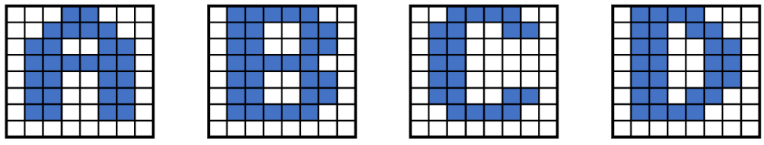
BANNER

Este documento explica un programa para VIC20 o Commodore PET, para visualizar texto ampliado en pantalla mediante “scroll” horizontal, como un “banner” o pancarta.

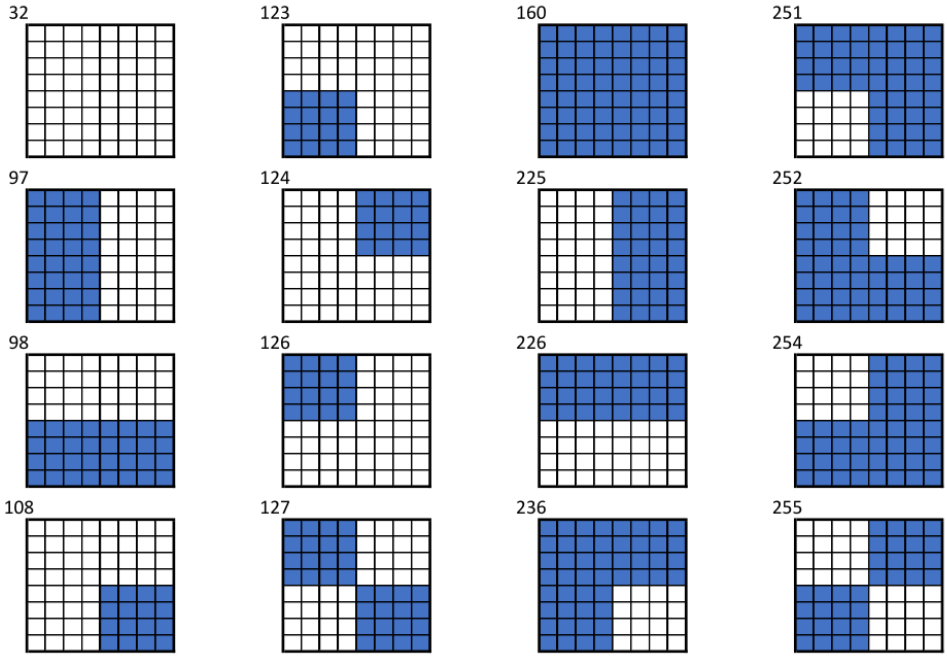
CONCEPTO

El juego de caracteres PETSCII está formado por 256 caracteres alfanuméricos y de gráficos sencillos que utilizan símbolos de un tamaño fijo de 8x8 pixels para ser visualizados en pantalla.

Esto es así en todos los modelos de 8 bits de Commodore, aunque los símbolos en los PET/VIC20 son algo diferentes de los de C64/C16/C128. Los caracteres de las imágenes siguientes corresponden a C64.



En el juego de caracteres PETSCII hay un subconjunto de 16 caracteres gráficos formados por “cuadraditos” de 4x4 pixels. Fue relativamente común utilizarlos en programas de representación de gráficos en los Commodore PET, que no tenían ninguna otra manera de representar gráficos. En adelante nos referimos a los “cuadraditos”, como “megapixels”.

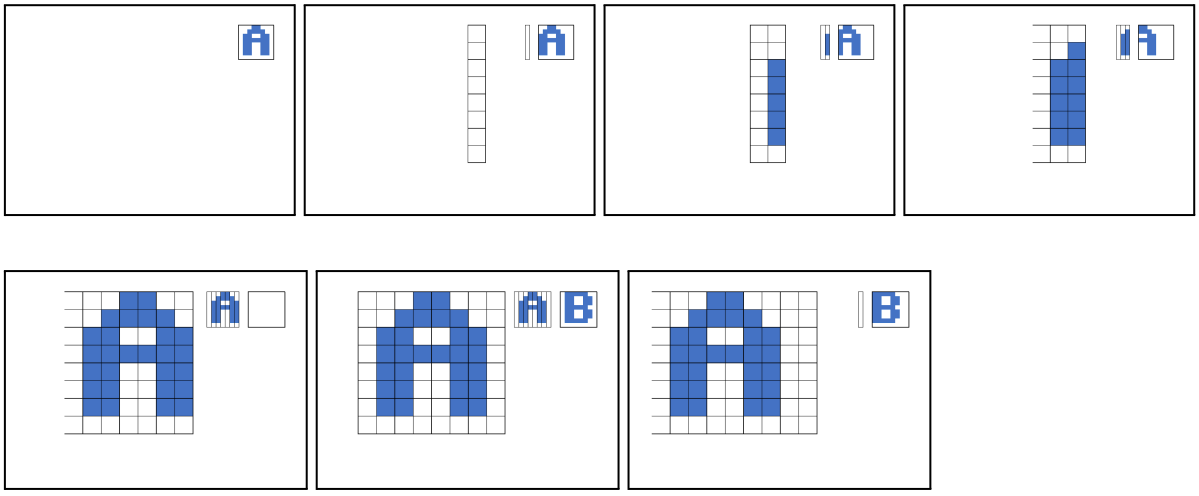


La idea en este programa es utilizar estos caracteres gráficos para componer texto ampliado, que se pueda leer desde lejos en el monitor. Es lo que se puede definir como la visualización de una pancarta o “banner”.

En el “banner” los “pixels” del texto van entrando por el margen derecho y desapareciendo por el margen izquierdo, el texto se desplaza hacia la izquierda. Esta forma de visualizar en pantalla se conoce como “scroll horizontal”.

A falta de poder mostrar el movimiento en este documento en vídeo, se muestra una secuencia de imágenes que pretende mostrar el mecanismo de “scroll”. No representan la visualización en el monitor sino el proceso de composición del texto ampliado.

La letra pequeña en la esquina superior derecha de los cuadros representa el caracter original. La letra mayor en el centro de la pantalla que se va desarrollando de derecha a izquierda representa el caracter compuesto por los “megapixel”. En la figura la escala de visualización es la misma, por lo que la proporción es fidedigna (la ganancia de tamaño).



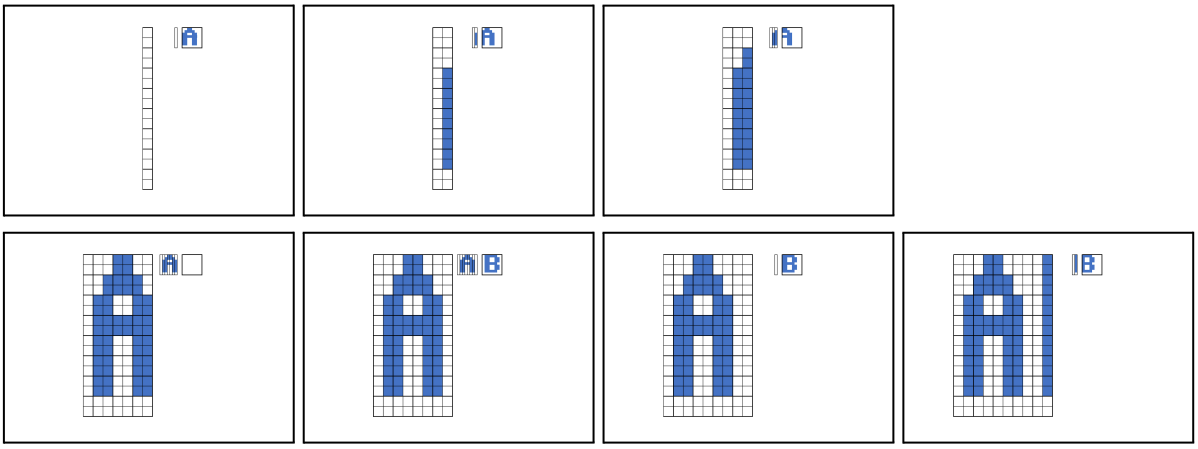
En concreto el caracter, cuya geometría en pixels se almacena en ROM, se despieza en columnas verticales de 8 pixels. Éstas se replican con “megapixels”, que se unen formando el mismo carácter, pero con la geometría ampliada. Las nuevas columnas van entrando por el margen derecho de la pantalla y se añaden al resto de la imagen que se desplazan a la izquierda.

Con este sistema cada caracter se acaba representando por un grupo de 16 caracteres, distribuidos en 4 columnas de ancho por 4 filas de alto.

DOBLE AMPLIACIÓN

Una vez escrito el programa y ejecutado una de las primeras cosas que se observa es que el “banner” visualizado en el monitor, obtenido al sustituir cada pixel por un “megapixel”, tiene un tamaño insuficiente. En especial la altura del texto resultante, 4 filas, resulta insuficiente.

Se puede hacer un “banner” del doble de altura, 8 filas en pantalla, sustituyendo cada pixel por una pila de 2 “megapixels”. Las siguientes imágenes muestran cómo empieza el “scroll”, con la doble ampliación vertical propuesta.



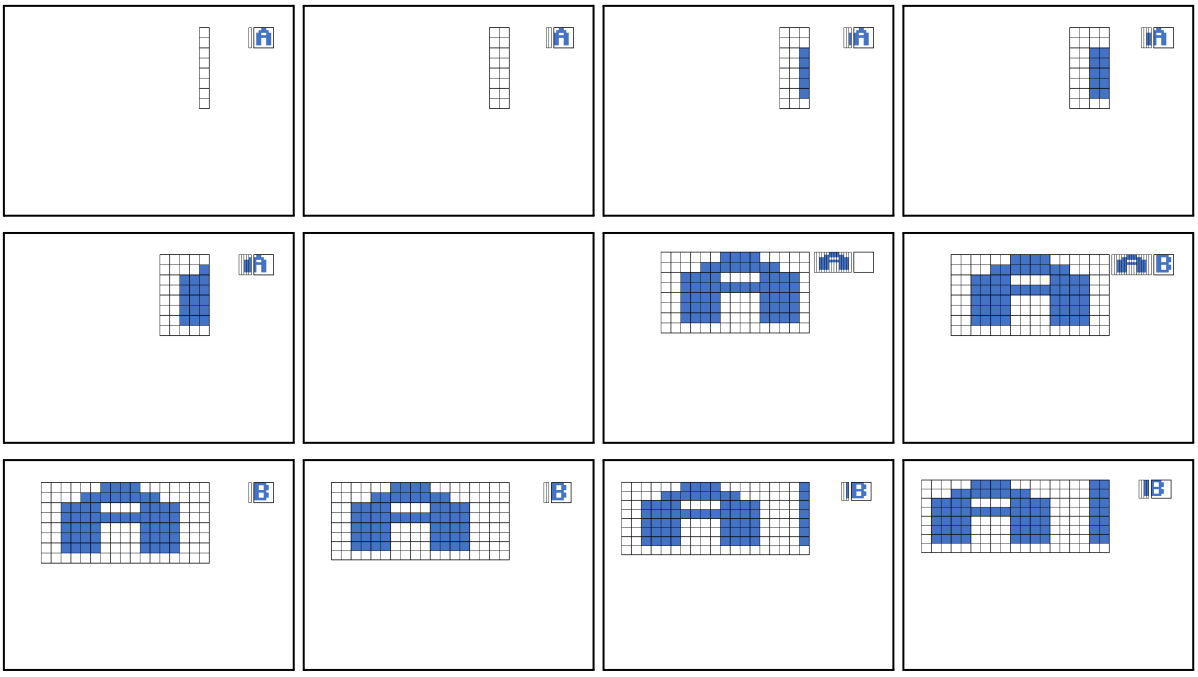
Se observa que utilizando esta doble ampliación vertical no se utiliza el juego completo de 16 caracteres con “cuadraditos”, sino el subconjunto de “rectangulitos” verticales de 4 x 8 pixels (H x V) que se visualiza aquí:

En adelante se habla del “factor de ampliación” como el número de “megapixels” por pixel que utiliza el programa para formar el texto ampliado del “banner”. En este apartado se ha mostrado cómo funciona la ampliación en el sentido vertical, al pasar del factor 1 al factor 2. En los apartados siguientes se muestran ampliaciones mayores.

