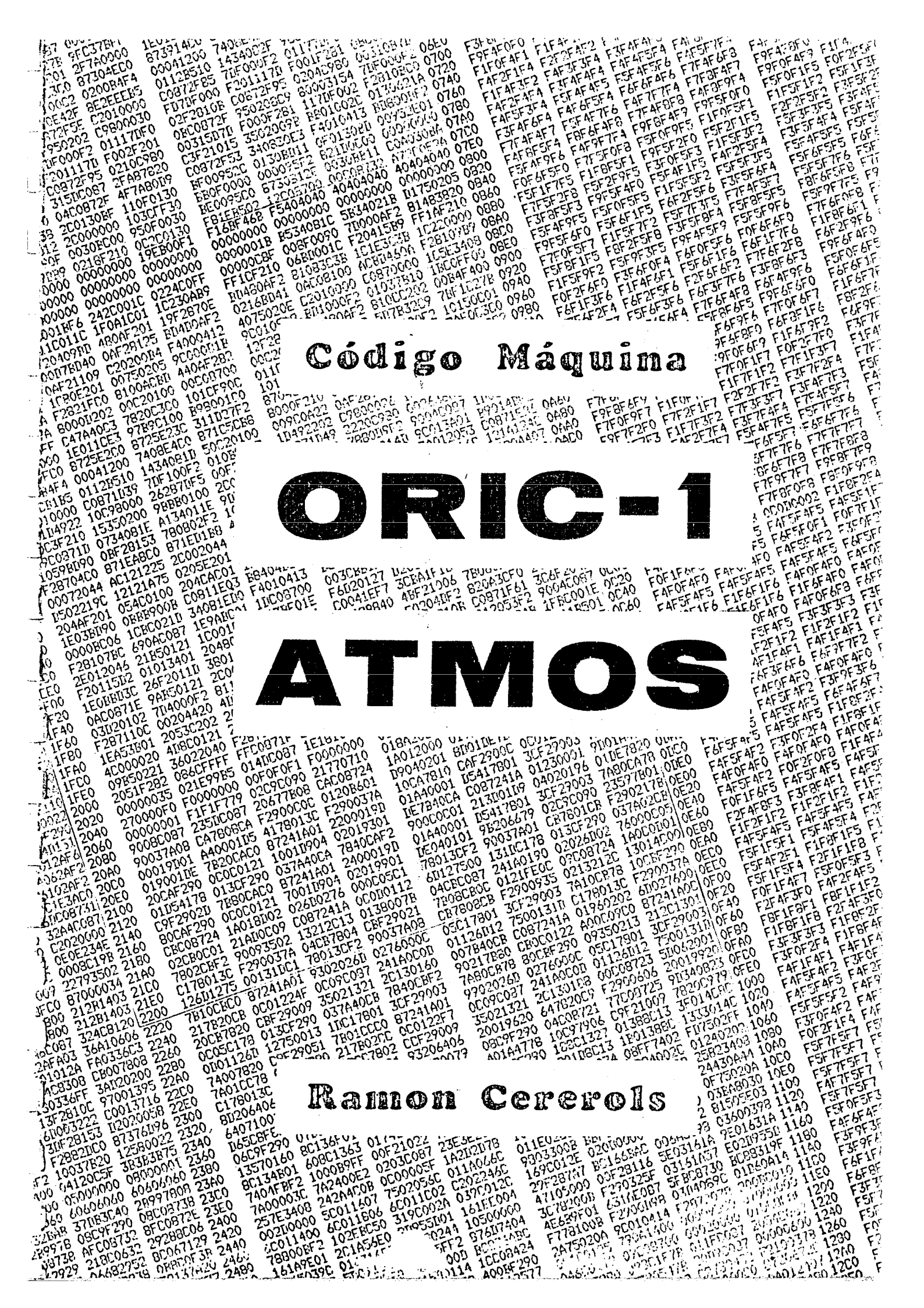

Código Máquina
ORIC-1
ATMOS
Ramon Cererols

I N D I C E :

Enero de 1.984

Ramon Cererols i Macià
Provença, 513-515, 5º, 4ª
Barcelona-25

Teléfono: (93)348.15.97

P R E S E N T A C I O N :

El ORIC-1 constituye un equipo con una relación prestaciones/precio realmente interesante, que ha permitido a muchos iniciarse en el apasionante mundo de la informática. Cuando uno ha conseguido, superando la insuficiencia del manual, llegar a un cierto conocimiento del lenguaje Basic, se da una idea de las grandes posibilidades de la máquina.

Sin embargo, incluso cuando se llega a dominar el Basic, queda por descubrir otro apasionante mundo que subyace en las entrañas del microprocesador (este minúsculo "corazón" que da vida al ORIC). Se trata del código máquina.

Este lenguaje, rodeado muchas veces de un cierto misterio, permite unas posibilidades que multiplican la potencia del equipo: por una parte, la velocidad de ejecución puede multiplicarse en ocasiones hasta por un factor de cien, por otra parte, con este método es posible realizar cosas que serían imposibles de otro modo.

Así, por ejemplo, en la página 30 se encontrará una breve rutina que permite hacer aparecer en la línea reservada superior (aquella en la que habitualmente aparecen los mensajes "Saving" y "Loading") una indicación horaria, con las horas, minutos y segundos.

Ello no tendría nada de particular, si no fuera porque el funcionamiento de este reloj no interfiere en absoluto el normal de la máquina. Así, podemos seguir escribiendo programas, ejecutándolos, modificándolos, etc. y el reloj sigue funcionando independientemente.

Este es solo un ejemplo. En el libro se encontrarán otras rutinas, siempre presentadas de modo que su introducción sea fácil, sin necesidad de conocimientos especiales, y con métodos de comprobación para verificar la correcta introducción de los datos.

El libro está pensado de modo que pueda resultar útil, tanto a los que, conocedores del Basic, deseen iniciarse en el código máquina, como a los que, conocedores ya de éste último, estén interesados en cómo aplicarlo al ORIC, aprovechando las rutinas de la ROM.

A este fin, a partir de una introducción general, se pasa a describir la estructura del Basic del ORIC-1, el funcionamiento general de la ROM, como se interpretan las sentencias del Basic, como funciona el cassette y el generador de sonido, etc.

El libro se completa con una tabla de direcciones útiles, ejemplos de rutinas ya preparadas, y un recetario para preparar fácilmente uno mismo sus propias rutinas.

Espero que este libro pueda cubrir la laguna informativa que sobre el tema existe y pueda proporcionar al lector nuevas satisfacciones con su ORIC.

He procurado evitar los errores, y repasar las rutinas una vez mecanografiadas. No obstante, supongo que se habrán deslizado algunos errores (espero que pocos). Agradeceré, pues, cualquier comentario que permitirá corregirlos en sucesivas ediciones.

C O D I G O M A Q U I N A

Existe un recelo generalizado entre quienes han dado sus primeros pasos con el Basic, a la hora de introducirse en la programación en código máquina. Se comentan y reconocen sus indudables ventajas, pero un cierto hálito de misterio impide su utilización.

Ello es debido a una serie de factores:

- por una parte, la falta de manuales de iniciación simples y accesibles (existen, eso sí, espesos y completos tratados, usualmente en inglés, pero poco aptos para quien pretende iniciarse en el tema).

- falta, en el ORIC-1, de un editor que permita introducir fácilmente las instrucciones en código máquina, intercalarlas, suprimirlas, modificarlas, etc. como puede hacerse con los programas Basic.

- dificultad en la depuración de errores, ya que un fallo en un programa en código máquina puede acarrear bloqueos imprevistos.

- falta de información sobre las rutinas de la ROM y el funcionamiento de la misma, de modo que puedan ser utilizadas en nuestros programas en código máquina.

Los anteriores problemas son subsanables: así, por lo que respecta a la falta de información, este libro pretende aportar bastantes datos al respecto (a la espera de que el fabricante la complete), en cuanto a la falta de editor, empiezan a existir algunos programas utilitarios en cassette, y yo mismo puedo facilitar a quien lo desee un completo ensamblador, y la depuración de errores se simplifica preparando los programas en bloques modulares (la llamada programación estructurada).

Por último, para tratar de paliar la falta de manuales de iniciación, en el presente capítulo voy a dar las nociones generales que permitan a cualquiera, si no un conocimiento a fondo del tema, imposible en tan corto espacio, sí por lo menos la comprensión suficiente para poder iniciar los primeros experimentos, y entender el funcionamiento de las rutinas.

Empezaremos diciendo que un programa en código máquina está formado por una sucesión de líneas, cada línea conteniendo una sola instrucción. Cada línea está representada en la memoria de la máquina por un conjunto de uno a tres bytes. El primer byte constituye el código de la instrucción; los otros dos, si existen, constituyen el operando sobre el que se aplica la instrucción.

El primer byte (el correspondiente a la instrucción), indica al microprocesador el número de bytes que forman la línea, pues no existe ningún tipo de separación entre líneas, ni numeración de las mismas.

Las operaciones que realizan las distintas instrucciones son muy simples, para comprenderlas, conviene conocer la estructura general del microprocesador.

El microprocesador posee una serie de registros de memoria internos, a saber:

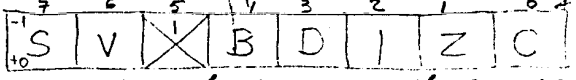
- el registro A, o acumulador, que es el usado comunmente para las operaciones aritméticas o lógicas.

-los registros X e Y, usados generalmente como índice (referencia a partir del origen de una tabla).

- el registro de estado P, que contiene una serie de indicadores (flags) que nos determinan el estado del microprocesador en un momento dado.
- el puntero S de stack, que nos señala la última posición de memoria utilizada para conservar las direcciones de retorno de los subprogramas.
- el contador de programa PC, que contiene la dirección de memoria donde se encuentra la instrucción que se está leyendo o ejecutando.

Excepto este último, que es de dos bytes (16 bits), los restantes solo son de un byte (8 bits), por lo que el máximo valor que pueden contener es 255, en decimal. Por contra, el contador de programa puede llegar a 65535, por lo que la máxima cantidad de memoria que es capaz de direccionar es de 64 K (los 48 de RAM y los 16 de ROM). Recordemos que 1 K vale 1Ø24 bytes.

CONTROL POR PROGRAMA	El registro de estado P contiene siete indicadores (el octavo bit no se utiliza), con las siguientes funciones:
NO	- S (signo): se pone a 1 cuando el resultado de la última operación ha sido negativo (se entiende por negativo todo byte superior o igual a 128).
SI	- V (overflow): se pone a uno cuando el resultado de la última operación ha excedido la capacidad (pasa de 255).
SI	- B (break): se pone a uno al producirse una interrupción (BRK).
SI	- D (decimal): se pone a uno para ejecutar las operaciones en decimal (de lo contrario se realizan en binario).
SI	- I (interrupciones): cuando está puesto a uno, las interrupciones no están autorizadas.
NO	- Z (cero): se pone a uno cuando el resultado de la última operación es un número nulo.
SI	- C (carry): arrastre en las operaciones de suma de varios bytes.



El microprocesador 65Ø2 admite un total de 56 instrucciones distintas, que podemos clasificar en varios tipos:

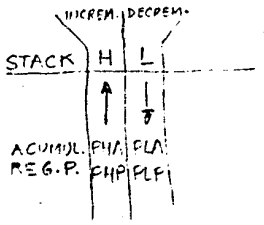
- 16 instrucciones de traslado de bytes.
- 26 instrucciones de operaciones entre bytes.
- 13 instrucciones de saltos.
- una última instrucción que no hace nada, la llamada NOP, usada para "borrar" instrucciones erroneas.

Todas las instrucciones son representadas mediante abreviaturas (mnemónicos) de sus nombres ingleses, y todas son de tres letras. Tengamos en cuenta, no obstante, que este nombre de la instrucción es solo el que le damos para mayor facilidad de utilización por nuestra parte, pero que en el interior de la máquina, a cada instrucción le corresponde un cierto valor del byte, que podemos obtener mediante unas tablas adecuadas.

Instrucciones de traslado de bytes:

LDA pasa el contenido de una posición de memoria al registro A.
LDX ídem al registro X.
LDY ídem al registro Y.

STA pasa el valor del registro A a una posición de memoria.
STX ídem del registro X.
STY ídem del registro Y.
TAX pasa el contenido del registro A al registro X.
TAY ídem al registro Y.
TXA pasa el contenido del registro X al registro A.
TYA ídem del registro Y.
TSX pasa el contenido del registro S al registro X.
TXS viceversa.
PHA pasa el contenido del registro A al stack. Incrementa S.
PLA recupera del stack un valor y lo pasa a A. Decrementa S.
PLP recupera del estack el registro P. Decrementa S.
PHP pasa el contenido del registro P al stack. Incrementa S.



Instrucciones de operaciones entre bytes:

ADC ADC suma el contenido de A al de una posición de memoria, con arrastre. → ADC
SBC ídem para la resta.
INC INC incrementa en una unidad una posición de memoria.
INX ídem el registro X.
INY ídem el registro Y.
DEC DEC decrementa en una unidad el contenido de una posición de memoria.
DEX ídem el registro X.
DEY ídem el registro Y.
CMP CMP compara el contenido de A con el de una posición de memoria. OPERACION AFECTA (A-M); N,Z y C; C=1 SI M <
CPX ídem el registro X.
CPY ídem el registro Y.
AND AND realiza el AND lógico de A con una posición de memoria.
ORA ídem la función OR.
EOR ídem el OR exclusivo.
BIT comprueba el signo del valor de una posición de memoria.
ASL desplaza a la izquierda los bits de una posición de memoria.
ROL ídem, con entrada a través del carry.
LSR ídem a la derecha.
ROR íedm, con entrada del carry.
STC SEC pone a uno el bit Carry.
SED ídem el bit Decimal.
STI SEI ídem el bit Interrupt. → STI
CLC CLC pone a cero el bit Carry.
CLD ídem el bit Decimal.
CLI CLI ídem el bit interrupt.
CLV ídem el bit overflow.

Instrucciones de salto:

BCC salta si el Carry está a cero.
JNE BNE ídem el bit Zero.
BPL ídem el bit Signo.
BVC ídem el bit Overflow.
BCS salta si el bit Carry está a uno.
BEQ ídem el bit Zero.
BMI ídem el bit Signo.
BVS ídem el bit Overflow.
JMP JMP salta incondicionalmente.
JSR salta a una subrutina.
RTS vuelve de una subrutina.
BRK salta a subrutina de interrupción.
IRET RTI vuelve de una subrutina de interrupción.

Como hemos indicado, representamos cada instrucción, en el interior de la memoria de la máquina, por un cierto valor del byte. Como quiera que solo hay 56 instrucciones, mientras que a un byte podemos darle 256 valores distintos, significa que podemos asignar más de un valor a la misma instrucción. ¿Por qué hacemos esto? Para comprenderlo, vamos a ver el ejemplo de una de las instrucciones más corrientemente utilizadas, LDA (que pasa un cierto valor al registro acumulador A).

Esta instrucción suele utilizarse para pasar el contenido de una posición de memoria determinada al acumulador A. Supongamos que se trate de la posición de memoria 4ØØØØ (en hexadecimal $9C4Ø) (* ver nota a pie de página). Representaremos la instrucción mediante LDA $9C4Ø, y diremos que el direccionamiento (forma de definir la dirección del operando) es absoluto, ya que nos indica la dirección de donde debemos leer el byte, codificado sobre dos bytes.

En este caso, la traducción de la instrucción (según las tablas) es $AD, y a continuación deberemos poner los dos bytes que forman la dirección pero, atención!, en orden inverso al habitual, es decir, primero el $4Ø (el de peso bajo) y luego $ 9C (el de peso alto). El conjunto se introducirá, pues, en la memoria de la máquina mediante la sucesión de los tres bytes $AD,$4Ø,$9C. Estos tres bytes forman el auténtico código máquina.

Pero sucede a menudo que utilizamos las direcciones de la parte baja de la memoria, de Ø a 255, que constituyen lo que se llama la página cero, porque el primer byte de la dirección es nulo. En este caso, este byte nulo constituiría una pérdida de memoria innecesaria, por lo que se opta por dar a la instrucción LDA una traducción distinta: $A5, que indica al microprocesador que se trata de una dirección codificada sobre un solo byte. Así, por ejemplo, LDA $4Ø se traduciría por los bytes $A5,$4Ø.

Empezamos a ver la conveniencia de emplear un programa utilitario que nos realice la conversión de los símbolos que nosotros utilizamos (el llamado lenguaje de ensamblaje) al auténtico código máquina, evitándonos tener que consultar continuamente las tablas para cada instrucción y modo de direccionamiento. Además, estos programas utilitarios (los ensambladores) tienen múltiples ventajas, que se irán viendo posteriormente.

Puede también suceder, que en lugar de querer leer el contenido de una posición de memoria, deseemos introducir un valor concreto en el acumulador. En este caso colocaremos el valor a introducir justo a continuación del código de la instrucción, por lo que a este modo de direccionamiento le llamaremos inmediato. En este caso, el código correspondiente a LDA es $A9. Este modo de direccionamiento lo señalamos con el símbolo #. Así, para introducir el valor $4Ø en A, haremos LDA #$4Ø, lo que se traduce por los bytes $A9,$4Ø. Obsérvense las diferencias con el caso anterior.

Pero la instrucción LDA posede todavía otros cinco tipos de direccionamiento más sofisticados, cuya utilidad radica en la lectura de tablas al ser utilizadas dentro de un bucle, y que describimos a continuación.

Direccionamiento preindexado (INDEXADO-INDIRECTO / (INDIRECTO-X)) por X: la dirección de donde debemos leer el valor que pasaremos a A debe obtenerse sumando el contenido de X al valor que sigue a la instrucción LDA, y leyendo el contenido de la dirección resultante, y la siguiente. ($AL)

EJEMPLO: X=5; LDA $Ø3 ¡LA DIRECCION QUE CONTIENE EL DATO A CARGAR EN EL A se FORMARA CON el contenido de la dirección Ø8 (LSB) y el contenido de la dirección Ø9 (MSB)

(*) Nota: En código máquina es usual representar los valores hexadecimales mediante el signo $. El signo # se reserva para el direccionamiento inmediato, que se describe en esta misma página.

(INDIRECTO-INDEXADO)
(INDIRECTO-Y) (DATO)
Direccionamiento postindexado por Y: la dirección de donde leeremos el valor a pasar a A se obtendrá sumando el contenido de Y a la dirección contenida en el byte señalado por el valor que sigue a LDA, y el siguiente.
($B1) EJEMPLO: LDA $03; suponemos que la posic $03 contiene el BYTE $9A y la $4 contendrá $F5, la dirección resultante será → $F59F
Y=5

(ABSOLUTO-X)
Direccionamiento indexado por X: la dirección de donde tomar el valor para A se obtendrá sumando la dirección absoluta ~~contenida~~ INDICADA en los dos bytes siguientes a LDA, más el contenido del registro X. EJEMPLO: LDA 3F4A; si X=$4, la dirección efect. será → 3F4E
($BD)

(ABSOLUTO Y)
Direccionamiento indexado por Y: igual al anterior, pero con el registro Y.
LDA=$B9

Direccionamiento indexado por X, en página cero: en un solo byte. ($B5)

Estos cinco tipos de direccionamiento se representan, respectivamente, mediante:

LDA (n,X) LDA (n),Y LDAnn,X LDAnn,Y LDA n,X

y en cada caso, la instrucción LDA se traduce por un cierto valor que indica el tipo de direccionamiento.

Las instrucciones que, como LDA, admiten estos tipos de direccionamiento, son las siguientes (con sus correspondientes traducciones, para que la tabla sea de utilidad para la escritura de rutinas):

	2 BYTES INMEDIATO #n	3 BYTES ABSOLUTO nn	2 BYTES PAGE-Z n	2 BYTES PREINDEX.-X (n,X) (INDIRECTO-X)	2 BYTES POSINDEX.-Y (n),Y (INDIRECTO-Y)	2 BYTES PAGE-Z/X n,X	3 BYTES INDEXADO-X nn,X (ABSOLUTO-X)	3 BYTES INDEXADO-Y nn,Y (ABSOLUTO-Y)
ORA	$09	$0D	$05	$01	$11	$15	$1D	$19
AND	$29	$2D	$25	$21	$31	$35	$3D	$39
EOR	$49	$4D	$45	$41	$51	$55	$5D	$59
ADC	$69	$6D	$65	$61	$71	$75	$7D	$79
STA	-	$8D	$85	$81	$91	$95	$9D	$99
LDA	$A9	$AD	$A5	$A1	$B1	$B5	$BD	$B9
CMP	$C9	$CD	$C5	$C1	$D1	$D5	$DD	$D9
SBC	$E9	$ED	$E5	$E1	$F1	$F5	$FD	$F9

Y antes de seguir estudiando nuevas instrucciones, vamos a realizar nuestra primera rutina máquina con lo ya conocido, para ver la forma práctica de llevarlo a cabo (la rutina, lógicamente, será muy simple).

Supongamos que queremos colocar una letra A en el extremo superior izquierdo de la pantalla (en la línea reservada normalmente para los mensajes del cassette). Ello significa que debemos colocar el valor $41 (que es el código ASCII para la letra A) en el primer byte de la zona de la pantalla (que en modo TEXT es $BB8$.(D48000)

Para ello deberemos primero colocar en el registro A el valor $41, y después pasar el contenido de A a la posición de memoria citada, es decir:

```
$A9,$41     LDA #$41      ;cargamos el valor de la letra A EN ACUMULADOR
$8D,$8$,$BB STA $BB8$     ;lo pasamos a la pantalla
```

Ahora bien, falta una instrucción que, una vez hecho lo anterior, nos retorne al control por teclado (o, en su caso, al programa que haya llamado la rutina). Esta instrucción es normalmente RTS (retorno de subrutina).

Si añadimos esta instrucción, y luego a la izquierda de cada línea anotamos los valores de los bytes correspondientes, según las tablas, obtendríamos:

```
$A9  $41        )   LDA #$41
$8D  $8Ø  $BB   }=  STA $BB8Ø
$6Ø             )   RTS
```

La rutina ocupará pues, seis bytes, que podemos colocar a partir de la dirección $4ØØ, que es la zona habitualmente reservada para los pequeños programas en código máquina escritos por el usuario. Una manera de entrar estos valores, sería utilizando seis veces la instrucción POKE, para lo que previamente deberíamos traducir a valores decimales los anteriores, obteniendo:

```
$A9   $41   $8D   $8Ø   $BB   $6Ø
169 , 65  , 141 , 128 , 187 , 96
LDA       STA             RTS
```

y ahora podemos hacer:

```
POKE #4ØØ,169          #400=1024
POKE #4Ø1,65
POKE #4Ø2,141
POKE #4Ø3,128
POKE #4Ø4,187
POKE #4Ø5,96
```

Ya tenemos la rutina en su sitio. Para poder utilizarla, recurriremos a la instrucción CALL #4ØØ y veremos como aparece la A en el sitio anunciado. Evidentemente, el único interés de esta rutina es servir de ejmplo de como puede procederse. Vemos una vez más, la conveniencia del programa ensamblador, que nos evita largas secuencias de POKEs, lentas y sometidas a errores.

No todas las instrucciones admiten tantos tipos de direccionamiento como LDA, así tenemos la siguiente tabla:

	INMEDIATO #n	ABSOLUTO nn	PAGE-Z n	ACUMULADOR A	PAGE-Z-X n,X	nn,X	nn,Y	n,Y
ASL		$ØE	$Ø6	$ØA	$16	$1E		
ROL		$2E	$26	$2A	$36	$3E		
LSR		$4E	$46	$4A	$56	$5E		
ROR		$6E	$66	$6A	$76	$7E		
DEC		$CE	$C6		$D6	$DE		
INC		$EE	$E6		$F6	$FE		
STY		$8C	$84		$94			
STX		$8E	$86					$96
LDY	$AØ	$AC	$A4		$B4	$BC		
LDX	$A2	$AE	$A6				$BE	$B6
CPY	$CØ	$CC	$C4					
CPX	$EØ	$EC	$E4					

y bastantes de las instrucciones tienen una traducción fija, bien porque no requieren operando o porque sólo admiten un tipo de direccionamiento:

BRK $ØØ	PHP $Ø8	BPL $1Ø	CLC $18
JSR $2Ø	PLP $28	BMI $3Ø	SEC $38
RTI $4Ø	PHA $48	BVC $5Ø	CLI $58
RTS $6Ø	PLA $68	BVS $7Ø	SEI $78
DEY $88	BCC $9Ø	TYA $98	TAY $A8
BCS $BØ	CLV $B8	INY $C8	BNE $DØ
CLD $D8	INX $E8	BEQ $FØ	SED $F8
TXA $8A	TXS $9A	TAX $AA	TSX $BA
DEX $CA	NOP $EA		

y ya para completar las 56 instrucciones, nos quedan solo dos casos especiales. Se trata de BIT, que admite direccionamiento absoluto ($2C) o en página cero ($24); y de JMP absoluto ($4C) o indirecto ($6C).

Ya con la tabla completa de traducciones, vamos aver cómo puede utilizarse el direccionamiento indexado. Para ello, supongamos que queremos hacer una rutina que llene los primeros 16 lugares de la línea reservada superior de la pantalla con letras "A". Podríamos repetir el proceso descrito en la primera rutina que habíamos preparado, tantas veces com letras queramos poner. Sin embargo, vemos que ello resultaría largo y monótono.

Podemos hacerlo de modo más simple. En lugar de colocar en la instrucción la dirección absoluta donde queremos colocar la "A" (es decir STA $BB8Ø), utilizaremos el direccionamiento indexado por X, así: STA $BB8Ø,X. Antes de empezar, pondremos a cero el registro X, mediante LDX #Ø, con lo que colocaremos la A en la dirección correspondiente a la suma de $BB8Ø más el contenido de X que era cero.

Hecho esto, incrementamos el valor de X, mediante INX, y comparamos el valor de X con 16, para ver si hemos llegado al final. De no ser así, volvemos al principio del bucle. La rutina, junto con los valores de los bytes a colocar (traducción a código máquina) queda así:

```
                          (65)          ACUMULADOR
                                          ↓
$A9 $41          LDA #$41       ;cargamos en A el carácter "A"
$A2 $ØØ          LDX #Ø         ;ponemos el índice X a cero
$9D $8Ø $BB lazo STA $BB8Ø,X    ;pasamos A a la pantalla
$E8              INX            ;incrementamos X
$EØ $1Ø          CPX $16        ;comparamos X con 16
$DØ $F8          BNE lazo       ;si no es igual, volvemos arriba
$6Ø              RTS            ;volvemos al Basic
```

Aunque en la rutina hemos usado una etiqueta ("lazo") para saber donde debíamos volver, ésta solo tiene utilidad como referencia, y no tiene traducción en código máquina. Después de la instrucción de salto (en este caso, BNE), debemos incluir la distancia, en bytes, hasta donde queremos ir (máximo, 127 bytes) o, si debemos volver para atrás, 256 menos dicha distancia.

Así, como en nuestro ejemplo, debemos volver 8 bytes atrás, colocamos 256 menos 8 igual a 248 bytes ($F8 en hexadecimal). Vemos, pues, otro motivo de la conveniencia de un programa ensamblador que nos evite estos cálculos.

Una vez sabemos qué bytes debemos introducir en la memoria, podríamos utilizar el sistema de ir POKEando los correspondientes valores en las sucesivas direcciones a partir de la que hayamos escogido como principio de la rutina. Ello resultaría tedioso. Es mejor preparar un pequeño programa de Basic que vaya leyendo los valores de una serie de DATA, por ejemplo:

```
 1Ø  FOR D = #4ØØ TO #4ØC
 2Ø  READ B : POKE D,B
 3Ø  NEXT : END
1ØØ  DATA #A9,#41,#A2,#ØØ,#9D,#8Ø,#BB,#E8,#EØ,#1Ø,#DØ,#F8,#6Ø
```

Una vez ejecutado, ya tenemos la rutina en su sitio, y podemos utilizarla, haciendo CALL #4ØØ.

Con estos pequeños ejemplos, sólo pretendo haber dado una cierta iniciación a lo que es el código máquina, cómo se preparan las rutinas y cómo se colocan en la memoria de la máquina. El lector interesado puede proseguir el estudio en los libros que tratan el tema, pero en realidad, la mejor manera de aprender es practicar pequeños ejemplos como los comentados, introduciendo paulatinamente nuevas instrucciones, ver los resultados, estudiar las distintas rutinas incluidas en este libro, analizando su funcionamiento, etc.

También, para poder aprovechar al máximo las posibilidades que la programación en código máquina nos ofrece, conviene conocer la estructura de la máquina, organización de los programas Basic y de las variables, cómo la máquina ejecuta las instrucciones Basic, la dirección de las rutinas de la ROM más útiles, etc. Esto, acompañado de ejemplos concretos, es lo que voy describiendo en el resto del libro.

Si alguien prefiere ir directo a las aplicaciones, al final encontrará una serie de aplicaciones prestas para su uso (como subrutinas de nuestros programas en Basic). Aunque están descritas de modo que puedan ser utilizadas sin haber leído el resto del libro, su funcionamiento sólo podrá ser comprendido después de dicha lectura.

- -

Antes de continuar, una sencilla aplicación. Las direcciones &3Ø6 y &3Ø7 controlan el tiempo que tarda en leerse el tablero. Normalmente contienen el valor 1ØØØØ que corresponde a los microsegundos que transcurren entre dos lecturas (1Ø milisegundos).

Podemos acelerar la ejecución de un programa disminuyendo el número de lecturas del tablero (aumentando el tiempo entre ellas). Podéis comprobarlo cronometrando un programa de prueba y haciendo DOKE #3Ø6,65535 y repitiendo el cronometraje. En estas condiciones, la escritura resulta más lenta.

En el otro extremo, podéis acelerar la velocidad de escritura haciendo DOKE #3Ø6,3ØØØ y comprobando la velocidad de repetición de las teclas. Conviene no bajar mucho de este valor, pues puede bloquearse la máquina (24ØØ suele ser el límite).

Estas manipulaciones alteran la duración de los WAIT. Así pues, si un programa no requiere introducciones por teclado, podemos acelerarlo incluyendo al principio del mismo DOKE #3Ø6,65535, y al final DOKE #3Ø6,1E4.

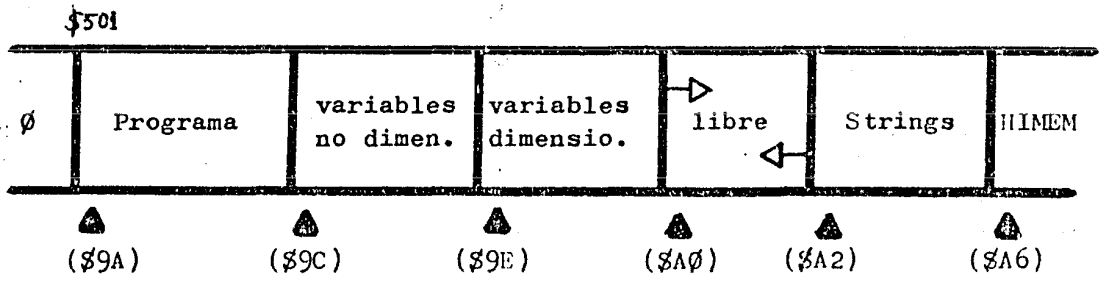
E S T R U C T U R A D E L B A S I C

El apéndice A del manual nos indica que el espacio de memoria entre las direcciones $5ØØ y $97FF (ó $B3FF si ejecutamos el comando GRAB), está destinado a los programas del usuario, pero no nos especifica cómo está distribuida esta zona, ni como se representan en la misma las variables, y el propio programa de Basic.

De hecho, la distribución es variable, en función de las necesidades de cada momento. Las distintas rutinas se encargan de adaptar esta distribución a la longitud del programa, número y tipo de las variables, etc.

Ello obliga a disponer en algún lugar unas referencias que nos indiquen dónde empieza y termina cada zona. Estas referencias son unos punteros (grupos de dos bytes consecutivos) situados en página cero, el primer byte de los cuales constituye los ocho bits de peso bajo de la dirección referenciada, y el segundo los ocho bits de peso alto. Con el fin de abreviar, representaremos el puntero formado por dos bytes (por ejemplo, $9A-$9B), por sólo la dirección del primero de ellos, entre paréntesis (en nuestro ejemplo ($9A)).

