REVISTA DE TECNOLOGÍA E INFORMÁTICA HISTÓRICA

Volumen 1 Número 1 2011

COMENTARIOS

i Presentación
Diego Gustavo Macchi

ARTÍCULOS

1 Conservación de plásticos y reversión de colores anómalos
Diego Gustavo Macchi

11 Memoria e introducción a las variables del sistema en la ZX Spectrum y TS 2068
Patricio Herrero Ducloux

30 Las variables del sistema de ZX Spectrum y TS 2068 en detalle
Patricio Herrero Ducloux

59 Programando la Commodore 64 en serio. Circuito de video y componentes básicos del sistema
Marcos Leguizamón

75 Programando la Commodore 64 en serio. Conexión de componentes y modernas aplicaciones
Marcos Leguizamón

ISSN 2250-5431

Revista de Tecnología e Informática Histórica
Volumen 1, Número 1, 2011

Diego G. Macchi
Editor responsable

Revista de Tecnología e Informática Histórica es una publicación periódica propiedad de la Fundación Museo ICATEC. Museo de Informática, Computadoras y Accesorios Tecnológicos.
Tucumán 810 1er. Subsuelo, CP C1049AAR, Capital Federal.

Registro intelectual en trámite

PRESENTACIÓN

La notable expansión de la informática en todos los aspectos de la vida cotidiana ha sido extraordinaria. No sólo ha creado una sociedad global sino que ha transformado nuestra forma de pensar y comunicarnos. Ciertamente, es virtualmente imposible en la actualidad concebir un mundo sin computadoras. Sin embargo, esta rápida expansión ha dejado de lado o no ha registrado adecuadamente otras facetas de su evolución al punto tal, que muchos objetos que la componen han sido destruidos desconociendo su verdadero valor histórico. De esta manera, los continuos avances técnicos tienden a hacernos olvidar rápidamente su propio pasado. Técnicas, conocimientos, usos y aún las representaciones simbólicas que se tienen de estos objetos, confluyen y forman parte del patrimonio cultural de un país. Es así, que no solamente desde un punto de vista técnico es importante conocer las capacidades y funcionalidades de cada objeto, sino también recuperar las historias de quienes estuvieron por detrás de ellas.

La Fundación Museo ICATEC se dedica a la preservación, documentación y exposición de la revolución informática y su impacto en la sociedad. Al preservar estos objetos con significancia histórica, se busca tener una mirada amplia y profunda sobre la evolución, los procesos históricos e implicancias sociales ligadas al desarrollo de una de las más grandes invenciones humanas, la computadora. El museo alberga la colección más grande de la Argentina de objetos vinculados a la historia de la computación, incluyendo hardware, software, documentación técnica, libros, propagandas y revistas, los que se encuentran disponibles para su uso e investigación. El museo ofrece una gran cantidad de información única con un catálogo en constante expansión y es un recurso importante para investigadores de la historia de la computación y público en general. A través de su colección, se busca capturar y explorar la evolución tecnológica y su contexto, además de las historias personales que a menudo, también corren el riesgo de perderse. La fundación cuenta con un plan activo de restauración de computadoras, que sirve para una mejor comprensión del hardware y software histórico, con especial acento en la producción nacional. Por otro lado, se llevan adelante programas de voluntariado en conservación y reparación de computadoras históricas para su exhibición. Asimismo, estos programas también incluyen la posibilidad de colaborar con proyectos de investigación, divulgación y administración del museo.

La gran mayoría de los objetos tecnológicos han sido reemplazados producto de la constante renovación y han desaparecido en poco tiempo. Así, las tareas que se realizan son esencialmente arqueológicas, con la reconstrucción de eventos y contextos particulares partiendo de información fragmentada y en donde los objetos que se investigan nos dan algunos indicios sobre nuestro pasado. Sin embargo, muchas de estas computadoras todavía

se encuentran en nuestra memoria, por lo que su recuerdo no es totalmente inaccesible y es posible seleccionar una mayor cantidad de evidencia. Por lo tanto, las actividades que se desarrollan en la fundación no sólo tienen como objetivo estimular el interés en la conservación de las primeras computadoras, sino permitir también la exploración de un abanico muy amplio de temas técnicos, históricos y sociales de nuestro pasado que de otra manera, habrían caído en el olvido.

La publicación de trabajos referidos a estas tecnologías tiene un rol fundamental en la protección del pasado. De esta manera, la Revista de Tecnología e Informática Histórica (ReTIH) se plantea como una publicación periódica que tiene el objetivo de dar trascendencia a nuevas y antiguas líneas de investigación, o vinculándose directamente a la resolución de problemas. Por otro lado, esta revista pretende ser un punto de encuentro en el cual se generen nuevos espacios de discusión en torno a la recuperación del patrimonio tecnológico en todas sus vertientes. Así, los artículos que se presentan, directa o indirectamente, son aportes generales y particulares dentro de una concepción de la investigación que permite tener miradas más amplias e integrales sobre aspectos de la realidad del pasado, contribuyendo a la construcción del conocimiento.

Diego Gustavo Macchi
Noviembre 2011

CONSERVACIÓN DE PLÁSTICOS Y REVERSIÓN DE COLORES ANÓMALOS

Diego Gustavo Macchi*

Todos los materiales se degradan con el tiempo y los plásticos que componen la mayoría de las estructuras externas de las computadoras no son la excepción. No solamente pueden exhibir cambios a nivel físico que son causados por los distintos esfuerzos a los que se encuentran expuestos y que pueden producir roturas, sino que también algunas veces presentan cambios químicos de sus mismos componentes y cuyo efecto más visible es la decoloración que sufren.

Los plásticos más comunes que se pueden encontrar en la industria informática son el polipropileno, el estireno y en particular Acrilonitrilo Butadieno Estireno (ABS). Si estos plásticos se encuentran sometidos a radiación ultravioleta (UV) se pueden originar fenómenos de oxidación que también aumentan su fragilidad estructural. Como una forma de reducir estos efectos, los fabricantes colocan en la mezcla aditivos tales como estabilizantes, filtros y retardantes ignífugos, además de pigmentos que dan color al plástico. Sin embargo, en condiciones normales la degradación es continua y no se puede detener.

Hasta hace poco tiempo, todos los métodos que se utilizaron para disminuir o retardar los efectos de la radiación UV habían sido ineficaces. Estos procedimientos incluyeron desde la utilización de espuma de melanina, lijas o pinturas hasta lavandina o acetona, que no siempre contribuyeron a la estabilidad estructural del plástico. Aunque no afectan en lo más mínimo las condiciones de operación de ningún componente electrónico, sí lo hace en su estética en tanto se quiera preservar su color original. La aplicación de un proceso simple que se ha desarrollado en los últimos años permite de una manera efectiva y segura, detener y revertir los efectos del envejecimiento en los plásticos.

EL EFECTO SOBRE LA ESTRUCTURA QUÍMICA DE LA RADIACIÓN UV

Los plásticos son polímeros que se encuentran compuestos por una gran cantidad de unidades o monómeros que forman cadenas largas. La unión por medio de enlaces químicos de cada una de estas cadenas forma moléculas de monómeros [1]. El plástico más común que se puede encontrar en las carcasas de las computadoras fabricadas desde fines de la década del ´70 es el ABS y fue uno de los primeros plásticos comerciales en desarrollarse en 1948 [2]. Su estructura química es relativamente simple y le confiere una elevada resistencia al impacto, dureza y elasticidad. Usualmente el ABS no es un polímero lineal y de estructura aleatoria, sino que se fabrica con estructuras complejas que optimizan las propiedades de cada monómero. Es decir, las propiedades del polímero dependen de los elementos y técnicas que se utilizaron en su fabricación. De esa forma, cada componente de este

* IMHICIHU, CONICET – LNSE – ICATEC, diegomacchi@fibertel.com.ar

terpolímero le proporciona una característica diferente [1]. Así, el acrilonitrilo (15 a 35 % de la mezcla) le entrega resistencia química y térmica, el butadieno (5 a 30 %) flexibilidad y resistencia al impacto, y el estireno (40 a 60 %) le otorga una superficie lisa y brillante. No obstante, este material tiene algunas desventajas importantes en cuanto a la temperatura de ablandamiento y a la resistencia ambiental y a los agentes químicos que no son muy elevados. La incorporación del acrilonitrilo le concede una mayor temperatura de ablandamiento y mejora considerablemente la resistencia química. Sin embargo, la resistencia ambiental disminuye por lo que se deben emplear aditivos para neutralizar sus efectos. El compuesto se forma por la unión de todos estos elementos y es totalmente al azar, por lo que dos lotes del mismo producto no serán nunca iguales y reaccionan de manera diferente con la radiación UV [2].

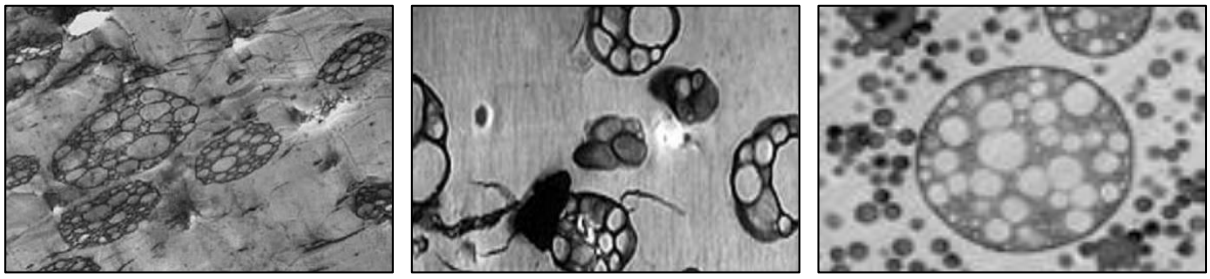

Figura 1. Distintas variedades de polímeros.

Como se puede observar en la Figura 1, cada polímero es diferente del otro, aunque los tres son ABS. Si aumentamos aún más la imagen del microscopio, la retícula del polímero será similar a la de la Figura 2 y en donde los huecos se llenan para modificar sus propiedades físicas [3].

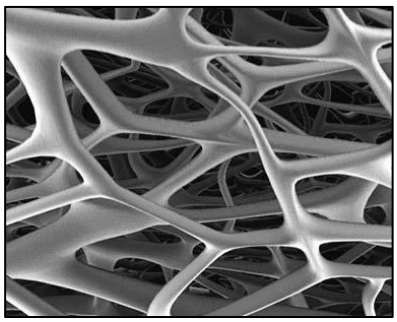

Figura 2. Retícula de ABS aumentada por microscopio.

Esta retícula es la que le confiere al plástico resistencia porque el estireno es un material frágil. Sin embargo, el estireno llena los agujeros y trabaja también como unión con el acrilonitrilo. El butadieno que se dispersa en el polímero trabaja en forma similar.

Los polímeros sintéticos son generalmente más inflamables que los naturales. No obstante, a diferencia de estos últimos que solamente pueden ser tratados con métodos superficiales, los sintéticos incorporan retardantes ignífugos. Estos aditivos tienen un amplio rango de efectos físicos en el polímero, que van desde la formación de capas que actúan como barreras hasta la migración del aditivo hacia la superficie [4].

De esta manera y como el ABS es combustible, se aplican distintos aditivos que tienden a reducir o detener la expansión de las llamas en caso de incendio. Uno de estos componentes químicos es el polibromodifenil éteres (PBDEs) que contiene una gran cantidad de bromina (Br), un elemento de la misma familia de la clorina (Cl) y la fluorina (F) y de gran popularidad en las décadas de los ´80s y ´90s. Estos elementos del Grupo VII de la tabla periódica son halógenos y actualmente son extensamente utilizados en los matafuegos y otros extintores químicos como BCF (bromoclorodifluorometano). Otros matafuegos también incluyen Halón y pertenecen a la familia de los clorofluorcarbonos (CFC), aunque se encuentra en desuso porque ataca a la capa de ozono.

En particular, la cantidad de bromina que se encuentra en la molécula del polímero es directamente proporcional a su efectividad como retardante pero también lo es a la degradación de la radiación UV que se encuentra presente en la luz solar y en menor medida, en las lámparas fluorescentes. El principal retardante empleado por los fabricantes en los polímeros ABS es tetrabromobisfenol-A (TBBP-A) que tiene cuatro átomos de bromina (Figura 3) [2].

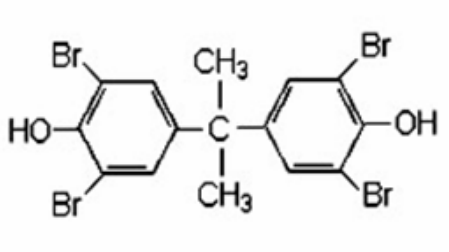

Figura 3. Estructura química del TBBP-A

Cuando el TBBP-A se expone a la radiación UV, las moléculas de bromina se separan y se convierten en radicales libres. Estos radicales necesitan otros electrones para que la capa exterior se estabilice, por lo que se pueden unir a otros átomos de oxígeno (O) y compartir uno de los electrones sobrantes por medio de un enlace covalente coordinado o dativo (Figura 4) [5]. Este tipo de enlace no es tan fuerte como el covalente porque comparte un electrón temporalmente, lo cual es un aspecto clave para revertir el proceso de envejecimiento.

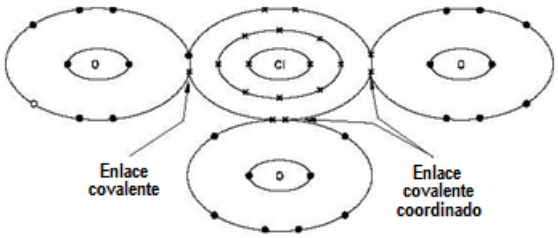

Figura 4. Tipos de enlace del TBBP-A

El retardante ignífugo se encuentra disperso en el polímero, lo que significa que se encuentra cercano o directamente en la superficie. A medida que se degrada el TBBP-A por la exposición a la radiación UV, algunas moléculas brominadas se mueven por el polímero. Si la degradación continúa, además de la existencia de radicales libres de bromina, estas moléculas también se convierten en bisfenol-A. En consecuencia y como la mayoría de estas

moléculas migran hacia la superficie con mayor frecuencia, las carcasas se tornan amarillas o marrones con el tiempo al vincularse con el oxígeno de la atmósfera.

La reacción de la bromina con el oxígeno se acelera por la exposición a luz ultravioleta y sus moléculas vibran cerca de los 300 a 350 nm en el espectro UVA (Figura 5). El número de electrones de su capa externa es impar, por lo que necesita tener un número par para que sea estable. La excitación por radiación UV hace que la bromina se encuentre activa y comience a buscar oxígeno que se encuentra presente en el aire para compartir electrones (Figura 6) [2].

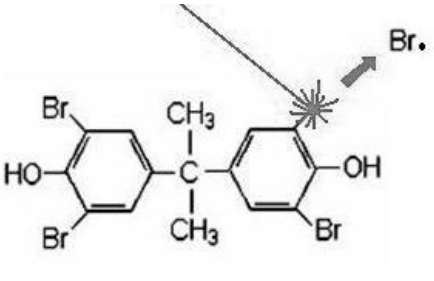

Figura 5. Acción de la radiación UV sobre el TBBP-A

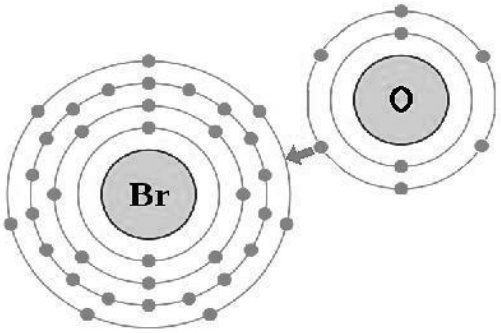

Figura 6. Estabilización de la capa externa de la bromina con átomos de oxígeno.

No obstante, se puede utilizar el efecto desestabilizador de la radiación UV para quebrar las relativamente débiles uniones de la bromina y oxígeno, reemplazando este último por un átomo de hidrógeno (H). El átomo de hidrógeno se encuentra cargado positivamente y neutraliza los electrones libres en el átomo de la bromina, lo cual estabiliza la capa externa con un número par de electrones, reduciendo el estado de la oxidación y finalmente revirtiendo al color original (Figura 7).

De esta manera, se necesita iniciar el proceso de reducción uniendo un átomo de hidrógeno a la bromina para estabilizarla [5]. Una buena alternativa es el peróxido de hidrógeno (H_2O_2) diluido en una solución acuosa con la presencia de un catalizador. La función de un catalizador es la activación y aceleración de la reacción química, necesitando menos energía para comenzar. La tetraacetiletilendiamina (TAED) es extensamente utilizada como catalizador en los detergentes con oxígeno activo, reaccionando con los perboratos y percarbonatos de su fórmula para producir peróxido de hidrógeno en el lavado. No obstante, el TAED va a actuar directamente sobre el peróxido de hidrógeno que se incorpora al proceso. Este compuesto activa el peróxido de hidrógeno, reduciendo la cantidad de energía

necesaria para alcanzar la ruptura del H_2O_2 en oxígeno, hidrógeno y agua. La radiación UV también puede iniciar esta reacción, aunque no de manera rápida y efectiva [2].

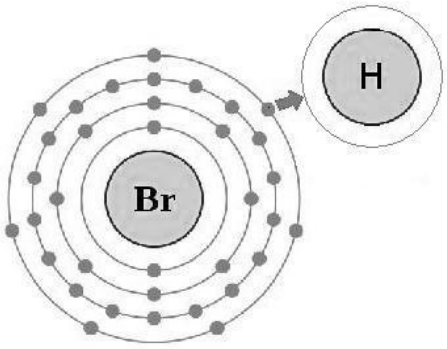

Figura 7. Estabilización de la capa externa de la bromina con átomos de hidrógeno.

El objetivo es romper el peróxido de hidrógeno (H_2O_2) en oxígeno (O), hidrógeno (H) y agua (H_2O) que a su vez, se disocia en H y OH. Por otro lado, el oxígeno de la bromina se obtiene con el TAED y la radiación UV, el cual se reemplaza con un hidrógeno. Los oxígenos removidos se unen con oxígeno del peróxido y se transforman en O_2, cuya reacción se puede observar en forma de burbujas que se forman sobre la superficie del plástico.

Si se sumerge la parte afectada en un tanque de peróxido de hidrógeno diluido junto con una pequeña cantidad de TAED y se la somete a una dosis de radiación UV, se puede desestabilizar la unión bromina-oxígeno de la misma forma que había ocurrido con la TBBP-A. Luego de haberse removido el oxígeno, el átomo de hidrógeno del peróxido catalizado va a tener mucha más atracción para la bromina que para el oxígeno, atrapando el electrón libre de la bromina y uniéndose con un enlace covalente. Esta acción revierte la dirección del flujo de electrones sobre la bromina, por lo que el proceso se revierte por reducción [5].

