

Cómo entrar en un programa y averiguar sus secretos

LA BIBLIA DEL «HACKER» (I)

José Manuel LAZO

Un «HACKER», según el diccionario de la lengua inglesa, es una persona capaz de enfrentarse (con éxito) a todas las dificultades que le impone un determinado sistema. ¿Cuántas veces has necesitado examinar el interior de un programa, y no has podido porque te has estrellado contra infranqueables protecciones? En esta serie vamos a abordar en profundidad este delicado tema.

Debido al masivo avance de la piratería del software, las casas productoras han añadido a sus creaciones una serie de protecciones para evitar que terceros se adueñen, copien o llenen sus bolsillos con ese producto que, la mayoría de las veces, ha requerido el esfuerzo de muchas personas durante bastante tiempo.

Esto, por una parte, está bien, ya que frena en lo posible la piratería, pero bloquea al usuario que legalmente ha adquirido un juego o una utilidad y, por cualquier circunstancia, desea modificar este programa en alguna de sus partes.

Porque, ¿cuántas veces te hubiera gustado ponerle vidas infinitas a ese juego que tienes arrinconado porque no logras pasar de la tercera pantalla o modificar las opciones de impresora en esta utilidad que tanto necesitas? Y no has podido, porque el programita en cuestión parece un cofre de titanio cerrado a cal y canto con mil cerrojos.

Y, ¿qué pasa con los poseedores de sistemas de almacenamiento más eficaces y fiables que la cinta de cassette?

Los compradores de unidades de disco, microdrives, etc., maldicen una y otra vez el día en que se les ocurrió adquirir uno de estos artilugios, ya que no existen programas en estos formatos. La única posibilidad que les queda es adaptar el software de la cinta original.

Por último, hay numerosos usuarios de software que encuentran mayor placer en «profanar» un programa y ver sus intimidades que en matar a tal o cual marciano.

No a la piratería

Con esta serie van a acabarse estos problemas, pero los piratas a los que ya se les están poniendo los dientes largos que no sigan leyendo, pues aquí NUNCA se va a explicar la manera en que se puede copiar un programa, cosa que, por otra parte, es legal si la copia la utilizamos SOLO como back-up de seguridad.

No creas que estás infringiendo alguna ley desprotegiendo un programa; es una labor perfectamente LEGAL siempre y cuando no negociemos con ello enriqueciéndonos a costa del esfuerzo de los demás. Lo hemos dicho muchas veces, y no está de más recordarlo aquí: estamos en contra de la PIRATERIA porque a la larga puede hundir la industria del software y eso no es bueno para nadie.

La protección del software

Ninguna cosa en el mundo de los ordenadores es más polifacética que la protección del software. Existen mil y un trucos con los cuales se puede proteger un programa y hacerlo inviolable a unos ojos no expertos en el tema; existen protecciones en el Basic, en el CM, aprovechando errores del microproce-

sador, etc. Cada programa se puede decir que es un mundo aparte, distinto de los demás. El sistema de protección que ha utilizado una casa, además de proteger el programa, tiene que protegerse a sí mismo para evitar que otra casa lo utilice.

Por otra parte, no existe un sistema de análisis que pueda aplicarse a todos los programas como si se tratase de la piedra filosofal. No existe lo que podríamos llamar «los diez mandamientos del Hacker», al contrario, en esta me-





táfora existiría toda una **Biblia** completa que podría llenarse de información referente al tema. De ahí el nombre de la serie.

Sólo la experiencia, un profundo conocimiento del lenguaje **Assembler** y, sobre todo, del sistema operativo del **Spectrum**, pueden ser las cualidades del verdadero «**Hacker**».

En esta serie utilizaremos en todo momento términos y sistemas **SENCILLOS**, dentro de lo que cabe. Si se tuviese cualquier duda puede ser una ines-

timable ayuda y complemento el curso de **C.M.** que está en las páginas centrales de esta revista desde el número 42.

Estructura de los sistemas de protección

Vamos a empezar por una clasificación genérica de las distintas protecciones con las que un usuario puede encontrarse:

En primer lugar existen:

- * Protecciones a nivel **Basic**.
- * Protecciones a nivel **Código Máquina**.
- * Protecciones a nivel **Hardware**.
- * Rutinas de carga distintas a las normales.

***** LAS PROTECCIONES A NIVEL BASIC.**

El **Basic** es un lenguaje bastante más sencillo que el árido **Assembler**, sin embargo, las protecciones a nivel **Basic** pueden producir más dolores de cabeza de lo que en un principio puede suponerse. Para enfrentarse con este tipo de protecciones es necesario tener conocimientos de cómo funciona el **SO** (Sistema Operativo) ante una situación determinada.

El 99 por 100 de los programas llevan protecciones de este tipo; piénsese que es lo primero que se encuentra el **Hacker** al intentar entrar en un programa y es el primer ladrillo que debemos apartar. El nivel de protección es, bajo cierto punto de vista, más alto que lo que se puede encontrar en **C.M.** ya que aquí se pueden hacer más trampas en el ya intrincado juego.

Dentro de las protecciones, a nivel **Basic**, podemos encontrar:

- * Líneas **Ø**.
- * Controles de color.
- * Basura en los listados.
- * **C.M.** en líneas **REM**.
- * Literales **ASCII** retocadas.
- * **Pokes** en las variables del sistema.
- * **Anti-merge** en los programas.
- * **C.M.** en zona de edición.
- * **C.M.** en zona de variables.
- * Protección turbo.

***** PROTECCIONES A NIVEL C.M.**

En lenguaje **Assembler** también se pueden hacer protecciones bastante potentes, sin embargo, a idénticos conocimientos de ambos lenguajes resulta más sencillo entrar al **C.M.**; piénsese

que al ser un lenguaje más rígido se pueden realizar menos trampas. Te puedes encontrar con:

- * Corrompimiento de la pila.
- * «**Popeo**» de la dirección de retorno.
- * Uso de nemónicos inexistentes.
- * Enmascaramiento de código con registro **R**.
- * **Cheksum's** variados.
- * Enmascaramiento con pantalla.
- * Longitud excesiva de bytes.
- * Solapamiento del cargador.
- * Opacidad en la asignación de los vectores de carga.
- * Basura en listados.
- * Saltos a **Ø** por error de carga.
- * Deshabilitación del «**Space**».
- * Protección turbo.

***** RUTINAS DE CARGAS DISTINTAS.**

La mayoría de los programas llevan ahora un sistema de carga distinto al estándar de la **ROM**. Esto se hizo en un principio para que los «copiones» no pudieran copiar el programa en cuestión. Se pueden encontrar rutinas de carga de todo tipo, algunas tienen sólo el objeto de hacer más vistosa la carga, pero complican las cosas a la hora de estudiarlas.

- * Protección turbo.
- * Distinta velocidad en baudios.
- * Tono guía de distinta frecuencia.
- * Tono guía ultracorto.
- * Programas sin cabecera.
- * Tono guía en medio de los bytes.
- * Bloques «pegados».
- * Rutinas de carga «aleatoria» en vez de secuencial.

***** PROTECCIONES DE HARDWARE.**

Por último, nos podemos encontrar con distintas protecciones **hardware**. Algunos programas necesitan que una tarjeta esté conectada en el bus de expansión para funcionar. Estos no nos darán excesivos problemas ya que la única finalidad de este dispositivo es cerciorarse de que se posee el programa original.

En otras ocasiones, parte del software se halla soportado por una memoria **EPROM**; en este caso un nombre más acertado es el de **FIRMWARE** por ser un software **FIRMEMENTE** unido a la memoria. Este es de difícil modificación y se precisan, además, conocimientos de **hardware**. Pero todo se andará.

Cómo entrar en un programa y averiguar sus secretos

LA BIBLIA DEL «HACKER» (II)

José Manuel LAZO

Prosiguiendo con la serie, esta semana vamos a empezar por lo que primero se puede encontrar en un programa: PROTECCIONES A NIVEL BASIC.

El cargador de cualquier programa suele estar protegido en un 99 por 100 de los casos para evitar que se pueda efectuar un «Break» una vez que éste se haya ejecutado. Lo primero que hay que hacer es quitar el auto-run. Es un secreto a voces que haciendo MERGE "" el programa se carga pero no se ejecuta. Debemos probar esta forma en primer lugar, si bien, también existen protecciones para esto corrompiendo alguna línea del Basic, con lo que se consigue que una vez cargado el programa el SO (Sistema operativo) se cuelgue intentando «Mergear» una línea fuera del Basic.

Si no se consiguen resultados positivos intentaremos hacer una copia del cargador sin auto-run, usando para ello el programa «Copyup», publicado en nuestros números 44 y 45. Con su ayuda se puede cargar un programa Basic y modificar cualquiera de los parámetros de la subrutina. En este caso cambiaremos la línea de auto-run del programa por el valor 32768, con lo cual el Basic, a la hora de cargarse, no se ejecutará.

Una vez que tengamos el cargador sin auto-run lo podremos cargar tranquilamente y después nos saldrá el informe «OK».

Líneas 0

Al eliminar el listado nos podemos encontrar con que algunas, o todas las líneas, tienen como número el 0. Esto evita el que se puedan editar y modificar. Hay una manera de cambiar el número de líneas, aunque esto no es necesario como más adelante se verá. De momento y para poner un número 0 a la primera línea del Basic, puedes probar lo siguiente:

```
POKE (PEEK 23635
+ 256*PEEK 23636) + 1
```

Con esto podremos editarla. La ra-

zón de editar una línea del listado es poder quitarle los controles de color que puedan existir dificultando su visión.

Los controles de color en un listado son los códigos ASCII entre el 16 y el 21, ambos inclusive. Sirven para cambiar la tinta, o el papel en medio de una línea y que ésta no pueda verse. Es interesante primero ver la forma de introducirlos para luego poder saber cómo quitarlos fácilmente.

Prueba a editar una línea de un programa Basic cualquiera y desplaza el cursor al medio de la misma. Pulsa el modo extendido y a continuación el 4, por ejemplo. Verás que todo el papel de la línea a continuación se pone de color verde. Pulsa otra vez el modo extendido, pero esta vez el 1 con «Symbol shift». Ahora es la tinta la que ha cambiado. Pulsa «Enter» y podrás comprobar que el listado Basic a partir de esta línea tiene otro color distinto y que las órdenes INK y PAPER no funcionan en esa zona.

Esto mismo, pero con PAPER e INK del mismo color, puede valer para que una línea de listado sea visible.

Vamos a ver ahora cómo quitarlos: primero edita la línea que has modificado y desplaza el cursor hasta que éste no se vea, o sus atributos cambien a los nuevos. Pulsa «delete» dos veces por cada control ya que el mismo tiene dos códigos dentro del listado, uno es el control propiamente dicho, y otro es el valor al que se cambia este.

Suponete que tienes un listado en la pantalla y que sólo ves un 0; haz el POKE para cambiar el valor por 1 y edita la línea. Ahora desplaza el cursor hasta la derecha pero con cuidado, parando en el momento en que éste no se vea. «Delete» hasta que éste sea perfectamente visible; seguro que la línea también lo es en este momento. Si no lo fue tendrías que volver a repetir la operación hasta conseguirlo, pues puede haber más de uno.

Hay otras dos formas de poder contemplar un listado, aunque posea controles de color, sin necesidad de tener que editar las líneas.

Una es haciendo un LIST con una impresora; como ésta no reconoce los controles de color el listado saldrá visible.

La otra forma se verá más adelante.

Un consejo: editar una línea de un programa puede llevar perfectamente a hacer que el listado se corrompa en caso de que se haya utilizado la protección de las literales ASCII retocadas.

Controles de cursor

De la misma forma, también pueden ponerse controles de cursor haciendo que el listado comience en la parte superior de la pantalla y que al continuar lo haga otra vez sobre la primera línea. También puede salir el informe «entero fuera de rango». Vamos a ver esto más detenidamente:

Entre los códigos ASCII el 22 es el control de AT; cuando el SO se encuentra este control interpreta los dos siguientes como las coordenadas del cursor, en baja resolución, donde se va a continuar el listado. En una línea del Basic se han de dejar tres espacios en blanco, en el primero irá el control de AT, y en los siguientes las coordenadas.

Pokeando en los dos espacios disponibles para las coordenadas se puede lograr que al hacer LIST salga ante nosotros el informe «entero fuera de rango» dando como coordenadas unas imposibles, por ejemplo, AT 48,0.

De igual manera, con los controles cursor izquierda, y derecha, 8 y 9, se puede lograr enmascarar parte del listado sobrescribiendo encima de él mismo.

Estos controles de cursor no se pueden poner ni quitar en modo edición por lo que hay que hacerlo a base de POKES.

Cómo entrar en un programa y averiguar sus secretos

LA BIBLIA DEL «HACKER» (III)

José Manuel LAZO

En el listado de un programa Basic pueden hacerse determinadas alteraciones de tal forma que sea imposible averiguar su contenido e incluso que, al intentarlo, el propio listado se modifique creando una gran confusión.

Un caso típico es que cuando hacemos LIST, no sale nada, y además se nos presenta el cursor con una interrogación. Esto es porque se ha utilizado un control de AT con coordenadas falsas. Todo esto es en realidad basura dentro del listado para evitar que se vea. Pero la mejor manera de aprender todo esto es practicando, por lo que vamos a exponer unos ejemplos sencillos:

Primero tecléa lo siguiente teniendo la memoria del ordenador limpia:

```

1 REM (PON AQUÍ TRES ESPACIOS
EN BLANCO)
2 REM (OTROS CUANTOS ESPACIOS
)
1000 INPUT "Poke? " n
1010 LET direccion=(PEEK 23635)+2
56*(PEEK 23635)+5
1020 FOR a=direccion TO direccion
n+1
1030 INPUT "Valor? ", b: POKE a, b
1040 NEXT a
1050 STOP

```

Tecléa GOTO 1000 y prueba algunos controles: Primero uno, por lo que responde a la primera pregunta con un 1, y a la segunda con un 6. Si haces LIST verás que el texto de la primera línea se ha desplazado a la columna central de TAB, tal y como si hubiéramos utilizado PRINT con coma. Este control es el 6.