La distribución general es la siguiente:



El principio del programa de Basic habitualmente está en la dirección $5Ø1, ya que el byte anterior (dirección $5ØØ) contiene un cero necesario para la correcta interpretación del programa. Este es pues, el valor contenido en el puntero ($9A), de inicio del programa de Basic. Normalmente, este puntero no sufre alteración durante el funcionamiento normal.

El puntero ($9C) contiene la dirección siguiente al último byte del programa de Basic, que es, a su vez, el primero ocupado por las variables normales, no dimensionadas (por ejemplo: A, N$, Z3%, etc.).

Por su parte, las variables dimensionadas ocupan la zona inmediata posterior a las anteriores, y el primer byte de las mismas está marcada por el contenido del puntero ($9E). El primer byte posterior a los ocupados por las variables dimensionadas, está marcado por el contenido del puntero ($AØ), y aquí empieza una zona libre, cuya longitud va variando durante el curso de la ejecución, según la utilización de variables.

El HIMEM, dirección a partir de la cuál no puede pasar el Basic (utilizable, por ejemplo, para rutinas en código máquina), viene fijado por el contenido del puntero ($A6), y de allí para atrás se sitúan los strings, o tiras alfabéticas, el principio de las cuales es marcado por el contenido del puntero ($A2).

A medida que se van ocupando las distintas zonas, los punteros ($AØ) y ($A2) se van acercando, hasta que, al juntarse, se produce el siguiente fenómeno:

- en primer lugar, se produce una limpieza de los espacios que ya no son necesarios, para abrir espacio libre,

- si el espacio así obtenido no es suficiente para la aplicación que debía dársele, se produce el mensaje "Out of memory error", y la detención de la ejecución.

Cuando ejecutamos un HIMEM, toda la zona de strings se desplaza, si el espacio libre es suficiente para ello, y con él; los punteros ($A2) y ($A6).

Por cierto, uno de los "bugs" (errores de diseño) del ORIC-1, consiste en que, en el momento de conectar la máquina, los punteros ($A2) y ($A6) son inicializados incorrectamente, a los valores $BFFF y $9F00, respectivamente.

Ello provoca que los strings entran en la zona de memoria habilitada para la pantalla ($BB80 a $BFDF), y que las variables pueden invadir la zona del generador de caracteres ($B400 a $BB7F). Para evitarlo, es una práctica muy conveniente que siempre que conectamos la máquina empecemos ejecutando un HIMEM #9800, así como incluir esta misma instrucción al principio de todos nuestros programas, para mayor seguridad.

Cuando al iniciar la ejecución de un programa se produce el borrado de todas las variables (o al ejecutar el comando CLEAR), en realidad lo único que sucede es que los punteros se desplazan. Así, los punteros ($9E) y ($A0), se colocan en coincidencia con el ($9C), y el ($A2) en coincidencia con el ($A6).

También vemos que hay que distinguir entre la memoria ocupada por un programa, y que nos vendrá dada por DEEK(#9C) - DEEK(#9A), de la memoria que necesita para su ejecución, que es la suma de la ocupada por el programa más la ocupada por todas las variables y strings.

Debido al fenómeno de limpieza antes descrito, un programa que requiera 12K para su ejecución, normalmente "rodará" más rápido con 40K disponibles hasta el HIMEM, que si sólo dispone de los 12K imprescindibles (pues no se harán tan frecuentes los procesos de limpieza).

Vista la distribución general, pasemos a comentar el contenido de cada zona. Empezaremos con el programa propiamente dicho.

La zona de programa contiene las distintas líneas de Basic, según la siguiente disposición:

- dos octetos iniciales, el contenido de los cuales es la dirección de comienzo de la línea siguiente,
- dos octetos más que contienen el número de línea expresado en binario (en ambos casos, el primer byte es el de peso bajo),
- la secuencia de instrucciones, en la que los nombres de las variables, los números, las tiras alfabéticas y los comentarios son representados por sus códigos ASCII (de $20 a $7F), mientras que las palabras reservadas del Basic son abreviadas a un solo byte, llamado token (de $80 a $F7), lo que ahorra espacio en la memoria y tiempo de ejecución,
- un octeto nulo que señala el fin de la línea.

La última línea va seguida de otros dos bytes nulos. La tabla de códigos de los bytes de un programa (caracteres ASCII y tokens), es la siguiente:

TABLA DE BYTES:

(de $2Ø a $7D: caracteres ASCII; de $8Ø a $F7: tokens palabras reservadas)

DEC.	HEX$	0	1	2	3	4	5	6	7	8	9	10 A	11 B	12 C	13 D	14 E	15 F
32	2	espacio	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	3	Ø	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	£
96	6	©	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	7	p	q	r	s	t	u	v	w	x	y	z	{	|	}		
128	8	END	EDIT	INVERSE	NORMAL	TRON	TROFF	POP	PLOT	PULL	LORES	DOKE	REPEAT	UNTIL	FOR	LLIST	LPRINT
144	9	NEXT	DATA	INPUT	DIM	CLS	READ	LET	GOTO	RUN	IF	RESTORE	GOSUB	RETURN	REM	HIMEM	GRAB
160	A	RELEASE	TEXT	HIRES	SHOOT	EXPLODE	ZAP	PING	SOUND	MUSIC	PLAY	CURSET	CURMOV	DRAW	CIRCLE	PATTERN	FILL
176	B	CHAR	PAPER	INK	STOP	OR	WAIT	CLOAD	CSAVE	DEF	POKE	PRINT	CONT	LIST	CLEAR	GET	CALL
192	C	!	NEW	TAB(	TO	FN	SPC(	@	AUTO	ELSE	THEN	NOT	STEP	+	-	*	/
208	D	↑	AND	OR	>	=	<	SGN	INT	ABS	USR	FRE	POS	HEX$	&	SQR	RND
224	E	LN	EXP	COS	SIN	TAN	ATN	PEEK	DEEK	LOG	LEN	STR$	VAL	ASC	CHR$	PI	TRUE
240	F	FALSE	KEY$	SCRN	POINT	LEFT$	RIGHT$	MID$	GO								

PUNTEROS { INICIO #9A / FINAL #9C } ‖ { CONTADOR INTERRUP-CIONES $276 / $277 }

La siguiente zona la ocupan las variables no dimensionadas (numéricas y alfabéticas). Cada una ocupa siete octetos de memoria, de los cuales los dos primeros son los códigos ASCII de los dos primeros caracteres del nombre de la variable. Si el nombre solo tiene un carácter, el segundo es nulo. En el caso de las variables alfabéticas y enteras, los caracteres $ y % no son considerados como formando parte del nombre de la variable. En su lugar, se posiciona el correspondiente bit de peso alto del primer o segundo carácter.

Los cinco octetos restantes tienen una estructura distinta, según se trate de una variable numérica o alfabética:

- en el caso de una variable numérica, estos cinco octetos contienen el valor, en binario y punto flotante, de la variable.

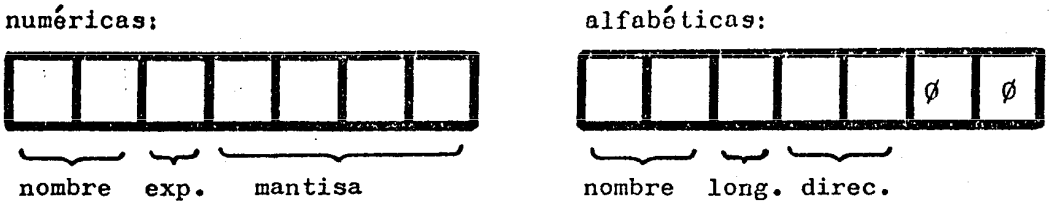
  El primer ócteto contiene el exponente, aumentado en $8Ø. En el caso excepcional que el valor de la variable sea Ø, el exponente también lo es. La mantisa ocupa los cuatro bytes restantes (los de peso alto, primero). Como que el primer bit, en punto flotante, sería siempre uno, lo aprovechamos para contener el signo (Ø = positivo, 1 = negativo).

Como que el exponente está formado por siete bits (el octavo determina el signo), el número más grande que podemos representar es 2↑127, es decir, la cantidad 1,7Ø1411835 x 1Ø↑38. Por otra parte, como que el número más grande que podemos representar con los 32 bits de la mantisa es 2↑32 - 1 = 4294967295, la precisión normalmente obtenida en los cálculos es de unas 9 cifras decimales.

En cambio, en el caso de una variable alfabética, la estructura de los cinco últimos bytes es distinta. En efecto, el primero de los cinco bytes nos indica la longitud de la tira (string), que puede ser, por tanto de Ø a 255 caracteres.

Los dos octetos siguientes indican la dirección donde realmente empieza el string, que puede ser, dentro del propio programa de Basic, si la variable se ha creado allí, o en la zona de strings, si ha habido que modificarla, o hacer alguna operación con ella.

Finalmente, los dos octetos siguientes no tienen ninguna utilidad, y suelen ser nulos. El esquema de cada tipo de variable, es, pues:



Pasemos a la siguiente zona, la de las variables dimensionadas. Son las tablas y, como antes, pueden ser numéricas o alfabéticas. En este caso, la longitud ocupada por una variable depende de su tipo y su dimensión.

Las variables dimensionadas numéricas de punto flotante ocupan un número de octetos igual a 5 + 2a + 5b, donde "a" es el número de dimensiones y "b" el número total de elementos. Así, por ejemplo, si hacemos DIM A(3,5,2) el espacio ocupado será de 5 + 2x3 + 5x(4x6x3) = 371 octetos. Recordemos que A(3) tiene cuatro elementos, ya que existe también A(Ø).

Su estructura es: 1) dos octetos que contienen el nombre de la variable, según el mismo sistema que en el caso de las variables no dimensionadas. 2) dos octetos más que indican el número de bytes ocupados por la tabla (en nuestro caso, 371). 3) un octeto que indica el número de dimensiones (en el ejemplo, 3). 4) tantos bloques de dos octetos como dimensiones haya, conteniendo cada bloque el número de elementos de la respectiva dimensión, empezando por la que está más a la derecha. 5) tantos bloques de cinco octetos como elementos contiene la tabla. Estos bloques contienen los valores de las variables, con la misma estructura descrita anteriormente. Al dimensionar la tabla, los valores se ponen todos a cero.

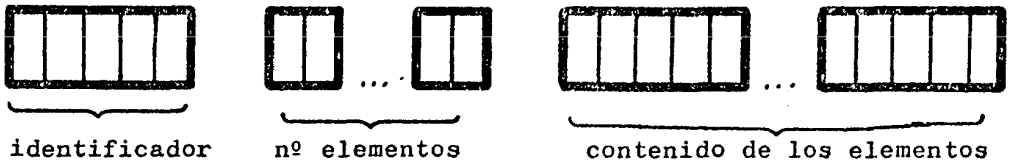
El orden de las variables sería, en nuestro ejemplo: A(Ø,Ø,Ø), A(1,Ø,Ø), A(2,Ø,Ø), A(3,Ø,Ø), A(Ø,1,Ø), A(1,1,Ø), A(2,1,Ø), A(3,1,Ø), A(Ø,2,Ø), etc.

Si se trata de una tabla de valores enteros, cada elemento ocupa solamente dos bytes, lo que representa un considerable ahorro de memoria.

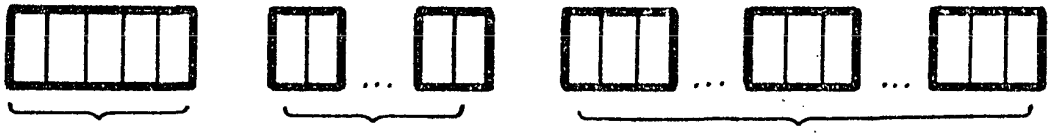
Si se trata de tablas de valores alfabéticos, su ocupación es 5 + 2a + 3b, siendo a y b los mismos valores anteriores. Ahora bien, hay que considerar que, en este caso, tal ocupación se refiere exclusivamente a los punteros, ya que las tiras alfabéticas propiamente dichas están en otras zonas.

La estructura en este caso es: 1) los dos octetos del nombre de la variable. 2) los dos octetos de ocupación de la tabla, el octeto del número de dimensiones, y los bloques del número de elementos de cada dimensión, como en el caso de las variables numéricas. 3) después, tantos bloques de tres octetos como elementos haya. En cada bloque, el primer octeto indica la longitud de la tira alfabética, y los dos octetos siguientes la dirección de comienzo de la misma. Inicialmente esta zona es puesta a cero.

Tablas numéricas:



Tablas alfabéticas:



Cuando en el programa se busca el valor de una variable, la búsqueda empieza por el principio de la respectiva zona (por ejemplo, si se trata de una variable no dimensionada, la búsqueda empezará en la dirección indicada por el puntero ($9C)).

Obviamente, cuantas más variables de aquél tipo haya, y cuanto más atrás se encuentre situada la que buscamos, más tiempo tardará la búsqueda. Por este motivo, si en un programa se utilizan muchas variables distintas, conviene crear primero las que vayamos a utilizar más frecuentemente, pues de este modo la velocidad de ejcución aumentará.

Para crear una variable, basta incluir en las primeras líneas del programa una instrucción del tipo A = 3, si queremos inicializarla a un valor concreto, o simplemente DIM A. Recordemos que la instrucción DIM admite variables múltiples, por ejemplo, DIM A, B, C, etc.

Otro método para aumentar la velocidad de ejecución a base de disminuir el número de variables, consiste en aprovechar el mismo nombre de variable para otros usos cuando ya no se necesita el valor anterior. De todos modos, esta práctica se presta a confusiones, por lo que no es muy recomendable.

I N T E R P R E T A C I O N

Como todos los micro-ordenadores de su tipo, el ORIC-1 es una máquina montada alrededor de un micro-procesador (en este caso, el 65Ø2). El único lenguaje que un micro-procesador es capaz de "entender" y ejecutar es el código máquina. Así pues, para que el ORIC-1 pueda ejecutar sentencias y programas en Basic, es necesario un programa (en código máquina) que traduzca las secuencias de instrucciones Basic a sus equivalentes en código máquina.

Este programa, que realiza la traducción o "interpretación", está escrito en la ROM (o EPROM, según modelo), y ocupa las posiciones de memoria desde $C000 hasta $FFFF, utilizando también la parte de RAM entre $0000 y $03FF.

El conocimiento de la estructura y funcionamiento del programa intérprete tiene un doble interés:

- por una parte, nos proporcionará información, no sólo sobre el ORIC-1, sino sobre la estructura general de un micro-ordenador;

- por otra, nos permitirá aprovechar para nuestros propios programas las subrutinas incluidas en la ROM, entender lo que hace en realidad cada instrucción del Basic, enmendar los bugs y, en general, obtener el máximo rendimiento de la máquina.

En el presente capítulo vamos a describir, por lo menos en su parte principal, la estructura del programa intérprete. Para ello incluiremos diversos diagramas de flujo que hagan más comprensibles las explicaciones.

Empezaremos por el momento en que conectamos el ORIC-1 a la red. En este momento, un circuito especial de RESET envía al microprocesador una señal de interrupción que hace que la ejecución empiece en la dirección $F42D (punto de arranque en frío).

Por cierto, en algunos modelos, si la aparición del voltaje no es suficientemente brusca, no se produce la señal de RESET, y el aparato presenta dificultades de arranque. Para evitarlo, hay dos soluciones:

- la primera, conectar el alimentador a la red antes de conectarlo al ORIC-1;

- la segunda, disponer entre el alimentador y el ORIC-1, un pequeño interruptor. Este sistema tiene la ventaja adicional de permitir efectuar en todo momento un RESET total en casos de bloqueo profundo (que puede producirse al trabajar en código máquina, si cometemos algún error).

En la dirección $F42D antes citada empieza la fase de inicialización general. En esta fase se procede a:

- inicializar el stack, el registro de estado y algunas direcciones de RAM empleadas para saltos;

- comprobar el estado de la memoria RAM, colocando ciertos valores en la misma y leyéndolos a continuación, así como comprobar la existencia de 16 K ó 48 K (según versión);

- inicializar los periféricos (VIA, circuito de sonido);

- inicializar toda una serie de registros en las primeras páginas de memoria RAM;

- limpiar la pantalla e imprimir el mensaje "Copyright...";

- inicializar el generador de caracteres.

Finalizada esta fase, se produce un salto a la dirección $C4B5, que es donde empieza el lazo principal del programa intérprete (ver "Secuencia principal del interpretador").

Ahí, en primer lugar, se produce la impresión del mensaje "Ready". A resaltar que, en lugar de llamar directamente a la rutina de impresión ($CBED), se produce un JSR $1A, donde hay un JMP $CBED (incluído en la fase de inicialización).

Dado que siempre que finaliza un programa (o la ejecución de una línea entrada directamente por teclado), se pasa por este punto, tanto si ha sido por finalización normal, por Control C, o por detección de error; podemos, alterando los valores contenidos a partir de $1A, obtener varios efectos, por ejemplo:

- que aparezca otro mensaje, en lugar de "Ready" (por ejemplo: "Listo").

- que, en caso de error, el programa no se detenga, sino que continue en algún lugar previamente determinado (simulación de ON ERROR GOTO), aunque antes habrá aparecido ya el mensaje de error.

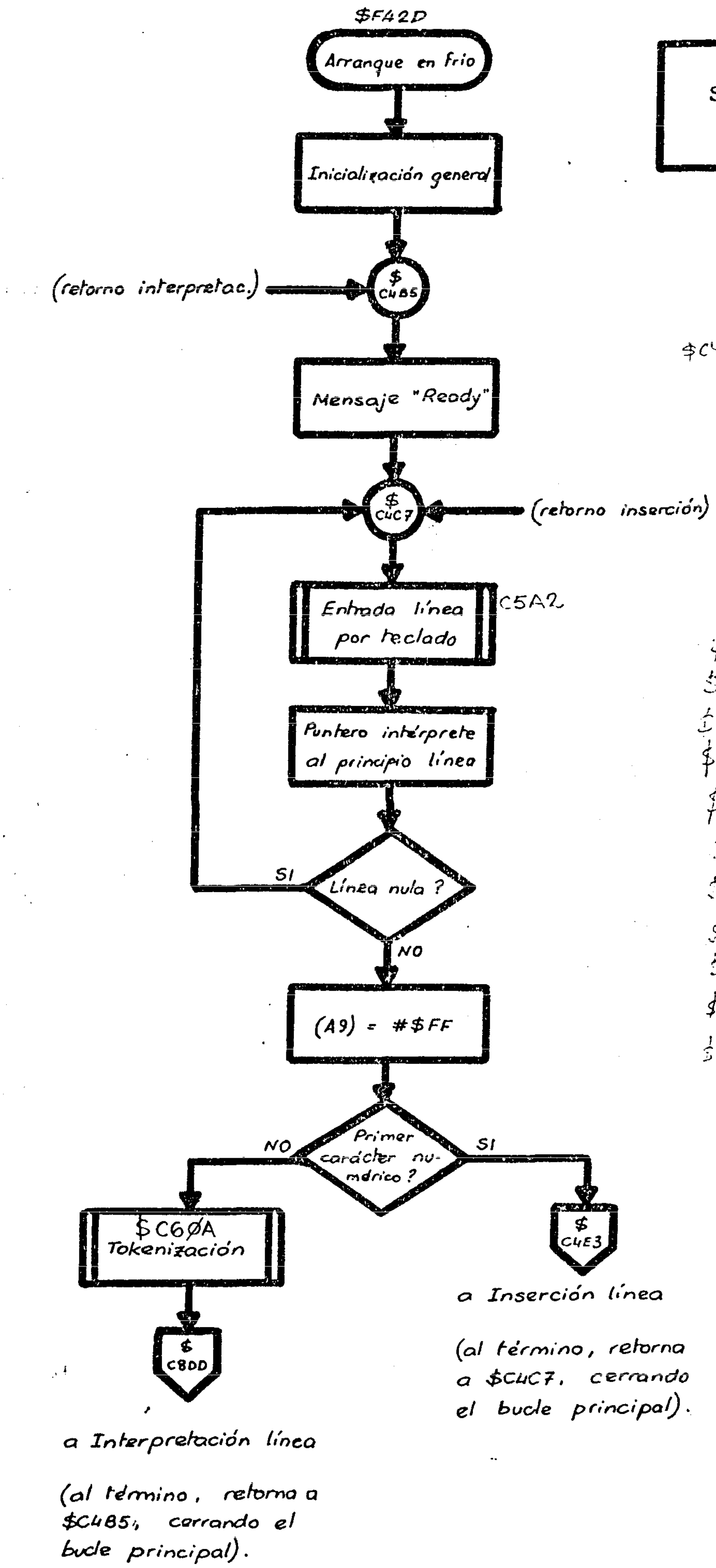
- establecer algún sistema de protección de programas, incluyendo en este lugar un JMP $F42D. Así, si alguien interrumpe por cualquier medio el programa, para leerlo o copiarlo, se ejecutará inmediatamente una inicialización general que lo borrará, junto con todo el contenido de la memoria. Para ello basta incluir como primera instrucción del programa la siguiente: DOKE #1B,#F42D.

Después del mensaje "Ready", pasamos a la espera de la introducción de una línea por teclado (rutina $C5A2, que estudiaremos más adelante). Hecho esto, el puntero del intérprete (que sigue las instrucciones a medida que las vamos leyendo) se coloca al principio de la línea introducida, y leemos el primer carácter de la misma.

Si es un byte nulo (caso de una línea vacía), volvemos a esperar una nueva entrada de línea. En caso contrario, comprobamos si el primer carácter de la línea era un número o no.

Si es un número, la línea va destinada a ser insertada en el programa de Basic actual, y se producirá un salto a $C4E3. De lo contrario, se producirá la tokenización (conversión de los caracteres que forman los nombres de las instrucciones de Basic a códigos que las simbolizan con un solo byte, ahorrando memoria y tiempo de ejecución), y un salto a $C8DD, donde se produce la interpretación y ejecución de las sentencias incluidas en la línea.

SECUENCIA PRINCIPAL DEL INTERPRETADOR




```
$C4B5  JSR  $CC8F
$C4E8  LSR  $2E      ANULA LA IMPRESION DE LA PANTALLA
       LSR  $2F1     IMPRESION POR PANTALLA
       LSR  $2F2
       LDA  #$B6
       LDY  #$C3
       JSR  $1A
$C4C7  LSR  $2F1
$C4CA  JSR  $C5A2    ENTRADA DE LINEA POR TECLADO
$C4CD  STX  $E9      PUNTERO DE INTERPRETE BASIC
$C4CF  STY  $EA
$C4D1  JSR  $E2
$C4D4  TAX
$C4D5  BEQ  ($C4C7)
$C4D7  LDX  #$FF
$C4D9  STX  $A9
$C4DB  BCC  ($C4E3)
$C4DD  JSR  $C60A
$C4E0  JMP  $C8DD

$C4E3  JSR  $CA98
$C4E6  JSR  $C60A
$C4E9  STY  $26
       JSR  $C6DE
       BCC  $44  (IR A C534)
```

Estudiemos ahora cómo se produce la introducción de la línea por teclado. Digamos antes que el objeto de la misma es pasar los caracteres introducidos a una zona de memoria en página cero, concretamente a partir de la dirección $35, seguida de un byte nulo, para señalar el final.

La rutina empieza en $C5A2, y en primer lugar pone a cero el registro X, que es el que se va utilizando durante la introducción como cuenta del número de caracteres entrados.

Esperamos la pulsación de una tecla. Si es Control A, cogemos de la posición actual del cursor en pantalla el carácter que contenga. Luego se procede a la impresión del carácter, o ejecución del correspondiente código de control, según el caso.

Se comprueban algunos casos especiales (ver diagrama "Introducción línea por teclado"), y se pasa el carácter leído a la correspondiente posición en el buffer de teclado (la zona antes mencionada a partir de $35). Comprobamos luego que no se haya superado la longitud máxima, y volvemos a leer una nueva tecla.

El proceso se repite hasta que la tecla pulsada es Return, en cuyo caso se añade el nulo de fin de línea, se posiciona el puntero al principio de la misma, y retornamos a la rutina llamante.

El proceso de tokenización va leyendo la línea carácter por carácter. A cada nuevo carácter, comprueba si existe coincidencia entre la secuencia de caracteres que empieza en el recién leído, y los nombres de palabras reservadas del Basic, incluídos en una tabla que empieza en $CØEA.

Si se produce la coincidencia, la secuencia de caracteres es sustituida por un único byte que contiene un código ("token") de la instrucción. De este modo, durante la ejecución de un programa, cada instrucción de Basic empezará por un token, que nos conducirá a la correspondiente rutina en código máquina.

Esta es la causa de algunos errores que se producen en ocasiones y que, si uno no cae en ello, pueden ser de difícil localización.

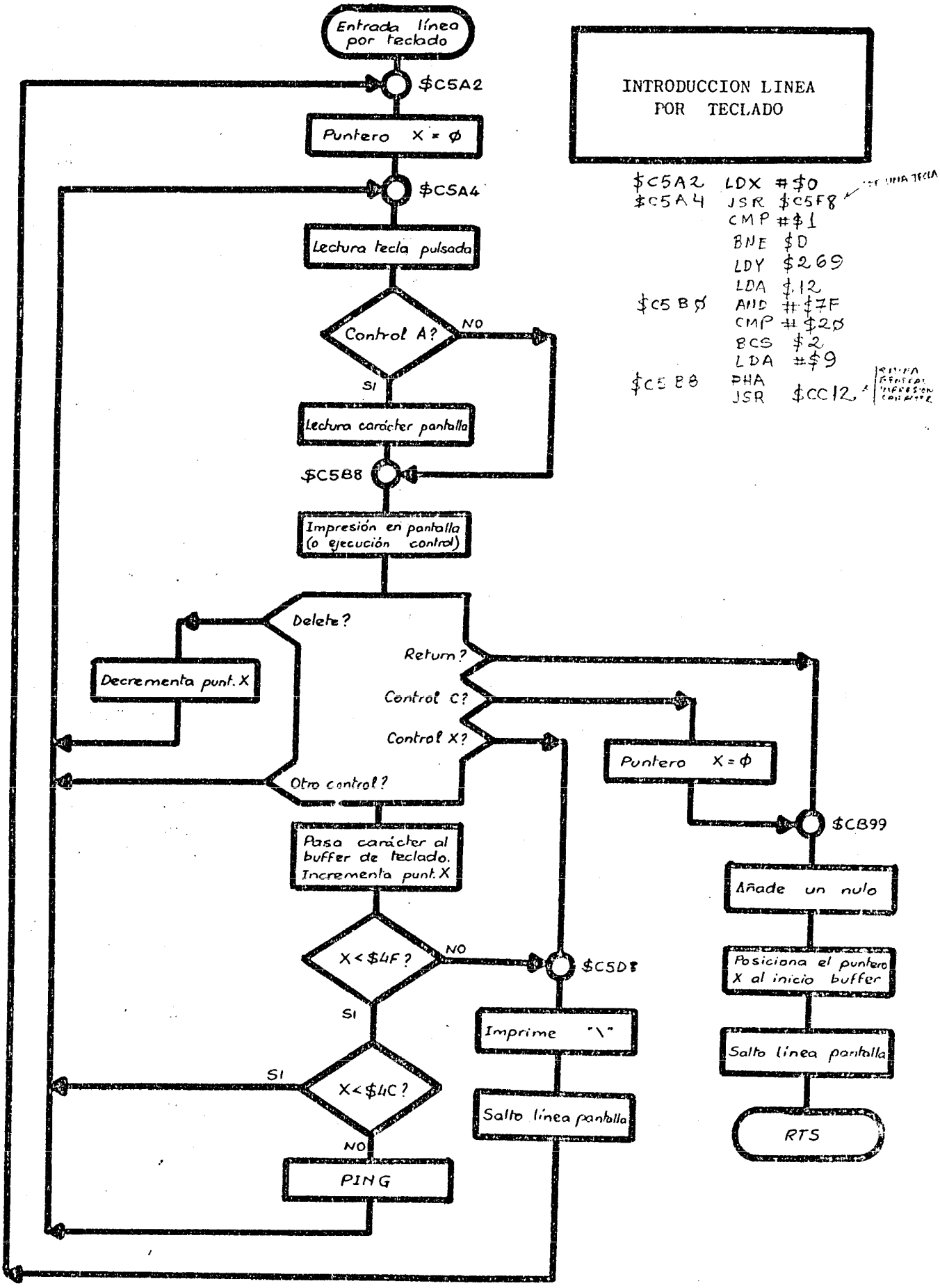
En efecto, si queremos escribir una línea de Basic que contenga lo siguiente:

```
1Ø  IF M = F OR N = G THEN ...
```

y la escribimos junta, sin espacios de separación, al ser tokenizada, la variable F y la palabra reservada OR formarán la palabra FOR, que será interpretada como un error de sintaxis.

Para evitarlo, hay que estudiar cada caso. En este, puede solucionarse intercalando un espacio entre la F y el OR, o bien intercambiando las dos condiciones, con lo que resultaría:

```
1Ø  IF N = G OR M = F THEN ...
```


Entrada línea por teclado
$C5A2
Puntero X = Ø
$C5A4
Lectura tecla pulsada
Control A?
NO
SI
Lectura carácter pantalla
$C5B8
Impresión en pantalla (o ejecución control)
Delete?
Return?
Control C?
Control X?
Otro control?
Decrementa punt. X
Pasa carácter al buffer de teclado. Incrementa punt. X
X < $4F?
NO
SI
X < $4C?
SI
NO
PING
$C5D8
Imprime "\"
Salto línea pantalla
Puntero X = Ø
$CB99
Añade un nulo
Posiciona el puntero X al inicio buffer
Salto línea pantalla
RTS
INTRODUCCION LINEA POR TECLADO
$C5A2 LDX #$0
$C5A4 JSR $C5F8
CMP #$1
BNE $D
LDY $269
LDA $12
$C5B0 AND #$7F
CMP #$20
BCS $2
LDA #$9
$C5B8 PHA
JSR $CC12

Supongamos que la línea que acabamos de introducir en el buffer iba precedida de un número. En este caso se deduce que la línea va destinada a ser insertada dentro del programa de Basic actualmente residente en la memoria RAM.

La rutina que realiza tal función empieza en la dirección $C4E3. En primer lugar, la sucesión de caracteres que forman el número de línea (colocados a partir de $35) deben ser transformados a su valor binario, sobre dos bytes, y pasados a las direcciones $33-$34. Esta tarea la desempeña la sub-rutina $CA98.

Por cierto, que esta rutina admite como máximo número de línea 63999. Ello se hace así porque durante la ejecución de un programa de Basic, el número de línea es salvado en las posiciones $A8-$A9. Si en lugar de un programa, la ejecución es de una línea entrada directamente por teclado, para distinguirlo cargamos el valor $FF en la dirección $A9 (byte de peso alto del número de línea). Si el número de línea fuera, por ejemplo, 6540Ø, el byte de peso alto adquiriría este mismo valor, prestándose a una falsa interpretación.

A continuación se produce la tokenización del resto de la línea, según la rutina $C6ØA, ya comentada. Al final, la longitud de la línea se coloca en la dirección $26, para los posteriores cálculos de variación de los punteros.

Luego se busca a partir del principio del programa de Basic para buscar una línea que contenga el mismo número que la que acabamos de crear. Si existe, procedemos a su eliminación, recopiando el resto del programa a partir del final de la línea anterior.

Ello obliga a variar el puntero del final del Basic, y recalcular los links, es decir, los dos bytes iniciales de cada línea que contienen la dirección de comienzo de la siguiente. Esta última función es encomendada a la subrutina $C56F. Estos cambios obligan, por último, a borrar las variables, cambiando los punteros que señalaban las zonas que ocupaban (subrutina $C733).