Finalmente, la lavandina de uso común que contienen hipocloritos sódicos fue un agente de limpieza muy utilizado para detener el envejecimiento en los plásticos. Si bien en teoría funciona, produce cambios estructurales en el plástico que no son beneficiosos. La bromina tiene un número atómico (Z) de 35 y una masa molecular (u) de 80, que en el caso de la clorina alcanzan un valor de 17 y 35,5 respectivamente. Es decir, el átomo de la clorina es la mitad del tamaño de la bromina. No obstante, si bien la clorina puede desplazar los átomos de bromina, su desventaja radica en que esencialmente forma pequeños agujeros en la superficie del plástico, simplemente porque la bromina es más grande que la clorina que debe reemplazarla. De esta manera, el plástico se vuelve frágil porque la estructura reticular no se llena completamente [6]. Por otro lado, la clorina retiene oxígeno del aire y se integra al proceso químico que inicia el cambio de color aunque no de manera tan pronunciada como el método anterior.

CÁLCULO DE LA RADIACIÓN UV. MOMENTOS PROPICIOS PARA LA APLICACIÓN.

La radiación ultravioleta que se encuentra presente en la luz solar es un componente clave en el proceso de reversión del envejecimiento. No obstante, su intensidad varía estacionalmente y con ella la eficiencia que se puede alcanzar. Por lo tanto, es útil conocer

cuál es el momento del año es el más indicado para llevar adelante el procedimiento y de esta manera, reducir los tiempos.

La luz ultravioleta es una radiación electromagnética con una longitud de onda que corresponde al rango de 100 a 400 nm. Estos valores del espectro electromagnético, que se encuentran entre los de la luz visible y los rayos X, se dividen en tres categorías: UVA (315-400 nm), UVB (280-315 nm) y UVC (100-280 nm).

La radiación UVC no se observa comúnmente en la naturaleza porque es absorbido por los elementos que componen la atmósfera tales como el ozono, vapor de agua, oxígeno, dióxido de carbono y diversos contaminantes. De la misma manera, el 90 % de la radiación UVB también es absorbida por estos componentes. Por otra parte, la radiación UVA casi no se ve afectada por las condiciones atmosféricas. En consecuencia, la radiación UV que alcanza la superficie terrestre se compone en su mayor parte por UVA y una pequeña parte de UVB.

Si se considera un punto geográfico particular, la intensidad de la radiación UV se encuentra vinculada a distintas variables, como la altura del sol, latitud, nubosidad, altitud, nivel de ozono y condiciones del suelo. Sin embargo, las tres primeras revisten cierta importancia [7].

a. Altura del sol

Las mayores intensidades se producen cuando el sol alcanza su máxima altura, alrededor del mediodía solar durante los meses de verano.

b. Latitud

A medida que el punto geográfico se encuentre más cercano al Ecuador, más intensa es la radiación UV.

c. Nubosidad

Si bien la radiación UV es mayor cuando no hay nubes, la dispersión de los rayos puede producir el mismo efecto que la reflexión en distintas superficies, aumentando la intensidad total de la radiación UV.

d. Altitud

Cuanto mayor es la altitud, la atmósfera es más delgada y absorbe una menor proporción de radiación UV. Se calcula que cada 1000 m, la intensidad aumenta entre un 10 y 12 %.

e. Ozono

El ozono absorbe parte de la radiación UV y su concentración varía diaria y anualmente.

f. Reflexión del suelo

Distintos tipos de superficies reflejan o dispersan la radiación UV en distinta medida. Por ejemplo, la nieve refleja un 80%, la arena un 15% y el agua un 25%.

El índice UV es un parámetro que indica la intensidad de la radiación ultravioleta durante la hora pico de iluminación solar y que se puede modelar estacionalmente con la siguiente fórmula:

$$IUV = k_{er} \cdot \int_{280}^{400} E_\lambda \cdot S_{er}(\lambda) \cdot d\lambda$$

en donde E_λ es la radiación espectral en la longitud λ, $S_{er}(\lambda)$ es el espectro de acción de referencia para el eritema y k_{er} es una constante [7]. De esta manera, se puede calcular el IUV para toda la región de la República Argentina [8], tal como se puede observar en Figura 8.

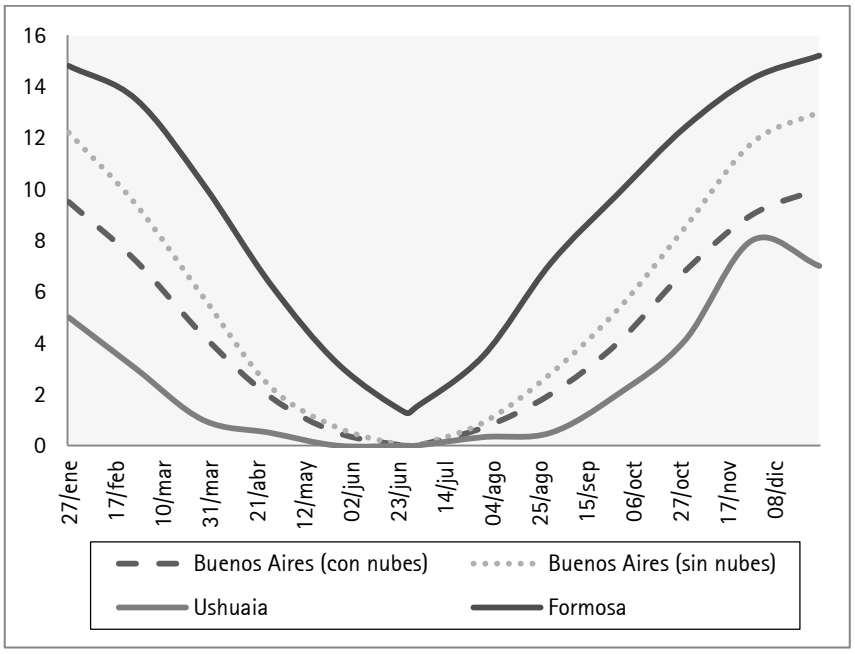

Figura 8. Cálculo de la radiación UV para los extremos geográficos de la República Argentina.

Si se consideran los valores que muestra la curva, el aumento progresivo del índice UV es proporcional a la excitación de los átomos de bromina, alcanzando su valor máximo en el mes de diciembre. En consecuencia, la reacción en el plástico ocurre de manera más rápida. Inversamente, el período con menor eficiencia tiene lugar durante los meses invernales en los que el IUV es extremadamente bajo si se comparan con los meses estivales. Por otro lado, para paliar la falta de radiación UV, se pueden utilizar lámparas UV de baja potencia. Las experiencias indican que no se logran grandes diferencias en cuanto al uso de lámparas de potencia media, por lo que es usual el empleo de lámparas de 11 W hasta 60 W.

METODOLOGÍA

En la práctica, los elementos de uso común y amplia disponibilidad en el mercado que se utilizaron en el proceso son los siguientes:

a. Agua oxigenada (H_2O_2) de 40 volúmenes.

b. Goma xántica o guar.

c. Glicerina líquida.

d. Cualquier polvo de limpieza que contenga TAED (comúnmente rotulados como Oxy).

e. Film plástico adherente.

El conjunto de actividades que se deben llevar adelante son bastante simples y pueden ser resumidas en pocos pasos de esta manera:

a. En un recipiente de plástico o cerámica, se pone ½ litro de H_2O_2.

b. Se colocan dos cucharadas soperas colmadas de goma xántica (o guar).

c. Se revuelve hasta llegar a la textura de un gel color perlado sin tener en cuenta la aparición de algunos grumos.

d. A la mezcla se le coloca glicerina en una cuchara de té para que los grumos pierdan consistencia.

e. Se revuelve hasta lograr un gel homogéneo color perlado sin mayor presencia de grumos.

f. Llegado a este punto, la mezcla que se ha formado se puede almacenar en algún lugar oscuro.

g. Si se utiliza en el momento, se vierte ¼ de la medida de una cucharada de té del polvo de limpieza que contiene el TAED.

h. Nuevamente, se revuelve bien hasta lograr un gel color perlado sin grumos.

i. Se proceder a la aplicación teniendo en consideración la utilización de material de protección tales como guantes y anteojos.

j. Con el plástico previamente limpio de grasa y distintas suciedades, y además sin partes metálicas porque se oxidan rápidamente, se aplica una capa generosa sobre toda la superficie a blanquear. No es conveniente utilizar un pincel porque tiende desparramar una capa muy fina del gel sobre el plástico que se seca y marca los pincelazos como trazos más brillantes que otros.

k. Luego de haber puesto una capa del gel se envuelve toda la pieza con film adherente, de forma que quede herméticamente sellada y no entre aire. Este paso es necesario para conservar la humedad interna y que el gel no se seque. Es muy importante que el film no se termine pegando a la pieza porque caso contrario, la reacción química se concentra en un área y se corre el riesgo de sobreexposición.

l. La pieza se pone al sol o bajo una lámpara de luz ultravioleta para dar inicio a la reacción.

m. Dependiendo de diversos factores (ver radiación UV) a las pocas horas se nota un cambio notable en la coloración. También se va a notar que el gel ha

desaparecido, sustituyéndose por un líquido acuoso que no permite que se pegue el film.

Puede ocurrir que, más allá de todas las precauciones, parte del film se termine pegando al plástico. Si ese es el caso, se puede lijar con una lija muy fina hasta emparejar. Por otro lado, otro método para enmascarar imperfecciones es colocar algún líquido con brillo acrílico.

La causa principal por los resultados rápidos que se obtienen es la elevada degradación del TBBP-A cuando se expone a radiación UV, pudiendo llegar de 6 a 8 horas bajo luz ultravioleta concentrada. Estas condiciones se alcanzaron al degradar TBBP-A y revertir los efectos de los radicales de la bromina. El efecto es superficial y sólo penetra en el plástico por algunos microns pero el resto no es afectado porque al no estar en contacto con el aire, no se oxida. Si bien el principal parámetro que determina la fuerza de un sistema de enlaces de hidrógeno es la cantidad de enlaces que existen [9], la degradación puede ocurrir nuevamente si estos enlaces no son fuertes. En última instancia, dado que este método revierte el proceso de envejecimiento pero no lo elimina del todo, el polímero se puede sellar con una laca resistente a la radiación UV, por lo que la decoloración no debería regresar.

CONCLUSIONES

Se ha explicado un método para eliminar y revertir el proceso de envejecimiento que ocurre en los plásticos utilizados en la mayoría de las computadoras desde hace tres décadas. El efecto más notorio es la decoloración del plástico y se explica por la oxidación de los átomos de bromina por exposición a una fuente de radiación UV. La reversión del proceso se efectuó con la utilización de peróxido de hidrógeno y un catalizador, que inició una reacción de reducción en el plástico.

Si bien el procedimiento es eficiente, las experiencias que se llevaron a cabo en distintos períodos del año, arrojaron una marcada disimilitud en cuanto al tiempo necesario para terminar la reversión. Se explica principalmente, por la intensidad de la radiación UV, que en el hemisferio sur alcanza un pico en el mes de diciembre y es inversamente proporcional a los tiempos empleados. No obstante, a mayor IUV, mayores son las precauciones que son necesarias para evitar que el proceso siga atacando al polímero. De esta manera, con altos IUV, es recomendable verificar el estado de la reacción cada dos horas aproximadamente.

Por otro lado, no existen rangos temporales precisos en cuanto a la finalización del proceso por la gran diferencia que existe entre los lotes de los plásticos. Como regla general, los plásticos utilizados por empresas como Atari y Commodore son más propensos a los errores que pueden ocurrir por la sobreexposición química. Por otro lado, los plásticos utilizados por Apple no sufrirían de este tipo de inconveniente [6].

En definitiva, este método permite revertir la decoloración de los plásticos y es la mejor alternativa que existe actualmente para hacer retornar las carcasas de las computadoras a sus condiciones cromáticas originales.

BIBLIOGRAFÍA

[1] D. Teegarden (2004). *Polymer Chemistry. Introduction to an indispensable science*. NSTA Press, Arlington, Virgina.

[2] D. Stevenson (2011). The Retrobright Project. http://retr0bright.wikispaces.com/home. (12 de Junio 2011)
[3] J. L. Leblanc (2010). *Filled Polymers. Science and Industrial Applications*. CRC Press, Florida, Estados Unidos.
[4] T. R. Hull y A. Stec (2009). Polymers and Fire. En: *Fire Retardancy of Polymers. New Strategies and Mechanism*, editado por T. R. Hull y B. Kandola, pp. 1-14. RSC Publishing, Cambridge, Reino Unido.
[5] M. Angelini, E. Baumgartner, C. Benitez, M. Bulwik, R. Crubellatu, L. Landau, L. Lastres Flores, M. Pouchan, R. Servant y M. Sileo (1992). *Temas de Química General*. EUDEBA.
[6] D. Stevenson (2011). Comunicación personal, 29 de agosto de 2011.
[7] World Health Organization (2002). *Global Solar UV Index: A practical Guide.* WHO, Ginebra, Suiza.
[8] Servicio Meteorológico Nacional (2011). http://www.smn.gov.ar/?mod=ozono&id=2 (15 de Junio 2011).
[9] W. Binder y R. Zirbs (2007). Supramolecular polymers and networks with Hidrogen Bonds in the main and Side-Chain. En: Hydrogen Bonded Polymers, editado por W. Binder, pp. 3-71, Springer.

MEMORIA E INTRODUCCIÓN A LAS VARIABLES DEL SISTEMA EN LA ZX SPECTRUM Y TS 2068[*]

Patricio Herrero Ducloux[†]

La ZX Spectrum es una computadora de 8 bits desarrollada por Sinclair y lanzada al mercado británico en 1982. Su popularidad, tanto en su tierra de origen como en otros países tales como España y Argentina, fue comparable al suceso de Commodore en Estados Unidos. Rápidamente se estableció como una de las computadoras para la que se produjo una gran cantidad de software, en su mayor parte europeo.

La CPU se encuentra basada en un Zilog Z80 corriendo a 3,5 MHz y cuenta con una ROM de 16 kB y originalmente, con una RAM de 16 kB que posteriormente se amplió a 48 kB. La salida de video es por medio de un conector RF, con una resolución de imagen de 256*192. Por otro lado, tiene la posibilidad de producir sonido por un canal con diez octavas que se reproduce por un parlante incorporado en la computadora. Por otro lado, la Timex Sinclair 2068 es una variante de la ZX Spectrum que fue comercializada particularmente en Estados Unidos a partir de 1983. Cuenta con algunas mejoras como un chip de sonido distinto, dos puertos de joysticks y modos de pantalla adicionales.

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. Por último, vamos también a realizar un pequeño análisis del uso de la memoria en ambos equipos.

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 64 kB, 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 64 kB 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 la Tabla 1 y 2 se muestra la distribución de memoria en la ZX Spectrum, la cual es vista como un único bloque de 64 kB. 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

[*] Este artículo se encuentra disponible en http://www.ssir.com.ar/VariablesZXSpectrum.pdf

[†] SSIR, phd@ssir.com.ar

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.

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* [1]. Básicamente, en esta oportunidad nos limitaremos a explicar que la ROM de la ZX Spectrum se divide en tres grandes secciones: rutinas de entrada y salida, intérprete de Basic y manejo de expresiones.

Desde	Hasta	Descripción
0 0000h	16383 3FFFh	Memoria ROM, de solo lectura. En total son 16384 bytes (16KBytes)
16384 4000h	22527 57FFh	Definición de píxeles en pantalla. La resolución de pantalla es de 256*192 puntos. Los ocho bits de cada byte representan un segmento de ocho puntos definiendo cuál está encendido o no. Por lo tanto, se emplean (256*192/8)=6144 bytes (6 kB).
22528 5800h	23295 5AFFh	Definición de atributos de color en pantalla; un byte por cuadro de 8*8 píxeles. Para cada cuadro, el byte define si hay o no flash o brillo, 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 observaremos 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

Tabla 1. Distribución de la memoria de la ZX Spectrum (Parte 1).

Desde	Hasta	Descripción
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.

Tabla 2. Distribución de la memoria de la ZX Spectrum (Parte 2).

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

a. Rutinas de entrada y salida

1. Ocho rutinas invocadas por las instrucciones RST, rutinas auxiliares y tablas de tokens[1] 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

b. 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

c. 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 que hemos mencionado.

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 carácter lo hacen en forma gráfica, como un cuadro de 8x8 píxeles). 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 píxeles en pantalla comienza en la posición 16384 (4000h). Los ocho bits de cada byte representan un segmento de 8 píxeles definiendo cuál está encendido y cuál no. Así, cada 32 bytes tenemos representada toda una línea horizontal de puntos. La pantalla se divide en tres bloques de 64 píxeles de alto (8 renglones de texto de 8 píxeles de

altura) cada uno, ocupando 2048 bytes (2 kB) cada bloque. Por lo tanto, se ocupa un total de 6 kB para la pantalla completa. Cada bloque se va completando de a una línea de píxeles por renglón de texto. De esta forma los primeros 32 bytes forman la línea 0 del renglón 0, los siguientes 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) y líneas 3, 4, 5, 6 y 7. En ese punto se terminó el primer bloque de 2 kB y los siguientes 32 bytes definirán la línea 0 del renglón 8 (comienzo del segundo bloque desde el renglón 8 hasta el 15).

La definición de atributos de pantalla comienza en la posición 22528 (5800h). Cada byte contiene la información sobre los atributos de un cuadro de 8*8 puntos (espacio ocupado por un carácter) 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: Brillo (1: con brillo, 0: sin brillo). Aplica tanto al fondo como a la tinta.
Bits 5 a 3: Papel (valores 0 a 7). Color de los píxeles que están apagados en el cuadro.
Bits 2 a 0: Tinta (valores 0 a 7). Color de los píxeles 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[2]:

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

Luego de ejecutarlo con RUN, 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 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 su instrucción más adelante y su detalle en otro trabajo.

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, interface 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: *ZX Interface 1 and Microdrive Manual* [2] y *The Companion to the Sinclair ZX Microdrive and Interfaces* [3].

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 ZX Spectrum, cada canal se define con una letra que lo identifica y por otro lado, 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 en particular, estos se hayan siempre 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 (interface RS 232 en modo binario), T (interface RS 232 en modo texto) o N (red local)[3]. 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. Por otra parte, en la revista *Microhobby Semanal* hay una completa nota sobre los canales de datos y la forma de acceder a los mismos [4][4].