AMPLIACIÓN HORIZONTAL

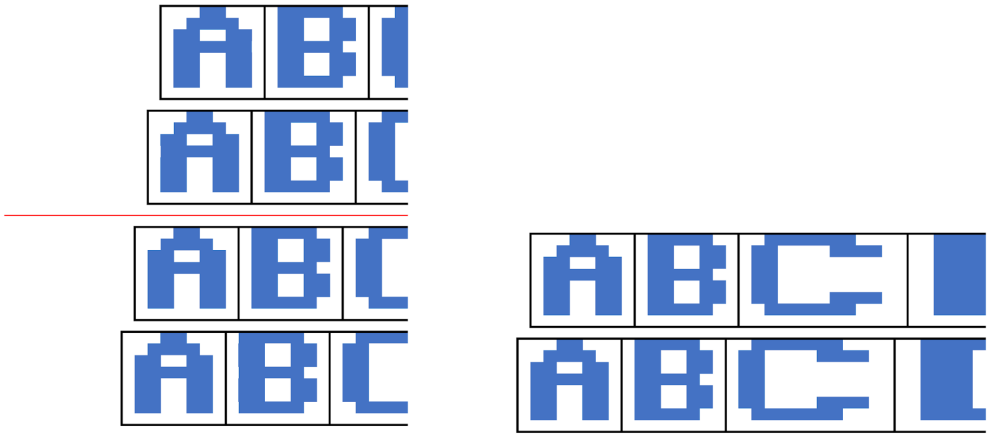
La ampliación de la escala horizontal puede ser necesaria también, más al ampliar la escala vertical.

La ampliación de la escala horizontal se efectúa repitiendo la entrada de los píxels por el margen derecho dos o más veces. Las siguientes imágenes muestran cómo empieza el “scroll”, con una ampliación de factor horizontal de 2 megapixels por pixel.



En este programa el mecanismo de ampliación funciona de manera diferente en horizontal y en vertical.

Tal y como está realizado este programa la escala en horizontal se puede cambiar sobre la marcha, y quedan visualizados a la vez caracteres con distintos factores de ampliación horizontal.



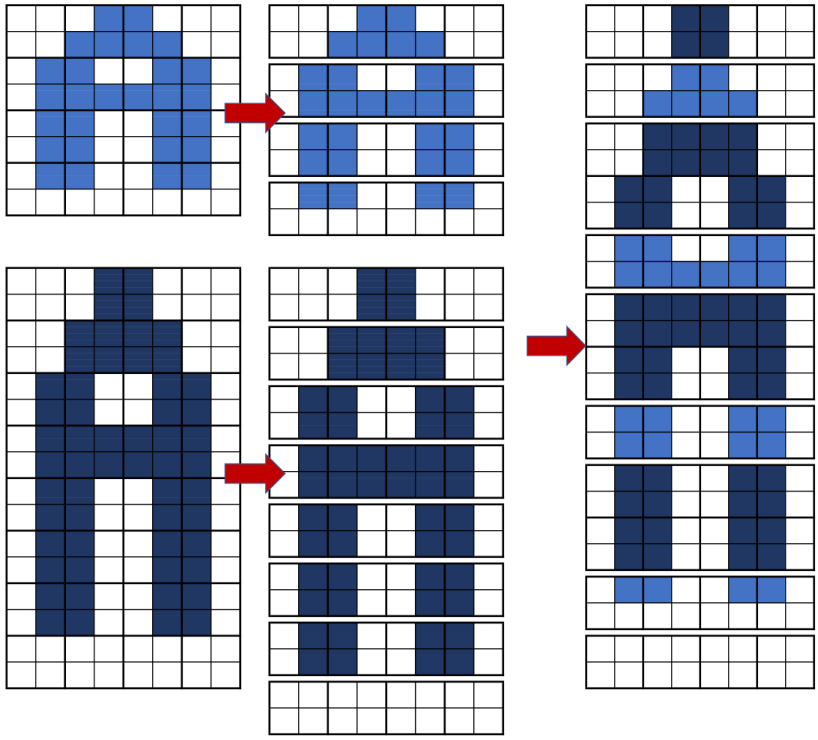
Este cambio sobre la marcha resulta más sencillo que redibujar el texto ya visualizado. En cambio, como se verá más adelante, si se modifica el factor de ampliación vertical sí se redibuja toda la pantalla.

En resumen, el programa permite un ancho del carácter ampliado de 4, 8, 12 y 16 columnas, compuesto según se repita 1, 2, 3 o 4 veces la entrada por la derecha de cada columna vertical de pixels.

AMPLIACIÓN VERTICAL

En apartados anteriores se ha mostrado cómo queda el texto al aplicar factores de ampliación vertical (en megapixels por pixel) de 1 y 2. Para factores de ampliación vertical de 3 y superiores no se generan gráficos nuevos, sino que se combinan los gráficos generados con esos dos factores de ampliación de 1 y 2.

Así, intercalando las 8 filas de “cuadraditos” y las 4 filas de “rectangulitos” se obtiene un “banner” de 12 filas de altura, correspondiente a un factor de ampliación vertical de 3. La figura siguiente muestra la idea:



Repitiendo las filas de “rectangulitos” se obtienen los “banner” de 16 y 24 filas de altura, correspondiente al factor de ampliación vertical de 4 y 6. Intercalando y repitiendo las filas de “cuadraditos” y de “rectangulitos” se obtiene el “banner” de 20 filas de altura, correspondiente al factor de ampliación vertical de 5.

Se ha mencionado que en este programa los cambios en el factor de ampliación horizontal se aplican para los pixels que entran por el margen derecho a partir del momento del cambio. En cambio al cambiar el factor de ampliación vertical los cambios se aplican a todo el texto que está presente en la pantalla. No es algo buscado sino lo que resulta más sencillo tal y como está construido el programa, a través de los códigos intermedios (ver apartado posterior).

VELOCIDAD DEL SCROLL

La velocidad de refresco de la pantalla es de 50 “frames” por segundo en sistemas PAL y de 60 “frames” por segundo en sistemas NTSC. En cada “frame” entra una nueva columna de pixels vertical por la derecha.

Mantener el ritmo constante de scroll y sincronizado con el framerate da la mayor suavidad a la visualización, pero resulta en una velocidad fija. Concretamente se representan 6.25 caracteres por segundo (PAL) o 7.5 caracteres por segundo (NTSC).

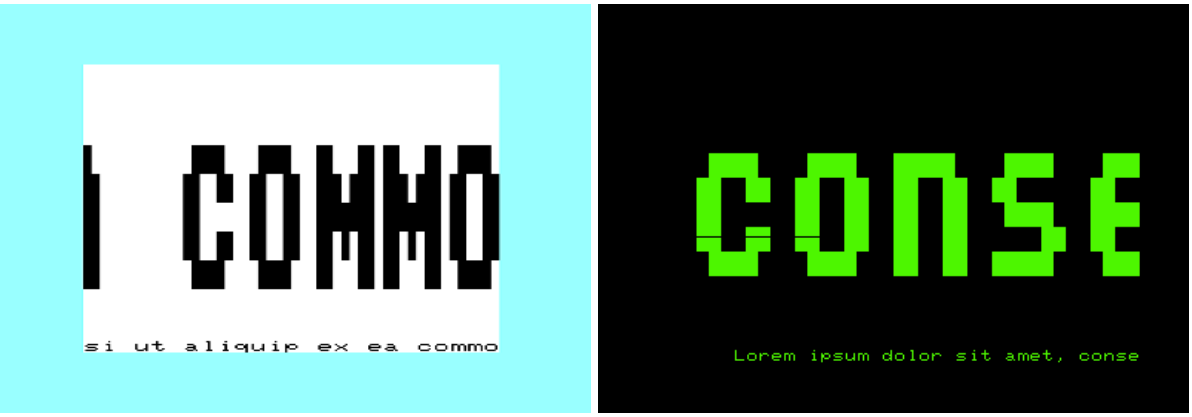
En realidad el movimiento suave no se consigue porque 1 de cada 8 llamadas a del programa BASIC a la rutina de lenguaje máquina (LM) que hace el scroll se toma más tiempo al tener que recuperar un nuevo carácter. Este tartamudeo se observa con claridad si se ejecuta el programa en el emulador al 20% de velocidad.

Por muy intensiva que sea la tarea que realiza el programa en LM, la parte del programa en BASIC siempre se toma más tiempo. Esta ralentización no sucedería si todo el programa estuviese escrito en LM.

Con factor de ampliación horizontal 1 se visualizan a la vez 10 caracteres (PET) o 6 caracteres (VIC20). Con factores de ampliación 2 o 3 se visualizan a la vez 5/3 caracteres (PET) o 3/2 caracteres (VIC20).

En la última fila de la pantalla, la 25, el programa muestra los caracteres sin ningún tipo de ampliación, así si se mira la pantalla de cerca se puede ver un trozo mucho mayor del texto del “banner”.

La figura siguiente muestra capturas de pantalla de los emuladores VICE de VIC20 y PET 40 COL.

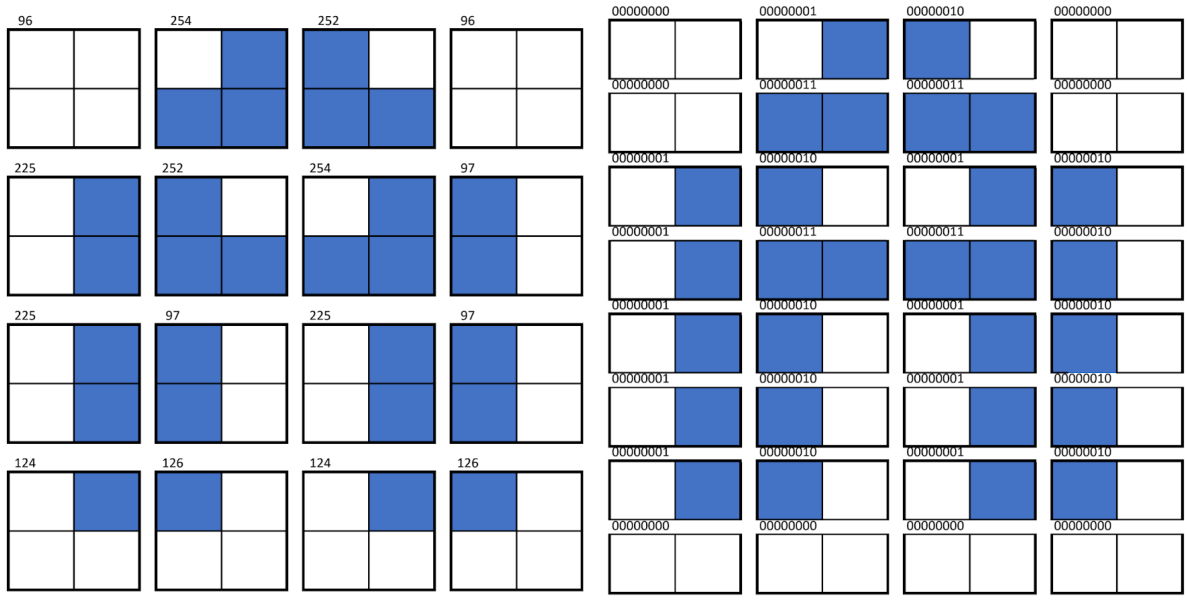


MECÁNICA DEL PROGRAMA

FORMATO INTERMEDIO

Para la conversión de los pixels de los caracteres en “megapixels” se utilizan unos códigos intermedios de 2 pixels por byte. En los caracteres originales se utilizan filas horizontales de 8 pixels por byte. El scroll en CM se consigue utilizando la instrucción de desplazamiento de bits a la izquierda.