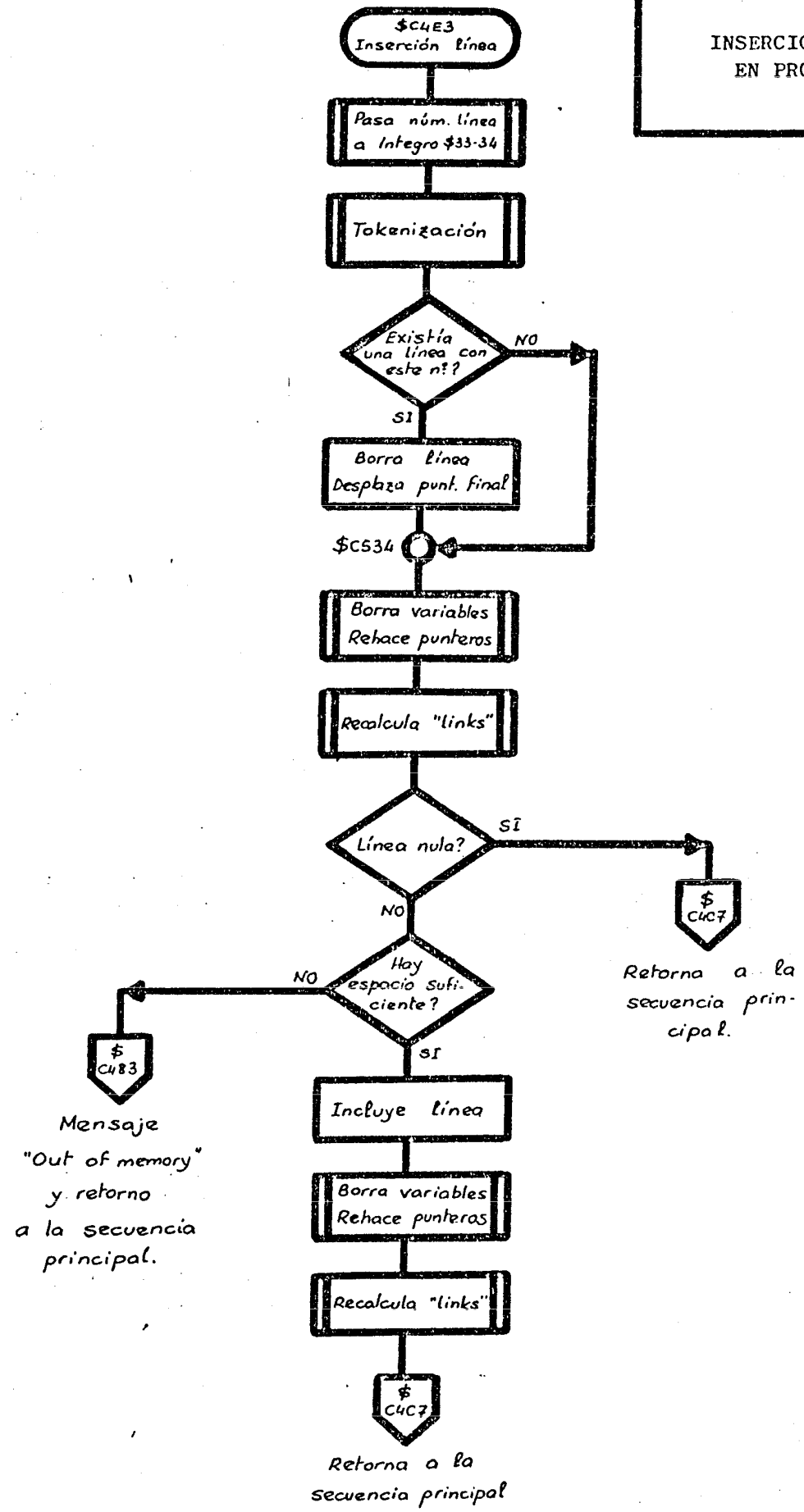
Si la nueva línea es nula (es el caso cuando simplemente introducimos un número de línea y pulsamos la tecla Return), hemos finalizado el proceso de inserción. En caso contrario, se abre un nuevo espacio en el programa de Basic corriendo el resto del programa tantos lugares como indique la dirección $26, y modificando de nuevo el puntero de fin de Basic.

La línea es recopiada en este espacio recién abierto, y nuevamente se repite el proceso de recalculado y modificación de los links, borrado de las variables y modificación de los punteros.

Terminado el proceso, se produce el retorno a la dirección $C4C7, en la secuencia principal del interpretador. Como que este punto de retorno es posterior al fragmento de rutina en que se produce la impresión del mensaje "Ready", este no se produce en tal caso.

Debido a todos los cambios introducidos, y al borrado de las variables, un programa de Basic interrumpido en el que se haya modificado alguna línea, no puede ser continuado. En efecto, la rutina $C733 coloca a Ø la dirección $AD. Al efectuar CONT, se comprueba esta dirección, y si es nula, se produce un salto a rutina de error, con impresión del mensaje "Can't continue".

INSERCION LINEA
EN PROGRAMA



Si la línea que hemos introducido por teclado no empieza con un número de línea, se entiende que va destinada a su ejecución inmediata (como hemos visto en el estudio de la secuencia principal del interpretador).

En este caso, se produce un salto a la dirección $C8DD, donde empieza el proceso de ejecución de las secuencias de instrucciones en Basic. En realidad, esta rutina sirve lo mismo para la ejecución de las líneas entradas directamente por teclado, que para la ejecución de un programa previamente introducido en la memoria (luego veremos cómo).

En primer lugar, se comprueba si estamos en modo TRON, comprobando el bit de peso alto de la dirección $2F4 (que es el que conserva este estado), y, en caso afirmativo, se procede a la impresión del número de línea entre paréntesis rectos.

A continuación, se procede a la lectura del primer carácter de la instrucción, y el saltó a la subrutina que ejecutará la instrucción propiamente dicha. Esta subrutina ($C8FE), por su importancia específica, será estudiada independientemente.

Terminada la ejecución de la instrucción, se lee el teclado para comprobar si entretanto se ha pulsado el Control C. En caso afirmativo, saltaremos a $C942, para proceder a la finalización del proceso.

De lo contrario, continuamos salvando los punteros. Leemos el siguiente carácter para comprobar si se trata del nulo del final de línea. Si no es así, comprobamos que sea el carácter dos puntos (":") que separa una de otra las distintas instrucciones. De no ser así, saltaríamos a la rutina de error. Si efectivamente, eraeste el carácter, volvemos a $C8DD para seguir con la siguiente instrucción.

Si habíamos encontrado el nulo de fin de línea, comprobamos si se trata del final de programa (leyendo el byte posterior al siguiente, y viendo si es también nulo). En este caso, saltamos a $C958 para proceder a la finalización (retorno a la secuencia principal).

Si no habíamos llegado al final del programa, leemos el número de línea, lo guardamos en $A8-$A9, corremos el puntero al principio de la primera instrucción de la nueva línea y seguimos el proceso.

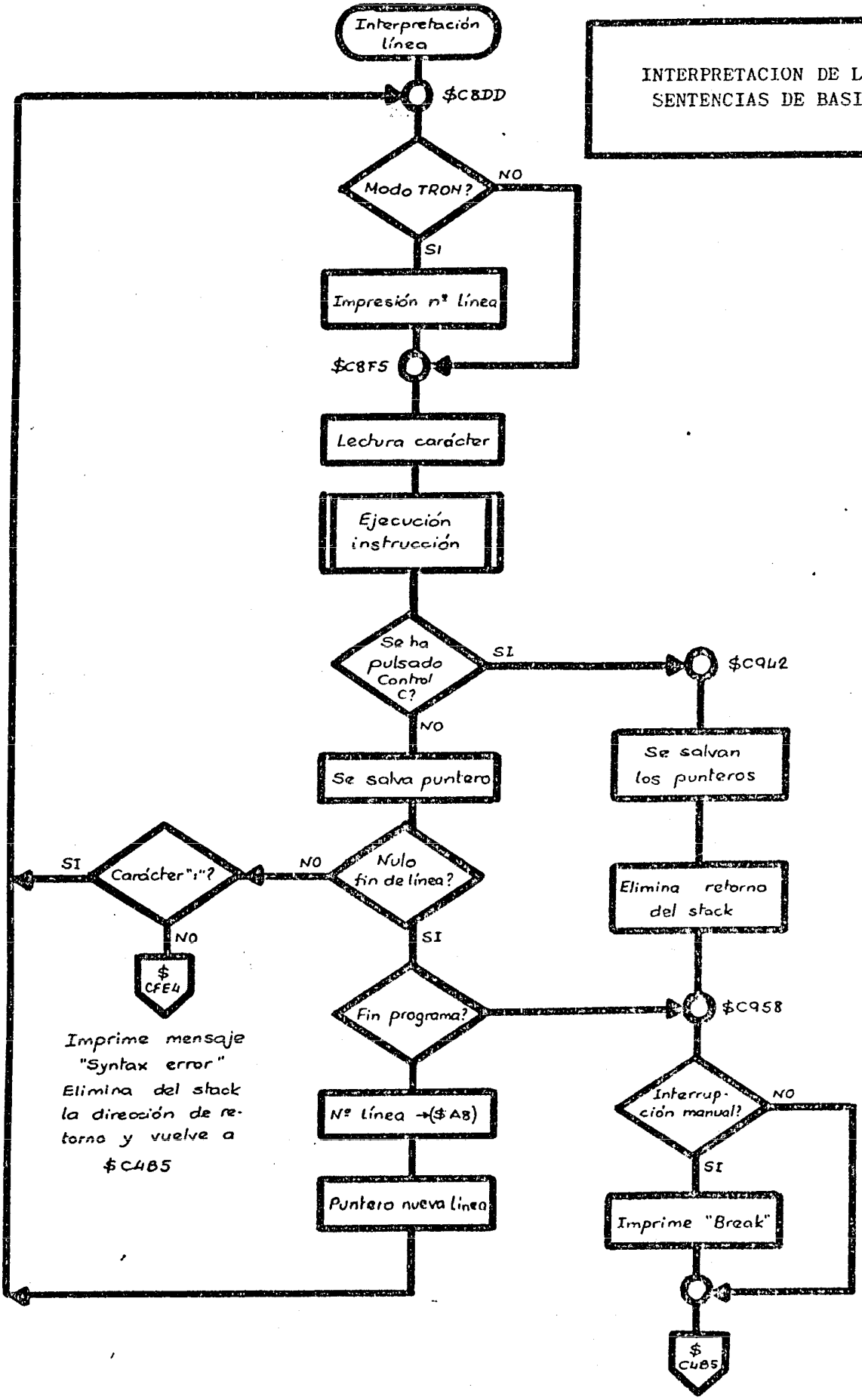
Como en los anteriores casos, el proceso se sigue mejor si simultáneamente se va repasando en el diagrama "Interpretación de las sentencias de Basic".

En modo ejecución directa, el modo TRON queda automáticamente desactivado, para evitar que en la primera parte de la rutina se produjera la impresión de un número sin sentido.

En caso de que la interrupción del programa haya sido provocada mediante un Control C, antes del retorno a la secuencia principal se produce la impresión del mensaje "Break".

Remarquemos que lo que señala el fin de un programa de Basic, es que el byte de peso alto del link siguiente sea nulo (en efecto, la siguiente línea no puede estar en página cero).

INTERPRETACION DE LAS SENTENCIAS DE BASIC



Antes de ver como se produce la ejecución de una instrucción, vamos a repasar un par de conceptos, que nos serán útiles luego para una mejor comprensión.

El primero de ellos es la estructura del propio programa de Basic. Este está formado por una sucesión de líneas. Cada línea empieza por un grupo de cuatro bytes que la identifican: los dos primeros nos señalan la dirección de comienzo de la siguiente línea del programa, mientras que los dos siguientes constituyen el número de la línea.

A continuación vienen las instrucciones, separadas por el carácter dos puntos, para terminar la línea con un byte nulo. Al final del programa, el segundo byte de lo que constituiría el identificador de una eventual siguiente línea es nulo también para indicar este fin (este byte es el final del programa).

Asimismo, antes del primer byte de la primera línea, debe haber también un nulo, por necesidades del programa de interpretación.

Cada instrucción empieza por el token que la identifica, y sigue con los parámetros sobre los cuales se aplica, por tanto, cuando leemos el primer byte de una instrucción de Basic, leemos dicho token. Hay, sin embargo, una excepción a esta norma: es el caso del token correspondiente a LET, que no es necesario en el Basic del ORIC-1.

El interpretador entiende en dicho caso (cuando el primer byte de la instrucción no corresponde a un token) que se trata de un LET. En cambio, en otros ordenadores, el LET es necesario para distinguir la asignación de un valor a una variable (LET A = 3), de una simple comparación lógica (IF A = 3).

El otro punto a considerar es la estructura del stack del 6502. Cada vez que se produce el salto a una subrutina (JSR), la dirección del último byte del JSR es subida al stack (que ocupa de $1FF hacia $100), ocupando dos bytes.

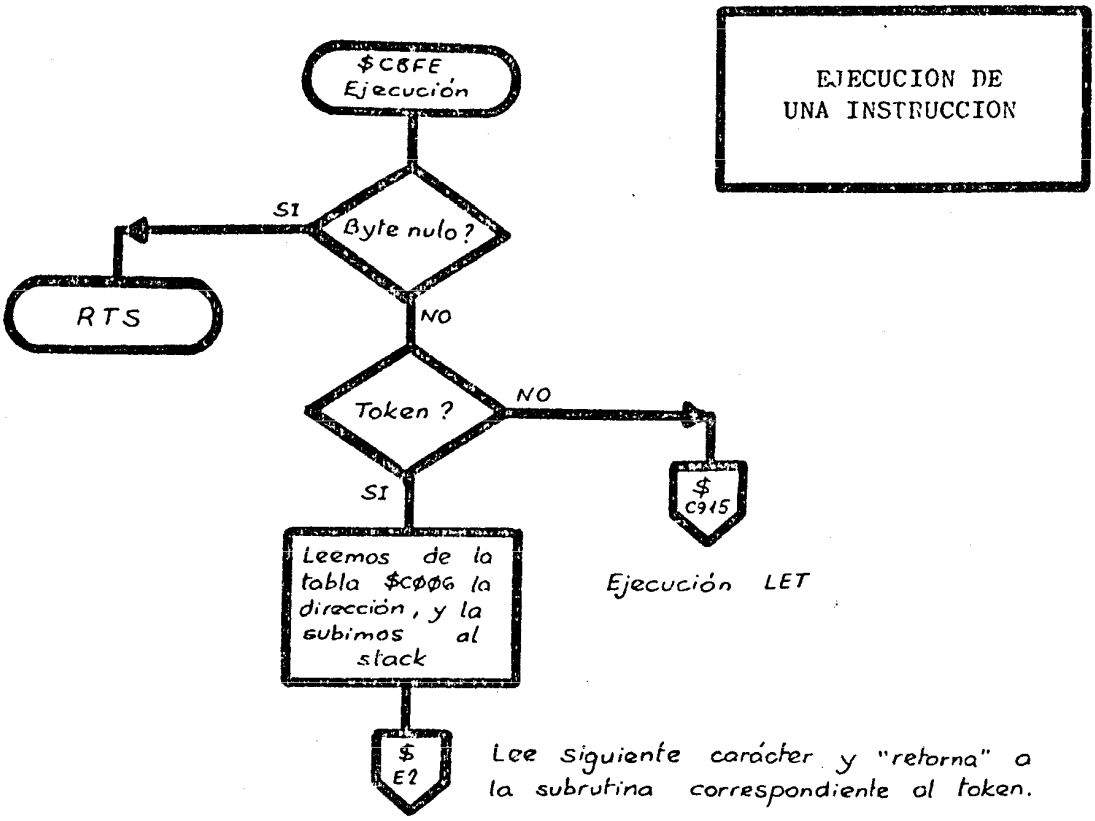
Al término de la subrutina, al encontrar RTS, se coge del stack la dirección allí almacenada, se le incrementa uno, y se coloca en el contador de programa, reanudándose la ejecución desde allí. El control del último byte del stack se lleva a cabo de forma automática en el registro Stack Pointer.

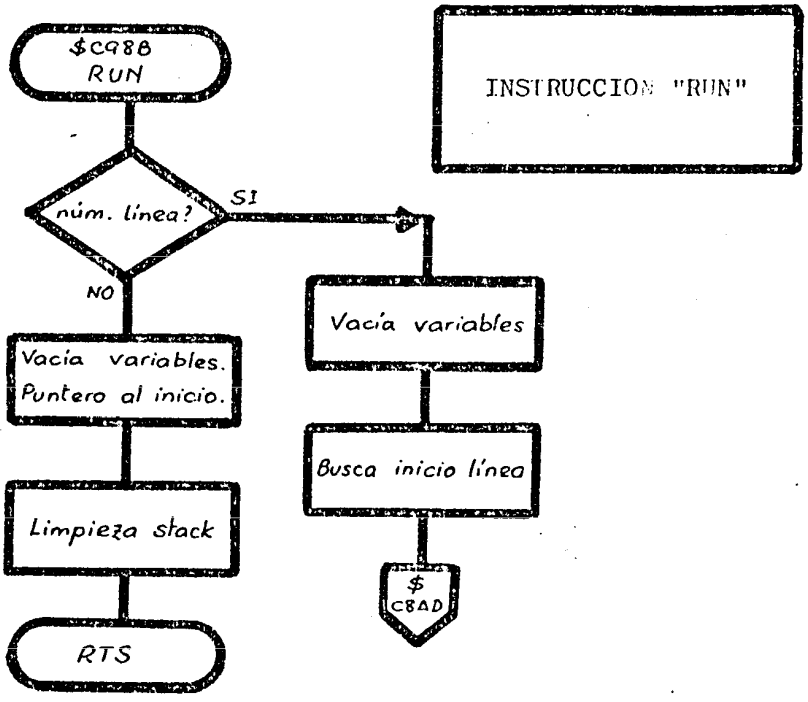
Debe tenerse también en cuenta que el stack es utilizado a menudo para el pase de parámetros entre rutinas, o para guardar los datos de los bucles, o los resultados intermedios de fórmulas complejas.

Hechas estas consideraciones, pasemos ya a ver el funcionamiento de la rutina que ejecuta una instrucción de Basic, y que empieza en $C8FE.

Se comprueba primero si el primer carácter corresponde a un byte nulo, en cuyo caso retornamos sin hacer nada. Luego comprobamos si se trata de un token, de lo contrario saltamos directamente a la subrutina correspondiente a la instrucción LET.

A continuación buscamos, en una tabla que empieza en $C006, la dirección de la correspondiente subrutina (menos uno, por lo antes comentado), y la subimos al stack. Finalmente, leemos el siguiente carácter, y al encontrar el RTS, se producirá un "retorno" a la dirección colocada en la pila, que será la de comienzo de la subrutina en código máquina correspondiente a la instrucción de Basic.





Al retorno, proseguimos la ejecución dentro del programa BASIC. (Éste es considerado como una prolongación de la línea que se estaba interpretando).

Cada subrutina en código máquina debe, pues, leer los parámetros, ejecutar sobre los mismos las operaciones correspondientes, y retornar con el puntero del intérprete corrido al primer byte siguiente al final de la instrucción (que será, habitualmente, el carácter dos puntos, o el nulo de fin de línea).

Todo lo explicado hasta ahora es válido para la ejecución de instrucciones entradas directamente por teclado, pero, cómo se inicia la ejecución de un programa? Simplemente, cuando encontramos la instrucción RUN, por el mecanismo que acabamos de describir, se produce un salto a $C98B. Allí se comprueba si el RUN va seguido de un número de línea, en cuyo caso la misma es buscada y el puntero del intérprete posicionado en el principio de la misma. En caso contrario se posiciona en el principio del Basic.

A continuación se limpia el stack, para eliminar los retornos innecesarios y, al encontrar el RTS, continuará la interpretación como si siguiéramos en la línea de Basic, pero ya desde el programa. A estos efectos, podemos imaginar que un programa de Basic constituye una prolongación de la línea entrada por el teclado que contiene el RUN. Las rutinas de interpretación y ejecución son las mismas.

Mirando la tabla $CØØ6 (y añadiéndo uno a los valores encontrados), hallaremos las direcciones de todas las rutinas correspondientes a las instrucciones del Basic, que son:

token	direcc.	nombre BASIC	token	direc.	nombre BASIC
128 $80	$C941	END	129 $81	$C6A5	EDIT
130 $82	$CFE4	INVERSE	131 $83	$CFE4	NORMAL
132 $84	$CC8C	TRON	133 $85	$CC8F	TROFF
134 $86	$C9E0	POP	135 $87	$D9C6	PLOT
136 $88	$DA16	PULL	137 $89	$D937	LORES
138 $8A	$D8AC	DOKE	139 $8B	$D9FA	REPEAT
140 $8C	$DA16	UNTIL	141 $8D	$C841	FOR
142 $8E	$C824	LLIST	143 $8F	$C832	LPRINT
144 $90	$CE0C	NEXT	145 $91	$CA0A	DATA
146 $92	$CCC9	INPUT	147 $93	$D0F2	DIM
148 $94	$CC0A	CLS	149 $95	$CCFD	READ
150 $96	$CAD2	LET	151 $97	$C9B3	GOTO
152 $98	$C98B	RUN	153 $99	$CA3E	IF
154 $9A	$C91F	RESTORE	155 $9B	$C996	GOSUB
156 $9C	$C9E0	RETURN	157 $9D	$CA61	REM
158 $9E	$E95B	HIMEM	159 $9F	$E974	GRAB
160 $A0	$E994	RELEASE	161 $A1	$E9A9	TEAT
162 $A2	$E9BB	HIRES	163 $A3	$F415	SHOOT
164 $A4	$F418	EXPLODE	165 $A5	$F41B	ZAP
166 $A6	$F412	PING	167 $A7	$E889	SOUND
168 $A8	$E889	MUSIC	169 $A9	$E889	PLAY
170 $AA	$E87D	CURSET	171 $AB	$E87D	CURMOV
172 $AC	$E87D	DRAW	173 $AD	$E87D	CIRCLE
174 $AE	$E87D	PATTERN	175 $AF	$E87D	FILL
176 $B0	$E87D	CHAR	177 $B1	$E889	PAPER
178 $B2	$E889	INK	179 $B3	$C93F	STOP
180 $B4	$CA78	OR	181 $B5	$D89D	WAIT
182 $B6	$E7AA	CLOAD	183 $B7	$E7DB	CSAVE
184 $B8	$D401	DEF	185 $B9	$D894	POKE

token	direcc.	nombre BASIC	token	direc.	nombre BASIC
186 $BA	$CB61	PRINT	187 $BB	$C96E	CONT
188 $BC	$C773	LIST	189 $BD	$C738	CLEAR
190 $BE	$CCBA	GET	191 $BF	$E80D	CALL
192 $CO	$CC89	!	193 $C1	$C719	NEW

Ello nos permite llamar las rutinas desde el código máquina (por ejemplo, podemos comprobar que la dirección de HIRES es $E9BB, haciendo CALL #E9BB. Debemos, sin embargo, hacer algunas aclaraciones:

- La tabla solo llega hasta el token $C1, ya que a partir del $C2 no corresponden a instrucciones, sino a operadores, y parte de otras instrucciones, y son manejados diferentemente.

- Puede parecer extraño que haya coincidencias en la tabla (más de una instrucción con la misma dirección). Ello es debido a dos causas, según los casos:
  a) En el caso de las parejas PULL-UNTIL y POP-RETURN, fijémonos que se trata en realidad de la misma instrucción, con dos nombres distintos.
  b) En el caso de los grupos SOUND-MUSIC-PLAY-PAPER-INK y CURSET-CURMOV--DRAW-CIRCLE-PATTERN-FILL-CHAR, se trata de rutinas que por conveniencias de programación han sido agrupadas (en concreto, para aprovechar el mismo proceso de lectura de parámetros, y por ciertas similitudes entre las mismas).
  En este caso, para desviar la ejecución al punto correcto, nos valemos del hecho que a la entrada de la subrutina el registro Y contiene un valor igual a dos veces la diferencia entre $8Ø y el token de la instrucción.

- Las rutinas que no requieren parámetros pueden ser llamadas mediante JSR (o desde el Basic, mediante CALL), por ejemplo CLS, HIRES, PING, etc. Son necesarias de todos modos, algunas precauciones especiales. Así, CLEAR (y otras), requieren que a la entrada el registro de estado tenga el indicador 7 en set, ya que empieza con un BNE $C764, que va a RTS. Podemos evitarlo entrando en la dirección siguiente.

- Por otra parte, las rutinas que requieren parámetros no pueden ser llamadas de esta forma, ya que intentarían leerlos a partir de la posición actual del puntero del intérprete (fin de la instrucción CALL). Más adelante veremos la forma de utilizarlas.

En la mayoría de los casos, si queremos utilizar una de estas rutinas llamándola desde el código máquina, resulta más ventajoso saltar la fase de lectura de parámetros, a veces la más lenta, e introducir sus valores directamente en los lugares adecuados (véase, a título de ejemplo, la sección dedicada al generador de sonido).

En este caso hay que cuidar que los valores sean válidos, pues no habrá detección de errores, a menos que expresamente la preveamos.

I N T E R R U P C I O N E S

Normalmente, la ejecución de una rutina procede de modo continuo, de una instrucción a la siguiente, con los eventuales saltos provocados por los Branchs y los Jumps. En ocasiones, sin embargo, conviene interrumpir este proceso para llevar a cabo otras tareas, como la lectura del teclado. A este fin, el microprocesador 65Ø2 dispone de una serie de dispositivos de interrupción.

Así, cuando al correspondiente terminal del 65Ø2 llega una señal o "demanda" de interrupción, el contenido del contador de programa y del registro de estado son salvados en el stack, y la ejecución desviada a una rutina especial (rutina de interrupción), finalizada la cual (al encontrar la instrucción RTI) se retorna al programa interrumpido.

Las rutinas de interrupción suelen empezar salvando el contenido de los registros internos, para restaurarlos antes del retorno al programa principal. Al recibir la demanda de interrupción, el microprocesador toma la dirección donde debe continuar del contenido de ciertas posiciones ("vectores"), situadas al final de la ROM.

Hay tres tipos de interrupción, que estudiamos acto seguido:

1) RESET GENERAL (arranque en frío). La demanda de interrupción la produce un circuito especial que se activa en el momento de conectar la máquina. El vector correspondiente se encuentra en las posiciones $FFFC-$FFFD, y contiene la dirección $F42D.
   La rutina que empieza en $F42D provoca la inicialización general, procediendo a una comprobación de la memoria, y colocación de los valores necesarios en la página cero, generador de caracteres, pantalla, etc.

2) RESET PARCIAL (arranque en caliente). Se produce cuando pulsamos el botón situado en la parte inferior de la máquina. Provoca una reinicialización parcial, sin borrar los programas ni las variables.
   Una particularidad de esta interrupción es que el vector, situado en las direcciones $FFFA-$FFFB, contiene el valor $22B, situado en RAM. Por este motivo, podemos modificar el efecto de esta interrupción, cambiando el contenido de $22B-$22C-$22D (que originalmente contiene un JMP $F43Ø).
   Así, por ejemplo, podemos inhibir el efecto de la pulsación del citado botón, simplemente haciendo POKE #22B,64, lo que puede ser útil para ciertos procesos de protección de programas.
   Para restaurar el funcionamiento normal, bastará hacer POKE #22B,76.

3) INTERRUPCION POR EL VIA. El VIA (6522 Versatile Interface Adapter) es el circuito encargado de comunicar el microprocesador con los periféricos (teclado, cassette, impresora, pantalla). Contiene un temporizador que, cada centésima de segundo, envía una demanda de interrupción al microprocesador.
   En este caso el vector está en $FFFE-$FFFF y contiene la dirección $228, también modificable, por tanto. En funcionamiento normal, esta interrupción provoca la lectura del teclado, entre otras funciones.

Vamos a ver un ejemplo de la utilidad de poder modificar la rutina de interrupción, lo que nos servirá para comprobar que, en código máquina, es posible efectuar cosas que serían imposibles desde el Basic.

En efecto, mediante un programa Basic, podemos realizar un indicador horario, que muestre permanentemente la hora actual, en el formato deseado. Ahora bien, ello exige que mientras esté en marcha este programa, no podemos realizar otros trabajos con el ORIC-1.

Pues bien, la rutina que llevaremos a cabo permitirá simultáneamente continuar el funcionamiento normal, y así podremos preparar programas, ejecutarlos, etc. mientras el reloj sigue funcionando. Con el fin de que sea totalmente compatible con este funcionamiento normal, haremos que la indicación horaria aparezca en la linea reservada superior (aquella en la que normalmente aparecen los mensajes Loading y Saving).

A continuación doy un programa de Basic que, al ejecutarlo, se encargará de introducir la rutina en código máquina. Una vez ejecutado, el programa puede ser borrado, pues solo sirve para cargar la rutina.

```
1Ø    A$ ="2ØCAE6A924AØØ48D29Ø28C2AØ2A24Ø8E5FØ42ØØ4E82ØD9CFA5E9
      A4EAA2Ø22Ø2F"
2Ø    A$ = A$ + "F84CØ1CAØ8CE5FØ4DØ21488A48A9648D5FØ4A2Ø7FE82BBB
      D82BBDD57Ø49ØØ9BD"
3Ø    A$ = A$ + "4FØ49D82BBCA1ØEC68AA68284CØ3EC3Ø3Ø3A3Ø3Ø3A3Ø3Ø
      3A3A3A363A3A363AØØ"
4Ø    FOR I = 1 TO LEN (A$) STEP 2
5Ø    A = #4ØØ + (I - 1)/2 : B = VAL("#" + MID$ (A$,I,2))
6Ø    POKE A,B : NEXT
7Ø    S = Ø : FOR D = #4ØØ TO #45E STEP 2
8Ø    S = S + DEEK (D) : NEXT
9Ø    IFS<> 1264179  THEN PRINT "Error escritura"
```

Una vez copiado, haced RUN. Si aparece Ready, todo ha ido bien. Si antes aparece el mensaje de error, repasad las líneas 1Ø, 2Ø y 3Ø. Hecho esto, es aconsejable salvar la rutina máquina haciendo:

CSAVE "RELOJ", A#4ØØ, E#45F

Ahora ya podemos utilizarla. Para ello, supongamos que queremos que el reloj empiece a las 15:35:ØØ. Entonces escribiremos la sentencia:

CALL #4ØØ,15:35:ØØ

Al pulsar el Return, el reloj se pone en marcha. Si queremos que funcione como cronómetro, hacemos CALL #4ØØ,ØØ:ØØ:ØØ. La rutina puede ser llamada desde el teclado o desde un programa, pero la sentencia que la llama debe ser la última de la línea.

El reloj llega hasta 99:59:59 y luego vuelve a empezar en ØØ:ØØ:ØØ. Todo aquello que inhiba las interrupciones, parará el reloj, así al funcionar el casssette, el reloj se detiene (además, los mensajes alteran la indicación horaria).

También las instrucciones de impresión producen algunos retrasos en el reloj, lo que hace que no sea muy exacto, pero de todos modos conserva su utilidad como demostración, o en algunos programas de juego.

La rutina en lenguaje de ensamblaje es la siguiente:

```
4ØØ  2Ø  CA  E6   rell  JSR  $E6CA      inhibimos las interrupciones
4Ø3  A9  24             LDA  #$24       \
4Ø5  AØ  Ø4             LDY  #$Ø4       | cambiamos el vector interrupción
4Ø7  8D  29  Ø2         STA  $229       |
4ØA  8C  2A  Ø2         STY  $22A       /
4ØD  A2  4Ø             LDX  #64        \ inicializamos contador
4ØF  8E  5F  Ø4         STX  $45F       /
412  2Ø  Ø4  E8         JSR  $E8Ø4      desinhibimos interrupciones
415  2Ø  D9  CF         JSR  $CFD9      saltamos la coma
418  A5  E9             LDA  $E9        \
41A  A4  EA             LDY  $EA        | escribimos el horario inicial
41C  A2  Ø2             LDX  #2         |
41E  2Ø  2F  F8         JSR  $F82F      /
421  4C  ØD  CA         JMP  $CAØD      saltamos al final de la línea
424  Ø8             int  PHP            salvamos el registro de estado
425  CE  5F  Ø4         DEC  $45F       \ cada 1ØØ veces, cambio horario
428  DØ  21             BNE  sorp       /
42A  48                 PHA             \
42B  8A                 TXA             | salvamos registros
42C  48                 PHA             /
42D  A9  64             LDA  #1ØØ#$64   \ reinicializamos el contador
42F  8D  5F  Ø4         STA  $45F       /
432  A2  Ø7             LDX  #7         \
434  FE  82  BB   llaç   INC  $BB82,X    / incrementamos la indicación horaria
437  BD  82  BB         LDA  $BB82,X    \
43A  DD  57  Ø4         CMP  $457,X     |
43D  9Ø  Ø9             BCC  sorx       | si hemos superado el máximo para
43F  BD  4F  Ø4         LDA  $44F,X     | esta cifra, incrementamos la in-
442  9D  82  BB         STA  $BB82,X    | mediata a la izquierda
445  CA                 DEX             |
446  1Ø  EC             BPL  llaç       /
448  68             sorx PLA            \
449  AA                 TAX             | restauramos los registros
44A  68                 PLA             |
44B  28             sorp PLP            /
44C  4C  Ø3  EC         JMP  $ECØ3      y vamos a la rutina normal
44F  3Ø  3Ø  3A  3Ø  3Ø  3A  3Ø  3Ø     cifras mínimas
457  3A  3A  3A  36  3A  3A  36  3A     cifras máximas
45F  ØØ                                 contador
```

De $4ØØ a $423 es la parte que se ejecuta cuando llamamos la rutina, al poner el horario inicial. De $424 a $44E es la parte que se ejecuta cada centésima de segundo, aunque sólo cada segundo se cambia la indicación horaria. Finalmente, de $44F a $45F son los distintos valores utilizados por la rutina.