Como se puede observar, 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 23632
  20 REM While peek(n)<>128 mostrar datos
  30 IF PEEK n=128 then go to 100
  40 LET a=PEEK n: LET b=PEEK (n+1): LET c=PEEK (n+2): LET d=PEEK
(n+3): LET e=PEEK (n+4)
  50 PRINT “Direccion ”;n
  60 PRINT a;TAB 6;b;TAB 12;c;TAB 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 podremos observar 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, tal como se muestra en la Figura 1.

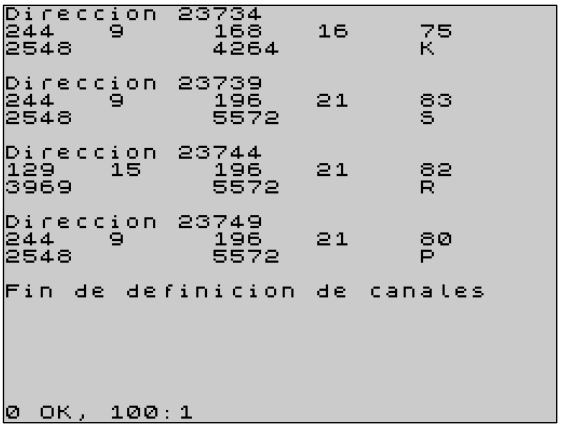

Figura 1.

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, si ejecutamos el programa:

```
10 LET n=PEEK 23635+256*PEEK 23636
20 POKE n,0
30 POKE n+1,0
```

En esta oportunidad, podemos observar que para la primera línea tenemos el número 0. Para volver las cosas a la normalidad, basta con colocar:

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

Ahora si cambiamos levemente el programa:

```
10 LET n=PEEK 23635+256*PEEK 23636
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, podemos probar el programa:

```
  10 LET n=PEEK 23635+256*PEEK 23636
  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 23636
  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+2 bytes de nro+2 bytes de
longitud de texto + 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 2da. linea
  80 FOR i=m+4 TO m+4+d-1
  90 PRINT i;“: ”;PEEK i,CHR$ PEEK i
 100 NEXT i
```

Luego de ejecutarlo, observamos byte por byte, como se encuentra guardada esta segunda línea en memoria. Notemos que cuando aparece un número literal luego es seguido por el carácter 14 que significa *número* y cinco bytes más con la representación del mismo. Esto es porque los números se representan de dos maneras diferentes: una primera si son enteros entre -65535 y 65535 y otro diferente si no lo son. Vamos a comentar más sobre estas formas cuando analicemos 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.

Posteriormente al 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 acuerdo a la Figura 2.

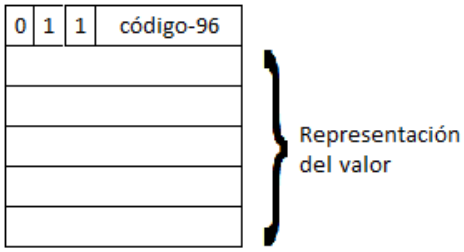

Figura 2.

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 carácter *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: el 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, si siendo entero se encuentra 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) y 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:

$$N = 2^{e-128} \cdot M$$

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 qué ocurre con 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 se encuentra en 1. Entonces se asume que siempre es así y se utiliza este primer bit para indicar si el número es positivo (se coloca en 0) o negativo (se coloca en 1). En el caso de las variables numéricas cuyo nombre tiene más de un carácter, la representación será la correspondiente a la Figura 3.

Es decir, cambian los tres bits más significativos del encabezado. Al código ASCII del primer carácter 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.

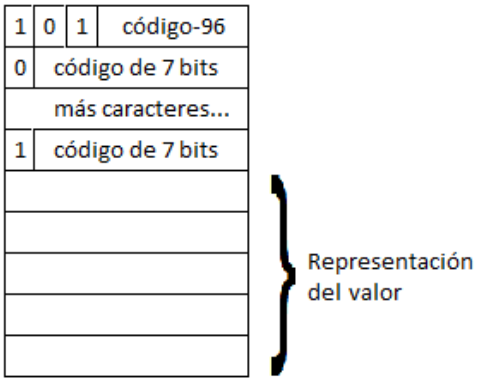

Figura 3.

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 la Tabla 3 que muestra las distintas posibilidades existentes:

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

Tabla 3.

Se puede examinar el área de variables con el siguiente 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 se muestra en Figura 4. 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 carácter *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+256*93=23296. El último byte es 0, como está establecido para los números representados como valores enteros.

Inmediatamente, en la posición 23939 encontramos la definición de la siguiente variable. El valor 97 se representa en binario de esta manera: 01100001. Los primeros tres bits (011) nos dicen que estamos ante una variable numérica representada con un solo carácter. Los últimos cinco bits nos dan el valor 1, al cual le sumamos 96 y llegamos a 97, código ASCII del carácter *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 carácter 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 es distinto de 0 e 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 1/2 (0,5) por lo que el primer bit toma el valor 1 siempre. ¿Por qué se encuentra en 0? De acuerdo a 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.

```
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
```

Figura 4.

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 como resultado 110, código ASCII de la letra *n*. En el caso de los iteradores, luego del nombre vendrán diferentes campos y 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; siempre los últimos cinco bits, los cuales 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 colocando 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, si ingresamos el programa:

```
10 LET a=PEEK 23730+256*PEEK 23731
20 PRINT a
30 PRINT PEEK a
```

se obtiene 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
```

Nuevamente, ejecutamos el programa y obtendremos una salida diferente:

```
64000
62
```

Podemos notar 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 (que otra vez se habrá borrado) 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 8*8 píxeles. 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
```

que produce la salida que se muestra en Figura 5.

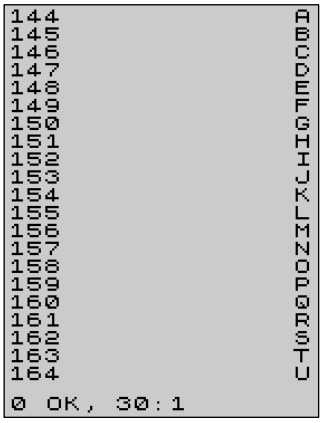

Figura 5.

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 se encontrará en la posición 65535 (o 32767) que es la 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.

EMPLEO DE LA MEMORIA EN LA TS 2068

Los conceptos que hemos visto en el punto anterior son aplicables en general también a la TS 2068. En este apartado simplemente se presentarán 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 con 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[5]. En teoría se pueden tener hasta 16 MB de memoria gracias a que disponemos de 256 bancos de 64 kB cada uno [5] 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 64 kB a la vez porque el bus de direcciones del Z80 es de 16 bits). Se definen 256 bancos de 64 kB cada uno (numerados del 0 al 255) y cada uno de ellos se divide en 8 bloques de 8 kB 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 16 kB de memoria ROM (ocupando los dos primeros bloques de 8 kB) y los 48 kB 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 8 kB hacen llegar el total de memoria ROM a 24 kB. El banco 0, llamado *Dock Bank*, es ocupado cuando colocamos 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. 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) se puede observar en Tabla 4 y 5.

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 píxeles en pantalla; un byte por segmento de 8 píxeles 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 píxeles. 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.
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

Tabla 4. Distribución de la memoria de la TS 2068 (Parte 1).

Desde	Hasta	Descripción
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

Tabla 5. Distribución de la memoria de la TS 2068 (Parte 2).

VISTA A VUELO DE PÁJARO. INTRODUCCIÓN A 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 en Tabla 6, 7, 8 y 9 junto a sus usos.

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.

Tabla 6. Variables del Sistema en la ZX Spectrum (Parte 1).

Bytes	Dirección	Nombre	Contenido
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 carácteres 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.
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.

Tabla 7. Variables del Sistema en la ZX Spectrum (Parte 2).

Bytes	Dirección	Nombre	Contenido
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.

Tabla 8. Variables del Sistema en la ZX Spectrum (Parte 3).

Bytes	Dirección	Nombre	Contenido
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.
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.

Tabla 9. Variables del Sistema en la ZX Spectrum (Parte 4).

La TS 2068, por su parte, utiliza además de las variables de la Tablas 5, 6, 7 y 8, las variables que se exhiben en Tabla 10.

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

Tabla 10. Variables del sistema específicas en la TS 2068.

CONCLUSIONES

Este artículo no pretende extenderse hacia otros equipos que la ZX Spectrum o la TS 2068, por lo que 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 128 kB ocupa con el mismo fin la memoria inmediatamente anterior a la primera variable, donde originalmente se ubicaba el buffer de impresora[6]. 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 esto ocurre solamente cuando la computadora trabaja en modo 128 kB; en el modo de 48 kB se continúa manteniendo la compatibilidad.

NOTAS

[1] 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.
[2] Todos los listados de programas se encuentran justificados de acuerdo a la programación de la ZX Spectrum.
[3] La totalidad de estos canales mencionados son manejables mediante la Interface 1.
[4] El contenido de la revista se puede leer desde http://www.microhobby.org/numero038.htm.
[5] 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.
[6] Probablemente, porque la memoria inmediatamente posterior podría ser ocupada por las variables empleadas por la Interface 1.

BIBLIOGRAFÍA

[1] I. Logan y F. O´Hara (1983), *The Complete Spectrum ROM Disassembly*. Melbourne House.
[2] Cambridge Communication Ltd. (1983), *Zx Interface 1 and ZX Microdrive Manual*. Sinclair Research Ltd.
[3] S. Cooke (1984), *The Companion to the Sinclair ZX Microdrive and Interfaces*. Pan Books Ltd.
[4] S. Martinez Lara (1985), Los canales en el Spectrum. *Microhobby Semanal* 38:14-16.
[5] F. Blechman (1983), *TS 2068 Begginer/Intermediate Guide*. Howard W. Sams & Co. Inc., Indianapolis.

LAS VARIABLES DEL SISTEMA DE ZX SPECTRUM Y TS 2068 EN DETALLE[*]

Patricio Herrero Ducloux[†]

En este trabajo analizaremos, cada una de las variables que hemos descripto en [1], además de ver ejemplos de uso. Como se ha mencionado en ese artículo, las variables del sistema corresponden a un bloque de memoria entre las direcciones 23552 y 23734 que son utilizadas para almacenar distintos valores y estados de programas en ejecución. Vamos a utilizar la siguiente notación:

(*X*)	significa el valor contenido en la posición X y similar a PEEK(X).
(*X*)→*a*	significa que el valor en la posición X lo guardamos en una variable a, por lo que sería como a=PEEK(X).
a→(*X*)	significa que en la posición X guardamos el valor (o una variable) a, de la misma forma que 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:

$$(X) + 256 * (X + 1) \rightarrow valor$$

Precisamente, esto se debe a la forma en que el Z80 toma los valores de dos bytes: el menos significativo en la primera posición seguida por el más significativo. Es interesante saber que el diseñador principal del Z80 trabajaba en Intel y estuvo involucrado en el 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.

[*] Este artículo se encuentra disponible en http://www.ssir.com.ar/VariablesZXSpectrum.pdf

[†] SSIR, phd@ssir.com.ar

VARIABLES DEL SISTEMA

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), el cual se utiliza normalmente. El superior se usa cuando, mientras se tiene presionada una tecla, otra nueva es presionada sin soltar la primera. La lógica es la siguiente:

a. Si no hay tecla presionada:

(KSTATE+4) = 255
(KSTATE+5) = 0

b. Si hay una tecla presionada:

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

c. Si hay una única tecla presionada o ninguna:

(KSTATE+0) = 255
(KSTATE+1) = 0

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

Estos estados los podemos ver con un programa[1]:

```
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
```

Luego de ejecutarlo, debemos presionar diferentes teclas para comprobar los valores que va tomando. Por ejemplo, notemos que el valor de la tecla pulsada queda guardado aunque se la deje de oprimir, 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)

En esta variable se almacena el valor de la última tecla que presionamos, sin importar si 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
```

Si se ejecuta nuevamente y presionamos la letra N. Al mismo tiempo, 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 habrá comenzado 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 lo combinamos también con un 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)

La dirección de la primera letra de argumento de una función definida por el usuario se guarda en esta variable, 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 con esto, tenemos 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*PEEK 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 23564
```

Posteriormente a su ejecución, vemos el siguiente resultado:

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

el cual se interpreta de esta manera: 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 siguientes ejemplos[2]:

a. 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 23565
```

Aparecerá en la pantalla el valor 4.

b. 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 23565
```

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

c. 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 23565
```

Nos va a aparecer en pantalla el valor 1, a pesar de que no estamos definiendo ningún color. 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. Es decir, 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 se dirige 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 carácter 33 (!) y así sucesivamente hasta el carácter 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 se debe de restar 256? Porque los caracteres de verdadero interés para mostrar o imprimir son los códigos 32 hasta 127. Fijémonos que 32*8=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)+8*65 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). Si probamos:

```
POKE 23605,8
```

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 en 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 debemos escribir sin prestar atención a lo que veamos en pantalla, que será:

```
QPLF!34717-1
```

Es verdad, en este lugar estamos poniendo POKE 23606,0. Lo que pasa es que la P se ve como si fuera la Q y así sucesivamente. Además, en pantalla hasta el cursor L cambió por M. De esta manera, 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. Por ejemplo, Alonso y Moreno [2] definen tres juegos de caracteres, de los cuales tomamos los datos del primero (Figura 1).

```
  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
```

Figura 1.

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 (cuyo concepto ampliaremos luego en la dirección 23730) dándole un valor suficientemente pequeño como para que no quede espacio libre:

```
CLEAR 23860
```

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
```

obteniendo el mensaje:

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

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

```
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 el comando BEEP (que admite decimales para el valor de la nota):

```
BEEP 1,34.4
```

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 porqué 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. Si hacemos las pruebas con:

```
POKE 23610,0
```

veremos que nos aparece el error:

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

Asimismo, si se ingresa:

```
POKE 23610,26
```

que nos da el siguiente error:

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

Con valores más altos, se muestran códigos sin sentido aparente, salvo observar 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 si no 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 23614
  20 PRINT a
  30 LET a$=STR$ PEEK a+":"+STR$ PEEK (a+1)+":"+STR$ PEEK
(a+2)+":"+STR$ PEEK (a+3)+":"+STR$ PEEK (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 de acuerdo a la Figura 2.

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+256*19=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.

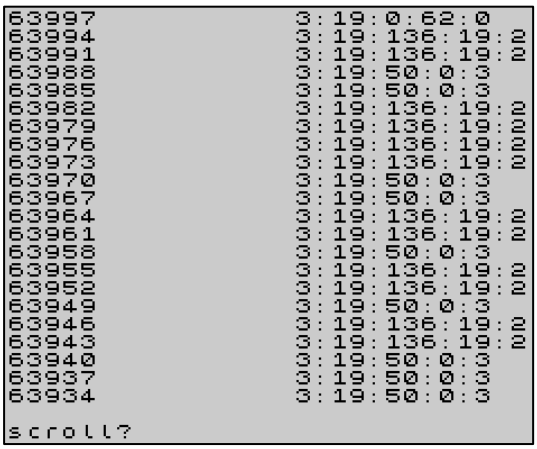

Figura 2.

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+256*0=50. Cuando el GOSUB es desde la línea 5000, los valores son 136 y 19, ya que 136+256*19=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+256*62 daría 15872, un valor de retorno imposible de tener. Y en realidad, la rutina que ejecuta la instrucción RETURN[3] 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 [3]:

```
  10 LET a=PEEK 23613+256*PEEK 23614
  20 POKE a,0
  30 POKE a+1,0
  40 FOR n=1 TO 1000
  50 PRINT AT 0,0;n
  60 NEXT n
```

Antes de hacer RUN, guardémoslo para más pruebas 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 apretemos BREAK o dejemos que el programa termine, se invocará la rutina de manejo…que hemos determinado que ahora se encuentre en la dirección 0. Y en esta dirección 0 está la rutina de inicio de la Spectrum. Como se puede observar, 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.

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 observar 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 23618+256*PEEK 23619
20 IF RND>0.5 THEN PRINT “De la linea 20 salto a la 40”: GO TO 40
30 PRINT “Linea 30: “;PEEK 23618+256*PEEK 23619
40 IF RND>0.5 THEN PRINT “De la linea 40 salto a la 30”: GO TO 30
50 PRINT “lINEA 50: “;PEEK 23618+256*PEEK 23619
60 IF RND>0.5 THEN PRINT “De la 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, como se muestra en la Figura 3.

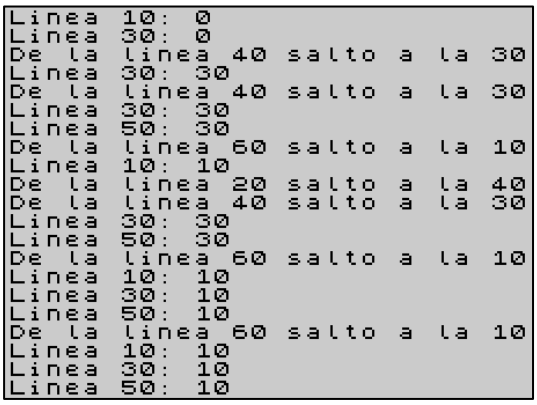

Figura 3.

En el 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 lo que se indica en la Figura 4.

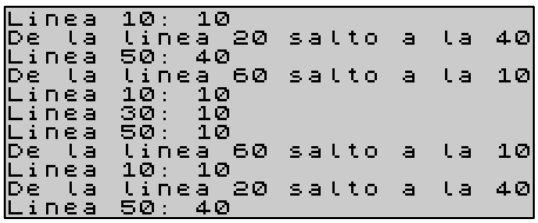

Figura 4.

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 Insr. 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[4]. 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 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 que se muestra en Figura 5:

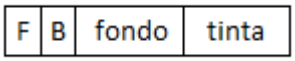

Figura 5.

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 [1].

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 [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 pantalla veremos el valor 128. Esto significa que no hay variables definidas. 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’PEEK 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 es, no por casualidad, el código de la palabra PRINT, ya que E LINE indica la dirección en memoria del comando que se está ingresando en modo directo.

DEST (23629, 2 bytes)

Esta variable del sistema está, en cierta forma, asociada a la anterior. El valor almacenado es la dirección de memoria donde se encuentra guardado el valor de la última variable en Basic a la que hayamos ejecutado una operación de asignación.

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 [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.

