



Hazte un Spectrum - Capítulo 1

Últimamente me ha dado (no sé por qué) por el tema “retrocomputing”. Quizá es que me estoy haciendo mayor y me ataca la nostalgia, o que simplemente la forma de hacer ahora las cosas es aburrida. Como en mi día fui “de los del Commodore 64”, me decidí por aprender cómo era el enemigo de por entonces: el ZX Spectrum. Y a hacerme uno.

Debo decir que si alguien piensa que los ordenadores de los '80 están olvidados, nada más lejos de la realidad: hay ingentes cantidades de información ahí afuera, incluso mucho mejores que durante la época dorada de estos aparatos. Para mi proceso de aprendizaje he encontrado absolutamente fundamental la información que se puede encontrar en WorldOfSpectrum.org principalmente.

Además me di cuenta de que iba a emplear varios lenguajes de programación, cada uno perfecto para una parte del desarrollo: desde C++ para los componentes de bajo nivel del emulador (donde la velocidad es primordial) a .NET y WPF para un “frontend” de usuario. Con esta idea en mente me propuse desarrollar todo esto de la forma más clara posible, para mostraros cómo mezclar herramientas, lenguajes y conceptos para aprovechar lo máximo de cada uno de ellos. Comenzaremos con una emulación de una cpu Z80 en C++, junto con los componentes hardware que componen un Spectrum, para convertirlo todo en un componente .NET reutilizable usando C++ gestionado, para terminar con programación C# y WPF para el programa final. Manos a la obra.



Necesito una CPU...

Más que nada, porque sin un procesador Z80 no vamos a ningún sitio. Afortunadamente también hay ingentes cantidades de información sobre este componente. No tuve que rebuscar mucho para encontrar el datasheet de Zilog y empezar a echarle un vistazo.

Acostumbrado como estaba a los procesadores 6502 y similares del Commodore Vic20 y 64, para mi el Z80 era simplemente un “conocido”, así que tras una primera lectura de la documentación, me puse manos a la obra para desarrollar un emulador de ésta CPU.

Antes de empezar con la parte “dura” de la emulación, pensé en cómo accedería la CPU a los elementos externos a la misma, como por ejemplo la memoria, la ROM y los dispositivos de entrada/salida. Tras darle algunas vueltas, decidí que los buses del procesador (de datos y de entrada/salida) serían realmente instancias de clases especializadas, así que para la emulación podría olvidarme de cómo leer la RAM o los dispositivos y centrarme únicamente en la emulación del procesador en sí. También tuve que decidir qué lenguaje utilizar y, para variar, me decanté por C++.

El primer paso es definir “algo” que me permita leer y escribir bytes en un bus, y a ésta clase la llamé (original que es uno) `BusComponentBase`:

```
class BusComponentBase
{
public:
    virtual unsigned int GetStartAddress() = 0;
    virtual unsigned int GetRegionSize() = 0;
    virtual void Write(unsigned int address,unsigned char value) = 0;
    virtual unsigned char Read(unsigned int address) = 0;
};
```

La clase no provee ninguna implementación para las funciones, así que cada componente puede hacerlo de la forma que mejor le venga. `GetStartAddress()` devuelve la dirección base dentro del bus del componente, y `GetRegionSize()` devuelve el tamaño del bloque dentro del bus. Las funciones `Read()` y `Write()` se encargan de las operaciones de lectura y escritura.

El siguiente paso es hacer una versión un poco más especializada de la función, aprovechando los templates de C++ e implementar en tiempo de compilación los tamaños y las direcciones base de los componentes:

```
template<unsigned int B,unsigned int S> class BusComponent : public BusComponentBase
{
public:
    virtual unsigned int GetStartAddress() { return(B); }
    virtual unsigned int GetRegionSize() { return(S); }
};
```

Y continuando con la programación con templates, definimos nuestro primer componente: un bloque de memoria RAM:

```
template<unsigned int B,unsigned int S> class RAM : public BusComponent<B,S>
{
public:
    RAM()
    {
        ZeroMemory(data,S);
    }
    void Write(unsigned int address,unsigned char value)
    {
        data[address-B] = value;
    }
    unsigned char Read(unsigned int address)
    {
        return(data[address-B]);
    }

protected:
    unsigned char data[S];
};
```

Al usar templates para el tamaño y la base, ya no es necesario reservar memoria dinámicamente. Simplemente usamos los argumentos del template para crear el bloque de memoria (unsigned char [S]) y acceder a los contenidos (data[address-B] = value).