C A S S E T T E

El interfase para cassette contenido en el ORIC-1 constituye un método sencillo y económico (y además, hasta la llegada de los discos, el único) para la salvaguarda de programas y ficheros. Esta simplicidad parece pagarse a costa de algunos inconvenientes. No obstante, si alguno de ellos puede ser cierto, la mayoría pueden subsanarse con un adecuado conocimiento de su funcionamiento.

El presente capítulo pretende describir este funcionamiento, y presentar una serie de rutinas de utilidad (verificación, identificación, etc.), así como los elementos para poder construir fácilmente uno mismo la rutina que precise para su caso particular.

Antes de pasar, sin embargo, al estudio del software, recordaré los consejos más corrientes sobre la utilización del magnetófono:

- si se va a adquirir uno exclusivo para esta misión, conviene probar varios modelos antes de decidir cuál adquirir. En general, los modelos más sencillos suelen dar mejores resultados que los de elevada calidad. También pueden encontrarse, en tiendas especializadas, modelos especiales para ordenador personal.

- el cabezal debe mantenerse siempre limpio y bien alineado. Puede comprobarse escuchando una cinta de audio y comprobando la nitidez de los agudos.

- Conviene que el altavoz incorporado quede desconectado al colocar el enchufe de conexión al ORIC-1 (el sonido de un fichero en cassette no resulta excesivamente agradable).

- Debe tantearse el volumen hasta obtener la máxima fiabilidad. Puede empezarse ajustándolo a los dos tercios del recorrido. Si existe control de tono, se colocará al máximo de agudos.

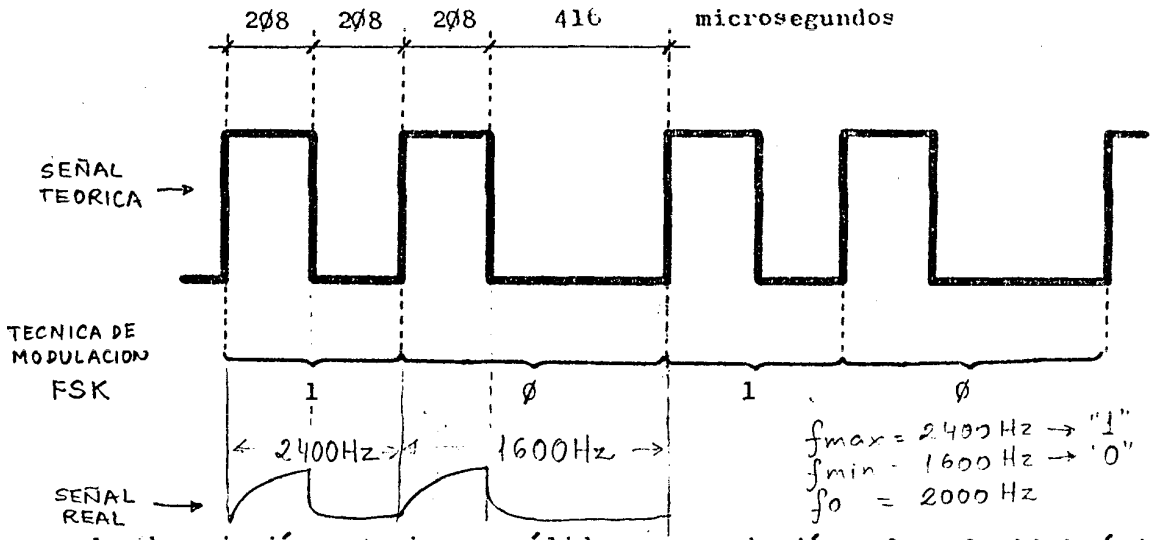
- Las conexiones deben ser sólidas, y los conductores alejados de cables eléctricos.

- Deben emplearse cintas de audio de buena calidad, que no se atasquen (el sistema es especialmente sensible a las fluctuaciones de velocidad).

Pasemos ahora a ver cómo está representado físicamente un programa en la cinta del cassette. Puesto que un programa es una sucesión de bytes, y un byte está compuesto de bits, empezaremos viendo cómo representamos un bit.

Como que en una cinta magnética podemos grabar impulsos, será esta la forma en que representemos un bit. Cada impulso dura 2Ø8 microsegundos, y está seguido de un espacio hasta el próximo impulso. Este espacio dura otros 2Ø8 microsegundos si queremos representar un bit 1, o el doble (416 microsegundos) para representar el Ø.

En la siguiente figura representamos una sucesión de bits: 1, Ø, 1, Ø, tal como aparecerían en la cinta magnética.



La descripción anterior es válida para grabación a la velocidad rápida. En el caso de la velocidad lenta, el sistema es distinto. Cada bit contiene 4 u 8 impulsos de distinta duración, según se trate de un 1 ó un Ø. A la lectura, comparamos el número de impulsos con 6, y así eliminamos un cierto porcentaje de errores, aumentando la fiabilidad.

Observemos que en el caso de una grabación continuada de bits 1 (a velocidad rápida), el período es de 2x208 = 416 microsegundos, lo que equivale a una frecuencia de 24ØØ Hz, por ello decimos que la velocidad es de 24ØØ baudios (bits/segundo).

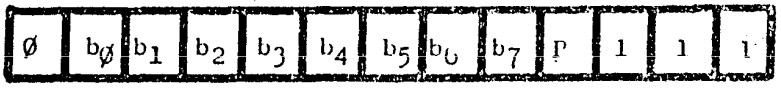
Sin embargo, si se trata de ceros, el período es un 5Ø % más largo, y la velocidad de transmisión menor. Podemos comprobarlo con el siguiente experimento:

En primer lugar, hagamos FOR A = #5ØØØ TO #5FFF : POKE A,Ø : NEXT para crear una zona de memoria de 4K con todos los bits nulos. Después hagamos CSAVE "", A#5ØØØ,E#5FFF y cronometremos el tiempo que dura el mensaje "Saving". Veremos que es de unos 31 segundos.

Ahora empezaremos haciendo FOR A = #5ØØØ TO #5FFF : POKE A,255 : NEXT para llenar la misma zona de memoria con bits todos 1. Si repetimos la misma prueba, veremos que el "Saving" dura ahora solo unos 24 segundos.

No obstante, si dividimos los 24 segundos entre los 4x1Ø24 bytes, veremos que nos da un tiempo para cada byte de unos 586Ø microsegundos, bastante superior a lo que resultaría de la sucesión de 8 bits (8x416 = 3328).

En efecto, el ORIC-1 no graba solamente los ocho bits componentes, sino un bit inicial nulo, luego los ocho bits empezando por el de peso más bajo, luego un bit de paridad, y finalmente tres bits con valor 1. En total 13 bits, según el siguiente esquema:



Veamos finalmente la estructura del fichero completo. Consta, por este orden, de:

- 259 bytes $16, que constituyen el tono inicial, que sirve, durante la grabación, para ajustar el control automático de volumen, y, durante la lectura, para permitir la sincronización del proceso (saber dónde empieza el primer bit de cada byte).

- un byte $24, que señala el final del tono, y puede ser utilizado como identificador del tipo de fichero.

- 9 bytes que contienen los punteros de las direcciones inicial y final, la indicación de si es código máquina o Basic, y si se trata de un programa de ejecución automática.

- el nombre del programa, en código ASCII, seguido de un byte nulo que señala el fin del nombre.

- el contenido del fichero propiamente dicho, byte a byte.

El comando CSAVE efectua lo siguiente:

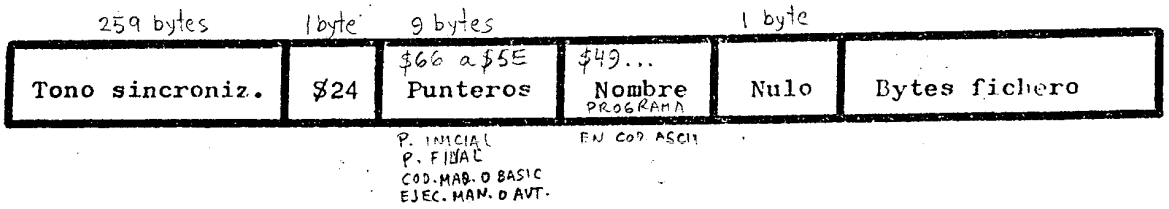
- coloca los punteros de inicio y fin del Basic en los de inicio y fin de fichero, como valores por defecto.

- lee los parámetros: nombre del programa, eventuales direcciones de inicio y fin, auto-load, velocidad de grabación; y los coloca en los lugares adecuados.

- bloquea el teclado y prepara el circuito de interfase para la grabación, activa el relé de control remoto del cassette.

- enviamos los 259 bytes $16, el byte $24, los punteros, el nombre del programa, y todos los bytes del mismo.

- devolvemos el control al teclado y, si estábamos en programa, volvemos a la ejecución del mismo (pero si era ejecución directa por teclado, se da por finalizada la línea, aunque contuviera más instrucciones. Ello se hace así porque la rutina altera el buffer de teclado).



El comando CLOAD, por su parte, actua así:

- colocamos el puntero de inicio del Basic como puntero inicial del fichero por defecto.

- leemos los parámetros, y los pasamos a las direcciones adecuadas.

- bloqueamos el teclado, y preparamos el circuito para la lectura, activando el relé.

- escribimos el mensaje "Searching".

- intentamos sincronizar el tono inicial.

- esperamos un byte con valor $24.

- leemos del cassette los 9 bytes siguientes, y los colocamos en los punteros (de $66 a $5E).

- leemos el nombre del programa, y lo pasamos a partir de $49, seguido de un nulo.

- comprobamos la coincidencia del nombre con el que habíamos colocado a partir de $35. Si no se produce, volvemos a empezar.

- escribimos "Loading" nombre del programa.

- leemos del cassette cada byte, y lo vamos pasando a la memoria, incrementando el puntero, hasta llegar al final.

- si era auto-load, saltamos a ejecución a partir del principio del programa.

- en caso contrario, volvemos al control por teclado (si estábamos ejecutando un programa, no retornamos al mismo).

Al final del capítulo concretaré, para el lector interesado en ello, las direcciones de las distintas rutinas y punteros. No obstante, antes describiré algunas aplicaciones de todo lo anterior.

En ocasiones escribimos programas Basic que contienen subrutinas en código máquina. Resultaría interesante poder salvar en un único fichero en cassette ambas partes (Basic y máquina). Pues bien, si la parte en código máquina no excede 256 bytes, hay un método muy simple para lograrlo.

Escribimos la parte en código máquina en la zona $4ØØ - $4FF. Cuando queramos salvar el programa, haremos: CSAVE "nombre",A#4ØØ. De este modo se salvará desde $4ØØ hasta el final del Basic (valor final por defecto). Para recuperar el programa, podremos hacer un CLOAD "nombre" totalmente normal.

Un fichero en cassette siempre va a parar a la misma dirección en que estaba cuando se creó, aunque en el CLOAD especifiquemos una dirección distinta.

Hay un efecto marginal de la instrucción CLOAD que debemos tener en cuenta porque puede causarnos problemas. Se trata de que después de un CLOAD, el puntero de final del Basic se posiciona al final del fichero leido.

Si este fichero era efectivamente un programa Basic, todo es correcto, pero, y ahí radica el problema, si era una rutina en código máquina, la alteración del puntero de fin de Basic puede provocar errores de ejecución.

Así, si el puntero se coloca después del HIMEM, cualquier intento de ejecutar el programa, o añadir alguna línea, se resolverá en la aparición del mensaje "OUT OF MEMORY ERROR".

Si, en cambio, el puntero queda por debajo del principio del Basic (caso frecuente cuando leemos rutinas que hemos escrito a partir de $4ØØ), las variables plancharán las líneas del Basic, con resultados imprevisibles.

Para evitarlo, podemos hacer dos cosas:

- Si hemos de leer una parte en Basic y otra en máquina, leeremos primero esta última y luego el Basic. Así el puntero queda finalmente en el lugar adecuado.

- Si el programa Basic ya existe, y queremos leer una rutina en máquina, podemos hacer:PRINT DEEK (#9C), luego CLOAD ..., y después DOKE #9C,valor. De este modo se restaura el puntero.
  Recordemos que estas tres sentencias no pueden ser incluidas en una sola línea, pues la ejecución se detiene después del CLOAD.

En ocasiones, durante la lectura de un fichero en cassette, se produce un error de lectura de un byte. Si el error es en un carácter o instrucción, veremos un cambio en tal lugar. Sin embargo, si el error se produce en los bytes iniciales de una linea, que contienen la dirección de la siguiente, suele ocurrir que aparecen al listar el programa una sucesión de letras U.

En este caso, podemos rehacer las uniones interlíneas mediante la ejecución de CALL #C56F. o Ø?ØRUN

Y ya que hablamos de errores de lectura, quizás el principal problema de la salvación de programas en cassette estriba en el hecho de no disponer de un sistema de verificación que permita conocer si la lectura se ha efectuado correctamente o no.

Ello sería muy interesante, pues nos permitiría trabajar tranquilamente a 24ØØ baudios, con el consiguiente confort. Pues bien, a continuación describiré una pequeña rutina que cumple esta función, fácil de incorporar a todos nuestros programas (sólo ocupa 56 bytes), y que permite trabajar con una seguridad total.

Se trata de una rutina que, inmediatamente después de la lectura, imprime un checksum del programa Basic leído. Nos basta comprobar que sea el mismo que habremos anotado en el momento de la grabación para estar seguros que la lectura ha sido correcta (o repetirla, en caso contrario).

La rutina en sí, en código máquina, ocupa de $4C8 hasta $4FF. Para crearla por primera vez, escribiremos el siguiente programa Basic:

```
1Ø A$ = "A2Ø48673AØØØ847284708471988818657Ø857Ø9ØØ2E671A2Ø3C8DØØ2E673B1"
2Ø A$ = A$ + "72DØECCADØF498AAA5732ØC1EØA92C2Ø4DCCA571A67Ø4CC1EØ"
3Ø FOR I = 1 TO LEN (A$) STEP 2
4Ø A = #4C8 + (I - 1)/2 : B = VAL ("#" + MID$ (A$,I,2))
5Ø POKE A,B : NEXT
6Ø S = Ø : FOR D = #4C8 TO #4FF STEP 2
7Ø S = S + DEEK (D) : NEXT
8Ø IF S<> 916324 THEN PRINT "Error escritura. Repasa A$"
```

Una vez escrito, hacéis RUN. Si todo va normal, ya tenéis la rutina en código máquina en su lugar. Conviene que la salvéis haciendo:

CSAVE "VERIF", A#4C8, E#4FF, S

En lo sucesivo, cuando queráis salvar un programa en Basic, leeréis primero la cinta con el "VERIF", y luego salvaréis el programa Basic junto con la rutina haciendo: CSAVE "nombre", A#4C8, AUTO.

Al mismo tiempo, hacéis CALL #4C8 y anotáis en la tapa del cassette las dos cantidades que aparezcan. Eso es todo.

Cuando luego leáis el programa, al terminar la lectura, aparecerán las mismas cantidades, si la lectura ha sido correcta. Para los interesados en la rutina, su contenido, en lenguaje de ensamblaje, es el siguiente:

```
        LDX  #4
        STX  $73        ;byte de peso alto del puntero
        LDY  #Ø
        STY  $72        ;byte de peso bajo del puntero
        STY  $7Ø        ;byte de peso bajo de la suma
        STY  $71        ;byte de peso alto de la suma
        TYA
        DEY             ;dirección inicial $4FF
suma    CLC
        ADC  $7Ø        ;incrementamos suma
        STA  $7Ø
        BCC  salto
        INC  $71
salto   LDX  #3         ;X contador de bytes seguidos nulos
dir     INY             ;incrementamos puntero
        BNE  lect
        INC  $73
lect    LDA  ($72),Y    ;leemos contenido byte
        BNE  suma
        DEX
        BNE  dir        ;si no hemos acabado, siguiente dirección
        TYA
        TAX
        LDA  $73
        JSR  $EØC1      ;imprimimos última dirección
        LDA  #44
        JSR  $CC4D      ;imprimimos una coma
        LDA  $71
        LDX  $7Ø
        JMP  $EØC1      ;imprimimos el checksum
```

El programa cargador no tiene otra misión que introducir fácilmente la rutina en código máquina y, por tanto, una vez ejecutado puede ser borrado. Al leer "VERIF" desde el cassette, recordar lo dicho sobre la alteración del puntero final del Basic.

Las rutinas de la ROM que escriben los mensajes "Searching", "Saving" y "Loading", no tienen en cuenta comprobar si la máquina está en modo HIRES, e imprimen el mensaje en una posición fija de la memoria. Si estamos en HIRES, como que esta posición corresponde al interior de la pantalla, en lugar del mensaje, aparece una fina línea horizontal punteada (hacia los dos tercios desde la parte superior de la pantalla).

Es corriente en muchas aplicaciones la actualización de ficheros en cassette. La programación de un trabajo de este tipo se enfrenta (en el ORIC-1) con dos problemas principales:

- no existe una instrucción específica para salvar (o leer) ficheros de datos numéricos o alfabéticos. Solo podemos salvar programas en Basic o fragmentos de memoria concretados por sus direcciones inicial y final.

- si dentro de un programa se incluye una instrucción de lectura de un fichero en cassette (CLOAD), la ejecución del programa se interrumpe inmediatamente después de efectuada la lectura.

Vamos a ver la manera de subsanar ambos inconvenientes.

Es factible crear rutinas en código máquina que permitan salvar y leer tablas numéricas o alfabéticas (en la revista del ORIC apareció una para tablas numéricas). No obstante, muchas veces resulta más simple lo siguiente:

Si queremos, por ejemplo, salvar la tabla A$(I), para los valores de I = Ø hasta I = 99, suponiendo que la longitud media de cada string sea de unos diez caracteres, ello quiere decir que necesitamos una zona de memoria libre de 1ØØ x 11 = 11ØØ bytes (el onceavo byte sirve para marcar el final del string). Conviene dejar un margen de seguridad.

Reservaremos, encima del HIMEM, una zona suficiente para ubicar estos 11ØØ bytes. Por ejemplo, haciendo HIMEM #9ØØØ tendremos más que lo que nos hace falta.

Entonces traspasaremos a esta zona la tabla, haciendo:

```
1Ø  D = #9ØØØ
15  FOR I = Ø TO 99
2Ø      A$ = A$(I)
25      FOR J = 1 TO LEN(A$)
3Ø          C = ASC(MID$(A$,J,1))
35          POKE D,C : D = D + 1
4Ø      NEXT J
45      POKE D,Ø : D = D + 1
5Ø  NEXT I
55  POKE D,Ø
```

Evidentemente, los números de línea pueden ser cualesquiera, y las instrucciones pueden ser agrupadas en menos líneas (las he dispuesto aquí así simplemente para una mejor comprensión).

Una vez transformada así la tabla, podemos salvarla en un fichero de cassette simplemente haciendo:

```
CSAVE "nombre", A#9ØØØ, ED
```

Cuando queramos recuperar la tabla, primero copiamos el fichero, mediante:

```
CLOAD "nombre"
```

y luego transformamos el fichero a la tabla original, mediante la rutina inversa, por ejemplo:

```
1Ø  F = -1 : D = #9ØØØ
15  REPEAT
2Ø      F = F + 1 : F$ = ""
25      REPEAT
3Ø          K = PEEK (D)
35          IF K THEN F$ = F$ + CHR$(K)
4Ø          D = D + 1
45      UNTIL K = Ø
5Ø      A$(F) = F$
55  UNTIL F$ = ""
```

Como antes, los números de línea pueden ser variados, y las instrucciones condensadas (pero cuidado con el IF de la línea 35). Esta rutina requiere que se haya dimensionado previamente A$(1ØØ), un elemento más que los necesarios.

Subsiste el segundo problema que habíamos mencionado, es decir, al hacer el CLOAD se interrumpe la ejecución del programa.

Para solventarlo, la manera más simple consiste en redactar una pequeña rutina en máquina, que podemos incluir en una línea de REM. Es:

```
5Ø6  Ø8          PHP
5Ø7  2Ø  D9  CF  JSR  $CFD9    ;saltamos la coma
5ØA  2Ø  25  E7  JSR  $E725    ;leemos los parámetros
5ØD  2Ø  CA  E6  JSR  $E6CA    ;preparamos el VIA
51Ø  2Ø  A8  E4  JSR  $E4A8    ;leemos fichero
513  2Ø  Ø4  E8  JSR  $E8Ø4    ;devolvemos control al teclado
516  28          PLP
517  6Ø          RTS
```

Para incluirla en el programa Basic, hacemos que la primera línea sea:

```
Ø  REM ABCDEFGHIJKLMNOPQR          (línea de REM con 18 caracteres)
```

y luego, mediante POKEs, introducimos a partir de la dirección $5Ø6, los bytes decimales: 8, 32, 217, 2Ø7, 32, 37, 231, 32, 2Ø2, 23Ø, 32, 168, 228, 32, 4, 232, 4Ø, 96. Es decir: POKE #5Ø6,8 : POKE #5Ø7,32 : etc.

Cuando dentro del programa Basic queramos leer el fichero, en lugar de CLOAD haremos: CALL #5Ø6,"nombre" (,S para velocidad lenta), y se producirá la lectura normal, pero retornando el control al programa Basic, al terminar.

Si queremos comprobar la correcta introducción de la rutina, podemos hacer:

```
S = Ø : FOR D = #5Ø6 TO #517 STEP 2 : S = S + DEEK(D) : NEXT : PRINT S
```

el resultado a obtener debe ser 273890.

En ocasiones puede resultar interesante poder saber qué contiene una cassette, sin necesidad de tener que leer el correspondiente fichero. Para ello podemos valernos de la siguiente rutina, que lee solo el cabezal del fichero, y nos imprime en la pantalla el nombre del citado fichero, y las direcciones de principio y fin del mismo.

Primero daré un programa de Basic para introducir la rutina:

IDENT

```
1Ø  A$ = "A9Ø32CA9ØØ8567Ø82ØCAE62Ø96E62Ø3ØE6C924DØF9A2Ø92Ø3ØE6955D"
15  A$ = A$ + "CADØF82Ø3ØE6FØØ59549E8DØF695492ØØ4E828A949AØØØ2ØEDCB"
2Ø  A$ = A$ + "A92C2Ø12CCA56ØA65F2ØC1EØA92C2Ø12CCA562A6612ØC1EØ4CFFE7ØØ"
25  FOR I = 1 TO LEN(A$) STEP 2
3Ø  A = #4ØØ + (I - 1)/2 : B = VAL("#" + MID$(A$,I,2))
35  POKE A,B : NEXT
4Ø  S = Ø : FOR D = #4ØØ TO #451 STEP 2
45  S = S + DEEK(D) : NEXT
5Ø  IF S <> 1294634 THEN PRINT "Error. Repasa A$"
```

Una vez ejecutado el programa, puede ser borrado. Para salvar la rutina en máquina hacer CSAVE "IDENT", A#4ØØ, E#451.

Cuando queramos identificar un fichero en cassette, haremos CALL #4ØØ (si es velocidad lenta) o CALL #4Ø3 (si es velocidad rápida), y pondremos en marcha la cinta. Al encontrar el principio del fichero, se imprimirán en la pantalla el nombre del programa, la dirección del primer byte y la del último.

La rutina, en lenguaje de ensamblaje, es como sigue:

```
lento  LDA  #3        ;velocidad lenta
       BIT (rap LDA #Ø);velocidad rápida
       STA  $67
       PHP
       JSR  $E6CA     ;preparamos VIA
       JSR  $E696     ;sincronizamos tono inicial
ini    JSR  $E63Ø     ;leemos un byte
       CMP  #$24      ;comprobamos si es el inicial
       BNE  ini       ;si no, leemos el siguiente
       LDX  #9
punt   JSR  $E63Ø     ;leemos los punteros
       STA  $5D,X     ;y los pasamos a página cero
       DEX
       BNE  punt
nomb   JSR  $E63Ø     ;leemos carácter
       BEQ  salto     ;si es nulo, saltamos
       STA  $49,X     ;lo pasamos a página cero
       INX
       BNE  nomb
salto  STA  $49,X
       JSR  $E8Ø4     ;devolvemos control al teclado
       PLP
       LDA  #$49      ;puntero nombre, peso bajo
       LDY  #Ø        ;puntero nombre, peso alto
       JSR  $CBED     ;imprimimos nombre
       LDA  #',
       JSR  $CC12     ;imprimimos la coma
      (...)
```

```
LDA  $6Ø      ;inicio, peso alto
LDX  $5F      ;inicio, peso bajo
JSR  $EØC1    ;imprimimos dirección
LDA  #',
JSR  $CC12    ;imprimimos la coma
LDA  $62      ;final, peso alto
LDX  $61      ;final, peso bajo
JSR  $EØC1    ;imprimimos dirección
JMP  $E7FF    ;volvemos a espera teclado
```

Como vemos, podemos construir rutinas a medida para cada necesidad. Para facilitar esta tarea, a continuación describo las direcciones de mayor utilidad relativas al cassette:

en página cero:

$35-$48	nombre del fichero a buscar, seguido de un nulo
$49-$5D	nombre del fichero encontrado, seguido de un nulo
$5F-$6Ø	dirección inicio fichero
$61-$62	dirección final fichero
$63	auto-load
$64	código máquina

en la ROM:

$E4A8	lee un fichero completo desde el cassette. Exige preparación previa del VIA, y los parámetros en página cero.
$E53E	rutina que retorna al Basic en caso de error durante una lectura de un fichero en cassette.
$E554	incrementa el puntero para la lectura y posiciona carry.
$E563	borra el mensaje de la linea reservada superior.
$E56E	escribe el nombre del programa en la línea reservada.
$E576	escribe un mensaje en la línea reservada superior.
$E57B	salva un fichero completo en cassette.
$E5C6	salva un byte en cassette.
$E5F3	salva un bit en cassette.
$E63Ø	lee un byte desde el cassette y lo pasa a $2F.
$E65E	lee un bit desde el cassette.
$E696	sincroniza el tono inicial.
$E6BA	escribe en el cassette el tono inicial.
$E6CA	inhibe interrupciones y prepara el VIA para el trabajo con el cassette.
$E6FØ	comprueba la coincidencia del nombre buscado con el del fichero encontrado.
$E725	lee los parámetros para el CLOAD y el CSAVE.
$E7AA	rutina CLOAD.
$E7DB	rutina CSAVE.
$E8Ø4	desinhibe interrupciones, devolviendo el control al teclado.

El timer 1 del VIA es el que se encarga de producir los impulsos de 2Ø8 microsegundos, para conformar o controlar los impulsos a través del bit 7 del Port B.

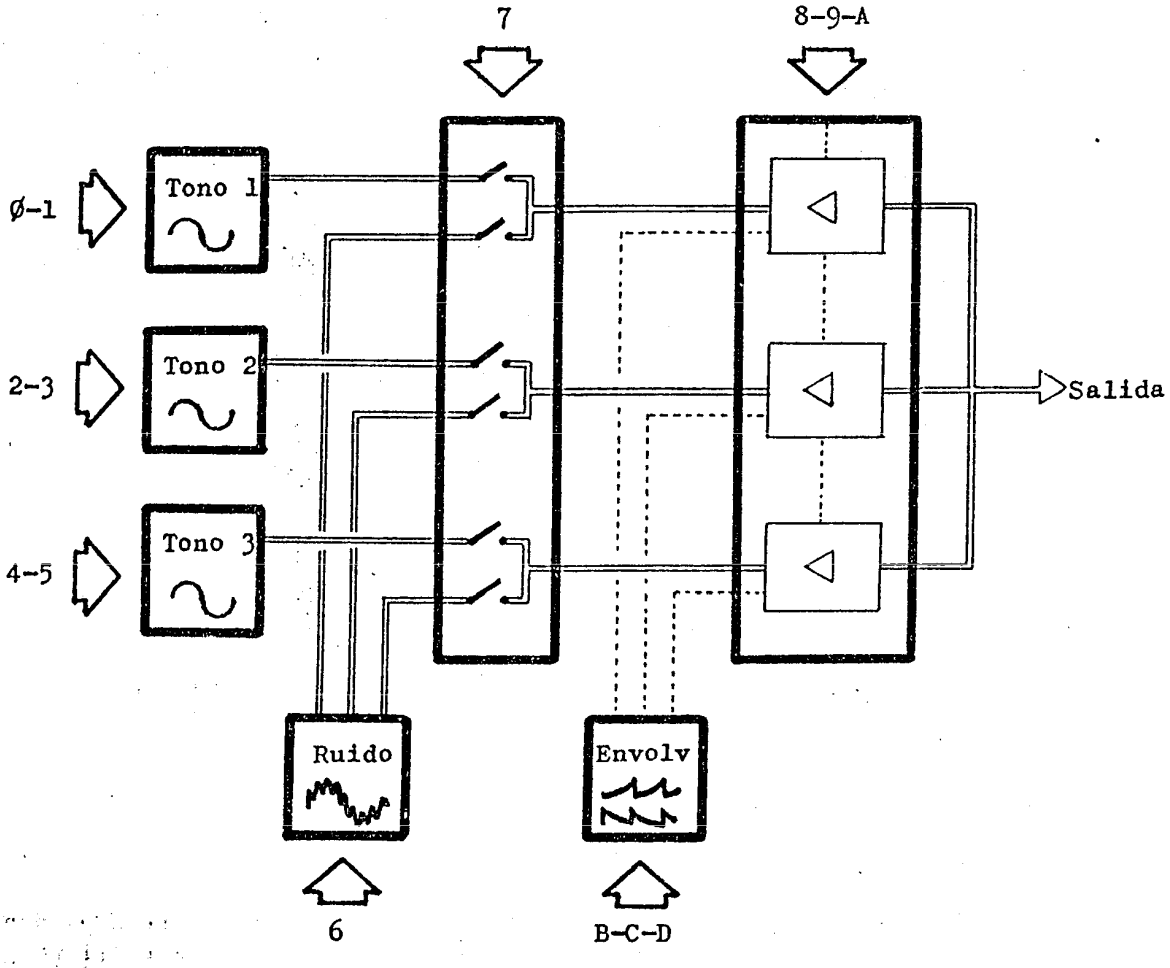
Nota: Podemos adaptar el programa de lectura de tablas para valores numéricos, mediante las instrucciones STR$ y VAL.

S O N I D O

En algunos microordenadores es el propio microprocesador el que se encarga de la producción del sonido. Ello implica que, mientras tanto, la unidad central queda imposibilitada para ejecutar otras tareas, lo que representa un inconveniente para programas de juegos y aplicaciones similares.

El ORIC-1, en cambio, dispone de un circuito independiente especializado en la generación del sonido. De este modo, una vez que el microprocesador ha enviado las órdenes adecuadas a dicho circuito, queda libre para otras tareas, como gestión de la pantalla, cálculos, o, simplemente, preparación de los datos para el próximo sonido a producir.

