Memoria y variables del Sistema en la ZX Spectrum y TS 2068

Nota preliminar:

En este trabajo intentaremos dar una introducción a lo que son las variables del sistema en las computadoras ZX Spectrum y TS 2068, dando aplicaciones prácticas. Vamos también a realizar un pequeño análisis del uso de la memoria en ambos equipos.

Este trabajo tiene una porción que es copia directa del Manual de Programación en Basic que venía con la Spectrum[1] (de donde copiamos casi literalmente el cuadro de resumen de variables del sistema). Otras referencias utilizadas (¡pero no copiadas literalmente!) son el segundo número de la revista MicroHobby Especial[2], el Manual de Usuario de la TS 2068[3], y los libros TS 2068 Technical Reference Manual[4], The Spectrum Operating System[5], TS 2068 Beginner/Intermediate Guide[6] y TS 2068 Intermediate/Advanced Guide[7].

[1] Se puede obtener una copia escaneada de http://www.speccy.org/manuales/indice.html
[2] Los 7 números se obtienen en ftp://ftp.worldofspectrum.org/pub/sinclair/magazines/MicroHobbyEspecial/
[3] Se puede descargar desde http://www.timexsinclair.org/unsorted/TS2068-Manual.pdf
[4] Se descarga desde http://www.timexsinclair.org/dl/TS2068TechnicalManual.zip
[5] ISBN 0-7447-0019-1, escrito por Steve Kramer, publicado en 1984 por la editorial Micro Press
[6] ISBN 0-672-22225-6, escrito por Fred Blechman, publicado en 1985 por la editorial Howard W. Sams & Co.
[7] ISBN 0-672-22226-4, escrito por Jeff Mazur, publicado en 1985 por la editorial Howard W. Sams & Co.

Cap. 1: Mapa de memoria de la ZX Spectrum e información general

Todo lugar donde almacenamos un byte tiene una dirección. En el caso de la ZX Spectrum, se puede acceder a 64KBytes, apuntados por las direcciones 0 hasta 65535 (0000h hasta FFFFh) que abarcan todas las posiciones referenciables con dos bytes (16 bits, son 65536 valores posibles). Si bien hay computadoras basadas en el mismo Z80 que la Spectrum que acceden a un rango mayor de memoria, todas acceden a un máximo de 64KB a la vez. El bus de direcciones del Z80 es de 16 bits, y la forma de acceder a más memoria es mediante la división de memoria en páginas y selección del conjunto de páginas que se podrán acceder a la vez en un determinado momento.

En este capítulo veremos un cuadro donde se muestra la distribución de memoria en la ZX Spectrum. Para cada segmento descripto, tenemos una columna donde indicamos la dirección de inicio y otra donde indicamos la dirección de fin. Las direcciones de inicio y fin son valores fijos en algunos casos, y en otros casos están determinadas por variables del sistema. Cuando los valores son números fijos, no determinados por variables del sistema, incluimos los mismos en formato decimal y abajo el número equivalente en hexadecimal.

Hechas estas aclaraciones, vamos al punto. La memoria de la Spectrum es vista como un único bloque de 64KB que se divide así:

Tabla 1: La memoria de la ZX Spectrum

Desde	Hasta	Descripción
0 0000h	16383 3FFFh	Memoria ROM, de solo lectura. En total son 16384 bytes (16KBytes)
16384 4000h	22527 57FFh	Definición de pixels en pantalla. La resolución de pantalla es de 256x192 puntos. Los ocho bits de cada byte representan un segmento de ocho puntos definiendo cuál está encendido o no. Por lo tanto, se emplean (256x192/8)=6144 bytes (6KB).
22528 5800h	23295 5AFFh	Definición de atributos de color en pantalla; un byte por cuadro de 8x8 pixels. Para cada cuadro, el byte define si hay o no flash o bright, y 8 colores de fondo y de tinta. Son 768 bytes, con las definiciones sucesivas para los cuadros desde el superior a la izquierda hasta el último de la línea inferior.
23296 5B00h	23551 5BFFh	Buffer de impresora. Son 256 bytes que almacenan temporalmente lo que se imprimirá. Si no tenemos impresora (o nuestro programa no la empleará) podemos aprovechar estos 256 bytes para almacenar una pequeña rutina en código máquina
23552 5C00h	23733 5CB5h	Variables del sistema. Contienen varios elementos de información que indican al ordenador en qué estado se halla y son el objetivo de este documento. Algunas de ellas establecen los límites de las próximas divisiones que veremos en este mismo mapa de memoria
23734 5CB6h		Usada por la Interface-1, que incorpora una RS-232, conexión a microdrives y a red local. Si no está conectada la Interface-1, esta zona de memoria no existe
CHANS		Información de los canales. Contiene información sobre los dispositivos de entrada y salida, es decir, el teclado, pantalla, la impresora y otros.
PROG		Programa en Basic
VARS		Variables
E LINE		Comando o línea de programa que se está editando
WORKSP		Entrada de datos y espacio para tareas eventuales
STKBOT		Pila de cálculo, donde se guardan los datos que el calculador tiene
STKEND		Espacio de reserva
SP	RAMTOP-1	Pila de máquina y de GOSUB. El límite SP es el registro físico del Z80, no una variable del sistema
RAMTOP	UDG-1	Espacio de reserva. Comienza con el valor 62 (3Eh) y ocupa solamente un byte (con ese valor 62) al comienzo. Mediante la instrucción CLEAR bajamos el valor de RAMTOP y de esa manera quedan en reserva más bytes.
UDG	UDG+167	Gráficos definidos por el usuario; acá van los bytes que los arman.

Uso de los distintos espacios de memoria

Para comprender mejor la distribución de la memoria ROM en la ZX Spectrum, nada mejor que leer el libro "The Complete Spectrum ROM Disassembly", de Ian Logan y Frank O'Hara[8]. En esta oportunidad nos limitaremos a explicar que, básicamente, la ROM de la ZX Spectrum se divide en tres grandes secciones: rutinas de entrada y salida, intérprete de Basic y manejo de expresiones.

Los autores del libro mencionado encuentran una primera subdivisión de estas secciones en diez partes en total:

Rutinas de entrada y salida:

1. Ocho rutinas invocadas por las instrucciones RST, rutinas auxiliares y tablas de tokens[9] y códigos de teclado
2. Rutinas de manejo del teclado
3. Rutinas de manejo del parlante
4. Rutinas para manejo de cassette
5. Rutinas de pantalla e impresora

Intérprete de Basic:

6. Rutinas ejecutivas (inicio, bucle de ejecución Basic, control de sintaxis, etc.)
7. Interpretación de líneas, tanto del programa en Basic como un comando directo

Manejo de expresiones:

8. Evaluación de expresiones
9. Rutinas aritméticas
10. Calculador de punto flotante

En este trabajo, entonces, nos enfocaremos más a la memoria RAM, pudiéndose profundizar sobre la ROM leyendo el libro de Logan y O'Hara.

A diferencia de otros equipos, en la ZX Spectrum (y también en la TS 2068) no hay memoria para vídeo que se acceda en forma diferenciada a la memoria principal, sino que lo que vemos en pantalla se almacena en el mismo bloque de memoria RAM que los datos de sistema y el programa en Basic. A la vez, no hay memoria específica para el texto y otra destinada para los gráficos, sino que se emplea el mismo espacio en RAM para ambas cosas (en realidad, las rutinas en ROM para dibujar un caracter lo hacen en forma gráfica, como un cuadro de 8x8 pixeles). Se tiene primero la representación de los puntos, especificando simplemente cuál está encendido y cuál no. Luego vienen los atributos de color, brillo y flash, que se definen para un cuadro del tamaño de un carácter como una única combinación.

La definición de pixeles en pantalla comienza en la posición 16384 (4000h). Los ocho bits de cada byte representan un segmento de 8 pixeles definiendo cuál está encendido y cuál no. Así entonces, cada 32 bytes tenemos representada toda una línea horizontal de puntos. La pantalla se divide en tres bloques de 64 pixeles de alto (8 renglones de texto de 8 pixeles de altura) cada uno, ocupando 2048 bytes (2KBytes) cada bloque. Por lo tanto, se ocupa un total de 6 KBytes para la pantalla completa. Cada bloque se va completando de a una línea de pixeles por renglón de texto. Esto es: los primeros 32 bytes forman la línea 0 del renglón 0, los siguiente 32 bytes forman la línea 0 del siguiente renglón, y así hasta el renglón 7. Luego empiezan a definirse las líneas 1 (desde el renglón 0 hasta el 7), líneas 3, 4, 5, 6 y 7. En ese punto se terminó el primer bloque de 2KBytes, y los siguientes 32 bytes definirán la línea 0 del renglón 8 (comienzo del segundo bloque desde el renglón 8 hasta el 15).

[8] En http://www.worldofspectrum.org/documentation.html se puede encontrar la referencia a este y otros libros

[9] El código ASCII original utiliza 7 bits (numerados de 0 a 6) para codificar los diferentes caracteres. En la Spectrum, aparece expandido haciéndose de 8 bits. Si el bit 7 es 0 entonces el código corresponde a un carácter estándar ASCII, pero si el bit 7 toma el valor 1 corresponde a un TOKEN, codificados desde 128 en adelante. Los tokens son cada una de las palabras clave del Basic del Spectrum junto a los caracteres gráficos.

La definición de atributos de pantalla comienza en la posición 22528 (5800h). Cada byte contiene la información sobre los atributos de un cuadro de 8x8 puntos (espacio ocupado por un caracter), desde el primero a la izquierda de la fila superior hasta el último a la izquierda de la fila inferior. Los 8 bits de cada byte representan:

Bit 7: Flash (1: con parpadeo, 0: sin parpadeo)

Bit 6: Bright (1: con brillo, 0: sin brillo). Aplica tanto al fondo como a la tinta

Bits 5 a 3: Paper (valores 0 a 7). Color de los pixeles que están apagados en el cuadro

Bits 2 a 0: Ink (valores 0 a 7). Color de los pixeles que están encendidos en el cuadro

El siguiente es un listado que muestra la forma en que se van llenando los bytes que definen la pantalla:

```
10 FOR n=16384 TO 23295
20 POKE n,INT (RND*255)+1
30 NEXT n
40 PAUSE 0
```

Ejecutémoslo con RUN, y veremos el orden de llenado de los puntos y atributos.

El buffer de impresora es la siguiente sección de memoria. Comenzando en la dirección 23296 (5B00h), ocupa 256 bytes. Simplemente tiene la representación de un renglón de 8 líneas con 256 puntos en cada una, esperando ser impreso. Para cada línea utilizamos tenemos 32 bytes; y como cada byte tiene 8 bits entonces nos sirve para representar los 256 puntos de la línea, indicando con cada bit si el pixel está encendido o no. Por último, 8 líneas representadas por medio de 32 bytes cada una explican por qué el buffer de impresora ocupa 256 bytes en total.

Si no pensamos utilizar la impresora, podemos emplear esta zona de memoria para almacenar una pequeña rutina en código máquina.

Luego del buffer de impresora, desde la posición 23552 (5C00h) comienzan las variables del sistema. Nos ocuparemos de sus descripciones y uso en el capítulo 3.

La posición 23734 (5CB6h) es la de comienzo de una sección de memoria descripta como “mapas de microdrives”, aunque veremos que no solamente contiene estos mapas a los que el nombre se refiere. Es utilizada por la Interface 1, la cual incorpora un conector a microdrives, interfase RS-232 y acceso a red local. Por lo tanto, si no está conectado este dispositivo de expansión esta sección no existe y la variable de sistema CHANS (posición 23631), que indica el comienzo de la siguiente sección de memoria, contiene este mismo valor 23734.

En esta área de memoria se guardan más variables que el sistema utiliza para manejar los dispositivos incluidos en la Interface 1. Las mismas ocupan las posiciones 23734 hasta la 23791. A partir de la dirección 23792 comienzan los mapas de microdrives (lo cual explica el nombre dado a esta sección de la memoria). Cuando un microdrive se pone en uso, se crea para el mismo un mapa y un canal de flujo de datos. El canal de flujo para el microdrive va en la siguiente área de memoria, junto al resto de los canales definidos. El mapa del microdrive es simplemente un bloque de 256 bits (32 bytes) representando los 256 sectores en que se divide un cartucho, donde cada bit indica si está libre (vale 0) o si está ocupado o inusable (vale 1).

Para más información sobre la Interface 1 y los microdrives, con descripción de las variables del sistema y las rutinas incorporadas en la ROM de 8KB que incorpora esta Interface 1, podemos recomendar dos libros: el manual del Microdrive e Interface 1 de Sinclair, y “The Companion to the Sinclair ZX Microdrive and Interfaces” de Stuart Cooke (Pan Personal Computer News – Computer Library).

Apuntada por la variable del sistema CHANS comienza luego una sección de memoria que contiene información sobre los canales de flujo de datos. En la Spectrum, cada canal se define con una letra que lo identifica, y las direcciones de las rutinas que manejan la entrada y la salida de

datos por ese canal. Lo normal es que haya cuatro canales, y estos en particular siempre están presentes:

K: Los bytes vienen del teclado y salen para la parte inferior de la pantalla

S: Pantalla, salvo las dos líneas inferiores que salen por el canal K

R: Espacio de trabajo

P: Impresora

Para cada canal tenemos cinco bytes: dos bytes almacenando la dirección de la rutina que maneja la salida de datos, dos bytes almacenando la dirección de la rutina que maneja la entrada de datos, y un byte con la identificación (el código ASCII de la letra) del canal.

Puede haber otros canales aparte de estos cuatro: M (microdrive), B (interfase RS 232 en modo binario), T (interfase RS 232 en modo texto) o N (red local)[10]. De hecho, se admiten hasta 16 canales de datos, apuntados con los índices 0 a 15, de los cuales los índices 0 a 3 son los cuatro canales que siempre están: K, S, R, P, en ese orden. Desde Basic accedemos a los mismos indicando el número de canal para las instrucciones PRINT, INPUT y LIST. Por supuesto, también podemos utilizar las instrucciones OPEN y CLOSE para tener un mayor control. En el número 38 de la revista Microhobby Semanal, página 14, hay una completa nota sobre los canales de datos y la forma de acceder a los mismos[11].

Como se ve, el número de canales no es fijo y por eso es necesario que el sistema sepa cuál es el último definido. Por esta razón, si cuando el sistema va a leer un grupo de cinco bytes se encuentra con que el primero tiene el valor 80h (128) entonces interpreta que ya no hay más información de canales para leer.

El siguiente programa muestra la información presente en el área de información sobre los canales de flujo de datos:

```
  10 LET n=PEEK 23631+256*PEEK 2
3632
  20 REM While peek(n)<>128 most
rar datos
  30 IF PEEK n=128 THEN GO TO 10
0
  40 LET a=PEEK n: LET b=PEEK (n
+1): LET c=PEEK (n+2): LET d=PEE
K (n+3): LET e=PEEK (n+4)
  50 PRINT "Direccion ";n
  60 PRINT a;TAB 6;b;TAB 12;c;TA
B 18;d;TAB 24;e
  70 PRINT a+256*b;TAB 12;c+256*
d;TAB 24;CHR$ e
  80 LET n=n+5
  90>PRINT : GO TO 20
 100 PRINT "Fin de definicion de
 canales"
```

Lo que veremos al ejecutarlo es una pantalla donde mostraremos lo que se desprende de cada grupo de cinco bytes: la dirección donde está almacenado el primero de estos bytes, los cinco bytes propiamente dichos y debajo de éstos, los datos que se toman en cuenta: dirección de rutina de salida de datos (bytes 0 y 1), dirección de rutina de entrada de datos (bytes 2 y 3), y el nombre del canal propiamente dicho:

[10] La totalidad de estos canales mencionados son manejables mediante la Interface 1

[11] El contenido de la revista se puede leer desde http://www.microhobby.org/numero038.htm

```
Direccion 23734
244    9      168    16     75
2548          4264          K

Direccion 23739
244    9      196    21     83
2548          5572          S

Direccion 23744
129    15     196    21     82
3969          5572          R

Direccion 23749
244    9      196    21     80
2548          5572          P

Fin de definicion de canales




0 OK, 100:1
```

Luego de la información de canales de flujo de datos, viene el programa en Basic que esté almacenado en memoria en este momento. La variable del sistema PROG (dirección 23635) indica el lugar donde comienza el mismo. Exactamente apuntará al primer byte de la primera línea de código. La estructura en bytes de un programa es simple. Para cada línea de código, se tienen dos bytes con el número de línea (contrariamente al formato empleado normalmente, el byte más significativo esta vez va primero), luego otros dos bytes con la longitud del texto incluyendo un último byte de fin de línea, el texto y el fin de línea.

Por ejemplo, ejecutemos el programa:

```
  10 LET n=PEEK 23635+256*PEEK 2
3636
  20 POKE n,0
  30 POKE n+1,0
```

Vamos a ver que ahora para la primera línea tenemos el número 0. Para volver las cosas a la normalidad, basta con poner:

```
POKE n+1,10
```

Ahora cambiemos levemente el programa:

```
  10 LET n=PEEK 23635+256*PEEK 2
3636
  20 POKE n,0
  30 POKE n+1,100
```

Al ejecutarlo, notaremos que la primera línea ahora tiene el número 100. Lo interesante es que si ejecutamos el programa, el mismo seguirá funcionando bien. En circunstancias normales, RUN limpia todas las variables antes de comenzar, y si la línea 20 se ejecutara antes de todas (porque la 10 ahora es 100) daría un error de variable inexistente, lo que en este caso no ocurre.