Responde ahora a la primera pregunta con 1, y a la segunda con 22; este es el control de AT. Puesto que los dos siguientes valores son dos 32, que corresponden al espacio, cuando pulsemos «ENTER» para hacer un listado automático no nos saldrá y tendremos el cursor junto con una interrogación. Sin embargo, al dar la orden LIST saldrá inmediatamente el error «entero fuera de rango».

Si tecléamos GOTO 1000 e introducimos el control de AT con unos valores adecuados cambiaremos las coordenadas del listado. Responde a la primera pregunta con 3 y a las tres siguientes con 22, 10 y 10. El listado aparecerá dividido en dos trozos.

Por último, vamos a ver la forma de sobrescribir en el listado; responde a la primera pregunta con 3, y a las 3 siguientes con 22, 0 y 0. Veremos como el número de la primera línea ha desaparecido imprimiéndose el texto de la

der ver un listado sin tener que modificar ninguna línea.

En primer lugar, es conveniente saber algunas cosas acerca de cómo se organiza un programa Basic en la memoria. Las líneas de programa se guardan en la memoria de la siguiente forma: primero dos octetos que indican el número de línea de que se trata. Si nosotros pokeamos en esa dirección con otro valor, cambiaremos el número de línea. Podemos ponerlo a «0» e incluso a un número imposible, mayor de 9999, dado que en dos octetos cabe cualquier número menor de 65535. Obviamente el efecto contrario también es posible, es decir, podemos cambiar el número para que sea legal.

Estos dos octetos se ponen al revés de como sería de esperar, el primero es el más significativo, y el segundo el de menor peso, esto es así para que el intérprete funcione más rápido.

Después de estos dos bytes vienen otros dos que indican la longitud de la línea incluyendo el código de «Enter» del final. Seguro que ya se te está ocurriendo que podemos variar también esta información para complicar más las cosas. Ello es posible haciendo que estos octetos contengan unos datos falsos, marcando más o menos longitud de lo normal. Lo hemos visto en muy pocos programas dado que también confunde al SO, y una cosa hay que tener muy clara, todas las protecciones a nivel Basic que podemos encontrar tienen la particularidad de que confunden el listado, pero nunca al SO.

En el texto de la línea se guardan todos los tokens y literales por sus respectivos códigos ASCII, pero hay una particularidad: los números. Después del texto de la línea viene un control de «Enter» (13), que marca la frontera entre líneas.

primera línea REM en las coordenadas 0,0.

Otra consecuencia de tener basura en el listado es que si conseguimos editar la línea, no la podemos modificar debido a que constantemente está sonando el zumbador de alarma por el error que, intencionadamente, se ha introducido en ella.

Esto último también puede ser debido a que en el listado exista un CLEAR que sitúe el RAMTOP excesivamente bajo para permitir la edición.

La basura de un listado se introduce con la finalidad de corromper el programa si tratamos de editar líneas o modificarlo en alguna de sus partes.

Hay que buscar alguna forma de po-

Protecciones aleatorias con el registro «R»

LA BIBLIA DEL «HACKER» (IX)

José Manuel LAZO

Siguiendo con el estudio detallado de los diferentes tipos de protecciones que se pueden llevar a cabo a nivel de código objeto, vamos a mostraros esta semana aquéllas que están directamente relacionadas con el registro de refresco o registro R.

El registro «R» es uno de los muchos que tiene el microprocesador de uso específico para él, en este caso para la memoria ya que se encarga de ir contabilizando la página de memoria que le toca ser refrescada por él mismo (para más información consultar los artículos de Primitivo en la sección Hardware).

Lo que a nosotros nos interesa es que su valor va variando secuencialmente con el tiempo, y muy rápido (relativamente), se puede decir que si consultamos su valor en un momento dado devuelve un número aleatorio, pero que para ciertas rutinas muy bien sincronizadas puede resultar predecible (vaya lío ¡eh!).

De esto se deduce que es muy sencillo que lo que se cargue de la cinta sea un montón informe de bytes y que, después de haberlos pasado por la piedra, oséase una rutina desenmascaradora, se conviertan en el verdadero código objeto limpio.

Una rutina desenmascaradora tiene un aspecto muy parecido a la del Cheksun salvo que todas las direcciones se Xorean con el

registro «R» para producir el verdadero código objeto.

El colmo del refinamiento viene cuando la rutina desenmascara otra que viene a continuación y pasa el control a la misma, la cual ya verdaderamente desenmascara el código limpio y entre ambas no hay ninguna inicialización del registro «R».

Por supuesto, ambas técnicas de protección se pueden mezclar y hasta incluso no hay nada que impida que esta última, en vez de producir el código a partir del registro «R» lo produzca a partir de la pantalla de presentación que acompaña al juego.

Esto podría ser así:

```

10 LD HL,40000
20 LD DE,16384
30 LD BC,6912
40 LOOP LD A,(DE)
50 XOR (HL)
60 LD (HL),A
70 INC HL
80 INC DE
90 DEC BC
100 LD A,B
110 OR C
120 JR NZ,LOOP
130 RET

```

Un ejemplo

Seguro que ya estabais pensando que nos habíamos olvidado de explicar la manera en que están protegidos ciertos programas, pues no, y como el movimiento se demuestra andando aquí y ahora os vamos a exponer, como primera mundial, la manera en que se protegió el EVERYONE'S A WALLY programa éste de MIKRO GEN.

De principio el programa se

halla protegido con una rutina de carga de velocidad distinta a la normal, cuestión ésta que estudiaremos más adelante. La rutina cargadora se ubica en la última página de memoria y el bloque que se carga se solapa por encima del cargador con lo cual todo lo que se encuentre por encima de la cargadora no tiene sentido ya que es lo que se carga de la cinta.

Después de la carga se procede a un Cheksun de la memoria, incluyendo el cargador, para comprobar que no se ha tocado nada.

Luego se salta directamente a una rutinilla ubicada en la memoria intermedia de impresora que se encarga de producir otra con unos valores situados después y otros ubicados en la pantalla de presentación mediante un sencillo, pero efectivo algoritmo.

Una vez que se ha producido esta rutina se pasa el control a la misma, la cual se encarga de desenmascarar todo el código que ha entrado de la cinta mediante el registro «R».

Llegados a este punto, ya se hace el salto al programa principal.

Vemos de esta forma como los programadores de MIKRO GEN han imprimido en sus creaciones una serie de protecciones bastante completas y difíciles de desproteger. Además, hay que reconocer que la rutina de carga rápida que se utiliza para cargar el código está perfectamente hecha siendo, hasta incluso, más fiable en la carga que la de la ROM estándar. Esto es todo por esta semana...

```

10 LD HL,25000
20 LD BC,40000
30 LD A,B
40 LD R,A
50 LOOP LD A,R
60 XOR (HL)
70 LD (HL),A
80 INC HL
90 DEC BC
100 LD A,B
110 OR C
120 JR NZ,LOOP
130 RET

```

Rutina desenmascaradora.

Cómo entrar en un programa y averiguar sus secretos

LA BIBLIA DEL «HACKER» (V)

José Manuel LAZO

La semana pasada analizábamos la utilidad de un programa, viejo conocido nuestro, **COPYLINE**, en las tareas de análisis de los cargadores Basic. Ahora continuaremos con esta labor incluyendo una interesante tabla que recoge todos los controles de color y cursor que maneja el Spectrum.

El programa en cuestión nos lista un Basic que esté ubicado en otra dirección aunque tenga cualquier protección de controles de color o cursor. El listado lo produce en 5 columnas cuyo significado se explicó la pasada semana.

En la antepenúltima columna van los controles de color, cursor, etc. Pero éstos no actúan sobre el listado. Consultando la tabla adjunta puedes averiguar la función de cada uno.

Estos últimos controles que son a modo de prefijos para los parámetros que le acompañan, con unos argumentos erróneos, hacen que el SO se confunda bastante a la hora de sacar el listado.

Con el Copyline tenemos, además, la ventaja de que al no modificar ninguna de las partes del programa y no estar éste en la zona del Basic no se corrompen la zona de las variables ni la zona de edición, lugar en el que se pueden volcar programas en CM tal y como veremos próximamente.

Vamos a tratar ahora de la protección que raya la frontera entre el Basic y el CM. Es el caso de los cargadores que tengan CM en las líneas del Ba-

sic o en las variables del mismo Basic.

Anteriormente apuntábamos la conveniencia de inspeccionar el listado Basic del cargador ubicando el mismo en otra dirección a fin de no modificar en nada su contenido, esta necesidad es imperiosa en el caso de que el programa Basic tuviera CM enmascarado en el mismo.

Supongamos que existe una línea Basic en el medio del listado en el que, después de un REM, se halla un programa en CM; supongamos igualmente que todo el resto del listado se haya protegido con controles de color. Si quitamos éstos, el programa en CM se reubicará con lo que cuando el cargador lo llame el mismo no funcionará. De ahí la necesidad de ver el listado con el Copyline publicado en anteriores semanas.

El CM en líneas REM se reconoce por la visión de ésta y a continuación una serie de tokens y literales incoherentes. Cuando veas esto... **NO LO TOQUES!!!**, es mejor inspeccionarlo tranquilamente con un desensamblador. Modifica su dirección con el Copyline y examina su contenido.

EJEMPLO DE UTILIZACION DEL PROGRAMA COPYUPI EN LA MODIFICACION DE UNA CABECERA

```
COPYUPI © 1985 MICROHOBBY
1 TIPO          program
2 NOMBRE        loader
3 LONGUITUD    107
4 COMIENZO     1
5 VARIABLES    107
6 TIPO DE FLAG 0
U - volver al menu
C - cambiar datos
```

Antes de modificar Basic con autoejecución en línea 1

```
COPYUPI © 1985 MICROHOBBY
1 TIPO          bytes
2 NOMBRE        loader
3 LONGUITUD    107
4 COMIENZO     30000
5 VARIABLES    107
6 TIPO DE FLAG 0
U - volver al menu
C - cambiar datos
```

Después de modificar Bytes, ubicado a partir de la dirección 30000

CONTROLES DE COLOR Y CURSOR

VALOR COMENTARIO

- | | | | | | | | |
|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------|----|-----------------------------------------------------------------------------------------------------------------------------------------|----|-------------------------------------------------------------------------------------------------------------------------------|----|---------------------------------------------------------------------------------------------------------------------------------------------------|
| 5 | Control de print con coma, sirve para que en este punto el listado se desplace a la próxima posición de TAB. Va solo. | 13 | que hacia la derecha. Código de Enter. Indica el final de una línea. Colocado en cualquier posición de una línea puede confundir al SO. | 18 | Control de flash. Indica si el flash está activado, si el próximo octeto es un 1, o no lo está, si el próximo octeto es un 0. | 21 | Control de over. Como los tres anteriores. |
| 8 | Cursor izquierda. Provoca el desplazamiento del cursor una posición a la izquierda sobreescribiéndose lo que a continuación vaya encima del anterior carácter. | 14 | Código de un número. Precede a los cinco octetos que representan a un número en coma flotante. | 19 | Control de brillo. Funciona de idéntica forma al control de flash. | 22 | Control de AT. Los dos octetos que le sigan indican las nuevas coordenadas por las que va a continuar el listado. |
| 9 | Cursor derecha. Igual que el anterior sólo | 15 | Control de tinta. El código que le siga indicará de qué color se va a poner la tinta. | 20 | Control de inverse. Como el control de flash y brillo. | 23 | Control de TAB. Funciona igual que el control de AT, pero con un solo octeto que indica la nueva columna hacia la que se va a dirigir el listado. |
| | | 17 | Control de papel. De | | | | |

Rutinas CM en la zona de variables del Basic

LA BIBLIA DEL «HACKER» (VI)

José Manuel LAZO

Una de las formas más habituales de guardar una rutina de carga de Código Máquina dentro de un programa BASIC es hacerlo dentro de la zona de variables. De esta forma, si alguien accede al listado no podrá verlo y si ejecuta comandos del tipo RUN o CLEAR, la rutina desaparece por arte de magia.

Un bloque de CM se puede guardar perfectamente en la zona de las variables, para comprender esto es necesario saber cómo el intérprete graba un programa en Basic:

Cuando damos la orden SAVE «nombre», el SO coge la variable PROG y toma la información que la misma contiene como el primer octeto a grabar, la longitud del Basic grabado depende de lo que marque la variable F-LINE que señala el final de la zona de variables del Basic. Además, en la cabecera del programa se guarda la longitud del listado Basic dentro del bloque grabado, que puede ser igual o inferior al mismo.

De todo esto se deduce que el señor que haga la protección puede guardar perfectamente un programa en CM en la zona de variables y grabarlo junto con el programa. Una consecuencia de lo mismo puede ser que si nosotros ejecutamos un RUN se nos borran las variables, y con ello el programa en CM con el consiguiente cuelgue.

Cuando nosotros grabamos un programa con AUTO-RUN no lo hacemos de forma que se haga un RUN a la línea que marquemos sino un GOTO, una expresión que sería adecuada es grabar un programa con AUTO-GOTO.

El código máquina cargador no tiene por qué estar necesariamente dentro del listado, al contrario, lo más sencillo para el programador es situarlo en un bloque de código grabado independientemente, aunque esto es más sencillo de desproteger. Sólo hay que averiguar la dirección donde se carga y donde se ejecuta.