Además, el generador de sonido contenido en el ORIC-1 dispone de una serie de posibilidades que le permiten obtener combinaciones de sonido muy elaboradas, ya que dispone de tres generadores de tono, un generador de ruido, un generador de envolventes preprogramadas, y tres canales de amplificación independientes. Su esquema es el siguiente:



Como vemos, dispone de 14 registros de control, numerados de Ø a D (en hexadecimal). La misión de cada uno de ellos es:

Registro	Contenido
Ø	Período canal 1, peso bajo
1	Período canal 1, peso alto
2	Período canal 2, peso bajo
3	Período canal 2, peso alto
4	Período canal 3, peso bajo
5	Período canal 3, peso alto
6	"Coloración" del ruido
7	Control autorización canales
8	Volumen canal 1
9	Volumen canal 2
A	Volumen canal 3
B	Período envolvente, peso bajo
C	Período envolvente, peso alto
D	Tipo envolvente

Vamos a concretar la utilización de cada registro:

- Registros Ø-5: período de cada canal, expresado en microsegundos. Recuérdese que el período es el inverso de la frecuencia. Así, por ejemplo, el máximo período posible, $FFFF (en decimal: 65535), corresponderá a una frecuencia de:

$$\frac{1}{0{,}065535} = 15 \text{ Hz}$$

  Evidentemente, no es necesario especificar el período de aquellos canales que no vayamos a utilizar.

- Registro 6: sus cinco bits de peso bajo determinan la "coloración", o tono principal del ruido. El valor Ø corresponde al ruido más agudo, y el valor 31 al más grave.
  Los tres bits restantes no son tomados en consideración, así las instrucciones: SOUND 4,Ø,7 y SOUND 4,32,7 son equivalentes.

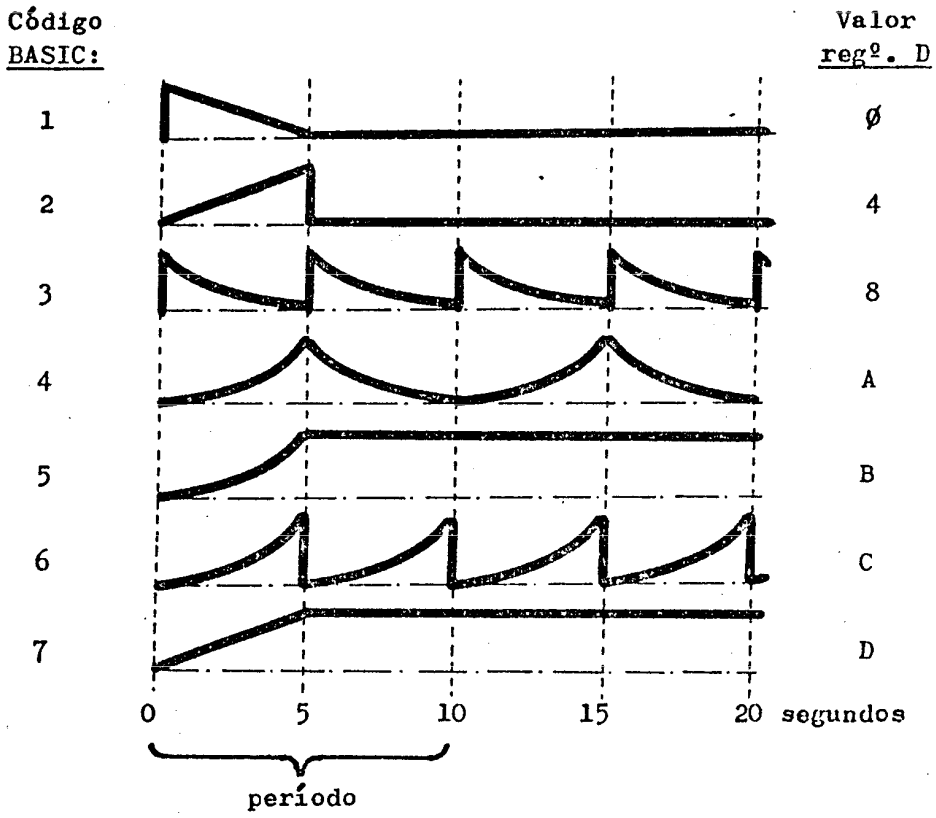
- Registro 7: sus 6 bits de peso bajo determinan qué canales quedan activos para el tono y cuáles para el ruido. Su estructura es:

$$X \; X \; \overline{N}_3 \; \overline{N}_2 \; \overline{N}_1 \; \overline{T}_3 \; \overline{T}_2 \; \overline{T}_1$$

  Donde $\overline{N}_i$ representa el bloqueo del canal "i" para el ruido, y $\overline{T}_i$ lo mismo para el tono.
  Así, por ejemplo, si deseamos que por el canal 1 pase un tono, por el canal 2 tanto tono como ruido, y por el canal 3 nada, el valor a incluir en el registro 7 será, en binario: ØØ1Ø11ØØ (en decimal, 44).

- Registros 8-A: volumen de cada canal, de 1 a 15. Si en lugar de un volumen constante, deseamos que el sonido evolucione siguiendo una de las envolventes preprogramadas, debemos introducir el valor 16 (en hexadecimal: $1Ø).

- Registros B-C: período de la envolvente, en milisegundos. A resaltar que se entiende por período el conjunto de dos rampas; así, si especificamos un valor de 10000, las distintas envolventes serán:



- Registro D: sus cuatro bits de peso bajo determinan el tipo de envolvente, de entre las preprogramadas, según el código indicado en la figura superior. Obsérvese que este código es distinto del utilizado en las sentencias del BASIC.

Vemos, pues, que colocando los valores adecuados en los distintos registros del circuito generador de sonido podremos obtener todos los efectos sonoros de que este es capaz, con las ventajas de mayor rapidez y control total que el código máquina nos proporciona.

Ahora bien, estos registros no forman parte de la memoria, y no son, por tanto, directamente accesibles. Para modificar los mismos es preciso pasar por el circuito de interfase VIA 6522.

Ello resulta, no obstante, muy sencillo, ya que en la ROM existe una rutina específica que cumple esta misión. Se trata de la que empieza en la dirección de memoria $F535, y exige a la entrada el número de registro en A y el contenido que queremos incluir en él en X.

Así, por ejemplo, si queremos que el volumen del canal 1 (registro 8) sea 5, haremos:

```
LDX  #5       ;valor a pasar
LDA  #8       ;número de registro
JSR  $F535    ;se efectua el traspaso
```

Cuando deseemos alterar todos o la mayoría de los registros del circuito de sonido, resulta más eficiente utilizar la rutina $FA6C , en lugar de repetir catorce veces el proceso anterior.

La rutina $FA6C pasa a los registros del Ø al D los valores contenidos en 14 bytes de memoria consecutivos. Para indicar a la rutina de dónde debe leer estos bytes, debemos introducir en X e Y, respectivamente, los bytes de peso bajo y alto del puntero al inicio de la tabla. Ejemplo:

```
FA85  LDX #$8D        FA9B  LDX#$A2
      LDY #$FA              LDY#$FA
      JSR $FA6C             JSR$FA6C
      RTS             FAA2  RTS
FA8C
```

Este sistema es el que utilizan los dos clics del teclado, y los comandos PING, SHOOT y EXPLODE, según los siguientes datos:

comando	rutina	tabla	reg.: Ø	1	2	3	4	5	6	7	8	9	A	B	C	D
PING	FA85	FA8D	18	00	00	00	00	00	00	3E	10	00	00	00	0F	00
SHOOT	FA9B	FAA3	00	00	00	00	00	00	0F	07	10	10	10	00	08	00
EXPLODE	FAB1	FAB9	00	00	00	00	00	00	1F	07	10	10	10	00	18	00
clic	FAFA	FB02	1F	00	00	00	00	00	00	3E	10	00	00	1F	00	00
clac	FB10	FB18	2F	00	00	00	00	00	00	3E	10	00	00	1F	00	00

En contraposición a los anteriores, el comando ZAP es algo más complejo, ya que requiere un desplazamiento continuo de la frecuencia. Para ello, empieza utilizando la rutina $FA6C para inicializar todos los registros, y luego emplea la rutina $F535 dentro de un bucle para producir por el canal 1 todos los períodos desde 1 hasta 111, sucesivamente, con una breve pausa en cada uno de ellos.

Los comandos SOUND, MUSIC y PLAY proceden a leer los parámetros, comprobar su validez, y pasarlos a los registros adecuados, previa modificación, si procede. Así, el código de volumen Ø se transforma en 16, y el código de envolvente 3 se transforma en 8, etc.

Igualmente, el código de nota (en el comando MUSIC) debe ser pasado a período para su introducción en el registro correspondiente. Para ello nos valemos de una tabla que empieza en $FC44 (bytes de peso alto) y $FC51 (peso bajo). Los valores así obtenidos son válidos para la octava Ø. Para las otras octavas se divide por 2 tantas veces como indica el número de octava.

Nota:	DO	DO#	RE	RE#	MI	FA	FA#	SOL	SOL#	LA	LA#	SI
Código ORIC	1	2	3	4	5	6	7	8	9	10	11	12
Período hexa	0777	070B	06A6	0647	05EC	0597	0547	04FB	04B3	0470	0430	03F4
íd. decimal	1911	1803	1702	1607	1516	1431	1351	1275	1203	1136	1072	1012

Sustituir los comandos SOUND y PLAY por sus equivalentes en código máquina resulta simple valiéndose de lo descrito hasta ahora. Sin embargo, para obtener el MUSIC, resulta más adecuado utilizar la siguiente secuencia:

```
LDA  # número de canal
STA  $2E1
LDA  # volumen (o Ø para envolvente)
STA  $2E7
LDX  # código nota (formato BASIC)
LDY  # número de octava
JSR  $FC1F
```

```
!!! BASIC
POKE #2E1, #Can.
POKE #2E7, Volm.
```

No hay que olvidarse de autorizar el canal seleccionado, mediante el adecuado valor en el registro 7.

Empleando la secuencia anterior, resulta fácil pensar en una rutina que reproduzca fragmentos musicales contenidos en una tabla, a razón de un byte por nota(por ejemplo, 4 bits para la longitud de la nota, y otros cuatro para el tono). De este modo, la ocupación de memoria es mínima.

El acceso directo a los registros del circuito generador de sonido permite, por ejemplo:

- modificar sucesivamente el volumen de cada canal, para lograr tipos de envolventes distintos de los preprogramados, independientes de uno a otro canal, o superpuestos unos a otros.

- modificar de modo continuo la frecuencia, produciendo efectos de tipo sirena, o al estilo del comando ZAP.

Para los interesados en estudiar las rutinas SOUND, MUSIC y PLAY, todas empiezan en $E889. Allí la rutina lee los parámetros, pasando sus valores a la dirección $2E1 y sucesivas (dos bytes para cada parámetro), y busca en la tabla $E84E la dirección real de la rutina específica, que en realidad contiene un salto a otra dirección, según el siguiente esquema:

comando	dirección	salta a
SOUND	$F41E	$FB26
MUSIC	$F424	$FBFE
PLAY	$F421	$FBB6

Aquí se produce el pase de los parámetros a los oportunos registros, valiéndose de las subrutinas antes explicadas.

Si durante el proceso se detecta algún tipo de error en los parámetros, el byte $2E0 es puesto a 1, para señalarlo al retorno.

Gran parte del tiempo de ejecución de estos comandos se destina al cálculo de los parámetros (conversión de caracteres ASCII a valores binarios) y verificación de los mismos, por lo que el código máquina permite una gran velocidad.

Nota: Anteriormente, al dibujar los gráficos de las envolventes, lo he hecho según se indica en el manual del fabricante del ORIC-1. No obstante, por lo menos en la versión que poseo, se aprecian dos variantes:

- la primera se halla en la envolvente número 4, que en lugar de empezar en el valle (punto bajo) de la envolvente, lo hace en la cresta. La evolución posterior es la indicada.

- la segunda, más importante, en la envolvente número 5, que empieza también en la cresta, para ir disminuyendo hasta 0 al llegar al semiperíodo, momento en que de golpe asciende al máximo, donde se estabiliza.

Otra nota: Podemos anular el tono producido por un canal, sin alterar el registro 7, simplemente introduciendo un 0 en el registro correspondiente al volumen del canal en cuestión. Recordemos que, para el control por envolvente, el valor a introducir era $10 (en decimal: 16).

D I R E C C I O N E S

Para lograr la máxima eficiencia en nuestros programas en código máquina, se requiere utilizar al máximo las subrutinas ya contenidas en la ROM, evitando duplicidades innecesarias, así como la manipulación de los punteros y parámetros contenidos en las primeras páginas de memoria, y utilizados por las rutinas de la ROM.

Ello precisa, de una parte, un conocimiento general de la estructura del sistema, según se ha descrito en todo lo anterior, y por otra, el conocimiento específico de la ubicación de las distintas rutinas y parámetros. Este es el objeto del presente capítulo.

Evidentemente, una descripción exhaustiva de todas las rutinas, con sus puntos de entrada, requerimientos y resultados escapa a las posibilidades del presente cuaderno. Por ello me limitaré a detallar las características más importantes de las rutinas más útiles, así como algunos de los parámetros más usados, con sus aplicaciones usuales.

La relación resultante debe, pues, ser considerada más bien como una guía para una más rápida búsqueda de la dirección necesaria. Una vez hallada, la rutina debe ser estudiada, mediante su desensamblado, para ver si se ajusta a lo que necesitamos, y como debemos utilizarla.

Las direcciones están ordenadas, por lo que empezamos por los parámetros de las primeras páginas de la memoria, para seguir con la ROM, a partir de $C000:

$C-$D: puntero usualmente empleado para la lectura de mensajes.

$12-$13: puntero a la dirección de principio de la linea de pantalla en que se encuentra actualmente el cursor.

$1A-$1C: contiene un JMP $CBED, y es usada por la secuencia principal del intérprete para la impresión del mensaje "Ready".

$1D-$1E: número de órden de la línea de Basic buscada por un GOTO. Aparentemente, su única utilidad sería servir para un eventual RENUMBER.

$1F-$20: puntero a la dirección actual del cursor en la pantalla.

$21-$23: contiene un JMP a la dirección donde hayamos definido la instrucción USR.

$24-$25: bytes utilizados para buscar el fin de una línea o de una tira alfabética dentro de una línea.

$26: utilizado como contador de la tabla de nombres de instrucciones, para calcular el token correspondiente a las mismas, durante el proceso de tokenización.

$27: usado para distinguir si estamos dimensionando una variable, o calculando su valor.

$28: contiene el valor $FF si la variable buscada es alfabética, y se pone a cero en caso contrario.

$29: contiene el valor $80 si la variable buscada es entera (dos bytes), y cero en caso contrario.

$2C: sus dos primeros bits sirven para fijar si estamos dentro de un READ, un GET o un INPUT.

$2E: el bit de peso alto determina el Control O (anula momentáneamente la impresión en la pantalla).

$2F: utilizado en los procesos de lectura y escritura de un byte en un fichero de cassette, para almacenar el byte a pasar al cassette.

$3Ø: lleva la cuenta del número de caracteres impresos en la línea actual por la impresora (erróneamente inicializado a 13, al principio de cada línea).

$31: número de caracteres por línea de la impresora.

$33-$34: acumulador íntegro (dos bytes). También destinado a contener el número de línea al introducir por teclado una.

$35-$84: buffer de entrada del teclado. Destinado a contener los caracteres que se van entrando. En él se lleva a cabo la tokenización. Las instrucciones CLOAD y CSAVE utilizan, como vemos a continuación, esta zona. Suele constituir la zona de página cero utilizable por las rutinas del usuario en código máquina, con un mínimo de precauciones.

$5F-$6Ø: dirección inicial del fichero en cassette.

$61-$62: dirección final del fichero en cassette.

$63: marca la característica de auto-load (ejecución automática al finalizar la lectura). { IF PEEK $63 = Ø THEN NO AUTO / IF PEEK $63 <> Ø THEN AUTO-LOAD

$64: señala que el fichero es en código máquina. { SI PEEK $64 = Ø EL PROGR. ES BASIC / SI PEEK $64 <> Ø EL PROGR. ES M/C

$67: señala que la grabación (o la lectura), debe efectuarse a velocidad lenta, de 3ØØ baudios. Ø = 2400 Bd, 3 = 300 Bd

$85-$86: puntero que apunta a $91, $94, etc., según cual sea el último puntero utilizado para salvar una variable.

$91-$93: punteros de una variable alfabética (longitud y dirección inicial) actualmente considerada. Según las necesidades, se emplean también las direcciones siguientes.

$9A-$9B: puntero hacia la dirección del primer byte del programa Basic residente en la memoria. (INICIALMENTE 1281)

$9C-$9D: puntero al primer byte ocupado por las variables no dimensionadas del programa de Basic. (INDICA TAMBIEN EL FINAL DEL BASIC)

$9E-$9F: puntero al primer byte ocupado por las variables dimensionadas.

$AØ-$A1: puntero al primer byte de la zona libre entre las variables dimensionadas y la zona de strings.

$A2-$A3: puntero al primer byte ocupado por los strings.

$A6-$A7: puntero al primer byte después de los ocupados por los strings, es decir, puntero al HIMEN.

$A8-$A9: durante la ejecución de un programa Basic, contiene el valor del número de línea en curso. Si se trata de ejecución directa por teclado, el byte contenido en $A9 vale $FF.

$AC-$AD: utilizado para salvar el puntero del intérprete, y también para indicar si se han producido modificaciones en el programa Basic antes de un intento de continuación.

$BØ-$B1: puntero de los DATA.

$B2-$B3: usado provisionalmente para la lectura de un DATA.

$B4-$B5: contienen los dos primeros caracteres de la variable buscada. Los bits de peso alto nos indican el tipo de variable.

$B6-$B7: puntero a la dirección de comienzo de la variable.

$CE-$CF: puntero a la dirección inicial de una línea de programa.

$DØ-$D5: acumulador de punto flotante.

$EØ-$E1: también utilizado en ocasiones para salvar el puntero del intérprete.

$E2: inicio de la rutina que lee el carácter siguiente al apuntado por el puntero del intérprete.

$E8: ídem. pero lee el carácter actual.

$E9-$EA: puntero del intérprete de Basic.

$1ØØ-$1FF: pila (stack). De $1FF para abajo contiene las direcciones de retorno y los parámetros que se pasan entre rutinas. De $1ØØ para arriba es utilizado para contener valores a imprimir.

$2Ø8: mientras mantenemos pulsada una tecla, este byte contiene un valor que depende de la tecla pulsada (no es el código ASCII).

$2ØC: el bit de peso alto determina el modo minúsculas.

$213: contiene la máscara del PATTERN.

$22Ø: contiene un 1, si la memoria es de 16K, un Ø si es de 48K.

$228-$22A: contiene el vector a la dirección de la rutina de interrupción por el VIA, cada centésima de segundo (lectura del teclado).

$22B-$22D: contiene el vector a la dirección de interrupción en caso de un Reset parcial (pulsación del botón situado en la parte inferior de la máquina).

$261-$262: puntero para salter a la ejecución del correspondiente carácter de control.

$263-$264: usado como valor intermedio para multiplicar por 4Ø el número de línea, durante el cálculo de la dirección de inicio de la misma en la pantalla.

$268: posición vertical del cursor, a partir de Ø.

$269: posición horizontal del cursor, a partir de Ø.

$26A: byte cuyos ocho bits determinan el estado del teclado (ver el apartado correspondiente que detalla la función de cada bit).

$26B: atributo PAPER.

$26C: atributo INK.

$26D-$26E: puntero a la dirección de principio de la pantalla.

$27E $26F: número de líneas de la pantalla.

$276-$277: contador de interrupciones.

$2CØ: controla el estado de la pantalla (HIRES, GRAB, etc.).

$2C1-$2C2: puntero a la dirección de donde no se puede pasar por el comando GRAB.

$2DF: código ASCII de la última tecla pulsada, más $8Ø.

$2EØ: a partir de aquí se colocan los parámetros para las rutinas musicales y de alta resolución.

$2F1: el bit de peso alto indica impresión en la impresora. De lo contrario, la impresión es por la pantalla.

$2F4: el bit de peso alto determina el modo TRON.

$2F5-$2F6: vector a la dirección de ejecución de la instrucción "!".

$2F7: usado por la máscara de impresión de los atributos.

$2F8: valor retornado por una expresión.

$2FC-$2FD: vector a la dirección de ejecución de la instrucción "&( )".

$3ØØ: registro de salida del port B del VIA.

$3Ø1: registro de salida del port A del VIA, con reconocimiento de recepción.

$3Ø2: registro de sentido de los datos del port B del VIA.

$3Ø3: registro de sentido de los datos del port A del VIA.

$3Ø4-$3Ø5: temporizador 1 del VIA.

$3Ø6-$3Ø7: registro latch del temporizador 1 del VIA.

$3Ø8-$3Ø9: temporizador 2 del VIA.

$3ØA: registro de desplazamiento en serie del VIA.

$3ØB: registro de control auxiliar del VIA.

$3ØC: registro de control periférico del VIA.

$3ØD: registro de estado de interrupciones del VIA.

$3ØE: registro de autorización de interrupciones del VIA.

$3ØF: registro de salida del port A, sin acuse de recepción, del VIA.

$CØØ6: inicio de la tabla de direcciones de las rutinas en código máquina correspondientes a las distintas instrucciones del Basic.

$CØCC: inicio de la tabla de direcciones de ejecución y grados de prioridad de los operadores +, -, x, /, ↑, AND y OR.

$CØEA: inicio de la tabla de los nombres de las distintas palabras reservadas del Basic. Para señalar el final de cada palabra, el último byte tiene el bit de peso alto puesto a 1. (VALOR 128)

$C2AC: inicio de la tabla de mensajes de error.

$C3F8: rutina que abre espacio dentro de un programa de Basic, para poder insertar en él una nueva línea.

$C43B: rutina que comprueba si hay espacio bastante bajo el stack para poder llevar a cabo el cálculo de una expresión, y almacenar allí los datos necesarios para los valores intermedios.

$C485: rutina general de error. Interrumpe la ejecución en curso, anunciando un mensaje de error, que lee de la tabla $C2AC. El registro X a la entrada debe contener la distancia desde $C2AC hasta el primer carácter del mensaje. Cada mensaje termina con un carácter que tiene el bit de peso alto a uno.

$C4B5: secuencia principal del Basic. Este es el punto por el que se entra una vez acabada la fase de inicialización general.

$C4E3: rutina que intercala una línea de Basic entrada por teclado, dentro del programa en curso.

$C56F: rutina que recalcula y modifica los links de un programa de Basic, después de haber sido éste modificado. También puede ser útil cuando por un error de lectura aparecen al listar el programa una serie de letras U (hacer CALL #C56F). ó también: Ø ? Ø RETURN

$C5A2: entrada de una línea de Basic por teclado.

$C5F8: rutina que lee la tecla pulsada y, si se trataba de Control O, posiciona los correspondientes bits del registro $2E para señalarlo.

$C6ØA: rutina que tokeniza la línea de Basic presente en el buffer de teclado, dejando en el registro Y la longitud final de la misma.

$C6A5: rutina correspondiente a "EDIT".

$C6DE: rutina que busca, a partir del principio del programa de Basic, una línea cuyo número corresponda con el que hayamos puesto en los bytes $33-$34. Al retorno, el puntero $CE-$CF contiene la dirección del primer byte de la línea buscada, y $1D-$1E el número de orden de la línea dentro del programa. El carry se pone a set si la hemos encontrado, y a clear en caso contrario.

$C6E8: rutina similar a la anterior, pero en lugar de iniciar la búsqueda por el principio del programa, se empieza a partir de la dirección contenida en A (peso bajo), X (peso alto), que debe ser la inmediata anterior al primer byte de una línea. En este caso, el puntero $1D-$1E es incrementado en el número de líneas saltadas hasta encontrar la buscada.

$C719: rutina "NEW".

$C738: rutina "CLEAR".

$C765: lleva el puntero del intérprete al byte anterior al primero del programa de Basic.

$C773: rutina "LIST".

$C824: rutina "LLIST".

$C832: rutina "LPRINT".

$C841: rutina "FOR".

$C8AD: interpretador del Basic. Este es el punto por donde se entra al ejecutar el comando RUN.

$C8DD: interpretador del Basic. Este es el punto por donde se entra para ejecución de una línea entrada directamente por teclado.

$C8FE: rutina que ejecuta una instrucción de Basic que empieza en el byte apuntado por el puntero del intérprete, que acabamos de leer.

$C91F: rutina "RESTORE".

$C93Ø: rutina que lee el teclado y comprueba si hemos pulsado Control C para detener la ejecución de una línea o programa de Basic.

$C93F: rutina "STOP".

$C941: rutina "END".

$C96E: rutina "CONT".

$C98B: rutina "RUN".

$C996: rutina "GOSUB".

$C9B3: rutina "GOTO".

$C9EØ: rutina "POP" y "RETURN" (equivalentes).

$CAØA: rutina "DATA".

$CAØD: rutina que incrementa el puntero del intérprete en el número de bytes indicado por el registro Y.

$CA1C: rutina que busca el fin de un DATA, o de una línea de Basic, según la entrada sea en este punto o en $CA1F.

$CA3E: rutina "IF".

$CA61: rutina "REM".

$CA66: rutina que busca un ELSE o el final de la línea de Basic.

$CA78: rutina "OR".

$CA98: rutina que convierte un número expresado por sus caracteres ASCII a íntegro sobre dos bytes, llevándolo a $33-$34. Esta rutina es más rápida que la general, pero solo trabaja en la gama Ø-63999.

$CAD2: rutina "LET".

$CB61: rutina "PRINT".

$CB9F: rutina que salta a una nueva línea.

$CBED: rutina que imprime un mensaje que empieza en la dirección apuntada por el puntero formado por los registros A, Y y termina con un byte nulo.

$CCØA: rutina "CLS". En realidad, imprime el carácter $C.

$CCØD: rutina que imprime un espacio.

$CC1Ø: rutina que imprime un interrogante.

$CC12: rutina general de impresión del carácter cuyo código esté contenido en A a la entrada. Sirve también para ejecutar los caracteres de control, y tiene en cuenta las dobles alturas, columnas reservadas, etc.

$CC4D: subrutina de la anterior que también imprime el carácter contenido en A, pero menos general.

$CC6D: contiene el mensaje "?PRINTER ERROR".

$CC89: rutina "!".

$CC8C: rutina "TRON".

$CC8F: rutina "TROFF".

$CCBA: rutina "GET".

$CCC9: rutina "INPUT".

$CCFD: rutina "READ".

$CDØ3: rutina que lee (del teclado o de un programa) un valor y lo asigna a una variable. Es una subrutina de GET y de INPUT.

$CEØC: rutina "NEXT".

$CE77: rutina que calcula el valor de una expresión que empieza en el lugar apuntado por el intérprete de Basic, y pasa su resultado al acumulador de punto flotante.

$CE8B: subrutina de la anterior que calcula la expresión, pero sin proceder a los tests de error al final.

$CF74: rutina que calcula el valor del siguiente número o variable apuntado por el puntero del intérprete (no el conjunto de la expresión).

$CF99: rutina que apunta y salta la tira alfabética contenida a partir de la posición actual del puntero del intérprete.

$CFD3: lee el carácter apuntado por el puntero del intérprete, comprueba que sea el deseado, si lo es, lo salta, y si no, va a un mensaje de error. Según cual sea el punto de entrada, el carácter buscado es:

$CFD3: ")"
$CFD6: "("
$CFD9: ","
$CFDB: el contenido en A

$CFE4: rutinas "INVERSE" y "NORMAL". Van directamente a error. Se supone que corresponden a rutinas que en un principio se planearon, pero que luego no se incluyeron en el proyecto definitivo.

$CFFØ: rutina que busca el valor de una variable.

$DØF2: rutina "DIM".

$DØFC: rutina que busca una variable o la crea, posicionando los registros $28, $29, $B4, $B5, $B6 y $B7.

$D186: rutina que comprueba que el carácter leído sea una letra A-Z y posiciona el carry a set en tal caso, y a clear en caso contrario.

$D21Ø: rutina que pasa el valor del acumulador de punto flotante a íntegro, retornando su valor en los bytes $D3-$D4.

$D3FD: rutina que pasa el valor del registro Y al acumulador de punto flotante.

$D4Ø1: rutina "DEF".

$D419: rutina que comprueba que estemos ejecutando un programa y, en caso contrario, salta a error.

$D4E8: rutina que busca espacio en la zona de strings.

$D4FA: rutina que, a partir de la dirección contenida en los bytes A, Y, busca la longitud de una tira alfabética, posicionando los punteros ($DE) y ($D1) al principio de la tira y ($EØ) al byte siguiente al final. El registro $DØ contendrá la longitud.

$D6F7: rutina que coloca una tira alfabética en la zona de strings.

$D715: rutina que retorna al acumulador A el número de caracteres de una tira alfabética.

$D8ØD: rutina que calcula el valor de una èxpresión como íntegro un byte, y lo lleva simultáneamente al registro X y a $D4.

$D852: posiciona el puntero del intérprete en el valor contenido en el puntero ($EØ) que suele ser el final de una tira alfabética.

$D85B: rutina que lee los parámetros para el POKE (responsable de que el segundo parámetro del POKE no pueda ser un valor hexadecimal, ya que conserva el primer parámetro en $33-$34, que son utilizados para la conversión de un valor hexadecimal.

$D867: rutina que pasa el acumulador de punto flotante a íntegro sobre dos bytes, llevándolo a $33-$34.

$D894: rutina "POKE".

$D89D: rutina "WAIT".

$D8AC: rutina "DOKE".

$D8D5: rutina que pasa el valor del íntegro formado por los registros A e Y al acumulador de punto flotante.

$D937: rutina "LORES".

$D965: rutina que calcula la dirección del primer byte de una línea de pantalla a partir del número de línea, colocado previamente en el acumulador, y devolviendo la dirección en $1F-$2Ø.

$D996: rutina que lee los parámetros X e Y para una posición en pantalla.

$D9C6: rutina "PLOT".

$D9FA: rutina "REPEAT".

$DA16: rutinas "PULL" y "UNTIL" (equivalentes).

$DE73: rutina que lee los cinco octetos a partir del apuntado por A, Y y los pasa al acumulador de punto flotante (lee una variable).

$DF22: rutina que pasa el íntegro contenido en $D1,$D2 al acumulador de punto flotante.

$DFCF: rutina que calcula el valor de un número expresado en ASCII.

$EØC1: rutina que imprime un íntegro contenido en los registros A y X.

$EØD1: rutina que pasa el valor del acumulador de punto flotante a su expresión en caracteres ASCII a partir de $1Ø1, terminado en un nulo.

$E4A8: rutina que lee un fichero desde el cassette.

$E5Ø3: mensaje "Searching". (RUTINA Searching = POKE#35,Ø:CALL#E6CA:CALL#E4A8:CALL#E804)

$E512: mensaje "Loading".

$E521: mensáje "File error. Load aborted".

$E53E: rutina de error durante la lectura de un fichero en cassette.

$E554: incrementa el puntero empleado para la lectura de un fichero en cassette ý posiciona el carry si hemos acabado.

$E563: rutina que borra el mensaje que pudiera haber en la línea reservada superior de la pantalla.

$E56E: rutina que escribe el nombre del programa en la línea reservada superior de la pantalla.

$E576: rutina que escribe un mensaje en la línea reservada superior.

$E57B: rutina que salva un fichero en cassette.

$E5BC: mensaje "Saving".

$E5C6: rutina que salva un byte en el fichero de cassette.

$E5F3: rutina que salva un bit en el fichero de cassette.

$E63Ø: rutina que lee un byte desde un fichero en cassette. El byte leído se va construyendo en el registro $2F.

$E65E: rutina que lee un bit desde un fichero en cassette.

$E67D: subrutina para la lectura de un bit. Cuenta el tiempo desde la anterior rampa hasta la siguiente, y lo compara con 512 microsegundos, posicionando el carry en consecuencia (recordemos que el tiempo sería de 416 microsegundos para un bit 1, y de 624 para un Ø).

$E696: rutina que lee el tono previo de un fichero de cassette, utilizándolo para sincronizar la sucesión de bits.

$E6BA: rutina que escribe al principio de un fichero en cassette el tono previo de sincronización, formado de 259 bytes $16.

$E6CA: inhibe las interrupciones por teclado, preparando el circuito de interfase VIA para la lectura del cassette.

$E6FØ: rutina que comprueba la coincidencia del nombre leído en un fichero de cassette con el que buscábamos.

$E725: subrutina de CLOAD y CSAVE que lee los parámetros necesarios.

$E79D: rutina que calcula el valor de la expresión siguiente, lo pasa a íntegro dos bytes y lo coloca en $33-$34.

$E7A5: códigos de !, E, A, S, AUTO utilizados por la rutina de lectura de parámetros para lectura y grabación de ficheros en cassette.

$E7AA: rutina "CLOAD".

$E7DB: rutina "CSAVE".

$E8Ø4: desinhibe las interrupciones, finalizando una operación con el cassette, y retornando a interrupciones por teclado.