Más aún, podemos poner una nueva línea 100. Para esto, probemos el programa:

```
  10 LET n=PEEK 23635+256*PEEK 2
3636
  20 POKE n,0
  30 POKE n+1,100
 100 POKE n+1,100
```

El siguiente código calcula el tamaño de las dos primeras líneas de un programa, y luego recorre la segunda de ellas:

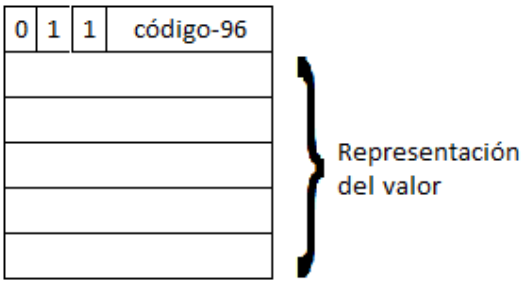
```
  10 LET n=PEEK 23635+256*PEEK 2
3636
  20 LET a=256*PEEK n+PEEK (n+1)
: REM numero de primera linea
  30 LET b=PEEK (n+2)+256*PEEK (
n+3): REM longitud de texto 1ra.
 linea
  40 LET m=n+4+b: REM Com. 2da.
linea = Com. 1ra. linea+2 bytes
de nro+2 bytes de longitud de te
xto + texto propiamente dicho
  50 LET c=256*PEEK m+PEEK (m+1)
: REM numero de segunda linea
  60 LET d=PEEK (m+2)+256*PEEK (
m+3): REM longitud de texto 2da.
 linea
  70 REM Ahora recorreremos la 2
da. linea
  80 FOR i=m+4 TO m+4+d-1
  90 PRINT i;": ";PEEK i,CHR$ PE
EK i
 100 NEXT i
```

Al ejecutarlo veremos, byte por byte, cómo está guardada esta segunda línea en memoria. Notemos que cuando aparece un número literal inmediatamente luego es seguido por el carácter 14 que significa “número” y cinco bytes más con la representación del mismo. Esto es porque los números siempre se representan en memoria con cinco bytes. En realidad, de dos maneras diferentes: una primera si son enteros entre -65535 y 65535 y otra diferente si no lo son. Vamos a comentar más sobre estas formas cuando veamos el área de variables, ya que la forma de representar estos valores en la zona de almacenamiento del programa en Basic es la misma que la de representar valores en la zona de variables; en este momento, para el caso del ejemplo nos limitaremos a señalar que los números son enteros y se representan como tales.

Luego del programa en Basic, la variable del sistema VARS (posición 23627) apunta al comienzo del área de definición de variables, la próxima zona de memoria.

En esta región, tenemos las variables definidas una detrás de la otra, sin separación alguna. En cada definición tenemos el nombre de la variable y el valor actual. Hay diferentes patrones para determinar tanto los nombres de las mismas como los tipos de datos, y están dados por los tres bits más significativos en el primer byte. Los nombres de variables numéricas pueden tener más de un carácter, pero los nombres de variables de tipo string solamente pueden tener una letra seguida del signo $. Si bien no se hace distinción entre mayúsculas y minúsculas, en la memoria los nombres de variables se guardan en minúscula.

Las variables numéricas cuyo nombre es una sola letra se almacenan de esta manera:



El primer byte indica en sus bits más significativos el tipo de variable (numérica con nombre de una sola letra), y en los cinco bits restantes el código ASCII del identificador al que se resta 96. Por ejemplo, para representar la variable numérica identificada con el caracter **a**, el código que se guarda en esos cinco bits será 00001.

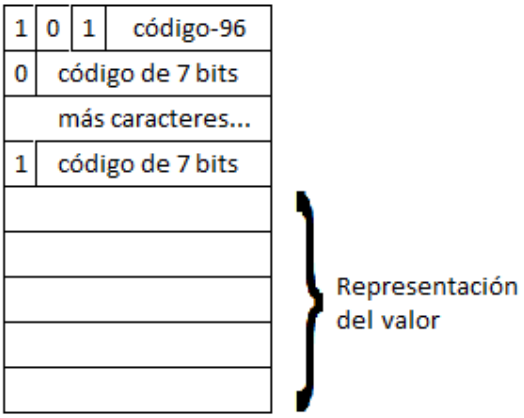
La representación del valor numérico varía si el número es un entero entre -65535 y 65535 o no. Si lo es, la forma de representación es: primer byte en 0, segundo byte en 0 si el número es positivo o 255 si es negativo, tercer y cuarto byte son la representación del número (que entonces puede ser 0 a 65535, o -65535 a 0 según el signo indicado en el segundo byte), y el quinto byte en

0. Si en cambio el número no es un entero, o siendo entero está fuera del rango de -65535 a 65535, se lo representa en coma flotante. También se utilizan cinco bytes, pero de esta manera: el primero de todos es el llamado exponente, mientras que los cuatro restantes forman lo que se denomina mantisa. El exponente es un valor entre 1 y 255 (el 0 se reserva para la representación de enteros). La mantisa es un valor entre 0,5 y 1. Es simple explicar el valor que se toma para el exponente, porque sencillamente se resta 128 al valor almacenado en el byte. Pero la mantisa es más compleja. Cada uno de los 32 bits (recordemos que son cuatro bytes) representa una potencia inversa de 2 que se suma o no para dar el valor final. El primero de los bits representa el valor 0,5 (1/2), el segundo es 0,25 (1/4), el tercero es 1/8, y así sucesivamente hasta $1/2^{32}$ que es el último bit. Con esto, la fórmula empleada para la representación en coma flotante de un número será la siguiente:

$N = 2^{(E-128)} \times M$, donde E es el exponente y M es la mantisa.

Por ejemplo: el valor 4,5 se representa con el exponente 131 y mantisa 0,5625. ¿Y los números negativos? Notemos que la mantisa es siempre mayor o igual que 0,5 y menor que 1. Esto implica que el primer bit siempre está en 1. Entonces se asume que siempre es así y se utiliza este primer bit para indicar si el número es positivo (se pone en 0) o negativo (se pone en 1).

En el caso de las variables numéricas cuyo nombre tiene más de un caracter, la representación será la siguiente:



O sea, cambian los tres bits más significativos del encabezado. Al código ASCII del primer caracter de la identificación se le sigue restando 96. Los siguientes caracteres se representarán con 7 bits, sin restarles ningún valor, y teniendo el bit más significativo en 0 hasta que sea el último, en cuyo momento el bit más significativo toma el valor de 1. Luego vendrá la representación del valor, ya sea en forma entera o coma flotante.

Como se puede observar, los tres primeros bits nos indican las características de la variable que se representará. Dejamos al lector el estudio de cómo son las demás representaciones, pero lo ayudamos con un cuadro que muestra las distintas posibilidades existentes:

Patrón	Descripción
0 1 0	Cadena (string), nombre de variable de un solo caracter seguida del signo “$”
0 1 1	Variable numérica identificada con un solo caracter
1 0 0	Matriz de números identificada con un solo caracter (no se puede más de uno)
1 0 1	Variable numérica identificada con más de un caracter
1 1 0	Matriz de cadenas de caracteres, nombre de variable de una sola letra
1 1 1	Iterador en bloque FOR/NEXT, identificado con un solo caracter (no se puede más de uno)

Vamos a examinar el área de variables con este programa:

```
  10 LET comienzo=PEEK 23627+256
*PEEK 23628
  20 LET a=4.5: REM Variable en
punto flotante
  30 FOR n=comienzo TO comienzo+
30
  40 PRINT "Valor en posicion ";
n;":";PEEK n
  50 NEXT n
```

Al ejecutarlo, obtenemos la salida que analizaremos:

```
Valor en posicion 23926:163
Valor en posicion 23927:111
Valor en posicion 23928:109
Valor en posicion 23929:105
Valor en posicion 23930:101
Valor en posicion 23931:110
Valor en posicion 23932:122
Valor en posicion 23933:239
Valor en posicion 23934:0
Valor en posicion 23935:0
Valor en posicion 23936:118
Valor en posicion 23937:93
Valor en posicion 23938:0
Valor en posicion 23939:97
Valor en posicion 23940:131
Valor en posicion 23941:16
Valor en posicion 23942:0
Valor en posicion 23943:0
Valor en posicion 23944:0
Valor en posicion 23945:238
Valor en posicion 23946:0
Valor en posicion 23947:0
Valor en posicion 23948:140
Valor en posicion 23949:93
Valor en posicion 23950:0
Valor en posicion 23951:0
Valor en posicion 23952:0
Valor en posicion 23953:148
Valor en posicion 23954:93
Valor en posicion 23955:0
Valor en posicion 23956:0
```

La representación en binario del primer byte que encontramos es 10100011. Los tres primeros bits, 101, nos dicen que se trata de una variable numérica cuyo identificador tiene más de una letra. La primera letra la encontramos con los cinco bits restantes: 00011, cuyo valor decimal es 3. Sumándole 96, llegamos al valor 99 que es el código ASCII del caracter **c**. Los siguientes caracteres forman la secuencia “omienzo” (notemos que en la última letra el bit 7 vale 1). Ya sabemos entonces que el nombre es “comienzo”. Desde la posición 23934 hasta la 23938 encontramos la representación del valor que toma. Como es un valor entero, el primer byte es 0. Como es positivo, el segundo byte es también 0. Los siguientes dos bytes dan el valor: 118+256x93=23296. El último byte es 0, como está establecido para los números representados como valores enteros.

Inmediatamente luego, en la posición 23939, encontramos la definición de la siguiente variable. El valor 97 se representa en binario así: 01100001. Los primeros tres bits (011) nos dicen que estamos ante una variable numérica representada con un solo caracter. Los últimos cinco bits nos dan el valor 1, al cual le sumamos 96 y llegamos a 97, código ASCII del caracter **a**. Notemos que esta clase de variables tiene la característica de que el valor en el primer byte se corresponde exactamente al código ASCII del caracter identificatorio, pero esto es solamente porque los bits 5 y 6 (valores 32 y 64 respectivamente) dan la casualidad de que están en 1, mientras que el bit 7 (valor 128) está en 0. El valor 131 en el siguiente byte, distinto de 0, indica que el número se representará en coma flotante y que el exponente es 3 (131-128). La representación de la mantisa son 32 bits de la forma 00010000-00000000-00000000-00000000. Recordemos que la mantisa siempre es mayor o igual que ½ (0,5); el primer bit entonces toma el valor 1 siempre. ¿Por qué está en 0? Esto es por lo antes explicado: dado que **siempre** el bit tiene este valor 1, se asume que es así y el bit se emplea para indicar el signo del número. En este caso, el número es positivo. Y la mantisa es 1001 seguida de 28 bits en cero. El valor calculado es: 1/2 + 1/16 = 0,5625.

En la posición 23945 comienza una nueva definición de variable. En este caso, el primer byte tiene el valor 238, representado en binario como 11101110. Los primeros tres bits (111) nos dicen que estamos ante un iterador. El nombre de la variable será “n”, dado que los cinco bits restantes representan el valor 14, al cual le sumamos 96 y nos da 110, código ASCII de la letra **n**. En el caso de los iteradores, luego del nombre vendrán diferentes campos, que este programa deja

sin mostrar (habría que incrementar el valor del índice final en la línea 30). Los campos que siguen al primer byte son: valor de comienzo (5 bytes), valor final (5 bytes), paso (STEP, 5 bytes), número de línea donde saltar al terminar el bucle (2 bytes) y número de sentencia dentro de la línea (1 byte).

Resta decir que el fin de la zona de variables se señala con el byte 80h (128d). Notemos que 128 es un valor de byte imposible de lograr en el primer lugar de la representación de una variable, dado que el primer byte es el que indica el tipo de variable y la primera letra del nombre que toma; y siempre los últimos cinco bits, que sirven para determinar el nombre que se le dio a la variable, tienen un valor distinto de 0. Entonces, si el sistema operativo está recorriendo esta zona de memoria y quiere leer los datos de la siguiente variable, al encontrar este primer valor en 80h se da cuenta de que se llegó al final de la misma y que ya no quedan más variables.

Luego de la definición de variables, el próximo bloque de memoria es el área de edición, tanto de una línea de programa como el momento de ingreso de un comando inmediato. Si estamos examinando esta área de memoria entonces en ese momento puntual de ejecución no estamos haciendo edición, así que siempre veremos el área de memoria vacía; solamente el sistema operativo la utiliza.

Después del área de edición, encontramos el área de entrada de datos (INPUT). Del mismo modo que en el caso anterior, no podremos ver el contenido porque si estamos haciendo eso no estamos haciendo un ingreso de datos.

A continuación llegamos a la pila de cálculo. Ahí guardamos información sobre números y cadenas de caracteres. Las operaciones aritméticas y otras hacen uso de esta pila de cálculo.

Apuntada por la variable de sistema STKEND comienza un área de memoria libre, que se irá utilizando por la pila de cálculo al crecer o por la pila de máquina. Es un espacio de reserva, y por ese motivo no conviene guardar ahí ningún valor importante porque corre el riesgo de perderse.

La siguiente área de memoria es el reservado para la pila de máquina y de GOSUB. No es apuntado por una variable de sistema sino por el registro SP del propio Z80. Cada vez que hacemos operaciones PUSH y POP vamos poniendo o sacando de la pila respectivamente los valores de los registros en el microprocesador. De esta forma implementamos la pila de GOSUB. Vamos guardando información sobre el punto exacto donde debemos saltar cuando encontremos una instrucción RETURN en nuestro programa Basic. Cuando el intérprete Basic encuentra una instrucción GOSUB guarda el número de línea e instrucción dentro de la misma donde deberá saltar luego de terminar la ejecución de la subrutina. Se guardan en una pila (LIFO: last in, first out), y así es como podemos ejecutar GOSUB dentro de subrutinas (el límite de invocaciones anidadas es la memoria) y aseguramos que ante cada RETURN volveremos al lugar exacto siguiente a la instrucción GOSUB correspondiente.

Luego de la pila de GOSUB viene un espacio de reserva de memoria, que no se borra ante una instrucción NEW, y normalmente se utiliza para guardar un programa en código máquina o datos que queremos preservar. Mediante la instrucción CLEAR le damos un valor a la variable del sistema RAMTOP (dos bytes desde la dirección 23730) y en esa posición se guarda el valor 62 (3Eh) indicando el comienzo de esta zona. Por ejemplo, escribamos el programa:

```
  10 LET a=PEEK 23730+256*PEEK 2
3731
  20 PRINT a
  30 PRINT PEEK a
```

Obtendremos la salida:

```
65367
62
```

El valor 65367 es el original que se carga en la variable de sistema RAMTOP cuando la computadora arranca. En esa dirección vamos a ver, por supuesto, el valor 62 referido anteriormente. Ahora vamos a cambiar el valor de la RAMTOP:

```
CLEAR 64000
```

Reejecutamos el programa y obtendremos una salida diferente:

```
64000
62
```

Notemos que el contenido de la variable RAMTOP cambió, tomando el valor que le dimos mediante la instrucción CLEAR. Veamos además que en la nueva posición apuntada por esta variable del sistema se guardó el mismo valor 62 que antes estaba en la posición 65367.

Si ejecutamos una instrucción NEW, y volvemos a cargar el mismo programa (lamentablemente se habrá borrado), obtendremos la misma salida que la última vez:

```
64000
62
```

Si ejecutamos en cambio la instrucción RANDOMIZE USR 0, la computadora se reinicia y luego de cargar el mismo programa (otra vez se borró) obtendremos la salida original:

```
65367
62
```

La última sección de memoria es la que guarda información sobre los gráficos definidos por el usuario (GDU en castellano, o UDG en inglés por “User Defined Graphics”). Son 168 bytes que definen 21 gráficos de 8 bytes cada uno. Cada byte es la representación de una línea de 8 puntos, donde cada bit en 1 o 0 indica si el punto está encendido o apagado. Por lo tanto, cada gráfico en realidad ocupa el mismo espacio que un carácter: un cuadrado de 8x8 pixeles. Los gráficos se identifican con las letras A hasta la U y se codifican como los caracteres 144 hasta 164.

Vale la pena notar que estos gráficos no se inician como cuadrados en blanco, sino que arrancan con la misma definición de los caracteres en mayúscula A hasta U. Esto es, antes de alterar alguno de estos gráficos, podemos ejecutar el programa:

```
10 FOR n=144 TO 164
20 PRINT n,CHR$ n
30 NEXT n
```

Y obtendremos la salida:

```
144               A
145               B
146               C
147               D
148               E
149               F
150               G
151               H
152               I
153               J
154               K
155               L
156               M
157               N
158               O
159               P
160               Q
161               R
162               S
163               T
164               U

0 OK, 30:1
```

Este bloque de memoria comienza en la dirección apuntada por la variable del sistema UDG (dos bytes desde la dirección 23675), y no nos debería sorprender que el valor inicial de esta variable sea 65368 (cuando tenemos 48KB de RAM, con 16KB será 32600), una posición más que la original de RAMTOP. Notemos que si a partir de esa posición tenemos 168 bytes, el último viene a estar en la posición 65535 (o 32767), última posición en toda la memoria física. Otra cosa que no nos debería sorprender es que como este bloque ocupa un lugar por sobre la RAMTOP, tampoco se borrará su contenido si ejecutamos un NEW. Si en cambio ejecutamos RANDOMIZE USR 0 se restablecerán los valores originales.

Existe la posibilidad de existencia de un “bloque extra de memoria”. Notemos que el mapa de memoria nos dice que el bloque donde se definen los GDU comienza en la dirección apuntada por la variable del sistema UDG, y termina en UDG+167. Este último valor coincide inicialmente con la variable del sistema P-RAMT (última posición de memoria física) pero varía si alteramos el valor de UDG. En este caso, si damos un valor menor a UDG quedaría libre un bloque desde UDG+168 hasta P-RAMT. Un ejemplo de utilidad puede ser el guardar más definiciones de GDU, accediendo a las mismas volviendo a modificar la variable del sistema UDG.

Cap. 2: Uso de la memoria en la TS 2068

Los conceptos que hemos visto en el capítulo anterior son aplicables en general también a la TS 2068. Este capítulo simplemente mostrará cosas específicas de la TS 2068.

No hemos encontrado un esquema de la organización de la ROM de la TS 2068 para comentarla como hicimos en el capítulo 1 referido a la ZX Spectrum. Pero dado que se trata de dos computadoras similares, una derivada del diseño de la otra, suponemos que encontraremos una disposición de secciones similar.

En el caso de la TS 2068 se accede a un rango mayor de memoria mediante el paginado[12]. En teoría se pueden tener hasta 16 MB de memoria gracias a que disponemos de 256 bancos de 64KB cada uno[13], pero como se desprende de los siguientes párrafos este es un límite teórico al que nunca llegaremos.

Como decíamos, en la TS 2068 la memoria se divide en bancos que se pueden intercambiar (superando así el límite de 64 KB aunque, como se dijo antes, nunca veremos más que 64KB a la vez porque el bus de direcciones del Z80 es de 16 bits). Se definen 256 bancos de 64KB cada uno (numerados del 0 al 255), y cada uno de ellos se divide en 8 bloques de 8KB cada uno (numerados del 0 al 7). Siempre se accede a 8 bloques a la vez, no necesariamente del mismo banco. Pero no se puede acceder a dos bloques en la misma posición de distintos bancos: siempre habrá un único bloque 0, un único bloque 1, y así hasta el único bloque 7.

El banco 255 es llamado "Home Bank", y es el que se selecciona por defecto al arrancar la computadora y contiene 16KB de memoria ROM (ocupando los dos primeros bloques de 8KB) y los 48KB de memoria RAM (los seis bloques del 2 al 7). El banco 254 es llamado "Extension ROM Bank" y en el bloque 0 del mismo está la ROM de extensión, que con sus 8KB hacen llegar el total de memoria ROM a 24KB. El banco 0, llamado "Dock Bank", es ocupado cuando ponemos un cartucho, que eventualmente puede tener los 64KB completos. Los 253 bancos restantes están libres y pueden ser ocupados por periféricos que se conecten a la TS 2068[14].