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 [1] hemos visto detalladamente la manera en que se guarda el programa en memoria. Sin embargo, podemos escribir un pequeño programa que se examina a sí mismo y muestra cómo está almacenada:

```
10 LET a=PEEK 23635+256*PEEK 23636
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.

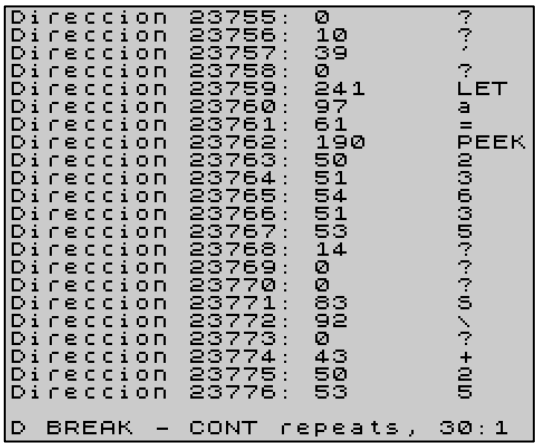

Figura 6.

En la Figura 6, podemos notar 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)

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.

DATADD (23639, 2 bytes)

Esta variable es importante cuando estamos leyendo datos mediante la instrucción READ. Apunta a la dirección donde está la coma inmediatamente después del último elemento leído.

E LINE (23641, 2 bytes)

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, ya que esta zona se encuentra inmediatamente después de la definición del programa.

K CUR (23643, 2 bytes)

Esta variable del sistema se encuentra relacionada con la 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.

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 carácter a interpretar. Podemos intentar el siguiente programa:

```
10 LET a=PEEK 23645+256*PEEK 23646
20 PRINT a
30 LET b=PEEK 23645+256*PEEK 23646
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 ya que se esperaría el Enter que termine las líneas 10 y 30 en lugar del 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*PEEK 23646
20 DEF FN g()=PEEK 23645-0+256*PEEK 23646
30 LET a=FN f()
40 PRINT a; TAB 10;PEEK a; TAB 20; CHR$ PEEK a
50 LET b=FN g()
60 PRINT b;TAB 10;PEEK b;TAB 20;CHR$ PEEK b
```

Las funciones f() y g() definidas son, técnicamente, iguales: al 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 carácter 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+PEEK 23645
20 DEF FN g()=PEEK 23645+256*PEEK 23646
30 LET a=FN f()
40 PRINT a;TAB 10; PEEK a; TAB 20; CHR$ PEEK a
50 LET b=FN g()
60 PRINT b;TAB 10;PEEK b;TAB 20;CHR$ PEEK b
```

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

```
23799       13
23821       43          +
```

En el primer caso, podemos observar que ahora estamos apuntando 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.

X PTR (23647, 2 bytes)

Dirección del lugar exacto donde el intérprete Basic encontró un error cuando se intentó introducir una nueva línea de programa. Es la posición del carácter que sigue al signo *?* que se muestra en tal caso de error.

WORKSP (23649, 2 bytes)

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. Es decir, donde 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 la Tabla 1 que se encuentra en [1] podemos ver el mapa de memoria y podemos deducir que si efectuamos la operación ((RAMTOP) + 256 * (RAMTOP+1)) - ((STKEND) + 256 * (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 si no 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)

Aquí 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 que 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 [4], quien concibió la fórmula:

$$S_{k+1} = A \cdot S_k \; mod \; N$$

Es decir, 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 [5]. 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:

$S_{k+1} = (75 \cdot (S_k + 1) - 1 \; mod \; 65537$

Dado que el valor inicial de esta variable del sistema es 0, podemos calcular el siguiente valor que tomará:

$S_1 = (75 \cdot (S_0 + 1) - 1 \; mod \; 65537 = (75 \cdot (0 + 1) - 1) mod \; 65537 = (75 - 1) mod \; 65537 = 74$

El valor que siga a este recién obtenido será:

$S_2 = (75 \cdot (S_1 + 1) - 1 \; mod \; 65537 = (75 \cdot (74 + 1) - 1) mod \; 65537 = (5625 - 1) mod \; 65537 = 5624$

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[5], 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.

```
10 DEF FN p()=PEEK 23670+256*PEEK 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*65536
80 NEXT n
```

Al ejecutar el programa, obtendremos la salida que se observa en Figura 7, no importa cuántas veces lo hagamos correr:

```
Nro   SEED      RND             Sgte.
 0    0         .0011291504     74
 1    74        .08561543       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
```

Figura 7.

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)

Tres 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 [1] se ha 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. Alonso Rodriguez [6] señala un truco que modifica el contenido de esta variable para duplicar o triplicar la cantidad original de 21 GDU disponibles, 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, 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.

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 carácter a imprimir. Se puede cambiar con POKE y se alterará la posición del próximo carácter 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 píxeles 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 carácter 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) (Figura 5).

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 de la Figura 8:

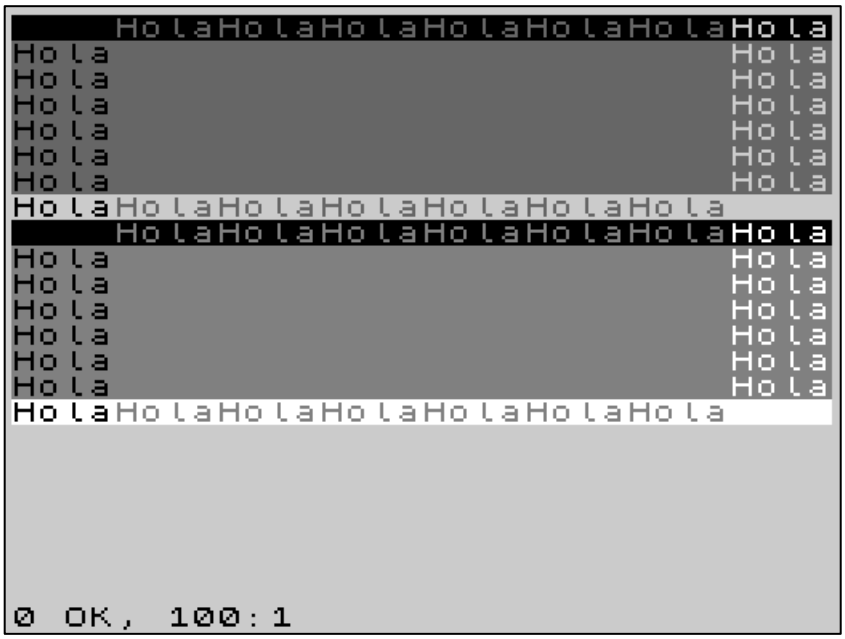

Figura 8.

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. Se puede probar 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. Es decir:

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 utiliza 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 se encuentra programado lo que hay que 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 se puede ver en el ejemplo que se encuentra en [1, p.21] 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 el fin de la zona donde se guarda el programa en Basic junto con los datos que utiliza y el 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.

CONCLUSIONES

Las variables del sistema son direcciones indexadas en la ROM de la ZX Spectrum que permiten a la computadora saber todas las cosas necesarias para llevar a cabo una operación. En cualquier tipo de programa, podemos hacer uso de la información almacenada en estas ubicaciones de la memoria, mediante su lectura o simplemente cambiando su valor. No todas ellas son de gran importancia y deben ser modificadas por el usuario porque, como hemos visto, pueden hacer colgar o ser ignoradas por la computadora. No obstante, otras pueden ser modificadas y ser de gran utilidad en la programación.

Finalmente, existen variables de sistema propias de la TS 2068 que agrega nuevas funcionalidades que son exclusivas de este modelo y serán analizadas en detalle en otro trabajo.

NOTAS

[1]Todos los listados de programas se encuentran justificados de acuerdo a la programación de la ZX Spectrum.

[2] 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).

[3] Se encuentra en la dirección 7971 (1F23h) y está explicada en *The Complete Spectrum ROM Disassembly*

[4] 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.

[5] Si alguien se lo planteara, 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 x (S_k + 1) - 1) mod 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 [mantisa|exponente] 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.

BIBLIOGRAFÍA

[1] Herrero Ducloux (2011), Memoria e introducción a las variables del sistema en la ZX Spectrum y TS 2068, *Revista de Tecnología e Informática Histórica* 1, pp. 11-29.

[2] J. Alonso y A. Moreno (1985), Los canales en el Spectrum. *Microhobby Semanal* 30:14-16.

[3] A. Hewson (1984), Helpline. *Sinclair User* 29. ECC Publications Ltd.

[4] Wikipedia encyclopedia (2011), *Lehmer random number generator*. http://wapedia.mobi/en/Park-Miller_function (11 de septiembre de 2011).

[5] J. Alonso Rodriguez (1985), Más de 21 U.D.G. en tu Spectrum. *Microhobby Semanal* 42:14-16.

PROGRAMANDO LA COMMODORE 64 EN SERIO
CIRCUITO DE VIDEO Y COMPONENTES BÁSICOS DEL SISTEMA

Marcos Leguizamón*

El objetivo de este artículo es explicar cómo se escriben los programas que aprovechan en gran medida los recursos con los que cuenta la Commodore 64 (C64). A grandes rasgos, las limitaciones de la C64 a tener en cuenta para diseñar un programa son:

- **a.** Velocidad. El microprocesador trabaja a una velocidad bastante baja de alrededor de un megahertz. Esta velocidad no es suficiente para realizar tareas que requieran mucho procesamiento y para los juegos o aplicaciones que necesiten manipular gráficos con rapidez no queda mucho margen.
- **b.** Cantidad de memoria. La RAM se utiliza tanto para almacenar el programa y los gráficos como los datos sobre los que se trabaja. Para programas o juegos con muchas prestaciones puede ser muy escasa si no se la usa con precaución.
- **c.** Paginación de Memoria: Para acceder a toda la memoria RAM, ROM y a los periféricos disponibles en un micro que solo puede direccionar 64 kB, hay que recurrir a distintas técnicas de paginación de la memoria. Esto puede ser una gran complicación para programas que usan mucha memoria.
- **d.** Acceso directo al hardware. El programa debe lidiar directamente con circuitos integrados, diseñados principalmente con la meta de simplificar la electrónica lo más posible según las limitaciones de la época. De esta forma se sacrifica claridad en la interface de programación.
- **e.** Poco soporte del sistema operativo. La ROM de la C64 no contiene funciones para realizar operaciones avanzadas en el hardware. Tampoco ayuda en la generación de gráficos, manejo de memoria, timers o reproducción de sonidos.

PROGRAMACIÓN DE BAJO NIVEL VS. ALTO NIVEL

Por las limitaciones enunciadas previamente, para escribir un programa que se encuentre a la altura de los mejores exponentes es necesario programar en bajo nivel, accediendo directamente al hardware. Se diferencia de los lenguajes de alto nivel donde existen distintas capas de abstracción entre el programa que escribimos y los componentes físicos. Lo ideal es programar en assembler, pero también se pueden lograr resultados con lenguajes de más alto nivel como el C, si se lo usa adecuadamente.

Al programar en bajo nivel, se está más cerca del hardware que en un entorno de desarrollo de alto nivel con una API que resuelve muchos detalles tediosos de la electrónica.

* RTI Latina S.R.L., leguizamon@gmail.com

Esta cuestión hace que muchas operaciones que en un entorno más avanzado se resolverían con una llamada a una función con un nombre autoexplicativo y parámetros entendibles, esa misma operación en la C64 se debe realizar escribiendo una secuencia de bits a una dirección de memoria. Esa secuencia de bits puede agrupar datos que para la operación del programa pueden parecer no relacionados entre sí y además, la forma en que están ordenados puede no tener lógica si uno no conoce el funcionamiento de los integrados sobre los que se trabaja.

Para implementar un programa es necesario recurrir a información detallada de referencia del sistema, tanto del microprocesador (tabla de instrucciones), los circuitos integrados y el mapeo de los distintos componentes del sistema (mapa de memoria).

CARACTERÍSTICAS GENERALES

La Commodore 64 es una computadora de 8 bits, determinado por su microprocesador, un 6510 (descendiente directo del 6502 y casi idéntico) y su bus de datos, con el que se interconectan todos los componentes. El bus de direcciones es de 16 bits, lo cual es soportado por el 6502 y con él se pueden direccionar hasta 64 kB de direcciones [1]. La memoria RAM es de 64 kB y tiene 20 kB de ROM con un sistema operativo muy básico, además de un intérprete de un lenguaje Basic también limitado en comparación a las características del equipo y los dos sets de caracteres (fonts) predefinidos del sistema.

Por otro lado, emplea dos circuitos integrados bastante complejos para la época, encargados de generar video (el VIC II) y sonidos (SID). El VIC es, sin tener en cuenta al microprocesador, lo más complejo de la computadora. En cierta forma, se puede considerar a la C64 como una computadora armada alrededor de este chip. Es junto con el CPU el único integrado que accede a la memoria, es decir, que tiene por sí mismo un control de los buses de la computadora y no es pasivo como el resto de los componentes [2]. Otras computadoras de la época tenían integrados de video y sonido mucho más simples que los incluidos en este equipo y tendían a realizar más funciones en software, por lo que el microprocesador se encargaba de tareas que aquí realizan los integrados en paralelo.

Los otros componentes importantes son las dos CIAs, dos integrados iguales pero que cumplen diferentes funciones al estar conectados de distinta forma. Son los encargados de controlar los periféricos de entrada y salida (teclado, joysticks, puerto de expansión) y también cumplen otras funciones (timers, mapeo del VIC, etc.).

Antes de ver detalles del sistema, es importante conocer los componentes que lo forman. La forma en que estos componentes se conectan entre si es un tema muy importante por sí mismo, por eso primero es necesario conocerlos por separado.

VIC II. CIRCUITO DE VIDEO

Capacidades

El VIC II es lo más complejo del sistema y es el que se debe conocer en profundidad, lo cual es proporcional a la extensión que se le dedica a este apartado.

El integrado genera una imagen de 320*200 píxeles con 16 colores y es rodeada por un borde de un solo color. Dentro de esa región se muestra una pantalla que puede ser un mapa de bits, con prácticamente cualquier gráfico. Asimismo, la pantalla puede ser de texto formada por una grilla de 40*25 caracteres. Cada una de estas dos formas de mostrar la

pantalla (texto o mapa de bits) tiene sus limitaciones y es necesario conocerlas para poder diseñar gráficos que aprovechen las propiedades de la máquina.

Sprites

Una propiedad importante del VIC es la capacidad de mostrar sprites. Los sprites son gráficos que, a comparación de los caracteres de texto o bloques gráficos, flotan sobre la pantalla y no están pegados a una posición fija en una grilla. Cada uno de estos sprites reproduce una imagen cargada en memoria, pero tienen también importantes limitaciones en su uso (de cantidad, colores, tamaño...) que es útil conocer. Estos sprites están pensados para representar elementos de los juegos, por ejemplo naves o pelotas que se desplazan en posiciones arbitrarias de pantalla o también para la flecha de selección o pincel en un programa de dibujo.

Funcionamiento

Barrido de pantalla (raster)

El VIC está diseñado para trabajar con los estándares de video comunes utilizados por los televisores en la década del ’80 (NTSC o PAL). El integrado genera video a medida que se va enviando la señal al televisor. Para entender cómo funciona la generación de video es necesario conocer de forma básica el funcionamiento de un televisor de rayos catódicos. Es similar para el caso en que se utilice la computadora con un monitor de Commodore, porque este es igual que un televisor pero sin sintonizador de canales. La forma en que trabajan estos estándares de video es enviar las imágenes divididas en líneas horizontales. Cada línea se dibuja de izquierda a derecha y comenzando con la primera línea de arriba se van dibujando las líneas hasta completar la imagen.

Cantidad de cuadros por segundo, video progresivo vs. entrelazado

Entre los estándares de video hay diferencias de cantidad de cuadros por segundo (60 o 50) y cantidad de líneas (525 contra 625). En ambos casos, a causa de las características técnicas de los televisores de tubo, se trabaja con imágenes entrelazadas. Es decir, en un cuadro solo se dibujan las líneas pares y en el siguiente solo las impares y así continuamente. En consecuencia, a pesar que generan 50 o 60 cuadros por segundo (dependiendo del estándar) solo dibujan en realidad 25 o 30 imágenes completas y es común que al describir los estándares de video se mencione 25 o 30 cuadros por segundo en lugar de 50 o 60.

En el caso del VIC (como en general con las computadoras hogareñas de la década del 80) no se utiliza el entrelazado [3]. Cada uno de los 50 o 60 cuadros que se generan por segundo son independientes entre sí o video progresivo, por eso en este caso si podemos hablar de 50 o 60 cuadros por segundo. Por supuesto, esto se logra a costa de perder resolución vertical. Es similar a que solo se dibujen las líneas impares y las pares siempre quedan en negro, por eso se produce en el televisor un efecto de líneas negras, pero se evita el molesto parpadeo del video entrelazado, que para lectura de textos tiene un efecto visual notorio.

Retraso vertical y horizontal

Una característica de los estándares de video empleados, es que hay tiempos de retraso dentro del dibujo de cada cuadro, para darle tiempo al televisor a ubicarse en la próxima posición antes de comenzar a dibujar lo siguiente. El retraso más grande es el vertical, que ocurre cuando se termina de dibujar la imagen abajo a la derecha y se le da tiempo al televisor a ubicar el haz que dibuja la imagen, arriba a la izquierda para el inicio del siguiente cuadro. El otro retraso, el horizontal, es mucho más corto y en lugar de ocurrir una vez en cada cuadro, ocurre en cada línea que se dibuja al llegar a la derecha. De esta manera, es el tiempo que se retrasa antes de comenzar a mostrar la siguiente línea desde la izquierda.

El retraso vertical es muy importante para programas y juegos que hacen uso intensivo de la generación de gráficos. Durante ese tiempo el VIC no está actualizando ninguna parte de la pantalla y se pueden realizar operaciones sin que se vean en pantalla. Por ejemplo, se puede hacer un desplazamiento de la pantalla antes que se comience a dibujar el próximo cuadro. Así, el movimiento va a ser mucho más limpio y agradable que si se modificara la información mientras el VIC muestra la imagen, en cuyo caso vamos a ver efectos desagradables al dibujarse cuadros con la pantalla en proceso de modificación.

Colores

El integrado contiene una paleta interna de 16 colores que no se puede modificar de ninguna forma y no existe manera de mostrar algo con otro color. La imagen está formada por 320*200 píxeles, pero esto no significa que en cualquiera de los 64000 píxeles se pueda poner cualquier color en cualquier momento. La pantalla está organizada por celdas donde solo se pueden seleccionar un subconjunto de los colores, dependiendo del modo gráfico.

Una variación de la paleta de colores es la que utiliza el modo texto monocromático/multicolor, que usa un subconjunto de 8 colores. Esto ocurre porque de los 4 bits de la RAM de colores, uno se usa para definir el tipo de celda. Solo se pueden especificar los primeros 8 colores de la paleta para esa celda.