En la figura siguiente en la mitad izquierda se muestra la representación de un caracter mediante caracteres de “cuadraditos”, en concreto un grupo de 16 caracteres (4x4). Los códigos 96, 254 etc. en texto pequeño los códigos PETSCII correspondientes.



En la mitad derecha de la figura anterior lo que se muestra son bytes que representan dos “megapixels” cada uno. Los valores posibles son 0, 1, 2 y 3.

Hay que emparejar los códigos intermedios para obtener los códigos de los 16 símbolos PETSCII utilizados. En la imagen las líneas negras un poco más gruesas muestran cómo agrupar esos códigos.

Las tablas siguientes muestran la relación de cada código intermedio con cada uno de los 4 caracteres del subconjunto formado por “rectangulitos”. También la relación de cada pareja de códigos intermedios con cada uno de los 16 caracteres del subconjunto formado por “cuadraditos”.

Factor vertical 2

codigo intermedio	binario	00	01	10	11
	decimal	1	2	3	4
codigo caracter		32	225	97	160

Factor vertical 1

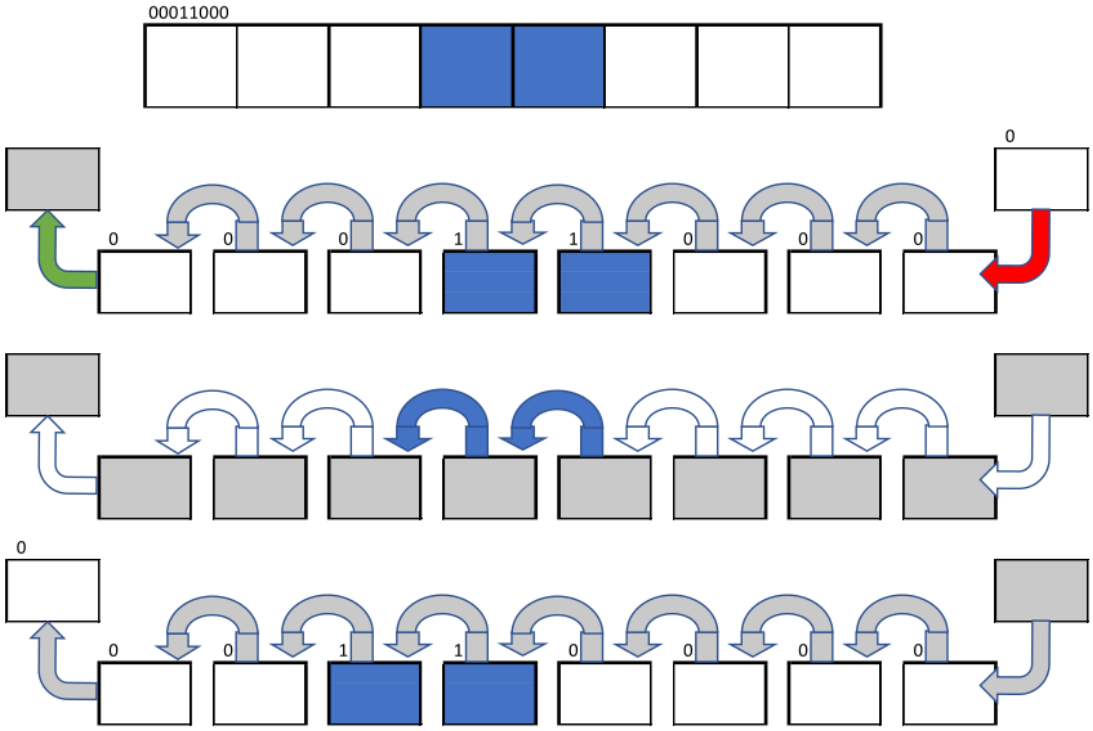
codigo intermedio	b. fila superior	00	00	00	00	01	01	01	01
	b. fila inferior	00	01	10	11	00	01	10	11
	decimal combinado	0	1	2	3	4	5	6	7
codigo caracter		32	108	123	98	124	225	255	254

codigo intermedio	b. fila superior	10	10	10	10	11	11	11	11
	b. fila inferior	00	01	10	11	00	01	10	11
	decimal combinado	8	9	10	11	12	13	14	15
codigo caracter		126	127	97	252	226	251	236	160

SCROLL MEDIANTE DESLIZAMIENTO DE BITS

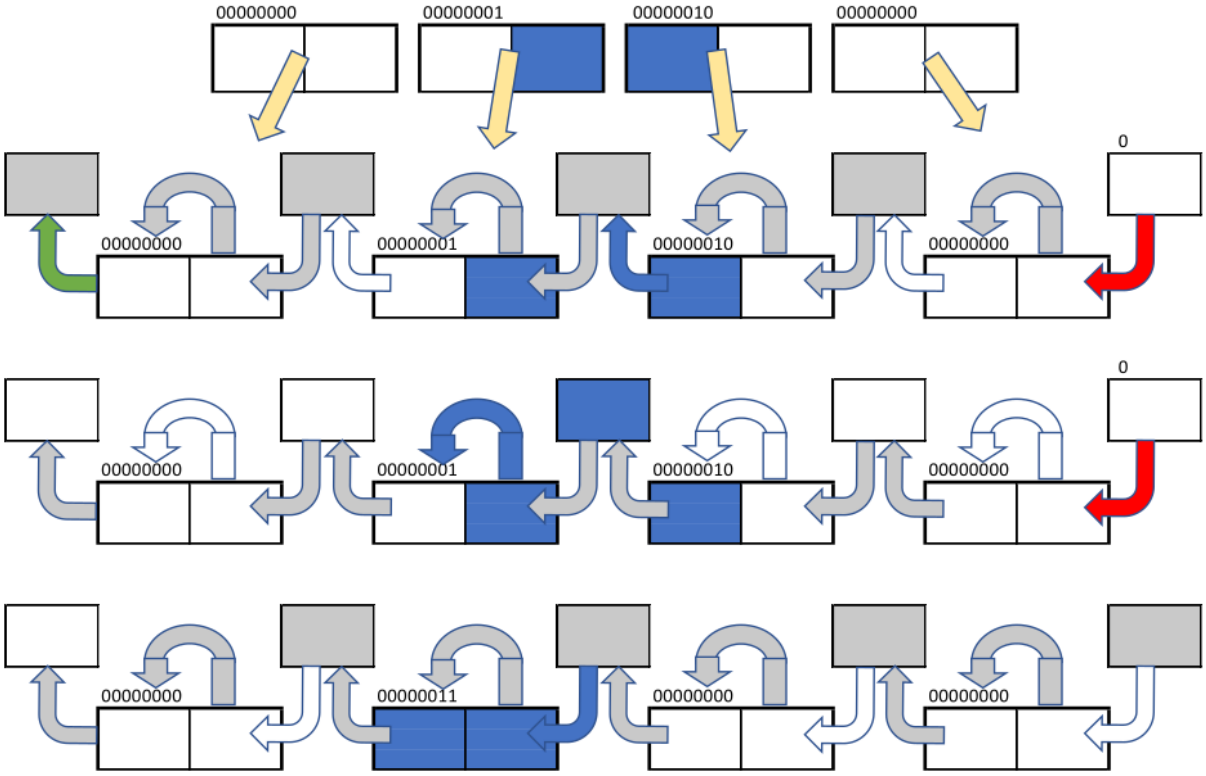
La figura siguiente representa el típico scroll de derecha a izquierda, cuando cada byte representa un bloque horizontal de 8 pixels. El proceso lo realiza prácticamente solo la instrucción ROL. El bit de acarreo posibilita el paso del bit que sale del byte anterior y entra en el byte posterior. En términos de pixels, el pixel que pasa de un bloque de 8 pixels al bloque siguiente, de derecha a izquierda.

El bucle para desplazar un grupo de bytes representando una línea horizontal de pixels sería algo así:



Pero no hay que confundirse, no hay ningún modo gráfico que almacene los pixels en memoria de esta manera en los Commodore. Sólo se explica para introducir el mecanismo de códigos intermedios utilizado para el scroll de megapixels.

Con el mecanismo de códigos intermedios cada byte representa 2 pixels. La figura siguiente representa el mecanismo de scroll de derecha a izquierda en esas condiciones. Como se ve sigue siendo necesario utilizar bits de acarreo pero ahora cada 2 bits, no cada 8 como proporciona de forma natural el procesador.



Es simplemente multiplicar por 4 en número de deslizamientos que se debían hacer cuando el byte representaba 8 pixels, y añadir a la instrucción ROL otra que detecte el acarreo en el bit 2 desde el bit 1 en vez de en el bit 8 (la bandera C) desde el bit 7. Esto se consigue con un CMP #4. En realidad no se complica demasiado.

BASIC vs LENGUAJE MAQUINA

El programa está en una mezcla de BASIC y lenguaje máquina (LM) para seguir una tradición que era bastante habitual en la época por dos razones:

- Permitía la distribución del programa en revistas en forma de listados a teclear, en una época en que la distribución de publicaciones en papel era la estándar.
- El BASIC facilitaba las pequeñas modificaciones desde el intérprete y la detección de errores, mucho más difícil en lenguaje máquina, en las condiciones de la época.

Algunos profesionales sí utilizaban emuladores de 6502 en ordenadores superiores en los que podían hacer una detección de errores en LM como la que se hace ahora, pero para la mayoría de la gente escribir en LM significaba prueba y error, que conducía a tener que reiniciar la máquina con pocos o ningún dato del fallo.

Otra cuestión que se puede plantear es porqué hablo de lenguaje máquina y no de ensamblador. En primer lugar porque en la época, al menos para los Commodore y el 6502, lo habitual eran los monitores de lenguaje máquina y no los ensambladores.

Los monitores no trabajaban con símbolos, o sea que había que introducir las variables y las etiquetas por sus direcciones de memoria. No había directivas, ni macros, ni ejecución

paso a paso. A duras penas calculaban los saltos relativos y traducían entre mnemónico y opcode, lo que no es gran cosa cuando no hay ni 200.

Pero esto no es un documento introductorio ni de BASIC ni de LM. Para introducirse al BASIC lo más recomendable sigue siendo el manual original del ordenador, porque introduce poco a poco los elementos del mapa de memoria que son necesarios para hacer cualquier cosa con gráficos en un Commodore.

A pesar de que comparten el mismo BASIC y el mismo lenguaje máquina y que en algunas ocasiones, como en este caso, los gráficos funcionan de una manera análoga, hacer que un programa, aunque sea corto como éste, funcione a la vez en el VIC20 con y sin ampliación, y comparta la mayor parte del listado con la versión de PET, con la memoria básica de 8kb, no es trivial.