Otros factores que obligan al cambio de distribución de memoria son que se tienen más variables de sistema y la posibilidad de contar con otros modos de video.

Así entonces, la distribución inicial de la memoria (y que normalmente no cambia salvo que se pongan en uso las características distintivas de esta computadora respecto a la Spectrum) es la siguiente:

Tabla 2: la memoria de la TS 2068

Desde	Hasta	Descripción
0 0000h	16383 3FFFh	Memoria ROM, de solo lectura. En realidad son 24576 bytes (24KBytes), pero divididos en tres bloques de 8KB cada uno, dos en el banco 255 y otro ocupando la posición 0 del banco 254 y reemplazando eventualmente al bloque 0 del banco 255.
16384 4000h	22527 57FFh	Definición de pixeles en pantalla; un byte por segmento de 8 pixeles definiendo cuál está encendido o no. En total son 6144 bytes (6KBytes). En el modo de 64 columnas (resolución de 512x192) se crea una segunda área de memoria, también de 6KB, que complementa esta área. Esta segunda área se ubica desde la posición 24576.
22528 5800h	23295 5AFFh	Definición de atributos de color en pantalla; un byte por cuadro de 8x8 pixeles. Para cada cuadro, el byte define si hay o no flash o bright, y 8 colores de fondo y de tinta. Son 768 bytes.
23296 5B00h	23551 5BFFh	Buffer de impresora. Son 256 bytes que almacenan temporalmente lo que se imprimirá.

[12] Los modelos de 128KB (Spectrum 128K, +2, +2A y +3) también emplean el paginado para acceder a más memoria. Para ver la forma en que lo hacen, podemos leer el artículo en http://www.speccy.org/magazinezx/17/128k-mode.html

[13] Fred Blechman: TS 2068 Begginer/Intermediate Guide (Sams, 1983) Capítulo 1

[14] De todos modos, Wes Brzozowski escribió un excelente artículo llamado "El misterio de los 253 perdidos" donde comenta la forma en que se pensaba trabajar con los bancos de memoria, y que finalmente no se llevó a cabo. En http://8bit.yarek.pl/interface/ts.beu/wes-1.html se puede acceder al texto completo.

Desde	Hasta	Descripción
23552 5C00h	23733 5CB5h	Variables del sistema, coincidentes con la totalidad de las utilizadas en la ZX Spectrum.
23734 5CB6h	23755 5CCBh	Variables del sistema, de uso específico por la TS 2068
23756 5CCCh	24297 5EE9h	Espacio reservado para variables del sistema, sin uso
24298 5EEAh	24575 5FFFh	Tabla de configuración de bancos de memoria. Los primeros 12 bytes son usados regularmente, y luego se tienen 11 grupos de 24 bytes cada uno que los utilizarán los bancos de expansión; finalmente hay un marcador de fin de tabla.
24576 6000h	25087 61FFh	Pila de máquina, son 512 bytes
25088 6200h	25206 6276h	Espacio reservado para ser usado por el distribuidor de funciones (function dispatcher), una rutina en ROM que facilita el uso de diferentes funciones: BEEP, COPY (pantalla a impresora), NEW, borrar buffer de impresora, abrir y cerrar canales, CLEAR, otras instrucciones BASIC, etc.
25207 6277h	26667 682Bh	Código para selección de bancos
26668 682Ch		Variables de código máquina
CHANS		Información de los canales
PROG		Programa en Basic
VARS		Definición de variables en Basic
E LINE		Comando o línea de programa que se está editando
WORKSP		Entrada de datos y espacio para tareas eventuales
STKBOT		Pila de cálculo, donde se guardan los datos que el calculador tiene
STKEND	RAMTOP-1	Espacio de reserva para la pila de cálculo
RAMTOP	UDG-1	Espacio de reserva, no se borra ante una instrucción NEW. Podemos determinar su tamaño mediante la instrucción CLEAR
UDG	UDG+167	Gráficos definidos por el usuario

Vamos a detenernos, para explicar mejor el tema de los bancos de memoria, en la organización de la tabla de configuración de bancos que ocupa los 278 bytes entre las direcciones 24298 y 24575. En esta tabla tendremos información sobre la memoria extra que podemos agregar a la TS 2068. La disposición que tenemos es la siguiente:
Primeros 8 bytes: información sobre AROS (Application ROM-Oriented Software)
Siguientes 4 bytes: información sobre LROS (Language ROM-Oriented Software)
Estos son los utilizados regularmente, ya que describen las dos clases de programas que podemos tener en los cartuchos que se insertan en el conector frontal[15]. A continuación, siguen 11 bloques de 24 bytes cada uno, que permiten declarar hasta 11 bancos de memoria. Si se tienen más bancos, se debe cambiar el valor de la variable del sistema SYSCON (dirección 23740) para apuntar a una zona de memoria más amplia donde se almacenen todas las descripciones juntas.
Por último, un byte con el valor 128 (80H) señala el final de la tabla, y queda un byte sin utilizar.

[15] La diferencia entre estos es que un cartucho LROS tiene software en código máquina que toma control inmediato de la computadora desde el reset inicial, mientras que el cartucho AROS depende de la ROM del sistema

Cap. 3 - Vista a vuelo de pájaro: las variables del sistema

Los octetos de la memoria cuyas direcciones están comprendidas entre 23552 y 23733 se han reservado para usos específicos del sistema. Es posible examinar los valores almacenados (mediante la función **PEEK**) para averiguar detalles acerca de la configuración actual del mismo y algunos de ellos también se pueden alterar (con la instrucción **POKE**). Estas variables del sistema ocupan en su mayoría uno o dos octetos, y las listamos aquí junto con sus usos:

Tabla 3: Variables del Sistema en la ZX Spectrum

Bytes	Dirección	Nombre	Contenido
8	23552	KSTATE	Se usa para leer del teclado.
1	23560	LAST K	Almacena el valor de la tecla pulsada últimamente.
1	23561	REPDEL	Tiempo (en 1 / 50 de segundo - o en 1 / 60 en Norteamérica) que debe presionarse la tecla para que se repita. Su valor parte de 35, pero es posible alterarlo (POKE) con otros valores.
1	23562	REPPER	Retardo (en 1 / 50 de segundo - o en 1 / 60 de segundo en Norteamérica) entre repeticiones sucesivas de una tecla mantenida oprimida: inicialmente vale 5.
2	23563	DEFADD	Dirección del argumento de una función definida por el usuario, si es que se está evaluando alguna; en otro caso vale 0.
1	23565	K DATA	Almacena información de color en la línea que estamos editando.
2	23566	TVDATA	Almacena octetos de color, controles AT y TAB que van a la televisión.
38	23568	STRMS	Direcciones de canales de comunicación de entrada y salida.
2	23606	CHARS	Dirección del conjunto de caracteres menos 256 (que comienza por un espacio y lleva el signo de copyright). Se encuentra normalmente en la ROM, pero es posible disponer una RAM para hacer que la función CHARS se dirija a la misma.
1	23608	RASP	Duración del zumbador de alarma.
1	23609	PIP	Duración del chasquido del teclado.
1	23610	ERR NR	El código de informe menos 1. Se inicializa a 255 (para -1) por lo que si se examina la posición 23610 (PEEK 23610) se obtiene 255.
1	23611	FLAGS	Varios indicadores para control del sistema BASIC.
1	23612	TV FLAG	Indicadores asociados con la televisión.
2	23613	ERR SP	Dirección del elemento de la pila de la máquina que es usado como retorno de error.
2	23615	LIST SP	Dirección de la posición de retorno tras un listado automático.
1	23617	MODE	Especifica el cursor K, L, C, E o G.
2	23618	NEWPPC	Línea a la que hay que saltar.
1	23620	NSPPC	Número de sentencia en una línea a la que hay que saltar. Si se ejerce la función POKE primeramente a NEWPPC, y luego a NSPPC, se fuerza un salto a una sentencia especificada en una línea.
2	23621	PPC	Número de línea de la sentencia en curso.
1	23623	SUBPPC	Número dentro de la línea de la sentencia en curso.
1	23624	BORDCR	Color del contorno *8; contiene también los atributos que se suelen usar para la mitad inferior de la pantalla.
2	23625	E PPC	Número de la línea en curso (con el cursor del programa).
2	23627	VARS	Dirección de las variables.
2	23629	DEST	Dirección de la última variable asignada.
2	23631	CHANS	Dirección de la definición de los canales de datos.
2	23633	CURCHL	Dirección que se destina en ese momento para entrada y salida de la información por un canal.
2	23635	PROG	Dirección del programa BASIC.
2	23637	NXTLIN	Dirección de la siguiente línea del programa.

Bytes	Dirección	Nombre	Contenido
2	23639	DATADD	Dirección de la terminación del último elemento de datos.
2	23641	E LINE	Dirección del comando que se está escribiendo en el teclado.
2	23643	K CUR	Dirección del cursor.
2	23645	CH ADD	Dirección del siguiente carácter a interpretar, durante la ejecución de un programa en Basic.
2	23647	X PTR	Dirección del carácter que sigue al signo ? .
2	23649	WORKSP	Dirección del espacio eventual de trabajo.
2	23651	STKBOT	Dirección del fondo de la pila de cálculo.
2	23653	STKEND	Dirección del comienzo del espacio de reserva.
1	23655	BREG	Registro b del calculador.
2	23656	MEM	Dirección de la zona utilizada para la memoria del calculador. (Normalmente MEMBOT, pero no siempre).
1	23658	FLAGS2	Más indicadores.
1	23659	DF SZ	Número de líneas (incluida una en blanco) de la mitad inferior de la pantalla.
2	23660	S TOP	El número de la línea superior del programa en los listados automáticos.
2	23662	OLDPPC	Número de línea a la que salta la sentencia CONTINUE.
1	23664	OSPCC	Número dentro de la línea de la sentencia a la que salta la instrucción CONTINUE.
1	23665	FLAGX	Indicadores varios utilizados para la instrucción INPUT.
2	23666	STRLEN	Longitud de la variable tipo de cadena que se esté utilizando.
2	23668	T ADDR	Dirección del siguiente elemento en la tabla de sintaxis (de poca utilidad en un programa de usuario).
2	23670	SEED	El origen para RND, es la variable que se fija mediante la función RANDOMIZE.
3	23672	FRAMES	3 octetos (el menos significativo en primer lugar) del contador de cuadros. Se incrementa cada 20 ms.
2	23675	UDG	Dirección del primer gráfico definido por el usuario. Se la puede cambiar, por ejemplo, para ahorrar espacio a costa de tener menos gráficos definidos por el usuario.
2	23677	COORDS	Coordenadas (x,y) del último punto trazado. Son dos variables; la primera es la coordenada x, la segunda es la coordenada y.
1	23679	P POSN	Número de 33 columnas de la posición de la impresora.
1	23680	PR CC	Octeto menos significativo correspondiente a la dirección de la siguiente posición a imprimir en la función PRINT (en la memoria temporal de la impresora).
1	23681		No se usa.
2	23682	ECHO E	Número de las 33 columnas y número de las 24 líneas (en la mitad inferior) del final de la memoria temporal de entrada.
2	23684	DF CC	Dirección de la posición de PRINT en el fichero de la presentación visual.
2	23686	DFCCL	Igual que DFCC, pero para la mitad inferior de la pantalla.
1	23688	S POSN	Número de las 33 columnas de la posición de PRINT.
1	23689		Número de las 24 líneas de la posición de PRINT.
2	23690	SPOSNL	Igual que S POSN, pero para la mitad inferior.
1	23692	SCR CT	Contaje de los desplazamientos hacia arriba ("scroll"): es siempre una unidad más que el número de desplazamientos que se van a hacer antes de terminar en un scroll. Si se cambia este valor con un número mayor que 1 (digamos 255), la pantalla continuará "enrollándose" sin necesidad de nuevas instrucciones.
1	23693	ATTR P	Atributos permanentes en curso, tal y como los fijaron las sentencias PAPER, INK, BRIGHT y FLASH.

Bytes	Dirección	Nombre	Contenido
1	23694	MASK P	Se usa para los colores transparentes. Cualquier bit que sea 1 indica que el bit correspondiente de los atributos no se toma de ATTR P, sino de lo que ya está en pantalla.
1	23695	ATTR T	Atributos de uso temporal: fondo, letra, flash y brillo.
1	23696	MASK T	Igual que MASK P, pero temporal.
1	23697	P FLAG	Más indicadores para la salida por pantalla
30	23698	MEMBOT	Zona de memoria destinada al calculador; se usa para almacenar números que no pueden ser guardados convenientemente en la pila del calculador.
2	23728	NMIADD	Dirección de la rutina de atención de una interrupción no enmascarable.
2	23730	RAMTOP	Dirección del último octeto del BASIC en la zona del sistema.
2	23732	P-RAMT	Dirección del último octeto de la RAM física.

La TS 2068, por su parte, utiliza además las siguientes variables:

Tabla 4: Variables del sistema específicas en la TS 2068

Bytes	Dirección	Nombre	Contenido
2	23734	ERRLN	Número de línea a donde saltar en caso de error de programa Basic
2	23736	ERRC	Número de línea donde ocurrió un error de programa Basic
1	23738	ERRS	Número de sentencia que no funcionó bien dentro de la línea donde ocurrió el error de programa Basic
1	23739	ERRT	Código de error
2	23740	SYSCON	Puntero a la Tabla de Configuración de Sistema
1	23742	MAXBNK	Cantidad de bancos de expansión en el sistema
1	23743	CURCBN	Número de banco actual
2	23744	MSTBOT	Ubicación del fin de la pila de máquina
1	23746	VIDMOD	Modo de video
1	23747		No se usa
7	23748		Variables reservadas para el uso de cartuchos con programas
1	23755	STRMNM	Número de canal de flujo de datos (stream) en uso

Como este apunte no pretende extenderse hacia otros equipos que la ZX Spectrum o la TS 2068, sugerimos buscar información sobre las variables del sistema en la ZX Spectrum 128. Notemos que la TS 2068 ocupa la zona de memoria inmediatamente posterior a la zona original de definición de variables en la ZX Spectrum para continuar con la definición de las variables que utiliza en forma exclusiva. En cambio, el modelo de 128K ocupa con el mismo fin la memoria inmediatamente anterior a la primera variable, donde originalmente se ubicaba el buffer de impresora[16]. Esto provocó graves problemas con algunos programas para Spectrum que alojaban rutinas en esta parte de la memoria, provocando que la computadora se colgara o reiniciara. Por supuesto, que solamente cuando la computadora trabaja en modo 128KB; en el modo de 48KB se sigue manteniendo la compatibilidad.

[16] Probablemente, porque la memoria inmediatamente posterior podría ser ocupada por las variables empleadas por la Interface 1

Cap. 4 - Vista en detalle: Las variables del Sistema

Vamos a analizar, una por una, estas variables que hemos descripto en el capítulo anterior y ver ejemplos de uso. Vamos a utilizar la notación siguiente:

(X)	significa el valor contenido en la posición X. Sería como PEEK(X)
(X) →a	significa que el valor en la posición X lo guardamos en una variable a. Sería como a=PEEK(X)
a → (X)	significa que en la posición X guardamos el valor (o una variable) a. Sería como POKE X,a
(VAR+d)	significa el valor contenido en la d-ésima posición de la variable VAR, contando desde 0. Por ejemplo, (KSTATE+4) es la posición guardada en la quinta posición de la variable KSTATE. Si KSTATE=23552, entonces es el valor en la dirección 23556.
(X) = n	significa que el valor contenido en la posición X pasa a valer n
[xxx]	descripción de un valor

Por otro lado, no está de más recordar la forma de guardar valores de variables que ocupen más de un byte. Siempre el byte menos significativo va primero, y luego el más significativo. De modo que la forma de calcular el valor es:

(X)+256 x (X+1) →valor

Esto se debe a esa es, precisamente, la forma en que el Z80 toma los valores de dos bytes: el menos significativo en la primera posición, seguido por el más significativo. Es interesante saber que el diseñador principal del Z80[17] trabajaba en Intel y estuvo involucrado en el proyecto del procesador Intel 8080, que toma los valores numéricos de la misma forma. Los procesadores actuales de Intel siguen trabajando de la misma manera. En cambio, los procesadores de Motorola (como el 6809 o el 68000) toman los valores en modo inverso: el byte más significativo primero.

KSTATE (23552, 8 bytes, valor inicial no determinado)

Contiene información sobre el estado en que se encuentra el teclado. Cada vez que el Z80 recibe una interrupción (esto ocurre normalmente 50 veces por segundo en la Spectrum y 60 veces por segundo en la 2068), una de las cosas que hace la computadora es leer el teclado y guardar el resultado en estos ocho bytes. Estos bytes tienen diferentes usos, y no todos son útiles para un programador. Para comprender su funcionamiento, podemos considerar que tenemos dos bloques: el superior (4 bytes desde 23552 hasta 23555) y el inferior (4 bytes desde 23556 hasta 23559), y normalmente se usa el inferior. El superior es utilizado cuando, mientras se tiene presionada una tecla, otra nueva es presionada sin soltar la primera. La lógica es la siguiente:

Si no hay tecla presionada

(KSTATE+4) = 255

(KSTATE+5) = 0

Si la hay

(KSTATE+4) = [valor de la tecla pero en mayúsculas], por ejemplo 65 si se presionó la A.

(KSTATE+5) = 5

(KSTATE+6) = cuenta atrás, variando con el tiempo, para repetir automáticamente la tecla

(KSTATE+7) = [código ASCII de la tecla pulsada]

Si hay una única tecla presionada o ninguna

(KSTATE+0) = 255

(KSTATE+1) = 0

Si hay más de una tecla presionada, o se presiona una segunda mientras había una presionada

(KSTATE+0) = [valor de la tecla pero en mayúsculas], por ejemplo 65 si se presionó la A.

(KSTATE+1) = 5

(KSTATE+2) = cuenta atrás, variando con el tiempo, para repetir automáticamente la tecla

(KSTATE+3) = [código ASCII de la tecla pulsada, teniendo en cuenta SHIFT o SYMBOL]

[17] Federico Faggin, ingresó a Intel en 1970 y en 1974 renunció a la empresa para fundar Zilog junto con Ralph Ungermann, otro ex empleado de Intel. En Julio de 1976 comenzó la venta del Z-80.

Lo vamos a ver con un programa:

```
   10 FOR n=0 TO 7
   20 PRINT "Pos. ";n+23552;": ";
PEEK (n+23552)
   30 NEXT n
   40 PAUSE 20
   50 CLS
   60 GO TO 10
```

Ejecutémoslo, y presionemos diferentes teclas para comprobar los valores que va tomando. Por ejemplo, notemos que el valor de la tecla pulsada queda guardado aunque se la suelte, hasta que pulsemos una nueva.
Un ejemplo de utilidad es para obtener siempre el carácter presionado en mayúscula: el código ASCII de la letra en mayúscula estará en la posición 23556.