Formas de ejecutar un CM. cargador

Partimos del caso de que en el programa en Basic, que actúa como cargador, no se ve una sentencia LOAD por ninguna parte, de esto se puede deducir que los demás bloques del programa se cargan desde CM. No vamos a entrar todavía en cómo carga el programa CM, pero vamos a ver las distintas

maneras que hay de llamar al mismo.

La forma más sencilla es RANDOMIZE USR dirección. Siempre que nos hayamos asegurado de que la dirección que se da sea la verdadera podemos pasar ya sin más al desensamblado, pero esta forma es poco corriente porque es muy fácil de desproteger y porque podría dar problemas si se tiene el interfaz I conectado.

Otra forma muy común es RANDOMIZE USR (PEEK 23635 + 256 * PEEK 23636) + n. Esto podría valer para arrancar un programa en CM ubicado en una línea REM al principio del listado. Si desajustamos desensamblar el CM tendremos que tener en cuenta que hemos cargado el Basic en otra posición para poderlo ver, por lo que en todos los CALL y JP que haya en su interior hay que calcular la dirección sobre la que funcionan.

Si tenemos CM en la zona de variables se puede usar la forma RANDOMIZE USR (PEEK VARS + 256 * PEEK VARS + 1) + N. Esto lo que hace es una llamada a una rutina a partir de lo que contiene la variable del sistema VARS. Cuando nos encontremos con ella habrá que tener cuidado, si estamos viendo el programa sin el Copyline, de no hacer ninguna operación que modifique las variables.

Otra manera de llamar a un programa en CM desde el Basic sin que esta llamada se note es hacer un POKE en la variable del sistema ERR SP o puntero de la dirección debido a que tiene un nivel de protección más superior a las anteriores. Vamos a estudiarla detalladamente.

Cuando se ha de presentar un informe de error el SO mira la variable ERR SP que indica la dirección del elemento de la pila de máquina que contiene, a su vez, la dirección donde se hallan las rutinas de tratamiento de errores y, por lo tanto, transfiere el control del programa a esa dirección.

El SO hace esto así por varias circunstancias, pero la más importante es que en el momento en que se produce el error normalmente la pila de máquina

está desequilibrada por lo que un simple RET produciría que el error no se pudiera tratar o que se colgara el ordenador.

Por esta razón, lo que se hace es guardar en esta variable de dos bytes la dirección del elemento de la pila donde se halla el retorno de error. Así, cuando el error se produce el SO mira esta dirección y hace un salto a la misma.

El programa que haga la protección puede aprovechar esto para pokear en esta variable una dirección y luego producir cualquier error, o bien por los métodos normales, BORDER 9 por ejemplo, o bien pokeando también en la variable del sistema ERR NR la cual se encarga de contener el informe de error que se ha producido.

Con esto se consigue que el SO haga directamente un salto a una rutina CM que se encuentre ubicada en la dirección contenida, a su vez, en los dos bytes haciendo lo que apunta la variable.

En esto se basa la protección turbo a nivel Basic, pero de eso ya hablaremos más adelante.

Por lo general la filosofía que hay que seguir a la hora de entrar en un programa Basic es muy sencilla:

- Modificar la cabecera por bytes para poderlo ubicar en otra dirección.
- Examinar el listado con el Copyline detalladamente, un POKE que se pase por alto puede ser luego una muralla infranqueable.
- Estudiar la carga de los demás bloques del programa, es posible que creamos que está superprotegido y luego sea un juego de niños.
- No dejarse engañar: muchas veces sentencias de un listado Basic pueden estar «de adorno» para confundir al «Hacker». Tampoco se debe despreciar ninguna: un simple BORDER 5 puede significar que luego se chequea la variable del sistema BORDER para ver si está del color previsto.
- En algunos listados los literales ASCII rotocodadas proliferan como setas, mientras que en otros no se ha usado esta protección.
- Es muy interesante que mientras se va viendo el listado con el Copyline se vaya abriendo en un papel un listado «limpio» para que después de quitar la «paja» se pueda estudiar con más facilidad.

Rutinas de carga en Código Máquina

LA BIBLIA DEL «HACKER» (VII)

José Manuel LAZO

Ya es hora de que estudiemos las distintas maneras en que puede cargarse un programa desde CM. En primer lugar veremos la correcta utilización de la rutina LOAD de la ROM.

Partimos del supuesto de que habéis aprendido ya los fundamentos que se han sentado en los capítulos anteriores sobre protecciones a nivel Basic, aunque volveremos a ello después, cuando nos centremos en la protección «turbo».

Ahora vamos a introducirnos de lleno ya en lo que se puede llamar protecciones a nivel CM, esto es, cuando el cargador del programa ejecuta una llamada a una rutina en CM para seguir cargando el resto del mismo.

La estructura general del cargador CM puede ser ésta:

LD A,255	LD IX,25000
LD IX,16384	LD DE,40000
LD DE,6912	SCF
SCF	CALL #556
CALL #556
LD A,255	

Una asignación de vectores y unas llamadas a rutinas de la ROM. Este es el caso más sencillo que usa la rutina de la ROM LOAD ubicada en la dirección #556 (1366 en decimal).

La rutina LOAD

Es muy interesante antes de proseguir, echar un vistazo al funcionamiento de la rutina LOAD de la ROM. Si de todas formas deseáis profundizar más sobre el tema os podéis dirigir al Especial n.º 2 de MICROHOBBY, donde se trata con mayor detalle este tema.

Esta rutina utiliza el registro IX para contener la dirección de comienzo donde se van a cargar los bytes, el registro DE para contener la longitud del bloque y el registro A para el flag de identificación.

Pero ¡ojo!, esto carga sin la «cabecera» que contiene la información del

nombre y longitud de los bytes, lo cual trae consigo el que se cargue lo primero que entre.

Si observáis el *Gráfico 1* podréis ver la manera en que están grabados unos bytes o un programa en la cinta: en primer lugar, el tono guía, y luego, la cabecera en sí que contiene un primer byte como flag de identificación (0) y otros 17 con la información de cabecera: nombre, comienzo, longitud, tipo y demás...

El segundo bloque es el que os interesa, es lo que se llama: «carga sin cabecera» ya que se prescinde de la misma, de lo cual se deduce que debemos de dar los valores de la dirección y longitud del bloque de datos en los registros que arriba se exponen.

Al elevar el banderín de Carry con la instrucción SCF provocamos que la rutina de la ROM cargue, ya que de lo contrario, sólo verificaría.

Primeros trucos en Assembler

Esta es una estructura general suponiendo que el programa al cargarse no tuviera cabecera y entrase a velocidad NORMAL. Por regla general se ha de buscar una asignación de vectores en los registros IX y DE los cuales indican comienzo y longitud, unas llamadas a rutinas cargadoras y un retorno a Basic o un salto ya al programa en sí.

Pero hay muchas formas de enredar esto tan sencillo para hacerlo menos inteligible.

Sentemos primero unos sencillos conceptos de Assembler:

En primer lugar la instrucción CALL dirección significa, como todos vosotros

sabéis, una llamada a una rutina en CM., pero agrupa una serie de operaciones como son:

CALL dirección= PUSH PC (Program Counter) JP dirección.

En segundo lugar, la instrucción RET que sirve para retornar de una rutina CM. Tendría el siguiente significado, en nuestros mnemónicos imaginarios:

RET= POP PC o JP (pila).

De esto, se deduce que cuando efectuamos un CALL guardamos la dirección de retorno en la pila, y si efectuáramos otro se guardaría la nueva encima sin borrarse la antigua de forma que los RET que se vayan ejecutando van sacando esas direcciones de retorno de la pila.

Es muy sencillo pokear la dirección de retorno en la rutina cargadora y cambiarla por otra para que la instrucción RET del final no ejecute un retorno a Basic como sería de esperar, sino un salto directo al programa en CM.

Por ejemplo:

LD IX,25000	POP HL
LD DE,1000	LD HL,25000
LD A,255	PUSH HL
SCF	RET
CALL #556	

Esto sería un ejemplo de una rutina que cargase otra en la dirección 25000 y a continuación ejecutase una llamada a esta rutina con la instrucción RET, fijaos bien en su estructura pues abunda más de lo que sería de esperar.

Otra forma es terminar el programa en vez de con un RET, con un JP a la rutina LOAD de la ROM, ya que el RET se halla en la propia rutina de la ROM.

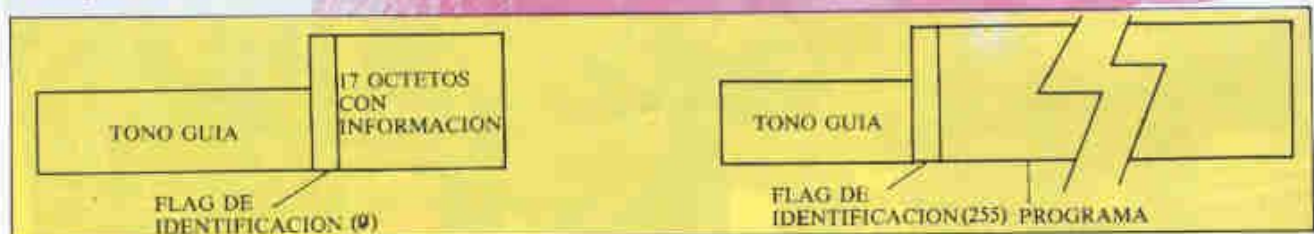


Gráfico 1.

Protecciones en la rutina de carga

LA BIBLIA DEL «HACKER» (VIII)

José Manuel LAZO

Cuando analizamos por primera vez una rutina de carga en Código Máquina es muy fácil que se nos pasen algunas cosas por alto, como por ejemplo, que la rutina cargadora esté en una dirección en donde se va a ubicar el propio bloque de bytes, solapándose con la primera.

Si el programa que estamos viendo tiene esta característica olvidarnos de todo lo que veas después del CALL a la rutina cargadora ya que después de concluir la carga es muy improbable que la rutina permanezca intacta. Esta es la protección del solapamiento del cargador. Incluso es posible que el programador que protegió el programa haya puesto cuidadosamente coherentes después del CALL a la rutina de carga, pero ello es únicamente para distraer.

Otra protección con la que nos podemos encontrar bastantes veces es que una vez sumada la longitud del bloque de código con la dirección donde se ubica este da un número mayor de 255 por lo que la carga, después de terminarse con la dirección más alta de la memoria del ordenador continúa con la ROM, y lo más probable que suceda es que termine hasta la pantalla. Ello no es más que una pérdida de tiempo y normalmente se utiliza para que al cargar el bloque de bytes, no caiga en el modo contador.

Carga desde la rutina LOAD de la ROM

Siempre que nos encontremos una rutina estándar de la ROM pero sin cabecera, lo mejor es averiguar la longitud y dirección donde se ubica con los bytes y hacer una cabecera a medida para poderlos cargar desde luego en otra dirección más cómoda para su estudio. Esto se hace de la siguiente manera:

Si vemos que el registro DE se le asigna el valor 3000, por ejemplo, desde la longitud del código es de 3000. Hay que apuntarlo para que no se nos olvide.

Luego buscamos el comienzo en el registro IN; supongamos que es de 2500.

En este ejemplo, para crear una cabecera teclaríamos: SAVE "nombre" CODE 2500,3000, pero grabaríamos sólo el primer bloque (cabecera), cortando la grabación justo en el espacio vacío entre ambos. A continuación, con el Copy/Print grabaríamos después el bloque sin cabecera para poderlo cargar más fácilmente.

Si su dirección de comienzo nos lo permite, se puede cargar en el sitio de trabajo o, mejor, ejecutando previamente un CLEAR dirección 1, y luego ubicar un desensamblador en algún sitio de la memoria libre para cargar a su desensamblado. Para este cometido es fundamental disponer de un desensamblador perfectamente reutilizable, como por ejemplo, DEMONS.

En el caso arriba expuesto de que los bytes que se cargan se solapan con la rutina cargadora no hemos podido averiguar la dirección de comienzo del programa. Es muy sencillo saberla; es la dirección de memoria que sigue al CALL a la rutina cargadora; por ejemplo:

```
25009 LD IX,3000
      LD DE,3000
      LD A,255
      SCI
      CALL LOAD
25013 PATATIN
      DA AYAN
```

La dirección de ejecución del código está aquí en la 25013, así que a partir de ahí es donde debemos desensamblar.

Enmascaramientos y checksums

Vamos a introducirnos ahora en el estudio de las distintas protecciones que se pueden encontrar en el código objeto del programa principal (una vez cargado) como puede ser checksums, enmascaramiento con el registro OR y otras alteraciones por el estilo.

Primero de los casos que ya tenemos el programa bien estudiado y sabemos donde se sitúa y en qué dirección termina.

Abordar ahora este asunto puede parecer difícil ya que faltan por explicar las rutinas de carga distintas a la normal y otras cosas interesantes, pero no lo es tanto si se piensa que este tema se encuentra dentro de las protecciones en CM.

Una vez que empezamos a desensamblar el código objeto se puede pensar que todo lo que encontramos a continuación está exento de protecciones y que ya tenemos un campo libre, sin ninguna trampa que nos estorbe. Nada más lejos de la realidad, ya que el código objeto del programa puede muy bien estar protegido de muchas formas por las protecciones que a continuación vamos a explicar. Esto ya no lo hace el programador para evitar la copia fraudulenta de su producto ya que se supone que si hemos llegado a este punto también podríamos copiarlo, sino para eludir el que pueda verse CÓMO lo hecho ciertas rutinas y evitar que otras personas puedan copiarlas.

Checksums

Checksum es una palabra inglesa que significa literalmente suma de chequeo y eso es lo

que es, una suma de todos los bytes que componen el programa y una comparación con una cifra. Huelga decir que si no coinciden el programa se colgará o saltará a la dirección 0.

