

Introducción

Hay tres razones por las que he decidido escribir este curso: La primera de ellas es tratar de acercar la programación de videojuegos para ZX Spectrum al mayor número de personas posible. La segunda que la mayoría de los tutoriales son demasiado complicados para mi gusto. La tercera es poder leer mi propio tutorial cuando se me haya olvidados más de una cosa. XD

De rebote, sin quererlo ni beberlo, este curso también servirá para hacer juegos de Amstrad CPC cambiando cuatro cosas. En primer lugar se explicará la forma de hacerlo para ZX Spectrum y a continuación la de Amstrad CPC y sus cambios.

Tras esto voy a ser totalmente sincero: este tutorial no va a mostrarte todo lo que se puede hacer en Z88DK, pero si te va a abrir las puertas para coger luego otro tutorial mejor y quedarte todo más claro que la ropa recién lavada con Ariel Blanco Nuclear. Cuando te hayas leído esto podrás pasar a uno de esos tutoriales machotes llenos de palabras que ahora mismo te suenan a chino.

Es decir, sólo voy a explicar lo básico para que puedas arrancar y defenderte. Pero creo que mejor lo vemos con la práctica.

Radastan

Preparando nuestro entorno de trabajo

Para poder empezar a trabajar es imprescindible tener lo más importante... Z88DK. De nada sirve que hagamos ejemplos y aprendas nada si no puedes ponerlo en práctica.

Voy a presuponer que tienes Windows, si tienes Linux tendrás que ingeniártelas (cosa que ya sabrás hacer si usas ese sistema operativo). Así que desde ahora todo irá explicado para Windows.

Para empezar descárgate el Z88DK de este enlace, que es la versión que vamos a usar:

<http://www.bytemaniacos.com/ficheros/zxspectrum/z88dk10.zip>

Se trata de una versión de Z88DK "portable", es decir, no hace falta instalar nada. Sólo descomprime el fichero tal cual en el directorio raíz C: de tu ordenador. Ojo, descomprimir allí, no en una subcarpeta. Debería crearse una nueva carpeta llamada z88dk10, y dentro todo el follón de cosas que trae este compilador.

Se trata de la versión que se usa en la "Churrera" de los Mojon Twins, por lo que de paso ya tenéis excusa para mirar su curso de paso si veis que esto es muy complicado. Ellos ya han hecho un motor basado en Z88DK y no hace falta programar nada, sólo dedicarse a hacer gráficos, mapa, y configurar un ficherito con la forma que queremos que actúe nuestro juego. Más info en:

<http://www.elmundodelspectrum.com/taller.php>

Bien, ahora el segundo paso. Créate donde te de la real y santísima gana una carpeta para trabajar. Da igual como se llame (Curso, Megamotor, Eljuegodelaño, CastlevaniaZX, etc), dentro vamos a crear nuestros ficheros para hacernos un entorno amigable.

Cómo editor puedes usar lo que te dé la gana: Notepad, Wordpad, Crimson Editor, etc. Yo te recomiendo el Crimson Editor porque es sencillo, te va a poner los comandos de distinto color, te mostrará los números de línea, y es gratis:

<http://www.crimsoneditor.com/>

Bien, y ahora como colofón vamos a descargar un esqueleto básico de Z88DK en:

<http://www.bytemaniacos.com/ficheros/zxspectrum/cursoz88dk.zip>

El contenido del ZIP lo metes dentro de la carpeta que has creado donde has querido (Curso, Megamotor, Pedazojuego, MetalGearZX, CPCpowa, etc).

Y ahora pasamos a lo bueno...

Familiarizándonos con el lenguaje C

Vete a la carpeta de trabajo y abre el fichero juegozx.c. Deberías ver algo como esto:

```
// juegozx.c
// Esqueleto de juegos de Radastan para ZX Spectrum
// version 0.1 beta

#include <config.h>
#include <spriteszx.h>
#include <lib\motorzx.h>

#pragma output STACKPTR=65000

void main (void)
{

}
```

Vamos a explicar de forma muy sencillita lo que es cada cosa:

- Los comentarios van siempre precedidos de `//`. En lenguaje C se hace así y hace que toda esa línea sea ignorada por el compilador (no, los comentarios no están en el fichero compilado). Si quieres poner como comentario varias líneas seguidas o anular parte de tu código (muy útil) puedes hacer lo mismo empezando con `/*` y terminando con `*/`.
- Los `#include` es el copy&paste de los programadores. Para no apilonar todo en un mismo fichero (que es posible) usamos `#include` para que el compilador meta todo lo que haya en el fichero que le mencionamos dentro de la compilación. Es decir, en este caso es como si en juego.c estuviera donde hay cada `#include` el contenido de dichos ficheros.
- El `#pragma` lo olvidas, tú lo usas como está y ya está. Básicamente dice dónde va a estar la pila del Z80, una zona de memoria donde se guardan cositas que si no sabes de ensamblador de poco o nada te va a sonar. Con 65000 tenemos 538 bytes reservados para ello y si no haces cosas muy raras sobrarán.
- Al principio también se pueden poner `#define`, pero no recomiendo que los pongas en el fichero principal. Sirve para definir variables. Sí, es posible crear variables globales para todo el programa. Las nuestras las vamos a meter en `config.h`, y nos servirán para especificar los valores que vayamos a necesitar. Es más abre `config.h` y verás que te he dejado una variable imprescindible lista para usar.

La chicha de “juegozx.c” empieza en `void main (void)` y lo que hay entre sus corchetes.

Definiendo funciones en C

En lenguaje C se trabaja con funciones. Las funciones son partes de código, y podemos crear tantas funciones como nos venga en gana. Lo normal es que hagamos funciones de cosas que repitamos de forma cíclica, así no tendremos que volver a escribir el mismo código en distintos sitios. También nos sirven para organizar mejor nuestro código.

La función principal es "main", y es la que primero se va a ejecutar cuando ejecutemos el código. Desde ella vamos a llamar a las distintas partes que conforman nuestro programa, y es conveniente que no pongamos en ella código en si mismo, sino llamadas a otras funciones.