DESCRIPCION DEL PROGRAMA BASIC (VIC20)

En este apartado se describe la funcionalidad de cada una de las líneas del programa BASIC en la versión de VIC20. Recordar que en este lenguaje se numeraban las líneas, y esto servía para definir el orden de ejecución. Empieza en la línea 100.

```
100 IFA=1THEN150
110 IFA=2THEN230
```

Por una parte en CBM BASIC no hace falta inicializar el valor de las variables con lo que al ejecutar el programa el valor de `A` es `0` y pasa directamente a la línea `120`.

Por otra parte, un quirk del CBM BASIC hace que cada vez que se utiliza LOAD para cargar datos en memoria desde la cinta o el disco, el programa se empieza a ejecutar desde el principio.

La línea `100` hace que después de ejecutar la línea `150` el programa prosiga por la línea `160`. La línea `110` hace lo mismo para las líneas `220`/`230`.

```
120 B=23:IFPEEK(44)=18THENB=25
130 POKE51,64:POKE52,B:POKE55,64:POKE56,B:POKE643,64:POKE644,B:CLR
```

La línea `120` lo que hace es comprobar si el VIC20 tiene ampliación de memoria o no. La dirección de memoria `44` contiene el byte alto de la dirección inicial de la memoria BASIC y se establece al arrancar en `18` si el VIC20 arranca con ampliación de +8kb o más y en `16` si arranca sin ampliación de memoria.

La variable `B` se utiliza en la línea `130` para establecer el tope de la memoria disponible para BASIC (la dirección superior de la memoria libre). El tope se baja para dejar espacio de memoria a las rutinas de LM.

```
140 A=1:C=PEEK(184):LOAD"MC",8,1
```

La línea `140` carga en memoria las rutinas en LM. Una vez en memoria a ese código se suele llamar código máquina (CM). La diferencia es sutil o quizá no haya pero bueno, da la idea de que una vez en memoria, al menos en la época, era un binario al que en general ya no se iba a poder acceder más. Esto es diferente en un emulador.

La dirección de memoria `184` a la que se hace referencia contiene el número de la unidad de almacenamiento que se utilizó en el último acceso (normalmente será la carga de éste mismo programa en BASIC). Si el programa se cargó desde cassette es 1, y si se cargó desde una unidad de diskette, 8 o 9.

Una particularidad del emulador VICE para VIC20, probablemente un bug histórico, es que cuando se utiliza el "filesystem" de windows como unidad de disco (unidad 8), esta dirección de memoria `184` no se actualiza a `8` sino a `0`.

Por tanto, si se utiliza el emulador VICE, en VIC20, utilizando el "filesystem" como unidad de disco, hay que hacer `POKE184,8`, antes de ejecutar `RUN`, para que arranque satisfactoriamente.

```
150 IFPEEK(44)<>18THEN240
160 POKE6498,0:POKE6499,48
```

La línea `150` comprueba, de la misma manera que lo hacía la línea `120`, si hay ampliación de memoria disponible. Si la hay se intenta cargar de disco un fichero de nombre `CHARSET` que debe contener datos de definición de un set de caracteres.

Si no hay ampliación o no existe el fichero, se utilizarán los datos del set de caracteres del VIC20 en ROM, del juego de caracteres mayúsculas y minúsculas. Para utilizar el juego de caracteres mayúsculas y gráficos de la ROM, entre las líneas 240 y 290, hay que insertar una línea `POKE6497,128`.

La alternativa es retocar el fichero MC que es más complicado. Respecto al fichero con el "set" de caracteres, se ha incluído el set de caracteres del C64 (mayúsculas y minúsculas).

La línea `160` traslada la zona de la memoria donde se realizan los cálculos intermedios para los VIC20 con ampliación (para almacenar los códigos intermedios y los códigos de los caracteres de las líneas básicas). Esto permite mantener las direcciones por defecto del inicio del programa BASIC (que cambia al conectar la ampliación), y permite situar la memoria de pantalla en una misma dirección, haya o no ampliación (lo que facilita la programación).

```
170 IFC=1THEN240
180 OPEN15,8,15,"R0:CHARSET=CHARSET"
190 INPUT#15,E,E$,T,S
200 CLOSE15
210 IFE=62THEN240
220 A=2:LOAD"CHARSET",8,1
230 POKE6496,0:POKE6497,32
```

La línea `170` comprueba si se ha cargado desde cassette, en ese caso también se utilizan los datos de la ROM. Se puede modificar para que cargue un juego de caracteres, saltando

a 220 en vez de a 240. Pero en este caso el usuario debe asegurarse de colocar el binario en la cinta después del programa MC, y probablemente de utilizar un fastloader de casette.

Las líneas 180-200 comprueban la existencia del fichero CHARSET. La línea 220 lo carga en memoria. La línea 230 modifica los 2 bytes que indican dónde se ha cargado el juego de caracteres, por defecto apunta a la ROM de caracteres. La línea 210 esquiva la carga si no existe (y así evita el error FILE NOT FOUND que terminaría la ejecución del programa).

```
240 SYS7034
250 A$="Lorem ipsum dolor sit amet, consectetur adipiscing elit "
290 I=1:GOTO320
```

Las líneas 240, 250, 290 marcan el inicio del programa. La línea 240 hace las inicializaciones correspondientes en LM en la dirección 7034 (7022 en el PET) correspondiente a la rutina initpet. La línea 250 es el string A$ con el texto que se quiere visualizar como banner.

La línea 290 establece el valor inicial de la variable I, que representa el índice del carácter (dentro de A$) que está entrando se está visualizando en la pantalla por el margen derecho. También establece el punto de entrada al bucle en la primera iteración, que es la línea 320.

Las líneas de BASIC pueden tener 88 caracteres en VIC20 y 80 caracteres en el resto de ordenadores Commodore 8 bit. Pero los strings pueden tener algo menos de 256 caracteres. Es perfectamente legítimo concatenar un mensaje más largo, en este caso 232 caracteres, de la siguiente manera:

```
250 A$="Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
do eiusmod"
260 A$=A$+" tempor incididunt ut labore et dolore magna aliqua. Ut
enim ad "
270 A$=A$+"minim veniam, quis nostrud exercitation ullamco laboris
nisi ut "
280 A$=A$+"aliquip ex ea commodo consequat. "
```

A partir de la línea 300 hasta el final se encuentra el bucle que va visualizando el banner de manera indefinida En las iteraciones subsiguientes,el bucle puede empezar en la línea 300 o en la línea 430.

```
300 I=I+1:IFI>LEN(A$)THENI=1
```

La bucle una vez ya ha arrancado

```
320 C=ASC(MID$(A$,I,1))
330 IFC=255THENC=94:GOTO400
340 IFC>191THENC=C-128:GOTO400
350 IFC>159THENC=C-64:GOTO400
360 IFC>127THENC=C+64:GOTO400
370 IFC>95THENC=C-32:GOTO400
380 IFC>63THENC=C-64:GOTO400
390 IFC<32THENC=C+128
```

```
400 POKE1,C
```

Cuando ha terminado de incorporarse un carácter por el margen derecho y entra otro nuevo, el bucle empieza en la línea `300`. Las líneas `300` y `320` van avanzando caracteres desde el string `A$`, de izquierda a derecha y vuelta a empezar.

Las líneas `330` a `390` convierten el código ASCII/PETSCII del carácter al código correspondiente para la memoria de pantalla.

```
410 SYS6660:GOTO500
430 SYS6726
440 IF PEEK(0)<>0THEN300
```

Hay dos puntos de entrada a LM para la actualización de los gráficos: la dirección `6655` a la rutina `loadnewchar` para la visualización de la primera columna de pixels (columna izquierda) del símbolo gráfico del carácter, que se invoca en la línea `410`.

Y para las otras 7 columnas del carácter la entrada a LM se produce en la dirección `6719` desde la línea `430` en el punto de entrada `nexpixchr`.

Hay dos direcciones de memoria que se están utilizando para coordinar el flujo de caracteres entre el BASIC y el LM. La dirección `1`, que se actualiza en la línea `400`, el BASIC indica al LM el código del carácter que debe actualizarse. La dirección `0`, que se comprueba en la línea `440`, el LM comunica al BASIC que necesita un carácter nuevo *(1)*.

```
500 GETB$:IFB$=""THEN430
510 IFB$="Q"THENSYS7133:STOP
520 B=ASC(B$):IFB<49THEN430
530 IFB<53THENPOKE253,B-48:GOTO430
540 IFB<65THEN430
550 IFB<69THENPOKE254,B-65:GOTO430
560 GOTO430
```

La línea `500` comprueba si se está pulsando una tecla: esto permite cambiar el tamaño/velocidad del texto del banner, y salir.

La línea `510` termina la ejecución del programa si se pulsa la tecla *Q*. La dirección `7133` hace la finalización del LM.

Las líneas `520` y `530` comprueban la pulsación de los números *1*,*2*,*3*,*4* que definen el ancho del carácter y la velocidad de visualización.

Las líneas `520` y `530` comprueban la pulsación de las letras *A*,*B*,*C*,*D*,*E* que definen el alto del carácter.

DESCRIPCION DEL PROGRAMA BASIC (CBM PET 40 COL)

En este apartado se describe la funcionalidad de cada una de las líneas del programa BASIC en la versión de PET. Hay que remarcar las similitudes entre el VIC20 y el PET, tanto el BASIC como el LM son básicamente idénticos, así como los códigos de los caracteres tanto en los string (ASCII) como en la memoria de pantalla que es de tipo alfanumérico en los dos casos.

Las diferencias, al margen de los valores concretos de las direcciones de memoria:

Ancho de pantalla.

> En el PET clásico es de 40 columnas, en el VIC20 de 25 columnas, esto hace algo más legible el scroll en el PET que en el VIC20, pero desde el punto de vista de afectación al programa es sólo un parámetro y afecta pocos cambios.
>
> En ambos casos la pantalla usa 25 filas.

Dirección de memoria de pantalla.

> Aunque hay PET con diferentes configuraciones de memoria (8kb, 16kb, 32kb) la dirección de la memoria de pantalla no cambia, es siempre 32768. Esto simplifica algo la parte en BASIC del programa para el PET.
>
> En el VIC20 puede estar en la dirección 4096 (por defecto en sistemas +8kb o más), en 7168 (por defecto en sistemas sin expandir o con expansión de 3kb) o en los valores intermedios de 4608, 5120, 5632, 6144, 6656 y 7168.

Dirección de la ROM de los caracteres.

> Los datos que definen los símbolos gráficos de los caracteres están en ROM. Pero mientras que en el VIC20 esta ROM está visible para la CPU, en el PET no. Esto hace que en la versión del PET sea necesario sí o sí cargar la definición de los caracteres desde un fichero.
>
> Se ha modificado un poco el programa para que la carga funcione desde cassette o desde disco, porque en el PET puede ser algo más difícil hacerse con una unidad de disco o el correspondiente sustituto basado en tarjetas SD.
>
> Por otra parte el PET siempre tiene 8kb o más y siempre tiene suficiente memoria para cargar esos datos. En el VIC20 con 5kb no, pero como se ha explicado no es tan relevante porque se pueden sacar de la ROM los datos por defecto.

Otras diferencias

> Las otras diferencias entre el VIC20 y el PET relacionadas con las anteriores que se introducen por dar algo de contexto pero que no influyen en este programa en concreto son:

En el VIC20 está el hecho de que tiene memoria de color, pero en este caso simplemente se inicializa al mismo valor para todos los caracteres.