Lápiz óptico

El VIC permite la conexión de un lápiz óptico, este dispositivo de entrada está íntimamente ligado con el funcionamiento del VIC sobre pantallas de tubo de rayos catódicos. El lápiz informa inmediatamente al VIC cuando lee el haz de rayos catódicos en la posición a la que está apuntando. Como el VIC sabe cuál es esa posición deduce la posición *x* e *y,* a donde apunta el lápiz óptico.

Registros del integrado

El VIC tiene 47 registros con los que se configuran algunos parámetros [2]. De estos registros, 25 son para configurar posición y color principal de los 8 sprites; luego están los registros para configurar el modo de video, los distintos colores, el manejo de interrupción, lectura del lápiz óptico y otros detalles finos de la operación del integrado. En cambio, la mayoría de la información de los gráficos en sí, los obtiene directamente de la memoria de la

máquina. Puede ver hasta 16 kB, más el kilobyte de RAM de color (en la sección de mapeo de memoria del VIC se especifica cuáles son estos 16 kB).

Modos de video

El VIC es capaz de generar los gráficos de pantalla de distintas formas, cada una con sus ventajas y desventajas, con mayor o menor definición, utilizando más o menos memoria RAM y con mayor o menor flexibilidad [2]. Cada modo de video define como obtiene la información de la memoria para mostrarla en pantalla. No existe un modo de video que permita mostrar en cada uno de los píxeles (320*200, 64000 en total) cualquier color a voluntad porque ocuparía mucha memoria y seria inmanejable.

*Celdas de 8*8 píxeles*

El componente mínimo de todos los modos de videos, es la celda de 8*8 píxeles. La pantalla está formada por estas celdas distribuidas en una grilla de 40 columnas por 25 filas (40*8 = 320, 25*8 = 200). Las distintas imágenes que se pueden representar en cada una de estas celdas ocupan siempre 8 bytes consecutivos y guardan el grafico empezando por el primer byte, de arriba hacia abajo. Existen dos tipos distintos de celdas en las cuales varía la interpretación de la información de sus 8 bytes.

Celdas de 2 colores (alta definición)

En este tipo de celdas, dentro del cuadro de 8*8 se pueden representar dos colores en cada píxel. Los píxeles, cada uno definido por un bit, pueden ser de cualquiera de los dos posibles colores según el valor del bit. Se habla de alta definición porque la celda utiliza todos los píxeles que tiene disponibles.

Celdas de 4 colores (baja definición)

Estas celdas pueden mostrar hasta cuatro colores distintos a costa de perder definición. La celda pasa a ser de 4*8 pero ocupando el mismo espacio que una celda de alta definición porque los píxeles son rectangulares y tienen el doble de ancho que de alto. Cada dos bits se representan uno de los 4 posibles colores para los píxeles rectangulares. Si la pantalla completa está formada por celdas de baja definición pasa a ser de 160*200 píxeles.

Modos de video tipo mapa de bits

En los modos de video tipo mapa de bits, la información de la pantalla, es decir, las 1000 celdas de 8*8, se almacenan independientemente para cada una de ellas. Esto significa que existen 8 bytes de datos para las 1000 celdas, lo que da como resultado 8000 bytes de información.

Estos modos de mapa de bits permiten una gran flexibilidad para graficar, porque todas las celdas son independientes entre sí. Pero, la contra de este tipo de modo de video, es que justamente ocupa mucha memoria y le lleva mucho trabajo al microprocesador realizar operaciones sobre la pantalla. Por ejemplo, para hacer un scroll de pantalla haría

falta mover al menos 8 kB de memoria, lo cual no se puede hacer entre cuadro y cuadro de video.

Modos de video tipo texto

Los modos de texto son más indirectos que los de mapa de bits. En estos modos se definen, en lugar de las 1000 celdas, solo 256 bloques de 8*8 píxeles en memoria (2 kB) y se almacenan como un set de caracteres. Aunque no necesariamente son caracteres de texto porque pueden ser bloques gráficos. Posteriormente, la pantalla se define con una grilla (matriz de pantalla) de 1000 bytes (contra 8 kB de los mapas de bits) donde existe un solo byte por cada uno de las 1000 celdas visibles. Ese solo byte lo que hace es apuntar al set de caracteres y seleccionar uno de los 256 posibles caracteres. El problema de este modo es que hay mucha menos flexibilidad para dibujar imágenes en la pantalla y las celdas ya no son independientes entre sí. De las 1000 celdas que se muestran, solo puede haber como máximo 256 diferentes y tienen que repetirse la mayoría. La ventaja es que ocupan mucha menos memoria y es más fácil de manipularlos, sobre todo porque el mapa de bits de los caracteres no se suele cambiar seguido, solo se modifican su distribución en pantalla. Para hacer un scroll hay que mover 1 kB de RAM y es una tarea que se puede hacer a velocidades aceptables.

Se llaman modos de texto porque son muy útiles para texto. Si se define cada carácter con una letra, con poner en el byte de la matriz de pantalla el carácter que deseamos ver, ya mostramos una letra. Además, al ocupar menos memoria, se pueden tener distintos buffers de pantalla a la vez, que nos permite mostrar uno, trabajar sobre el otro y luego intercambiarlos.

Bloques de memoria para definir la pantalla

Dentro de la memoria a la que accede el VIC para mostrar la pantalla, se encuentran los bloques que se enumeran en Tabla 1, cada uno con una función definida.

Matriz de pantalla (1 kB) (grilla de caracteres / colores de mapa de bits)	Esta matriz de 40*25 tiene un byte por cada celda de 8*8 de la pantalla y se utiliza siempre, sea modo texto o modo mapa de bit. En los modos de texto, cada byte representa un carácter que se saca de la memoria de definición de caracteres y según el modo algunos bits pueden tener la función de elegir cual color de fondo usar para el carácter En los modos de mapa de bit cada byte en lugar de tener un carácter, tiene dos colores (4 bits por color) que se van a usar para especificar los colores de la celda.
Mapa de bits (8 kB)	Solo se usa en los modos de mapa de bits. Cada 8 bytes se define el contenido de una celda de pantalla.
Definición de caracteres (2 kB)	Solo utilizado en los modos de texto. Cada 8 bytes define un carácter (mapa de bits de 8*8) y se definen en total 256 caracteres.

Tabla 1.

El VIC maneja 16 kB de memoria y se puede configurar (en los registros del VIC) para que dentro de ese tamaño, estos tres tipos de bloques estén ubicados en cualquier parte,

siempre y cuando sean múltiplos de su tamaño. Por ejemplo, la matriz de pantalla ocupa 1 kB así que, se le puede dar 16 distintas ubicaciones dentro de los 16 kB, desde el inicio de la memoria del VIC y avanzando de a 1 kB. El mapa de bits ocupa 8 kB, así que solo existen dos posibles posiciones, al principio y en la mitad de los 16 kB. La definición de caracteres tiene 2 kB y se puede ubicar en 8 lugares distintos, todos separados por 2 kB entre sí.

Colores de la pantalla: comunes y propios de las celdas

En los modos de alta definición van a haber 2 colores por cada celda para elegir entre sus 64 píxeles y en los de baja definición van a ser cuatro los colores posibles para cada píxel (32 en este caso). El problema es de donde se obtienen y cuáles son, los dos o cuatro colores de cada celda. Cada modo de pantalla tiene una forma distinta de obtener los colores, cada uno con sus pros y contras.

En los modos de mapas de bits, configurar el color de cada celda es más flexible porque se usa la matriz de pantalla junto con la RAM de colores. Pero, en los modos de texto la matriz de pantalla se usa para guardar los caracteres así que hay menos información de color por cada celda. El resto de los colores que no son propios de la celda, son colores globales a todas por lo que quita mucha libertad para graficar.

Lista de modos de pantalla

Los modos de video que soporta el VIC II (Tabla 2 y 3), tienen distintas combinaciones entre texto y mapa de bits, y celdas de alta o baja definición, además de selección de origen de colores usados en las celdas.

Texto Standard	Texto en alta definición, usa 1 kB de pantalla, más 2 kB de caracteres (si usa la ROM de caracteres ahorra 2 kB de RAM). Los colores se retiran del color de fondo común a toda la pantalla y de la RAM de color. Hay un solo color propio de cada celda. Es el modo estándar en la máquina, el que usa el Basic
Texto con alta o baja definición	Es similar al modo anterior (ocupa la misma cantidad de memoria), con la diferencia que cada celda puede ser de alta o baja definición de forma independiente. En cada celda se determina si es de alta o baja definición por el bit 3 de color de la RAM de color. La limitación que tiene es que solo se pueden usar 3 bits de color de la RAM, por eso solo se puede asignar a la celda uno de los primeros 8 colores de la paleta. Para las celdas de alta definición, los colores se sacan del color de fondo, para los de baja definición, se usan además los otros dos colores de fondo (comunes a toda la pantalla)
Mapa de bits de alta definición	Cada celda puede tener 2 colores propios que se sacan de la memoria de la matriz de video (un color en cada *nibble*). Ocupa 8 kB el mapa de bits, más 1 kB de colores. La RAM de colores no se usa.

Tabla 2. Modos de video del VIC II (Parte 1).

Mapa de bits de baja definición	Es similar al modo anterior, pero con las celdas de baja definición, la pantalla queda en 160*200. Existe un color común a toda la pantalla que es el color de fondo, luego cada celda tiene 3 colores propios, 2 se sacan de la matriz de pantalla (uno por *nibble*) y el otro de la RAM de color que en este modo si se usa.
Texto de alta definición con selección de fondo	Este modo es casi idéntico al de texto estándar, salvo por la diferencia que permite seleccionar para cada celda uno entre cuatro colores de fondo. Para lograr esto, se utilizan solo 6 bits de la matriz de video para guardar el carácter, con lo cual solo se pueden mostrar hasta 63 caracteres distintos por celda. Los 2 bits superiores del carácter se usan para seleccionar cual color de fondo usar, en lugar de usar siempre el color de fondo común, permite usar el color de fondo 1,2 y 3 (distintos registros del VIC)
Pantalla deshabilitada	Se muestra la pantalla con el color del borde solamente, no hay ningún contenido

Tabla 3. Modos de video del VIC II (Parte 2).

Sprites

Los sprites son iguales para todos los modos de video y no son afectados por la configuración del VIC, salvo en el caso de la pantalla deshabilitada donde no se muestran tampoco los sprites. Los sprites tienen una definición de 24*21 píxeles (3 celdas de ancho por 2 y media de alto). Cada sprite tiene una posición en la pantalla *x* e *y*. El sprite puede estar en cualquier píxel de la pantalla e incluso puede no entrar entero. Esto ocurre porque la posición 0 no se encuentra dentro del área visible, sino dentro del borde. A medida que se incrementa la coordenada va entrando al área visible. Cada sprite ocupa 64 bytes de memoria, aunque solo se usan 24*21 bits = 3*21 bytes lo que da 63 bytes. El último byte no cumple ninguna función.

Sprites monocromáticos o multicolor (alta o baja definición)

En el caso de los sprites ocurre lo mismo que en los bloques de la pantalla. Cada bit puede representar un píxel para los sprites monocromáticos (color o transparente) o cada dos bits se pueden representar 4 colores distintos (en realidad 3 colores o transparente) para cada píxel rectangular de ancho doble. Los sprites monocromáticos tiene un solo color asignado y cada uno de los píxeles puede ser de ese color o transparente. Cuando es transparente se muestra lo que hay debajo del sprite. Los sprites multicolor, además del color propio del sprite y del transparente, tienen otros dos colores que son comunes a todos los sprites multicolor. Esto de los dos colores comunes es una limitación importante a lo que se puede mostrar en varios sprites. La otra limitación de los sprites multicolor es la poca definición, 12*21 con píxeles rectangulares doble ancho.

Sprites doble ancho y doble alto

Una opción que tiene cada sprite de forma independiente, es duplicar el ancho y/o el alto de cómo se muestra. El sprite sigue teniendo la misma definición pero se duplica el tamaño en pantalla, lo que significa que los píxeles son el doble de alto y/o ancho según el

caso. Si el sprite se lo representa como doble ancho y a su vez multicolor, los píxeles van a tener cuatro veces el ancho normal.

Superposición y colisión de sprites y pantalla

Cuando dos sprites se muestran en el mismo lugar, uno se mostrara por encima del otro. La prioridad va a depender del número de sprite. El sprite 4 siempre se va a mostrar por encima del sprite 2, pero por debajo del sprite 5. Esta cuestión hace extremar los cuidados al seleccionar lo que va a mostrar cada sprite, porque hay que tener en cuenta cual se tiene que ver por delante. Si ocurre que en un momento uno debe pasar del fondo hacia el frente, hay que intercambiar los sprites. El VIC también nos indica con un bit por sprite, si el sprite colisionó con otro. Es decir, que un píxel no transparente del sprite se esté dibujando en la misma posición del píxel no transparente de al menos otro sprite. Por ejemplo, esta información es útil para los juegos, para detectar que un objeto esté tocando a otro y aunque la información es bastante limitada, luego el programa tiene que detectar con que chocó y en qué lado.

Rotación, espejado y escalado de sprites

Excepto lo antes detallado, no hay otras manipulaciones que se puedan hacer sobre los sprites o al menos no de forma directa. Se pueden obtener efectos modificando los atributos del sprite a medida que el VIC va barriendo la pantalla, pero es una operación bastante complicada. Operaciones que serían útiles como espejar el sprite, no las realiza el VIC. Si queremos mostrar una misma imagen apuntado hacia un lado y hacia el otro, habrá que tener dos imágenes distintas y cambiar de imagen para girarlo. El escalado solo se puede realizar, si se muestra a dos escalas (normal o doble) pero no es posible elegir una arbitraria.

Memoria de los sprites

En los registros del VIC se guarda la posición y atributos de los sprites (Tabla 4). Los gráficos (hasta 256) se guardan en la RAM y la indicación de cual de esos posibles gráficos debe mostrar cada sprite (puntero del sprite) se encuentran también en RAM en lugar de en el VIC.

Utilización de más de 8 sprites a la vez

Es posible utilizar más de 8 sprites a la vez en la pantalla y de hecho es muy común en los juegos más avanzados. Básicamente no es que se usen más de 8 sprites a la vez porque esta es la cantidad de sprites que existen, sino que a medida que se va dibujando la pantalla se van cambiando los sprites de posición y se reutilizan sprites que estaban en la parte superior de la pantalla para mostrarlos más abajo. Esta técnica es compleja porque hay que estar sincronizado con el barrido de pantalla.

Gráficos de sprites (hasta 16 kB)	Cada 64 bytes se define un sprite, toda la memoria que no se usa para matriz de pantalla, mapa de bits o definición de caracteres se puede usar para sprites. Como máximo teórico pueden ser 256, aunque siempre existe al menos una matriz de pantalla de 1 kB y una definición de caracteres de 2 kB, así que el máximo posible son 208 sprites. Si se usa un buffer de pantalla (1 kB) hay que restar otros 16, y por cada tabla de definición de carates otros 32.
Punteros de sprites (8 bytes)	Por cada uno de los 8 sprites hay que especificar cuál de los posibles 256 gráficos de sprite tiene que dibujar. Para especificar eso, se usan 8 bytes que están al final de la matriz de pantalla (como tiene 1024 bytes y se usan 1000 para caracteres, le sobran 24). Hay que tener en cuenta que si se cambia de matriz de video, también se cambian los punteros de los sprites.

Tabla 4.

Bordes

El borde es el área de la pantalla en que no se muestra información. La pantalla de 320*200 ocupa solo una región de la pantalla total y el resto se llena con un solo color. Existen formas avanzadas de mostrar sprites en el borde, pero hay que realizar operaciones sobre el VIC en momentos adecuados del refresco de pantalla y siempre de forma dinámica, por lo que el programa tiene que estar atento al VIC. La idea es engañar al circuito para hacerle creer que no termino de dibujar la pantalla. Esta tarea es más sencilla si se realiza para aprovechar los bordes de abajo y de arriba porque hay que sincronizarse con determinada línea. En cambio, para aprovechar los bordes izquierdo y derecho, en cada línea hay que estar atento al refresco de pantalla y el procesador ya no le queda tiempo para hacer otra cosa.

Barras de colores en cargadores

Los cargadores turbo suelen tener un efecto de barras de colores al azar en la pantalla. Esto se logra cambiando el color de fondo de la pantalla más rápido que el dibujo de la pantalla completa. Si por ejemplo, se dibujan 60 cuadros por segundo y se cambia el color de fondo 120 por segundo, se va a poder observar la pantalla cortada en dos de forma horizontal. Esto ocurre porque el VIC todo el tiempo está dibujando la pantalla con la configuración actual de los registros y el contenido actual de la RAM. Si lo cambiamos mientras dibuja, se van a dibujar una misma pantalla con distintos datos. Si en vez de dos veces por pantalla se cambia el color del fondo lo más rápido que soporta el micro, se van a dibujar montones de líneas de un píxel de alto y unos cuántos píxeles de ancho en toda la pantalla. Los cargadores no cambian la información en pantalla lo más rápido posible porque están ocupados cargando el programa, pero lo hacen unas cuantas veces por pantalla.

Efectos de movimiento en barras de colores

Si el efecto de cambiar el color de fondo mientras se dibuja se lo hace de forma coordinada y por ejemplo se cambia al llegar a la línea 100 y se vuelve al color original 20 líneas después, se va a mostrar una barra horizontal de 20 píxeles de alto. Es muy común este efecto en animaciones en la C64, que implica que en cada cuadro se muestre la línea en una posición ligeramente distinta a la anterior, haciendo que se mueva la barra. También, se

puede hacer que dentro de la barra se cambie el color por cada línea y se tenga una barra con una cantidad de colores muy superior a lo común en la C64.

Scroll

Como la pantalla se encuentra formada por celdas de 8*8, al hacer un desplazamiento simplemente moviendo bytes, los vamos a tener que mover de a 8 píxeles a la vez (tanto de forma horizontal como vertical). En un momento la pantalla está en una posición y en el siguiente cuadro se movió 8 píxeles rápidamente. Esto es un movimiento muy brusco y es mucho más agradable a la vista que la pantalla se mueva de a un píxel por vez. Para poder hacer este movimiento suave, el VIC cuenta con dos registros de scroll suave y que pueden mostrar la pantalla con desplazamiento de 0 a 7 píxeles verticalmente y otros 0 a 7 horizontalmente. Si a esta posición fina de 8 píxeles la combinamos con el desplazamiento de memoria, podemos tener un desplazamiento de pantalla de a un píxel.