LAST K (23560, 1 byte, valor inicial 255)
Acá se almacena el valor de la última tecla que presionamos, no importando si ahora no tenemos ninguna tecla apretada. Contiene el valor ASCII de la tecla presionada. Para comprender la diferencia entre esta variable y el valor en KSTATE+7, podemos cambiar la línea 10 del listado anterior:

```
10 FOR n=0 TO 8
```

Y lo volvemos a ejecutar. Presionemos la letra N, mientras está apretada presionemos la M y luego soltemos la N sin dejar de apretar la M.

REPDEL (23561, 1 byte, valor inicial 35)
Esta variable indica el tiempo en cincuentavos (sesentavos en la TS 2068) de segundo que debemos mantener pulsada una tecla para que la misma se empiece a repetir. Si le ponemos 0 como valor, esto equivale a ponerle 256. Si le ponemos 1, dificultamos mucho el tipeo, porque cuando alguien presione una tecla, antes de soltarla ya se habrá empezado a repetir.

REPPER (23562, 1 byte, valor inicial 5)
Esta variable es de utilidad parecida a la anterior. En este caso, el objeto de la misma es especificar el tiempo en cincuentavos de segundo que transcurrirá entre las repeticiones de una tecla que se mantiene presionada. Si le ponemos el valor 1, y combinamos esto con ponerle de valor 1 a REPDEL, será casi imposible lograr ingresar una línea de texto o programa en forma correcta.

DEFADD (23563, 2 bytes, valor inicial 0)
Acá se guarda la dirección de la primera letra de argumento de una función definida por el usuario, si es que se está evaluando alguna y solamente si estamos en el momento de la evaluación. Nos estamos refiriendo a la dirección dentro de la memoria del carácter que indica el argumento. Si la función no tiene argumentos, la dirección apunta al byte donde está almacenado el paréntesis de cierre en el código del programa Basic. Una utilidad es que tenemos con esto un método de pasar parámetros a una función desde código máquina dado que cargamos en IX el contenido de esta variable y entonces IX pasa a apuntar a los parámetros de la función.
Vamos a ver un programa que muestra el comportamiento de esta función:

```
   10 DEF FN f(x)=PEEK 23563+256*
PEEK 23564
   20 DEF FN g()=PEEK 23563+256*P
EEK 23564
   30 LET a=FN f(1)
   40 PRINT a
   50 PRINT PEEK a;" ";CHR$ PEEK
a
   60 LET b=FN g()
   70 PRINT b
   80 PRINT PEEK b;" ";CHR$ PEEK
b
  100 PRINT PEEK 23563+256*PEEK 2
3564
```

Cuando lo ejecutamos, vemos el siguiente resultado:

```
23762
120 x
23814
41 )
0
```

Que se interpreta así:
La primera línea, es la posición del carácter “x” en la definición de la función f(x). En la segunda línea vemos el valor almacenado en esa dirección: 120, que es el código ASCII de “x”
En la tercera línea vemos la posición esperada para el primer argumento en la función g(); como en realidad no tiene argumentos, en la cuarta línea vemos que el carácter almacenado es el código 41, o sea el paréntesis de cierre.
En la quinta línea vemos que cuando no hay evaluación de función de usuario, el valor en esta variable del sistema es 0.

K DATA (23565, 1 byte, valor inicial 0)
Almacena información del último cambio de color de la línea que estamos editando y, como veremos luego, del cambio de otros atributos también. Por ejemplo, escribimos un comando o editamos una línea de programa y cambiamos el color en medio de la edición (presionamos Caps+Symbol para el cursor E, y a continuación las teclas 0 a 7 y continuamos escribiendo, la información se almacena en esta variable. Contra lo que podríamos pensar, no se tienen como en los atributos de pantalla, los ocho bits en uso aprovechándolos para definir papel, tinta, flash o brillo; simplemente el color de la última selección realizada. Esto lo vemos en los ejemplos (nota: la parte de papel o tinta que veamos de distinto color en lo que sigue, será vista como un gris claro si este documento es impreso en blanco y negro, y en realidad es de color verde, número 4 en la definición de Spectrum).

Primer ejemplo: pediremos el valor en esta dirección de memoria como un comando directo. Luego de la función PEEK ponemos cursor en modo E y presionamos el 4; después ponemos el número 23565 y finalmente ENTER.

```
PRINT PEEK 23565L
```

Aparecerá en la pantalla el valor 4.

Segundo ejemplo: similar al anterior, pero luego de la función PEEK ponemos cursor en modo E, presionamos la tecla de mayúsculas y sin soltarla presionamos el 4; después ponemos el número 23565 y finalmente ENTER.

```
PRINT PEEK 23565L
```

Veamos que, a pesar de que antes habíamos definido el color del papel y ahora estamos definiendo el color de la tinta, nos aparecerá en pantalla el mismo valor 4 que antes.

Tercer ejemplo: no solamente tenemos información de color. Por ejemplo, ahora vamos a cambiar de modo Inverse Video y True Video. Luego de la función PEEK, presionamos la tecla de mayúsculas y sin soltarla presionamos el 4 para poner video invertido.

```
PRINT PEEK 23565L
```

Nos va a aparecer en pantalla el valor 1, a pesar de que no estamos definiendo color de nada. En realidad, está en 1 porque lo último que hicimos fue activar un atributo que toma ese valor.

Finalmente, notemos que en todo caso si volvemos a pedir ver el valor en esa posición de memoria, seguirá estando el último cambio que hagamos, aunque la línea actual no tenga ningún atributo definido. O sea, nos volverá a aparecer el valor 4, 1 o el que sea que hayamos puesto.

Si hacemos POKE con cualquier valor, no alteramos en nada el funcionamiento del sistema.

TVDATA (23566, 2 bytes, valor inicial 0)
Acá tendremos la información sobre las coordenadas de impresión y el color cuando el sistema operativo imprime en la pantalla. No tiene mucha utilidad para el programador, porque solamente es utilizada en la rutina de la ROM que comienza en la dirección 2669 (0A6Dh), llamada "Control Characters with operands". Esta rutina guarda el carácter de control (PAPER, INK, OVER, FLASH, AT, TAB) en el byte menos significativo, y el atributo en el byte más significativo. Para los caracteres de dos operandos (como AT), se utiliza además el registro A del Z80.

STRMS (23568, 38 bytes, valor inicial no determinado)
Direcciones de rutinas de atención de canales de comunicación de entrada y salida. Son dos bytes por canal. En realidad son direcciones relativas a la variable del sistema CHANS (23631). Los primeros 14 bytes contienen las direcciones de los canales numerados desde -3 hasta 3 que en ese orden son: teclado, parte superior de pantalla, inserción en RAM, comandos, datos de INPUT, datos de PRINT y datos de LPRINT. Los siguientes 24 bytes no están definidos por Sinclair y son utilizables con cualquier fin.

CHARS (23606, 2 bytes, valor inicial 15360)
A la dirección de comienzo del conjunto de bytes donde se definen las figuras de los caracteres se le resta el valor 256, y el resultado va a esta variable. El conjunto de caracteres de la Spectrum se define mediante 8 bytes por carácter, y en la dirección 15616 (15360+256) se encuentra la definición del primer carácter (espacio, código 32), seguida de la definición del caracter 33 ("**!**"), y así sucesivamente hasta el caracter 127 ("©"), que se encuentra en la dirección 15376. En total son 96 caracteres, cuyas definiciones ocupan un total de 768 bytes. Si bien es obvio que la definición original está guardada en la ROM, es posible cambiar el valor de esta variable para que apunte a otra dirección en RAM, donde definimos nuestro propio juego de caracteres.
¿Por qué eso de restar 256? Porque los caracteres de verdadero interés para mostrar o imprimir son los códigos 32 hasta 127. Fijémonos que 32x8=256. Entonces se facilitan las rutinas que busquen la definición de caracteres, porque si queremos saber dónde se define el carácter 65 simplemente se hace (CHARS)+8x65 para averiguar la dirección.
Para las siguientes pruebas, vamos a explicar primero que el valor inicial 15360 se guarda en memoria así: 0→(CHARS), 60→(CHARS+1)

Probemos hacer:
```
POKE 23606,8
```
Con esto, nuestra variable CHARS tiene el valor 15368. Nos aparecerá el mensaje:
```
1!PL-!1;2
```
Este mensaje, en realidad es `0 OK, 0:1` y nuestra computadora realmente quiso darnos ese mensaje. Lo que pasa es que le hemos corrido 8 bytes la definición de los caracteres, y por lo tanto la definición de cada carácter apunta a lo que originalmente era la definición del siguiente carácter. Ahora el 0 se define como se definía el 1, el espacio como se definía el signo de admiración, etc. Volvamos atrás el valor, volviendo a poner 0. Recordemos que los caracteres cambiaron su definición, por lo que deberemos escribir sin prestar atención a lo que veamos en pantalla, que será:
```
QPLF!34717-1
```
Sí, acá estamos poniendo **`POKE 23606,0`**. Lo que pasa es que la P se ve como si fuera la Q, y así sucesivamente… Hasta el cursor cambió por .
Recibimos el mensaje final:
```
0 OK, 0:1
```

Por último, mostramos la forma en que podríamos cambiar el juego de caracteres completo de la Spectrum. Este programa reserva memoria inmediatamente debajo de la definición de los UDG, y llena esta parte con los 768 bytes de definición. En el número 30 de la revista MicroHobby hay una nota donde se definen tres juegos de caracteres, de los cuales tomamos los datos del primero.

```
  10 CLEAR 64000
  20 FOR n=64600 TO 65367
  30 READ a: POKE n,a
  40 NEXT n
 100 POKE 23607,251: POKE 23606,
88
 110 STOP
 120 POKE 23607,60: POKE 23606,0
 130 STOP
1000 REM Datos desde linea 1032
para el espacio hasta linea 1127
 para el copyright
1032 DATA 0,0,0,0,0,0,0,0
1033 DATA 0,16,16,24,24,0,24,0
1034 DATA 0,108,40,0,0,0,0,0
1035 DATA 0,36,126,36,36,126,36,
0
1036 DATA 0,16,126,80,126,22,126
,16
1037 DATA 0,98,100,8,16,38,70,0
1038 DATA 124,0,126,66,98,98,98,
0
1039 DATA 0,24,24,16,0,0,0,0
1040 DATA 0,4,8,8,8,8,4,0
1041 DATA 0,32,16,16,16,16,32,0
1042 DATA 0,0,20,8,62,8,20,0
1043 DATA 0,0,8,8,62,8,8,0
1044 DATA 0,0,0,0,0,24,24,8
1045 DATA 0,0,0,0,62,0,0,0
1046 DATA 0,0,0,0,0,24,24,0
1047 DATA 0,0,2,4,8,16,32,0
1048 DATA 0,126,70,74,86,102,126
,0
1049 DATA 0,24,40,8,24,24,24,0
1050 DATA 0,126,66,2,126,96,126,
0
1051 DATA 0,124,68,30,6,70,126,0
1052 DATA 0,4,68,68,68,126,12,0
1053 DATA 0,124,64,126,6,70,126,
0
1054 DATA 0,126,64,126,70,70,126
,0
1055 DATA 0,62,2,6,6,6,6,0
1056 DATA 0,60,36,126,70,70,126,
0
1057 DATA 0,126,66,66,126,6,6,0
1058 DATA 0,0,24,24,0,24,24,0
1059 DATA 0,0,24,24,0,24,24,8
1060 DATA 0,0,4,8,16,8,4,0
1061 DATA 0,0,0,62,0,62,0,0
1062 DATA 0,0,16,8,4,8,16,0
1063 DATA 0,62,2,62,48,0,48,0
1064 DATA 0,124,0,126,66,98,98,0
1065 DATA 0,60,36,126,98,98,98,0
1066 DATA 0,124,68,126,98,98,126
,0
1067 DATA 0,126,66,96,96,98,126,
0
1068 DATA 0,124,66,98,98,98,124,
0
1069 DATA 0,126,64,126,96,96,126
,0
1070 DATA 0,126,64,126,96,96,96,
0
1071 DATA 0,126,64,102,98,98,126
,0
1072 DATA 0,66,66,126,98,98,98,0
1073 DATA 0,60,16,24,24,24,60,0
1074 DATA 0,4,4,6,6,70,126,0
1075 DATA 0,68,68,126,98,98,98,0
1076 DATA 0,64,64,96,96,96,124,0
1077 DATA 0,126,74,106,106,106,1
06,0
1078 DATA 0,126,66,98,98,98,98,0
1079 DATA 0,126,70,70,66,66,126,
0
1080 DATA 0,126,66,126,96,96,96,
0
1081 DATA 0,126,66,66,66,94,126,
0
1082 DATA 0,124,68,126,98,98,98,
0
1083 DATA 0,126,64,126,6,6,126,0
1084 DATA 0,126,16,24,24,24,24,0
1085 DATA 0,66,66,98,98,98,126,0
1086 DATA 0,98,98,98,102,36,60,0
1087 DATA 0,66,74,106,106,106,12
6,0
1088 DATA 0,66,66,60,98,98,98,0
1089 DATA 0,66,66,126,24,24,24,0
1090 DATA 0,126,2,124,96,98,126,
0
1091 DATA 0,14,8,8,8,8,14,0
1092 DATA 0,0,64,32,16,8,4,0
1093 DATA 0,112,16,16,16,16,112,
0
1094 DATA 0,16,56,84,16,16,16,0
1095 DATA 0,0,0,0,0,0,0,255
1096 DATA 0,62,34,120,48,48,62,0
1097 DATA 0,0,124,4,124,108,124,
0
1098 DATA 32,32,62,34,50,50,62,0
1099 DATA 0,0,60,32,48,48,60,0
1100 DATA 4,4,124,68,100,100,124
,0
1101 DATA 0,0,124,68,124,96,124,
0
1102 DATA 0,60,32,120,32,48,48,0
1103 DATA 0,0,124,68,100,124,4,6
0
1104 DATA 64,64,124,68,100,100,1
00,0
1105 DATA 0,16,0,16,24,24,24,0
1106 DATA 0,16,0,16,24,24,8,24
1107 DATA 0,32,36,62,50,50,50,0
1108 DATA 0,16,16,24,24,24,24,0
1109 DATA 0,0,126,74,106,106,106
,0
1110 DATA 0,0,124,68,100,100,100
,0
1111 DATA 0,0,124,68,100,100,124
,0
1112 DATA 0,0,124,68,100,100,124
,64
1113 DATA 0,0,124,68,100,100,124
,4
1114 DATA 0,0,60,32,48,48,48,0
1115 DATA 0,0,124,64,124,12,124,
0
1116 DATA 16,16,56,16,24,24,24,0
1117 DATA 0,0,68,68,100,100,124,
0
1118 DATA 0,0,100,100,108,40,56,
0
1119 DATA 0,0,66,74,106,106,124,
0
1120 DATA 0,0,68,68,56,100,100,0
1121 DATA 0,0,68,100,100,124,4,6
0
1122 DATA 0,0,124,4,56,96,124,0
1123 DATA 0,12,8,16,8,8,12,0
1124 DATA 0,16,16,16,16,16,16,0
1125 DATA 0,48,16,8,16,16,48,0
1126 DATA 0,122,94,0,0,0,0,0
1127 DATA 126,129,157,209,209,22
1,193,126
```

Al ejecutar este código, se copian los bytes con la definición del nuevo juego de caracteres a partir de la dirección 64600, desde el espacio hasta el "©". Luego cambiamos el valor de CHARS a 64344 (64600-256). Para restablecer la definición original, basta con ejecutar el programa desde la línea 120.

RASP (23608, 1 byte, valor inicial 64)

Cuando el buffer de edición se llena, el sistema no permite ingresar nuevos caracteres y a la vez avisa de esta condición con un zumbido de alarma. Esto ocurre si tecleamos o intentamos editar una línea demasiado larga. En esta variable indicamos la duración de este sonido, medida en 1/50 de segundo (1/60 en la TS 2068). Poniendo el valor en cero, escucharemos solamente un leve chasquido, mientras que si lo ponemos en 255 será algo decididamente molesto.

Esta condición de error también se da si la memoria se agota. Para probarlo, vamos a bajar el valor de RAMTOP[18] dándole un valor suficientemente pequeño como para que no quede espacio libre:

```
CLEAR 23860
```

[18] Más sobre la RAMTOP cuando veamos la variable del sistema correspondiente en la dirección 23730

En realidad, con esto prácticamente no dejamos espacio para nada. De hecho, veamos que si queremos ingresar una simple línea el sistema no lo permite:

```
10 REM ■
```

Nos saldrá el mensaje:

```
G No room for line, 0:1
```

Volviendo al ejemplo con esta variable del sistema, luego de esta instrucción CLEAR probamos hacer:

```
REM prueb■
```

El sistema no nos permite siquiera completar la palabra, y escucharemos el sonido de error.

PIP (23609, 1 byte, valor inicial 0)

En esta variable guardamos la duración del sonido que escuchamos al presionar cada tecla. Su funcionamiento es análogo al de la variable anterior, pero en vez de un zumbido es un sonido intermedio entre SI bemol y SI. De hecho, probemos:

```
POKE 23609,255■
```

Y luego:

```
BEEP 1,34.4■
```

(sí, el comando BEEP admite decimales para el valor de la nota)

Notaremos que el sonido que sale al presionar Enter para ejecutar la instrucción es prácticamente el mismo que el resultado de la ejecución.

ERR NR (23610, 1 byte, valor inicial 255)

La Spectrum define estados de error con valores numéricos. En esta posición de memoria se guarda el código de error ocurrido al que se le resta 1. Esto explica el por qué del valor inicial 255: 0 es el código de "ejecución exitosa", y 255 es la representación binaria de 11111111, que es la representación de -1 en complemento a 2, Podemos hacer pruebas:

```
POKE 23610,0
```

Veremos que nos aparece el error:

```
1 NEXT without FOR, 0:1
```

El último código error es el obtenido poniendo:

```
POKE 23610,26
```

Y se nos muestra:

```
R Tape loading error, 0:1
```

Valores más altos, nos muestran códigos sin sentido aparente, salvo ver que el código en letra va avanzando. Es interesante ver el efecto de asignar 28 a esta variable.

FLAGS (23611, 1 byte, valor inicial 204)

Esta variable debe ser vista en realidad como un grupo de 8 bits, donde se guardan indicadores de control utilizados por el Basic. Así tenemos:

Bit 0: Cuando se debe imprimir una palabra clave (PRINT, POKE, BEEP, THEN, OR, etc.), por pantalla o impresora, indica si se debe anteponer un espacio (en ese caso vale 0) o no (vale 1).