$E8ØD: rutina "CALL".

$E813: calcula la expresión hexadecimal ASCII que empieza en el byte siguiente al apuntado por el puntero del intérprete (que será normalmente el carácter =), y lo pasa a $33-$34.

$E848: rutina que calcula una expresión hexadecimal.

$E84E: inicio de la tabla de direcciones de ejecución de, por este orden, SOUND, MUSIC, PLAY, CURSET, CURMOV, DRAW, CIRCLE, PATTERN, FILL, CHAR, PAPER e INK.

$E866: tabla del número de parámetros necesarios para las mismas funciones.

$E872: tabla que indica cuáles de las anteriores instrucciones admiten valores negativos como parámetros.

$E87D: entrada común de CURSET, CURMOV, DRAW, CIRCLE, PATTERN, FILL y CHAR.

$E889: entrada común de SOUND, MUSIC, PLAY, PAPER e INK.

$E9Ø5: rutina que retorna a A el código ASCII de la última tecla pulsada.

$E95B: rutina HIMEM.

$E974: rutina GRAB.

$E994: rutina RELEASE.

$E9A9: rutina TEXT.

$E9BB: rutina HIRES.

$EA24: fragmento destinado a ser copiado a partir de la dirección $E2, durante la fase de inicialización general.

$EA41: rutina que, en caso de encontrar un ELSE o un comentario, salta al fin de la línea de programa, y posiciona el Carry según se trate de un número.

$EA4E: posiciona el carry a clear si el carácter contenido en A es un número (códigos ASCII de $3Ø a $39), y pone el Zero a set si era ":" o un nulo (fin de instrucción o fin de línea).

$EA59: fragmento de la fase de inicialización general.

$EB43: mensaje "Bytes free".

$EB51: mensaje "Oric extended basic..."

$EBDØ: empieza una tabla de JMP.

$EC8Ø: rutina que resta el contenido de ($2ØØ,X) menos ($2ØØ,Y) y retorna el ORA de los dos bytes.

$EC95: suma ($2ØØ,X) más ($2ØØ,Y) dejando el resultado en ($2ØØ,X).

$ECAF: rutina que resta uno al puntero ($2ØØ,X).

$EDØ1: inhibe las interrupciones del VIA (desactiva el teclado).

$EDØ9: rutina de interrupción cada centésima de segundo.

$ED7Ø: rutina que vacía las direcciones $272 a $277, conservando A, X, Y.

$EDF6: rutina CURSET. Requiere la posición X en el registro X; la posición Y en el registro Y, y el color del cursor (fondo, frente...) en el registro $212.

$EEFF: rutina que transforma el parámetro de coordenada horizontal del modo alta resolución (colocado previamente en $C-$D) a número de byte dentro de la línea (que queda en $C-$D) y número de bit dentro del byte (en $E-$F). Resulta útil cuando queramos hacer una división de valores sobre dos bytes. Conserva A, X e Y. (Dividendo en $C-$D, divisor en $2ØØ-$2Ø1, el cociente queda en $C-$D y el resto en $E-$F).

$FØ2D: rutina CURSET.

$FØ79: rutina DRAW.

$FØ93: rutina PATTERN.

$FØA5: rutina CHAR.

$F25Ø: rutina que verifica el valor de un parámetro para las rutinas musicales o de alta resolución. Comprueba que el contenido de $2ØØ,X-$2Ø1,X sea mayor que cero pero menor que A, y posiciona el carry, según el resultado de la comprobación.

$F264: parecida a la anterior, pero además admite el valor cero.

$F4ØØ: principio de una tabla de JMP.

$F535: rutina que pasa el contenido del registro X al registro del circuito generador de sonido indicado por el contenido de A.

$F5B3: principio de la tabla que hay que sumar a la dirección $F5F2 para obtener la dirección de la subrutina correspondiente a un carácter de control.

$F5D3: incrementa el puntero del cursor de la pantalla.

$F5F2: rutina para el control H (retrocede un lugar).

$F5FC: rutina para el control K (sube un lugar).

$F6Ø2: rutina para el control I (adelanta un lugar).

$F6ØF: rutina para el control J (desciende un lugar).

$F615: rutina para el control M (vuelve al principio de la línea actual de la pantalla).

$F61B: rutina para el control N (vacía la línea actual en pantalla).

$F624: rutina para el control L (vacía la pantalla).

$F631: rutina para el control ^ (retorna el cursor al principio de la pantalla, sin limpiar esta).

$F671: mensaje "CAPS".

$F677: mensaje "   " (cuatro espacios, para borrar el CAPS).

$F6D3: rutina que vacía una línea de la pantalla.

$F6F1: transforma el número de línea a dirección inicial de la misma, retornando el valor en los registros A e Y.

$F7ØØ: multiplica el valor de A por 4Ø y coloca el resultado en $263-$264.

$F729: rutina que imprime (o no) el mensaje CAPS en la parte superior derecha, según el estado actual del teclado.

$F73F: rutina que imprime el carácter contenido en X (subrutina de $CC12).

$F7AC: ídem carácter contenido en A.

$F82F: rutina que coloca un mensaje en la línea superior de la pantalla. Los registros A e Y deben contener la dirección del primer carácter del mensaje (que debe terminar con un byte nulo), mientras que el registro X debe contener la distancia desde el primer byte de la pantalla hasta el lugar donde se debe empezar a imprimir el mensaje.

$F841: bloque destinado a ser copiado en las direcciones $228 y sucesivas durante la inicialización general.

$F84A: rutina de inicialización general.

$F874: mensaje "Memory error".

$F888: rutina de inicialización de los periféricos.

$F8D1: rutina de inicialización del teclado y los colores pantalla.

$F96Ø: rutina de inicialización del VIA durante el reset general.

$F97F: rutina que inicializa la pantalla.

$F9CØ: rutina que comprueba el tipo de memoria y su corrección.

$F9DD: subrutina para la comprobación de la memoria.

$FAØ6: otra subrutina para la comprobación de la memoria. Es la responsable de que en la zona libre de la memoria apare can los códigos $55.

$FA49: comprueba la memoria, dejando pasar un cierto intervalo entre la escritura y la lectura, para comprobar el correcto refresco de la RAM dinámica.

$FA61: rutina que produce un retardo de unos 8 milisegundos.

$FA63: rutina que provoca un retardo variable, en función del valor que contenga el registro X a la entrada (entre 1,5 y unos 3ØØ mseg.).

$FA85: produce el PING.

$FA9B: produce el SHOOT.

$FAB1: produce el EXPLODE.

$FAFA: produce un clic del teclado.

$FB1Ø: produce el otro clic.

$FBB6: rutina PLAY.

$FBFE: rutina MUSIC.

$FC45: tabla que nos da el período correspondiente a cada nota musical (bytes de peso alto).

$FC52: ídem. los bytes de peso bajo.

$FC7Ø: generador de caracteres (destinado a ser copiado en RAM).

$FFFA: vector para el reset parcial. Contiene la dirección $22B, en RAM, lo que permite desviarla.

$FFFC: vector para el reset general. Contiene la dirección $F42D.

$FFFE: vector para la interrupción para el VIA. Contiene la dirección $228, en RAM, por lo que también puede ser desviada.

- - - - - - - - - - -

Nota:

Las direcciones detalladas son válidas para las versiones a la venta en el mercado en el momento de preparar el presente cuaderno. No obstante, entra dentro de lo posible que, en el afán de realizar ciertas mejoras, eventuales nuevas versiones presenten modificaciones en algún punto.

Si el ordenador se ha adquirido con posterioridad a enero de 1984, deberá comprobarse este punto, realizando un checksum de la ROM, así:

```
S = Ø : FOR A = #CØØØ TO #FFFF STEP 2 : S = S + DEEK(A) : NEXT : ? S
```

da, en mi máquina, el valor: 245258461.

A P L I C A C I O N E S

En este capítulo vamos a ver algunos ejemplos de la utilidad de incluir en un programa de Basic subrutinas en código máquina. Procuraré asimismo que las aquí incluidas puedan ser de aplicación más o menos general, breves y seguras, con el fin de que el lector pueda adaptarlas a sus propios programas.

Antes, sin embargo, de pasar a los ejemplos concretos, voy a dar algunas indicaciones generales.

Acostumbro a colocar estas subrutinas en código máquina justo antes del principio del programa Basic, es decir, de modo que el último byte de la subrutina sea el de la dirección $4FF. Recordemos, en efecto, que el programa Basic empieza en la dirección $5Ø1, y que debe ir precedido de un byte nulo, que ocupa la dirección $5ØØ.

La ventaja de obrar de este modo es la posibilidad de salvar conjuntamente en un solo fichero de cassette tanto la subrutina como el programa. Para ello, si el nombre del programa es NOMBRE, y la dirección de inicio de la subrutina $4EØ (por ejemplo), bastará hacer: CSAVE "NOMBRE",A#4EØ

El problema que suele presentarse a quien no posee un programa ensamblador, o un utilitario parecido, es la introducción de la rutina. Para subsanar este inconveniente, a continuación daré un pequeño programa en Basic que servirá para cargar cualquiera de las rutinas de este capítulo y que, además, contiene un proceso de verificación para comprobar la correcta introducción de los datos. Ello evita sorpresas inesperadas (en código máquina cualquier error puede provocar el bloqueo de la máquina u otros comportamientos anómalos).

El programa es el siguiente:

```
1Ø  HIMEM #98ØØ : F = #4FF
2Ø  INPUT "Direccion inicial";I
3Ø  IF I<#4ØØ OR I>#4FF THEN 2Ø
4Ø  FOR D = I TO F : PRINT HEX$(D);
5Ø  INPUT B : POKE D,B : NEXT
6Ø  INPUT "Checksum";C
7Ø  S = Ø : FOR D = I TO F STEP 2
8Ø  S = S + DEEK(D) : NEXT
9Ø  IF C<>S THEN PRINT "Error datos" : PING : GOTO 2Ø
```

Para introducir una subrutina en código máquina, hacer RUN y responder a la pregunta "Dirección inicial?" introduciendo la del primer byte de la subrutina (no olvidar el signo # para indicar las direcciones hexadecimales).

Después el programa va preguntando uno por uno los contenidos de todos los bytes de la subrutina (si los introducís en decimal, no hace falta el signo #) hasta el último, después del cual, os pedirá la suma de comprobación, o "checksum", que facilito con cada subrutina, a fin de detectar cualquier error de introducción, que obligaría a volver a repetir el proceso.

Finalizada con éxito la introducción, podéis eliminar el programa Basic con un NEW (es aconsejable que lo salvéis antes en cassette, para futuras ocasiones), y pasáis a escribir el verdadero programa de Basic. Hecho esto, podéis salvar el conjunto según comenté antes y ya, siempre que queráis utilizarlo, os bastará un CLOAD "NOMBRE" normal, para recuperar conjuntamente Basic y código máquina.

Entrada de expresiones en un INPUT.

En ocasiones, interesaría poder entrar una expresión algebráica, o una línea completa de Basic, en respuesta a un INPUT ejecutado dentro de un programa. Por ejemplo, en los programas de dibujo de gráficas, es necesario detener el programa, crear una línea de Basic con la función a representar (con el adecuado número de línea), y reemprender el programa con el RUN nº de línea necesario.

Todo ello resulta poco práctico, y sujeto a errores, complicando el proceso de utilización del programa. Vamos a ver como una sencilla subrutina en código máquina nos resuelve el problema.

Voy a empezar dando la "receta" para introducir el programa, y luego comentaré su funcionamiento. En primer lugar, mediante el programa auxiliar descrito en la introducción del capítulo, vamos a crear la subrutina. Para ello, después del RUN, entramos como dirección inicial la #4D8, y a continuación los siguientes valores:

```
165,234,72,165,233,72,32,162,197,134,233,132,234,32,226,Ø,24Ø,244,32,1Ø,
198,162,Ø,181,53,24Ø,6,157,12,5,232,2Ø8,246,1Ø4,133,233,1Ø4,133,234,96.
```

Finalmente, como checksum, el valor 6951Ø6, y si aparece el Ready, es que todo ha ido bien, sino, más atención la próxima vez.

Ya podemos hacer NEW, e introducir el siguiente programa:

```
ØGOTO5
1AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
2RETURN
5 FOR D = #5ØC TO #558 : POKE D,32 : NEXT : ? "Entra la funcion Y=f(X)" :
  CALL #4D8
1Ø HIRES : CURSET 2Ø,1Ø,3 : DRAW Ø,18Ø,1 : CURSET 2Ø,1ØØ,3 : DRAW 2ØØ,Ø,1
15 M = 2Ø : N = 1ØØ : CURSET M,N,3 : FOR P = 2Ø TO 22Ø
2Ø X = P - 2Ø : GOSUB 1 : Q = INT(1ØØ - Y)
25 IF Q < Ø THEN Q = Ø
3Ø IF Q > 199 THEN Q = 199
35 IF M = P AND N = Q THEN NEXT
4Ø DRAW P-M,Q-N,1 : M = P : N = Q : NEXT
45 INPUT "Otra funcion";R$
5Ø IF R$ > "O" THEN 5
```

Observaciones:

- las líneas Ø y 1 deben ser introducidas tal cual, sin ningún espacio intermedio.
- la línea 1 debe constar de, exactamente, 77 caracteres "A" (dos líneas completas de 38 caracteres, y el último, que estará justo debajo del primero).
- en la línea 5Ø, el carácter entre comillas es la letra O, no el número Ø.

Utilización:

- Hacer RUN. La máquina te pide la función a representar. Introduce, por ejemplo: Y = 8Ø * SIN(X/3) * SIN(X/3Ø). Puedes utilizar la tecla DELETE en caso necesario. Termina la introducción con RETURN.
- El programa pasa a HIRES, dibuja los ejes, y luego la curva, desde X = Ø hasta X = 2ØØ, y para unos valores de Y entre -99 y +1ØØ. La curva es dibujada mediante trazo continuo, no por puntos, y el programa está protegido contra valores de Y que se salgan de la pantalla.

- Al acabar la gráfica, el programa pregunta si deseas representar otra función. En caso afirmativo, responde S, o SI (observa que también comprende YES, o OUI). En caso negativo N, o NO (o NON). Si has respondido que sí, el proceso se repite.

Para los interesados en ello, la rutina en código máquina es la siguiente:

```
$4D8  LDA  $EA          }
      PHA               }
      LDA  $E9          } salvamos el puntero del intérprete en el stack
      PHA               }
entr  JSR  $C5A2        entrada línea en el buffer de teclado
      STX  $E9          }
      STY  $EA          } si la línea era nula, esperamos otra
      JSR  $E2          }
      BEQ  entr         }
      JSR  $C60A        tokenizamos la línea presente en el buffer
      LDX  #0           inicializamos índice para la copia
lazo  LDA  $35,X        leemos carácter del buffer
      BEQ  fin          si era el nulo final, saltamos
      STA  $50C,X       lo colocamos en la línea 1 del programa Basic
      INX               aumentamos índice
      BNE  lazo         pasamos al siguiente carácter
fin   PLA               }
      STA  $E9          } recuperamos puntero antes de retornar al Basic
      PLA               }
      STA  $EA          }
      RTS
```

RESTORE a un número de línea.

En el ORIC-1, cuando ejecutamos un RESTORE, el puntero de los DATA vuelve al principio del programa de Basic, de modo que el siguiente READ leerá el primer DATA contenido en el programa.

Sin embargo, ciertas versiones de Basic disponen de un RESTORE n, que permite colocar el puntero de los DATA en el número de línea especificado en la sentencia RESTORE, de modo que el siguiente READ lea el primer DATA que se encuentre después de la línea n.

Si queremos disponer en algún programa de esta función, podemos crearla con la adecuada subrutina. Para ello, empezaremos, como siempre, empleando nuestro programa auxiliar de entrada de rutinas en código máquina. En esta ocasión, la dirección de comienzo es la #4E4, y los valores de los bytes:

32,157,231,165,154,166,155,32,232,198,176,3,76,241,201,165,206,233,1,
133,176,165,207,233,0,133,177,96.

El checksum vale ahora 544744. Si no ha habido errores, hacer NEW.

Para mostrar cómo se puede utilizar en un programa la nueva instrucción, escribir el siguiente programa:

```
10  DOKE #2F5,#4E4
20  REPEAT : INPUT "Numero de linea";L : ! L : READ A$
30  ? "He leido la " ; A$ : PULL
```

```
  4Ø  DATA linea 4Ø
  5Ø  DATA linea 5Ø
 1ØØ  DATA linea 1ØØ
 25Ø  DATA linea 25Ø
 365  DATA linea 365
```

Haced RUN. El programa os pregunta qué linea deseáis leer, y a continuación os indica la línea leída. Comprobaréis que el DATA leído será siempre el primero que haya empezando en la línea especificada.

Así, por ejemplo, si indicáis la línea 2Ø, el DATA leído será el de la línea 4Ø. Si la línea especificada no existe, aparece el mensaje UNDEF'D STATEMENT ERROR.

El programa no se detiene hasta que no lo interrumpáis con Control C.

La rutina es la siguiente:

```
$4E4  JSR  $E79D     leemos la expresión y calculamos su valor
      LDA  $9A
      LDX  $9B
      JSR  $C6E8     buscamos la dirección de la línea especificada
      BCS  salto     si la hemos encontrado, saltamos
      JMP  $C9F1     en caso contrario, vamos a rutina de error
salto LDA  $CE
      SBC  #1
      STA  $BØ       restamos uno a la dirección hallada, y colocamos
      LDA  $CF       este valor en el puntero de los DATA ($BØ-$B1)
      SBC  #Ø
      STA  $B1
      RTS            retorno al Basic
```

Observaréis que, para mayor comodidad al llamar la rutina, utilizamos la instrucción "!". Para ello, al principio del programa, definimos la misma en la dirección de inicio de la subrutina (línea 1Ø DO'E #2F5,#4E4).

Hecho esto, basta hacer ! n para restaurar el puntero a la línea n.

Impresión de caracteres gigantes.

Cuando se desee imprimir en la pantalla caracteres ocho veces mayores que lo habitual, por ejemplo para títulos, juegos, anuncios, etc. puede emplearse la siguiente rutina, que se utiliza con la misma simplicidad que la función PLOT.

Para ello, utilizaremos como siempre nuestro programa auxiliar y entraremos los siguientes valores, a partir de la dirección $4B8 (que en el ORIC se entra como #4B8):

32, 15Ø, 217, 32, 217, 2Ø7, 32, 13, 216, 16Ø, 7 14Ø, 227, 2, 138, 32, 248, 24Ø, 16Ø, Ø, 177, 12, 162, 6, 172, 248, 2, 1Ø, 1Ø, 1Ø, 72, 144, 3, 169, 16Ø, 44, 169, 32, 145, 31, 1Ø4, 2ØØ, 192, 4Ø, 176, 3, 2Ø2, 2Ø8, 236, 169, 4Ø, 24, 1Ø1, 31, 133, 31, 144, 2, 23Ø, 32, 23Ø, 12, 2Ø8, 2, 23Ø, 13, 2Ø6, 227, 2, 16, 2Ø3, 96.

Por último, el checksum debe valer 718931, si todo ha ido bien.

Para utilizar la rutina más cómodamente, conviene definir la instrucción "!" mediante DOKE #2F5,#4B8. Hecho esto, para imprimir, por ejemplo, el carácter A (de código 65) de modo que el márgen superior izquierdo esté en la columna 15 y la fila 8, haremos: !15,8,65

Para demostrar las posibilidades de la rutina, probad el siguiente programa de Basic que permite escribir directamente por teclado caracteres gigantes a medida que los vamos tecleando:

```
1Ø  DOKE #2F5,#4B8 : POKE #26A,2
2Ø  CLS : A$ = "      " : REPEAT
3Ø  GET K$ : A$ = RIGHT$(A$,5) + K$
4Ø  X = 1 : FOR C = 1 TO 6
5Ø  ! X,1Ø,ASC(MID$(A$,C,1))
6Ø  X = X + 6 : NEXT
7Ø  UNTIL K$ = CHR$(3)
```

Para interrumpir el programa basta hacer Control C. La tira alfabética en la línea 2Ø está formada por seis espacios. La rutina desensamblada es:

```
plotg  JSR  $D996       ;lee los parámetros columna-fila
       JSR  $CFD9       ;lee y salta la coma
       JSR  $D8ØD       ;calcula la expresión a valor íntegro
       LDY  #7
       STY  $2E3        ;doble uso: contador y base tabla
       TXA
       JSR  $FØF8       ;calcula puntero al primer byte carácter
arrib  LDY  #Ø
       LDA  ($C),Y      ;lee el byte del dibujo del carácter
       LDX  #6
       LDY  $2F8        ;posición horizontal
       ASL  A
       ASL  A
bucle  ASL  A           ;leemos bit
       PHA
       BCC  den
       LDA  #16Ø        ;bloque en video inverso
       BIT  [den LDA #32]
       STA  ($1F),Y     ;colocamos bloque
       PLA
       INY              ;incrementamos columna
       CPY  #4Ø         ;comprobamos si salimos de pantalla
       BCS  fin
       DEX
       BNE  bucle       ;repetimos seis veces
fin    LDA  #4Ø
       CLC
       ADC  $1F
       STA  $1F         ;incrementamos puntero al inicio de línea
       BCC  salto
       INC  $2Ø
salto  INC  $C          ;incrementamos puntero al dibujo
       BNE  luogo
       INC  $D
luego  DEC  $2E3        ;lo repetimos ocho veces
       BPL  arrib
       RTS              ;retornamos al Basic
```

Verificación de la lectura desde cassette.

A poco que queramos efectuar un trabajo de cierta importancia con el cassette nos veremos en la necesidad de incorporar algún tipo de rutina para la verificación de la lectura. Es por ello que, aunque en el capítulo dedicado al cassette ya he incluído una de tales rutinas, añadiré aquí otra.

Esta presenta con respecto a la anterior la ventaja de ser utilizable también para rutinas en código máquina y todo tipo de ficheros, a diferencia de aquella, solo utilizable para programas en Basic. Además, resulta más corta, lo que la hace por ello mismo más fiable.

Como contrapartida a estas ventajas está el hecho de un uso algo más complejo, que detallaré posteriormente.

Empezaremos por introducir la rutina con ayuda de nuestro programa auxiliar. A partir de la dirección inicial #4DA, los valores de los bytes son:

165, 95, 164, 96, 133, 51, 132, 52, 169, Ø, 17Ø, 168, 24, 97, 51, 144, 1, 2ØØ, 72, 32, 84, 229, 1Ø4, 144, 243, 17Ø, 152, 32, 193, 224, 76, 214, 231, 32, 75, 231, 24Ø, 218.

El checksum final deberá ser 6243Ø3. Si todo ha ido bien, ya tenemos la rutina dispuesta. En su forma actual, sirve para checksumar programas Basic, según el siguiente proceso:

Introducimos el programa de Basic que queramos salvar, y hacemos:

CALL #4FB, A#4DA, EDEEK(#9C)

obtendremos un valor entre Ø y 65535, que deberemos anotar en la tapa del cassette, pues constituye el checksum del programa. Ahora podemos salvar el programa mediante la sentencia:

CSAVE "nombre", A#4DA, AUTO

añadiendo ,S si lo queremos a velocidad lenta. Es todo.

Al leer el programa, con un CLOAD normal, debemos obtener el mismo checksum anotado en la tapa (o de lo contrario, repetir la lectura).

Para utilizarlo con rutinas en código máquina u otro tipo de ficheros, deberemos trasladar la rutina de verificación de modo que quede situada antes del principio del fichero a salvar. Así, el principio de la rutina de verificación deberá situarse 38 bytes antes del principio del fichero propiamente dicho. Para obtener el checksum, deberemos hacer:

CALL manu, Achsum, Efinfichero

siendo manu la dirección de inicio del fichero real menos 5 bytes, y chsum la dirección de inicio del fichero real menos 38 bytes. Hecho esto y anotado el checksum, para salvar el conjunto, haremos:

CSAVE "nombre", Achsum, Efinfichero, AUTO

El posterior CLOAD será totalmente normal.

La rutina en lenguaje de ensamblaje, comentada, es la siguiente:

```
chsum   LDA  $5F      }
        LDY  $6Ø      } colocamos la dirección inicial en el puntero
        STA  $33      }
        STY  $34      }
        LDA  #Ø       }
        TAX           } inicializamos a cero los registros
        TAY           }
suma    CLC           }
        ADC  ($33,X)  } sumamos el contenido del byte leído
        BCC  next     }
        INY           }
next    PHA           }
        JSR  $E554    } comprobamos si hemos llegado al final
        PLA           }
        BCC  suma       si no, volvemos a sumar el siguiente byte
        TAX           }
        TYA           } imprimimos en la pantalla el checksum
        JSR  $EØC1    }
        JMP  $E7D6      retornamos al Basic por teclado
manu    JSR  $E74B      leemos las direcciones de inicio y fin
        BEQ  chsum      saltamos al principio de la rutina
```

Aprovecharé la ocasión para comentar un hecho que podría ser motivo de dudas al checksumar programas de Basic. Cuando salvamos un programa de Basic en cassette, el último byte incluído en el fichero de cassette no es el último del programa de Basic, sino el siguiente a este (que es el primero destinado a las variables).

Por este motivo, dos programas de Basic exactamente iguales pueden dar valores distintos para el checksum, según el valor que tuviera el byte excedente.

Para comprobarlo, dado un cierto programa, hagamos:

```
POKE DEEK(#9C), Ø          y calculemos el checksum.
```

luego hagamos:

```
POKE DEEK(#9C), 1          y repitamos el cálculo del checksum.
```

Comprobaremos que el checksum habrá variado (aumentado en una unidad). Similarmente, podemos probar dando otros valores y para cada caso se obtendrá un checksum distinto. Por este motivo, conviene calcular el checksum inmediatamente antes de salvar el fichero, sin realizar ninguna otra operación intermedia que pudiera alterar el byte final.

También es recomendable, como garantía suplementaria, comprobar el checksum, inmediatamente después de salvado el fichero, para asegurarnos que no ha variado por cualquier causa.

Finalmente, como ejemplo de los valores a utilizar en el caso de rutinas en código máquina, para el "ORICMON", que empieza en #A8ØØ y termina en #B18Ø, las sentenciás serían:

-cálculo del checksum: CALL #A7FB, A#A7DA, E#B18Ø

-para salvarlo: CSAVE "ORICMON", A#A7DA, E#B18Ø, AUTO

Conversión a binario.

La siguiente rutina permite imprimir en la pantalla el equivalente binario de un número. En realidad presento dos versiones de la misma rutina: la primera para números hasta 255 (ocho bits), y la segunda hasta 65535 (dieciseis bits).

Para la primera versión, la dirección inicial es #4EF, y los bytes:

32, 13, 216, 16Ø, 8, 169, 24, 6, 212, 42, 32, 18, 2Ø4, 136, 2Ø8, 245, 96.

El checksum vale 2Ø3Ø16. Una vez entrada la rutina, definiremos el comando "!" mediante DOKE #2F5,#4EF.

Hecho esto, para convertir, por ejemplo, el número 73 a binario, basta hacer

! 73 y aparece: Ø1ØØ1ØØ1

Si el número es mayor que 255, aparecerá el mensaje Illegal Quantity Error. La parte fraccionaria, si existe, será ignorada. Para obtener una tabla de equivalentes binarios, podéis hacer:

```
FOR N = Ø TO 255 : PRINT N, : ! N : PRINT : NEXT
```

La rutina en lenguaje de ensamblaje, comentada, es la siguiente:

```
        JSR  $D8ØD     ;calcula la expresión que sigue al ! y la pasa a
                       ;valor íntegro sobre un byte colocándolo en $D4
        LDY  #8        ;utilizado para contar los ocho bits
bit     LDA  #$18      ;este valor, al añadirle un bit por la derecha
                       ;se convertirá en el código ASCII de Ø ó 1
        ASL  $D4       ;pasa el bit de peso alto al carry
        ROL  A         ;y del carry a la derecha del acumulador
        JSR  $CC12     ;imprime el valor resultante en el acumulador
        DEY            ;contamos los bits
        BNE  bit       ;si no hemos acabado, volvemos al siguiente
        RTS            ;retornamos al Basic
```

Segunda versión: dirección inicial #4ED. Valores de los bytes:

32, 157, 231, 16Ø, 16, 169, 24, 6, 51, 38, 52, 42, 32, 18, 2Ø4, 136, 2Ø8, 243, 96.

Checksum: 249Ø1Ø. Para definir el comando: DOKE #2F5,#4ED. La utilización es semejante al caso anterior: !#FØFØ 1111ØØØØ1111ØØØØ

```
        JSR  $E79D     ;calcula la expresión que sigue al ! y la pasa a
                       ;valor íntegro dos bytes en $33-$34
        LDY  #16       ;contador de los 16 bits
bit     LDA  #$18      ;como en la anterior rutina
        ASL  $33       ;el resto es similar al caso anterior, solo que el
        ROL  $34       ;bit de peso alto de $33 entra por la derecha del
        ROL  A         ;$34, y de la izquierda de éste, pasa al acumulador
        JSR  $CC12
        DEY
        BNE  bit
        RTS
```

Cómo tener dos programas Basic en memoria.

Sucede que en ocasiones queremos simultanear el uso de dos distintos programas de Basic. Por ejemplo cuando preparamos un programa, y para ello utilizamos otro de tipo utilitario. Si los programas son algo largos, el proceso de lectura desde el cassette se hace pronto monótono y tedioso.

Sin embargo, la memoria del ORIC-1 48 K es suficiente para poder contener simultáneamente dos programas de longitud normal. En efecto, restando las zonas ocupadas por el sistema para sus variables internas, caracteres y pantalla, aun nos quedan unos 4Ø K. La mayoría de los programas de uso corriente caben en 2Ø K, por lo que resulta posible partir en dos la zona útil.

Es lo que realiza la siguiente rutina, que permite guardar el programa existente, en una zona por encima del HIMEM, de donde se recupera cuando vuelve a hacer falta. Para ello es necesario que el HIMEM esté situado en #4DFF (19967) o menos (de lo contrario, al llamar la rutina, se produce un mensaje de error).

Pero empecemos por introducir la rutina. La dirección inicial es #4BB (1211) y los valores de los bytes:

165,167,2Ø1,78,144,3,76,131,196,169,Ø,133,176,133,178,169,5,133,177,169,
78,133,179,16Ø,1,162,72,177,176,72,177,178,145,176,1Ø4,145,178,2ØØ,2Ø8,
243,23Ø,177,23Ø,179,2Ø2,16,236,32,111,197,24,165,145,1Ø5,2,133,156,165,
146,1Ø5,Ø,133,157,32,51,199,76,255,231.

El checksum vale 1239577. Una vez la rutina en su sitio, introducimos el primero de los dos programas de Basic en la memoria (por teclado o por cassette, por los procedimientos normales). Hecho esto, hacemos HIMEM #4DFF (19967) y luego DOKE #4EØ1 (19969),Ø (esto sólo es necesario la primera vez).

Luego hacemos CALL #4BB (1211) con lo que el programa de Basic quedará guardado en una zona encima del HIMEM (si hacemos un LIST, no aparecerá nada). Podemos ahora introducir el segundo programa, normalmente. Este segundo programa será el que quedará activo, pudiendo ejecutarlo, listarlo, modificarlo, etc. normalmente.

En cualquier momento podemos intercambiar ambos programas simplemente haciendo otra vez CALL #4BB (la rutina tarda algo menos de un segundo en intercambiar la zona "activa" con la zona "de reserva"). Como efecto adicional, la rutina borra todas las variables, para evitar que las utilizadas por un programa puedan alterar los resultados del otro.