Por ejemplo, tenemos la pantalla con un desplazamiento de 7 píxeles a la derecha, pasamos 6, luego a 5, 4, 3, 2, 1, 0. El siguiente paso es hacer el movimiento de la memoria de la pantalla y la pantalla se corre 8 píxeles. De esta manera, al mismo tiempo se vuelve a dejar el desplazamiento en 7. Así, con estos dos movimientos coordinados se ve como un desplazamiento suave. En este caso, el problema es que se vería en los bordes cuando pasa de 0 a 7 un movimiento brusco de 8 píxeles a pesar que dentro se vería perfecto. Para esto el VIC tiene una opción de agrandar los bordes en 8 píxeles. Con esto, la pantalla sigue siendo de 40*25, aunque completo solo podríamos ver 39*24 (una fila y una columna oculta) aunque cambiando los registros de desplazamiento se verían por ejemplo las 40 columnas pero dos de las columnas estarían parcialmente cubiertas. A pesar de achicarse aún más la pantalla visible, es lo que se utiliza normalmente en los juegos en los que es necesario un scroll suave.

Interrupciones

El integrado genera interrupciones para que atienda el microprocesador en estos casos:

- **a.** Al generar una línea en particular y si previamente se configuro para que dispare la interrupción al llegar a esa línea.
- **b.** Cuando ocurre una colisión entre un sprite y la pantalla.
- **c.** Con la lectura del lápiz óptico.

Es común utilizar la interrupción de barrido de pantalla, así se pueden realizar efectos cuando se está dibujando determinada porción de la pantalla. Por ejemplo, para cambiar de modo y tener la pantalla dividida en dos, arriba en modo texto y abajo en modo gráfico. Por otro lado, también se utiliza para usar los 8 sprites de distinta forma en regiones diferentes y así tener más de 8 sprites a la vez. De todas formas, estos efectos se pueden lograr sin interrupciones leyendo todo el tiempo el estado del VIC desde el programa.

COMPONENTES BÁSICOS DEL SISTEMA

Aparte del VIC hay otros integrados que componen al sistema y es necesario conocer.

Microprocesador 6510

El microprocesador incluido en la C64 es el 6510 que es un derivado directo del 6502 con el agregado de un puerto de entrada/salida. Para manejar este puerto se utilizan las dos primeras direcciones de memoria (0000 y 0001) que ya no están disponibles para otro uso. El resto de las características es idéntico. Se puede considerar este puerto de datos como un periférico externo y porque de hecho sería fácil implementarlo externamente en un 6502 sin requerir ninguna modificación del microprocesador, aunque en este caso se encuentra incluido dentro del integrado. En el integrado, se encuentran disponibles 6 patas que se pueden configurar para usar como entradas o como salida individualmente. En el sistema, cada una tiene una función fija asignada, así que no será posible decidir cuál es entrada y cual es salida.

RAM principal (64 kB)

Esta es la RAM principal del sistema y desde aquí es donde corren los programas escritos o cargados por el usuario, y también en donde se encuentran los gráficos que muestra el VIC. Estos 64 kB de RAM son los que le dan nombre a la computadora. A pesar de eso, el equipo cuenta con 512 bytes más de RAM en la memoria de color. Se pueden utilizar casi todos los bytes de estos 64kB. La excepción son los dos primeros (0000 y 0001) que no pueden ser accedidos ni para lectura o escritura por el microprocesador porque esas direcciones las usa internamente para el puerto de entrada/salida. La mayor parte de la RAM (la parte baja) es visible siempre, el resto comparte el espacio con los demás integrados del sistema y hay que recurrir al mapeo de memoria para accederlo.

ROM de Kernal (8 kB)

Esta ROM tiene las rutinas de más bajo nivel para manejo de entrada/salida, entrada y salida de caracteres, funciones de básicas para mostrar textos en pantalla, cargar y grabar programas, manejo de teclado, etc.

ROM de Basic (8 kB)

Esta ROM contiene el intérprete del lenguaje Basic junto con su interface de pantalla.

ROM de caracteres (4 kB)

Aquí se encuentran definidos dos sets de mapas de bits de 256 caracteres de 2 kB cada uno. El primer set es el de los caracteres en mayúsculas con caracteres gráficos y el segundo set son los caracteres en minúscula y mayúsculas. El VIC dibuja las letras con la información obtenida de esta ROM. Solo trabaja con un solo set de 256 caracteres a la vez.

RAM de colores (1 kB de 4 bits)

Esta RAM almacena los colores de los caracteres (dependiendo del modo grafico del VIC). Es visible tanto desde el microprocesador como desde el VIC y salvo cuando el video esta desactivado, siempre se está empleando para generar la imagen. No tiene utilidad para

otra cosa porque solo guarda los 4 bits inferiores de cada byte para especificar un color entre 0 y 15. Es importante saber que al leer esta memoria en los 4 bits superiores del byte, los que no existen en la memoria se leen como valores indeterminados y su información es prácticamente aleatoria. Si uno escribe un programa que lee información de esta memoria, el programa va a funcionar de forma extraña porque en una maquina puede leer esos bits siempre en 0 y en otra siempre en 1, o pueden variar de un momento al otro. Lo mismo va a ocurrir si se escribe un valor y se efectúa la lectura esperando encontrar el mismo valor.

CIA 1 y 2. Manejo de periféricos y timers.

En el sistema hay dos integrados CIA iguales, pero conectados a distintos periféricos. La función de cada CIA es múltiple:

- **a.** Interface de líneas de entrada y salida. Maneja 16 líneas de entrada y salida. Se puede configurar cada línea como entrada o como salida e internamente tienen *pull-ups* para poder controlar entradas de interruptores de forma sencilla.
- **b.** Timers. Tiene dos contadores de ciclos configurables.
- **c.** Timer con interrupción. Una de las CIAs puede generar interrupciones con su timer para uso del programa. La otra también puede producir una interrupción pero una no enmascarable de uso limitado para los programas comunes.
- **d.** Reloj de tiempo real. Mientras y desde que el equipo se encuentra encendido, lleva un contador de horas, minutos y segundos. Este reloj de tiempo real no está protegido por una batería, por eso con cada reinicio del sistema se pierde la fecha y hora.

SID. Sonido.

Este integrado es el generador de sonidos. Genera tres voces independientes y cada una de las voces puede reproducir un sonido de determinada frecuencia, forma de onda y duración sin importar lo que hacen las otras voces. Por ejemplo, para reproducir una música, con las tres voces se pueden reproducir tres instrumentos, cada uno con distinto tipo de sonido. De esta manera, se puede escuchar un piano junto a un bajo acompañados por una batería. Cada uno de los instrumentos/voces solo podrá tocar una sola frecuencia sin acordes y cada voz tiene un generador de ondas simple [4],[5].

Tipo de onda

Cada voz se configura para el tipo de geometría de onda que va a generar. Esta geometría es de cada cresta de la onda. Por ejemplo, si es triangular y se genera sonido a 440 hertz, se van a generar en un segundo 440 crestas de forma triangular. La geometría determina el timbre del sonido; una misma frecuencia tocada con distintas geometrías va a producir sonidos notablemente distintos. Los distintos tipos de onda son los siguientes:

- **a.** Cuadrada: es la más simple, no hay tiempo de transición entre el valle (el punto más bajo) y la cresta (el punto más alto). Necesita que se configure el ancho de pulso, que es el tiempo en que la onda está arriba. Para distintos anchos de pulsos se obtienen distintos timbres.

b. Triangular: la onda sube linealmente hasta llegar a la cresta e inmediatamente empieza a bajar hasta volver al valor mínimo
c. Diente de sierra: similar a la onda triangular, pero solo sube linealmente y luego cae de golpe
d. Ruido: no tiene una forma definida, es ruido azul.

Forma envolvente

Otra de las características de la voz que define como va a sonar es la forma de la onda envolvente. Básicamente, lo que determina es como es el volumen del sonido generado desde que se dispara el sonido hasta que se detiene. Las etapas son:

a. Duración de ataque: se configura cuánto tarda el sonido desde que se dispara hasta llegar al volumen máximo (el volumen máximo que tiene el SID). Puede ser un ataque rápido, por ejemplo de un disparo o una batería, o uno más lento de una flauta.
b. Duración del decaimiento: es el tiempo que tarda desde que se llegó al volumen máximo durante el ataque, para caer hasta el volumen de sostenimiento.
c. Volumen de sostenimiento: es el volumen al que permanece la voz hasta que se apague el sonido.
d. Duración de relajación: es el tiempo que tarda desde que se apaga el sonido hasta que el volumen de la voz se hace cero. La sigla en inglés por la que se conoce esta función es ADSR (Attack Decay Sustain Release).

Configurando distintas formas de la envolvente de la voz se pueden producir sonidos de características muy distintas.

Técnicas avanzadas

El SID permite reproducir sonidos más complejos en algunos casos combinando sonidos de dos voces para usar la salida de una para modular a la otra; también tiene controles de reverberación.

Reproducción de samples

Existen técnicas rebuscadas para reproducir samples en el SID, función para la que no fue pensado. Aprovechando un defecto del SID, cada vez que se cambia el volumen general (valor de 4 bits, de 0 a 15) produce un ruido que sale al máximo del nuevo volumen. Si se transfirieren los muestreos de la onda al control de volumen, se reproduce la onda del sample por medio de ese ruido. Por supuesto que la RAM no alcanza para guardar samples muy largos, pero en los juegos solía haber frases cortas grabadas con esta técnica [5].

Conversor analógico digital

Aparte de su función de generador de audio tiene dos conversores analógico-digital para digitalizar la entrada de los pads (potenciómetros). Esta entrada también se utiliza para

el mouse, dispositivo para el que no estaba diseñado el equipo pero que se adaptó aprovechando esta entrada.

PLA, unión de todos los integrados.

En realidad, este integrado no se accede en ningún momento desde el programa y es completamente invisible, pero es el que controla todas las conexiones entre los distintos integrados del sistema. La única forma de alterar el mapeo es desde el puerto de expansión que desarrollaremos en otro artículo. Para ver el trabajo realizado por la PLA hay que ver la parte de mapeo de memoria de la C64. Este integrado a su vez se vale de la ayuda de otras compuertas lógicas para controlar a todos los integrados [7].

Control desde el puerto de expansión

Desde el puerto de expansión utilizado por los cartuchos, hay dos líneas que se conectan directamente a la PLA y que afectan de forma notoria el mapeo que realiza de la memoria [6]. Estas líneas son /GAME y /EXROM. Los detalles de funcionamiento son relativamente complejos y dependen de las otras entradas de la PLA, pero lo importante es saber que con estas líneas se indica que se empleen las memorias ROM del cartucho en reemplazo de la ROM de Basic o de un bloque de RAM. No hay forma de controlar estas dos líneas desde el microprocesador, por eso si el cartucho no tiene forma de deshabilitar estas líneas, no se podrá acceder a toda la memoria de la maquina mientras esté conectado. Los cartuchos de Fast Load suelen tener un circuito que deshabilita estas líneas cuando el cartucho se inutiliza desde el programa para que el equipo pueda usar toda la memoria de la forma habitual.

CONCLUSIONES

Hemos analizado en detalle el circuito de video de la Commodore 64, junto con los componentes básicos del sistema. Es importante conocer estos temas, dado que los programas escritos para esta computadora superan las limitaciones propias de la máquina, como por ejemplo mapear la RAM sobre la ROM para obtener menor tiempo de procesamiento. El conocimiento de esta y otras técnicas permiten aprovechar eficientemente los recursos disponibles del sistema.

En otro artículo, exploraremos la conexión de los componentes del sistema y cómo se comportan [8], además de examinar las modernas aplicaciones que se pueden utilizar y que son de gran ayuda para escribir un programa.

AGRADECIMIENTOS

A Pablo Roldán por sus correcciones y sugerencias.

BIBLIOGRAFÍA

[1] F. Monteil (1985), Como programar su Commodore 64: Basic, Gráficos, Sonido. Paraninfo.
[2] C. Bauer (1996), The MOS 6567/6569 video controller (VIC-II) and its application in the Commodore 64. http://www.zimmers.net/cbmpics/cbm/c64/vic-ii.txt. (27 de septiembre de 2011).
[3] Wikipedia, The Free Encyclopedia (2011), Interlaced Video. http://en.wikipedia.org /wiki/Interlaced_video (27 de septiembre de 2011).

[4] Commodore Business Machines, Inc (1982), *Commodore 64 Programmer's Reference Guide.* Commodore Business Machines, Inc. and Howard W. Sams & Co.
[5] D. Kubarth (2004), Sid in-depth information site. http://sid.kubarth.com (27 de septiembre de 2011).
[6] J. West y M. Mäkelä (1994), 64 doc, v. 1.8. http://www.viceteam.org/plain/64doc.txt (27 de septiembre de 2011).
[7] Commodore Business Machines, Inc (1985), *Service Manual Model C64 Computer* Commodore Business Machines, Inc.
[8] M. Leguizamón (2011), Programando la Commodore 64 en serio. Conexión de componentes y modernas aplicaciones, *Revista de Tecnología e Informática Histórica* 1, pp. 75-90.

PROGRAMANDO LA COMMODORE 64 EN SERIO
CONEXIÓN DE COMPONENTES Y MODERNAS APLICACIONES

Marcos Leguizamón*

Los distintos circuitos integrados que conforman el sistema fueron analizados anteriormente [1], por lo que ahora es necesario conocer la forma en que se interconectan entre sí. Además, es interesante conocer un conjunto de herramientas que ayudan a generar programas y datos desde otras computadoras modernas con distintos sistemas operativos. De esta manera, se logra vincular pasado y presente con el desarrollo de nuevos productos para la Commodore 64.

CONEXIÓN DE COMPONENTES EN EL SISTEMA

Diferencias de velocidad entre sistemas de video

Hay distintas versiones de la Commodore 64 para distintos sistemas de video (NTSC, PAL-N, PAL-B). La diferencia entre las distintas versiones no acaba en la generación de video sino que afecta a todo el sistema. En el caso de la C64 hay un solo cristal que genera una frecuencia. De ese cristal, por medio de divisores digitales se obtienen las dos frecuencias necesarias: la del CPU (que idealmente no debería depender del estándar de video) y la señal de reloj para generar la salida de video. Como lo importante es que la salida de video tenga la frecuencia correcta para que la imagen sea válida, se emplea un cristal y los divisores adecuados para tener esta frecuencia y la frecuencia del CPU es la más cómoda según el cristal original y los divisores utilizados. La consecuencia es que el CPU corre a distintas velocidades para los distintos estándares de video (Tabla 1).

NTSC	1.023 MHz
PAL-B	0.985 MHz
PAL-N	1.023 MHz

Tabla 1.

Mapeo de memoria

El microprocesador solo es capaz de direccionar hasta 16 bits, que es el ancho del bus de direcciones, no se puede superar este límite. Esto significa que el programa solo puede manejar 65536 posiciones distintas de memoria y todas las operaciones que realiza el

* RTI Latina S.R.L., leguizamon@gmail.com

microprocesador deben realizarse dentro de este rango de direcciones. La forma más fácil de organizar los componentes del sistema es asignarle un rango de direcciones a cada uno, para que luego el programa pueda dirigirse a cada integrado según en qué rango de direcciones escriba o lea. La complicación que surge enseguida en el caso de la C64, es que la cantidad de memoria RAM y ROM y periféricos que se pueden direccionar supera ampliamente los 64 kB (Tabla 2).

Memoria RAM principal	64 kB
Memoria ROM Basic	8 kB
Memoria ROM Kernal	8 kB
Memoria RAM de colores	1 kB (1024 direcciones de 4 bits)
Memoria ROM con caracteres	4 kB
VIC	1 kB (47 bytes repetidos)
SID	1 kB (29 bytes repetidos)
CIA 1	256 bytes (16 bytes repetidos)
CIA 2	256 bytes (16 bytes repetidos)
Memoria ROM de Cartucho (opcional)	8 kB
Memoria ROM de cartucho alto (opcional)	8 kB
I/O externa 1	256 bytes
I/O externa 2	256 bytes

Tabla 2.

Como se puede observar, si se suman todas las memorias y periféricos, y aunque no tengamos en cuenta a los opcionales (cartucho de expansión), se superan ampliamente los 64 kB de direcciones. Para poder manejar todas estas direcciones es necesario recurrir a una técnica que es sin duda una de las mayores complicaciones del sistema: la paginación de memoria.

Paginación de memoria

El mapeo de memoria se realiza principalmente desde el puerto interno de I/O del 6510 (en direcciones 0000 y 0001) [2]. Tres de las líneas de salida de este puerto se llaman: LORAM, HIRAM y CHAREN. A pesar que tienen nombres específicos que refieren a la RAM inferior, la RAM superior y la ROM de caracteres, la forma en que se mapea es más compleja que una línea manejando un bloque de memoria. El mapeo se define según las distintas combinaciones de estas tres líneas. La única línea que cumple una función fija es CHAREN, que elige sí mapeamos las I/O y la memoria de color o en el mismo lugar mapeamos la ROM de caracteres. Sin entrar en especificaciones de los valores para realizar cada mapeo, las distintas formas de mapear el equipo se puede observar en Tabla 3.

Mapeo de ROM de caracteres

Cuando usamos un mapeo en el que podemos acceder en el área D000-DFFF a las entradas salidas (VIC, SID, CIAs, expansión de usuario) y la RAM de color, podemos seleccionar en su lugar ver la ROM de caracteres en este bloque. Esto se hace con la línea CHAREN, en cualquiera de los modos de mapeo que estemos usando. Si usamos toda la

RAM, por supuesto no cumplirá ninguna función. Poder leer la ROM de caracteres no es algo muy útil normalmente, solo sirve cuando deseamos hacer alguna modificación (copiándola a RAM) sin tener que tener guardados los 4 kB que ocupan los dos sets de caracteres en disco. En circunstancias normales en este bloque vamos a acceder a la RAM o los periféricos y la RAM de color.

	Default	Sin ROMs y con I/O	Kernal con I/O	Todo RAM
E000-FFFF	Kernal	RAM	Kernal	RAM
D000-DFFF	I/O y Color	I/O y Color	I/O y Color	RAM
C000-CFFF	RAM	RAM	RAM	RAM
A000-BFFF	Basic	RAM	RAM	RAM
0002-9FFF	RAM	RAM	RAM	RAM

Tabla 3.