Los símbolos gráficos de los caracteres que vienen por defecto en la ROM se pueden redefinir de otra manera en la RAM. Es lo que permite que, por ejemplo, se pueden definir sprites por software, aunque no sea demasiado cómodo.

Pero en este programa esta característica no se utiliza, porque cada pixel se representa por un cuadrado de 4x4 pixels o un múltiplo. Puede ser algo confuso a nivel conceptual porque cuando se habla del juego de caracteres en el VIC20 se hace pensando en redefinirlo para poder hacer algún juego.

El listado empieza en la línea 100.

```
100 IFA=1THEN150
110 IFA=2THEN230
```

Ver explicación de lo que hacen las líneas `100`, `110` en la versión del fichero para VIC20

Recordemos que el PETSCII tiene dos conjuntos de caracteres, unos con letras mayúsculas y minúsculas y otro que no tiene las letras minúsculas y en su lugar tiene más símbolos gráficos.

La dirección de memoria `59468` a la que se accede en la línea `120` lo que hace es seleccionar el conjunto de caracteres que tiene letras mayúsculas y minúsculas. Para seleccionar el conjuinto de letras mayúsculas y caracteres gráficos hay que retirar ese POKE.

En todo caso no afecta al texto ampliado, sólo al texto pequeño que se representa en la línea inferior de la pantalla. Como se expuso en el Apartado “Dirección de la ROM de los caracteres” la CPU del PET no puede acceder a la ROM de caracteres y para generar el texto ampliado hay que cargar dichos datos en RAM, desde un fichero.

```
120 POKE59468,14:B=17
130 POKE48,0:POKE49,b:POKE52,0:POKE53,B:CLR
140 A=1:C=PEEK(212):LOAD"MC",C,1
150 IFC=1THEN220
```

Como en la versión de VIC20, la variable `B` se utiliza en la línea `130` para establecer el tope de la memoria disponible para BASIC (la dirección superior de la memoria libre). El tope se baja para dejar espacio de memoria a las rutinas de LM. Se asume que el PET tiene 8kb y si tiene más memoria simplemente no se utiliza.

La línea `140` carga en memoria las rutinas en LM. La dirección de memoria `212` a la que se hace referencia contiene el número de la unidad de almacenamiento que se utilizó en el último acceso (que normalmente será éste mismo programa en BASIC). Si el programa se cargó desde cassette es `1`, y si se cargó desde una unidad de diskette, `8` o `9`. El PET tiene unidades de diskette con uno o dos mecanismos. No me he preocupado de comprobar unidades con dos mecanismos.

En el emulador VICE para VIC20, cuando se utiliza el “filesystem” de *Windows* como unidad de disco (unidad 8), la dirección de memoria análoga a la `212` que es la dirección `184` no se actualiza a `8`, sino a `0`. Este “bug” no afecta al emulador VICE para PET.

En el caso de utilizar *cbmprg* como ensamblador, a la hora de ensamblar, hay que hacerlo por separado para las partes del programa en BASIC y en código máquina. Así crea por separado los ficheros correspondientes a las partes en BASIC y en LM. En general cuando se utilizan a la vez BASIC y LM no se puede ensamblar todo de una vez, como cuando se trata de un programa en LM, con una sola instrucción `sys` al principio.

```
170 OPEN15,8,15,"R0:CHARSET=CHARSET"
180 INPUT#15,E,E$,T,S
190 CLOSE15
210 IFE=62THENSTOP
220 A=2:LOAD"CHARSET",C,1
```

Las líneas `170` a `190` comprueban si el fichero existe en el disco, si se cargó desde cassette esta comprobación se evita en la línea `160`, que se salta directamente a la línea `220`.

En la línea `200`, si se determinó la no existencia del fichero en el disco, el programa se detiene. En la versión de VIC20 por defecto se podía tomar la dirección en ROM del juego de caracteres, pero en la versión de PET, como se ha indicado, no es posible.

Si se utiliza el cassette la instrucción LOAD en la línea `140` espera encontrar el fichero, o bien grabado después del programa, o bien que se cambie la cinta por otra en la que esté grabado. Si no se suministra el fichero la ejecución del programa no proseguirá a esa instrucción.

A partir de la línea `220` todas las líneas son iguales a las de la versión para VIC excepto las 4 direcciones que se utilizan para intercambiar datos entre la parte en BASIC y la parte en LM. En vez de las direcciones `0`, `1`, `253`, `254` se utilizan las direcciones `240`, `241`, `246`, `247`.

La línea 510 también cambia, no hay ninguna llamada al sistema antes de acabar porque no hay que restablecer parámetros de vídeo.

```
230 SYS7022
250 A$="Lorem ipsum dolor sit amet, consectetur adipiscing elit"
290 I=1:GOTO320
300 I=I+1:IFI>LEN(A$)THENI=1
320 C=ASC(MID$(A$,I,1))
330 IFC=255THENC=94:GOTO400
340 IFC>191THENC=C-128:GOTO400
350 IFC>159THENC=C-64:GOTO400
360 IFC>127THENC=C+64:GOTO400
370 IFC>95THENC=C-32:GOTO400
380 IFC>63THENC=C-64:GOTO400
390 IFC<32THENC=C+128
```

```
400 POKE241,C
410 SYS6655:GOTO500
430 SYS6719
440 IFPEEK(240)<>0THEN300
500 GETB$:IFB$=""THEN430
510 IFB$="q"THENSTOP
520 B=ASC(B$):IFB<49THEN430
530 IFB<53THENPOKE246,B-48:GOTO430
540 IFB<65THEN430
550 IFB<69THENPOKE247,B-65:GOTO430
560 GOTO430
```

SINCRONIZACION DE LA VISUALIZACION CON EL RASTER

En el VIC20 es fácil porque el registro 4 del chip de vídeo VIC, en la dirección de memoria $9004 (en el listado representada por `vic+4`), contiene el número de la línea raster que se está visualizando.

El bucle del programa es lo suficientemente rápido para que quede tiempo libre para un bucle de espera a una línea de raster concreta, en este sistema el ritmo de visualización está sincronizado con el CRT, a 50 frames por segundo (PAL) o 60 frames por segundo (NTSC).

En este caso el número de la línea se almacena en la variable `datvid`. Se puede ajustar al inicio del borde inferior de la pantalla o un poco antes, para maximizar el tiempo en el que se puede modificar sin ningún tipo de precaución la memoria de pantalla.

En tiempo en que la pantalla esté visualizando contenidos de la memoria de pantalla hay que sincronizar con el movimiento del raster los cambios en la memoria (sólo modificar la parte inferior). Es lo que se conoce como “correr delante del raster”.

En el PET sólo se puede conocer el “retrace”, para entonces ya se ha dejado atrás la visualización del marco inferior y sólo queda el marco superior para poder escribir en la memoria de pantalla sin ninguna interferencia posible con la visualización.

Una posibilidad para sincronizar la ejecución del programa con una línea concreta de raster en el PET es utilizar los TIMER del 6522 VIA. Al final del documento se muestra una modificación de este listado que funciona con los PET con BASIC 2 o BASIC 4.

Abajo se muestra el bucle de espera de sincronización del listado ensamblador.

```
ifdef TGT_VIC20
        lda datvid
@waxca1 cmp vic+4
        bcc @waxca1
endif
ifdef TGT_PETBV2
@waitra lda $E840
        and #$20
        bne @waitra
endif
```

DESCRIPCION DEL PROGRAMA EN LENGUAJE MAQUINA

DECLARACIONES DE VARIABLES Y CONSTANTES

Se utiliza un mismo listado para ambas versiones, pero se incluyen unas directivas para incluir o excluir unas pocas diferencias al ensamblar el programa con el VIC20 o con el PET como “target”.

La dirección de carga es la misma para ambos listados, 6656, eso simplifica algo la elaboración de las dos versiones diferentes para ambas máquinas.

La manera de sincronizar el refresco de la pantalla con el retorno del raster es la diferencia más importante entre ambas versiones.

La directiva TARGET la utiliza el ensamblador de *cbmprg* para saber qué bloques de código procesar si hay varias posibilidades. En otros ensambladores se puede utilizar #define, la parte del programa en ensamblador se va a procesar igual. Lo que no hacen muchos otros ensambladores es procesar la parte del programa en BASIC, quizá sólo *C64Studio*.

```
target TGT_VIC20
target TGT_PETBV2
```

En esta parte del programa se definen las constantes, variables y direcciones de memoria. Se da una corta explicación de su utilidad, pero lo normal es tener que referirse a la parte del código que las utiliza para entender el funcionamiento o el concepto.

Se empieza con la definición de los valores de las constantes y direcciones de memoria. De la dirección de la memoria de pantalla ya se ha hablado, va a la constante `mempan`.

```
; CONSTANTES
conbit=8       ; 8 pixels por byte
ifdef TGT_PETBV2
stride=40  ; ancho pantalla
mempan = $8000  ;
copiacar=$1FF0 ;
endif
ifdef TGT_VIC20
stride=25  ; ancho pantalla
mempan = $1C00 ;
vic=$9000
datvid=127 ; 254
```

```
copiacar=$1FF0 ;
endif

stride1=stride-1  ; ancho pantalla
```

Se sigue con las variables que tienen que ir en localizaciones de memoria que no interfieren con el intérprete BASIC.

Podrían estar en el bloque de memoria libre que se reservó para el CM, y de hecho así se ha hecho con los bloques de variables más grandes (códigos de caracteres). Pero otras variables es mejor que estén en direcciones más bajas que son más cortas. Por ejemplo, para hacer más corto e inteligible el listado BASIC.

En el caso de los punteros, éstos sólo pueden estar en la página cero (direcciones 0 a 255).

```
; PET seguras $A2, $ED a $F7, $FF not used
; PET quiza
;     temporales $54 - $5D, $5E - $63, $66 - $6B
; VIC seguras    $FB - $FE, $00 - $06
;     temporales $57 - $60, $61 - $66, $69 - $6E
```

Declara las siguientes variables:

`conrephor, conrepver`: número de veces que se repite el pixel en vertical y en horizontal

```
; CONTROL
; conrephor         1    2    3    4
; columnas x char   4    8   12   16
; conrepver         1    2    3    4
; filas x char      8   12   16   20

ifdef TGT_PETBV2
conrephor=$F6       ; el pixel horizontal se repetira 1, 2, 3 o 4
veces
conrepver=$F7
endif
ifdef TGT_VIC20
conrephor=$FD ; puntero al grafico
conrepver=$FE
endif
```

Declara las siguientes variables:

`banfincha`: bandera que indica a la parte BASIC del programa que la parte en ensamblador requiere otro carácter

`copachar`: código del carácter actual, entre 0 y 255.

```
ifdef TGT_PETBV2
banfincha=$F0
```

```
copachar=$F1        ; codigo pantalla caracter
endif
ifdef TGT_VIC20
banfincha=$00
copachar=$01        ; codigo pantalla caracter
endif
```

Declara las siguientes variables:

`indconbit`: contador indica cuál de las 8 columnas de pixels del carácter actual se está visualizando (entrando por el margen derecho)

`indconrephor`: contador indica cuantas veces se ha repetido ya la misma columna de pixels. Itera desde `0` hasta `conrephor-1`.

```
; ESTADO
ifdef TGT_PETBV2
; persistente
indconbit=$F2
indconrephor=$F3
endif
ifdef TGT_VIC20
; persistente
indconbit=$02
indconrephor=$03
endif
```

Declara los siguientes punteros (los punteros tienen que estar en la página cero, son direcciones de un solo byte):