Cuando termine "main" de ejecutarse el programa también lo hará, así que tenemos que evitar que esto suceda a menos que lo deseemos expresamente.

Toda función tiene una entrada y una salida, sino queremos tener alguna de las dos (o las dos) usaremos void para indicarlo. La entrada va entre paréntesis, al final de la declaración, y la salida al principio de la misma.

Si quisiéramos crear una función que sumara dos valores, por ejemplo, y nos diera el resultado sería algo como esto:

```
int suma (int valor1, int valor2)
{
    int total;
    total=valor1+valor2;
    return (total);
}
```

Ignorad los int de momento, lo importante ahora es ver que hemos definido una función llamada "suma" con valores de entrada y un valor de salida, el valor lo devolvemos mediante el comando return.

De todas formas este ejemplo nos sirve para enseñar un par de cosas más:

- Todos los comandos en C deben acabar con punto y coma (salvo las funciones que definamos).
- Las funciones engloban su contenido mediante corchetes.

¿Cómo usaremos esta función que hemos creado? llamándola desde otra función pasando los parámetros que necesita y recogiendo el valor que proporciona. Un ejemplo desde main sería:

```
void main (void)
{
    int valor1;
    int valor 2;
    int resultado;
    valor 1 = 10;
    valor 2 = 35;
    resultado = suma (valor1, valor2);
}
```

Definiendo variables y constantes

Las variables son el alma de todo programa, con ellas pasamos datos de un lugar a otro y podemos realizar operaciones. Las variables pueden ser globales o locales:

- Las variables globales se usan en todas las funciones, son las que hemos definido al principio del código con #define.
- Las variables locales sólo se usan dentro de cada función, no son vistas por el resto de funciones, dichas variables no pueden llamarse como ninguna variable global.

Un ejemplo de variable local es la que hemos definido en config.h:

```
#define NUMERO_DE_VIDAS 0
```

A la hora de definir la variable podemos indicar también un valor inicial, en nuestro caso hemos puesto el valor cero. No hace falta indicar el tipo de variable que es, sólo el nombre y su valor. Tampoco hay que terminar con punto y coma.

En las variables internas la cosa cambia:

```
int total;
```

Una variable local termina su declaración con punto y coma, y además se debe crear indicando el tipo de variable que es. Los tipos de variables básicos (hay más) que podemos manejar en C con Z88DK son:

char, 8 bits, se usa para definir caracteres

signed char, 8 bits con signo (-127 a +127)

unsigned char, 8 bits sin signo (0 a 255)

int, 16 bits con signo (-32767 a 32767)

signed int, igual que el anterior (es lo mismo)

unsigned int, 16 bits sin signo (0 a 65535)

long, 32 bits con signo

signed long, igual que el anterior (es lo mismo)

unsigned long, 32 bits sin signo

float, 32 bits coma flotante con precisión simple (para definir valores reales)

double, 64 bits coma flotante con precisión doble (para definir valores reales)

No os recomiendo el uso de más de 16 bits a menos que sea absolutamente imprescindible, ya que todas las operaciones que hagamos serán seriamente penalizadas (el Z80 está pensado para operaciones con 8 y 16 bits). A todos los efectos con char e Int tenemos de sobra.

Unos ejemplillos:

```
unsigned char valor_x; // Define una variable llamada valor_x, que puede valer desde 0 a 255
```

```
unsigned int memoria_pantalla; // Define una variable de 16 bits sin signo (de 0 a 65535)
```

```
unsigned char valor1, valor2; // También es posible definir dos variables a la vez (o más)
```

Os quedará más claro cuál usar en cada caso cuando os llegue el momento.

El ZX Spectrum por dentro

Para poder programar en una máquina es absolutamente imprescindible conocerla. Así por encima, y yendo a lo básico, podemos dividir el ZX Spectrum por dentro en:

- Memoria ROM, es la que tiene el sistema operativo (el BASIC), no modificable
- Memoria RAM, es modificable y es la que usa el programa
- Puertos de entrada/salida, nos comunican con el teclado, el sonido, etc.
- El Z80, la CPU, el procesador que va a ejecutar el programa

Tranquilos, no vamos a hacer un cursillo de hardware, sólo vamos a conocer lo estrictamente necesario para poder programar nuestro juego.

La memoria

La memoria la vamos a dividir en tres bloques, aunque realmente sean dos (ROM y RAM):

- 16384 bytes de ROM, que contiene el BASIC, va de 0 a 16383 en decimal.
- 6192 bytes de RAM, que contiene la imagen de pantalla, va de 16384 a 22575.
- 42960 bytes de RAM, que están disponibles para el programa (teóricamente)

La pantalla se divide a su vez en dos zonas, la que contiene los pixels (puntos) de pantalla y la que contiene los colores (atributos) por cada 8x8 pixels (carácter). La estructura es algo rara, pero tiene su sentido si programas en ensamblador (cosa que no vamos a hacer).

Pixels

La pantalla posee una resolución de 256x192 pixels, que traducido en bytes son 32 bytes de ancho por 192 de alto (cada byte son 8 bits, recordad).

Es decir, los pixels van agrupados de 8 en 8 bits (un byte) desde 16384 a 22527 (6144 bytes). No lo hacen de forma lineal, sino dividiendo la pantalla en tres tercios:

| |
|--------------------------|
| Tercio 1 – 16384 a 18431 |
| Tercio 2 – 18432 a 20479 |
| Tercio 3 – 20480 a 22527 |

¿Os acordáis cómo carga una pantalla de presentación de un juego? Primero se rellena el tercio superior: se empieza con la primera línea, luego la primera de la siguiente línea de caracteres, así hasta llegar a la parte baja del tercio y continuar con la segunda línea de cada fila de caracteres. Cada fila de caracteres (8x8 pixels) tiene 8 líneas de altura.

Luego se pasa al tercio medio, y luego al tercio de abajo.