Paginación cuando se encuentra conectado un cartucho

Cuando hay un cartucho conectado y tiene las líneas /GAME o /EXROM (ver sección de PLA) habilitadas, todo el mapeo de memoria es distinto al mostrado aquí.
Si se va a desarrollar un programa para cartucho hay que tener en cuenta esto. Lo más sencillo es deshabilitar el cartucho desde software una vez que se cargó el programa en RAM y así utilizar la máquina de la forma habitual. Si el cartucho no tiene un circuito que le permita deshabilitarse, el programa hay que desarrollarlo teniendo en cuenta este mapeo distinto. Por este motivo no es posible copiar un programa de cartucho a disco y ejecutarlo normalmente sin el cartucho, o el caso opuesto, pasar un programa de disco a cartucho.

RAM oculta detrás de la ROM

Un caso especial que ocurre en el mapeo de la memoria, es la escritura de RAM en áreas donde esta mapeada memoria ROM. Si escribimos sobre la memoria ROM por supuesto no podemos modificar su contenido, pero la PLA se encarga que aunque estemos accediendo a un área de ROM, cuando la operación es de escritura, la operación se redirecciona a la RAM correspondiente (el bloque de la RAM de 64 kB oculta). Por ejemplo, si escribimos el valor XX a la dirección B000h y luego lo leemos, no vamos a obtener el mismo valor. El valor XX se escribió sobre la RAM (esa área de RAM se encuentra invisible en la configuración actual) pero cuando leemos, sacamos el byte de la dirección de ROM.

Esta función es práctica para los siguientes casos:

- **a.** Al realizar copias de ROM antes de switchear el paginado. Leemos un bloque de ROM y lo que vamos leyendo lo escribimos sobre la misma dirección. Por ejemplo, al leer el byte en B005 y guardarlo otra vez en B005, al ser una operación que sin conocer la paginación puede parecer inútil. Luego cambiamos el paginado para ver la RAM y el contenido de la RAM en este momento es idéntico al de la ROM.
- **b.** Cuando se guardan gráficos para el VIC. El VIC tiene su propio mapeo de memoria como veremos más adelante, si se configura para leer los gráficos a mostrar desde el área de memoria donde el microprocesador mira la ROM, podemos aprovechar la RAM oculta por la ROM para guardar gráficos y a la vez seguir utilizando las rutinas

de ROM; sin tener que hacer cambios temporales en el paginado y otros detalles importantes (deshabilitación de interrupciones, etc.).

Mapeo de Memoria del CPU vs. Mapeo del VIC.

A pesar que desde el programa se accede a las memorias y los periféricos utilizan un determinado mapeo de direcciones, hay que tener en cuenta que este es el mapeo visible al microprocesador. Sin embargo, el VIC (que es el otro integrado que accede la memoria), tiene un mapeo independiente. El VIC a diferencia del CPU, solo direcciona 14 bits (16 kB). Dentro de esos 16 kB de direcciones el VIC necesita ver distintos bloques de memoria con el contenido que se muestra en Tabla 4.

Modo gráfico	Mapa de bits (8 kB)	Pantalla completa en mapa de bits.
Modo texto	Definición de caracteres (2 kB)	Mapa de bits de los posibles 256 caracteres.
Siempre	Matriz de pantalla (1kB)	Caracteres (texto) o colores (mapa de bits).
	Sprites (hasta 16 kB)	Mapa de bits de los sprites. Toda la RAM de video que quede libre para otros usos, se puede usar para sprites.

Tabla 4.

El VIC también lee la RAM de colores, pero esta acción es por fuera de los 16 kB de mapeo a la memoria. El VIC siempre es capaz de leer esa memoria a diferencia de la RAM principal. Esto lo logra, porque el VIC II en realidad tiene un bus de datos de 12 bits en vez de 8. Así, lee los 8 bits del bus común al mismo tiempo que los 4 bits de la RAM de colores.

La memoria que ve el VIC es siempre la RAM principal de 64 kB, con la excepción de la ROM de caracteres que se mapea en su área de direcciones según que bloque de RAM este mirando, por eso existen dos áreas de 4 kB de RAM no visibles desde el VIC al ser tapados por la ROM de caracteres. Existen cuatro formas de mapear la memoria desde el VIC (Tabla 5) que básicamente dividen la RAM de 64 kB en 4 bloques de 16 kB. Según el bloque, se inserta en el medio la ROM de caracteres para que se vea desde el VIC.

0 default	RAM 0000-0FFF ROM de caracteres en 1000-1FFF RAM 2000-3FFF
1	RAM 4000-7FFF
2	RAM 8000-8FFF ROM de caracteres en 9000-9FFF RAM A000-BFFF
3	RAM C000-FFFF

Tabla 5.

Como se puede observar, los bloques 0 y 2 no dejan ver al VIC toda la RAM por tener que leer los 4 kB de la ROM de caracteres. Estas dos configuraciones son útiles para programas que trabajan en modo texto y usan las fonts predefinida en la máquina. Para los

juegos que necesitan muchos gráficos y que no hacen uso de los caracteres predefinidos es preferible configurar al VIC para usar los bloques 1 o 3; así se pueden definir 16 kB completos de gráficos (sumando pantallas, buffer temporal, set de caracteres y sprites).

Último bloque de mapeo del VIC

El último bloque además de poder usar los 16 kB completos (igual que el segundo) tiene una ventaja extra. En el rango D000-DFFF el microprocesador es donde puede acceder a los periféricos y la RAM de color, y es útil poder acceder a este bloque en cualquier momento para realizar una operación con un integrado, por eso conviene tener siempre mapeados los periféricos. El problema es que si se tiene siempre mapeados los periféricos se desperdician 4 kB de RAM que están ocultos detrás. Lo que se puede hacer para aprovechar las dos cosas es, al momento de iniciar el programa, mapear en ese bloque la RAM en lugar de los I/O y volcar en esa RAM los datos gráficos que no se modifican durante la ejecución del programa (sprites y/o definición de caracteres). Una vez cargada esta información en este bloque, se vuelven a mapear para ver las I/O en este rango de direcciones. Ahora, desde el microprocesador se puede trabajar sobre las I/O y el VIC puede leer los gráficos de la RAM oculta. La otra ventaja de este bloque es que permite que el programa pueda usar la RAM desde la parte más baja hasta por lo menos C000 sin ser cortada por el área de buffer de pantalla. Por otra parte, es más cómodo el manejo de bloques grandes de información.

Doble buffer de pantalla

En algunos casos es útil trabajar sobre dos buffers de pantalla, uno es el que muestra el VIC y otro es el que modifica el programa. De esta forma, se pueden hacer cambios en un buffer mientras se muestra el otro y realizar el cambio de buffer visible recién cuando la imagen está bien formada y el VIC comenzó a mostrar un nuevo cuadro. Esto es muy útil para los scrolls, pero hay que tener en cuenta que consume memoria RAM.

CIA 1 (teclado, joystick, timer con interrupción)

Timer

Esta CIA tiene acceso a la línea de la interrupción (IRQ) del microprocesador. Se puede programar el timer de esta CIA para que cada tanto tiempo genere una interrupción. Esto es utilizado por la ROM para leer el teclado pero se puede usar para muchas funciones.

Lectura del teclado

El puerto A y el B (ambos de 8 bits) están conectados al teclado. Este teclado es una matriz de 8 columnas por 8 filas, lo que da unas 64 teclas en total (RESTORE no cuenta porque es una tecla especial, como veremos más adelante). Para leer el estado del teclado se debe recorrer la matriz y para eso, se habilita (poniéndola en 0) a una sola fila de la matriz que lee el estado de sus 8 columnas. Un puerto de 8 bits se utiliza para habilitar una (o más de una) fila y con el otro puerto, se lee el estado de cada columna de esa fila. Cualquiera de los dos puertos de la CIA se puede usar para leer una línea o configurar su estado, solo alcanza con hacerlo con uno de los puertos como salida y el otro como entrada. Una forma

rápida para ver si hay alguna tecla presionada sin recorrer las 8 columnas, es encender las 8 columnas (dejar todos los bits en 0) y verificar si alguna de las entradas se encuentra en 1. Si el resultado no es 255 significa que hay al menos una tecla presionada pero no hay forma de saber cuál es y hay que hacer el barrido de las 8 filas para sacarlo. Al presionar una tecla, no se produce ningún tipo de evento ni en la CIA ni en ningún lado, es responsabilidad del programa recorrer periódicamente la matriz del teclado para ver si hay una tecla presionada. A su vez, esta rutina de lectura del teclado tiene que llevar el estado de cada tecla para generar un nuevo evento solo cuando recién se presione la tecla, pero a su vez si la tecla esta presionada el tiempo suficiente, debe generar la autorepetición si corresponde. Tiene que diferenciar el tipo de tecla, si es de control (SHIFT, CTRL. o C=) solo debe guardar el estado, si es un carácter debe guardarlo en el buffer, resolver el código de la tecla en base a las coordenadas, etc. Normalmente de todo esto se encarga la Kernal por medio de la interrupción del timer.

Lectura del joystick

Existen dos joysticks conectados a la CIA y cada uno de ellos tiene 5 entradas binarias: el botón y las cuatro posiciones (arriba, abajo, izquierda y derecha). De cada joystick se puede leer una combinación de las 5 entradas, por ejemplo arriba más derecha para una diagonal, derecha más botón, pero en condiciones normales no tiene sentido tener arriba y abajo presionadas a la vez, lo mismo con izquierda y derecha. Un joystick se lee de un puerto, así como el otro en un puerto diferente de entrada/salida de la CIA. Estos puertos son los mismos usados por la matriz del teclado.

Conflicto entre el joystick y el teclado

Como la matriz para leer el teclado y los dos joysticks se conectan eléctricamente al mismo puerto, se encuentran en conflicto. Cuando no hay ninguna tecla presionada y ningún switch de ninguno de los joysticks cerrados (o ni siquiera están conectados los joysticks) entonces no lee ninguna señal. El problema es cuando se cierra un circuito. Si se cierra un circuito del joystick pero leemos la matriz del teclado y dependiendo de la entrada del joystick, se puede leer como que se presionó una tecla cuando en realidad solo se ha movido el joystick. Esto pasa por ejemplo en el Basic, que siempre está leyendo el teclado, cuando movemos la palanca del joysticks aparecen las letras en pantalla. Lo mismo puede ocurrir en el caso inverso, de un programa que no lee el teclado pero si los joystick. Si presionamos determinadas teclas y según el estado de los joystick, se puede generar una lectura errónea de uno de los dos joysticks.

En definitiva, no se puede tener un programa que lea a la vez los joysticks y el teclado. Aunque no hay una solución general, se puede realizar para casos reducidos y estudiando bien las líneas en conflicto, aceptar algunas teclas que no van a producir lecturas erróneas de joystick y a su vez, no van a ser mal interpretadas por estar moviendo el joystick.

CIA 2 (Bus serial, Puerto de usuario, mapeo de VIC, timer con interrupción NMI).

Manejo de bus serial

Desde esta CIA se maneja el puerto serial que comunica con periféricos como disqueteras e impresoras. Se maneja de forma independiente cada una de las líneas, el protocolo de transmisión en si se ejecuta desde software.

Puerto de usuario

Este puerto no cumple ninguna función en el sistema estándar y su funcionamiento depende del periférico que se le haya conectado; desde el software se leen o escriben pines digitales. Este puerto es útil para realizar expansiones caseras, periféricos que no se cuelgan al bus, con toda la lógica y timing que eso necesita, sino que se comunican con la computadora con líneas dedicadas a velocidades bajas o altas.

Mapeo del VIC

A través de esta CIA es que se selecciona el bloque de 16 kB al que va a acceder el VIC. Como el VIC maneja 14 líneas de direcciones, desde el VIC se configuran las dos líneas mayores para seleccionar entre cuatro bloques de 16 kB.

Timer con interrupción no enmascarable

Este timer se puede usar sin interrupción, pero si se habilita, produce una interrupción no enmascarable (NMI) que se va a ejecutar como sea, cortando todo lo que se esté haciendo en el momento. Disparar esta interrupción es similar a presionar la tecla RESTORE. Tiene uso para cosas muy técnicas (emulación de un hardware, de forma muy lenta por supuesto) pero normalmente es poco seguro usar esta interrupción. A pesar de esto, algunos programas y juegos (*Batman: The Caped Cruzader*, por ejemplo) utilizan este timer, pero no es recomendable porque además de todo puede producir conflictos con cartuchos que dependan de la línea de NMI para realizar alguna tarea.

I/O externas

En el bus de expansión hay reservados dos espacios de 256 bytes para expansión. Estas dos entradas/salidas se conectan directamente al bus del sistema y tienen dos líneas de selección propias. Su funcionamiento va a depender del periférico que se le conecte. A diferencia de los periféricos conectados al puerto de usuario, para usar alguno de estos dos espacios de I/O hace falta estar conectado al bus. Lo que simplifica tener estas dos líneas de selección, es que no es necesario tener circuitos para detectar si se encuentra direccionando el rango de direcciones que le corresponden al periférico. De esta manera, con leer las líneas de selección alcanza aunque luego debamos enfrentar toda la complejidad de la conexión al bus, el timing, etc.

Reseteo y mapeo de memoria

Cuando el 6502 recibe una señal de reseteo al encender o presionar un botón de reseteo (no disponible en el equipo pero de fácil agregado) comienza la ejecución desde una dirección fija. El problema es que en la C64 la dirección donde se ejecuta el arranque está en la ROM y esta memoria no está siempre visible al microprocesador y según como este mapeada la memoria en esa dirección puede que este visible la RAM. Cuando recién se enciende la computadora, las líneas que definen si hay ROM o RAM en esa área de memoria comienzan en un valor prefijado y el equipo arranca siempre desde la ROM.

En cambio, cuando se envía la señal de reset, las líneas usadas para mapeo mantienen el valor que tenían antes del reset, a pesar de ser parte del integrado del microprocesador (ver mapeo de memoria). La consecuencia es que cuando se resetea y esta mapeada la RAM en la parte alta de la memoria, se va a saltar a la dirección de arranque que figure en la RAM y no en la ROM original. Por eso, el programa (si va deshabilitar a la ROM) debe dejar una rutina de reseteo válida para el caso que se presione el botón de reset. Algunos programas optan por usar esta línea para reiniciarse a sí mismo. Cuando se presiona reset salta a la pantalla inicial del programa o juego en lugar de volver a arrancar todo el equipo. Otros programas o juegos ponen una rutina que lo que hace es remapear la ROM y saltar a la rutina normal de arranque de la ROM. Lo más conveniente para el usuario es que el reset haga lo que es espera (volver a arrancar la maquina) sino se lo está obligando a que prenda y apague el equipo.

Tecla RESTORE. Interrupción no enmascarable

En la C64 esta interrupción está pegada a la tecla RESTORE. Cada vez que se presiona RESTORE pero solo si se toca y suelta rápido (hay un circuito con un Flip-Flop para forzar esto) se dispara la interrupción no enmascarable. No hay forma de evitar que se ejecute esta interrupción y se va a ejecutar sin importar si están deshabilitadas las interrupciones normales. Generalmente, los programas no hacen nada cuando se produce este evento. Conviene dejar al menos una rutina que en esta interrupción no haga nada y termine directamente para que en el caso que se presione esa tecla no termine el programa.

Interrupción enmascarable (IRQ)

Esta interrupción puede ser causada por el VIC (por las distintos eventos que genera), por el timer de la CIA 1 o por un periférico externo conectado al puerto de expansión. Al entrar en una interrupción lo primero que hay que hacer es ver que es lo que lo causó y si es una interrupción de VIC (determinado que evento) o es del timer de la CIA 1. En base a que produjo la interrupción se ejecuta el código correspondiente aunque normalmente conviene, por simplificación, configurar una sola cosa para producir interrupciones. El Basic utiliza esta interrupción desde el timer de la CIA 1 para leer el estado del teclado periódicamente.

DESARROLLO DE UN PROGRAMA

Recién cuando se tiene un buen conocimiento del sistema, conviene encarar el diseño del programa. Acá se describen algunos de los muchos aspectos a tener en cuenta

para el desarrollo de un programa para la C64. No es un análisis exhaustivo, porque el desarrollo de un sistema más o menos complejo es un tema muy amplio, pero se da un pantallazo de pasos importantes en la implementación para la Commodore 64.

Modelo de memoria a usar

Principalmente, debemos definir antes de empezar a escribir el programa como se va a mapear la memoria. Las cuestiones a determinar se pueden resumir en los siguientes puntos:

Utilización de rutinas de la ROM de Basic

La principal razón para prescindir de esta ROM es aprovechar la RAM que hay debajo y permitirnos tener un bloque continuo grande de RAM, porque por encima de esta ROM hay 4 kB de RAM. Eliminando esta memoria, tenemos un bloque continuo de 12 kB agregados al final de la RAM habitual.

Utilización de rutinas de la ROM de Kernal

Desde esta ROM se procesan las interrupciones pero se pueden hacer de todas formas que apunten a nuestras propias rutinas en RAM. Cada vez que la rutina de interrupción en ROM se produce, salta a la dirección almacenada en una variable de la RAM que normalmente apunta a una rutina de la ROM. Esto de por si es lento si queremos hacer un procesamiento muy ajustado. Si no usamos esta ROM e igual necesitamos usar el teclado, hará falta escribir una rutina para recorrer la matriz del teclado porque esa es una de las cosas de la que se encarga esta ROM. (ver Lectura de Teclado) También todas las rutinas de comunicación básica con periféricos y manejo de texto en pantalla se realizan en esta ROM. Por eso, hay que tener en cuenta esta situación al deshabilitarla porque habrá que reemplazar estas rutinas por otras propias.

Ubicación de la memoria del VIC

Como se detalló en [1] y en el apartado del mapeo de memoria del CPU vs. mapeo del VIC, se debe seleccionar uno de cuatro posibles bloques de 16 kB de memoria para que utilice el VIC. En este bloque es donde van a estar almacenados todos los gráficos, ya sean pantallas de mapa de bits, matriz de pantalla, fonts y sprites.

Pantalla de presentación (mapa de bits)

Los juegos suelen tener una pantalla de mapa de bits en la presentación. Esta pantalla no es interactiva, solo muestra el contenido del mapa de bits y ocupa 8 kB más 1 kB de colores y otro kilobyte más si se usa en baja definición, por lo que ocupa entre 9 y 10 kB. Si es un juego que requiere mucha memoria por lo general solo se muestra una sola vez, luego se puede escribir la memoria que ocupa y utilizarla para almacenar variables del juego. En este caso, si se quiere verla nuevamente, habrá que cargar el juego otra vez. De todas formas, aunque se la muestre una o varias veces, hay que tener en cuenta que no debe estar junto con los 16 kB de memoria de video que se van a usar durante el juego en si porque se

pierde mucho espacio. Es necesario manejarlo como dos cosas distintas, ya sea empleando dos configuraciones del VIC que apunten a bancos distintos o moviendo memoria.