Ya tenemos los bloques básicos que se pueden conectar a un bus, pero nos falta definir el "bus" mismo... Y siguiendo con la analogía de cómo funcionan los SoCs ("System On Chip") que incluyen buses internos que a su vez agrupan diversos dispositivos, los buses también derivan de BusComponent, solo que incluyen funciones para "añadir" y "eliminar" componentes del bus:

```
template<unsigned int B,unsigned int S> class Bus : public BusComponent<B,S>
{
public:
    HRESULT AddBusComponent(BusComponentBase *newComponent)
    {
        for (unsigned int dd=0;dd<m_components.size();dd++)
        {
            if (m_components.at(dd) == newComponent)
                return(ERROR_ALREADY_EXISTS);
        }
        m_components.push_back(newComponent);
        OnComponentsUpdated();
        return(S_OK);
    }

    HRESULT RemoveBusComponent(BusComponentBase *component)
    {
        for (unsigned int dd=0;dd<m_components.size();dd++)
        {
            if (m_components.at(dd) == component)
            {
                m_components.erase(m_components.begin()+dd);
                OnComponentsUpdated();
                return(S_OK);
            }
        }
        return(ERROR_NOT_FOUND);
    }

protected:
    virtual void OnComponentsUpdated()
    {
    };

protected:
    vector<BusComponentBase*> m_components;
};
```

Sólo queda hacer una clase que implemente un bus de 16 bits para acceder a los 64KB de direccionamiento del Z80. Para optimizar la cosa, la implementación crea una tabla de 64 entradas, cada una de ellas apuntando al componente que gestiona "ese kilobyte" de almacenamiento:

```
class Bus16 : public Bus<0x0000,0x10000>
{
public:
    Bus16()
    {
```

```

        ZeroMemory(m_pagedComponents,sizeof(m_pagedComponents));
    }

public:
    void Write(unsigned int address,unsigned char value)
    {
        BusComponentBase *pComp = m_pagedComponents[(address & 0xFFFF) / 1024];
        if (pComp)
            pComp->Write(address,value);
    }
    unsigned char Read(unsigned int address)
    {
        BusComponentBase *pComp = m_pagedComponents[(address & 0xFFFF) / 1024];
        return(pComp ? pComp->Read(address) : 0xFF);
    }

protected:
    virtual void OnComponentsUpdated()
    {
        // Build a quick lookup table por each component. Address
        // space is split in 1 KiloByte segment (64 entries).
        for (unsigned int dd=0;dd<m_components.size();dd++)
        {
            BusComponentBase *pComp = m_components.at(dd);
            if (pComp)
            {
                unsigned int start = pComp->GetStartAddress() / 1024;
                unsigned int end = start + (pComp->GetRegionSize() / 1024);
                for (unsigned int zz = start;zz<end;zz++)
                    m_pagedComponents[zz] = pComp;
            }
        }
    };

protected:
    BusComponentBase* m_pagedComponents[64];
}

```

La clase deriva del template `Bus<0x0000,0x10000>`, esto es, un bus que empieza en la dirección "0" y termina en "0xFFFF". El array `m_pagedComponents` se actualiza cada vez que se añade o elimina un componente del bus, y cada entrada apunta al bloque asignado a cada kilobyte. Decidir a qué componente enviar la petición de lectura o escritura es tan simple como

```
BusComponentBase *pComp = m_pagedComponents[(address & 0xFFFF) / 1024];
```

Con nuestros buses y bloques de RAM en el arsenal, ya podemos empezar a implementar nuestra flamante CPU...