La zona "activa" ocupa de #5Ø1 a #4DFF y la "de reserva" de #4EØ1 a #96FF. La rutina es la siguiente:

```
        LDA  $A7        ;leemos el byte de peso alto del HIMEM
        CMP  #$4E       ;lo comparamos con $4E
        BCC  salto      ;si el HIMEM es inferior, saltamos
        JMP  $C483      ;de lo contrario, vamos a mensaje de error
salto   LDA  #Ø
        STA  $BØ
        STA  $B2
        LDA  #5
        STA  $B1        ;colocamos $5ØØ en el puntero ($BØ)
        LDA  #$4E
        STA  $B3        ;colocamos $4EØØ en el puntero ($B2)
        LDY  #1         ;Y es el índice dentro de la página
```

```
        LDX  #$48        ;X es el contador de páginas a copiar
lazo    LDA  ($BØ),Y     ;leemos un byte de la zona activa
        PHA              ;lo salvamos en el stack
        LDA  ($B2),Y     ;leemos un byte de la zona de reserva
        STA  ($BØ),Y     ;lo pasamos a la zona activa
        PLA              ;recuperamos el otro byte
        STA  ($B2),Y     ;y lo pasamos a la zona de reserva
        INY              ;incrementamos índice
        BNE  lazo        ;seguimos hasta el fin de la página
        INC  $B1         ;incrementamos puntero "activo"
        INC  $B3         ;incrementamos puntero "reserva"
        DEX              ;decrementamos contador de páginas
        BPL  lazo        ;seguimos hasta acabar las páginas
        JSR  $C56F       ;buscamos fin de programa
        CLC
        LDA  $91
        ADC  #2
        STA  $9C
        LDA  $92
        ADC  #Ø
        STA  $9D         ;rehacemos puntero fin de programa
        JSR  $C733       ;borramos zona de variables
        JMP  $E7FF       ;retornamos al intérprete
```

Combinación de varias rutinas.

Todas las rutinas se han preparado de modo que acaben en #4FF. Ello impide usar más de una al mismo tiempo, ya que se solaparían mutuamente.

Podemos evitarlo, trasladándolas a otra zona (la más usual es de #4ØØ en adelante). Para ello, supuesto que tenemos una de estas rutinas que empieza en la dirección "inicio" y queremos pasarla a una zona que empiece en la dirección "destino", podemos hacer:

```
I = inicio : F = destino : FOR D = I TO #4FF : POKE F,PEEK(D) :
F = F + 1 : NEXT D
```

Lógicamente, si antes la rutina se llamaba mediante la instrucción CALL inicio, ahora deberá serlo mediante CALL destino.

De este modo, podemos colocar todas las rutinas en la zona de #4ØØ a #4FF, pudiendo ser utilizadas simultáneamente. Otra zona conveniente es de #97ØØ a #97FF, supuesto que el HIMEM haya sido colocado por debajo de esta zona.

E R R O R E S

En el curso de las distintas rutinas, se procede a una serie de comprobaciones, por ejemplo, si los parámetros están dentro de la gama de valores válidos, o si el modo de funcionamiento en curso es el adecuado, etc.

De encontrarse alguna condición de error, la ejecución se desvía hacia una rutina que para el programa en curso, restaura una serie de registros en página cero, imprime un mensaje de error, y retorna a la espera de nuevas instrucciones por teclado.

Podemos hacer que nuestras propias rutinas en código máquina lleven a cabo las verificaciones necesarias para evitar bloqueos imprevistos, y se desvíen a las rutinas de error antes citadas, indicando el mensaje adecuado.

La forma general de llamar a la rutina de error consiste en cargar el registro X con el valor adecuado (ver tabla), y saltar a $C485. Sin embargo, para la mayoría de tipos de error, existen direcciones de memoria en la ROM a las que podemos saltar directamente sin necesidad de inicializar X.

LDX #$1Ø
P.2. 13

R U T I N A S D E E R R O R		
X	Mensaje	Dirección
$ØØ	NEXT WITHOUT FOR	-
$1Ø	SYNTAX	$CFE4
$16	RETURN WITHOUT GOSUB	$C9EE
$2A	OUT OF DATA	-
$35	ILLEGAL QUANTITY	$D2AØ
$45	OVERFLOW	$DBEØ
$4D	OUT OF MEMORY	$C483
$5A	UNDEF'D STATEMENT	$C9F1
$6B	BAD SUBSRIPT	$D29D
$78	REDIM'D ARRAY	-
$85	DIVISION BY ZERO	$DE5B
$95	ILLEGAL DIRECT	$D41E
$A3-$A8	DISP / TYPE MISMATCH	$DA72-$CE86
$B5	STRING TOO LONG	$D6C7
$C4	FORMULA TOO COMPLEX	$D53F
$D7	CAN'T CONTINUE	-
$E5	UNDEF'D FUNCTION	$D421
$F5	BAD UNTIL	$DA22

Ejemplo: para señalar un error "ILLEGAL QUANTITY" podemos hacer, mediante la rutina general: LDX #$35 JMP $C485 o bien, simplemente: JMP $D2AØ.

F4C

C O N T R O L

El ORIC-1 utiliza, como la mayoría de ordenadores, el código ASCII para representar los diferentes caracteres. En este código, cada letra mayúscula o minúscula, los números, signos de puntuación, el espacio y caracteres especiales tienen una representación específica mediante un código de siete bits.

Como que un byte tiene ocho bits, el octavo, de peso alto, queda libre para usos especiales. Así, por ejemplo, al representar un mensaje en código máquina, este bit suele utilizarse para marcar el fin del mensaje. También, en la impresión en la pantalla, sirve para representar que aquel carácter debe ser representado en video invertido (en los colores complementarios).

Los siete bits permiten representar cantidades del Ø al 127. Sin embargo, sólo los códigos del 32 al 126 se utilizan para representar caracteres. El resto, del Ø al 31 y el 127, al ser "imprimidos" mediante el PRINT o su equivalente en máquina, la rutina $CC12, en lugar de imprimir un carácter, ejecutan ciertas funciones de control.

Así por ejemplo, si hacemos PRINT CHR$(13), o bien, en máquina: LDA #13, JSR $CC12, se producirá un retorno del cursor al principio de la línea actual en la pantalla.

Todos los caracteres de control pueden lograrse por el método antes descrito, pero hay algunos que, por su uso más frecuente, disponen de teclas especiales en el propio teclado del ORIC-1. Así, las teclas de las cuatro flechas corresponden a los caracteres de códigos 8 a 11.

Para el resto, disponemos de un sistema más cómodo que el PRINT CHR$(n). Consiste en la pulsación de la tecla de Control y, simultáneamente, otra tecla cualquiera. En efecto, el resultado de pulsar la tecla de Control es que los dos bits de peso alto del código de la tecla pulsada son puestos a Ø. Así, Control D equivale a PRINT CHR$(4), ya que la letra D tiene el código ASCII 68, es decir ØlØØØlØØ en binario. Al poner a cero los dos primeros bits, se transforma en ØØØØØlØØ, o sea, el carácter de código 4.

Los caracteres de control de utilidad en la máquina de base son:

Control	CHR$(n)	Efecto obtenido:
A	1	Copia en el buffer de teclado el carácter contenido en la posición actual del cursor, si no es atributo
C	3	Detiene la ejecución de un programa
D	4	Pasa a la impresión en doble altura, o la cancela, si ya estábamos en este modo de impresión
F	6	Anula los clics del teclado, o los retorna si ya estaban inhibidos
G	7	Produce el PING
H	8	Desplaza el cursor a la izquierda (←)
I	9	Desplaza el cursor a la derecha (→)
J	1Ø	Desplaza el cursor hacia abajo (↓)

K	11	Desplaza el cursor hacia arriba (↑)
L	12	Retorna el cursor al principio de la pantalla, borrándola
M	13	Retorna al principio de la línea actual de la pantalla, sin borrarla (RETURN)
N	14	Borra la línea actual de la pantalla
O	15	Anula momentáneamente la impresión de los caracteres en la pantalla. La impresión suele retornar al terminar la instrucción en curso.
P	16	Determina impresión por la impresora
Q	17	Anula la aparición del cursor en la pantalla, o lo retorna, si no aparecía
S	19	Anula la impresión en la pantalla de modo definitivo, hasta nueva pulsación de Control S
T	2Ø	Intercambia el modo mayúsculas-minúsculas
X	24	Termina la introducción de la línea actual en el buffer, anulándola
[	27	ESCAPE. El siguiente carácter será interpretado como un atributo.
]	29	Autoriza impresión en la columna protegida, hasta que vuelva a pulsarse otra vez
	3Ø	Devuelve el cursor al principio de la pantalla, sin borrarla
	127	DELETE. Anula último carácter introducido

Algunos de estos caracteres son bastante útiles en los programas. Un ejemplo es cuando queremos que en un programa aparezca un texto en la pantalla, sin que aparezca el cursor parpadeante.

En este caso, podemos incluir al principio del programa la instrucción PRINT CHR$(17). Ahora bien, esto plantea un problema. Si, por cualquier causa, cuando iniciamos la ejecución del programa mediante RUN, habíamos inhibido ya previamente el cursor mediante Control Q, al ejecutarse el programa, la instrucción PRINT CHR$(17) nos producirá el efecto contrario al deseado.

En efecto, los Controles Q, S, P, F, [,] , D y T actúan a modo de balanza, simplemente invirtiendo el modo anterior, por lo que su efecto depende del estado en que se encontraran cuando se ejecuta el mandato Control.

Hay una manera de solucionar este problema. Estos ocho controles determinan el estado de los ocho bits de la posición de memoria %26A, que es la que determina en todo momento el modo de funcionamiento.

Podemos, pues, actuar directamente sobre este registro, para controlar simultáneamente los ocho factores, y ello independientemente del estado anterior. Para ello es necesario conocer la estructura concreta de la citada posición de memoria:

bit	valor	efecto si está SET/CLEAR	Control	CHR$( )
Ø	1	cursor aparece/no aparece	Q	17
1	2	caracteres se imprimen/no se imprimen	S	19
2	4	impresora desconectada/conectada	P	16
3	8	clics inhibidos/audibles	F	6
4	16	siguiente carácter atributo/normal	[	27
5	32	columna no reservada/reservada	]	29
6	64	doble altura/altura normal	D	4
7	128			

Cuando conectamos la máquina, el registro $26A contiene el valor 3, es decir, ØØØØØØ11 en binario, con lo que el cursor aparece, los caracteres se imprimen, los clics son audibles, etc.

Si queremos que durante la ejecución de un programa no aparezca el cursor, y que los clics queden inhibidos, podremos colocar en $26A el valor ØØØØ1Ø1Ø, es decir, 10, lo que lograremos implemente incluyendo al principio la instrucción POKE #26A,10.

De este modo logramos lo deseado, independientemente del estado anterior, y en una sola instrucción.

- -

PRINT AT

El nuevo ORIC-Atmos posee la posibilidad de realizar un PRINT a partir de cualquier posición de la pantalla. Esta es una función de interés, por lo que a los usuarios del ORIC-1 que puedan necesitarla, pueden simularla mediante la siguiente rutina en Basic:

```
PRINT CHR$(17) : POKE #269,X : POKE #268,Y : DOKE #12,#BB8Ø + 4Ø * Y :
PRINT CHR$(17);
```

Antes de llamar la rutina, debemos asegurarnos que el cursor está activo (podemos hacer POKE #26A,3), y debemos colocar los valores deseados de columna y fila en X e Y, respectivamente.

Una particularidad es que permite la impresión en la línea reservada superior.

Es conveniente introducirla como una subrutina para poder ser llamada cuando haga falta (para posicionar el cursor antes de la impresión).

R E C E T A R I O

El presente capítulo pretende proveer una serie de "recetas" que permitan construir, paso a paso, las rutinas que precisemos para cada caso.

El primer problema que se nos plantea es decidir donde colocaremos la rutina. Si su longitud no sobrepasa los 256 octetos, el lugar más común suele ser la página 4, es decir, de $4ØØ a $4FF. Es la zona que menos problemas plantea, y además permite salvarla conjuntamente con el programa de Basic, según hemos descrito en varias ocasiones.

Para rutinas de mayor longitud podemos utilizar la zona de $98ØØ a $B3FF, siempre que entonces no utilicemos la alta resolución, ni el comando GRAB. En efecto, esta zona es ocupada por los caracteres y la pantalla en el modo HIRES.

De hecho, todo el espacio de memoria puede destinarse a estos fines, si tenemos en cuenta los posibles efectos que ello comporta (una línea de REM, la zona de la pantalla, los generadores de caracteres, etc. pueden contener rutinas en código máquina).

Ubicada la rutina, el segundo problema es donde disponer el espacio para las variables que la misma necesite. Las mismas zonas anteriores pueden destinarse a este fin, sin embargo, es interesante disponer de una zona en página cero para algunos valores.

Podemos hacer dos cosas:
- utilizar la zona del buffer de teclado (de $35 a $8Ø), si la rutina ha sido llamada desde un programa de Basic. Si, en cambio, ha sido llamada desde el teclado, la alteración del buffer nos puede provocar problemas al retorno. En este caso, no se puede retornar con un simple RTS, sino con un JMP $E7FF, que provoca la cancelación del resto de la línea.
- utilizar las mismas variables del sistema, o las que no está utilizando en este momento. En general, pueden usarse las direcciones de $Ø a $B sin ningún problema.

Una vez pensada qué zona destinaremos a la rutina y sus variables, debemos pensar cómo será llamada la rutina desde el Basic, y cómo le pasaremos los parámetros necesarios. Para llamar una rutina desde el Basic, disponemos de cuatro instrucciones: dos actuan como mandatos, y dos como funciones.

Los mandatos son CALL n, que transfiere la ejecución a la rutina en máquina que empieza en la dirección n, y ! (signo de admiración), que transmite la ejecución a la rutina que empieza en la dirección que previamente hayamos definido mediante DOKE #2F5,n. En ambos casos, al encontrar el RTS, se ejecuta la instrucción de Basic que se encuentre en la posición actual del puntero (si no lo hemos cambiado, la siguiente al CALL o al !).

En cuanto a las funciones, son USR() y &(), que transfieren el valor entre paréntesis al acumulador de punto flotante, antes de traspasar la ejecución a la rutina en máquina, cuya dirección se habrá definido mediante DEF USR o bien DOKE #2FC,n. En este caso, el Basic espera al retorno un valor en este mismo acumulador flotante.

Por lo que respecta al paso de parámetros, en el caso de las dos funciones que acabamos de citar, este paso es automático. En el caso de los mandatos CALL y ! debemos recurrir a los métodos de lectura de expresiones.

Cuando se inicia la ejecución de la rutina en código máquina, el puntero del Basic se encuentra situado en el siguiente carácter al último de la instrucción que la ha llamado. La rutina puede, pues, empezar con una subrutina que lea un parámetro colocado a continuación de la citada expresión. El método permite leer varios parámetros, que deben estar separados por algún símbolo adecuado, normalmente una coma.

Las rutinas apropiadas al caso son:

$CE77 : calcula el valor de la expresión siguiente, y lo pasa al acumulador flotante.

$D8ØD : calcula el valor de la expresión siguiente, comprueba que no supere 255, y lo pasa al registro X y a $D4.

$E79D : calcula el valor de la expresión siguiente, comprueba que no supere 65535, y lo pasa a los bytes $33 y $34. (también $CA98)

$CFD9 : comprueba que el siguiente carácter sea una coma, y salta el puntero al siguiente carácter después de la coma.

$CFDB : idéntica a la anterior, solo que el carácter buscado y saltado será el que tenga por código el valor contenido en A.

Una vez pasados los parámetros, suele ser necesaria su conversión a otras formas, para lo cual nos serán útiles las siguientes rutinas:

$D21Ø : transforma el valor contenido en el acumulador de punto flotante a valor íntegro sobre dos bytes, y lo coloca en $D3, $D4.

$D3FD : pasa el valor contenido en el registro Y al acumulador de punto flotante.

$D867 : pasa el valor del acumulador de punto flotante a íntegro sobre dos bytes, y lo coloca en $33-$34.

$D8D5 : pasa el valor del íntegro contenido en los registros A,Y al acumulador de punto flotante.

$DE73 : lee los cinco octetos a partir del apuntado por el contenido de los registros A,Y y los copia en el acumulador de punto flotante.

$EØD1 : transforma el contenido del acumulador de punto flotante en su expresión en caracteres ASCII, y los pasa a partir de la dirección $1Ø1, seguidos de un byte nulo.

Puede interesarnos también que la rutina pueda leer un carácter entrado por teclado, en este caso tenemos:

$E9Ø5 : lee la última tecla pulsada, y borra el buffer.

$C5F8 : espera la introducción de una tecla y la lee.

$D186 : *comprueba si el carácter contenido en A es una* letra de A a Z, y pone el carry a set en caso afirmativo y a clear en caso contrario.

$EA4E : comprueba si es un número, poniendo el carry a clear en caso afirmativo, y a set, en caso contrario.

Cuando, por el contrario, queremos bloquear el teclado, por ejemplo, para aumentar la velocidad de ejecución (anulando las interrupciones):

$E6CA : inhibe las interrupciones, anulando el control por teclado.(o $EDØ1)

$E8Ø4 : desinhibe las interrupciones, devolviendo el control al teclado.

Leídos los parámetros, y convertidos a la forma adecuada, solemos tener que realizar operaciones con ellos. Para ello, disponemos de dos acumuladores de punto flotante: elprimario, situado de $DØ a $D5, que llamamos ACC1; y el secundario, de $D8 a $DD, que llamamos ACC2. Las principales rutinas son:

Nombre	Direcc.	Utilidad
MOVFM	$DE73	Pasa el contenido de la memoria al ACC1. La dirección de la memoria viene dada por el puntero formado por A,Y.
CONUPK	$DD4D	Idem al ACC2.
MOVMF	$DEA5	Pasa el ACC1 a la memoria.
MOVAF	$DEDD	Pasa el ACC1 al ACC2.
MOVFA	$DECD	Pasa el ACC2 al ACC1.
FDIVT	$DDE3	Divide el ACC2 por el ACC1 dejando el cociente en ACC1.
FDIV	$DDEØ	Divide la memoria por el ACC1.
FDIV2	$DDDA	Divide el ACC2 por la memoria (cociente en ACC1).
DIV1Ø	$DDBF	Divide el ACC1 por 1Ø.
SQR	$E22A	Calcula la raíz cuadrada de ACC1.
POWER	$E231	Calcula el ACC2 a la potencia memoria (resultado en ACC1).
INT	$DFA5	Calcula la parte entera de ACC1.
ABS	$DF31	Calcula el valor absoluto de ACC1.
FADD	$DA97	Suma la memoria al ACC1.
FADDH	$DA79	Añade .5 al ACC1.
FSUB	$DA8Ø	Resta el ACC1 de la memoria (resultado en ACC1).
FMULT	$DCB7	Multiplica la memoria por el ACC1.
MUL1Ø	$DDA3	Multiplica el ACC1 por 1Ø.
SGN	$DF12	Calcula el signo del ACC1.
LOG	$DC79	Calcula el logaritmo del ACC1.
NEGOP	$E26D	Cambia el signo de ACC1.
RND	$E34B	Genera un número aleatorio en ACC1.
SIN	$E38E	Calcula el seno de ACC1.
COS	$E387	Calcula el coseno de ACC1.
TAN	$E3D7	Calcula la tangente de ACC1.
ATN	$E43B	Calcula el arco tangente de ACC1.
GIVAYF	$D3ED	Pasa el íntegro A,Y a ACC1.
QUINT1	$D871	Pasa el ACC1 a íntegro en $D3,$D4.
FOUT	$EØD1	Pasa el ACC1 a tira ASCII en $1ØØ y posteriores.

También, para la división de íntegros, disponemos de una rutina especialmente rápida en $EEFF, que divide el valor en $C,$D por el contenido en $2ØØ,$2Ø1 y deja el cociente en $C,$D y el resto en $E,$F.

Una vez realizados los cálculos, necesitaremos imprimirlos. Las rutinas que nos pueden resultar útiles para esta tarea son:

$CBED : imprime a partir de la posición actual del cursor un mensaje que previamente habremos colocado a partir de la dirección apuntada por el contenido de los registros A(peso bajo) Y(peso alto). El byte siguiente al último del mensaje debe ser un nulo.

$F82F : imprime un mensaje en la línea reservada superior. La dirección inicial de donde debe leerse el mensaje se encuentra en A,Y como antes. La distancia desde el byte inicial de la pantalla al primero donde debe colocarse el mensaje debe colocarse en el registro X.

$EØC1 : imprime a partir de la posición actual del cursor el valor decimal de un número entero contenido en los registros X(peso bajo) y A(peso alto).

$CC12 : imprime en la posición actual del cursor el carácter cuyo código ASCII se encuentre en el registro A. Acepta los caracteres de control como tales (la subrutina $CC4D no imprime en la impresora).

$CB9F : pasa el cursor al principio de la siguiente línea en la pantalla.

$F7AC : imprime en la posición actual del cursor el carácter cuyo código esté en A, pero considera los caracteres inferiores a 32 como atributos, no caracteres de control.

$CC0D : imprime un espacio.

$CC0A : limpia la pantalla, colocando el cursor al principio.

También resulta útil la combinación de las rutinas $E0D1 y $CBED para imprimir valores numéricos en punto flotante. Asimismo, las rutinas $D965 y $F6F1 nos permiten calcular la dirección del primer byte de una línea en pantalla, cuyo número de orden se haya colocado en A a la entrada de la rutina. La primera de estas rutinas retorna la dirección en $1F,$20; y la segunda a A,Y.

También podemos acelerar la ejecución de dibujos en alta resolución, llamando a las rutinas desde el código máquina, lo que nos evita el tiempo necesario para el pase de valores de los parámetros. Así tenemos:

<u>CURSET</u>	<u>DRAW</u>
LDX coordenada x	LDX distancia x
STX $219	STX $2E1
LDY coordenada y	LDY distancia y
STY $21A	STY $2E3
LDA código color 0-3	LDA código color 0-3
STA $212	STA $2E5
JSR $EDF6	JSR $F079

Siempre que trabajamos en alta resolución, tanto en Basic como en código máquina, conviene inicializar el PATTERN, por si un programa anterior lo hubiese alterado. El byte que contiene el código PATTERN es el $213. Pasamos a HIRES mediante JSR $E9BB, y volvemos a TEXT con $E9A9

Una de las ventajas del código máquina es su velocidad, pero puede suceder que en algunas ocasiones (juegos, música) nos interese provocar un retardo deliberado, entonces:

$FA61 : provoca un retardo de unos 8 milisegundos.

$FA63 : provoca un retardo de aproximadamente tantos milisegundos como indique el contenido del registro X.

Finalmente, cuando la rutina ha realizado su tarea, debemos retornar al control del Basic, para lo cual:

- el método más simple es un RTS, con lo que prosigue la ejecución del programa o línea desde donde se hubiera llamado la rutina (requiere que no se hayan alterado los punteros esenciales).
- JMP $E7D6 o bien JMP $E7FF cancela el resto del programa o de la línea, y deja la máquina a la espera de nuevas introducciones por teclado.
- LDX #tipo error, JMP $C485 envia a la rutina de error, para los casos en que hayamos detectado una incorrección en algún parámetro, etc.
- JMP $F42D provoca el borrado total de la memoria, e inicialización (puede ser útil para proteger programas).

- FIS

DICCIONARIO ORIC-1 / ORIC-ATMOS:

El ORIC-Atmos incorpora una nueva versión de la ROM (la versión 1.1) que presenta una serie de mejoras con respecto a la anterior. Por una parte, se han eliminado los célebres "bugs", como la incorrecta inicialización del HIMEM, el TAB, el ancho de impresora, el EDIT durante el scrolling, el uso de valores hexadecimales en el segundo parámetro del POKE, la posición del cursor al inicio de línea, la aparición del CAPS al conectar la máquina, el carácter inicial del STR$, el DRAW Ø,Ø, el cambio de punteros al leer un programa en código máquina, el funcionamiento en ocasiones anómalo del THEN y ELSE, el uso del apóstrofe al principio de linea, etc.

Por otraparte, se han añadido nuevas funciones, como el STORE y el RECALL, o nuevas posibilidades de las ya existentes, como la verificación de ficheros en cassette, el PRINT@, la mayor velocidad de los scrollings, etc. Todo ello se ha logrado mediante una reorganización de la ROM, rediseñando algunas rutinas para optimizarlas.

Como consecuencia de lo anterior, se ha producido un cambio prácticamente total en los puntos de entrada a cada rutina, lo que ha tenido como consecuencia la imposibilidad de utilizar la mayoría de programas en código máquina diseñados en principio para el ORIC-1, en el ORIC-Atmos, así como aprovechar la información existente para poder crear nuevas rutinas para la versión 1.1.

Sin embargo, en la mayor parte de los casos, basta saber los nuevos puntos de entrada a las rutinas, para poder realizar fácilmente la conversión. Es con este motivo, y a fin de permitir a los compradores del ORIC-Atmos poder sacar fruto del libro "Programación en Código Máquina del ORIC-1", que se ha preparado el presente diccionario.

En general, pues, bastará sustituir los JMP y JSR que vayan dirigidos a rutinas de la ROM, por la traducción que se encontrará a continuación en la tabla. Sin embargo, puede suceder en algunos casos que la traducción no sea totalmente equivalente, en especial las rutinas relativas al cassette han cambiado totalmente, y también algunos punteros situados en las primeras páginas de la memoria.

Sobre estos últimos hay ya información en el manual del ORIC-Atmos, por lo que pasaremos directamente a la tabla de conversión de direcciones de la ROM (si alguna dirección de interés no se encuentra en la misma, probablemente se pueda localizar por interpolación):

Nota: Por comodidad mecanográfica, y ya que en este caso no se presta a confusión, las cantidades hexadecimales se presentan sin ningún símbolo inicial, y los ceros se escriben "O" (sin la barra cruzada).

ORIC-1	ATMOS	Función de la rutina
C006	C006	tabla de punteros a las rutinas del Basic
COEA	COEA	tabla de los nombres reservados del Basic
C2AC	C2A8	tabla de los mensajes de error
C3F8	C3F4	abre espacio dentro de un programa para insertar una línea
C43B	C437	comprueba si hay espacio suficiente en el stack
C485	C47E	rutina general de error
C4B5	C4A8	entrada al lazo principal después de la inicialización
C4C7	C4B7	punto de retorno de la inserción
C4E3	C4D3	inserción de una línea en un programa de Basic
C56F	C55F	recalcula los links de las líneas de Basic
C5A2	C592	entrada de una línea en el buffer
C5F8	C5E8	lee una tecla
C60A	C5FA	tokenización de la línea presente en el buffer
C6A5	C692	rutina EDIT del Basic
C6DE	C6B3	busca un determinado número de línea
C6E8	C6BD	ídem a partir de cierto punto
C719	C6EE	rutina NEW del Basic
C733	C708	borra todas las variables del Basic
C738	C70D	rutina CLEAR del Basic
C765	C73A	lleva el puntero al principio del Basic
C773	C748	rutina LIST del Basic
C824	C7FD	rutina LLIST del Basic
C832	C809	rutina LPRINT del Basic
C841	C855	rutina FOR del Basic
C8DD	C90C	interpretación de las sentencias de Basic
C8FE	C915	ejecución de una instrucción
C91F	C952	rutina RESTORE del Basic
C930	C962	comprueba si se ha pulsado Control C
C93F	C971	rutina STOP del Basic
C941	C973	rutina END del Basic
C96E	C9A0	rutina CONT del Basic
C98B	C9BD	rutina RUN del Basic

CALL#C98D CALL#C9BF

C996	C9C8	rutina GOSUB del Basic
C9B3	C9E5	rutina GOTO del Basic
C9E0	CA12	rutinas RETURN y POP del Basic
CA0A	CA3C	rutina DATA del Basic
CA0D	CA3F	incrementa el puntero del intérprete en el valor de Y
CA1F	CA51	busca el fin de la línea de Basic
CA3E	CA70	rutina IF del Basic
CA61	CA99	rutina REM del Basic
CA66	CA9E	busca el ELSE o el final de línea
CA78	CAC2	rutina OR del Basic
CA98	CAE2	calcula la siguiente expresión y la pasa a 33-34
CAD2	CB1C	rutina LET del Basic
CB61	CBAB	rutina PRINT del Basic
CB9F	CBF0	salto a nueva línea de pantalla
CBED	CCB0	imprime mensaje que empieza en (A,Y) y termina con nulo
CC0A	CCCE	rutina CLS del Basic
CC12	CCD9	rutina general de impresión de un carácter
CC4D	CCFB	rutina particular de impresión de un carácter
CC89	CD13	rutina ! del Basic
CC8C	CD16	rutina TRON del Basic
CC8F	CD19	rutina TROFF del Basic
CCBA	CD46	rutina GET del Basic
CCC9	CD55	rutina INPUT del Basic
CCFD	CD89	rutina READ del Basic
CE0C	CE98	rutina NEXT del Basic
CE77	CF03	calcula la expresión siguiente
CE8B	CF17	subrutina de la anterior
CEE7	CF73	otra subrutina
CF74	D000	calcula el valor del siguiente número o variable
CFD3	D05F	comprueba el siguiente carácter
CFD6	D062	comprueba que sea "("
CFD9	D065	comprueba que sea ","
CFDB	D067	comprueba que sea el contenido en A
CFF0	D07C	busca el valor de una variable
D0F2	D17E	rutina DIM del Basic
D0FC	D188	busca una variable o la crea
D101	D18D	subrutina de la anterior
D186	D216	comprueba que sea un carácter A-Z

D210	D2A2	pasa el acum. de punto flotante a entero
D2A0	D336	subrutina
D3ED	D499	GIVAYF: pasa el entero A,Y al acumulador flotante
D401	D4BA	rutina DEF del Basic
D4FA	D5B5	calcula los punteros de la tira alfabética
D595	D650	rutina de "garbage collection"
D715	D7D0	calcula la longitud de una tira alfabética
D80D	D8C8	calcula la expresión siguiente como entero un byte
D852	D90D	posiciona puntero intérprete al final de una tira alfabética
D85B	D916	lee los parámetros para el POKE
D867	D922	acumulador flotante a entero sobre dos bytes
D871	D92C	QUINT1: acumulador flotante a entero dos bytes en D3-D4
D894	D94F	rutina POKE del Basic
D89D	D958	rutina WAIT del Basic
D8AC	D967	rutina DOKE del Basic
D8D5	DF40	entero A,Y a flotante
D937	D9DE	rutina LORES del Basic
D965	DA0C	calcula la dirección de inicio de una línea de pantalla
D996	DA22	GETVALS: lee los parámetros X,Y
D9C6	DA51	rutina PLOT del Basic
D9FA	DA85	rutina REPEAT del Basic
DA16	DAA1	rutinas PULL y UNTIL del Basic
DA6B	DAF6	subrutina
DA79	DB04	FADDH: añade .5 al acumulador flotante
DA80	DB0B	FSUB: resta el acumulador de la memoria
DA97	DB22	FADD: suma la memoria al acumulador
DC79	DCAF	LOG: calcula el logaritmo
DCB7	DCED	FMULT: multiplica la memoria por el acumulador
DD4D	DD51	CONUPK: pasa el valor de la memoria al segundo acumulador
DDA3	DDA7	MUL10: multiplica el acumulador por 10
DDBF	DDC3	DIV10: divide el acumulador por 10
DDDA	DDDE	FDIV2: divide el acumulador 2 por la memoria
DDE0	DDE4	FDIV: divide la memoria por el acumulador
DDE3	DDE7	FDIVT: divide el acumulador 2 por el acumulador 1
DE73	DE7B	MOVFM: pasa la memoria al acumulador 1
DE98	DEA0	subrutina
DEA5	DEAD	MOVMF: pasa el acumulador a la memoria

DECD	DED5	MOVFA: pasa el acumulador 2 al acumulador 1
DEDD	DEE5	MOVAF: pasa el acumulador 1 al acumulador 2
DF04	DF13	subrutina
DF12	DF21	SGN: calcula el signo del acumulador
DF1D	DF2C	subrutina
DF22	DF31	pasa el entero D1-D2 a flotante
DF31	DF49	ABS: calcula el valor absoluto del acumulador
DFA5	DFBD	INT: calcula la parte entera del acumulador
DFCF	DFE7	calcula el valor de un número expresado en ASCII
E0C1	E0C5	imprime la expresión del entero A,X
E0D1	E0D5	FOUT: pasa el acumulador a su expresión desde 101
E22A	E22E	SQR: calcula la raíz cuadrada del acumulador
E231	E235	POWER: eleva el acumulador a la memoria
E26D	E271	NEGOP: cambia el signo del acumulador
E34B	E34F	RND: genera un número aleatorio
E387	E38B	COS: calcula el coseno del acumulador
E38E	E392	SIN: calcula el seno del acumulador
E3D7	E3DB	TAN: calcula la tangente del acumulador
E43B	E43F	ATN: calcula el arco tangente del acumulador
E4A8	E4AC	lee un fichero en cassette (rutina distinta)
E563	E5F5	borra el mensaje de la línea de estado
E576	E5F5	otra entrada (las rutinas de cassette son distintas)
E630	E6C9	RDBYTE: lee un byte del cassette
E696	E735	GETSYN: sincroniza el tono inicial
E6CA	E76A	inhibe las interrupciones
E725	E7B2	lee los parámetros para el cassette
E79D	E853	calcula la expresión siguiente y la pasa a $33-34
E7AA	E85B	rutina CLOAD del Basic
E7DB	E9o9	rutina CSAVE del Basic
E804	E93D	desinhibe las interrupciones
E80D	E946	rutina CALL del Basic
E813	E94C	calcula la expresión hexadecimal siguiente
E87D	EAF0	rutinas gráficas del Basic
E889	EAFC	rutinas sonoras del Basic
E95B	EBCE	rutina HIMEM del Basic
E974	EBE7	rutina GRAB del Basic
E994	ECOC	rutina RELEASE del Basic

E9A9	EC21	rutina TEXT del Basic
E9BB	EC33	rutina HIRES del Basic
EA41	EC89	comprueba que el siguiente carácter sea un número
EBD0	EDE0	tabla de JMPs
EBDF	F0C8	CURSET
EBE2	F0FD	CURMOV
EBE5	F110	DRAW
EBE8	F37F	CIRCLE
EBEB	F11D	PATTERN
EBEE	F12D	CHAR
EBF4	F204	PAPER
EBF7	F210	INK
EBFA	F268	FILL
EBFD	EE1A	inhibe teclado
EC03	EE22	rutina de interrupción por el VIA
ED01	EE1A	inhibe teclado
ED09	EE22	rutina de interrupción por el VIA
EDF6	EEE8	CURSET
F02D	F0C8	CURSET (según condiciones de entrada)
F064	F0FD	CURMOV
F079	F110	DRAW
F093	F11D	PATTERN
F0A5	F12D	CHAR
F17F	F204	PAPER
F18B	F210	INK
F1E5	F268	FILL
F2E5	F37F	CIRCLE
F412	FA9F	PING
F415	FAB5	SHOOT
F418	FACB	EXPLODE
F41B	FAE1	ZAP
F41E	FB40	SOUND
F421	FBD0	PLAY
F424	FC18	MUSIC
F439	F9AA	subrutina
F535	F590	W8912: escribe en el generador de sonido
F89B	F8D0	reinicializa los caracteres
FA6C	FA86	pasa parámetros en bloque al generador de sonido

FAFA	FB14	KBBEEP: clic del teclado
FB10	FB2A	CONTBP: clic del control
FB26	FB40	SOUND
FBB6	FBDO	PLAY
FBFE	FC18	MUSIC
FFFA	FFFA	reset parcial: contiene 247
FFFC	FFFC	reset general: contiene F88F
FFFE	FFFE	interrupción por el VIA: contiene 244
F42D		arranque en frío

- -

Especial interés suelen revestir en algunas rutinas los vectores relativos a las interrupciones, a saber:

0228	0244	INTFS salta a la rutina de tratamiento de interrupción por VIA
022B	0247	NMIJP salta a la rutina de tratamiento de la NMI
0230	024A	INTSL retorno de la rutina de tratamiento de interrupción

La rutina de reloj y la que usa la tecla FUNCT son ejemplos de uso de estos vectores.