Bit 1: Indica hacia dónde va la salida impresa. Si está en 1, va a la impresora; en 0 va a la pantalla

Bit 2: Usada para imprimir los caracteres de una línea de programa, indica si se hace en modo K (valor=0) o modo L (valor=1).

Bit 3: Si está en 1, el cursor del editor es K.

Bit 4: No se utiliza

Bit 5: Indica si se ha presionado una nueva tecla. Cada 20 milisegundos una rutina inspecciona el teclado y pasa el valor de la tecla presionada a la variable del sistema LAST K (dirección 23560) y pone el valor de este bit en 1 para indicar que se ha presionado una nueva tecla.

Bit 6: Indica el tipo de argumento de una función. Si está en 1, es numérico y sino es alfanumérico.

Bit 7: En 1, significa que estamos listos para ejecutar un comando. En 0, es que todavía hay que controlar que esté bien la sintaxis del mismo.

Es interesante notar que vamos a encontrar más útiles estas variables si utilizamos las rutinas correspondientes en ROM que si queremos hacer algo desde el Basic en sí.

TV FLAG (23612, 1 byte, valor inicial 1)
Indicadores asociados con la pantalla. Se accede de a bit, y encontraremos:
Bit 0: indica si estamos utilizando la parte inferior de la pantalla (vale 0 señalando que utilizamos las dos últimas líneas, por ejemplo en un INPUT) o la parte superior (vale 1)
Bit 3: indica si el modo de impresión (K o L) cambió (1) o no (0).
Bit 5: indica si se debe limpiar la parte inferior (1) o no (0)

ERR SP (23613, 2 bytes, valor inicial 65364)
Indica la dirección en la pila de máquina que contiene la dirección a donde se debe saltar en caso de error del Basic, aunque en este caso el concepto de condición de error también incluye la terminación del programa en sí.
La rutina de reporte de errores del Spectrum está en la dirección 4867 (1303h) de la ROM. Es la que imprime el código de error junto con la descripción del mismo (el código 0 de OK también está contemplado en la rutina), según el valor que encuentre en la variable del sistema ERR NR (dirección 23610), y luego se queda a la espera de que el usuario presione una tecla. Nosotros podemos alterar esta dirección para tener nuestra propia rutina de manejo de errores. Esto nos permitiría cosas como, por ejemplo, proteger nuestro programa para que el usuario no lo pueda detener y examinar (por ejemplo: lo normal es que se muestre el mensaje BREAK u otro, pero podemos hacer que en vez de mostrar el mensaje la computadora se reinicie).
¿Por qué esta variable contiene "la ubicación de la ubicación" de la dirección de la rutina de manejo de errores? Porque la pila de máquina es alterada cuando ejecutamos la instrucción GOSUB (ya que se guarda en memoria adónde retornar), y como la dirección de la rutina de manejo de errores también está dentro de esta pila, es necesario no perderla.
Vamos a ver el siguiente programa de ejemplo, donde hacemos GOSUB sin el correspondiente RETURN:

```
  10 LET a=PEEK 23613+256*PEEK 2
3614
  20 PRINT a,
  30 LET a$=STR$ PEEK a+":"+STR$
 PEEK (a+1)+":"+STR$ PEEK (a+2)+
":"+STR$ PEEK (a+3)+":"+STR$ PEE
K (a+4)
  40 PRINT a$
  50 IF RND>0.5 THEN GO SUB 10
5000 GO SUB 10
```

Al ejecutarlo, iremos viendo cómo la dirección original va disminuyendo:

```
63997             3:19:0:62:0
63994             3:19:136:19:2
63991             3:19:136:19:2
63988             3:19:50:0:3
63985             3:19:50:0:3
63982             3:19:136:19:2
63979             3:19:136:19:2
63976             3:19:136:19:2
63973             3:19:136:19:2
63970             3:19:50:0:3
63967             3:19:50:0:3
63964             3:19:136:19:2
63961             3:19:136:19:2
63958             3:19:50:0:3
63955             3:19:136:19:2
63952             3:19:136:19:2
63949             3:19:50:0:3
63946             3:19:136:19:2
63943             3:19:136:19:2
63940             3:19:50:0:3
63937             3:19:50:0:3
63934             3:19:50:0:3

scroll?
```

Notemos que la disminución es de a tres, lo que indica que el guardado de la dirección de GOSUB toma tres bytes y se corre "para adelante" el dato de la dirección adonde saltar en caso de error. No

debemos confundirnos por esta disminución de a tres, porque eso es, como decíamos, debido a los datos de retorno del GOSUB. La dirección de la rutina de error se almacena en dos bytes, originalmente 3 y 19 ya que 3+256x19=4867 (la dirección final que nos interesa). A propósito hemos hecho que se muestren cinco, de modo que podemos notar que:

a) Luego del primer grupo de dos bytes, vemos un 0 seguido el valor 62 que indica fin de la zona de Basic y comienzo de la memoria reservada. En breve veremos algo más al respecto.
b) En la siguiente línea, notamos cómo el byte en la posición 63997, que guardaba el valor 3, pasó a guardar otro valor. Lo mismo para el byte en la posición 63998, que guardaba el valor 19 y ahora tiene otro. Esto se repetirá de ahora en adelante para las siguientes líneas. En realidad, los valores 3 y 19 no se perdieron, sino que se guardaron a partir de la nueva dirección señalada por la variable ERR SP.
   ERR SP disminuyó su valor en tres, los dos bytes con el dato de la dirección de la rutina de reporte de error se copiaron a esta nueva dirección, y los tres bytes liberados guardaron la información de a dónde retornar cuando se encuentre el RETURN. De estos bytes, los dos primeros indican el número de línea, y el tercero es el número de instrucción dentro de esa línea.

Habremos notado que si el GOSUB es desde la línea 50, los bytes que indican adónde retornar tienen los valores 50 y 0 para el número de línea. Vemos que 50 + 256x0=50. Cuando el GOSUB es desde la línea 5000, los valores son 136 y 19, ya que 136+256x19=5000. Cuando no hay ningún GOSUB indicado (al comienzo de la ejecución), vemos que los bytes son 0 y 62 (que indica el fin del espacio Basic y comienzo de la RAMTOP). Si quisiéramos ejecutar una instrucción RETURN en ese punto (o antes de comenzar la ejecución del programa), nos daría el mensaje de error:

```
7 RETURN without GOSUB, 0:1
```

Hay dos motivos por los cuales la Spectrum no puede hacer el RETURN: por un lado, la pila de GOSUB se pasaría al otro lado de la RAMTOP, y por otro el número más grande de línea en un programa Basic es 9999, mientras que la operación 0+256x62 daría 15872, un valor de retorno imposible de tener. Y en la realidad, la rutina que ejecuta la instrucción RETURN[19] obtiene los datos de la pila y compara la dirección con el valor 3Eh (62); si encuentra ese valor, muestra el mensaje de error.

Ya estuvimos viendo que esta variable de sistema cambia de valor para que podamos apuntar siempre a la dirección donde encontraremos la ubicación de la rutina de manejo de errores, independientemente de la corrupción de la pila provocada por la ejecución de instrucciones GOSUB. Nos falta mostrar un ejemplo de uso de esta variable en particular. Probemos el programa, adaptado de una rutina aparecida en la revista Sinclair User[20]:

```
  10 LET a=PEEK 23613+256*PEEK 2
3614
  20 POKE a,0
  30 POKE a+1,0
  40 FOR n=1 TO 1000
  50 PRINT AT 0,0;n
  60 NEXT n
```

Guardémoslo para más pruebas antes de hacer RUN, porque el resultado irreversible de ejecutarlo será el reinicio de la computadora. Lo primero que se hace es poner el valor 0 en la posición apuntada por la variable del sistema ERR SP. Luego comienza un loop que muestra los números de 1 hasta 1000. Ya sea que apretamos BREAK o dejamos que el programa termine, se invocará la rutina de manejo… que hemos determinado que ahora se encuentra en la dirección 0. Y en esta dirección 0 está la rutina de inicio de la Spectrum.
Como se ve, esto puede servir para evitar que un usuario detenga nuestro programa (por ejemplo, para copiarlo); apenas lo haga, el sistema se reinicia y se perderá todo.

[19] Se encuentra en la dirección 7971 (1F23h) y está explicada en el libro "The Complete Spectrum ROM Disassembly"
[20] Número 29 (Agosto de 1984). Accesible en http://www.sincuser.f9.co.uk/029/index.htm

LIST SP (23615, 2 bytes, valor inicial 0)
Se utiliza para guardar el registro SP de modo que pueda ser recuperado luego de hacer un listado automático. Hay dos formas de obtener el listado de un programa Basic: mediante la orden LIST o cuando presionamos ENTER. Esta última manera es llamada "listado automático". La necesidad de esta variable se entiende cuando pensamos que la ejecución de la rutina en ROM que muestra el listado puede terminar por varios motivos: o se hizo BREAK, o se respondió "n" a la pregunta de Scroll, o simplemente el listado se terminó de mostrar. De esta manera podemos pensar que funciona de forma similar a la variable ERR SP guardando la "ubicación de la ubicación"; en este caso, la ubicación donde encontremos el valor del Stack Pointer.

MODE (23617, 1 byte, valor inicial 0)
Especifica el cursor K, L, C, E o G. Podemos probar con esta rutina para ver el significado de los valores:

```
   10 FOR n=0 TO 10
   20 POKE 23617,n
   30 PRINT n
   40 INPUT a
   50 NEXT n
 1000 POKE 23617,0
```

Podemos ver que incluso aparecen cursores con cualquier letra, sin significado particular para la computadora. Más aún, podemos probar muchos otros valores para ver diferentes efectos gráficos o incluso aparición de tokens como cursor. No hay problema en cambiar este valor por el que sea, el sistema no se cuelga por eso.

NEWPPC (23618, 2 bytes, valor inicial 0)
Mantiene el valor al que un programa Basic debe saltar luego de una instrucción de bifurcación (GOTO/GOSUB) o de ejecución de programa (RUN). Su valor no cambia hasta que se vuelva a ejecutar un nuevo salto. Para comprender cómo va tomando los valores, consideremos este programa:

```
  10 PRINT "Linea 10: ";PEEK 236
18+256*PEEK 23619
  20 IF RND>0.5 THEN PRINT "De l
a linea 20 salto a la 40": GO TO
 40
  30 PRINT "Linea 30: ";PEEK 236
18+256*PEEK 23619
  40 IF RND>0.5 THEN PRINT "De l
a linea 40 salto a la 30": GO TO
 30
  50 PRINT "Linea 50: ";PEEK 236
18+256*PEEK 23619
  60 IF RND>0.5 THEN PRINT "De l
a linea 60 salto a la 10": GO TO
 10
```

Las líneas 10, 30 y 50 muestran el número de línea en ejecución y el valor de la variable en ese momento. Las líneas 20, 40 y 60 saltan o no a otras líneas según el resultado de la función RND. Cuando van a realizar el salto, previamente dejan un mensaje indicando en qué línea estaban y adónde saltarán. Analizaremos algunos ejemplos de salida de pantalla al ejecutarse la instrucción RUN:

```
Linea 10: 0
Linea 30: 0
Linea 50: 0
```

En este primer caso, la variable arranca con el valor 0 (la instrucción RUN no especifica valor y se asume 0) y, aunque la ejecución pasa por las 6 líneas, evidentemente el valor de RND siempre dio menos que 0.5 y por lo tanto no se efectuó ningún salto. La variable entonces siguió manteniendo el valor 0 hasta el fin de la ejecución del programa.

```
Linea 10:  0
Linea 30:  0
De la linea 40 salto a la 30
Linea 30:  30
De la linea 40 salto a la 30
Linea 30:  30
Linea 50:  30
De la linea 60 salto a la 10
Linea 10:  10
De la linea 20 salto a la 40
De la linea 40 salto a la 30
Linea 30:  30
Linea 50:  30
De la linea 60 salto a la 10
Linea 10:  10
Linea 30:  10
Linea 50:  10
De la linea 60 salto a la 10
Linea 10:  10
Linea 30:  10
Linea 50:  10
```

En este segundo caso, encontramos un salto en la línea 40 y, entonces, la variable cambia de valor. Notemos que, como en el caso anterior, el valor de la variable se mantiene mientras no haya salto. Por último, si ejecutamos el programa desde la línea 10 (instrucción RUN 10) tendremos

```
Linea 10:  10
De la linea 20 salto a la 40
Linea 50:  40
De la linea 60 salto a la 10
Linea 10:  10
Linea 30:  10
Linea 50:  10
De la linea 60 salto a la 10
Linea 10:  10
De la linea 20 salto a la 40
Linea 50:  40
```

Veamos que en este caso, la primera línea de salida por pantalla ya está mostrando el valor de la línea de programa a partir de la cual se ordenó ejecutar el programa.
Si en este momento, al terminar la ejecución del programa escribimos la orden:

```
PRINT PEEK 23618+256*PEEK 23619
```

Notaremos que la computadora nos devuelve el valor 40. Esto demuestra lo dicho antes, que solamente las instrucciones RUN, GOTO y GOSUB alteran este valor.

NSPPC (23620, 1 byte, valor inicial 255)
Número de sentencia en una línea a la que hay que saltar. Si se ejerce la instrucción POKE apuntando primeramente a NEWPPC, y luego a NSPPC, se fuerza un salto a una sentencia especificada en una línea. Esto implica que tenemos la posibilidad de contar con una forma de GOTO más potente, donde detallamos hasta la sentencia adonde continuar la ejecución. Vamos a ver el siguiente programa de ejemplo:

```
   10 POKE 23618,30
   20 PRINT "Linea 20"
   30 PRINT "Linea 30"
   40 POKE 23618,100
   50 POKE 23620,2
   60 PRINT "Linea 60"
  100 PRINT "Linea 100 Inst. 1":
PRINT "Linea 100 Inst. 2"
```

Al ejecutarlo, notaremos que la salida por pantalla es la siguiente:

```
Linea 20
Linea 30
Linea 100 Inst. 2
```

La primera línea pone el valor 30 en la variable NEWPPC[21]. Sin embargo, el programa no saltó a la línea 30 porque la variable NEWPPC se utiliza en instrucciones GOTO o GOSUB y este no es el

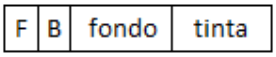
[21] En realidad esta variable del sistema ocupa dos bytes, y hemos tocado solamente el menos significativo, ubicado en la dirección 23618. El byte más significativo está en la dirección 23619 y sigue valiendo 0; es por eso que no precisamos modificar ese dato.

caso. En la línea 40 volvemos a cambiar el valor de esa variable a 100. Como antes, no hay consecuencia a causa de esta modificación y el programa continúa su ejecución normal, y en la línea 50 se cambia el valor de NSPPC. Pero ahora esta última modificación sí tiene efecto, y el sistema no solamente sabe que tiene que ejecutar la segunda sentencia, sino que también utiliza el valor en NEWPPC para determinar de qué línea es la segunda sentencia a la que tiene que saltar. Por ese motivo es que observamos el salto a la segunda instrucción de la línea 100.

PPC (23621, 2 bytes, valor inicial 65534)
Número de línea de la sentencia en curso. No debemos confundir esta variable con NEWPPC a pesar de que cumplen una función similar y en muchas oportunidades contienen el mismo valor. A diferencia de NEWPPC, esta variable siempre es modificada cada vez que la ejecución del programa en Basic pasa a otra línea. El intérprete de Basic necesita saber en todo momento qué línea se está ejecutando y para esto utiliza esta variable.

SUBPPC (23623, 1 byte, valor inicial 0)
Número dentro de la línea de la sentencia en curso. Funciona de modo similar a PPC, y combinada con la anterior podrían ser utilizadas para implementar una rutina de seguimiento de ejecución del programa, para buscar errores.

BORDCR (23624, 1 byte, valor inicial 56)
Contiene atributos del borde y el área de edición (las dos líneas inferiores de la pantalla). En el caso del borde, solamente se guarda la información del color; pero para el área de edición se puede controlar además el brillo, flash y color de tinta, cosa que no es posible realizar desde el Basic. Para esto, los ocho bits que forman el byte almacenado se corresponden con la siguiente estructura:



El bit más significativo (el bit 7) es el parpadeo o flash: en 1 si ocurre, en 0 sino.
El siguiente bit (bit 6) indica el brillo: 1 si el área es más clara, 0 si es sin brillo.
Los bits 3, 4 y 5 indican el color del fondo para el área de edición y a la vez el color del borde.
Los bits 0, 1 y 2 indican el color de la tinta.
Los colores representados con tres bits para el fondo y la tinta son los ocho conocidos, que toman los valores de 000 (0, negro) 001 (1, azul) y así hasta 111 (7, blanco).
No es sorprendente que toda esta representación de atributos de borde y área de edición sea la misma que la empleada para los atributos de los caracteres en pantalla, explicados en el primer capítulo de este apunte.

E PPC (23625, 2 bytes, valor inicial 0)
Cuando se tiene un listado en Basic, el sistema conserva lo que se llama cursor de edición de programa o cursor de línea. Este valor se conserva en la variable del sistema E PPC, y viene a ser el número identificador de la última línea que hemos editado. Cuando presionemos Caps Shift + 1 para editar una línea, se seleccionará la línea con el número guardado en esta ubicación de memoria. Podemos cambiar el valor de E PPC mediante POKE, o desplazando el cursor con Caps Shift + 6 y Caps Shift + 8.

VARS (23627, 2 bytes, valor inicial 23755)
Dirección de comienzo de definición de las variables en el intérprete Basic. En el capítulo 1 se explica cómo funciona el área de memoria apuntada por esta variable. Si utilizamos VARS junto con la variable E LINE (23641), podremos calcular el espacio total ocupado por las variables Basic. Vamos a ver con un ejemplo. Al momento de encender la computadora ejecutamos esta instrucción:

```
PRINT PEEK 23755
```

¿Por qué pedimos ver qué hay en la dirección 23755? Porque la variable del sistema VARS arranca teniendo este valor. Cuando ejecutemos esta instrucción, por pantalla veremos el valor 128. Esto

significa que no hay variables definidas[22]. Si ahora pedimos el siguiente cálculo:

```
PRINT PEEK 23641+256*PEEK 23642-
PEEK 23627-256*PEEK 23628
```

La computadora nos devolverá el valor 1. Este resultado es lógico, ya que la zona de variables en este momento alberga un único byte (el valor 128 que indica fin de definición de variables). Si ahora creamos una variable con la instrucción:

```
LET a=0
```

Y volvemos a ejecutar la instrucción anterior, la computadora ahora nos calculará el valor 7. Eso significa que asignar el valor a la variable A ocupó en total 6 bytes de la memoria, y el séptimo byte es el valor de fin de definición de variables. De hecho, podemos comprobarlo:

```
PRINT PEEK 23755'PEEK 23756'PEEK
 23757'PEEK 23758'PEEK 23759'PEE
K 23760'PEEK 23761'PEEK 23762
```

Hemos ejecutado así la instrucción en modo directo para evitar que un programa o la creación de nuevas variables alteren la disposición actual de memoria. Veremos por pantalla:

```
97
0
0
0
0
0
128
245
```

El primer byte nos marca que se está definiendo una variable de tipo numérica identificada con una sola letra, y que el nombre de la misma es "a". Los siguientes cinco bytes son la representación del valor 0. Y por último, vemos el valor 128 (fin de zona de variables), seguido por el primer byte de la siguiente zona de memoria, la apuntada por la variable E LINE. El valor 245 que vemos ocupando este byte es, no por casualidad, el código de la palabra ("token") PRINT, ya que E LINE indica la dirección en memoria del comando que se está ingresando en modo directo.