El checksum se hace principalmente para evitar el que podamos modificar con algún POKER el programa en cuestión. Normalmente no se ve en un principio, y hasta es posible que el programa lo podamos arrancar modificado sin darnos cuenta. Pero es probabilísimo que se halle en el programa que estamos mirando y que actúe en el momento en que menos nos lo esperamos. Esta protección se conoce como BOMBA DE TIEMPO en la jerga informática.

Una forma genérica de checksum sería la siguiente:

```
LD HL,25000
LD BC,0000
LD A,0
LOOP: NOR (HL)
      INC HL
      DEC BC
      LD A,B
      OR C
      JR NZ,LOOP
CP (HL)
JP NZ,0
RET
```

Este es un método sencillo, pero es el más utilizado debido a que consume poca memoria. Otra forma puede dar lugar a un checksum podría ser que en vez de efectuar una operación NOR en la instrucción LOOP se efectuara en ADD, con resultados ligeramente distintos.

Se podrían añadir varios checksums seguidos con diversos sistemas, con un alto grado de inteligencia en las operaciones realizadas, pero, afortunadamente, en nuestros modestos Spectrum no se pueden desperdiciar unos preciosos bytes en codificar algo tan complejo (o si...) por lo que será, normalmente un simple checksum, eso sí, debidamente escondido, es decir, que no estaca en la línea de desensamblado que normalmente seguimos.

Una solución para evitar los efectos de un checksum puede ser un simple POKER en una dirección de memoria que no se use pero que sea una de las posiciones que espere el checksum, cambiando los bits POKETS que vamos a realizar. Esta simple solución evita el simple desensamblado que normalmente es el método utilizado para realizar la comprobación.

Toda esta ya no hemos conseguido encontrar la rutina que lo efectúa dado que entonces sería necesario grabarla y se encuentra.

Más protecciones en el código objeto

LA BIBLIA DEL «HACKER» (X)

José Manuel LAZO

Volviendo sobre el tema que empezamos la semana pasada, vamos a seguir estudiando las distintas protecciones que se pueden realizar sobre el código objeto

Usando el registro «R» se pueden hacer algunos trucos en el tema de protecciones. Uno de ellos, difícilmente controlable, es el siguiente.

Imaginemos que después de tener cargado el código objeto se pasa el control a una rutina desensambladora, y luego al programa principal. Pero es probable que antes de entrar en la rutina desensambladora nos encontremos con otras instrucciones, tal como estas:

LD A,76

LD R,A

Y que después nos encontremos nada relacionado con el registro «R».

Luego, cuando tenemos el código objeto limpio y queremos ejecutarlo veremos que en un punto específico del programa, no necesariamente al principio, se carga o salta a la dirección 0, reseteando el sistema.

Cuando nos encontremos con una situación como la anterior, podemos decir que estamos ante una protección de difícil control que puede abordarse de dos formas distintas, según el propósito que llevemos: analizar el programa o pasarlo a disco o microdrive.

Si lo que queremos es analizarlo deberemos buscar en todo el programa o el sitio donde se efectúa la comparación con el contenido del registro «R» y quitarla. Este método es más tedioso que el que después se explica, pero tiene la ventaja de que ya deja el código limpio de protecciones y podemos empezar a analizarlo.

La manera de hacer esto es, o bien desensamblarlo o buscar el código de la instrucción LD A,R por todo el programa.

Esta última forma tiene pocas posibilidades de éxito, ya que es bastante probable que la rutina de comprobación esté enmascarada para evitar el que podamos encontrar la comparación con este método.

Si lo que deseamos es, sin embargo, pasar el programa a disco, podemos hacerlo con el código «sucio» e incluir la rutina desensambladora con la inicialización del registro «R».

Este tipo de protección la utiliza, por ejemplo, el programa NIGHT SHADE de ULTIMATE, el cual inicializa el registro «R» en una rutina en la memoria intermedia de impresora antes de pasar el control a la rutina desensambladora. Luego, en medio del programa se efectúa una comparación del registro «R» y si no corresponde, se salta a la dirección 0.

Los nemónicos falsos

Los nemónicos que manejan los pares de registros IX e IY, llevan los prefijos DD y FD, respectivamente. Cuando el microprocesador encuentra uno de estos prefijos en la memoria, sabe que el próximo octeto marca una instrucción del juego que existe para estos registros.

Las instrucciones que manejan el registro «HL», como pareja o separado, no necesitan de ningún prefijo.

Solamente está el byte de la instrucción y a continuación, el del dato si existe.

Imaginémonos que a una instrucción normal de manejo del registro «HL» se le pone delante de un prefijo para manejo de los registros «IX» o «IY». Si además sabemos que no hay ninguna instrucción de manejo del registro «HL» que tenga el mismo código que las del manejo del registro «IX» o «IY», así tendremos certeza que con esta formación, una instrucción del Z-80 es imposible.

Realmente es imposible ya que juntamos los prefijos de un tipo de instrucciones con otras distintas. Con esto logramos varias cosas: la primera es confundir a todos los que no conozcan precisamente este tipo de instrucciones.

La segunda, es muy importante para el programador ya que se produce una instrucción que hace una cosa, por ejemplo: LD A,Y, y además tiene la particularidad de que ningún desensamblador puede leerla bien o si puede, la confunde con otra (LD A,L en caso del anterior ejemplo).

Todo esto puede llevar consigo que lo que nosotros estemos desensamblando sea mentira viendo en la pantalla una serie de operaciones que luego son otras. Vamos a ver esto más profundamente con algunos ejemplos ya que es un tema complicado.

Ejemplos

Supongamos que desensamblando un programa nos encontramos:

LD A,L

Si vemos que los códigos de esta instrucción son: FD, 7D podemos estar seguros de que no es LD A,L sino LD A,Y.

Otro ejemplo:

Si vemos INC HL y los códigos de operación son: DD, 23 podemos estar seguros de que no es INC HL sino INC IY.

Todo esto se puede averiguar de una forma sencilla con el MONS debido a que cuando se encuentra una instrucción de estas la pone de la siguiente forma:

Primero un NOP cuyo código de operación es el prefijo con un (*) indicando con esto que así se hizo algo que no está claro. Luego coloca la instrucción, pero operando con el registro «HL» tal y como si no tuviera prefijo.

Si vemos esto, la forma de interpretarlo es muy sencilla. Si el prefijo es FD entonces es que la operación se realiza sobre el registro «IY» y si es DD es con el registro «IX».

Con estos ejemplos creemos que será suficiente para su perfecta comprensión.

Rutinas de carga distintas a la de la ROM

LA BIBLIA DEL «HACKER» (XI)

José Manuel LAZO

Hay muchas formas de cargar un programa en la memoria del ordenador, aunque hasta ahora sólo hemos tenido en cuenta el uso de la rutina LOAD de la ROM. Sin embargo, ello no es necesario, y de hecho actualmente casi ningún programa utiliza este sistema.

Hace ya bastante tiempo que los programadores se dieron cuenta de que cambiando alguno de los parámetros de la rutina de carga: distinta velocidad en baudios, tono guía en otra frecuencia o con cortes como en la protección «turbo» o simplemente quitar el byte de paridad, se conseguía que los «copiadores» que por entonces existían no pudiesen copiar el programa.

Para ello es necesario desarrollar una rutina de carga distinta a la de la ROM y usar ésta en el cargador. Algunas de estas rutinas son extremadamente parecidas a la original ya que se han copiado íntegras y lo único que se ha hecho es variar los parámetros de ajuste de la velocidad. Otras, sin embargo, son de nueva concepción. En el siguiente cuadro podemos ver las variaciones típicas.

RUTINAS DE CARGA DISTINTA A LA DE LA ROM

- Distinta velocidad de carga
- Cambio de frecuencia en el tono guía
- Pausas en el tono guía (protección turbo)
- Cambio de longitud en el tono guía
- Quitar el byte de identificación
- Poner dos bytes de identificación seguidos
- Quitar el byte de paridad o falso
- Añadir otros condimentos a la carga:
 - Textos o movimiento de gráficos según se carga.
 - Distintas rayas de color en el borde.
 - Quitar las mismas.
 - Tonos guías en medio de los bytes.
 - Carga aleatoria.
 - Carga al revés.

En la rutina de carga de la ROM los registros tienen los siguientes cometidos: el IX contiene la dirección donde va a ir el byte en que se está cargando, el L contiene este

byte según se carga, el H contiene una suma de todos los bytes que se cargan para luego compararla con el byte de paridad, el último. El B siempre se encarga de guardar lo referente a los parámetros de la velocidad de carga, y el C guarda dos cosas: los tres bits de menor peso, el color actual del borde y el quinto bit el tipo de señal que se ha de encontrar en la entrada «EAR» de media onda o de onda completa.

De igual manera, el registro A contiene el byte de identificación o flag y los diversos banderines/estados de la carga.

Esto es así en la rutina de la ROM, pero si se trata de otra cualquiera no tiene por qué ser necesariamente de esta forma. Sin embargo, en la mayoría de los casos con que nos vamos a encontrar, la rutina de carga es una modificación de la de la ROM por lo que la utilización de los registros va a ser prácticamente la misma.

Problemas con el hardware

Ya os estaréis preguntando: bueno y el hardware ¿qué tiene que ver con esto? Pues mucho, como a continuación veremos. En el caso de que se utilicen rutinas de carga distintas, debido a que, por arte y gracia del señor Sinclair, ninguna rutina en CM se puede correr entre la dirección 16384 y 32767 de forma que ésta funcione a una velocidad constante.

La razón es que la ULA del Spectrum, que se encarga entre otras cosas de generar la señal de vídeo del ordenador, se halla conectada a la memoria según el sistema DMA o



En algunos casos las rutinas de cargas especiales se hacen por razones estéticas.

lo que es lo mismo, acceso directo para poder leer fácilmente el contenido de la memoria de pantalla.

Como sólo existe un Bus de direcciones en el ordenador, cuando la ULA está accediendo al mismo no puede hacerlo el microprocesador por lo que éste se detiene momentáneamente.

Esta circunstancia sólo sucede cuando el micro accede a las direcciones comprendidas entre la 16384 y la 32768, es decir, aquellas en las que el bit A15 del bus de direcciones está bajo (0) y el A14 alto (1) (página 1 si consideramos toda la memoria dividida en cuatro páginas).

De todo esto se deduce, y para que veamos las cosas más claras, que cualquier rutina cargadora distinta a la de la ROM, ha de estar ubicada forzosamente en los 32 K superiores de la memoria RAM porque si estuviera en los 16 K inferiores, o sea, en la página conflictiva, se vería interrumpida cada cierto tiempo por la ULA, por lo que la carga daría error.

Lo primero que tenemos que hacer es distinguir perfectamente la parte que gobierna la rutina de carga en el CM del cargador de la rutina propiamente dicha; hay que tener en cuenta para esto dónde se empieza a cargar la parte distinta del programa. Si comienza en la pantalla tendríamos que buscar un LD IX, #4000 debido a que, en líneas generales, este registro contiene la primera dirección donde se van a cargar los bytes cuando éstos entren desde la cinta.

Todo lo que llevamos dicho de protecciones usando la rutina de carga de la ROM, vale perfectamente para el caso que nos ocupa esta semana, sólo hay que tener en cuenta que en vez de hacer el CALL a la dirección #0556 se hace a donde está ubicada la rutina de carga.

El problema viene en aquellos programas en los que los bytes que se cargan de cinta, se solapan encima de la rutina cargadora o del programa que la maneja. Afortunadamente estos programas son los menos, tal y como comentamos hace algunas semanas, y el método que se ha de seguir para poderlos analizar es cargarlos algunas direcciones antes para que no se solapen.

Rutinas de carga aleatorias

LA BIBLIA DEL «HACKER» (XII)

José Manuel LAZO

Una de las protecciones más sorprendentes que podemos encontrar son las Rutinas de carga aleatoria. Puesto que en el Spectrum la aleatoriedad es perfectamente controlable, algunos programadores se aprovechan de esta facilidad para diseñar rutinas de carga vistosas, a la vez que muy difíciles de desproteger.

Hay un procedimiento bastante curioso para poder cargar bytes aleatoriamente de cinta, esto es, para que el bloque de datos que esté grabado en la cinta no se cargue secuencialmente desde la primera dirección a la última sino que se cargan unos bytes en una dirección, otros en otra, etc. Todo esto sin cabecear de por medio sino que hay una única cabecera al principio del bloque y luego éste de una longitud variable.

Para esto se han de utilizar rutinas de carga un tanto especiales que tienen dos entradas: la primera espera la cabecera y luego el número de bytes, y la segunda carga directamente los bytes sin esperar cabeceras de ningún tipo. Como el CM es tan rápido, resulta despreciable el tiempo que se desperdicia en la asignación de parámetros en la carga y la rutina no se entera de que ha habido una infima pausa entre el último byte cargado y el que va a entrar ahora.

Estas protecciones pueden dar muchos dolores de cabeza debido a que el programador puede perfectamente cargar un montón de cachitos del programa en distintas zonas de memoria, o lo que es lo mismo: el programa se halla desordenado dentro del bloque grabado en cinta.

Desgraciadamente una gran mayoría de los programas que hemos visto protegidos con este sistema tienen la particularidad de que uno de los primeros bloques que se cargan va encima del propio cargador, perdiendo

sentido la asignación de vectores que van a continuación.

Pero se dice que a listo, listo y medio, y este sistema presenta una gran ventaja sobre todos los llamados de carga rápida. Como la rutina cargadora tiene dos puntos de entrada, podemos usar el que espera los bytes sin como guía para desviar hacia la ROM un trozo de programa que al cargarnos estorbe, siempre y cuando no sea parte del mismo, esto es, sea una parte del cargador que conozcamos.

Como ejemplo os proponemos una corta rutina de carga aleatoria, a la velocidad estándar de la ROM. Esta rutina está muy optimizada aunque, eso sí, no es capaz de verificar, pero sin embargo, podemos jugar con ella un bloque de bytes de forma aleatoria tal y como hemos explicado.