`pungra`: puntero al gráfico del símbolo

`punpix1`, `punpix2`: puntero a las tablas de datos intermedios: `punpix1` para las pares, `punpix2` para las impares

`puncha2`, `puncha3`: puntero a las tablas de caracteres procesados, para diferentes alturas de banner: `puncha2` para cuadraditos de 2 “megapixels”, `puncha3` para cuadraditos de 1 megapixel

`punscrn1`: puntero a la memoria de pantalla

```
; PUNTEROS
ifdef TGT_PETBV2
; persistente
pungra=$F4 ; puntero al grafico
; temporales
punpix2=$55
puncha2=$57
```

```
punpix1=$59
; punteros
punscrn1=$5B
endif
ifdef TGT_VIC20
; persistente
pungra=$FB ; puntero al grafico
; temporales
punpix2=$58
puncha2=$5A
punpix1=$5C
; punteros
punscrn1=$5E
endif

puncha3=puncha2
```

Declara las siguientes variables (locales, el contenido se puede perder entre llamada y llamada al CM desde el BASIC). Para bucles, etc.:

`conbyte`: contador del byte del carácter (al iterar los 8 bytes que definen el gráfico del símbolo)

`indfila`, `confilas`: para los factores verticales posibles, indica dónde buscar las combinaciones de las 12 filas “básicas”.

```
; LOCALES
ifdef TGT_PETBV2
conbyte=$5D
indfila=$5D
confilas=$5E
endif
ifdef TGT_VIC20
conbyte=$60
indfila=$60
confilas=$61
endif
```

Calcula las direcciones de memoria de pantalla de cada una de las filas. Se podrían calcular sobre la marcha pero es más fácil calcular las constantes aquí y ponerlas en una tabla.

```
mempan0=mempan
mempan1=mempan0+stride
mempan2=mempan1+stride
mempan3=mempan2+stride
mempan4=mempan3+stride
mempan5=mempan4+stride
mempan6=mempan5+stride
```

```
mempan7=mempan6+stride
mempan8=mempan7+stride
mempan9=mempan8+stride
mempan10=mempan9+stride
mempan11=mempan10+stride
mempan12=mempan11+stride
mempan13=mempan12+stride
mempan14=mempan13+stride
mempan15=mempan14+stride
mempan16=mempan15+stride
mempan17=mempan16+stride
mempan18=mempan17+stride
mempan19=mempan18+stride
mempan20=mempan19+stride
mempan21=mempan20+stride
mempan22=mempan21+stride
mempan23=mempan22+stride
mempan24=mempan23+stride
```

ENTRADA AL PROGRAMA

El programa en código máquina se carga siempre a partir de esta dirección.
Los 4 primeros bytes contienen dos direcciones a la memoria:

`bacharset` apunta la tabla donde se almacena el juego de caracteres: en el VIC20, en ROM, en este caso $8800 para el juego de caracteres mayusculas/minusculas. La dirección $8000 corresponde al juego de caracteres mayusculas/caracteres graficos. Si se carga el juego de caracteres en RAM el programa BASIC cambia este puntero. En el PET siempre en RAM, porque la ROM de caracteres no es accesible.

Concretamente, `bacharset` es la dirección base de los datos del juego de caracteres (correspondiente al primer byte del carácter de código 0)

Al espacio `adfidain`, donde se almacenan los valores de las 8 filas de códigos intermedios y los valores de las 12 filas de caracteres básicos.

En este listado la rutina `initpet` inicializa este espacio a 0 y a la vez va calculando los punteros dentro de este espacio a cada una de las 8 filas de códigos intermedios y 12 filas de caracteres básicos. Esta manera de hacerlo proporciona flexibilidad para cargar el programa en distintas direcciones de memoria y crear las tablas en distintas direcciones de memoria también, pero finalmente quedaron en direcciones fijas, así que tendría más sentido definirlas ya antes de ensamblar como se presenta al final.

```
*=$1960 ; 6496

ifdef TGT_PETBV2
bacharset word $2000
adfidain word $1780 ; 6658 - 6659
```

```
endif
ifdef TGT_VIC20
bacharset word $8800 ; 6656 - 6657
adfidain word $1780 ; 6658 - 6659
endif
```

Tablas `tabcodcarme` y `tabcodcarcu`, convierten los códigos intermedios a los códigos PETSCII de las imágenes

```
;
; TABLAS

; CODIGOS PESCI
;                00   01   10   11
tabcodcarme byte 32, 225,  97, 160

;          00   00   00   00   01   01   01   01   10   10
;          00   01   10   11   00   01   10   11   00   01
;        0000 0001 0010 0011 0100 0101 0110 0111 1000 1001

;          10   10   11   11   11   11
;          10   11   00   01   10   11
;        1010 1011 1100 1101 1110 1111

tabcodcarcu
    byte  32, 108, 123,  98, 124, 225, 255, 254, 126, 127
    byte  97, 252, 226, 251, 236, 160
```

Siguiente, tablas de punteros. Contienen las direcciones calculadas por `initvic` / `initpet`. La razón de no incluir las tablas ya precalculadas es que dependen de dónde se sitúe el espacio para resultados intermedios, y éste se sitúa en una u otra dirección, en función de si hay disponible ampliación de memoria o no. En el caso del PET sí que se podrían incluir y al final pongo un esquema alternativo en el apartado de posibles simplificaciones.

```
; punteros filas datos intermedios
tapufidainlo bytes 8 ; de 1700 a 17C7

; punteros filas datos caracteres gráficos rectangulitos
tapuficamelo bytes 8 ; de 17C8 a 188F

; punteros filas datos caracteres graficos cuadritos
tapuficaculo bytes 4 ; de 1890 a 18F3

; punteros filas datos intermedios
tapufidainhi bytes 8

; punteros filas de caracteres gráficos rectangulitos
tapuficamehi bytes 8
```

```
; punteros filas de caracteres graficos cuadritos
tapuficacuhi bytes 4

; esto son 20*stride bytes 800 en PET y 500 en VIC parte pordira
estar en el bufer caset?
; o al menos fuera de lo que carga
```

Definición de las siguientes tablas y variables:

`tabconfilas`: alturas totales en filas del banner, correspondientes a los factores de ampliación vertical del texto de 2, 3, 4 y 5 megapixels por pixel.

`tabindfilas`: índices a la tabla `tabindfilacha`, correspondientes a los 4 factores de ampliación vertical del texto.

`tabfilainic`: fila de la pantalla en la que empieza la visualización del banner. Cuanto más altas son las letras, más cerca está del borde superior de la pantalla y menor es el índice de fila, que se incrementa hacia abajo.

`tabindfilacha`: tabla que contiene las 4 secuencias de índices de filas básicas, correspondientes a los 4 factores de ampliación vertical del texto. La tabla contiene 56 índices agrupados de 4 secuencias de 8, 12, 16 y 20 índices.

Las filas básicas precalculadas son 12: 8 corresponden al factor de ampliación vertical de 2 megapixels por pixel (o sea 1 fila completa por pixel) y 4 corresponden al factor de ampliación vertical de 1 megapixel por pixel.

```
; visualizacion

; x 0 filas 21-24, bloque 8,9,10,11
; x 1 filas  6-13, bloque 0,1,2,3,4,5,6,7
; x 2 filas  4-15, bloque 0,8,1,2,9,3,4,10,5,6,11,7
; x 3 filas  2-17, bloque 0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7
; x 4 filas  0-19, bloque 0,0,8,1,1,2,2,9,3,3,4,4,10,5,5,6,6,11,7,7

tabconfilas byte 8, 12, 16, 20
tabindfilas byte 0, 8, 20, 36
tabfilainic byte 6, 4, 2, 0

tabindfilacha byte 0,1,2,3,4,5,6,7, 0,8,1,2,9,3,4,10,5,6,11,7,
0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7,
0,0,8,1,1,2,2,9,3,3,4,4,10,5,5,6,6,11,7,7

loadnewchar
```

Inicializa las siguientes variables:

Calcula `pungra`: puntero a los datos del gráfico, `pungra = bacharset + copachar * 8`

Inicializa `indconbit` a 8: contador de las 8 columnas de pixels del carácter.

Inicializa `banfincha` a 0: indica que se han visualizado las 8 columnas de pixels y se requiere que la parte en BASIC actualice el código de `copachar`.

Inicializa los siguientes bloques de memoria:

`copiacar`: espacio de 8 bytes correspondiente la definición gráfica del carácter actual

`mempan24`: memoria de pantalla de la última línea, donde se representa el texto del banner sin ampliar, para conseguir visualizar un desplazamiento a la izquierda se desplazan cada uno de los valores a la dirección anterior

El punto de entrada `loadnewchar` en la dirección 6655 es el que se encarga de empezar la visualización de un nuevo carácter y proseguir metiendo una columna de pixels por el margen derecho.

```
; CODIGO

loadnewchar ; load new char pointer, init bit counter
; calcular puntero al grafico del caracter, inicializar contador
bits
calcpunchar
        lda copachar
        asl
        rol pungra+1
        asl
        rol pungra+1
        asl
        rol pungra+1
        clc
        adc bacharset
        sta pungra
        lda pungra+1
        and #7
        adc bacharset+1
        sta pungra+1

        ldy #0
        sty banfincha
@copcar
        lda (pungra),y
        sta copiacar,y
        iny
        cpy #8                  ; copiar 8 bytes
        bne @copcar
        sty indconbit           ; restablecer copiar 8 bits
```

```
        ldy #0
@scrola lda mempan24+1,y
        sta mempan24,y
        iny
        cpy #stride1
        bne @scrola
        lda copachar
        sta mempan24+stride1

        ldx #0
        bpl calcodchar          ; BRA
```

El punto de entrada `nexpixchr` en la direccion 6719 se encarga de actualizar el contador de columnas pendientes `indconbit`, y si en la llamada anterior se hizo la última columna, sale activando la bandera de señalar que se requiere un código de carácter nuevo. En caso contrario sigue y mete otra columna de pixels por el margen derecho.

```
nexpixchr ; check next and get pixel column in char
; comprobar que la columna de pixels no fue la ultima del caracter
; el pixel horizontal se repetira 1,2,3 o 4 veces
; cuando toque desplayar los bytes del char se activa la bandera
banfinpix
; esta bandera se limpia aqui mismo
        ldx #0
        dec indconbit
        bne @fincha
; cuando haya hecho ultimo bit salir y requerir otro caracter
        inc banfincha
        rts
@fincha
@finpix
```

Refresco de la visualización: primero se sincroniza la ejecución con el raster del CRT.

```
; recuperar columna de pixels y pasarla al espacio intermedio
calcodchar
; aqui visualizar
ifdef TGT_VIC20
        lda datvid
@waxca1 cmp vic+4
        bcc @waxca1
endif
ifdef TGT_PETBV2
@waitra lda $E840
        and #$20
        bne @waitra
```

```
endif
```

COPIA CODIGOS CARACTER A MEMORIA DE PANTALLA

Restablece el puntero `punscrn1` al inicio de la memoria de pantalla.

```
        lda #<mempan
        sta punscrn1
        lda #>mempan
        sta punscrn1+1
```

Restablece el contador `confilas` a partir de la tabla `tabconfilas`, el índice a la secuencia de filas `indfila` a partir de la tabla `tabindfilas`, y carga en el registro X el índice de la fila superior del banner desde la tabla `tabfilainic`, antes de invocar `delrows` (rutina que borra las primeras filas formando un margen superior).

```
        ldy conrepver
; provisu
        lda tabconfilas,y       ; 8, 12, 16, 20
        sta confilas
        lda tabindfilas,y       ; 4, 12, 24, 40
        sta indfila

        ldx tabfilainic,y       ; 6, 4, 2, 0
        beq @txtext
; tercio superior
        jsr delrows
```