- -

Cuando uno se cansa de ver siempre el mismo tipo de letras, le queda la posibilidad de redefinir los caracteres. Esta es una tarea tediosa, y en todo caso limitada a la matriz 6x8, que no permite excesivas filigranas.

Sin embargo, una manera fácil de cambiar el tipo de letra puede ser la ejecución del siguiente programa:

```
1Ø  TEXT
2Ø  FOR A = #B5ØØ TO #B7FF
3Ø  B = PEEK (A)
4Ø  IF (B AND #38) = #2Ø THEN POKE A,B OR #1Ø
5Ø  NEXT : PING
```

Recordemos que cuando queramos volver al tipo de letra inicial, nos bastará hacer CALL #F8DØ (o bien CALL #F89B en el ORIC-1).

- -

Otra diferencia entre las dos versiones del ORIC es la numeración de las columnas de la pantalla de baja resolución, que empieza en la segunda columna en el ORIC-1, y en la primera en el ORIC-Atmos (con lo que llega a 38 en la primera versión, y a 39 en la segunda).

Hay que tenerlo presente al convertir algunos programas en Basic que utilicen gráficos.

LA TECLA "FUNCT":

Uno de los hechos que más comentarios suscitó ya desde la llegada de las primeras informaciones gráficas sobre el nuevo ORIC-Atmos, fue la existencia de una tecla marcada con "FUNCT" en el extremo inferior derecho del teclado, así como su posible utilidad.

La aparición del manual no aclaró mucho las cosas, pues en el mismo ni siquiera se cita dicha tecla. ¿Cuál es, pues, su misión?

Según parece, se trata de una tecla de "reserva", prevista por los constructores para futuras versiones del ORIC que incorporarán este mismo teclado, y en las cuales la tecla FUNCT tendrá un cometido específico.

También se prevé que determinados programas de Tansoft la utilicen, así como los que el mismo usuario pueda preparar, para funciones especiales, generalmente desde el código máquina, aunque también es accesible desde el Basic.

La manera más simple de detectar la pulsación de esta tecla es leyendo el contenido de la dirección de memoria $2Ø9. Esta dirección es la que detecta la pulsación de las teclas CTRL y SHIFT. Mientras ninguna de estas teclas está pulsada, su contenido vale $38, pero durante la pulsación de una de ellas su contenido se modifica a los valores:

- pulsando la tecla CTRL, se obtiene $A2
- pulsando la tecla SHIFT izquierda $A4
- pulsando la tecla SHIFT derecha $A7
- pulsando la nueva tecla FUNCT $A5

En todos los casos, el valor $38 retorna al dejar de pulsar la tecla. Es de destacar el hecho de que ambas teclas SHIFT no están interconectadas, sino que corresponden efectivamente a teclas diferenciadas, lo que puede ser utilizado mediante las adecuadas rutinas.

Una interesante aplicación de la utilización de la tecla FUNCT lo constituye la rutina que se describe a continuación. Se trata de una corta (183 bytes) rutina en código máquina, emplazada en las posiciones de memoria desde $4ØØ hasta $4B6, que redefine el teclado de modo que cuando la tecla FUNCT se mantiene pulsada, al pulsar simultáneamente cualquier otra tecla, aparece directamente el nombre completo de una instrucción del Basic.

De este modo, el trabajo de copiar largos programas se ve considerablemente simplificado. Para facilitar su utilización, las principales instrucciones han sido asignadas a la inicial correspondiente (por ejemplo: F = FOR, G = GOTO, R = REPEAT, L = LIST, I = INPUT, etc.), o si esta se hallaba ya ocupada, a otra letra característica (X = EXPLODE, Y = PLAY, O = POKE, S = GOSUB, etc.), o a algún signo que sea fácilmente recordable (así, las flechas izquierda y derecha corresponden a LEFT$ y RIGHT$).

Las teclas conservan la facultad de repetición, y se niegan a continuar cuando llegamos al final de la línea, para evitar la pérdida total de la misma. Evidentemente, si no pulsamos FUNCT, el teclado funciona como de costumbre.

Para poder introducir fácilmente la rutina, he preparado un programa cargador autoverificante en Basic, que sirve a este fin, y es el siguiente:

9C6D

```
1Ø ' Redefinicion del teclado
15 D = #4ØØ
2Ø FOR L = 1ØØ TO 114
25 :  S = Ø : READ A$
3Ø :  FOR C = 1 TO 25 STEP 2
35 :     B = VAL("#" + MID$(A$,C,2))
4Ø :     S = (S+B) AND #FF : POKE D,B : D = D + 1
45 :  NEXT C
5Ø IF S<>VAL("#" + RIGHT$(A$,2)) THEN PRINT "Error en la línea" L
55 NEXT L
6Ø END
65 ' Si todo ha ido bien, no olvides hacer CALL #4ØØ
7Ø ' y sálvalo mediante CSAVE "", A#4ØØ, E#4B6, AUTO, S
1ØØ DATA A91ØAØØ4A24C8D4BØ28C4CØ28E8D
1Ø1 DATA 4AØ26Ø48ADØ9Ø2C9A5DØ4B984815
1Ø2 DATA 2Ø78EB1Ø42C95E9ØØ2E921C9ØE6F
1Ø3 DATA 9ØØ2E9ØDA8B965Ø485ØØA9ECAØØC
1Ø4 DATA CØ85Ø184Ø2AØØØB1Ø13ØØ7C8DØED
1Ø5 DATA F9E6Ø2DØF5C6ØØDØF5EØ4DBØ1725
1Ø6 DATA C8DØØ2E6Ø2B1Ø1Ø8297F95352ØCE
1Ø7 DATA D9CCE8281ØE968A8684ØA9Ø72Ø36
1Ø8 DATA D9CC4C61Ø4747 5Ø922Ø11C31Ø1B9
1Ø9 DATA Ø1Ø1Ø176Ø1Ø1Ø1Ø1Ø1Ø11DØ1Ø19E
11Ø DATA Ø1Ø12A6D2B2C7Ø6FØA1A5C571EC4
111 DATA 2115 11Ø16AØ12DØ1Ø1Ø16C583FE6
112 DATA 6748ØD175C123E713C2810396 6Ø3
113 DATA 32ØB1B49ØC6B35242925364237 6E
114 DATA 145555555555555555555555551Ø
```

Si al copiar la rutina os equivocáis en alguna línea de DATA, el programa lo detecta y avisa del número de línea en que se ha producido el fallo, a fin de que pueda ser corregida. Para iniciar la redefinición del teclado, una vez ejecutado el programa, hay que hacer CALL #4ØØ.

Como es natural, el programa Basic solo sirve para cargar la rutina, por lo que puede ser borrado, después de haber salvado la rutina mediante un

```
CSAVE "TECLADO", A#4ØØ, E#4B6, AUTO, S
```

De este modo, un simple CLOAD" nos dejará el teclado redefinido.

La rutina utiliza los registros Ø, 1 y 2, lo que debe tenerse en cuenta si los empleamos simultáneamente para alguna otra función.

(La rutina ha sido comprobada, copiándola del texto mecanografiado, para garantizar la ausencia de errores). Para los interesados en su funcionamiento, es el siguiente:

```
400  LDA  #$10      ;dirección de inicio, peso bajo
     LDY  #$04      ;ídem, peso alto
     LDX  #$4C      ;código JMP
     STA  $24B      ;modificamos vector INTSL (retorno interrupción)
     STY  $24C
     STX  $24A
     RTS
410  PHA            ;esta rutina se ejecuta después de cada interrupción
     LDA  $209
     CMP  #$A5      ;¿estaba pulsada la tecla FUNCT?
     BNE  $463      ;en caso contrario, retornamos
     TYA
     PHA            ;salvamos Y
     JSR  $EB78     ;leemos carácter introducido por teclado
     BPL  $461      ;si no lo había, retornamos
     CMP  #$5E      ;serie de operaciones destinadas a empaquetar los
     BCC  $425      ;códigos válidos, con el fin de reducir el tamaño
     SBC  #$21      ;de la tabla necesaria
425  CMP  #$0E
     BCC  $42B
     SBC  #$0D
42B  TAY
     LDA  $465,Y    ;leemos número de orden de la palabra reservada
     STA  0         ;lo salvamos en el registro 0
     LDA  #$EC      ;dirección inicio tabla nombres, peso bajo
     LDY  #$C0      ;ídem, peso alto
     STA  1         ;lo salvamos en el puntero (1-2)
     STY  2
     LDY  #0
43B  LDA  (1),Y     ;leemos un carácter de la tabla de nombres
     BMI  $446      ;si era el final de un nombre, saltamos
43F  INY            ;incrementamos puntero
     BNE  $43B
     INC  2
     BNE  $43B
446  DEC  0
     BNE  $43F      ;si no era el nombre buscado, saltamos
44A  CPX  #$4D      ;estamos en los últimos bytes de la línea?
     BCS  $465      ;si es así, saltamos y suena la campana
     INY
     BNE  $453
     INC  2
453  LDA  (1),Y     ;leemos carácter de la tabla de nombres
     PHP
     AND  #$7F
     STA  $35,X     ;lo pasamos al buffer de entrada de teclado
     JSR  $CCD9     ;lo imprimimos
     INX            ;incrementamos puntero del buffer
     PLP
     BPL  $44A      ;si no era el último, leemos el siguiente
     PLA
```

```
     TAY
463  PLA            ;recuperamos registros antes de retornar
     RTI
465  LDA  #$Ø7      ;código ASCII de la campana
     JSR  $CCD9
     JMP  $461
46D  . . .          ;empieza la tabla de números de orden
```

A partir de este listado debe resultar fácil adaptarlo para ser ubicado en otra zona de memoria, o bien para el uso de la tecla SHIFT derecha, en lugar de FUNCT, a fin de adaptarla a su uso en el ORIC-1.

En este último caso, hay que recordar las siguientes diferencias entre el ORIC -1 y el ATMOS:

- el retorno de la rutina de interrupción, que en el caso del ORIC-Atmos se halla en $24A-$24C, en el ORIC-1 está en $23Ø-$232;
- las subrutinas utilizadas deben ser cambiadas, según la siguiente tabla:

Atmos	ORIC-1	
$EB78	$E9Ø5	GTORKB lee un carácter del teclado
$CCD9	$CC12	rutina general de impresión

- por último, deberá modificarse la tabla, ya que al utilizar la tecla SHIFT el carácter introducido queda modificado.

Para un funcionamiento normal, el teclado debe estar en modo CAPS, ya que en caso contrario, las funciones asignadas a cada tecla pueden cambiar. Cabría aprovechar este hecho para ampliar el número de funciones directamente accesibles, pero pienso que ello eliminaría la simplicidad, que es el principal objetivo de la rutina.

Lo usual es que desde un programa en Basic llamemos a una rutina en código máquina, y en este libro hemos visto distintas maneras de lograrlo.

Sin embargo, puede suceder que deseemos que una rutina en código máquina nos envíe a determinada línea de Basic (el equivalente de un GOTO número de línea) sin borrar el contenido de las variables que previamente pudieran existir.

Ello puede lograrse de modo sencillo sin más que colocar en las direcciones $33-$34 el valor del número de línea al que queramos saltar, y a continuación hacer un JMP a la dirección $C9FC (si se trata de un ORIC-Atmos) o a la dirección $C9CA (en el caso del ORIC-1).

CASSETTE:

Es quizás en el apartado del trabajo con el cassette donde el nuevo ORIC-Atmos presenta más diferencias con el ORIC-1. No solo se han incluido dos nuevas instrucciones: STORE y RECALL, sino que se han añadido nuevas funciones de las ya existentes, habiéndose procedido a rediseñar completamente el conjunto.

Así, ya no se utiliza la zona del buffer de teclado para guardar los distintos parámetros, sino determinados registros de la página 2, que luego comentaremos. De este modo, en una misma línea podemos incluir varias instrucciones de salvaguarda o lectura (es decir, si queremos obtener dos copias de un programa, podemos hacer: CSAVE"PROGRAMA":CSAVE"PROGRAMA").

También la rutina de lectura de parámetros es más flexible: así podemos hacer, por ejemplo: A$ = "PROGRAMA" : CSAVE A$.

La impresión de los distintos mensajes en la línea de estado queda inhibida si nos hallamos en alta resolución, evitando la línea negra que aparecía en estos casos.

Asimismo, el bug que hacía que los punteros de fin del Basic se modificaran al leer un programa en código máquina, también ha sido solucionado. Sin embargo, ello obliga a adoptar alguna precaución cuando queramos salvar conjuntamente rutinas en máquina y un programa Basic en un mismo fichero. Pero resumamos primero las principales direcciones de esta sección:

$C55F recalcula y corrige los links de un programa de Basic. Es sistemáticamente ejecutada después de leer un programa de Basic, para evitar la sucesión de "U" que se obtenía en ocasiones al listar un programa con algún error.

$C7Ø8 borra las variables (en realidad corre los punteros a sus valores iniciales), limpia el stack y posiciona el puntero del intérprete al principio del programa de Basic. También es ejecutado después de toda lectura de un fichero de programa Basic.

$CCBØ imprime un mensaje apuntado por (A,Y) y terminado en un nulo, después de la posición actual del cursor. Usado para los mensajes: "Errors found" (a partir de $E527), y "Verify errors detected" (a partir de $E552).

$DØ7Ø imprime el mensaje "SYNTAX ERROR" y aborta la ejecución. Usado durante la lectura de los parámetros, en caso de detección de error.

$EØ5C imprime un número, cuyo contenido se haya colocado previamente en los registros A y X. Usado para imprimir el número de errores de verificación (contenidos en $25C-$25D).

$E4AC lee el cabezal de un fichero en cassette, pasando los parámetros a partir del registro $2A7, y el nombre a partir de $293. Si este no coincide con el que previamente hubiéramos introducido a partir de $27F, imprime el mensaje FOUND y busca el siguiente.

$E4EØ lee el fichero desde el cassette, pasándolo a la memoria, o comparándolo con ella, según el contenido del registro $25B (distinto de cero si estamos en Verificación).

$E56C comprueba si hemos llegado al fin del fichero, poniendo el carry en consecuencia.

$E57D imprime el mensaje "Searching" en la línea de estado, si el registro $21F (que señala el modo HIRES) está a cero

$E585 ídem el mensaje "Saving".

$E594 imprime el nombre del fichero en la línea de estado.

$E59B ídem mensaje "Loading", o "Verifying".

$E5F5 borra la línea de estado, excepto el CAPS (siempre, según el estado del registro $21F).

$E607 salva al cassette el cabezal con los datos de los parámetros y el nombre del fichero.

$E62E salva en el cassette el fichero propiamente dicho.

$E651 imprime el mensaje "Errors found".

$E65E rutina OUTBYT, envía un byte al cassette.

$E6C9 rutina RDBYTE, lee un byte desde el cassette, posicionando el registro $2B1 a un valor distinto de cero si no se verifica la paridad (recordar, en efecto, que en la página 33 ya se indicaba que los bytes se grababan con un bit suplementario de paridad, que en el ORIC -1 no se utilizaba).

$E735 rutina GETSYN, va leyendo bits hasta formar el valor $16 (código ASCII de sincronización), con lo que queda sincronizado el tono inicial. Ello es necesario porque los bits se transmiten en serie, y es necesario saber donde empieza cada byte.

$E75A rutina OUTLED, manda al cassette una serie de bytes $16 para sincronización.

$E76A inhibe interrupciones de teclado, y prepara al VIA para el trabajo con el cassette.

$E790 compara el nombre hallado en el fichero leído, con el especificado.

$E7B2 rutina general de lectura de los parámetros (ver luego el detalle de cada uno de ellos).

$E85B rutina CLOAD del Basic.

$E909 rutina CSAVE del Basic.

$E93D activa nuevamente el teclado, desconectando el cassette.

$F865 rutina STOUT, imprime un mensaje en la línea de estado.

Todas estas rutinas utilizan una serie de parámetros en la página 2, cuya misión específica es la siguiente:

$21F si su contenido es distinto de cero, indica que estamos en modo HIRES (se inhibe la impresión en la línea de estado)

$24D si es distinto de cero, velocidad lenta (300 bauds)

$25A si es distinto de cero, fichero a mezclar (J)

$25B si es distinto de cero, se trata de verificación (V)

$25C-$25D contiene el número de errores hallados durante el proceso de verificación de un fichero.

$27F-$28F contiene el nombre especificado del fichero a buscar o salvar, terminado con un byte nulo.

$293-$2A3 contiene el nombre del fichero hallado, terminado en un nulo.

$2A7 dirección a partir de la cual se conservan los nueve bytes del cabezal del fichero.

$2A9-$2AA dirección de inicio del fichero.

$2AB-$2AC dirección final del fichero.

$2AD un valor distinto de cero indica que el fichero es de auto-carga.

$2AE el bit de peso alto puesto a uno indica fichero en código máquina. Se considera que un fichero a salvar en cassette es de código máquina si se especifica al menos una de las direcciones de inicio o fin mediante A o E.

$2B1 un valor distinto de cero indica la existencia de un error de lectura.

Después de un CLOAD, el ORIC-Atmos comprueba si el programa leído era Basic o código máquina. Si se trataba de un programa Basic, actualiza los punteros de fin de Basic (puntero $9C) al valor del fin del fichero leído ($2AC). Sin embargo, si se trataba de un programa en código máquina, tal actualización no es efectuada (precisamente para eliminar uno de los "bugs" del ORIC-1).

Ahora bien, cuando queremos grabar conjuntamente un programa de BASIC junto con una rutina en máquina, al especificar la dirección de comienzo se activa la señal indicativa de código máquina ($2AE), con lo que la actualización de los punteros no se llevará a cabo.

La manera de solucionarlo es grabar el fichero con AUTO, iniciando la rutina con las instrucciones:

```
LDA  #Ø
STA  $2AD      ;desactivamos auto-arranque
JMP  $E8E9     ;y saltamos a actualización de los punteros
```

Desactivamos el auto-arranque, porque de lo contrario, al saltar a $E8E9, se pondría en marcha el programa de Basic (a no serque sea éste el fin perseguido). Si queremos que se ejecute la rutina en máquina, la secuencia anterior debe colocarse al final de la misma (en lugar del habitual RTS).

Un inconveniente de este procedimiento es que si se produce durante la lectura algún error, se anula la autoejecución, y por tanto, no se produce la actualización del puntero. Conviene no utilizar el programa en tales condiciones.

Y a propósito de errores de lectura, algunos magnetofones producen un error en el primer byte del fichero (según se comenta en el manual), debido al tiempo de estabilización de la ganancia del amplificador, al pasar de nivel de entrada nulo al nivel standard. El propio manual sugiere una rutina que anula la señal de error.

Ello presenta, sin embargo, el inconveniente de que anulamos la verificación del fichero (todos darán correctos, aunque haya errores), y de requerir la presencia del programa citado para poder proceder a la lectura.

Una solución que suele dar mejor resultado en estos casos es anular externamente el efecto del control automático de ganancia, anteponiendo a la entrada de señal del magnetofón un potenciómetro graduado al mínimo necesario para obtener la señal justa.

APLICACIONES PARA EL ORIC-ATMOS:

Las rutinas descritas en el capítulo "Aplicaciones" (a partir de la página 59) son fácilmente adaptables al ORIC-Atmos sin más que utilizar el diccionario de conversión. Sin embargo, para evitar esta tarea y colocar así en igualdad de condiciones a los usuarios del nuevo equipo, a continuación describo los valores correspondientes al ORIC-Atmos.

El programa cargador, las instrucciones generales, así como las particulares de cada rutina, son idénticas que para el ORIC-1, salvo que se indique lo contrario. De todos modos, para salvar conjuntamente la rutina en código máquina junto con el programa de Basic, debe grabarse este con AUTO, a fin de que la parte inicial de la rutina restaure el puntero de fin del Basic (consúltese para mayores aclaraciones a este punto, el capítulo dedicado al cassette en el Atmos).

Entrada de expresiones en un INPUT.

Dirección inicial: #4DØ

Bytes: 169, Ø, 141, 173, 2, 76, 233, 232, 165, 234, 72, 165, 233, 72, 32, 146, 197, 134, 233, 132, 234, 32, 226, Ø, 24Ø, 244, 32, 25Ø, 197, 162, Ø, 181, 53, 24Ø, 6, 157, 12, 5, 232, 2Ø8, 246, 1Ø4, 133, 233, 1Ø4, 133, 234, 96.

Checksum: 87613Ø.

A resaltar que aunque la dirección inicial ha cambiado, en cambio el CALL #4D8 de la línea 5 no debe variar.

RESTORE a un número de línea.

```
LDA #$ØØ    BCS $Ø3     LDA $CF
STA $Ø2AD   JMP $CA23   SBC #$ØØ
JMP $E8E9   LDA $CE     STA $81
JSR $E853   SBC #$Ø1    RTS
JSR $C6B3   STA $BØ
```

Dirección inicial: #4EØ

Bytes: 169, Ø, 141, 173, 2, 76, 233, 232, 32, 83, 232, 32, 179, 198, 176, 3, 76, 35, 2Ø2, 165, 2Ø6, 233, 1, 133, 176, 165, 2Ø7, 233, Ø, 133, 177, 96.

Checksum: 511649.

La línea 1Ø debe ser sustituida por 1Ø DOKE #2F5,#4E8.

Impresión de caracteres gigantes.

Dirección inicial: #4BØ

Bytes: 169, Ø, 141, 173, 2, 76, 233, 232, 32, 34, 218, 32, 1Ø1, 2Ø8, 32, 2ØØ, 216, 16Ø, 7, 14Ø, 227, 2, 138, 32, 126, 241, 16Ø, Ø, 177, 12, 162, 6, 172, 248, 2, 1Ø, 1Ø, 1Ø, 72, 144, 3, 169, 16Ø, 44, 169, 32, 145, 31, 1Ø4, 2ØØ, 192, 4Ø, 176, 3, 2Ø2, 2Ø8, 236, 169, 4Ø, 24, 1Ø1, 31, 133, 31, 144, 2, 23Ø, 32, 23Ø, 12, 2Ø8, 2, 23Ø, 13, 2Ø6, 227, 2, 16, 2Ø3, 96.

Checksum: 861Ø63.

Se conserva el DOKE #2F5,#4B8.

Verificación de la lectura desde cassette.

Dada la verificación automática del Atmos, decrece su interés, por lo que no la he traducido.

Conversión a binario.

Primera rutina:

Dirección inicial: #4E7

Bytes: 169, Ø, 141, 173, 2, 76, 233, 232, 32, 2ØØ, 216, 16Ø, 8, 169, 24, 6, 212, 42, 32, 217, 2Ø4, 136, 2Ø8, 245, 96.

Checksum: 425513.

Se mantiene el DOKE #2F5,#4EF.

Segunda rutina:

Dirección inicial: #4E5

Bytes: 169, Ø, 141, 173, 2, 76, 233, 232, 32, 83, 232, 16Ø, 16, 169, 24, 6, 51, 38, 52, 42, 32, 217, 2Ø4, 136, 2Ø8, 243, 96.

Checksum: 4Ø4692.

Y también se mantiene el DOKE #2F5,#4ED.

Cómo tener dos programas Basic en memoria.

Dirección inicial: #4B3.

Bytes: 169, Ø, 141, 173, 2, 76, 233, 232, 165, 167, 2Ø1, 78, 144, 3, 76, 124, 196, 169, Ø, 133, 176, 133, 178, 169, 5, 133, 177, 169, 78, 133, 179, 16Ø, 1, 162, 72, 177, 176, 72, 177, 178, 145, 176, 1Ø4, 145, 178, 2ØØ, 2Ø8, 243, 23Ø, 177, 23Ø, 179, 2Ø2, 16, 236, 32, 95, 197, 24, 165, 145, 1Ø5, 2, 133, 156, 165, 146, 1Ø5, Ø, 133, 157, 32, 8, 199, 76, 168, 196.

Checksum: 1339ØØ.

R E L O J

Para adaptar la rutina reloj incluida en la página 3Ø, basta cambiar las líneas:

```
1Ø  A$ = "2Ø6AE7A924AØØ48D45Ø28C46Ø2A24Ø8E5FØ42Ø3DE92Ø65DØA5E9A4EAA2Ø22Ø65F84C"
2Ø  A$ = A$ + "3FCAØ8CE5FØ4DØ21488A48A9648D5FØ4A2Ø7FE82BBBD82BBDD57Ø49ØØ9BD4F"
3Ø  A$ = A$ + "Ø49D82BBCA1ØEC68AA68284C22EE3Ø3Ø3A3Ø3Ø3A3Ø3Ø3A3A3A363A3A363AØØ"
```

(las líneas 4Ø a 8Ø permanecen invariables)

```
9Ø  IF S<>12795Ø5 THEN PRINT "Error escritura"
```

La utilización es similar a la descrita en el capítulo correspondiente. Podemos parar el reloj mediante DOKE #245,#EE22.

A resaltar el hecho de que en este caso el reloj es independiente del funcionamiento de la máquina, mientras que en el ORIC-1 las funciones de impresión lo retrasaban en cierta medida. Ello le da una mayor utilidad en el Atmos.

TIEMPOS DE EJECUCION

Una de las principales razones para utilizar el código máquina es su velocidad de ejecución. Sin embargo, es posible con un adecuado estudio de las rutinas, mejorar también la velocidad de ejecución de nuestros programas en Basic, sin tener que recurrir siempre a la complicación del código máquina.

Evidentemente, la mejor manera de ir aprendiendo como mejorar la rapidez de nuestros programas es:

- comprender el funcionamiento del Basic,
- y, a partir de este, comparar distintos procedimientos para apreciar las mejoras conseguidas.

De este modo, iremos aprendiendo distintas técnicas. Sin embargo, debemos pensar que no son totalmente independientes, por lo que el resultado de aplicar una de ellas depende del hecho de que se aplique o no otra de las mismas.

Conviene, pues, efectuar las pruebas sobre el programa al que vayan a aplicarse, y para ello requerimos medir la duración de ejecución de determinada sección de un programa.

Esta tarea de cronometraje puede realizarse de modo manual, pero resulta poco práctico e inexacto, por lo que interesa dejar esta labor a la propia máquina. Afortunadamente, el ORIC posee un contador de interrupciones de dos bytes, situado en las direcciones $276 y $277, que nos servirá para este fin.

En efecto, como que las interrupciones se producen cada centésima de segundo, contar las interrupciones equivale a contar las centésimas de segundo transcurridas.

Aquí debemos hacer una observación. En el ORIC-1, algunas funciones de impresión inhibían las interrupciones, falseando el procedimiento. En cambio, en el ORIC-Atmos, no sucede tal cosa, por lo que podemos utilizar este sistema, mientras no utilicemos la función WAIT, o las de trabajo con el cassette, que anulan ambas las interrupciones.

El procedimiento consistirá en poner a cero el contador de interrupciones, mediante DOKE #276,Ø o mejor mediante un simple WAIT Ø, y, al final de la rutina a cronometrar, poner PRINT (#FFFF - DEEK (#276))/1ØØ ; "segundos".

Así, por ejemplo, para cronometrar un bucle, haremos:

```
1Ø WAIT Ø
2Ø FOR N # 1 TO 1ØØØØ
3Ø NEXT N
4Ø PRINT (#FFFF - DEEK(#276))/1ØØ ; "segundos"
```

Haciendo RUN, veremos que el tiempo empleado es de 17.15 segundos (o parecido). Si eliminamos el N del NEXT N de la línea 3Ø, el tiempo se reduce a 13,43 segundos, y si incluímos el NEXT en la misma línea 2Ø, a 12,93 segundos.

Un hecho que enseguida observaremos es que si en un programa debemos utilizar una constante, es mucho más rápido escribirla en hexadecimal. Así, por ejem-

plo:

```
2Ø  FOR N = 1 TO 1ØØØ
25  A = 15Ø : B = 342 : C = 412
3Ø  NEXT
```

tarda 16,Ø8 segundos, mientras que su equivalente en hexadecimal:

```
2Ø  FOR N = 1 TO 1ØØØ
25  A = #96 : B = #156 : C = # 19C
3Ø  NEXT
```

solo emplea 8.23 segundos en la misma operación, es decir, aproximadamente la mitad. Y aun puede ganarse algo más de tiempo sustituyendo los valores constantes por variables adecuadamente inicializadas al principio del programa.

De todos modos, al sustituir constantes por variables, debemos ponderar el negativo efecto producido por el hecho de que al haber más variables, las que ocupen los últimos lugares serán de acceso mucho más lento, y en conjunto el programa puede ser menos veloz.

Así pues, un punto interesante es ver en qué partes concretas de un programa es necesario apurar al máximo la velocidad (las rutinas que más veces se repitan), y en cuáles no tiene tanta importancia, y sí, en cambio la concreción y claridad de exposición.

Otra utilidad del procedimiento radica en ciertos juegos en los que pueda resultar interesante medir el tiempo de respuesta. Por ejemplo:

```
WAIT Ø : GET K$ : T = (#FFFF - DEEK (#276))/1ØØ
```

nos calculará el tiempo que tardamos en introducir la respuesta (medición de reflejos).

Al aplicarlo para medir la duración de juegos (o de programas de muy larga duración) debemos tener presente que el mayor tiempo a cronometrar es de 655.35 segundos (unos 11 minutos). Pasado este tiempo, el contador vuelve a empezar desde cero.