Tiene dos puntos de entrada: LOAD y BYTES. Si entramos por LOAD conseguimos que está espere una cabecera al cargarse, pero si entramos por BYTES se procede a la carga de los bytes directamente.

La actualización de los parámetros es la normal en los dos puntos de entrada: en IX comienza, y en L longitud. Hay que tener en cuenta que el byte de identificación y el byte de paridad no intervienen, como tampoco se verifica si se ha producido un error de carga.

Esta vez no le acompaña el listado en hexadecimal dado que la rutina sólo se puede

usar desde CM y con las interrupciones deshabilitadas. Es completamente reubicable siempre y cuando la usemos en los 32 K superiores por las razones ya aludidas.

Esta técnica de carga se puede combinar con otra rutina, muy parecida, que efectúa una carga de bytes al revés; esto es, desde el final de la dirección de memoria especificada hasta el principio. Esto lo encontraremos en programas que carguen los atributos de pantalla desde abajo hacia arriba, por ejemplo.

Otros métodos «Hackerizantes»

De todas formas, si en la carga se han usado rutinas secuenciales (las normales) podemos usar un método paralelo para analizar el problema, y es cargar el bloque de bytes en cinta rápida con un copiador que tenga esta facilidad y pasar esto a carga normal con el mismo. Luego se opera como si de un programa en carga normal se tratase.

Si vemos que la carga es aleatoria y queremos usar este método porque nos parezca más sencillo, podemos luego usar la rutina que proponemos, que con toda seguridad ocupará menos memoria que la que utilice el programa, para efectuar el análisis del mismo.

RUTINA DE CARGA ALEATORIA

10	LOAD	IN	A, (#FE)	130	CALL	EDGE2	250	JR	NC, SYNC	380	CP	B		
20		INC	DE	140	JR	NC, START	260	CALL	EDGE1	390	RL	L		
30		RRA		150	LD	A, #C6	280	LD	B, #80	400	LD	B, #80		
40		AND	#20	160	CP	B	290	LD	A, C	410	JP	NC, BITS		
50		OR	2	170	JR	NC, START	300	XOR	#3	420	LD	(IX+B), L		
60		LD	C, A	180	INC	H	310	LD	C, A	430	LOOP2	INC	IX	
70		CP	A	190	JR	NZ, LEADER	320	JR	BYTES	440	DEC	DE		
80	START	CALL	EDGE1	200	LD	B, #C9	330	LOOP	LD	B, #B2	450	BYTES	LD	A, 0
90		JR	NC, START	210	CALL	EDGE1	340	MARKER	LD	L, 1	460	OR	E	
100		CALL	EDGE2	220	JR	NC, START	350	BITS	CALL	EDGE2	470	JR	NZ, LOOP	
110		JR	NC, START	230	LD	A, B	360	RET	NC	480	RET			
120	LEADER	LD	B, #9C	240	CP	#04	370	LD	A, #CB	490	EDGE1	EQU	#5E7	
										500	EDGE2	EQU	#5E3	

Código Máquina con Autoejecución

LA BIBLIA DEL «HACKER» (XIII)

José Manuel LAZO

Una forma de protección no muy usada, pero que podréis ver en algunos programas, es que el Basic cargador es ridículo y a continuación vienen unos bytes que se cargan y se ejecutan sin que ninguna sentencia los active. Veamos esto con más profundidad...

Es muy común la creencia de que se puede grabar en cassette un programa en CM o unos bytes de forma que se ejecuten a la hora de cargarse. Lo sabemos por la gran cantidad de cartas que recibimos preguntando cómo quitar el autorun a un programa CM. Pues bien, de una vez por todas, es imposible de todo punto grabar solo un programa CM en cassette de forma que con la sentencia inicial de carga, LOAD " " CODE, estos se ejecuten.

Lo que ocurre, y aquí viene la explicación, es que la protección consiste en grabar el CM, junto con un Basic que lo arranca todo en un bloque, digamos que es un programa en Basic con la cabecera como bytes.

¿Que cómo se hace esto?... Muy sencillo, probar lo siguiente y lo comprenderéis rápidamente.

```
10 PRINT «Antes de la carga»
20 SAVE «Ejemplo» CODE
23296, (PEEK 23641+256 *
PEEK 23642)—23296
30 PRINT «Después de la carga»
```

Ejecutarlo y salvar los bytes en una cinta, luego inicializar el ordenador y cargar el programa con LOAD " " CODE. Como podréis comprobar los bytes contenían nuestro programa en Basic que arrancó en la línea 30. Los programadores más «veteranos» que llevén más tiempo metidos en este mundo seguramente lo asociarán a la forma en que tenía el ZX-81 de grabar un programa con AUTO-RUN: si el programa se grababa en modo directo éste iba en el AUTO, sin embargo, si lo grabábamos desde una línea de programa éste se ejecutaba en la próxima línea.

Volviendo a nuestro Spectrum, la explicación de que esto se produzca así es que grabamos junto con el programa todas las variables del sistema, y recordemos que hay dos que marcan la línea y sentencia que se está ejecutando, pues bien, cuando cargamos los bytes inicializamos todas las variables tal y como estaban, por lo que el programa sigue corriéndose en la próxima sentencia a la grabación.

Cómo analizar estos cargadores

Lo primero es enterarse de dónde se cargan los bytes con el Copyzap (puede ser en la dirección 23296 u otra parecida), y una vez averiguado este dato cargar los 10000 bytes desplazados hacia arriba, es decir, si van en la dirección 23296 los cargaremos con la orden LOAD " " CODE 33296.

En segundo lugar, y con el Copyline en memoria, averiguamos la dirección del Basic en el bloque de bytes de la siguiente forma: PRINT PEEK 33635+256 * PEEK 33636, y luego la sentencia y línea de AUTO-RUN del programa así: PRINT PEEK 33618+256 * PEEK 33619 para la línea y PRINT PEEK 33620 para la sentencia.

Cuando sepamos todo esto arrancamos el Copyline y sólo nos queda darle estos datos para ver el programa Basic como si de otro cualquiera se tratase.

Esto, normalmente, nos lo encontramos sólo en programas cargadores con lo que su longitud es corta y lo podemos desplazar 10000 posiciones de memoria hacia arriba sin ningún problema, pero si nuestro caso fuera otro, que todo el programa estuviera grabado con

este sistema, tendríamos que ser más meticulosos.

En este caso lo podríamos cargar 10000 bytes más arriba, pero ejecutando un CLEAR dirección de carga-1 para evitar que la pila se nos corrompa. Si deseáramos tener el Basic limpio habríamos de enterarnos de su longitud y comienzo mirándolo en las variables del sistema del bloque de bytes cargado. Mirar también la dirección relativa con respecto al inicio del Basic donde están las variables del mismo y este dato, junto con su longitud, apuntarlos muy bien.

Es muy conveniente también enterarse de cual es la línea de AUTO-RUN en la correspondiente variable. Una vez hecho esto grabamos un bloque de bytes en una cinta con la siguiente orden: SAVE "Nombre" CODE inicio programa Basic, longitud del mismo, teniendo en cuenta que los datos están desplazados 10000 posiciones hacia arriba.

La forma de calcular la longitud es: PRINT (PEEK 33641+256 * PEEK 33642)—(PEEK 33635+256 * PEEK 33636).

Cuando lo hayamos grabado procedemos a cambiar la cabecera del bloque de bytes por una correspondiente a un programa Basic con el Copyzap, no olvidándonos de cambiar la longitud del programa Basic sin variables dentro del bloque, así como ponerle una línea de AUTO-RUN igual a 32768 para que éste no se ejecute al cargarse, y ya está.

Un último caso, que puede darse en este tipo de cargadores, es que el bloque de bytes empieza a cargarse en la pantalla y no termine hasta después de que ésta esté completa, o incluso continúe así hasta el final del programa estando este grabado en un solo bloque. Este supuesto lo analizaremos la próxima semana.

MINI MONITOR «TURBO». Para analizar los registros en los programas «TURBO»

LA BIBLIA DEL «HACKER» (XIX)

José Manuel LAZO

Uno de los problemas más grandes que podemos encontrarnos al analizar un programa TURBO es conocer el estado de los registros en cada momento, con objeto de poder predecir el curso lógico del programa. Para echaros una mano en este terreno, hemos preparado este MINI MONITOR «TURBO».

Esta corta rutina que os ofrecemos tiene como finalidad la de poder averiguar el valor de los registros y el contenido de la pila en cualquier punto del programa. Presenta las siguientes particularidades y defectos:

— Es corta, apenas 80 octetos y totalmente reubicable, por lo que, si es necesario cargarla en el sitio donde deseamos insertarla en el programa, podrá usarse.

— Proporciona además los registros cuando están sobrecargados o el bit de estado alterado, por lo que no es necesario tener un bit de ROM.

El único inconveniente que tiene es el hecho de que el programa no puede guardar el valor de los registros.

Utilización

La forma de utilizarla es la siguiente: si tenéis un ensamblador (o compilador) de programa TURBO, después de ensamblar o grabar el código objeto resultante en una cinta con la orden SAVE "break" CODE 60000,80. Si por el contrario no tenéis ensamblador, utilizar el listado hexadecimal número 2. Con el cargador universal, introducirlo y efectuar un DUMP en la dirección 40000. Luego lo

salváis con la orden SAVE "break" CODE 40000,80.

Para usarlo sólo tenemos que cargarlo en la posición donde precisemos un «breakpoint», luego ejecutar el programa (junto con la orden GO TO B). En ese momento nos saltarán los registros en la pantalla de una forma un tanto especial.

A cada registro se le asigna un bloque gráfico con rayas verticales y debajo de cada bloque se halla una máscara para poder leer o escribir fácilmente. Y debajo de cada bloque se indica un bit de estado, por lo que habiendo de tener en las ocho rayas de cada bloque (que corresponden con los ocho bits de cada registro en binario) a su equivalente en hexadecimal.

En la pantalla saldrán, por lo tanto, 14 bloques gráficos que corresponden de izquierda a derecha, a los registros A, B, C, D, E, H, L, X, Y, IV, CV. Los últimos dos bloques de la pantalla son el valor interno que contenía la pila en ese momento y una zona muy útil para desmenuzarnos.

El Mini Monitor «TURBO» ejecutará un programa completo al microprocesador, un programa de un DI seguido de un programa de control, lo cual nos facilita el poder «pi-

llar» con tranquilidad el valor de todos los registros.

Esta excesiva aridez para con el usuario es inevitable si se desea ocupar la menor cantidad posible de memoria, buscando total independencia de la ROM del ordenador.

Para ver, por ejemplo, dónde saltaría un RET en caso de que se cumpliera, podemos utilizar el Mini Monitor ubicándolo precisamente en la dirección del RET y viendo el contenido de la pila, es decir, los últimos bloques gráficos que da el monitor:

LISTADO 2

LINEA	DATOS	CONTROL
1	FDE5DDDE5E5D5C5F5E100	1549
2	4000078E68017877823	707
3	2372838241BF97CD608	883
4	67232323230000E72100	852
5	5885403E47772310FCF3	956
6	3E9A21204006040E0E5	628
7	7723238020F9E1241BF3	1004
8	76F30000000000000000	361

DUMP: 40000
N.º DE BYTES: 80

DESENSAMBLE DEL MINI MONITOR

10 ; MINI-MONITOR	160 LOOP2	LD B,B	310	INC HL	460	LD B,4	
20 ; POR J.M.LAZO	170	POP DE	320	INC HL	470 LOOP5	LD C,14	
30 ;	180 LOOP1	LD A,D	330	INC HL	480	PUSH HL	
40 ; ES REUBICABLE	190	LD (HL),A	340	INC HL	490 LOOP4	LD (HL),A	
50 ;	200	LD A,E	350	DEC C	500	INC HL	
60	ORG 60000	210	INC HL	360	JR NZ,LOOP2	510	INC HL
70	EVT 5	220	INC HL	370	LD HL,22528	520	DEC C
80	PUSH IY	230	LD (HL),A	380	LD B,64	530	JR NZ,LOOP4
90	PUSH IX	240	DEC HL	390	LD A,71	540	POP HL
100	PUSH HL	250	DEC HL	400 LOOP3	LD (HL),A	550	INC H
110	PUSH DE	260	INC H	410	INC HL	560	DJNZ LOOP5
120	PUSH BC	270	DJNZ LOOP1	420	DJNZ LOOP3	570	HALT
130	PUSH AF	280	LD A,H	430	DI	580	DI
140	LD HL,16384	290	SUB B	440	LD A,%10101010	590 ZINAL	
150	LD C,7	300	LD H,A	450	LD HL,16416		

No te turbes con el turbo

LA BIBLIA DEL «HACKER» (XV)

José Manuel LAZO

Por fin le ha tocado el turno a la protección TURBO, la cual hemos dejado para el final debido a su extrema complejidad. Apostaríamos sin riesgo de equivocarnos que una gran mayoría de vosotros, asiduos lectores, estábais deseando que llegase este momento.

El sistema turbo es, sin lugar a dudas, la protección de las protecciones. Tiene unas interesantes características y para lo antiguo que es, reúne casi todas las protecciones que hemos explicado hasta ahora en una sola. Sólomente esta protección justificaría la serie, y aunque sólo hablaríamos de ella, habríamos tocado, con ello, una gran mayoría. Por otra parte, cabe también esa satisfacción tan grande que siente un «hacker» cuando llega a lo alto de una protección considerada por todo el mundo poco menos que invulnerable.

Por todo esto y por mucho más vamos a tratar el sistema turbo desde un punto de vista muy especial, profundizando en ello todo lo posible por que, si conseguimos entrar a un turbo, ya nada se nos resistirá.