El primer tercio ocupa 2048 bytes (32x64 bytes) y empieza en 16384, el segundo empieza en 18432, y el tercero en 20480 para terminar en 22527.

No os volváis locos desde C es absurdo generar gráficos porque aunque se compile, el código resultante no llega a ser tan eficiente como el lenguaje ensamblador optimizado puro y duro. Lo suyo es emplear librerías de terceros (o el fichero que os tengo preparado).

De todas formas viene bien saber cómo es la pantalla para optimizar nuestro juego.

Colores

Los colores son más sencillos, ya que la pantalla se divide en 32x24 caracteres (8x8 bits) y están ordenados linealmente. Es decir se empieza por la esquina superior izquierda y se termina por la parte inferior derecha. Cada carácter posee un valor de pixel (tinta) y un valor de fondo (papel), que están definidos por la siguiente tabla para dar el valor de 8 bits que se almacena (sumad el valor de tinta con el valor de papel):

| Color de tinta | valor decimal |
|----------------|---------------|
| Negro | 0 |
| Azul | 1 |
| Rojo | 2 |
| Magenta | 3 |
| Verde | 4 |
| Azulado | 5 |
| Amarillo | 6 |
| Blanco | 7 |

| Color de Papel | valor decimal |
|----------------|---------------|
| Negro | 0 |
| Azul | 8 |
| Rojo | 16 |
| Magenta | 24 |
| Verde | 32 |
| Azulado | 40 |
| Amarillo | 48 |
| Blanco | 56 |

También podemos definir si queremos los colores sin brillo (por defecto) o con brillo, basta sumar el valor 64 a la suma resultante.

Puertos de entrada y salida

Los puertos de entrada y salida son nuestra forma de relacionarnos con el exterior, a ellos están conectados el teclado, el altavoz, la entrada MIC, la salida EAR, etc. Lo fundamental para nosotros es saber usar los puertos del teclado, los de joystick, y la salida de sonido (que también es la del borde de pantalla).

La gran pega es que desde C a pelo no se pueden acceder a los puertos, pero como esto va de hacer sencillo lo complejo os he creado sendas rutinas de lectura y escritura en puertos que veremos más adelante.

El teclado

El teclado del ZX Spectrum se puede describir de cara a nosotros como la siguiente tabla:

| Puerto | D0 | D1 | D2 | D3 | D4 |
|--------|------------|--------------|----|----|----|
| 65278 | CAPS SHIFT | Z | X | C | V |
| 65022 | A | S | D | F | G |
| 64510 | Q | W | E | R | T |
| 63486 | 1 | 2 | 3 | 4 | 5 |
| 61438 | 0 | 9 | 8 | 7 | 6 |
| 57342 | P | O | I | U | Y |
| 49150 | ENTER | L | K | J | H |
| 32766 | ESPACIO | SIMBOL SHIFT | M | N | B |

Si leyéramos el puerto correspondiente habría que usar los bits D0 a D4, que en decimal corresponden:

$$D0 = 1, D1 = 2, D2 = 4, D3 = 8, D4 = 16$$

Es decir, si hay más de una tecla pulsada tendríamos la suma de las teclas. Pero OJO: las teclas pulsadas son bits a cero, es decir, en estado normal los bits están a uno.

Joystick Sinclair

El joystick Sinclair usa los números del teclado, por lo que:

Sinclair Joystick 1: 1(izq) 2(dcha) 3(abajo) 4(arriba) 5(disparo)

Sinclair Joystick 2: 6(izq) 7(dcha) 8(abajo) 9(arriba) 0(disparo)

El beeper y el borde de la pantalla

Ambos comparten el mismo puerto, el 254.

Si usamos un valor de 0 a 7 cambiaremos el color del borde (la tabla es la misma que para la tinta del área de atributos de la pantalla).

Si cambiamos el bit 4 (valor 16 en decimal) mandaremos un pulso al beeper (el altavoz interno del ZX Spectrum).

Aunque podemos crear nuestra propia rutina para usar el Beeper lo suyo es usar alguna de las librerías o rutinas que ya están más que probadas y optimizadas para ello. Beepola, por ejemplo, aparte de servirnos para componer excelentes melodías multicanal también no genera el código que podemos usar en Z88Dk casi directamente (hay una sección dedicada a ello).

No obstante, como esto va de aprender, os enseñaré cómo hacer algunos FX sencillitos por vosotros mismos para dar algo de animación a los juegos.

Nuestros primeros pinitos en C

Bueno, hemos llegado al momento cumbre, la de empezar a hacer cosas. Como me gusta ser práctico vamos a ver la programación C desde cero y con ejemplos que os sirvan para hacer un juegucillo. Vamos a abstraernos lo máximo posible de la máquina gracias a las subrutinas (nuestra librería) para que si el día de mañana queremos cambiar a otro ordenador nos sirva el código base del juego y sólo tenemos que cambiar la librería.

Para empezar vamos a sacar partido del fichero (motorzx.h) que os he preparado. No es la octava maravilla y seguramente sea optimizable hasta la saciedad, pero para empezar proporciona una sencillez inusitada y es como cien veces más rápida que el BASIC.

Así que vamos a ver la primera rutina molona, y es la que nos proporciona un borrado de la pantalla:

```
void cls (int color)
```

Para usar la rutina tenemos que usar la famosa tabla de colores, pero lo suyo es comenzar con el clásico color negro. Así que poned lo siguiente dentro de los corchetes de void main (void) en el fichero juego.c:

```
cls(0);
```

Os quedará así:

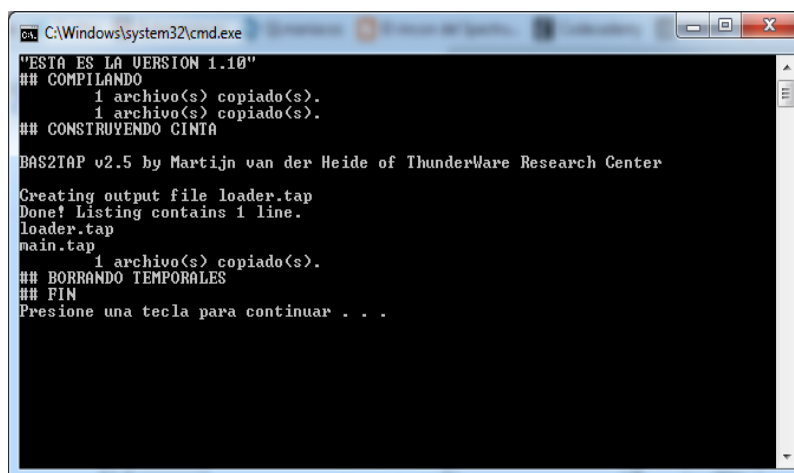
```
void main (void)
```

```
{
```

```
    cls(0);
```

```
}
```

Vale, ya hemos escrito nuestro programa... ¿cómo genero el juego? Pues muy sencillo, os he ahorrado todos los pasos intermedios y basta ejecutar el fichero makezx.bat que hay en la misma carpeta de juegozx.c. Ejecutadlo y si todo va bien os saldrá algo como:



```
C:\Windows\system32\cmd.exe
"ESTA ES LA UERSION 1.10"
## COMPILANDO
   1 archivo(s) copiado(s).
   1 archivo(s) copiado(s).
## CONSTRUYENDO CINTA
BAS2TAP v2.5 by Martijn van der Heide of ThunderWare Research Center
Creating output file loader.tap
Done! Listing contains 1 line.
loader.tap
main.tap
   1 archivo(s) copiado(s).
## BORRANDO TEMPORALES
## FIN
Presione una tecla para continuar . . .
```

Os habrá aparecido un ficherito llamado "juegozx.tap" que podéis cargar en cualquier emulador. Si lo ejecutáis os saldrá una bonita pantalla en negro y se habrá salido al BASIC.

Pero esto no tiene emoción, necesitamos sentir el poder del lenguaje C y de Z88DK desde el primer momento, así que dejémonos de chorradas y vamos a algo serio, pintar un sprite.

Pon la siguiente línea después del `cls(0)`; que hemos puesto:

```
put_sprite_x16 (sprite_cubo, 0, 0);
```

Vuelve a compilar y... si, ahora se ha borrado la pantalla y tenemos un precioso cubo de colorines en pantalla. ¡Wow! Dos pasos y ya estamos pintando sprites. Las subrutinas que lo hacen posible están en `motorzx.h`, por si queréis mirar, pero están en completo lenguaje ensamblador. Se hace así para ganar toda la velocidad posible.

¿Dónde está el sprite? Lo tenemos en `spriteszx.h` y está generado también con lenguaje ensamblador. Tranquilos, basta una herramienta como SevenUp para hacer sprites y que se genere este código de forma automática (está en la carpeta `utilszx`). Pero ahora vamos a seguir jugando con los sprites que os he dejado de regalo para hacer un juego sencillito.

La rutina put_sprite_x16

Esta rutina no es de Z88DK, la hemos añadido nosotros gracias al fichero motorzx.h. Como hemos visto previamente hemos añadido dicho fichero a la compilación mediante el comando #include y de esta forma todas las subrutinas que hubiera dentro de ese fichero forman parte de nuestro programa.

Z88DK trae varias librerías, como todo compilador, y es posible incluso usar librerías de terceros que os darán todas las facilidades necesarias para no preocuparos de tareas tediosas o complejas. motorzx.h no es una librería, es un fichero con rutinas, que es lo mismo pero no es igual. Quiere decir que las rutinas que no emplees de ese fichero seguirán presentes en el código final aunque no las uses, cosa que no pasa con una librería, pero como ocupan muy poquito no debes preocuparte.

Pero entrando en detalles, veamos cómo se realiza una llamada a put_sprite_x16:

```
put_sprite_x16 (puntero_a_sprite, x, y);
```

Tenemos tres parámetros de entrada y ninguno de salida (la salida es que se imprime el sprite en pantalla). Creo que todos sabéis lo que significa X e Y, las coordenadas, lo que debéis saber es que se refieren a caracteres (celdas de 8x8 pixels) y que la dirección 0,0 está en la parte superior izquierda.

Ojito con poner una dirección fuera de pantalla, ya que la rutina no comprueba si nos hemos equivocado y normalmente obtendremos un bonito cuelgue de nuestro juego si lo hacemos así.

Puntero_a_sprite debemos cambiarlo por el puntero a nuestro sprite (fichero spriteszx.h). Si seguís mis instrucciones veréis que en dicho fichero hay una serie de sprites ya definidos para que comencéis a probar, basta que uséis el nombre que les precede sin el carácter “_” del comienzo.

Es decir, si queréis ver un bonito bicho en pantalla podéis hacer:

```
put_sprite_x16 (sprite_prota1, 12, 12);
```

Realizad un makezx, comprobáis el resultado, y probad vosotros con otro sprite de los que veáis en spriteszx.h.

Bucles for

Dibujar algo en pantalla mola, pero nosotros queremos hacer más cosas. Mucho más.

Por eso vamos a aprender lo que son los bucles, que tienen exactamente la misma función que en BASIC, se llaman igual, pero se usan de forma ligeramente distinta.

Supongamos que queremos pintar toda la fila de arriba de la pantalla con bonitos árboles. Tenemos un sprite precioso a todo color en spriteszx.h, que curiosamente nos ocupa 3x3 caracteres... ondia, no podemos usar entonces `put_sprite_x16...` no, pero tenemos otra para ese tipo de gráficos:

```
put_sprite_x24 (puntero_a_sprite, x, y);
```

Para ver cómo queda basta un:

```
put_sprite_x24 (sprite_arbol, 0, 0);
```

El código completo de juegozx.c sería:

```
// juegozx.c
// Esqueleto de juegos de Radastan para ZX Spectrum
// version 0.1 beta

#include <config.h>
#include <spriteszx.h>
#include <lib\motorzx.h>

#pragma output STACKPTR=65000

void main (void)
{
    cls(0);
    put_sprite_x24 (sprite_arbol,0,0);
}
```

Si hacemos el `makezx` y ejecutamos el `tap` resultante nos saldrá una bonita pantalla negra con un árbol. ¡Que bonitooooo! Pero está solo, muy solo, lo suyo es acompañarlo, así que hacemos un bucle.