En un bucle exterior, carga en el registro Y el índice `indfila` para iterar la secuencia contenida en la tabla `tabindfilacha`, recuperando en el registro X los índices de filas precalculadadas. Para iterar en el bucle interior la secuencia de caracteres de cada fila se obtiene el puntero `puncha2` al espacio de memoria correspondiente, desde las tablas `tapuficamelo` y `tapuficamehi`.

```
; tercio letras
@txtext
        ldy indfila
        ldx tabindfilacha,y
        lda tapuficamelo,x
        sta puncha2
        lda tapuficamehi,x
        sta puncha2+1
```

El bucle interior copia la secuencia de códigos precalculados a la memoria de pantalla, de acuerdo con el ancho de pantalla `stride1`.

```
        ldy #stride1
@txtint
        lda (puncha2),y
        sta (punscrn1),y
        dey
        bpl @txtint
```

Al acabar el bucle interior se actualiza el puntero de la memoria de pantalla a la siguiente fila.

```
        lda #stride
        clc
        adc punscrn1
        sta punscrn1
        bcc @txtove          ; salta si CC comprobacion byte HI
        inc punscrn1+1
@txtove
```

Después se actualiza el puntero `indfila` a la siguiente fila de la secuencia y se comprueba el contador de filas para determinar el final del bucle exterior.

```
        inc indfila
        dec confilas
        bne @txtext
```

Acabada la representación del banner, se calcula el número de filas del margen inferior restando a la altura de pantalla el número de filas del banner y el número de filas del margen superior y se invoca `delrows` para borrar las últimas filas formando un margen inferior.

```
        lda #20
        ldy conrepver
        sec
        sbc tabconfilas,y       ; 8, 12, 16, 20
        sbc tabfilainic,y       ; 6, 4, 2, 0
        tax
        beq @finvis
; tercio inferior
        jsr delrows

@finvis
```

ACTUALIZACIÓN DEL SCROLL

ENTRADA DE PIXELS POR LA DERECHA

Aquí empieza el cálculo del scroll utilizando códigos intermedios. Cada código es un byte que representa 2 pixels en horizontal. Hay 8 filas de códigos intermedios, cada una corresponde a cada una de las 8 filas de pixels de la definición gráfica del símbolo.

En cada “frame” se realiza la entrada de una columna de pixels. El bucle exterior repite la entrada de 1 a 4 veces controlar el ancho del texto del banner. El número de veces a repetir se contiene en la variable `conrephor`. La cuenta de las veces que se ha repetido se lleva en `indconrephor`.

```
        ldy conrephor
@burege
        dey
        sty indconrephor
```

El bucle intermedio itera sobre los 8 pixels de la columna vertical, correspondientes a los bytes izquierdos de la secuencia de 8 bytes en `copiacar`. También restablece el puntero `punpix2` al primer byte de la fila de códigos intermedios correspondiente.

Además, en el caso de que el bucle exterior esté en la última iteración, se actualiza `copiacar` para que la próxima vez que se invoque esta rutina el pixel a la izquierda sea ya el siguiente. Esto se consigue comprobando si es cero el contenido de la variable `indconrephor`.

```
; aqui desplaza las filas con pixels de medio caracter
        ldy #7
@buexin sty conbyte
        lda tapufidainlo,y
        sta punpix2
        lda tapufidainhi,y
        sta punpix2+1
        lda copiacar,y
        asl
        ldx indconrephor
        bne @finpix
        sta copiacar,y
@finpix
```

SCROLL DE CODIGOS INTERMEDIOS

El bucle interior itera sobre la fila de códigos intermedios, cada código intermedio representando 2 megapixels. Se recorre la fila desde el margen derecho hacia el margen izquierdo.

En la interación inicial es la bandera C la que indica si entra o no un pixel por el margen derecho. En las iteraciones subsiguientes se va actualizando C señalando si para cada columna llega un pixel desde la columna a su derecha.

Cuando un byte representa 8 pixels la instrucción `rol` hace el desplazamiento de bits (scroll) de manera natural. Pero al representar sólo 2 hacen falta las instrucciones `cmp#4` y `sbc#4` para ajustar convenientemente los valores del byte y del acarreo.

```
        ldy #stride1
@buinin
        lda (punpix2),y
        rol
        cmp #4    ; 2 bits por byte
        bcc @nexbit
        sbc #4    ; quitar el bit 2 pero dejar CS
@nexbit sta (punpix2),y
        dey
        bpl @buinin
```

Las siguientes instrucciones cierran el bucle intermedio, `conbyte` indica la fila de códigos intermedios correspondiente a una de las 8 filas de 8 pixels del carácter.

```
        ldy conbyte
        dey
        bpl @buexin
```

CONVERSIÓN EN CARACTERES DE LOS CÓDIGOS INTERMEDIOS

Esta última parte convierte los códigos intermedios en los códigos de caracteres de las filas básicas-

Primero calcula las 8 filas básicas con pixels representados por cuadrados de 2 megapixels de altura (una fila de caracteres completa por cada fila de pixels horizontal del carácter).

El puntero `punpix2` apunta a la secuencia de códigos intermedios y el puntero `puncha2` a la secuencia de bytes de códigos de carácter que se está precalculando.

```
; procesar filas graficos
; aqui calcula caracteres de las 8 filas con pixels de medio
caracter
        ldx #8
@buexme
        stx conbyte
        lda tapufidainlo-1,x
        sta punpix2
```

```
        lda tapufidainhi-1,x
        sta punpix2+1
        lda tapuficamelo-1,x
        sta puncha2
        lda tapuficamehi-1,x
        sta puncha2+1

        ldy #stride1
@buinme
        lda (punpix2),y
        tax
        lda tabcodcarme,x
        sta (puncha2),y
        dey
        bpl @buinme
        ldx conbyte
        dex
        bne @buexme
```

Después calcula las 4 filas básicas con píxels representados por cuadrados de 1 megapixel de altura. Como cada fila de caracteres completa representa dos filas de horizontal de píxels del carácter hacen falta punteros a 2 secuencias de códigos intermedios, `punpix1` y `punpix2`.

```
; aqui calcula las 4 filas con pixels de un cuarto de caracter
        ldx #4
@buexcu
        stx conbyte
        txa
        asl
        tay
        jsr copipun1
```

La rutina `copipun1` lo único que hace es inicializar unos punteros para el siguiente bloque de copia, muy parecido a lo que hacen las 8 instrucciones después de `@buexme`. Pero se pusieron por separado para que todo este bloque de código del scroll sea corto y la instrucción del salto condicionado (branch) del bucle en la fila quede dentro del rango de -128/+127 bytes. Así no hay que recurrir a `jmp`, que introduce puede complicar la legibilidad del código.

Aquí es donde brilla la utilización de los códigos intermedios porque con dos desplazamientos `asl` de un bit del código, una operación ORA y la búsqueda en una tabla indexada `tabcodcarcu` se obtiene el código del carácter a representar, del subset de 16 caracteres formados por todas las combinaciones de 2x2 megapixels. Son las instrucciones

```
        ldy #stride1
@buincu
```

```
        lda (punpix1),y
        asl
        asl
        ora (punpix2),y
        tax
        lda tabcodcarcu,x
        sta (puncha3),y
        dey
        bpl @buincu
        ldx conbyte
        dex
        bne @buexcu
        ldy indconrephor
        bne @burege
        rts

copipun1
        lda tapufidainlo-2,y
        sta punpix1
        lda tapufidainhi-2,y
        sta punpix1+1
        lda tapuficaculo-1,x
        sta puncha3
        lda tapuficacuhi-1,x
        sta puncha3+1

        lda tapufidainlo-1,y
        sta punpix2
        lda tapufidainhi-1,y
        sta punpix2+1
        rts
```

RUTINAS AUXILIARES

Borrar las filas que forman el margen superior.

```
; reg.X numero filas a borrar
delrows
@higext
        lda #32
        ldy #stride1
@higint
        sta (punscrn1),y
        dey
        bpl @higint
        lda #stride
        clc
```

```
        adc punscrn1
        sta punscrn1
        bcc @higbyo          ; salta si CC comprobacion byte HI
        inc punscrn1+1
@higbyo
        dex
        bne @higext
        rts
```

RUTINAS DE INICIALIZACIÓN

```
*=$1B6E
```

La rutina `initpet` en la dirección 7022 calcula los valores de las tablas con los punteros a las secuencias intermedias `tapufidainhi` y `tapufidainlo`. La dirección inicial en dicha tabla viene en las variables `adfidain` y `adfidain+1`.

```
initpet

; filltab
        ldy #0
        lda adfidain+1
        sta pungra+1
        lda adfidain
        sta pungra

@nexdir sty conbyte
        lda #0
        ldy #stride1
@bufill sta (pungra),y
        dey
        bpl @bufill

        ldy conbyte
        lda pungra+1
        tax
        sta tapufidainhi,y
        lda pungra
        sta tapufidainlo,y
        clc
        adc #stride        ; ancho columnas
        bcc @inbyhi
        inx
@inbyhi
        stx pungra+1
        sta pungra
        iny
```

```
        cpy #20
        bne @nexdir
; inicializar conrepver, conrephor,
        ldx #1
        stx conrepver
        inx
        stx conrephor
        rts
```

La rutina `initvic` también calcula los valores de las tablas con los punteros a las secuencias intermedias `tapufidainhi` y `tapufidainlo`. La dirección inicial en dicha tabla también viene en las variables `adfidain` y `adfidain+1`.

Pero además inicializa los registros del chip de vídeo, la memoria de pantalla y la memoria de color. La memoria de pantalla se inicializa porque no coincide con el espacio asignado por defecto, en el caso de ampliación de memoria cambiar la dirección base, y en todo caso cambia el tamaño de pantalla de los 23x22 caracteres por defecto a 25x25.

```
vic=$9000

datvid=127 ; 254
altopan=25
nufilpan=2*altopan
nucolpan=stride


; datos video  si bit 7 $9002 es 0
;     bits 4-7   dir memoria     bits 0-3   dir memoria
;                pantalla                   chrset
;     $C0        $1000           $00        ROM (may/graf)
;     $D0        $1400           $02        ROM (may/min)
;     $E0        $1800           $0C        $1000
;     $F0        $1C00           $0D        $1400
;                                $0E        $1800
;                                $0F        $1C00 solo 128caracteres
; datos video  si bit 7 $9002 es 1
;     bits 4-7   dir memoria     bits 0-3   dir memoria
;                pantalla                   chrset
;     $C0        $1200           $00        ROM (may/graf)
;     $D0        $1600           $02        ROM (may/min)
;     $E0        $1A00           $0C        $1000
;     $F0        $1E00           $0D        $1400
;                                $0E        $1800
;                                $0F        $1C00 solo 128caracteres
;

; columnas      26    25   24
; despl hor      8     9    ?
; filas         25    21   23
```

```
; despl ver     30    40    ?
```

Inicialización de los registros.

```
;                                  bit 7 es bit 4 datos video
;               desp.h   desp.v      ncols    2*nrows    * memoria
datvicpal  byte     10,      35,  nucolpan,  nufilpan, 0,      $F2
datvicini bytes 6

initvic ldy #5
@setpal lda vic,y
        sta datvicini,y
        lda datvicpal,y
        sta vic,y
        dey
        bpl @setpal
        lda #$1C
        sta $288
```