Tiempo ha que se trató este tema en nuestra revista, por aquel entonces se dieron unas pistas sobre puntos sueltos de la protección. Ahora vamos a ser más explícitos y explicaremos todos estos puntos y sus conexiones entre sí. Por ello, no os extrañéis si veis que un tema se queda colgado una semana para la próxima, esto es debido a la gran extensión requerida por cada fundamento para su perfecta comprensión.

El sistema turbo: fundamentos

En principio la protección Turbo tiene un basic con controles de color, líneas 0 y literales ASCII retocadas. Además, como luego se verá, el Basic tiene poco sentido, y casi todo lo que tiene es incoherente. Este Basic únicamente hace unos Pokes en las variables del sistema, y luego nada más, pero cuando esperamos que nos dé el informe de los nos encontramos con que ya está esperando la carga turbo.

El código máquina de la rutina cargadora se halla en el mismo listado Basic, aunque no se ve; de ubicarlo en su sitio y desenmascararlo se encarga otra rutina que también se halla en el Basic,

pero esta vez en la zona de variables.

La rutina cargadora es especial: tiene una velocidad de carga distinta a la normal y además espera un tipo de tono guía que tiene pausas (el clásico pitido entrecortado), esto es así para que ningún copión pueda copiarlo.

Quizá el corazón de la protección turbo es el sistema que emplea para detectar que se está utilizando una copia: cuando se hace la misma vía analógica, esto es, de un cassette a otro, ocurre que en el momento en que el original está silencioso el cassette que está grabando aumenta su sensibilidad de entrada, lo que provoca que se grabe ruido en la cinta, aunque este desaparezca en el momento en que entre una señal. Pues bien, la rutina cargadora verifica el ruido existente entre la cabecera turbo y el bloque de datos. Si este es excesivo, el cargador considera que es una copia ilegal y actualiza una variable del mismo indicando. Al terminar la carga del programa este se autodestruye en virtud del valor almacenado en esa variable.

Como arriba hemos dicho, la rutina cargadora, que ha de ir en los 32 K superiores de la memoria RAM, se halla en el listado del programa Basic, concretamente detrás de la última línea. De todas formas, no intentéis mirarlo con un desensamblador puesto que está enmascarado.

Para poder analizar la protección turbo hemos de centrarnos en dos objetivos primordiales y bien diferenciados: por una parte, lograr ubicar y obtener la rutina cargadora limpia de polvo y paja en su lugar de trabajo, esto lo logramos estudiando el listado Basic y las rutinas que incorporan el mismo como de punto de partida.

Después de obtener la rutina cargadora se puede pasar ya sin más dilación a su estudio con el fin de poder crear un bloque de código compacto del programa protegido, tanto si nuestra intención es transferir el programa a alguna memoria externa distinta al cassette (disco, microdrive, etc.) como si deseamos analizar el programa en sí.

Hablando de microdrives y discos: para poder estudiar un cargador (el Basic) turbo se ha de seguir una filosofía un poco diferente a la que hasta ahora hemos impuesto debido a la gran complejidad del Basic y las rutinas asociadas al mismo. Hemos de tener el Basic en su zona de trabajo, y no hacer ninguna modificación al mismo (nada de editar líneas, crear variables, borrarlas o mucho menos introducir más Basic). De esto se deducen dos cosas: la primera es que si tienes microdrive tienes que desconectarlo inmediatamente, piensa que este artefacto tiene la virtud de crear los mapas del microdrive cuando se ha de presentar un informe de error o hacemos uso de él, y como estos mapas consumen una importante cantidad de memoria y desplazan el Basic hacia arriba nos hace tediosa nuestra labor «hackeriana».

Respecto al disco ya es otro cantar debido a que, aunque también consume una pequeña porción de memoria, esta no se utilizará a no ser que hagamos una llamada al DOS por lo que, en principio, no molestará.

El segundo objetivo es encontrar un medio de ver el Basic sin tener ningún programa en Basic (curiosa ironía). La forma más factible de lograr esto es con el mismo programa que veníamos usando hasta ahora (el copyline) pero compilándolo con un buen compilador que acepte manejo de coma decimal flotante (unos resultados excelentes se consiguen con el «Coly» o el «FP Compiler»). Este de la coma decimal flotante es importante para que el programa funcione bien a la hora de presentar una literal ASCII retocada.

Si tuviéramos que ver listado en CM (todo se andará) utilizaríamos un desensamblador. Todas estas operaciones sin tocar el Basic cargador para nada.

Huelga decir que lo que primero tendremos que hacer es quitarle el auto-run al Basic para cargarlo con tranquilidad, aunque esta vez no habremos de transformar la cabecera en bytes.

El Basic de la protección Turbo

LA BIBLIA DEL HACKER (XVI)

José Manuel LAZO

Habiendo dado en el número anterior una pasada general sobre el tema, vamos a continuación a analizarlo en profundidad y quéé mejor que empezar por lo primero que nos encontramos: el Basic de un programa TURBO.

No vamos a referirnos a ningún programa en especial, y es por una razón muy importante: todos los cargadores turbo son hermanos gemelos, esto es, los listados Basic son parecidísimos, siempre y cuando ambos estén protegidos en turbo. De esta manera las rutinas cargadora y desenmascaradora tienen también una gran similitud.

De momento podemos ver el listado 1, es un ejemplo del Basic de un cargador turbo protegido. Debido a que a la impresora no le afectan los controles de color, no salen mensajes de error al imprimir. Este Basic, como podemos comprobar tiene 4 líneas cuyos números son 0. Y además, posee la protección de los literales ASCII retocados como vemos en el programa que hemos confeccionado (listado 2) y que viene a significar lo mismo que el primero, una vez «traducidos».

En líneas generales y después de analizar lo que hace en realidad este cargador, se puede llegar a concluir en lo siguiente:

- POKE dir E-LINE, WORKSP
- POKE dir ERR-SP, VARS
- POKE OLD PPG, NIWPPC
- POKE OSPPC, PPC

Por favor, pensar y reflexionar sobre esto, y hasta es interesante que intentéis sacar vuestras propias conclusiones antes de seguir leyendo.

Qué ocurre en realidad

Lo primero que vemos es que apunta la dirección de la variable del sistema E-LINE con el contenido de la variable WORKSP, eso puede llevarnos a pensar que con esta operación creamos una zona de comandos en modo directo que se ejecutarán después del listado Basic. La variable E-LINE se encarga de contener la dirección donde se hallan los comandos

que introducimos en modo directo en el ordenador y, la variable WORKSP apunta a la dirección de memoria donde se halla el espacio de trabajo del SO, así cuando este tiene que guardar datos de importancia para no perderlos lo hace en el sitio donde apunta la variable WORKSP.

Sin embargo, en la cinta el Basic está grabado sólo el listado del mismo y las variables Basic, estando el espacio de trabajo después de esto por lo que este POKE no hace nada esencial.

Después se poke otra dirección, hacia donde apunta ERR-SP con el contenido de la variable VARS. Este poke es el fundamental y es el que verdaderamente arranca el código máquina desenmascarador. Veamos como es esto:

En primer lugar tenemos que tener en cuenta que cuando el ordenador ha de presentar un informe de error (póiz OK en la pantalla se trata como un error), mira el contenido de esta variable, ERR-SP, para ver donde está la dirección de la rutina de error, normalmente en la ROM.

Esto es así porque al producirse un error normalmente la pila está desequilibrada con valores de los últimos cálculos que se han realizado. ERR-SP entonces, apunta a la dirección de la pila donde se halla el retorno por error. Si pokeamos con la dirección de las variables logramos que al presentarse un error no se salte a la ROM como sería de esperar, sino directamente a la zona de variables del Basic.

De esta forma cuando todo el Basic se ha ejecutado y se ha de imprimir en la pantalla el error «OK» para indicar que todo ha ido bien, se ejecuta primero a las variables del Basic, sino este donde se halla la rutina desenmascaradora.

Los últimos dos Pokes también van de momento, es decir, no hacen nada específico y de aquí que logran ser confusivos.

Ya sabemos como un listado Basic Turbo arranca la rutina desenmascaradora, considerando además la particularidad de que todos los cargadores Basic en el sistema Turbo arrancan de la misma manera. De esto se deduce que no es necesario que analicemos el Basic del cargador sino que, directamente, podemos ver donde se hallan las variables y empezar desde allí. De todas formas es interesante que analicemos por lo menos el primero para que comprendamos cómo funciona, por si en un futuro sale alguna protección TURBO II, por ejemplo, que utilice un sistema parecido, pero distinto.

LISTADO 1

```
0)
REM PROTECTED BY SPEEDLOCK
0: BORDER 0 PAPER 0 INK 0 B
RIGHT 1 CLS POKE 20624,0
0:POKE IPEEK 20641+256+PEEK 0
2642:PEEK 20640 POKE IPEEK 236
41+256+PEEK 20642+1:PEEK 20650
0:POKE IPEEK 20630+256+PEEK 0
2604:PEEK 20647 POKE IPEEK 236
23+256+PEEK 20634+1:PEEK 20646
0:POKE 23662:PEEK 20618 POKE
20663:PEEK 20619 POKE 20664:PE
EK 20601
```

Ejemplo de listado Basic protegido con el sistema «TURBO». Además de las líneas 0, los literales ASCII están retocados.

LISTADO 2

```
10 BORDER 0 PAPER 0 INK 0 B
RIGHT 1 CLS POKE 20624,0
20 POKE IPEEK 20641+256+PEEK 0
2642:PEEK 20640 POKE IPEEK 236
41+256+PEEK 20642+1:PEEK 20650
10 POKE IPEEK 20630+256+PEEK 0
2614:PEEK 23687 POKE IPEEK 236
10+256+PEEK 20614+1:PEEK 20628
40 POKE 20662:PEEK 20618 POKE
20663:PEEK 20619 POKE 20664:PE
EK 20601
```

El mismo listado, equivalente al 1, una vez quitadas las líneas 0 y traducidos los literales ASCII a su valor correcto.

Rutina desenmascarada del «TURBO»

LA BIBLIA DEL HACKER (XVII)

Jose Manuel LAZO

Habíamos quedado la semana pasada en que todos los programas TURBO tienen la rutina de carga enmascarada para dificultar las labores de análisis. En este capítulo explicamos cómo funciona la dichosa rutina desenmascaradora.

Una vez que tenemos localizada la rutina desenmascaradora en la zona de variables, podríamos pensar que con un RANDOMIZE USR a la correspondiente dirección arrancamos esta rutina, siguiendo sin dificultades la línea del programa. Nada más lejos de la realidad, pues la protección turbo se caracteriza, como hemos mencionado en algún capítulo anterior, por tener un CM desconcertante, superprotegido y totalmente oscuro. ¿Qué significa está? Básicamente que si no arrancamos la rutina con un GOTO 0 al Basic, no conseguiremos nada. Esto es así porque en el punto de entrada de la misma los registros han de estar de una forma precisa, como los han dejado los sentencias del programa Basic que se han ejecutado. De igual manera, la zona del Basic se ha configurado un poco con los pokes que nos servirán para nada concreto. De todo esto se deduce que si no se han hecho estas operaciones y los registros no tienen los valores esperados el cargador no funcionará.

Lo primero que tenemos que hacer, por tanto, es averiguar el valor de los registros a la entrada en el CM. Esto se puede hacer de varias maneras, pero la más sencilla consiste en tener un monitor en memoria con la facilidad de Breakpoints y colocar uno justamente en la dirección de las variables. Entonces salimos del monitor y tecleamos GOTO 0, con lo que el Basic del cargador hace los pokes y salta a lo que él espera sea su rutina desenmascaradora, pero antes tropieza con nuestro Breakpoint y volvemos al monitor para inspeccionar y apuntar el valor de los registros, tanto de los normales como de los complementarios.

A partir de este punto, y para asegurarnos de que vamos por buen camino, podemos ir colocando breakpoints en distintos sitios de la rutina desenmascaradora, ejecutándola hasta el mismo y luego volver a lanzarlo de forma que podamos comprobar si funciona o no funciona.

El CM. de la rutina desenmascaradora

Ahora nos tendremos que armar de una tremenda paciencia para estudiar el CM. desenmascarador, éste se halla protegido con nemónicos inexistentes y lo primero que tendremos que hacer es sacar un listado del CM y ponernos a traducir todas estas instrucciones incoherentes por otras que no lo son tanto. Este tema ya se trató en un capítulo anterior, por lo que no nos vamos a detener en él. De todas formas, se pueden estudiar las transformaciones que se han realizado sobre el Listado 1, el cual, además de servir

nos para aprender a hacer esto, es el principio de una rutina desenmascaradora de turbo.

Como podéis ver, este listado tiene muy poco sentido y habremos de tener una gran paciencia para descifrarlo y es muy probable que tengamos que volver a empezar desde un principio varias veces. Es conveniente apuntar en una hoja el valor de todos los registros e ir calculando «a mano» todas las operaciones de cada uno de ellos. Para esto es preciso tener grandes conocimientos de CM, tema éste que oscura al comando de esta serie, pero que podréis aprender en las páginas centrales de la revista.

Otra posibilidad es correr el programa paso a paso o por bloques con un monitor que tenga esta facilidad para ir viendo el valor que toman los distintos registros. Esto último es indudablemente más cómodo, sin embargo, tiene el inconveniente de que cuando nos encontremos con un LDIR habrá que tener cuidado con él.

El bucle del principio

Como podéis ver, en la dirección 615F del Listado 1 existe un RET PO; esta condición PO correspondiente al banderín de paridad-rebote, en este caso a la paridad, es decir, el número de bits elevados del registro A es par, entonces el RET PO no se cumple,

pero si es impar el RET PO se cumple y se tira un RET PE el que no se cumpliría. (PO = parity odd, PE = parity equal.)