Modo de video y tamaño de buffer

Hay que administrar los 16 kB de RAM del VIC de la forma más óptima. Al diseñar el juego es necesario definir cuantos mapas de caracteres se van a usar de forma normal (2 kB por cada uno). Además, cuantos buffers de pantalla se van a utilizar a la vez. Por ejemplo, si se va a usar scroll con doble buffer, se puede determinar cuanta memoria queda para los sprites. Recién con un estimativo de estas cosas conviene encarar el desarrollo del juego, tratando de aprovechar cada elemento lo más posible.

Administración de la página cero

Las 254 direcciones de la página cero (de 0002 a 00FF porque la 0000 y 0001 son usadas por el puerto interno del 6510) se tienen que administrar con cuidado porque hay una cantidad limitada pero son muy útiles, tanto para hacer más corto y rápido el código ejecutable, como para las funciones especiales (modos indirectos, puntero de 16 bits). Si el programa utiliza interrupciones, conviene reservar espacio para las variables que más se usen para que se ejecute lo más rápido posible.

Timing en los juegos

Los juegos necesitan una referencia de tiempo para controlar por ejemplo, las animaciones (un cuadro cada determinada cantidad de tiempo) y el movimiento de los objetos (que se muevan tantos píxeles en determinado lapso). Además, para mantener todo elemento sincronizado como por ejemplo, que todos los personajes se muevan a la misma velocidad o a las velocidades relativas que le corresponden a cada uno. Es común que los juegos se ejecuten en ciclos y que en cada uno se realicen todas las operaciones necesarias: lectura de estado de joystick, actualización de estado del jugador según la posición actual del joystick, detección de colisiones entre elementos del juego (personajes, paredes, ítems, etc.). También, la actualización de la estrategia de la maquina según lo que se encuentre haciendo el jugador, disparos de eventos si se llegó a determinada condición (por ejemplo si llega a una parte de la pantalla abre una puerta), si se avanza un cuadro en las animaciones o se mueven los personales la unidad de movimiento que le corresponda, etc.

Salvo en juegos por turnos (por ejemplo ajedrez o de cartas) es necesario que cada ciclo del juego dure lo mismo y por lo general que se ejecuten varios ciclos por segundo, así con los pequeños cambios entre ciclo y ciclo se logra la sensación de algo que ocurre de forma continua. Para lograr que cada ciclo dure lo mismo, hay que recurrir a una referencia de tiempo porque no es muy práctico escribir el código del ciclo para que siempre ejecute la misma cantidad de instrucciones y no varíe el tiempo de ejecución. Puede haber variaciones muy importantes en la duración de ejecución de cada ciclo si en uno de ellos ocurrió un evento importante que llevo mucho más procesamiento que lo habitual como por ejemplo, si se mató a un enemigo o se realizó un scroll de pantalla. Lo que se suele hacer, es tener un ciclo de la duración estimada para la mayoría y si la ejecución del código del ciclo se realizó más rápido, el resto del tiempo el programa queda en espera a que termine el tiempo

asignado. De esta forma cada ciclo va a durar lo mismo, por más que en no todos se realicen la misma cantidad de operaciones.

Timers de las CIAs

Una forma de tener una referencia de tiempo, es guiarse por uno de los timers de las CIAs. Si usamos la CIA 1 podemos tener una interrupción cuando el tiempo asignado al ciclo se cumpla. De todas formas, no es necesario usar la interrupción porque se puede programar a uno de los timers para que cuente un lapso determinado de tiempo. Cuando terminamos de ejecutar el código del timer, leemos el timer esperando que se termine el lapso (*polling*) antes de continuar con el ciclo siguiente. La ventaja de usar como referencia de tiempo el timer, es que es muy preciso y no hay diferencias de tiempo importante entre distintos modelos de máquina, como ocurre con los timings de la generación de video.

Sincronismo vertical del VIC. Timing desde el VIC

Por distintos motivos, es conveniente usar como referencia de tiempo el sincronismo vertical del VIC. Esto nos permite tener sincronizado el manejo de la pantalla con el procesamiento del juego. El inconveniente de esta técnica, es que los ciclos van a tener distinta duración según si el estándar de video es de 50 cuadros por segundo o es de 60. Si se desarrolla el juego teniendo en cuenta solo un estándar de 50 cuadros por segundo, en los que se puede ejecutar una cantidad apreciablemente mayor de código entre cada cuadro, es muy probable que no pueda funcionar de forma correcta en una máquina de 60 cuadros por segundo. Por ejemplo, puede ejecutarse con efectos extraños en pantalla (parpadeos, elementos que desaparecen) o colgarse directamente.

Sonido y Música

La reproducción de sonidos y música en el SID se realiza enviándole al SID nota por nota lo que tiene que reproducir en el momento que lo tiene que hacer. Para tocar la música en paralelo a la ejecución de un juego o programa, es necesario que la rutina que alimenta al SID con los sonidos se ejecute de forma periódica, varias veces por segundo y lo necesario para poder cubrir las notas más rápidas en las que es necesario disparar un sonido con un lapso rápido entre cada uno. También es necesario ejecutarse muchas veces por segundo si además de iniciar y terminar notas (o sonidos para efectos de sonido del juego) se van a realizar efectos sobre los sonidos que están ya en ejecución, por ejemplo, de variación de ancho de pulso en ondas cuadradas para hacer un cambio de timbre en el sonido, un efecto muy común en el SID. Por eso la rutina de reproducción de sonido se ejecuta de forma periódica de algunas de estas formas.

Por interrupción de timer

Se programa al timer de la CIA 1 para que genere una interrupción cada determinado lapso de tiempo y ejecute la rutina de reproducción de sonidos y música. El problema es que no está sincronizado con el resto del programa y esta interrupción puede entrar en momentos de procesamiento de gráficos y producir un efecto en la pantalla al desincronizar el VIC del programa. Si en cambio, se deshabilita la interrupción mientras se realiza una

operación así, puede ocurrir que la interrupción de sonido entre un poco tarde y se note un efecto raro al oído.

Sincronismo vertical del VIC

Si todo el juego e incluso la rutina de reproducción de sonidos y música se encuentran sincronizados con la generación de video, se evitan muchos efectos molestos de desincronización entre el video, el procesamiento de pantalla y la reproducción de música. Además, se simplifica mucho la codificación utilizando como patrón el timing de video porque todo se ajusta en función de este. El inconveniente de esta técnica, es que es común que un juego escrito para una norma de video de 60 cuadros por segundo, al correr en un equipo de 50 cuadros por segundo reproduzca la música a una velocidad más lenta y el efecto inverso cuando se pasa de 50 a 60 cuadros por segundo.

Llamador en Basic

Desde el Basic de la C64 solo se pueden ejecutar programas escritos en Basic con el comando RUN. Los programas que se autoejecutan desde cassette son similares por lo que es como si se ejecutara RUN al terminar de cargar (existen otras formas de autoejecutar programas al terminar de cargar de cassette pero es más complicado y especifico). Para correr rutinas en assembler es necesario escribir SYS seguido de la dirección de memoria o incluir el SYS dentro de un programa Basic. Para poder ejecutar un programa en assembler se puede incluir un programa Basic mínimo que solo ejecuta un SYS con la dirección de inicio del programa en assembler. Por ejemplo para hacerlo todo desde assembler, se puede hacer, con una secuencia de datos iniciada en la dirección 801h:

```
!word BasicEnd     ;Direccion del final del programa Basic
!word 2011         ;Numero de linea, cualquier cosa mayor a 0
!byte $9e          ;Indica instruccion SYS
!text "2100 MENSAJE A MOSTRAR JUNTO AL SYS",0  ;Direccion de carga
BasicEnd: !word 0  ;terminador de programa Basic
```

Una vez que se carga el programa, se puede listar y ver el programa que consta del SYS. El SYS apunta a la dirección (en decimal) donde arranca el programa en assembler. Luego del SYS es común incorporar un texto cualquiera, que el intérprete Basic ignora.

Compresión del programa

Para reducir el tiempo de carga del programa se lo suele guardar comprimido. La descompresión no es un proceso inmediato, puede estar varios segundos descomprimiendo, pero incluso esto es más rápido que cargar el programa desde disquete o cassette aunque se utilice un cargador turbo. Es habitual que al terminar de cargar un programa, durante unos segundos se muestre información rara en pantalla (caracteres al azar), se reproduzca un ruido extraño, se muestren barras de colores o todas estas cosas al mismo tiempo. Esto ocurre porque mientras se descomprime y para mostrar que se encuentra haciendo algo, las rutinas de descompresión suelen hacer alguna de esas cosas que son relativamente fáciles de hacer y no consumen muchos recursos. Lo ideal sería mostrar una barra o un contador

como en programas más modernos, pero la rutina de descompresión no suele ser tan compleja, justamente para ahorrar espacio. Los caracteres raros que se mueven a toda velocidad en pantalla también se encuentran relacionados con el hecho de que la rutina de descompresión tiene disponible el área de RAM por debajo de 802 h. En estas direcciones, se pueden utilizar para guardar variables y es ahí donde está por default la matriz de pantalla, así al actualizar las variables y buffers temporales se muestra información al azar en pantalla.

Un utilitario moderno muy útil y eficiente para comprimir programas desde la PC es el Exomizer [3]. A un ejecutable, este programa lo comprime y le agrega una rutina de descompresión adelante. Opcionalmente permite configurar que rutina ejecutar mientras descomprime para mostrar actividad mientras realiza la tarea. La compresión desde un hardware con la velocidad y memoria de una PC moderna es una operación trivial, pero realizar la misma operación en una C64 puede llevar varios minutos porque es una operación de uso intensivo de microprocesador y memoria.

Almacenamiento en disco o cassette

Los protocolos de transmisión de datos que tiene la computadora para comunicarse con el datasette y la disquetera son muy lentos. En el caso del datasette son unos cuantos minutos para cargar un programa que use unos 40 kB y la disquetera también es muy lenta, aunque no tanto. A pesar de realizar normalmente la lectura a esta velocidad tan baja, los periféricos y la computadora soportan velocidades muy superiores utilizando otras técnicas de comunicación. Los programas suelen incorporar rutinas de carga turbo para poder cargarse a velocidades más aceptables y que básicamente pasan por encima al sistema operativo.

Turbo de Datassete

En el caso del datasette, el protocolo estándar (y la forma en que se graba la cinta) codifica la información de forma muy lenta por la forma que se representa cada bit y porque se transmite cada bloque dos veces. Esta operación se realiza para detectar errores de transmisión aunque si detecta un error no lo corrige, solamente detiene la carga. Los programas con carga turbo emplean al principio del programa una rutina que se carga de la forma habitual pero rápidamente porque es de tamaño muy reducido. Luego toma el control y carga el resto del programa utilizando otro formato propio porque el resto del programa se grabó especialmente para ser cargado solo por esta rutina. Existen decenas o cientos de rutinas distintas para grabar y cargar de cassette en un formato turbo, pero todas se basan en no darle tanta importancia en grabar información con demasiada redundancia, por lo que lo hacen de forma mucho más compacta. Una complicación que surge de estas rutinas, es que se pueden leer solo con la rutina que le corresponde y que espera encontrar datos en una secuencia exacta. En consecuencia, para copiar el cassette a un emulador, se debe guardar pulso por pulso del *track* del cassette en un archivo TAP, el cual suele ser mucho más grande que el programa en si por la forma ineficiente en que se graba cada bit.

Turbo de disquetera

En la disquetera, la forma en que se transmite cada bit es muy lenta para asegurar la correcta sincronización de tiempo entre la disquetera y la computadora bit por bit. Además, no aprovecha todas las líneas de comunicación que tiene el cable de comunicación, sino que solo transmite datos por una línea y las otras las usa para sincronización. A diferencia del caso del datasette, la rutina turbo no modifica como se guarda el programa en el disquete porque este se guarda de la forma habitual y solo la transmisión se realiza de la forma más óptima desde la disquetera a la computadora. La rutina de carga rápida aprovecha que la disquetera (Commodore 1541) es en sí una computadora y que permite cargarle programas y ejecutarlos en su RAM interna. De esta forma, le carga una rutina a la disquetera para que reemplace a la que usa para su comunicación con la computadora y a su vez, desde la C64 utiliza un código propio para comunicarse con la disquetera. Esta forma de comunicar es más rápida porque realiza una sincronización más simple. Por ejemplo, en vez de sincronizar por cada bit se hace una vez por byte, confiando en que se pueden transmitir un grupo chico de bits sin perder sincronía. Además, también se aprovechan las líneas extras para enviar más de un bit a la vez. Así, se pueden lograr velocidades bastante más elevadas.

Finalmente, es un resumen aproximado de cómo trabajan las rutinas de carga turbo e incluso los cartuchos Fast Load, pero el tema este es muy avanzado en sí mismo y necesita un desarrollo muy extenso. Toda esta interacción entre la disquetera y la computadora hace que sea tan difícil emular en un 100% la disquetera de una C64, como intenta hacer por ejemplo la placa SD2IEC [4] con un microcontrolador de 8 bits. Para emularla completamente es necesario un sistema mucho más complejo y caro, como la 1541 Ultimate [5].

HERRAMIENTAS DE DESARROLLO MODERNAS

Finalizando, con la potencia de las computadoras modernas y el mayor acceso a información técnica, surgieron muchas herramientas de desarrollo cruzado. Significa, que se pueden generar programas y datos para una plataforma (en este caso la Commodore 64) pero se ejecutan desde otro sistema más completo (una PC con Windows o Linux por ejemplo) miles de veces más rápido, con millones de veces la capacidad de almacenamiento y mayor facilidad de uso.

Ensambladores

Dos ensambladores para el 6502 son el DASM [6] y el ACME [7] que se pueden ejecutan en distintos sistemas operativos de PC. Es mucho más fácil trabajar con estos ensambladores que con un ensamblador en la C64 por el acceso a mejores editores de texto modernos, la mayor libertad para organizar los archivos con códigos fuentes en un disco rígido, con herramientas de backup y control de versiones. Además se puede hacer más entendible el código creando códigos fuentes más documentados, con macros y mayor uso de definiciones que la que se puede lograr en la maquina original.

Compiladores

El cc65 [8] es un compilador cruzado para 6502 que se puede usar para generar código para la C64. Con lenguaje C y usando algunas rutinas en assembler en el medio en lugares que necesiten velocidad, se pueden realizar cosas muy avanzadas para este hardware.

Herramientas para gráficos y sonidos

Un programa muy fácil de usar y poderoso para generar gráficos en los distintos modos de video que soporta la C64 es el Timanthes [9]. Con este programa desde la PC, uno va realizando un gráfico con las limitaciones propias del VIC II, por ejemplo la paleta de 16 colores y la limitación de colores en cada celda. También existen programas para crear sprites en PC, por ejemplo el Sprite Pad [10] y para crear mapas de caracteres y tiles está el CharPad [11].

Para música existe el GoatTracker [12], un tracker que permite componer música para el SID, y generar un archivo SID que podemos insertar en un programa desarrollado por nosotros incluyendo unas rutinas de reproducción de música.

Emuladores y debuguers

El WinVice [13] es el emulador de código fuente más completo para la Commodore y que soporta distintos modelos. Incluye un debugger que permite ejecutar el código paso por paso, ver el contenido de la memoria, modificarla y guardar la información de memoria a disco. Este debugger es una herramienta, si bien no muy amigable para usar, muy poderosa, que nos da acceso a cosas que en el hardware real no podríamos ver y solo deducir luego de pruebas exhaustivas. Por ejemplo, al ejecutar paso a paso, no es solo el microprocesador el que funciona más despacio sino todo el sistema incluyendo el VIC II pudiéndose pulir detalles finos de programas que dependen del timing de distintos componentes del sistema.

Herramientas personalizadas

Al ser formatos relativamente simples los que utiliza el sistema como por ejemplo para gráficos (sprites, mapas de bits, fonts o textos) no es complicado realizar herramientas en lenguajes de alto nivel para generar estos datos, por ejemplo basado en archivos de gráficos estándares (BMP, GIF, etc.) realizados con programas profesionales.

AGRADECIMIENTOS

A Pablo Roldán por sus correcciones y sugerencias.

BIBLIOGRAFÍA

[1] M. Leguizamón (2011), Programando la Commodore 64 en serio. Circuito de video y componentes básicos del sistema, *Revista de Tecnología e Informática Histórica* 1, pp. 59-74.
[2] J. West y M. Mäkelä (1994), 64 doc, v. 1.8. http://www.viceteam.org/plain/64doc.txt (27 de septiembre de 2011).
[3] M. Lind (2011), *Exomizer 2.0.2*. http://hem.bredband.net/magli143/exo/ (14 de septiembre de 2011).

[4] I. Korb (2011), sd2iec, an AVR-based C64 floppy drive emulator. http://www.sd2iec.de/ (21 de noviembre de 2011).
[5] G. Zweijtzer (2011), 1541 Ultimate. http://www.1541ultimate.net/ (21 de noviembre de 2011).
[6] M. Dillon (1988), *DASM 2.2x*. http://dasm-dillon.sourceforge.net/ (14 de septiembre de 2011).
[7] Smørbrød Software (2006), ACME 0.90. http://www.esw-heim.tu-clausthal.de/~marco/smorbrod/acme/ (14 de septiembre de 2011).
[8] J. Dunning (1989), cc65 5.6.7. http://www.cc65.org/ (14 de septiembre de 2011).
[9] Focus (2009), Timanthes 3.0 beta. http://noname.c64.org/csdb/release/?id=75871 (14 de septiembre de 2009).
[10] Subchrist Software (2011). Sprite Pad 1.8.1. http://www.coder.myby.co.uk/spritepad.htm (14 de septiembre de 2011).
[11] Subchrist Software (2011). CharPad 1.8 (revision 3). http://www.coder.pwp.blueyonder.co.uk/charpad.htm (14 de septiembre de 2011).
[12] L. Öörni (2011), GoatTracker 2.72. http://sourceforge.net/projects/goattracker2/ (14 de septiembre de 2011).
[13] A Boose, T. Biczo, D. Lem, A. Dehmel, A. Matthies, M. Pottendorfer, S. Trikaliotis, M. van den Heuvel, C. Vogelgsang, F. Gennari, M. Kiesel, H. Nuotio, D. Kahlin, A. Lankila, M. Brenner, D. Hansel, T. Bretz, D. Sladic, E. Perazzoli, A. Fachat, T. Rantanen, J. Valta, J. Sonninen (2009), VICE 2.2. http://www.viceteam.org/ (14 de septiembre de 2011).