DEST (23629, 2 bytes, valor inicial 0)

Esta variable del sistema está, en cierta forma, asociada a la anterior. El valor almacenado es la dirección de memoria donde esté almacenado el valor de la última variable en Basic a la que hayamos accedido o asignado. Es interesante igualmente ver el comportamiento de la misma. Si ejecutamos el programa:

```
  10 PRINT "Antes de crear varia
bles:"'"VARS: ";PEEK 23627+256*P
EEK 23628'"E LINE: ";PEEK 23641+
256*PEEK 23642'"DEST: ";PEEK 236
29+256*PEEK 23630
  20 LET a=0
  30 PRINT "Luego de crear la va
riable a:"'"VARS: ";PEEK 23627+2
56*PEEK 23628'"E LINE:";PEEK 236
41+256*PEEK 23642'"DEST: ";PEEK
23629+256*PEEK 23630
  40 LET b=0
  50 PRINT "Luego de crear la va
riable b:"'"VARS: ";PEEK 23627+2
56*PEEK 23628'"E LINE:";PEEK 236
41+256*PEEK 23642'"DEST: ";PEEK
23629+256*PEEK 23630
  60 LET a=1
  70 PRINT "Al refenciar la vari
able a:"'"VARS: ";PEEK 23627+256
*PEEK 23628'"E LINE:";PEEK 23641
+256*PEEK 23642'"DEST: ";PEEK 23
629+256*PEEK 23630
```

[22] Como vimos en el capítulo 1, las variables se guardan una detrás de otra en memoria, y en el primer byte de cada definición se combina la descripción del tipo de variable con la primera letra de la misma, siendo 128 (80h) un valor imposible de lograr, por lo que se utiliza para señalar el fin de estas enumeraciones. Y si el primero de todos los bytes tiene este valor, es porque directamente no hay variables.

Obtendremos el siguiente resultado:

```
Antes de crear variables:
VARS: 24533
E_LINE: 24534
DEST: 0
Luego de crear la variable a:
VARS: 24533
E_LINE:24540
DEST: 23976
Luego de crear la variable b:
VARS: 24533
E_LINE:24546
DEST: 24164
Al refenciar la variable a:
VARS: 24533
E_LINE:24546
DEST: 24533
```

La explicación, válida como ejemplo para las variables del sistema VARS, E LINE y DEST es la siguiente. VARS apunta al comienzo de la zona de definición de variables. Como ahora hay un programa, que ocupa una zona anterior de memoria[23], la zona de variables se "corrió" de la ubicación original (23755) a esta nueva (24533). Como vemos, este valor no cambia durante la ejecución del programa, lo cual es lógico porque el mismo no es alterado al correr. Más interesante es ver cómo cambian E LINE y DEST. Al comienzo, E LINE vale VARS+1, también lógico porque al comienzo no hay variables definidas (Basic es un lenguaje interpretado, y no se reserva espacio para una variable mientras no se ejecute la instrucción que la declare) y solamente se guarda el valor 128.

Cuando creamos la primera variable, E LINE se incrementa en 6 (al insertarse la definición de **a** con su valor) y DEST pasa a valer 23976. Este es un valor intermedio entre 23755 y 24533, ¿por qué se tomó este valor que parece estar en medio del programa Basic? Efectivamente, porque se refiere a la parte del programa donde está la definición de la variable **a**. Si ejecutamos:

```
PRINT PEEK 23976
```

Nos devolverá el valor 97. Preguntar por el contenido de las direcciones 23976 a la 23984 nos dará los valores 97, 61, 48, 14, 0, 0, 0, 0 y 0. O sea, "a", "=", "0", indicación de que viene un número (valor 14) y los cinco bytes en 0 que representan el valor numérico que tomará **a**. Interesante, DEST entonces también es utilizada por el sistema operativo para apuntar a la definición de una variable en el programa al momento de ejecutar la interpretación de la misma[24].

Notemos que al crear la variable **b** ocurre algo similar, apuntando DEST a la definición de la misma dentro del programa. E LINE se vuelve a incrementar en 6, dado que ahora la zona de variables alberga la nueva recién creada.

La diferencia la vemos cuando volvemos a asignar un valor para **a**. En este último caso, DEST apuntará (ahora sí) a la definición de la variable dentro de la zona de memoria específica para almacenar variables, no la definición en el programa. Veamos que E LINE no cambia, simplemente porque no creamos nuevas variables.

CHANS (23631, 2 bytes, valor inicial 23734)

Dirección donde podremos encontrar información sobre los canales de datos a través de los cuales la computadora se comunica con el exterior. En el capítulo 1 se puede leer una explicación más detallada de cómo se sabe la cantidad de canales definidos, cuáles son, y cómo interpretar la información por canal. En este párrafo nos limitaremos a comentar que se pueden tener hasta 16 canales de comunicación, que los primeros cuatro están predefinidos, y que para la definición de cada canal tenemos cinco bytes.

---

[23] Ver en el mapa de memoria (capítulo 1) que el programa Basic ocupa la zona inmediata anterior a las variables. Apuntada por la variable PROG, la operación (VARS) – (PROG) daría la longitud del programa.

[24] Véase en el capítulo 1 para más detalles sobre la forma de representar variables numéricas

CURCHL (23633, 2 bytes, valor inicial 23734)
Dirección utilizada en el momento que se ejecuta la entrada o salida de información por un canal. Durante una operación de entrada/salida, apunta al bloque de cinco bytes que contiene la información del canal que se está utilizando: nombre del canal, y dónde encontrar las rutinas que manejan la entrada y la salida de datos a través del mismo.

PROG (23635, 2 bytes, valor inicial 23755)
Almacena la dirección donde comienza el programa en Basic. En el capítulo 1 hemos visto detalladamente la manera en que se guarda el programa en memoria; sin embargo, acá les dejamos un pequeño programa que se examina a sí mismo y muestra cómo está almacenado:

```
  10 LET a=PEEK 23635+256*PEEK 2
3636
  20 FOR n=0 TO 100
  30 PRINT "Direccion ";(a+n);":
 ";PEEK (a+n);TAB 25;CHR$ PEEK (
a+n)
  40 NEXT n
```

Al ejecutarlo, podemos ver los primeros bytes que definen la primera línea del programa:

```
Direccion 23755: 0         ?
Direccion 23756: 10        ?
Direccion 23757: 39        '
Direccion 23758: 0         ?
Direccion 23759: 241       LET
Direccion 23760: 97        a
Direccion 23761: 61        =
Direccion 23762: 190       PEEK
Direccion 23763: 50        2
Direccion 23764: 51        3
Direccion 23765: 54        6
Direccion 23766: 51        3
Direccion 23767: 53        5
Direccion 23768: 14        ?
Direccion 23769: 0         ?
Direccion 23770: 0         ?
Direccion 23771: 83        S
Direccion 23772: 92        \
Direccion 23773: 0         ?
Direccion 23774: 43        +
Direccion 23775: 50        2
Direccion 23776: 53        5

D BREAK - CONT repeats, 30:1
```

Notemos la representación del número de línea, la longitud en bytes del texto de la misma, y el comienzo del texto de la línea en sí.

NXTLIN (23637, 2 bytes, valor inicial 0)
Dirección de la siguiente línea del programa. Es decir, cuando estamos ejecutando el programa, la dirección donde se encuentra la línea que sigue a la que actualmente se está interpretando. No debemos confundirnos con el número de línea: es la posición en memoria donde comienza, en el listado Basic, la definición de la siguiente línea de programa (o, en modo directo, el byte posterior al último de la instrucción a ejecutar). Si bien el valor inicial es 0, al ejecutar la primera instrucción, cualquiera sea, ya es alterado su valor. Por ese motivo, si encendemos la computadora e inmediatamente pedimos lo siguiente:

```
PRINT PEEK 23637+256*PEEK 23638
```

El sistema nos responderá con un valor distinto de 0:

```
23793
```

Pero notemos qué pasa si pedimos lo siguiente:

```
PRINT PEEK 23637,PEEK 23638
```

Se retornan los valores 231 y 92, y notamos que 231+256x92=23783. Si en cambio ejecutamos dos instrucciones por separado, una detrás de la otra:

```
PRINT PEEK 23637
```
nos devuelve 218

```
PRINT PEEK 23638
```
nos devuelve 92

Y al hacer la cuenta, 218+256x92 nos da 23770.
¿Por qué hemos obtenido estos valores diferentes cuando todas las veces estamos pidiendo el mismo dato? Porque el resultado depende de la longitud de la instrucción empleada para obtenerlo. La variable del sistema E LINE (ocupa la dirección 23641, la estudiaremos más adelante) guarda la dirección del comando que escribimos en teclado. Por ahora, baste con decir que el valor inicial de la misma es 23756. Si restamos este valor a los resultados obtenidos previamente obtendremos la cantidad de bytes que ocupó cada una de las instrucciones anteriores. Por ejemplo, la primera instrucción ocupó 23793-23756=37 bytes, y por lo que vimos en el capítulo 1 sobre cómo se guardan las instrucciones en un programa (lo cual también es aplicable para instrucciones en modo inmediato) podemos deducir cómo se llegó a este valor:

PRINT	1 byte
PEEK	1 byte
23637	11 bytes (5 para la cadena "23637", un byte con el valor 14 y 5 bytes representando el número)
+	1 byte
256	9 bytes (3 para la cadena "256", un byte con el valor 14, y 5 bytes para representar el número 256)
*	1 byte
PEEK	1 byte
23638	11 bytes (5 para la cadena "23638", un byte con el valor 14 y 5 bytes representando el número)
<ENTER>	1 byte

Por último, así como dijimos (y explicamos) que no debíamos confundir dirección de memoria con número de línea, también es importante dejar claro que el valor contenido es el de la ubicación en memoria de la siguiente línea en un listado Basic, y no de la línea que debe ser ejecutada posteriormente a la línea que se está ejecutando. O sea, en el caso de un GOTO, un NEXT o RETURN, esta variable contiene la dirección de la siguiente línea y no de la línea a la que se saltará. Por ejemplo, veamos el programa:

```
  10 PRINT "Linea 10: ";PEEK 236
37+256*PEEK 23638: FOR n=1 TO 3
  20 PRINT "Linea 20 (";n;"): ";
PEEK 23637+256*PEEK 23638
  30 PRINT "Linea 30: ";PEEK 236
37+256*PEEK 23638: NEXT n: PRINT
 "Linea 30 despues ";PEEK 23637+
256*PEEK 23638
  40 PRINT "Linea 40: ";PEEK 236
37+256*PEEK 23638
```

O sea, iterar **n** de 1 a 3, e ir mostrando el valor de NXTLIN para cada línea. El resultado no nos debería sorprender:

```
Linea 10: 23828
Linea 20 (1): 23890
Linea 30: 24004
Linea 20 (2): 23890
Linea 30: 24004
Linea 20 (3): 23890
Linea 30: 24004
Linea 30 despues 24004
Linea 40: 24058
```

Notemos que, en el caso de la línea 30, siempre dio el mismo valor (un número mayor al de las líneas anteriores), independientemente de si debía ejecutarse luego la línea 20 (**n** no llegó a 3) o la instrucción luego del NEXT N (cuando se cumplieron las tres iteraciones). Como podemos suponer, el valor mostrado en la línea 30 es la posición en memoria donde comienza la línea 40 en todos los casos.

DATADD (23639, 2 bytes, valor inicial 23754)
Esta variable es importante cuando estamos leyendo datos mediante la instrucción READ. Apunta a la dirección en memoria donde está el byte inmediatamente después del último elemento leído. Normalmente en esa posición encontraremos la coma (si hay varios elementos después de una instrucción DATA) o el ENTER (fin de línea).
Veamos el siguiente ejemplo:

```
  10 FOR n=1 TO 15
  20 READ a
  30 PRINT "Iteracion ";n;": ";T
AB 14;a;
  40 LET p=PEEK 23639+256*PEEK 2
3640
  50 PRINT TAB 18;p;TAB 26;PEEK
p;TAB 30;CHR$ PEEK p
  60 NEXT n
  70 PRINT "Fin de proceso"
1000 DATA 3,5,7,9,11,13,15,17,19
,21
1010 DATA 23,25,27,29,31
```

Leemos 15 datos, y en cada iteración anotamos el valor de la variable DATADD, el byte que encontramos en esa posición y el caracter que representa dicho byte. A propósito separamos los datos en un primer grupo de 10 elementos seguido de otro con 5. El resultado obtenido es:

```
Iteracion 1:   3   23952   44   ,
Iteracion 2:   5   23960   44   ,
Iteracion 3:   7   23968   44   ,
Iteracion 4:   9   23976   44   ,
Iteracion 5:   11  23985   44   ,
Iteracion 6:   13  23994   44   ,
Iteracion 7:   15  24003   44   ,
Iteracion 8:   17  24012   44   ,
Iteracion 9:   19  24021   44   ,
Iteracion 10:  21  24030   13

Iteracion 11:  23  24044   44   ,
Iteracion 12:  25  24053   44   ,
Iteracion 13:  27  24062   44   ,
Iteracion 14:  29  24071   44   ,
Iteracion 15:  31  24080   13

Fin de proceso
```

Notemos que luego de leer cada dato esta variable quedó apuntando siempre a la siguiente coma o al fin de línea. También notemos que, como siempre que hay números, aparte de la representación en ASCII se guarda en el programa el byte 14 y los cinco bytes representando el número para el sistema. Por eso vemos que la variable salta de a 8 bytes cuando el número leído es de una cifra (la coma, la cifra, el byte 14 y los cinco bytes con la representación) y pasa a hacerlo de a 9 cuando tenemos dos cifras. El salto de 14 cifras en la iteración 11 se explica porque entre el décimo y undécimo valor está el número de la nueva línea 1010 y el token DATA.

E LINE (23641, 2 bytes, valor inicial 23756)
Dirección del comando que se está escribiendo en el teclado. Apunta a la zona de edición, y su valor cambia a medida que vamos modificando el programa en Basic o declaramos variables, ya que esta zona se encuentra inmediatamente después de la definición del programa y la definición de variables. En la descripción de la variable DEST (posición 23629) podemos ver un programa que muestra cómo va cambiando de valor a medida que declaramos nuevas variables. Podemos hacer, como segundo ejemplo, un pequeño truco para demostrar que esta variable apunta a lo que se esté escribiendo por teclado. Ejecutemos lo siguiente en modo inmediato (no como parte de un programa sino que directamente):

```
FOR n=0 TO 100: PRINT CHR$ PEEK
((PEEK 23641+256*PEEK 23642)+n),
: NEXT n
```

El resultado será el siguiente:

```
 FOR                n
=                   0
?                   ?
?                   ?
?                   ?
TO                  1
0                   0
?                   ?
?                   d
?                   ?
:                   PRINT
CHR$                PEEK
(                   (
PEEK                2
3                   6
4                   1
?                   ?
?                   Y
/                   ?
+                   2
5                   6
?                   ?

scroll?
```

Vemos, uno por uno, los bytes que forman la instrucción inmediatamente ejecutada. Los caracteres "**?**" que aparecen se deben a que el sistema pone este signo para mostrar como texto los códigos menores a 32. En este caso, estamos observando las representaciones de los valores numéricos enteros entre 0 y 65535 (en esta instrucción son todos así), que son secuencias de 5 bytes armadas con dos 0, dos bytes con el valor y un 0 al final.

K CUR (23643, 2 bytes, valor inicial 23637)

Esta variable del sistema está relacionada con E LINE, la variable anterior. Guarda la dirección en memoria donde se encuentra el cursor mientras editamos una línea. Si no estamos editando texto, el valor almacenado será idéntico al que contenga E LINE. No es mucho lo que podemos hacer, el valor inicial cambia apenas presionamos la primera tecla. Si presionamos la tecla P (aparece el token PRINT) y sacamos una foto de la memoria veremos que el nuevo valor pasó a ser 23757, lo cual es lógico si en 23756 está el comienzo de la línea de edición (ver E LINE).

CH ADD (23645, 2 bytes, valor inicial 23779)

Recordemos que Basic es un lenguaje interpretado. Cuando se está ejecutando un programa, esta variable contendrá la dirección en memoria donde encontrar el próximo caracter a interpretar. Podemos intentar el siguiente programa:

```
  10 LET a=PEEK 23645+256*PEEK 2
3646
  20 PRINT a
  30 LET b=PEEK 23645+256*PEEK 2
3646
  40 PRINT b
  50 PRINT CHR$ PEEK a
  60 PRINT CHR$ PEEK b
```

Veremos que la salida nos muestra la dirección donde se encuentra el signo "+" en las líneas 10 y 30:

```
23774
23824
+
+
```

Quizás no es la salida que esperamos que el programa nos devuelva, uno esperaría el Enter que termine las líneas 10 y 30 en lugar de el signo "+" en el medio de las mismas. Como esto no termina de clarificar la forma de utilizar esta variable, probaremos otro enfoque:

```
  10 DEF FN f()=PEEK 23645+256*P
EEK 23646
  20 DEF FN g()=PEEK 23645-0+256
*PEEK 23646
  30 LET a=FN f()
  40 PRINT a;TAB 10;PEEK a;TAB 2
0;CHR$ PEEK a
  50 LET b=FN g()
  60 PRINT b;TAB 10;PEEK b;TAB 2
0;CHR$ PEEK b
```

Las funciones f() y g() definidas son, técnicamente, iguales: restar 0 (función en línea 20) no debería alterar el resultado. Sin embargo, la salida del programa ahora es distinta:

```
23776      43          +
23821      45          -
```

La explicación es la siguiente: cuando se evalúa PEEK 23645 el siguiente caracter que sigue en la expresión es el signo "+" en la línea 10 y el signo "-" en la línea 20. Al final, en ambos casos, se evalúa PEEK 23646, se lo multiplica por 256 y finalmente se efectúa la suma con el valor anterior. Como prueba final, podemos ver el programa:

```
  10 DEF FN f()=256*PEEK 23646+P
EEK 23645
  20 DEF FN g()=PEEK 23645+256*P
EEK 23646
  30 LET a=FN f()
  40 PRINT a;TAB 10;PEEK a;TAB 2
0;CHR$ PEEK a
  50 LET b=FN g()
  60 PRINT b;TAB 10;PEEK b;TAB 2
0;CHR$ PEEK b
```

La diferencia con el anterior, es que hemos intercambiado las direcciones que miramos en la línea 10 y eliminamos la resta de 0 de la línea 20. La salida será:

```
23799      13

23821      43          +
```

Podemos ver que ahora estamos apuntando, en el primer caso, al Enter final que termina la línea 10. De hecho, la salida del programa muestra una línea de separación que es el salto de línea provocado por este carácter 13. La explicación es simple: el siguiente caracter, luego de PEEK 23645 es el fin de la línea 10 del programa. Al evaluar PEEK 23645, y en ese preciso momento, la variable del sistema CH ADD apunta al fin de línea.