Bueno, para el caso que nos ocupa ahora, este RET PO se cumple varias veces al principio, con lo que estamos en una caótica situación.

Al ejecutar la instrucción RET volvemos al Basic y el SO intenta nuevamente presentar el informe de error OK, para ello mira la variable del sistema ERR-SP que recordamos está modificada apuntando a un sitio donde se halla la dirección de las variables. Con esto conseguimos que el flujo del programa vuelva a la rutina desenmascaradora, pero esta vez con los registros un poco cambiados, y otra vez el RET PO, aunque antes de llegar al mismo se pasa por unas ordenes que los modifican aún más. Así entramos en este bucle unas cuantas veces para luego salir.

Cuando salgamos de él, otra vez hemos perdido el rastro de los registros porque se han ejecutado rutinas de la ROM. Se impone paciencia y volver a hacer la misma operación anterior. Situar un Breakpoint en el punto después del RET y lanzar la ejecución del mismo con un GOTO 0.

Para que os vayáis enterando también os ofrecemos en el listado, el estado de los registros después de este RET PO para este caso específico. Puede que no sirva para el que deseáis, pero os hacéis una idea.

LISTADO 1 DE LA RUTINA DESENMASCARADORA

6154 60	LD L,L		616C 77	LD (HL),A	
6155 45	LD B,L		616D 77	LD (HL),A	LD (R),A00
6156 48	LD B,B		616E F0*	NOP	
6157 79	LD A,C		616F 94	ADD A,H	ADD A,Iy
6158 92	SUB D		6170 F0*	NOP	
6159 ED57	LD A,I		6171 A0	XOR L	XOR Y
615B 00*	NOP		6172 39	ADD HL,SP	
615C 62	LD H,D	LD Ix,D	6173 F0*	NOP	
615D 44	LD H,H		6174 82	LD H,D	LD Iy,D
615E 53	LD D,E		6175 0D*	NOP	
615F 58	RET PO	SE CUMPLE	6176 54	LD D,H	LD D,Ix
6160 15	DEC D		6177 F0*	NOP	
6161 15	DEC D		6178 AC	XOR H	XOR Iy
6162 52	LD D,D		6179 77	LD (HL),A	LD (XFFFE),A
6163 09	EXX		617A 68	LD L,B	
6164 00*	NOP		617B F0*	NOP	
6165 68	LD L,B	LD X,I	617C 69	LD L,B	LD Y,C
6166 6A	XOR D		617D 62	LD H,D	
6167 ED62	SBC HL,HL		617E F0*	NOP	
6168 A8	XOR E		617F 63	LD H,C	LD Iy,E
616A F3	DI		6180 09	EXX	
616B 60	LD L,L				

La rutina desenmascaradora del turbo, paso a paso

LA BIBLIA DEL «HACKER» (XVIII)

José Manuel LAZO

Una vez que tengamos la rutina desenmascaradora limpia de polvo y paja, esto es, que hayamos «traducido» todos los nemónicos falsos por las operaciones que éstos realizan realmente, como vimos en el capítulo anterior, podemos pasar a su estudio. Esta rutina hace una serie de operaciones muy bien definidas y tiene unas características muy especiales.

Como ya dijimos, al principio se encuentra un bucle con un RET PO que tiene como fin el despistarnos.

En medio de la rutina hay una inicialización del registro R con un valor, el contenido en A, producto de una serie de operaciones más o menos oscuras. Este registro, el R, luego se utilizará para el desenmascarado de la rutina cargadora.

Hay un LDIR de una buena parte de la memoria que tiene como fin destrozarse cualquier intento de situar un programa en la misma. Los valores que pueden tomar los registros antes del mismo son variables y después de hacerlo toman otros que luego son necesarios, por lo que no es factible quitar esta instrucción de en medio.

Se asignan dos valores en la pila, uno, el primero, corresponde a la dirección del bucle desenmascarador y otro, situado después, que corresponde a la dirección donde arranca la rutina cargadora.

Hay generalmente una llamada a una rutina de la ROM ubicada en la dirección 3008 que se cumple, contrariamente a lo que podamos pensar. Esta rutina, la de ROM, se encarga de incrementar el valor de uno de los registros complementarios.

Directamente después se entra en el bucle desenmascarador. Este utiliza el valor que tenga el registro R a su entrada para desenmascarar el código objeto de la rutina cargadora. Como arriba se inició su valor con uno determinado, en este punto de entrada R contendrá un valor previsto para la protección. De igual mane-

ra el contenido del registro HL a la entrada de esta rutina marca el comienzo del código a desenmascarar, el DE el destino y el BC el número de bytes que se han de «pasar por la piedra».

El valor de estos registros a la entrada del bucle es producto de un montón de desequilibradas operaciones que se han realizado con los mismos a lo largo y ancho de toda la rutina desenmascaradora. Digamos que esto funciona así «por casualidad», aunque realmente demuestra la precisión «relojera» de esta rutina.

Durante toda la rutina se hacen un montón de operaciones con los registros normales y alternativos, lo cual imposibilita totalmente cualquier intento de vuelta al Basic una vez arrancada. De igual manera se realizan algunas operaciones sin sentido. Podemos decir, sin intención de ofender a nadie, que la rutina desenmascaradora parece ser producto de una mente totalmente desequilibrada.

De la primera parte de este último punto se deduce la imposibilidad de usar ningún monitor comercial para inspeccionar el valor que contengan los registros en algún punto. Esto es debido a que todos utilizan rutinas de la ROM para sus cálculos y presentación en pantalla, y estas rutinas no funcionan muy coherentemente si están corrompidos los registros alternativos, por lo que se hace necesario el buscar alguna forma de arreglar esto. En el próximo número solucionaremos adecuadamente este «pequeño» problema.

El corazón de la rutina desenmascaradora

Indudablemente, el sitio donde se realiza el desenmascarado de la rutina cargadora y a la vez su ubicación en la zona de trabajo, es en el bucle representado en el Listado adjunto (aunque no os lo creáis es un bucle).

Y es un bucle en virtud de los dos valores que en medio de la rutina se han «pokeado» en la pila. Examinemos esto con calma:

- Primero se carga en el registro A el valor que contenga el registro R; éste se puede considerar que devuelve un valor «aleatorio pero controlado».

- Luego se XOREa con este registro (el A) el contenido de la celdilla a la que apunta HL, que, recordemos, contiene la dirección de origen del código objeto enmascarado. Y después se guarda en esta celdilla el valor XOREado, con esto ya hemos desenmascarado el primer byte por lo que ya sólo queda trasladarlo a su sitio real, en la parte superior de la memoria, con la instrucción LDI.

- Esta instrucción coge el valor de la dirección a la que apunta HL y lo pone en la dirección a la que apunta DE, luego incrementa HL y DE y decrementa BC.

- Si BC vale 0, cosa que ocurrirá cuando el bucle haya terminado de desenmascarar todo el código objeto, el RET PO que viene a continuación se cumplirá, pero este RET no vuelve a Basic, sino que salta directamente a la rutina cargadora en virtud del segundo valor que se haya almacenado en la pila. Si no lo comprendéis repasar los capítulos anteriores de esta misma serie.

- Caso de que BC no contenga 0, se decrementa el valor de la pila en dos unidades, para que al ejecutarse el siguiente RET PE, si se cumple, se salte al primer valor introducido en la pila (en este ejemplo éste es el 61EE).

BUCLE DE LA Rutina DESENMASCARADORA

61EE	ED5F	LD	A,R	←	
61F0	AE	XOR	(HL)		
61F1	77	LD	(HL),A		
61F2	EDAB	LDI			
61F4	EB	RET	PO		PUNTO DE SALIDA
61F5	38	DEC	SP		
61F6	38	DEC	SP		
61F7	E8	RET	PE	SE CUMPLEN N VECES	

Cómo aislar la rutina cargadora del «TURBO»

LA BIBLIA DEL HACKER (XX)

J.M. LAZO

Dentro del complejo entramado que es una protección Turbo, podemos distinguir un bloque que es precisamente la rutina cargadora. Sin embargo, ésta se encuentra fuertemente enmascarada y además se reubica en otras direcciones de memoria una vez arrancada. Para aislarla y proceder a su cómodo estudio, os presentamos hoy la mini rutina «Break-Turbo».

En capítulos anteriores de esta misma serie y con referencia a la protección Turbo, hemos explicado que la rutina de carga se encuentra camuflada, siendo desenmascarada por otra rutina específicamente diseñada para ello.

Toda esta teoría está muy bien pero, ¿cómo nos las apañamos para conseguir el código desenmascarado si no podemos ver el Bugie para grabarlo una vez lanzada la rutina o tampoco situar ningún programa en memoria debido a que éste se nos reubicaría en virtud del LDR que se efectúa en mitad de la memoria?

Esta cuestión, después de pensarla muy bien, se nos ve solucionada por un dignísimo «Eso de orgullo» de los diseñadores de la protección (oh yes).

Resulta que si desensamblamos en ASCII la rutina cargadora nos podemos encontrar con una increíble sorpresa: en la misma se halla un mensaje en inglés con el copyright del autor de la protección y un «moralizable» consejo que nos dice que intentar romper esta protección puede ser perjudicial para nuestra salud (y casi tiene razón).

Este mensaje se halla detrás del bucle desenmascarador por lo que en este sitio podemos poner perfectamente un programa que nos salve el trabajo de memoria que nos interese cuando se termine

de desenmascarar la rutina cargadora.

La manera de hacer esto es muy sencilla. En el ejemplo que nos ocupa, situaríamos un programa CM, tal como el que se halla en el listado 1 en el punto en el que se encuentra el RET PO (dirección #61F4). Esta parte de la rutina desenmascaradora se publica en el número 92.

Después grabaríamos una caseteca en la cinta con la orden SAVE «cargadora»: CODE 32768, 32768 (solo la caseteca).

Por último, sólo tendríamos que arrancar el programa con un GOTO, o no sin antes haber situado la cinta virgen en nuestro cassette y veríamos como por arte de POKF se nos graba la rutina cargadora limpia de polvo y paja.

Una vez conseguido esto podemos ya relajarnos, pues lo más difícil ha sido ya superado.

Cuando hayamos llegado a este punto dispondremos en una cinta una grabación de toda la parte superior de la memoria, y como dijimos en capítulos anteriores, en este sitio es en donde debe ir la rutina de carga especial.

Lo primero que tenemos que hacer es localizar su punto de inicio, esto se puede hacer de dos formas distintas; la primera es haberlo visto en el RET PE que arranca esta cargadora en el bucle desenmascarador con el MINI-monitor.

La segunda bastante más sencilla y casi infalible, se basa en otro pequeño fallo de la protección turbo: resulta que todas las rutinas cargadoras de turbo que hemos tenido ocasión de analizar empiezan con la instrucción: LD SP, #FFFF.

Esta instrucción tiene como códigos de operación: 31, FF y FF, por lo que es muy sencillo cargar un monitor en la parte interior de la memoria, por ejemplo el MONS, y poniendo su Memory Pointer en la posición 8000 buscar (con la orden Get) por toda la memoria la sucesión de números: 31, FF, FF.

Veremos como a la primera nos sale el principio de la rutina cargadora. Esta se puede diferenciar en dos bloques principales: el primero es el que se encarga de cargar el programa en alta velocidad con esa caseteca tan especial y además se encarga de detectar si se está usando una copia pirata o una original con un sofisticado algoritmo que más tarde se explicará. Esta parte posee dos puntos de entrada: uno para cargar directamente y otro para cargar y luego comprobar la cinta.

El segundo bloque es el que maneja el primero actualizando los registros según sea necesario y haciendo las comprobaciones oportunas. La semana próxima trataremos en profundidad este tema.

RUTINA «BREAK-TURBO» LISTADO 1

10	ORG	#61F4
20	JP	PO, GRABA
30	DEC	SP
40	DEC	SP
50	RET	PE
60	GRABA	LD A, 255
70	LD	IX, 32768
80	LD	DE, 32768
90	SCF	
100	CALL	#4C2
110	JP	B

LISTADO HEXADECIMAL DE LA RUTINA «BREAK-TURBO»

Linea	Datos	Control
1	E2FA613B38E63EFFDD021	1494
2	0080110080037CDC204C3	926

**DUMP: 40.000
N.º BYTES: 20**

**UTILIZACION: LOAD
""LODE 25076
(Cargando antes el
programa «TURBO»)**

El corazón de la rutina cargadora del Turbo

LA BIBLICA DEL HACKER (XXI)

José Manuel LAZO

Como vimos la pasada semana, existen dos bloques dentro de la rutina cargadora del TURBO. El segundo es el verdadero corazón de la misma y esta semana lo analizaremos en profundidad.

Vamos a echar un vistazo a la rutina ejemplo del listado 1. En ella podemos observar cómo en el primer lugar se actualiza el registro SP (Stack pointer) con el valor #FFFF; esto nos pone la pila detrás para que no nos estorbe. Luego, si os dais cuenta, viene un bucle que se encarga de poner la memoria de atributos con el papel negro y la tinta también negra, lo que tiene como fin el que no se vea la pantalla hasta el final de la carga. No es imprescindible su presencia en el programa, pero hace bonito.

Después se inicializan unos registros: IX y DE, que, al igual que con la rutina LOAD de la ROM, se encargan de contener el comienzo y longitud del bloque que se va a cargar. En este caso se trata de la cabecera del programa turbo que tiene 20 bytes de longitud como podéis ver y se carga en la dirección #8000, osea, en el principio de la parte superior de la memoria.