Pero antes de hacerlo, recordad, no podemos salirnos de la pantalla o pasarán cosas graciosas.

El bucle sería algo así:

```
void main (void)
{
    unsigned char a;
    cls(0);
    for (a=0;a<30;a=a+3)
    {
        put_sprite_x24 (sprite_arbol,a,0);
    }
}
```

La sintaxis del bucle for está entre paréntesis y separados en tres campos con punto y coma. El primer campo indica el inicio de la variable, el segundo la condición para continuar con el bucle, y la tercera el incremento.

En nuestro caso hemos usado la variable "a", que es del tipo unsigned char (de 0 a 255). La vamos a incrementar desde cero a un valor menor de 30, incrementado de tres en tres. Los caracteres de pantalla van de 0 a 31 (32 en total por fila), por lo que el valor máximo para pintar un árbol sin salirnos sería 29 (que es menor que 30).

Lo que hay entre los corchetes del bucle for es lo que se va a ejecutar todo el tiempo y es ahí donde ponemos nuestra llamada para dibujar arbolitos usando "a" como variable horizontal.

Hacemos un makezx... y tenemos un bonito seto.

Pero, no sé, es que uno es muy ecológico y le gusta fomentar el verde... pues nada vamos a crear un doble bucle y a pintar toda la pantalla de arbolitos:

```
void main (void)
{
    unsigned char a, b;
    cls(0);
    for (a=0;a<30;a=a+3)
    {
        for (b=0;b<22;b=b+3)
        {
            put_sprite_x24 (sprite_arbol,a,b);
        }
    }
}
```

Pero que bosque más bonnnnnito, dan ganas de perderse en él oye. Una cosa, ¿y eso de definir dos variables a la vez? Pues sí, basta separa por comas las variables y todas las que sean del mismo tipo las podemos definir de golpe. Así ahorramos líneas y nos queda chulo total, que caramba.

Sacando cosas por los puertos – color de borde de pantalla en ZX Spectrum

Como ya os comenté previamente, el ANSI C (como se conoce al C estándar), no tiene funciones para escribir datos en puertos de salida. Pero como la idea es que os olvidéis de cosas mundanas os dejo la función chachi siguiente:

```
port_out (puerto,valor)
```

Tanto puerto como valor son variables de tipo INT a nivel interno de la rutina.

Supongamos que queremos cambiar el color del borde por uno azul oscuro, que es el valor 1 según la tabla de colores. Bastaría hacer un:

```
port_out (254,1); // Ponemos el borde de color azul
```

Con esto ya podéis personalizar la pantalla al 100%.

Scroll - while

Más de uno pensará, llegados al título de esta sección, que os voy a dar rutinas para hacer scroll sin esfuerzo alguno como el que no quiere la cosa. Estáis en lo cierto.

Cuando uno hace scroll mueve una parte de la pantalla hacia una dirección. La pega es que al mover todo se pierde lo que está al final del scroll, cosa que no suele ser importante, pero nos deja pegote la parte del principio. Se debe a que la rutina copia consecutivamente caracteres de una dirección a otra, y en la parte del comienzo del scroll todos los caracteres se mantienen como estaban.

Lo suyo es verlo con un ejemplo:

```
void main (void)
{
    unsigned char y;
    cls (0); // Borramos la pantalla
    port_out (254,1); // Ponemos el borde de color azul
    while (1)
    {
        for (y=0;y<22;y=y+3)
        {
            put_sprite_x24 (sprite_arbol, 29, y);
        }
        scroll_izquierda (0,0,32,24);
        scroll_izquierda (0,0,32,24);
        scroll_izquierda (0,0,32,24);
    }
}
```

Si ejecutáis la rutina, aparte de dejaros con la boca abierta la velocidad a la que va esto, os daréis cuenta que al comienzo del scroll pasa una cosa curiosa:



Es decir, el tile del inicio se ve repetido en su final. En este caso es tres veces porque el tile es de 3x3 caracteres, y cada vez que hacemos scroll nos queda el mismo rastro de la columna final (el borde del árbol).

Mirando la rutina veréis que dibujo una columna de árboles a la derecha de la pantalla, hago un scroll de la totalidad de la pantalla tres veces (cada llamada desplaza un carácter), y así sucesivamente.

Pero veamos las cosas nuevas que vemos por aquí:

```
while (1)
```

¡Epa! ¿Y esto? Os presento formalmente: while, alumnos, alumnos, while.

While sirve para hacer bucle, como for, pero es mucho más simple. Realiza el bucle hasta que la condición que ponemos dentro de los paréntesis deja de darse. En este caso decimos que se ejecute siempre (valor 1), por lo que tenemos un bucle infinito.

En segundo lugar tenemos a la función estrella de este apartado:

```
scroll_izquierda (x,y,ancho,alto);
```

Esta función, que está dentro de motorzx.h, realiza un scroll a la izquierda del área de pantalla definido por los valores x/y que dan la esquina superior izquierda y los valores de anchura y altura del rectángulo. Como queremos scrollear toda la pantalla lo hacemos con la esquina 0,0 y los 32 caracteres de ancho por 24 de alto que tiene en su totalidad.

Si os parece que va muy rápido el scroll podemos hacer que vaya más lento con otra rutinilla de regalo (ni los Reyes Magos, oye):

```
delay (espera);
```

En espera ponemos un valor entre 1 y 32768. Con que pongáis un valor de 10 basta. Pones un delay después de cada llamada a scroll, para que sea uniforme, y mirad el resultado.

Otras funciones de scroll son:

```
scroll_derecha (x,y,ancho,alto);
```

```
scroll_arriba (x,y,ancho,alto);
```

```
scroll_abajo (x,y,ancho,alto);
```

Lo suyo es que probéis a realizar un scroll en cada dirección, modificando la rutina “for” para que dibuje la línea de árboles en el lugar necesario.