Inicialización de la memoria de color y la memoria de pantalla.

```
; resetear tabla codigos
        lda #0
        tay
@buczer
        sta $9400,y
        sta $9500,y
        sta $9600,y
        iny
        bne @buczer
        lda #$20
@bucspc
        sta $1C00,y
        sta $1D00,y
        sta $1E00,y
        iny
        bne @bucspc
```

Cálculo de los valores de las tablas con los punteros a las secuencias intermedias. Esta parte del código es idéntica a la de `initpet`, pero cambia el “stride” de 40 a 25. También inicializa `conrepver` y `conrephor`, a 1 y 0 respectivamente.

```
filltab
;        ldy #0
        lda adfidain+1
        sta pungra+1
        lda adfidain
        sta pungra
```

```
@nexdir sty conbyte
        lda #0
        ldy #stride1
@bufill sta (pungra),y
        dey
        bpl @bufill

        ldy conbyte
        lda pungra+1
        tax
        sta tapufidainhi,y
        lda pungra
        sta tapufidainlo,y
        clc
        adc datvicpal+2     ; ancho columnas
        bcc @inbyhi
        inx
@inbyhi
        stx pungra+1
        sta pungra
        iny
        cpy #20
        bne @nexdir
; inicializar conrepver, conrephor,
        ldx #1
        stx conrepver
        inx
        stx conrephor
        rts
```

Finalmente `finvic` restituye los registros del VIC al tamaño de pantalla de 22x23, para que el editor de BASIC funcione bien, al salir a BASIC.

```
finvic  ldy #4
@setpal lda datvicini,y
        sta vic,y
        dey
        bpl @setpal
        rts

endif
```

POSIBLES MODIFICACIONES

SIMPLIFICACIONES

En el caso del PET no hay diferentes configuraciones de memoria y el espacio para cálculos intermedios es un bloque de 800 bytes en la dirección $1C00. Se puede retirar la rutina `initpet` excepto la parte final que inicializa `conrepver` y `conrephor`, si se definen las tablas que se calculaban en tiempo de ejecución en el mismo listado de ensamblador.

Primero particiona dicho espacio de memoria para cada una de las filas de cálculos intermedios, y se etiqueta cada parte para que su dirección de inicio pueda ir a la tabla de punteros que corresponda de las 6 que se definieron (`tapufidainlo`, `tapuficamelo`, etc.).

```
*=1C00

; esto son 20*stride bytes 800 en PET y 500 en VIC
fidain0 bytes stride
fidain1 bytes stride
fidain2 bytes stride
fidain3 bytes stride
fidain4 bytes stride
fidain5 bytes stride
fidain6 bytes stride
fidain7 bytes stride

; filas datos caracteres graficos medios
fidame0 bytes stride
fidame1 bytes stride
fidame2 bytes stride
fidame3 bytes stride
fidame4 bytes stride
fidame5 bytes stride
fidame6 bytes stride
fidame7 bytes stride

; filas datos caracteres graficos medios
fidacu0 bytes stride
fidacu1 bytes stride
fidacu2 bytes stride
fidacu3 bytes stride
```

Como se trata de unas reservas de memoria que pasan al ensamblado, el ensamblador las inicializa a cero.

Luego ya se aprovechan esas etiquetas para construir las tablas de punteros que antes construía la rutina, sustituyendo las tablas vacías al principio del listado de CM, que reservaban el bloque de memoria entre $1978 y $19A0.

```
; punteros filas datos intermedios
```

```
tapufidainlo byte <fidain0, <fidain1, <fidain2, <fidain3, <fidain4,
<fidain5, <fidain6, <fidain7

; punteros filas datos caracteres graficos medios
tapuficamelo byte <fidame0, <fidame1, <fidame2, <fidame3, <fidame4,
<fidame5, <fidame6, <fidame7

; punteros filas datos caracteres graficos cuadritos
tapuficaculo byte <fidacu0, <fidacu1, <fidacu2, <fidacu3

; punteros filas datos intermedios
tapufidainhi byte >fidain0, >fidain1, >fidain2, >fidain3, >fidain4,
>fidain5, >fidain6, >fidain7

; punteros filas datos caracteres graficos medios
tapuficamehi byte >fidame0, >fidame1, >fidame2, >fidame3, >fidame4,
>fidame5, >fidame6, >fidame7

; punteros filas datos caracteres graficos cuadritos
tapuficacuhi byte >fidacu0, >fidacu1, >fidacu2, >fidacu3
```

SINCRONIZACION RASTER CON EL PET

En este apartado se explica una sincronización un poco burda del reloj con el “raster retrace” para que las modificaciones en la memoria de pantalla puedan empezar antes del “retrace” y ganar el espacio del margen superior.

Aunque hay que decir que no me ha conducido a mucha mejora al ejecutar en el ordenador. Pero es un ejemplo de lo que representa un conjunto de técnicas que se utilizan en muchos ordenadores de 8 bits en muchas circunstancias.

Lo que se hace es modificar la rutina `initpet` y donde se acaba añadir unas instrucciones para inicializar las interrupciones del reloj y desviar la dirección que procesa las interrupciones hacia nuestro código.

El reloj del PET funciona a 1MHz exacto, cosa que no es habitual. La tasa de refresco de la pantalla es 60Hz en PET americanos y 50Hz en PET europeos. Por tanto el tiempo del frame es de 16666 ciclos por frame y 20000 ciclos por frame respectivamente. Alguna manera habrá de discernirlo en el runtime pero en principio no la conozco.

Constantes

```
tiempoajuste=10000
;tiempoajuste=13333
tiempopframe=16666
;tiempoajuste=16000
;tiempopframe=20000
```

Direcciones

```
TIMERLOW=$E844 ; si timer 1, E848 si timer 2
TIMERHIGH=$E845 ; si timer 1, E849 si timer 2
VIAACR=$E84B
VIAIFR=$E84D
VIAIER=$E84E

IRQBASV1=$219
IRQBASV2=$90

initpet

; filltab
        ldy #0
…
…
        stx conrephor
;        rts
```

Se comenta el `rts` y las siguientes instrucciones lo que hacen es cambiar el puntero de la interrupción a la rutina `interup1`. La instrucción `cmp #$E0` detecta si está en BASIC 1 o superior, aunque al final la modificación en BASIC 1 no funciona, porque parece ser que utiliza el timer para sus propios propósitos.

```
; cambiar irq
        sei
        ldx #<interup1
        ldy #>interup1
        lda IRQBASV2+1
        cmp #$E0
        bcc @basic1
        sta oldirq+1
        lda IRQBASV2
        sta oldirq
        stx IRQBASV2
        sty IRQBASV2+1
        bcs @verbas
@basic1
        lda IRQBASV1+1
        sta oldirq+1
        lda IRQBASV1
        sta oldirq
        stx IRQBASV1
        sty IRQBASV1+1
@verbas cli
        rts
```

En el PET el “raster retrace” dispara una interrupción, por lo que cuando se ejecute `interup1` lo más probable es que esa sea la situación. Las tres primeras instrucciones comprueban que ese sea el caso y si no, procede a la rutina de tratamiento de interrupciones de la ROM a través de `jmp (oldirq)`.

Si está en situación de retrace inicializa el “timer” a través de `VIAACR` y `VIAIER`, y lo activa a la vez que define el tiempo de alarma con la instrucción `stx TIMERHIGH`. Esta alarma se ejecuta una sola vez. Se cambia el vector de interrupción a `interup2`.

```
interup1
; sincronizar con VBLANK para arrancar retraso
        lda $E840
        and #$20
        bne @salida
        lda #%00000000 ; TIMER 1 ONE SHOT
        sta VIAACR
        ldx #>tiempoajuste
        lda #<tiempoajuste
        sta TIMERLOW
        stx TIMERHIGH
        lda #%11000000 ; ENABLE TIMER1
        sta VIAIER
;        sei
        ldx #<interup2
        ldy #>interup2
        jsr auxinterup
@salida
        jmp (oldirq)
```

A esta rutina se llega después de cambiar por segunda vez el vector de interrupciones. Si llega aquí es porque el timer que se arrancó en el “retrace” está corriendo y es posible incluso que haya llegado a 0 y disparado la alarma causando la interrupción.

Las tres primeras instrucciones comprueban que ese sea el caso y si no, procede a la rutina de tratamiento de interrupciones de la ROM a través de `jmp (oldirq)`.

```
interup2
        bit VIAIFR
        bvs @inter1 ; deberia pasar siempre qe esta
interrupcionantes del siginete blank
        jmp (oldirq)
```

Si la interrupción la ha causado el “timer” quiere decir que el “raster” ahora está algo por debajo de la mitad de la pantalla. Esto por haber usado como intervalo temporal el 60% del tiempo del “frame”, pero podría funcionar mejor el 70% o el 80%, es un parámetro que se puede intentar ajustar por prueba y error.

De nuevo lo que hace la rutina es reinicializar el “timer” a través de `VIAACR` y `VIAIER`, y activarlo a la vez que define el tiempo de alarma con la instrucción `stx TIMERHIGH`. Ahora la alarma es de repetición.

De ahora en adelante la alarma dispara una interrupción en cada frame y además en el mismo punto del “raster” si el tiempo de alarma se estimó correctamente. En estas máquinas el reloj está sincronizado con el vídeo y no hay retraso. Podría ser que el “timer” tarda 2 ciclos en inicializarse, en ese caso habría que restar 2 ciclos al tiempo de alarma.

Se cambia el vector de interrupción a `interup3`.

Entonces se pueden empezar a hacer los cálculos y a modificar la memoria de pantalla correspondiente a las filas superiores e ir bajando a las inferiores después, porque el “raster” va escapando.

```
@inter1
        lda #%01000000 ; TIMER 1 CONTINUOUS
        sta VIAACR
        ldx #>tiempopframe
        lda #<tiempopframe
        sta TIMERLOW
        stx TIMERHIGH
;        sei
        ldx #<interup3
        ldy #>interup3
        jsr auxinterup
        jmp salinter
```

La rutina `interup3` es la rutina definitiva, lo que hace es indicar al programa que puede empezar a escribir la memoria de pantalla. Como en las otras dos rutinas de tratamiento de interrupción, las 2 primeras líneas comprueban la causa de la interrupción y si no es la que esperamos tratar nosotros se desvía a la ROM con `jmp (oldirq)`.

Para que la sincronización funcione correctamente el programa debería haber acabado todas las tareas del “frame” y estar ejecutando un bucle de espera, a la espera de que cambie el estado de la bandera cosa que aquí hace `inc banraster`. En este caso al mezclar BASIC y LM es más difícil asegurar eso.

Lo que hace la instrucción `lda TIMERLOW` es informar al “timer” que se ha procesado la interrupción, si no seguiría generando “requests” continuamente.

```
interup3
        bit VIAIFR
        bvs @inter3
        jmp (oldirq)
@inter3
        lda TIMERLOW
        inc banraster
```

Para salir de las interrupciones que tuvo que procesar nuestro programa y no necesitaron ser desviadas a la ROM hace falta terminar con las siguientes instrucciones:

```
salinter
        pla
        tay
        pla
        tax
        pla
        rti

auxinterup
        sei
        lda oldirq+1
        cmp #$E0
        bcc @basic1
        stx IRQBASV2
        sty IRQBASV2+1
        bcs @verbas
@basic1
        stx IRQBASV1
        sty IRQBASV1+1
@verbas
        cli
        rts

oldirq bytes 2
```