El CALL que se ejecuta después a la dirección #FFF1 se ocupa de cargar los bytes que marquen los registros IX y DE, y después proceder a la identificación de la cinta; esto se hace, a grandes rasgos, de la siguiente manera: se mira si hay ruido en el puerto del cassette durante un máximo de 2 segundos, y se comprueba el ruido que ha entrado con un umbral que separa la copia legal de la pirata. Si la copia es legal habrá muy

poco ruido porque estará grabada con un sonido limpio; sin embargo, si la copia es pirata, en virtud del control automático de ganancia (CAG) que llevan incorporado prácticamente todos los cassettes para hacer las grabaciones, se grabará un pequeño pero suficiente zumbido en el sitio en donde la cinta debería estar silenciosa.

La rutina detectará esto, pero no saltará a la cero como sería de esperar, si ocurriese, sino que inicializa una variable del cargador con un valor específico (1), y la carga continúa tan normal como de costumbre.

La detección se produce más tarde

Después de haber cargado la cabecera tendremos que cargar el resto del programa; de esto se encarga la inicialización de registros que viene después: como veis se empieza a cargar en la pantalla (dirección #4000) y con una longitud de #BE00 que sumados a la dirección de comienzo nos indica que la carga termina en la dirección #FE00.

De esto se deduce una cosa muy importante: los bytes no se solapan con el cargador, el cual siempre está en una dirección intocable por la cinta. Además, no hemos tenido ocasión de ver ningún

turbo en el que los bytes se solapan con el cargador. Esto es así por una razón muy sencilla: si se solaparan se actualizaría la única variable que tiene el cargador y que indica la originalidad de la cinta, con el valor que entrase de la misma cuando se cargue esa dirección. Por lo que siempre se detectaría lo mismo: original o no original. Esta circunstancia la aprovecharemos después para poder pasar el programa a velocidad lenta con objeto de almacenarlo en un disco o microdrive.

Como veis, el CALL a la cargadora es en este momento distinto, ya que ahora no es necesario el detectar ruido después de la carga.

Una vez finalizado todo el proceso, se compara el valor de una celdilla de memoria con 0, esto es..., efectivamente, la variable que indica la originalidad de la cinta. La rutina situada en la dirección #FE00 se encarga de pokear toda la memoria con 0 si la copia es pirata. Si tuviésemos una cinta turbo ya muy gastada y que no entrara bien por esta circunstancia, sólo es preciso quitar este JP NZ, #FE00 para que el programa entre aunque tenga ruido donde no deba tenerlo.

Por último, ya se efectúa un LDIR de una rutina a la memoria intermedia de impresora y se pasa el control a la misma. A partir de aquí se puede considerar que arranca el programa en sí.

LISTADO 1

FF85 31FFFF	LD SP,#FFFF	FF96 111400	LD DE,#0014	FFAB C200FE	JP NZ,#FE00
FF8B 0640	LD B,#40	FF99 CD11FF	CALL #FF11	FFAE 21BCFF	LD HL,#FFBC
FF8A 21C05A	LD HL,#5AC0	FF9C DD210040	LD IX,#4000	FFB1 11005B	LD DE,#5B00
FF8D 3600	LFF8D LD (HL),#00	FFA0 11009E	LD DE,#BE00	FFB4 011100	LD BC,#0011
FF8F 23	INC HL	FFA3 C027FE	CALL #FE27	FFB7 ED80	LDIR
FF90 10FB	DJNZ LFF8D	FFA6 3AB4FF	LD A,(#FFB4)	FFB9 C3005B	JP #5B00
FF92 DD210080	LD IX,#8000	FFA9 FE00	CP #00		

Cómo pasar un programa «Turbo» a disco o microdrive

LA BIBLIA DEL HACKER (XXII)

Al estudiar el cargador de un programa TURBO pueden perseguirse varios objetivos, pero principalmente suele pretenderse adaptar estos programas a otro dispositivo de carga distinto al cassette, como suele ser el disco o el microdrive. En este capítulo explicaremos cómo hacerlo.

Para adaptar un programa Turbo a disco o microdrive tendremos que hacerlo de una manera muy especial, ya que la memoria RAM a partir de la dirección #5B00, o sea, la memoria intermedia de impresora, se ha de quedar de igual manera que si lo hubiéramos cargado de cinta, por el caso.

Desgraciadamente, el disco y sobre todo el microdrive utilizan la zona de antes del Basic para guardar datos de la carga. En el caso del disco no es tan problemático ya que solo se necesitan 112 octetos, pero el microdrive ocupa sus buenos 600 ó 700 octetos de memoria para almacenar.

Por eso se deducen varias cosas, la primera es haciendo un esfuerzo para que el que vamos a hacer valga lo mismo para microdrive que para disco, es que el trozo de programa que va desde la 23296 hasta la 25000 no lo podamos cargar directamente en su sitio, sino que hay que hacerlo en otro sitio y después reubicarlo.

El trozo de programa que esté situado en la memoria desde la dirección 25000, en adelante, hasta el final puede ser cargado desde disco en su verdadera dirección de trabajo sin ningún problema.

Lo que deberemos de hacer ahora es dividir el programa original en tres trozos bien diferenciados por una parte la pantalla, que aunque no es imprescindible para poder jugar, para distraernos durante la carga, sí puede ser objeto de un *checksum*, tal como se explicó en un capítulo anterior.

En segundo lugar el trozo de programa que esté en la memoria desde la dirección 23296 hasta la 24999, ambas in-

clusive, y en último lugar el trozo restante, hasta el final de la memoria.

La manera de sacar estos tres trozos del programa es muy sencilla: habremos de buscar un lugar libre de la memoria, que tiene que haberlo, de unos 40 ó 50 octetos donde poder situar un breakpoint al cargador. Este sitio normalmente está ubicado detrás mismo del cargador, cuando veamos al desensamblador solo saldrán incoherencias.

Pero cuidado, la rutina que carga la cabecera entrecortada utiliza una mini tabla de unos 5 octetos situada detrás mismo del cargador; esto quiere decir que estos valores son intocables.

Una vez encontremos este lugar libre y nos aseguramos de que realmente este libre (cuidado con la pila si lo encontramos muy arriba, o con los bytes que entran de cinta si está muy abajo), tenemos que ubicar en este sitio un programa *Breakpoint*, tal y como está impreso en el **Listado 1**.

El programa «Breakpoint»

Como podéis ver, este mini-breakpoint se encarga, en el momento de llamarlo, de grabar en cinta toda la memoria del ordenador tal y como este, en tres trozos, precisamente los pedazos de programa de los que hablábamos arriba.

Entre trozo y trozo, y antes de grabar el primero, espera la pulsación de la tecla *Func* para dejar un adecuado espacio entre los tonos guía de los distintos bytes que se van a grabar. En estos espacios silenciosos en la cinta, que habremos de calcular cronómetro en mano, situaremos después las cabeceras para poder cargar los bytes libremente desde Basic.

Una vez situemos el breakpoint después del cargador, teniendo en cuenta que no es reubicable y que la labor se habrá de hacer con un ensamblador, hay que cambiar el salto al principio del programa que se hace en el cargador después de cargar los bytes por otro que no lleve a la dirección donde hemos ubicado nuestro breakpoint. Con esto conseguiremos que cuando se acabe de cargar se salte a nuestro mini-programa que se encarga de grabarnos en una cinta el programa que ha entrado de cinta en Turbo, a velocidad normal.

Si no encontramos sitio en la memoria suficiente para el breakpoint habremos de hacer unas cuantas *peripetias* para poder desmembrar el programa: básicamente utilizaremos un breakpoint igual a este pero mucho más corto que se encargue de grabarnos sólo un trozo de programa; consecuentemente, tendremos que cargar el programa original tres veces para poder lograr los tres bloques a los que arriba aludíamos.

Este segundo breakpoint es el del **Listado 2**, y para usarlo las instrucciones son las mismas que para el primero; situarlo en un sitio libre y cambiar el salto del cargador al programa por otro que salte al breakpoint.

Por supuesto, hay que actualizar el valor de los registros IX y DE, de acuerdo con la longitud y comienzo de los bloques que vayamos a grabar, esto es, el primero 16384 y 6912, el segundo 23296 y 1704, y el tercero 25000 y 40535.

LISTADO 1

10	BREAKPOINT PARA SAVE	100	LD	IX,23296	190	CALL	SAVE
20	CALL ENTER	110	LD	DE,1704	200	CALL	ENTER
30	LD A,255	120	SCF		210	JP	#5B00
40	LD IX,16384	130	CALL	SAVE	220	SAVE	EQU #4C2
50	LD DE,6912	140	CALL	ENTER	230	ENTER	LD A,191
60	SCF	150	LD	A,255	240	IN	A,(#FE)
70	CALL SAVE	160	LD	IX,25000	250	BIT	0,A
80	CALL ENTER	170	LD	DE,40535	260	JR	NZ,ENTER
90	LD A,255	180	SCF		270	RET	
					280	ZINAL	

LISTADO 2

10	BREAKPOINT PEQUERITO.
20	!
30	LD A,255
40	BCF
50	LD IX,COMIENZO
60	LD DE,LONGITUD
70	CALL SAVE
80	JP 0
90	SAVE EQU #4C2
100	ZINAL

Cómo pasar un programa «TURBO» a DISCO O MICRODRIVE

LA BIBLIA DEL HACKER (Y XXIII)

José Manuel LAZO

La semana pasada nos introducíamos en el apasionante tema de la conversión de programas comerciales a otros periféricos más fiables y rápidos que el clásico cassette, como puede ser el microdrive o cualquiera de las unidades de disco existentes para el Spectrum. Ha llegado el momento de completar esta información adecuadamente.

Una vez que hemos conseguido, tal como explicábamos en el capítulo anterior, tener en una cinta de cassette los tres bloques que constituyen el programa Turbo, procederemos a introducir las correspondientes cabeceras, ya que en la cinta habíamos dejado previamente espacio para ello. Estas cabeceras son fundamentales para poder manejar el programa desde Basic y posteriormente grabarlo en disco o microdrive.

Esta operación es muy sencilla, situamos la cinta antes del primer bloque, la pantalla, y escribimos en modo directo: SAVE «pantalla» SCREEN5, pero solo grabamos la cinta al principio del segundo bloque, y hacemos lo mismo: SAVE «primer» CODE 23296, 1704, y por último, realizamos la misma operación con el último bloque: SAVE «orden» CODE 25000, 40535.

Una vez que tengamos esto hecho, hemos de proceder a grabar estos bloques en disco o microdrive y hacer un cargador para los mismos. De igual manera, al segundo bloque debemos de hacerle un reubicador para poderlo cargar en la dirección de pantalla y luego reubicarlo en su sitio.

El programa al disco

El reubicador para el bloque pequeño sería tal y como el que mostramos en el listado 3. Este va colocado detrás del bloque que se pretende reubicar en la dirección 23296 y siguientes. La forma de preparar este bloque pequeño es como sigue:

Primero volcamos el reubicador en alguna dirección de la memoria, por ejemplo la 61704 (no es capricho esta dirección tan exacta). Luego cargamos en la dirección 60000 el bloque pequeño y a continuación lo grabamos en el *drive* con la orden SAVE «nombre» CODE 60000, 1750.

El motivo de todas estas direcciones tan exactas es porque el trozo pequeño tiene un total de 1704 bytes, va en la dirección 23296 y lo cargamos en la 16384.

Después de pasar al *drive* el trozo pequeño de código pasamos el grande, operación esta muy sencilla ya que solo hay que hacer un CLEAR previo a la dirección 24999 y con un LOAD «nombre» CODE lo cargamos sin más, luego lo grabaremos en el *drive* con un SAVE «nombre» CODE 25000, 40535.

Respecto a la pantalla huelgan comentarios.

Cuando tengamos los tres bloques grabados en el *drive*, hemos de hacer un cargador para poderlos utilizar, este puede ser uno tal como el del Listado 4. El bloque pequeño se ha de cargar en último lugar, y arrancar con un RANDOMIZEUSR 18088, esta dirección es así porque $16384 + 1704 = 18088$, y recordemos que el reubicador del trozo pequeño se basaba de pues del mismo.

Como podéis ver el cargador que hemos hecho para el programa nos presenta una dirección del mismo para poderlo cargar, esto se ha previsto así para utili-

zaciones futuras de POKES, etc... De momento, indicar 0.

Recapitemos

Básicamente, la filosofía que hemos tenido que seguir para llegar a lo alto de la protección Turbo se puede resumir en los siguientes puntos:

- Estudio y comprensión del Estado Basic, sito en el cargador.
- Estudio de la rutina desenmascaradora del código objeto.
- Obtención del código objeto cargador limpio de polvo y paja.
- Estudio del mismo.
- Obtención en tres trozos gracias a un breakpoint en la rutina cargadora.
- Grabación de estos tres trozos en un *drive* (disco o microdrive, da lo mismo).
- Grabación en esta misma memoria de masa de un cargador capaz de cargar estos bloques y arrancar el programa.

El camino hasta aquí ha sido, quizás, un poco largo, pero apostamos a que ha valido la pena.

LISTADO 3

```
10 : REUBICADOR DEL TROZO
20 : PEQUEÑO,
30 :
40 : LD HL,16384
50 : LD DE,23296
60 : LD BC,1704
70 : LD SP,65535
80 : LDH
90 : JP PRINCIPIO
100 : DEL PROGRAMA.
```

LISTADO 4

```
10 REM Cargador para disco
Beta.
20 BORDER VAL "0": PAPER VAL "
0": INK VAL "4": POKE VAL "232624", VAL "4": CLEAR VAL "24999"
30 RANDOMIZE USR VAL "15363":
REM : LOAD "pantalla" CODE 16384,
6912
40 RANDOMIZE USR VAL "15363":
REM : LOAD "orden" CODE 25000,40535
35
50 RANDOMIZE USR VAL "15363":
REM : LOAD "printer" CODE 16384,1750
60 INPUT "Direccion? " : dir
70 INPUT "Valor? " : val
80 POKE dir, val
90 RANDOMIZE USR VAL "18088"
```