Scroll con buffer ficticio – switch/case

Ahora vamos a arreglar el estropicio que nos hace el scroll en la zona del comienzo, y de paso os enseño otra función molona del ANSI C. La idea es rellenar esa parte con contenido nuevo, fresquito, refrescante, y nada mejor que poner sólo la parte del tile que nos hace falta. Pero claro, ¿Cómo ponemos sólo parte del tile? Pues con otras funciones que me he sacado de la chistera:

```
put_partial1h_sprite_x16 (puntero_sprite,x,y); // Imprime la parte izquierda del tile
put_partial2h_sprite_x16 (puntero_sprite,x,y); // Imprime la parte derecha del tile
put_partial1v_sprite_x16 (puntero_sprite,x,y); // Imprime la parte superior del tile
put_partial2v_sprite_x16 (puntero_sprite,x,y); // Imprime la parte inferior del tile
```

Estas funciones imprimen un tile de 16x16 pixels de forma parcial, de tal forma que os servirán para las cuatro direcciones posibles del scroll. Así no tendréis que preocuparos de hacer un buffer, mantenerlo actualizado, e ir pasando lo que corresponda a pantalla. Un ejemplillo rápido:

```
void main (void)
{
    unsigned char y,paso;
    cls (0); // Borramos la pantalla
    port_out (254,1); // Ponemos el borde de color azul
    paso=0;
    while (1)
    {
        for (y=18;y<23;y=y+2)
        {
            switch(paso)
            {
                case 0: put_partial1h_sprite_x16 (sprite_cubo, 31, y); break;
                case 1: put_partial2h_sprite_x16 (sprite_cubo, 31, y); break;
                case 2: put_partial1h_sprite_x16 (sprite_moneda, 31, y); break;
                case 3: put_partial2h_sprite_x16 (sprite_moneda, 31, y); break;
            }
        }
        if (paso>=3)
        {
            paso=0;
        }
        else
        {
            paso=paso+1;
        }
        delay(10);
        scroll_izquierda (0,18,32,6);
    }
}
```

}

Ala, hacéis un makezx y a disfrutar de un scroll sin imperfecciones.

Como es lógico os estaréis fijando en esa instrucción “if” que he metido, además de un “else” posterior. Lo que hace, al igual que el BASIC, es ejecutar lo que hay entre corchetes si se cumple la condición, en caso contrario ejecuta lo que hay en los corchetes del “else”. La parte de “else” no es obligatoria, podemos poner “if” sin el “else”, pero en este caso es evidente que nos viene bien para no tener que poner dos “if” con una condición similar. Recordad que todo “if” requiere consultar la condición, así que si con else os evitáis otra instrucción más hacedlo porque esa minúscula velocidad y memoria que ganáis os puede venir bien al final.

Para el caso de switch, como podéis más o menos deducir, sirve para ejecutar una acción para cada valor de una variable que evaluemos. En este caso hemos creado la variable “paso”, que vamos incrementando de forma consecutiva entre 0 y 3. Con “case” vamos definiendo cada caso, y metemos tantas instrucciones como deseemos. Al final ponemos un break, ya que en caso contrario se ejecutará la siguiente línea y se evaluará nuevamente un case. Fijaros en el ejemplo y veréis que no es tan complicado.

Lo bueno que tienen estas rutinas put_partial es que es sencillísimo usar un mapa con ellas si usas tiles de 16x16, por lo que vais a poder hacer juegos con scroll en la dirección que os dé la gana del tipo que queráis (Saimazoom Starbuck Edition, Zelda Blood Revenge, Tour de France Doping Race, etc).

Mapeando que es gerundio – Matrices

Ya sabemos colocar cosas en pantalla, al igual que hacer preciosos scroll de arbolitos. Pero en un juego siempre tenemos un mapa, a menos que sea un arcade, y es importante saber cómo hacerlos y manejarlos.

En primer lugar vamos a imaginarnos una hoja de papel cuadriculado. Ahora imaginad que cada cuadradito es un tile... efectivamente, eso. Un mapa no es sino una cuadrícula de tiles, un rectángulo repleto de tiles que podemos usar como habitaciones si agrupamos los tiles dentro de ese rectángulo en partes más grandes.

Vamos a suponer que tenemos en pantalla 3x3 tiles, algo sencillito para empezar, y que vamos a hacer un mapa de 3x2 pantallas. Eso nos da un rectángulo real de 9x6 tiles. Vamos, algo así:

```
1 1 1 2 2 2 3 3 3
1 1 1 2 2 2 3 3 3
1 1 1 2 2 2 3 3 3
4 4 4 5 5 5 6 6 6
4 4 4 5 5 5 6 6 6
4 4 4 5 5 5 6 6 6
```

Los números indican las habitaciones, y cada número es un tile. Eso, en matemáticas, se llama matriz, y en ANSI C podemos definir una matriz mediante una variable:

```
unsigned char mapa [] = {}
```

En este caso hemos definido el mapa como una matriz de bytes (unsigned char), por lo que cada byte se corresponderá con un tile. Es responsabilidad nuestra asignar un número a cada tile, lo que nos define un tileset. Un tileset no es más que un conjunto de tiles, agrupados, que servirán para dibujar el mapa en pantalla.

Como hemos usado un byte en el mapa para definir un tile sólo vamos a poder usar hasta 256. Tranquilos, es más que suficiente para cualquier juego de ZX Spectrum (los juegos de los Mojon Twins no suelen usar más de 48), pero si es necesario se pueden usar varios tileset en el mismo juego (por ejemplo un tileset para cada fase).