X PTR (23647, 2 bytes, valor inicial 0)
Dirección del lugar exacto donde el intérprete Basic encontró un error cuando se intentó introducir una nueva línea de programa. Es la posición del caracter que sigue al signo "?" que se muestra en tal caso de error. No es mucho lo que podemos hacer con esta variable, ya que si toma algún valor es porque está funcionando el editor del sistema y no está corriendo un programa nuestro.

WORKSP (23649, 2 bytes, valor inicial 23781)
Dirección del espacio temporal de trabajo. Este espacio es utilizado por las operaciones que requieran que se guarde un valor en memoria por un período corto. Por ejemplo, la información de cabecera de un bloque guardado en cinta se almacena en este espacio temporal.

STKBOT (23651, 2 bytes)
Dirección del fondo de la pila de cálculo. O sea, dónde comienza la pila que se utiliza para guardar los valores de una expresión a la espera de que sean evaluados.

STKEND (23653, 2 bytes, valor inicial 23781)
Dirección del fin de la pila de cálculo, y comienzo del espacio de reserva. Es interesante porque utilizando esta variable junto con la variable RAMTOP (23730) podemos hacernos una idea del espacio libre en memoria. En el primer cuadro del capítulo 1 podemos ver el mapa de memoria, y podemos deducir que si efectuamos la operación ((RAMTOP) + 256 x (RAMTOP+1)) - ((STKEND) + 256 x (STKEND+1)) obtendremos la memoria libre en sistema. Esto es "casi" cierto,

y lo notamos a simple vista al examinar cómo está formada y cómo funciona la instrucción que ejecutaríamos en modo comando para realizar el cómputo:

```
PRINT (PEEK 23730+256*PEEK 23731
)-(PEEK 23653+256*PEEK 23654)
```

Para realizar la operación, valores como el contenido de las cuatro posiciones de memoria examinadas y resultados intermedios irán a la pila de cálculo, y los operadores también ocuparán espacio al momento de su evaluación. Por lo tanto, el resultado obtenido no será del todo exacto respecto al verdadero tamaño de la memoria libre en el momento de ser evaluada esta expresión.

BREG (23655, 1 byte)
Registro B del calculador. Se utiliza principalmente en la rutina CALCULATE de la ROM (dirección 335Bh. 13147 en decimal), y no está relacionada con el registro B del Z80.

MEM (23656, 2 bytes)
Dirección de la zona utilizada para la memoria del calculador. (Normalmente MEMBOT, pero no siempre).

FLAGS2 (23658, 1 byte, valor inicial 16)
Otra variable que, como FLAGS1 (dirección 23611), debe ser vista en realidad como un grupo de 8 bits, donde se guardan indicadores de control utilizados por el Basic. En este caso tenemos:
Bit 0: Se pone en 0 para indicar que se limpió la pantalla, cuando se ejecuta la rutina CL-ALL en ROM (dirección 0DAFh, 3503 en decimal).
Bit 1: Si hay datos para imprimir en el buffer de impresión.
Bit 2: Si el cursor está entre comillas
Bit 3: Estado del Caps Lock. Si está en 1, entonces el cursor será C y sino será L.
Bit 4: Si se está utilizando el canal K (dos líneas inferiores de pantalla) para mostrar mensajes de error.

DF SZ (23659, 1 byte)
Número de líneas de la parte inferior de la pantalla, donde se muestran los mensajes del sistema. De hecho, si ponemos el valor 0 a esta variable podemos implementar una buena protección de software, ya que se colgará el sistema cuando quiera imprimir cualquier mensaje y no encuentre espacio para hacerlo.

S TOP (23660, 2 bytes)
Contiene el número de línea de la primera línea de programa que debe ser listado cuando se ejecuta la instrucción LIST o se ejecutan los listados automáticos.

OLDPPC (23662, 2 bytes)
Acá se guarda el número de línea del programa que se estaba ejecutando cuando se detuvo porque el usuario hizo BREAK o porque encontró una instrucción STOP. Indica dónde seguir la ejecución del programa si se escribe la sentencia CONTINUE.

OSPCC (23664, 1 byte)
Asociada a la variable anterior, indica el número de sentencia dentro de la línea interrumpida por BREAK o STOP adonde la ejecución seguirá cuando se escriba la instrucción CONTINUE.

FLAGX (23665, 1 byte, valor inicial 0)
Indicadores varios, pero asociados a la forma en que el sistema ejecuta comandos INPUT.
Bit 0: Se pide el valor para una nueva variable, así que hay que eliminar el valor anterior
Bit 1: Si se está pidiendo el valor para una nueva variable
Bit 5: Indica, cuando vale 1, si se está en modo INPUT. Si se está en modo de edición, vale 0.
Bit 6: Si se está pidiendo un string (valor en 1) o un número (valor en 0)
Bit 7: Si estamos ejecutando un INPUT LINE

De hecho, una rutina en código máquina podría poner en 0 este valor, forzando a la computadora a salir del modo INPUT.

STRLEN (23666, 2 bytes)
Acá se guarda o bien la longitud de una variable de tipo cadena que se está utilizando en el momento o, para variables numéricas o una nueva variable de tipo string, el byte menos significativo tendrá la letra que identifica la variable.

T ADDR (23668, 2 bytes)
Dirección del siguiente elemento en las tablas de sintaxis que se encuentran a partir de la dirección 6728 (1A48h) de la ROM. No tiene mucha utilidad para programas de usuarios, y es usada para otros propósitos por algunas rutinas en ROM. Por ejemplo: acceso a cinta (LOAD, SAVE, MERGE, VERIFY) o manejo del color.

SEED (23670, 2 bytes, valor inicial 0)
Guarda lo que se denomina la "semilla" que se usa para calcular la función RND, la cual devuelve un número "aleatorio" entre 0 y 1. Una fórmula obtiene a partir del valor en SEED el valor para devolver con RND, y el mismo valor devuelto se utilizará como nueva semilla que se guarda en esta variable. ¿Cómo se puede transformar un valor en punto flotante entre 0 y 1 en un valor decimal entre 0 y 65535? Porque en realidad primero se calcula la nueva semilla y el nuevo valor es el que finalmente se emplea para determinar el número "al azar" devuelto. Para generar estos números "aleatorios" se utiliza una variación del método de Lehmer[25], quien concibió la fórmula:
$S_{k+1} = A \times S_k \bmod N$
O sea, que la siguiente semilla surja de la multiplicación de la semilla actual por un valor A y a ese resultado lo divida por N, quedándose con el resto. Obviamente que A y N no son valores cualquiera, sino que se exige que N sea un número primo y que A sea un número tal que sea una raíz primitiva módulo N[26]. Quien quiera profundizar el tema puede leer las referencias al pie de página, pero en este caso y por razones de simplicidad nos limitaremos a decir que en la ZX Spectrum los valores utilizados son A=75 y N=65537, y la fórmula se alteró levemente a la siguiente:
$S_{k+1} = (75 \times (S_k + 1) - 1) \bmod 65537$
Dado que el valor inicial de esta variable del sistema es 0, podemos calcular el siguiente valor que se tomará:
$S_1 = (75 \times (S_0 + 1) - 1) \bmod 65537 = (75 \times (0 + 1) - 1) \bmod 65537 = (75-1) \bmod 65537 = 74$
El valor que siga a este recién obtenido será:
$S_2 = (75 \times (S_1 + 1) - 1) \bmod 65537 = (75 \times (74 + 1) - 1) \bmod 65537 = (5625-1) \bmod 65537 = 5624$
Así se sigue con el resto de los valores. Para obtener el valor "al azar", la rutina divide el valor de la nueva semilla por 65536. Si consideramos que el valor en SEED siempre es un entero entre 0 y 65535[27], podemos comprender por qué al dividir por 65536 obtendremos siempre un valor mayor o igual a 0 y estrictamente menor que 1.
La función en ROM que calcula estos valores se llama S-RND y se ubica en la dirección 9720 (25F8h).
Vamos a mostrar con un programa de ejemplo el comportamiento de esta función. Para asegurar que arrancamos con SEED=0 (el valor con que se inicia esta variable al prender la computadora), utilizaremos la sentencia POKE para poner los dos bytes que forman el valor de SEED en 0.

---

[25] Una explicación del método la encontraremos en http://wapedia.mobi/en/Park-Miller_function.
[26] Ver http://es.wikipedia.org/wiki/Ra%C3%ADz_primitiva_m%C3%B3dulo_n
[27] Por si alguien se lo plantea, un resto de dividir por 65537 podría ser 65536 (valor imposible de almacenar en los dos bytes ocupados por SEED), pero en este caso nunca ocurre que la cuenta $(75 \times (S_k + 1) - 1) \bmod 65537$ arroje ese valor. Igualmente, si llegara a ocurrir, la rutina que calcula la nueva semilla, entre los pasos que ejecuta, efectúa un llamado a la función FP-TO-BC ubicada en la dirección 11682 (2DA2h) de la ROM, que toma cualquier número en formato [mantisaexponente] y lo transforma a entero de dos bytes que devuelve en el registro BC del Z80, avisando de eventuales desbordamientos con indicadores (flags) del registro F.

```
  10 DEF FN p()=PEEK 23670+256*P
EEK 23671
  20 POKE 23670,0: POKE 23671,0
  30 PRINT "Nro  SEED     RND
      Sgte."
  40 FOR n=0 TO 9
  50 PRINT TAB 1;n;TAB 5;FN p();
  60 LET a=RND
  70 PRINT TAB 14;a;TAB 27;a*655
36
  80 NEXT n
```

Al ejecutar el programa, obtendremos la salida, no importa cuántas veces lo hagamos correr:

```
Nro   SEED      RND             Sgte.
 0    0         .0011291504    74
 1    74        .08581543      5624
 2    5624      0.43719482     28652
 3    28652     0.79025269     51790
 4    51790     0.2691803      17641
 5    17641     0.18934631     12409
 6    12409     0.20188904     13231
 7    13231     0.14257813     9344
 8    9344      0.69433594     45504
 9    45504     .075531006     4950
```

Notemos en la cuarta columna que el valor devuelto por la función RND, multiplicado por 65536, coincide con la siguiente semilla que tendremos, lo que confirma lo que explicábamos al comienzo: que primero se calcula la próxima semilla y que luego se utiliza la misma para dividirla por 65536 y devolver ese valor como el nuevo valor "aleatorio".
Es evidente que podríamos conocer los números que van a ir saliendo. Para disminuir esta posibilidad está la instrucción RANDOMIZE en Basic, que si no recibe parámetro o si le pasamos el valor 0, toma y copia en SEED los dos bytes menos significativos de la variable del sistema FRAMES (dirección 23672) y ese valor es impredecible para nosotros porque se va incrementando de a 1 cada 20 milisegundos (o 17 milisegundos en la TS 2068) desde el momento en que se encendió la computadora.

FRAMES (23672, 3 bytes)
3 octetos (el menos significativo en primer lugar) del contador de cuadros de pantalla. Se incrementa 50 veces por segundo en la ZX Spectrum, cada vez que se completa el refresco de pantalla. En la TS 2068 esta frecuencia es de 60 veces por segundo. Una utilidad de esta variable es el poder utilizarla como reloj, ya que si tomamos los valores en diferentes momentos podremos ver el tiempo transcurrido entre uno y otro.

UDG (23675, 2 bytes, valor inicial 65368)
Dirección del primer gráfico definido por el usuario (GDU, UDG en inglés y de ahí el nombre de la variable). Originalmente se definen 21 GDU, y en el capítulo 1 hemos visto cómo los interpreta la computadora. Se puede cambiar el valor almacenado en esta variable, por ejemplo, para ganar espacio en memoria pero a costa de tener menos GDU disponibles. Sin embargo también podemos alterarla para "aumentar" la cantidad de GDU. En el número 42 de la revista MicroHobby, página 14, encontraremos un truco que modifica el contenido de esta variable para duplicar o triplicar la cantidad original de 21 GDU disponibles[28], aunque solamente se pueden utilizar de a bloques de 21 GDU a la vez (porque en realidad se apunta la variable UDG a distintos bloques de memoria, pero la computadora siempre los ve como únicos 21 GDU).

COORDS (23677, 2 bytes)
Coordenadas (x,y) del último punto trazado. Son dos variables; la primera es la coordenada **x**, la segunda es la coordenada **y**.

[28] El contenido de la revista se puede leer desde http://www.microhobby.org/numero042.htm

P POSN (23679, 1 byte, valor inicial 33)
Número que indica la posición (columna) del buffer de impresora. Trabaja de esta forma: originalmente toma el valor 33 y se va decrementando a medida que vamos enviando caracteres mediante instrucciones LPRINT. De hecho, el valor es 33-P, donde P es la posición y toma los valores de 0 a 31.

PR CC (23680, 1 byte, valor inicial 0)
Dirección de la siguiente posición a imprimir en la instrucción LPRINT (en la memoria temporal de la impresora). Se combina como byte menos significativo con la dirección 23681, indicando así la ubicación en memoria del byte que define los 8 puntos de la fila superior del caracter a imprimir. Se puede cambiar con POKE, y se alterará la posición del próximo caracter a imprimir. Esto último siempre que también cambiemos el contenido de la variable P POSN para que sea coherente con el nuevo valor de PR CC, sino va a parecer que funciona bien pero al final de la línea habrá problemas.

Variable sin uso (23681, 1 byte, valor inicial 91)
Esta posición de memoria no tiene utilidad para el usuario y contiene el valor 91. No casualmente (ver la variable anterior PR CC) este valor 91 es el mismo que el del byte más significativo de la dirección de comienzo del buffer de impresora (dirección 23296) y si bien nosotros lo podemos alterar con POKE, el mismo se restablecerá automáticamente a 91 cuando ejecutemos cualquier operación con la impresora.

ECHO E (23682, 2 bytes)
En realidad dos variables en una, almacena el número de columnas que restan para llegar a la derecha de la pantalla, y el número de líneas restantes para llegar al final de la memoria temporal de entrada.

DF CC (23684, 2 bytes)
Almacena la dirección de la línea superior de pixeles para la próxima posición de PRINT. Se puede utilizar para cambiar la ubicación de los caracteres impresos normalmente, pero puede causar efectos inesperados.

DFCCL (23686, 2 bytes)
Igual que DFCC, pero para la mitad inferior de la pantalla.

S POSN (23688, 2 bytes)
Indica en sus dos bytes la posición del próximo caracter a imprimir. El primer byte es el número de columna y el segundo es el número de línea.

SPOSNL (23690, 2 bytes)
Igual que S POSN, pero para la mitad inferior.

SCR CT (23692, 1 byte)
Cuenta de los desplazamientos hacia arriba ("scroll"): es siempre una unidad más que el número de desplazamientos que se van a hacer antes de terminar en un scroll. Si se cambia este valor con un número mayor que 1 (digamos 255), la pantalla continuará "enrollándose" sin necesidad de nuevas instrucciones.

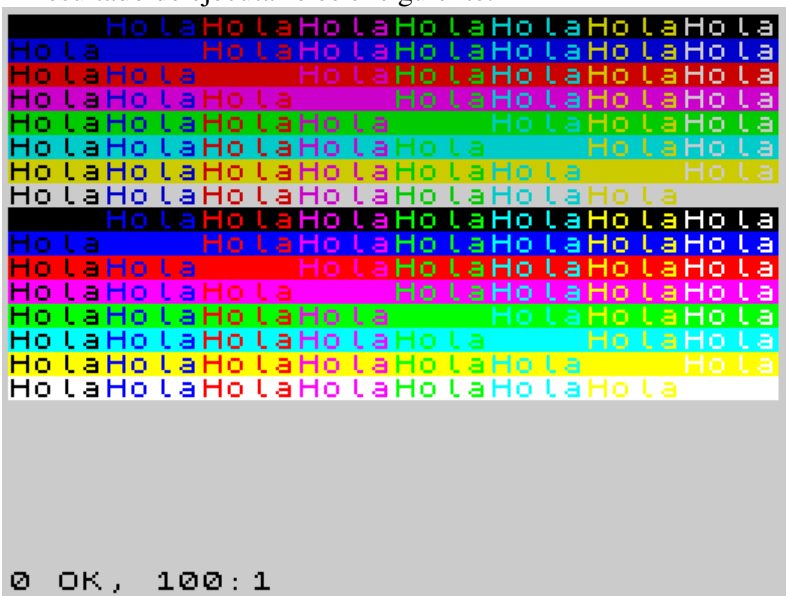
ATTR P (23693, 1 byte, valor inicial 56)
Colores y atributos permanentes en curso, tal y como los fijaron las sentencias para el color. El valor contenido varía cuando nosotros especificamos PAPER, INK, FLASH o BRIGHT. Los ocho bits guardan la información en una estructura idéntica a la que vimos para la variable del byte es como hemos visto en la variable BORDCR (dirección 23624):

F	B	fondo	tinta

Vamos a probar este programa, que imprime la palabra “Hola” repetidamente en pantalla con diferentes colores.

```
 10 FOR n=0 TO 127
 20 POKE 23693,n
 30 PRINT "Hola";
 40 NEXT n
100 POKE 23693,56
```

El resultado de ejecutarlo es el siguiente:



El hecho de tener una palabra a mostrar por pantalla de cuatro letras permite repetir la misma ocho veces para completar una línea de exactamente 32 caracteres y saltar a la siguiente. El efecto obtenido por esta “casualidad” entonces es que para cada línea el valor de los tres últimos bits unidos de la variable va desde el 0 (000) hasta el 7 (111), cambiando el color de la letra. A su vez, las primeras ocho líneas mostrarán la variación de los bits 3 a 5 para cambiar el color de fondo desde negro hasta blanco, y para cada línea cambiarán los bits 0 a 2 determinando el color de la letra, también desde negro hasta blanco. Las siguientes ocho líneas muestran el mismo patrón, pero ahora el bit 6 de brillo está en 1 y por eso el resultado es más claro.
Invitamos al lector a extender el valor de n hasta 255 y probar el resultado.
La última línea del programa cambia el valor almacenado a 56; es decir que pone los bits de 3 a 5 en 1 y los demás en 0. Con esto se indican los valores FLASH 0, BRIGHT 0, PAPER 7, INK 0. Si agregamos una sentencia STOP antes de que se ejecute este POKE (por ejemplo insertando una línea con el número 50) y volvemos a correr el programa, notaremos que el listado del programa no se puede leer por estar tanto la letra como el fondo del mismo color blanco.