Lo suyo es que hagamos un ejemplo práctico. Vamos a usar varios sprites de 16x16 que tenemos ya definidos en los ficheros de ejemplo del curso y vamos a asignarles un número para crear un tileset:

- 1- sprite_negro
- 2- sprite_cubo
- 3- sprite_moneda

Ah, ya tenemos un tileset. Sólo hemos usado tres de los 256 posibles tiles, pero es que esto sólo es un ejemplo. Ahora vamos a definir un mapa, que va a ocupar una única pantalla, y será de 5x5 tiles:

```
unsigned char mapa [] = {1,1,1,1,1,1,0,0,0,1,1,0,2,0,1,1,0,0,0,1,1,1,1,1,1};
```

¡Ojo! Este mapa hay que definirlo fuera de la función MAIN, por lo que debemos meterlo o en un fichero aparte y añadirlo como #include o definir el mapa justo antes de MAIN.

El mapa es tal que así si ordenamos los números por filas y columnas:

```
1 1 1 1 1
1 0 0 0 1
1 0 2 0 1
1 0 0 0 1
1 1 1 1 1
```

Es decir, queremos dibujar como una sala con cuadrados alrededor y en el centro una moneda.

La rutina para dibujar esto sería:

```
unsigned char mapa [] = {1,1,1,1,1,1,0,0,0,1,1,0,2,0,1,1,0,0,0,1,1,1,1,1,1};
void main (void)
{
    unsigned char x,y,z;
    cls (7); // Borramos la pantalla
    port_out (254,0); // Ponemos el borde de color negro
    x=0;
    y=0;
    for (z=0;z<25;++z)
    {
        switch (mapa[z])
        {
            case 0: put_sprite_x16 (sprite_negro, x, y); break;
            case 1: put_sprite_x16 (sprite_cubo, x, y); break;
            case 2: put_sprite_x16 (sprite_moneda, x, y); break;
        }
        ++x; // Cada tile es de 2 caracteres de ancho
        ++x;
        if (x==10) // 10 caracteres es el ancho del mapa en pantalla
        {
            x=0;
            ++y; // Cada tile es de 2 caracteres de alto
            ++y;
        }
    }
    while (1)
    {
    }
}
```

Evidentemente un mapa no va a ocupar una única pantalla, pero para empezar os deja una idea de cómo crearlo y cómo dibujarlo.

¡Luces, cámara, acción! – Usando los puertos e/s, lógica de bits

Bueno, ha llegado el momento, es hora de coger todo lo que hemos visto y empezar a hacer un juego. En primer lugar vamos a mover un muñequito por la pantalla con las teclas y de paso a que suene un sonidito de pasos cuando eso suceda.

¿Cómo leemos el teclado? Si recordáis la sección del ZX Spectrum por dentro os dejé una bonita tabla donde ponía cada tecla y su puerto/bit correspondiente. También os comenté que dichos bits estaban a 1 si la tecla no estaba pulsada y a cero si lo estaban. Para las teclas Q/A/O/P sería:

| Tecla | Puerto | Bit |
|-------|--------|-----|
| Q | 64510 | 0 |
| A | 65022 | 0 |
| O | 57342 | 1 |
| P | 57342 | 0 |

En lógica de bits el bit cero tiene un valor 1, el bit 2 vale 2, el bit 3 vale 4, el bit 4 vale 8... y así sucesivamente. Pero mejor lo vemos con un ejemplo práctico y de paso os enseño a operar con bits:

```
unsigned char x,y;
cls (7); // Borramos la pantalla
x=10;
y=5;
while (1)
{
    if ((port_in(64510)&1)==0 && y>0) // Q
    {
        put_partial1v_sprite_x16 (sprite_negro,x,y+1);
        --y;
    }
    if ((port_in(65022)&1)==0 && y<22) // A
    {
        put_partial1v_sprite_x16 (sprite_negro,x,y);
        ++y;
    }
    if ((port_in(57342)&1)==0 && x<30) // P
    {
        put_partial1h_sprite_x16 (sprite_negro,x,y);
        ++x;
    }
    if ((port_in(57342)&2)==0 && x>0) // O
    {
        put_partial1h_sprite_x16 (sprite_negro,x+1,y);
        --x;
    }
}
```

```

        put_sprite_x16 (sprite_prota1, x, y);
    }

```

¡Pero cómo va esto! Si, al no poner pausa alguna esto va que se las pela, ahora lo arreglamos, pero os hace una idea de la velocidad que vais a tener disponible.

Empecemos a ver las cosas nuevas que han aparecido, centrémonos en la tecla Q:

```

if ((port_in(64510)&1)==0 && y>0) // Q
    {
        put_partial1v_sprite_x16 (sprite_negro,x,y+1);
        --y;
    }

```

Para empezar vemos que para leer el puerto hemos usado la instrucción:

```
port_in(64510)
```

Efectivamente, `port_in (puerto)` nos sirve para leer el puerto que queramos y nos devolverá un valor de lectura del mismo. Al ser un puerto de 8 bits el valor estará comprendido entre 0 y 255. Cada bit tiene un peso específico que viene dado por potencia de 2:

| Bit | Valor |
|-----|-------|
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |

Si sumáis todos los valores de los bits obtenéis 255. ¿Cómo saber entonces si un bit está activado o no? Pues realizando una operación AND (&) en lógica booleana. Es como multiplicar el valor de todo el byte por otro valor pero a nivel de bit, quiere decir que si realizamos un AND del byte con un valor, supongamos, de 16 el resultado será 0 o 16 (que el bit 4 está a 0 o a 1).

En nuestro ejemplo hemos hecho:

```
port_in(64510)&1
```

Esto hace que realicemos un AND entre el valor de lectura del puerto y el bit 0, obteniendo exclusivamente un valor 0 o positivo si dicho bit está a 0 o a 1.

Como también queremos comprobar que nos movemos dentro del área de pantalla (recordad, las rutinas gráficas son TONTAS), añadimos en el IF con && (es como decir Y) la condición lógica que si la coordenada vertical está dentro de los límites podemos efectuar el movimiento:

```
if ((port_in(64510)&1)==0 && y>0)
```


Y para remate, para no dejar rastro, dentro de la función usamos la conocida función de impresión parcial del sprite para reponer el fondo (en este caso negro).

```
put_partial1v_sprite_x16 (sprite_negro,x,y+1);
```

Llegados aquí se nos viene a la cabeza un par de mejoras, por un lado meter algo de pausa para que nos dé tiempo a ver algo, así que añadid las siguientes líneas dentro del WHILE justo al final:

```
wait_int();  
wait_int();  
wait_int();  
wait_int();
```

Si, nada menos que cuatro esperas al retraso vertical de pantalla. Esto nos viene haciendo ya la idea de cómo puede ser nuestro programa para estar correctamente sincronizado y que todo vaya bien:

Primer ciclo de pantalla: movemos nuestro personaje y las colisiones con el escenario

Segundo ciclo de pantalla: movemos los enemigos

Tercer ciclo de pantalla: vemos las colisiones con los enemigos

Cuarto ciclo: animación del escenario si lo hubiera

Es sólo una sugerencia, evidentemente cada tipo de juego tendrá sus particularidades.

Ahora vamos a meterle algo de animación a este personaje, para lo cual usamos otra variable, frame, que nos dirá que secuencia de animación vamos a poner cuando redibujemos el personaje:

```
if (frame==0)  
{  
    put_sprite_x16 (sprite_prota1, x, y);  
    frame=1;  
}  
else  
{  
    put_sprite_x16 (sprite_prota2, x, y);  
    frame=0;  
}
```

Recordad dar de alta frame al comienzo del listado, ponerlo a cero antes de empezar el WHILE, y colocar el código anterior en sustitución de la línea:

```
put_sprite_x16 (sprite_prota1, x, y);
```

El remate a todo esto sería crear otra variable, llamada Z, que indique si hay que redibujar el sprite o no, para minimizar el gasto de dibujado en pantalla (que es lo más costoso en un juego).

El código completo, con todas las modificaciones sería:

```

void main (void)
{
    unsigned char x,y,z,frame;
    cls (7); // Borramos la pantalla
    x=10;
    y=5;
    z=0;
    frame=0;
    put_sprite_x16 (sprite_prota1, x, y);
    while (1)
    {
        if ((port_in(64510)&1)==0 && y>0) // Q
        {
            put_partial1v_sprite_x16 (sprite_negro,x,y+1);
            z=1;
            --y;
        }
        if ((port_in(65022)&1)==0 && y<22) // A
        {
            put_partial1v_sprite_x16 (sprite_negro,x,y);
            z=1;
            ++y;
        }
        if ((port_in(57342)&1)==0 && x<30) // P
        {
            put_partial1h_sprite_x16 (sprite_negro,x,y);
            z=1;
            ++x;
        }
        if ((port_in(57342)&2)==0 && x>0) // O
        {
            put_partial1h_sprite_x16 (sprite_negro,x+1,y);
            z=1;
            --x;
        }
        if (z)
        {
            if (frame==0)
            {
                put_sprite_x16 (sprite_prota1, x, y);
                frame=1;
            }
            else
            {
                put_sprite_x16 (sprite_prota2, x, y);
                frame=0;
            }
        }
    }
}

```

```
        }
        z=0;
    }
    wait_int();
    wait_int();
    wait_int();
    wait_int();
}
}
```

¡Como molaaaaa! Pero nos queda el remate, el sonido.

Es hora de que veamos cómo modificar también los puertos de salida mediante:

`port_out (puerto, valor);`

Con esta función vamos a sacar un valor por un puerto. La gran pega es que modificamos todos los bits, no es posible activar o desactivar un bit en especial sin modificar el resto. Afortunadamente sólo nos va a interesar modificar el puerto dedicado a cambiar el color del borde y generar sonido, por lo que con inteligencia no vamos a tener problema.

Sonido – cómo crear FX

Para empezar os he dejado una minilibrería de soniditos básicos, que debéis incluir al principio del programa:

```
#include <lib\sonidoszx.h>
```

Con ella basta meter justo después del último `z=0` la siguiente instrucción:

```
sonido_andar();
```

¡Voilà! ¿Pero cómo se hace el sonido? Si abrimos la librería veremos que es la mar de sencillo:

```
void sonido_andar (void)
{
    port_out (254,16);
    port_out (254,16);
    port_out (254,0);
    port_out (254,16);
    port_out (254,16);
    port_out (254,0);
}
```

Lo que hacemos es cambiar el estado del bit 4 del puerto 254, que si recordamos es el que se dedica a sacar sonido por el “speaker”. Primero lo ponemos a 1 y luego a cero. Tan rápido que apenas nos retrasa la ejecución del programa. Esto equivale a genera una onda tal que así:



Como lo que generamos es una onda cuadrada de sonido podemos jugar y hacer que suene más flojito, cambiadlo por:

```
void sonido_andar (void)
{
    port_out (254,16);
    port_out (254,0);
}
```

Ahora veréis que los pasos son mucho menos fuertes, ese es el mínimo sonido que podemos realizar con un ZX Spectrum. Lo suyo es que probéis combinaciones y creéis vuestras propios FX. Un sonido personalizado es lo que le da el punto final a un juego e incluso en muchos casos le da un toque muy especial.

Más adelante veremos cómo se pueden crear notas, y con ello música.

El gran retro, nuestro primer juego

Ya sabemos:

- Movernos por la pantalla
- Hacer un mapa
- Dibujar lo que nos venga en gana
- Hacer sonidos

¿A qué esperamos? Pongámonos manos a la obra y hagamos un juego.

Para empezar hay que pensar en una historia:

“Grub, el magnífico, ha perdido toda su fortuna en manos de los Gorp. Le han robado todo su dinero y se enfrenta al difícil reto de recuperar lo que es suyo. Así que ánimo y trata de encontrar las 10 monedas que hay repartidas por el mapa.”

Ahora tenemos que pensar en qué tipo de juego queremos, y vamos a pensar en uno de exploración en plan Sabre Wulf.

Con el juego en mente tenemos que crear un mapa... pero un mapa grande implica que a mano nos podemos tirar horas haciéndolo, así que vamos a usar un editor que nos lo genere.