MASK P (23694, 1 byte)
Se usa para los colores transparentes, etc. Cualquier bit que sea 1 indica que el bit correspondiente de los atributos no se toma de ATTR P (dirección 23693), sino de lo que ya está en pantalla.

ATTR T (23695, 1 byte)
Atributos de uso temporal, con la misma estructura de byte que en ATTR P (dirección 23693). Son los que ponemos en una instrucción PRINT cuando precisamos imprimir con colores que no son los fijados actualmente como los que están en curso.

MASK T (23696, 1 byte)
Igual que MASK P, pero temporal.

P FLAG (23697, 1 byte, valor inicial 0)
Más indicadores para la salida por pantalla. Son los que empleamos cuando utilizamos los modificadores OVER e INVERSE, o el color especial 9 para INK y PAPER (este color no existe, sino que en realidad se puede poner para indicar que se deja al sistema elegir el color más conveniente para la mejor legibilidad). Dado que son solamente cuatro atributos que ocupan un bit cada uno, se decidió unir los permanentes con los temporarios, siendo utilizados del mismo modo que el sistema utiliza los atributos ATTR P y ATTR T. O sea:
Bit 0: OVER, para una salida temporal como ATTR T
Bit 1: OVER, para uso permanente como ATTR P
Bit 2: INVERSE, para salida temporal
Bit 3: INVERSE, para uso permanente
Bit 4: INK 9, uso temporal
Bit 5: INK 9, uso permanente
Bit 6: PAPER 9, uso temporal
Bit 7: PAPER 9, uso permanente

MEMBOT (23698, 30 bytes)
Zona de memoria destinada al calculador; se usa para almacenar números que no pueden ser guardados convenientemente en la pila del calculador.

NMIADD (23728, 2 bytes, valor inicial 0)
Dirección adonde se ubica la rutina de atención de una interrupción no enmascarable (NMI). En la dirección de memoria 0066h (102d) de la ROM está programado qué hacer en caso de ocurrir una NMI. Se supone que ahí se controlaría que el valor en esta variable NMIADD sea distinta de 0 (lo que señalaría la existencia de la rutina de atención, porque saltar a la dirección 0 es reiniciar la computadora) y que en caso de encontrar un valor diferente a 0 se saltaría a esta dirección señalada por la variable. Por error, en la ROM se programó que el salto se haga si la dirección es cero en vez de ser distinta a cero; es por eso que esta variable no tiene función alguna y en muchos textos (incluso los manuales de usuario) aparece como “no utilizada”. Este error de programación fue corregido en el Spectrum +3.

RAMTOP (23730, 2 bytes, valor inicial 65367)
Dirección del último octeto del BASIC en la zona del sistema. Como vimos en el ejemplo del capítulo 1 cuando analizábamos el espacio de reserva en memoria luego de la pila de GOSUB, podemos alterar este valor mediante la instrucción CLEAR. En la dirección apuntada por esta variable de sistema aparece el valor 62, que indica fin de la zona donde se guarda el programa en Basic junto con los datos que utilice y comienzo de la memoria reservada, que no se borrará si hacemos NEW.

P-RAMT (23732, 2 bytes, valor inicial 32767 o 65535)
Dirección del último octeto de la RAM física. El valor inicial cambia si se trata del modelo de 16KB o 48KB de la Spectrum. La explicación es lógica: si la Spectrum tiene 16KB de memoria ROM y 16KB de RAM, al contabilizarlos juntos tenemos un total de 32KB, y estos ocupan las posiciones 0 hasta 32767. Si en cambio tenemos 48KB de RAM el total de memoria es 64KB, ocupando las posiciones 0 hasta 65535. En realidad, esta variable guarda la posición del último octeto verificado en buen funcionamiento al encender la computadora; si alguno de los chips de memoria estuviera estropeado el valor almacenado sería inferior.

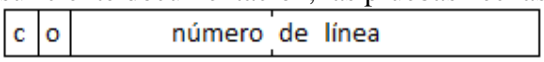
Cap. 5 - Vista en detalle: Las variables del Sistema en la TS 2068

Analizaremos las variables del sistema que son de uso exclusivo en la Timex Sinclair 2068. Esta computadora comparte, como dijimos anteriormente, todas las variables del sistema empleadas en la ZX Spectrum pero agrega algunas, para las funcionalidades exclusivas en este modelo.

Las primeras cuatro variables que explicaremos están relacionadas entre sí:

ERRLN (23734, 2 bytes, valor inicial 0)

Esta variable debe ser vista como una cadena de 16 bits (como es esperable, los bits 0 a 7 se guardan en la posición 23734 y los bits 8 a 15, en la posición 23735). Si bien no se encontró suficiente documentación, las pruebas hechas mostraron el siguiente patrón de uso:



O sea, el bit 15 indica que se está haciendo control de errores. El bit 14 se pone en 1 cuando ocurre una condición de error. Y los restantes catorce bits indican el número de línea a donde saltar en caso de error de programa Basic[29].

ERRC (23736, 2 bytes, valor inicial 0)

Número de línea donde ocurrió un error de programa Basic

ERRS (23738, 1 byte, valor inicial 0)

Número de sentencia que no funcionó bien dentro de la línea donde ocurrió el error de programa Basic

ERRT (23739, 1 byte, valor inicial 0)

Código de error ocurrido. Como veremos más adelante, no es necesariamente coincidente con los códigos que devuelve el sistema cuando ocurre un error en la ejecución de un programa o comando en Basic.

Para comprender cómo se utilizan estas cuatro variables, vamos a mostrar ejemplos de utilización de la instrucción ON ERR, que es una de las extensiones al Basic de la ZX Spectrum que se agregaron en la TS 2068. Si bien no tenemos por objetivo de este texto el explicar instrucciones en Basic, creemos que es interesante hacerlo en este caso. Esto ayudará, a quienes no tuvieron acceso a la TS 2068, a entender mejor este capítulo.

La sintaxis de la instrucción ON ERR tiene tres posibilidades de estructura:

```
ON ERR GO TO [Nro.Línea]
```

En caso de encontrarse un error durante la ejecución del programa, se salta a la línea indicada

```
ON ERR CONTINUE
```

En caso de encontrarse un error, el mismo es ignorado

```
ON ERR RESET
```

Si existía un ON ERR CONTINUE o un ON ERR GO TO, ahora se deshabilita esta función y en caso de error el programa terminará de la forma habitual, mostrando el mensaje que corresponda al problema encontrado.

Así entonces, vamos a probar el siguiente programa:

```
 10 ON ERR GO TO 100
 20 POKE 23610,0
 30 PRINT "Esta es una prueba"
 40 PRINT PEEK 23610
100 PRINT "Fin"
```

Cuando lo ejecutamos, obtendremos la salida:

```
Esta es una prueba
0
Fin
Fin
```

[29] 2^{14}=16384, por lo tanto alcanza y sobra para el límite de 9999, el mayor número de línea permitido en Basic.

La explicación es la siguiente: la computadora sabe que en caso de encontrarse un error debe ir a la línea 100. En la línea 20 introducimos un valor distinto a 255 en la variable del sistema ERR NR, lo que hace que no se interrumpa la ejecución del programa pero se engaña a la computadora haciéndole creer que hay un código de error. Luego se muestran el texto de la línea 30 y el valor puesto en la dirección de ERR NR. Por último, sale por pantalla el texto de la línea 100, y cuando el programa está por terminar su ejecución ve la condición de error y por ese motivo vuelve a ir a la línea 100. Así entonces se vuelve a mostrar el texto de la línea 100, y como esta instrucción no provoca error; la computadora termina de ejecutar el programa normalmente.
Si eliminamos la línea 10, veremos que el programa termina su ejecución pero mostrando una sola vez la palabra "Fin" y con código de error 1 (NEXT without FOR).

El próximo programa demostrará cómo cambian los valores de estas cuatro variables según establezcamos o no rutinas de control de errores, o qué pasa cuando sucede un error:

```
  10 LET c=0: GO SUB 1000
  20 ON ERR GO TO 500
  30 GO SUB 1000
  40 ON ERR RESET
  50 GO SUB 1000
  60 ON ERR CONTINUE
  70 GO SUB 1000
  80 ON ERR GO TO 500
  90 PRINT 1/0
 100 GO TO 2000
 500 PRINT "Error codigo: ";PEEK
 23739
 510 GO SUB 1000
 520 GO TO 2000
1000 LET c=c+1
1010 PRINT "Volcado de control "
;c
1020 FOR n=23734 TO 23739
1030 PRINT n;": ";PEEK n,
1040 NEXT n
1050 RETURN
2000 ON ERR RESET
2010 PRINT "Fin de ejecucion"
```

Se van cambiando las formas de tratamiento de error y se hace un volcado de control de las variables de sistema relacionada con los errores. La variable Basic c nos indica el momento de cada volcado: 1 (condición normal), 2 (luego de marcar la línea 500 para control de errores), 3 (eliminamos el control de errores, el Basic procederá como normalmente hace), 4 (que no se tome el error), 5 (volver a marcar la línea 500 para control de errores y provocar un error).
El resultado obtenido es el siguiente:

```
Volcado de control 1
23734: 0          23735: 0
23736: 0          23737: 0
23738: 0          23739: 0
Volcado de control 2
23734: 244        23735: 129
23736: 0          23737: 0
23738: 0          23739: 0
Volcado de control 3
23734: 244        23735: 1
23736: 0          23737: 0
23738: 0          23739: 0
Volcado de control 4
23734: 244        23735: 1
23736: 0          23737: 0
23738: 0          23739: 0
Error codigo: 6
Volcado de control 5
23734: 244        23735: 193
23736: 90         23737: 0
23738: 1          23739: 6
Fin de ejecucion

0 OK, 2010:1
```

Al comienzo están todos los valores en 0. Cuando establecemos que en caso de error se deberá saltar a la línea 500, la variable ERRLN toma el valor de (256x129)+244=33268. Notemos que 33268=32768 (bit 15 en 1) + 500 (número de línea 500); o sea, que se controlará error y que en caso de ocurrir se deberá ir a la línea 500. Luego eliminamos el control de errores (ON ERR RESET) y el bit 15 se pone en 0, quedando solamente el valor 500 en ERRLN. Poner que se ignore el error (ON ERR CONTINUE) no tiene efecto sobre estas variables, tal como eran los valores anteriores (pondría en 0 el bit 15 si antes estaba en 1). Y finalmente, volvemos a decir que queremos que se controlen errores e inmediatamente provocamos un error (división por 0), lo que muestra el último volcado. Veremos ahora que el valor de ERRLN es (256x193)+244=49652. Visto en bits, notaremos que están en 1 los bits 15 (controlar error) y 14 (error ocurrido). A la vez, ahora la variable ERRC (2 bytes desde la dirección 23736) guarda el valor 90, indicando el número de línea que provocó el error. Con ERRS (1 byte en la dirección 23738) sabemos que dentro de la línea 90, fue la primera sentencia. Y, finalmente, con ERRT (1 byte en la dirección 23739) sabemos que el error fue de código 6.

Ahora plantearemos un programa que permite ver diferentes códigos de error:

```
  10 ON ERR GO TO 1000
  20 LET nl=(PEEK 23734+256*PEEK
 23735)-32768: REM Demostrar que
 nos queda el valor 1000 antes i
ndicado
  30 PRINT "La rutina de control
 de errores esta en la linea ";n
l
  40 PRINT
  50 PLOT 1000,1000
  60 DIM x(10): LET x(11)=1
  70 GO SUB 70
  80 RETURN
  90 PLOT 10,10: DRAW 300,300
 100 LET c=(1e10)↑100
 110 NEXT n
 120 POKE 10,10000
 130 PRINT d
 140 LET a=1/0
 150 READ a$
 160 POKE 23610,4
 900 GO TO 1100
1000 REM Rutina de control
1010 LET l=PEEK 23736+256*PEEK 2
3737: REM Linea con error
1020 LET s=PEEK 23738: REM Sente
ncia con error
1030 LET c=PEEK 23739: REM Codig
o de error
1040 PRINT "Error con codigo ";c
'"Sentencia ";s;" de la linea ";
l
1050 GO TO l+1: REM seguir la ej
ecucion
1100 ON ERR RESET
1110 PRINT '"Fin de ejecucion de
 programa"
```

Puntualmente, establecemos la rutina de control de errores en la línea 1000 (que lo único que hará es un volcado de las variables y pasar a la siguiente línea), mostramos cómo obtener el número de línea de control (restando 32768 a ERRLN, ya que es como poner en 0 el bit 15) y provocamos diferentes condiciones de error.
Al ejecutar este programa, obtendremos la salida:

```
La rutina de control de errores
esta en la linea 1000

Error con codigo 11
Sentencia 1 de la linea 50
Error con codigo 3
Sentencia 2 de la linea 60
Error con codigo 4
Sentencia 1 de la linea 70
Error con codigo 7
Sentencia 1 de la linea 80
Error con codigo 11
Sentencia 2 de la linea 90
Error con codigo 6
Sentencia 1 de la linea 100
Error con codigo 2
Sentencia 1 de la linea 110
Error con codigo 11
Sentencia 1 de la linea 120
Error con codigo 2
Sentencia 1 de la linea 130
Error con codigo 6
Sentencia 1 de la linea 140
Error con codigo 14
Sentencia 1 de la linea 150

Fin de ejecucion de programa

5 Out of screen, 1110:1
```

Como se ve, los diferentes errores que le provocamos tienen códigos distintos según sea el error. Pero notablemente, no coinciden con los errores que normalmente nos muestra el sistema cuando una condición anómala ocurre. De hecho, hicimos una tabla con algunos de ellos según fuimos probando:

Código de error Basic	**Instrucción ejemplo**	**Código ERRT**
1 NEXT without FOR	NEXT n	2
2 Variable not found	PRINT d	2
3 Subscript wrong	DIM x(10): LET x(11)=1	3
4 Out of memory	70 GOSUB 70	4
5 Out of screen	PRINT AT 30,30	11
6 Number too big	LET c=1e10^100 PRINT 1/0	6
7 RETURN without GOSUB	RETURN	7
A Invalid argument	LET c=LN 0	10
B Integer out of range	PLOT 1000,1000	11

Dada la escasa documentación existente, no pudimos establecer el motivo de estas diferencias. Dejamos a consideración del lector el realizar nuevas pruebas

SYSCON (23740, 2 bytes, valor inicial 24298)
Puntero que indica la posición de memoria donde se ubica la Tabla de Configuración de Sistema. Vemos en el capítulo 2 que esta tabla es un conjunto de 278 bytes, que describían la memoria agregada en la computadora, permitiendo hasta 11 bancos. Si utilizáramos más bancos que estos 11, deberíamos crear una tabla más grande en otra ubicación y escribir en esta variable SYSCON la nueva dirección de memoria donde comienza[30].

[30] Nuevamente recomendamos la lectura del artículo de Wes Brzozowski, donde se explica detalladamente cómo desarrolló Timex este concepto, en http://8bit.yarek.pl/interface/ts.beu/wes-1.html.

MAXBNK (23742, 1 byte, valor inicial 0)
Cantidad de bancos de expansión en el sistema.

CURCBN (23743, 1 byte, valor inicial 0)
El nombre viene de la abreviación de "current channel bank number", o sea el número de banco actual. No es utilizada por la máquina sola ni por los cartuchos, sino exclusivamente cuando se agregan bancos de memoria.

MSTBOT (23744, 2 bytes, valor inicial 25088)
Ubicación del próximo byte luego de la pila de máquina. O sea, el comienzo del espacio reservado para ser usado por el distribuidor de funciones (function dispatcher). Cambia si se activa el video en alta definición ya que las posiciones entre 24576 (6000h) y 30719 (77FFh) son ocupadas en ese momento por la memoria secundaria de video.

VIDMOD (23746, 1 byte, valor inicial 0)
Modo de video. La documentación oficial dice que toma un valor distinto a cero si se utiliza el bloque 6000h-77FFh como memoria secundaria de video, pero no se especifica qué valor es. Si alteramos el valor de esta variable éste no se pierde pero el modo de video sigue siendo el mismo; algunos autores piensan que forma parte de algo que Timex planificó pero no fue hecho. Tampoco se da lo inverso, ya que no cambia el valor si activamos algún modo de video especial a través del puerto 255 mediante la instrucción OUT.

Espacio reservado (23748, 7 bytes)
Variables reservadas para el uso de cartuchos con programas

STRMNM (23755, 1 byte, valor inicial 0)
Número de canal de flujo de datos (stream) en uso para las instrucciones que utilizan canales. Podemos verificar la manera en que el sistema operativo utiliza esta variable con el siguiente programa:

```
10 FOR n=0 TO 3
20 PRINT #n;PEEK 23755
30 PAUSE 0
40 NEXT n
```

Veremos como se muestran por pantalla los valores que va tomando la variable STRMNM. Los canales estándar 0 y 1 se relacionan con la parte inferior de la pantalla, el canal 2 es la impresión regular en pantalla y el canal 3 es la impresora.
Notemos igualmente que podemos alterar con instrucciones POKE esta variable y el sistema operativo no lo tiene en cuenta salvo cuando se le dice explícitamente que trabaje con canales. Por ejemplo, en modo directo escribamos:

```
POKE 23755,20
```

E inmediatamente, si nosotros ahora solicitamos:

```
PRINT PEEK 23755
```

Obtendremos como respuesta en pantalla el valor 20 que pusimos previamente. Se supone que íbamos a ver el valor 2, que es el que identifica al canal asociado a la salida por pantalla, ya que es por ese medio que estamos viendo el resultado de nuestra petición. Hasta podía sonar lógico razonar en forma inversa y pensar en que llegaríamos a un error, ya que el canal 20 no está asociado a ningún dispositivo y estaríamos pidiendo una salida de texto teniendo este valor en STRMNM. En cambio, si escribimos:

```
PRINT #2;PEEK 23755
```

Ahí sí que podremos ver el valor 2 por pantalla.