Cómo es un Z80 por dentro

El procesador Z80 era extraordinariamente complejo comparado con los 6502 de la época. Incluye la escandalosa (para la época) cantidad de 16 registros de 16 bits cada uno, y para muchos de dichos registros permite accederlos por "mitades", pudiendo trabajar con la parte superior o inferior de cada uno de ellos de forma individual. Por ejemplo, el registro de 16 bits HL puede usarse "entero" (por ejemplo para leer una posición de memoria con `"LD A, (HL)"`) o bien su mitad superior o inferior individualmente (por ejemplo `"INC H"` ó `"DEC L"`). El registro AF es incluso "peor": el byte alto del registro ("A") es el acumulador del procesador, esto es, la unidad en la que se efectúan las operaciones matemáticas. El byte menos significativo ("F") contiene los "flags" o "banderas" del procesador, que indican el estado en que se encuentra en cada momento. Concretamente, el registro F contiene los siguientes campos, cada uno de ellos un bit:

A	F	AF'	I	R
B	C	BC'	IX	
D	E	DE'	IY	
H	L	HL'	SP	
			PC	

S	Z	-	H	-	P	N	C
---	---	---	---	---	---	---	---

El bit 'S' indica si la última operación realizada por el procesador generó un resultado positivo o negativo. El bit 'Z' indica que la última operación dió resultado cero. El bit 'H' ("Half Carry") indica si la última operación rebasó la capacidad de los cuatro bits inferiores del registro y se usa para operaciones aritméticas en "Base 10". 'P' indica bien la paridad (número de bits a "1") de la última operación o si la operación excedió los límites (overflow). Y finalmente, el bit 'C' indica si la última operación generó acarreo ("Carry" en la jerga).

Y vista esta estructura, me alegré de haber escogido C++ para implementar el procesador. Para los programadores en C#, implementar cualquiera de los registros del Z80 implicaría el uso de montones de "getters" y "setters", para acceder a las mitades superior e inferior de cada registro. En C++ se usan "unions".

Un "union" es una forma especial de estructura, donde los miembros no están "uno detrás de otro", sino "uno encima de otro". De esta forma se puede leer y escribir datos en formatos diferentes. Por ejemplo, el registro HL del Z80 puede codificarse como sigue:

```

struct Z80Registers
{
    ....
    // HL register pair
    union

```

```

{
    unsigned short HL;
    struct
    {
        unsigned char L;    // LSB
        unsigned char H;    // MSB
    };
};
....
};

```

al definir el registro HL empleamos un union sobre un “unsigned short” (16 bits) y una estructura que contiene dos “unsigned char” (bytes), que ocupan (“union”) las MISMAS posiciones de memoria. De esta forma se puede escribir “Z80Registers.HL”, o “Z80Registers.H” o “Z80Registers.L”. No hay getters, no hay setters, no hay ciclos de cpu desperdiciados para desmontar y montar variables: el compilador lo hace todo.

El registro AF tiene más gracia todavía:

```

struct Z80Registers
{
    // AF register pair
    union
    {
        unsigned short AF;
        struct
        {
            union
            {
                unsigned char F;    // LSB
                struct
                {
                    unsigned char CF : 1;
                    unsigned char NF : 1;
                    unsigned char PF : 1;
                    unsigned char XF : 1;
                    unsigned char HF : 1;
                    unsigned char YF : 1;
                    unsigned char ZF : 1;
                    unsigned char SF : 1;
                };
            };
            unsigned char A;    // MSB
        };
    };
};

```

Gracias a esta estructura y el uso de bitfields de C++, podemos acceder individualmente al registro AF completo (“Z80Registers.AF”), al acumulador del procesador (“Z80Registers.A”), al conjunto de flags (“Z80Registers.F”) o a cualquier flag individual sin necesidad de recurrir a ninguna operación de máscara de bits (“Z80Registers.CF”).

Con nuestra declaración de los registros de un procesador Z80 y nuestros buses, podemos empezar a construir el procesador:

```

class Z80
{
public:
    Z80()
    {
        ZeroMemory(&regs,sizeof(regs));
        DataBus = NULL;
        IOBus = NULL;
        tStates = 0;
    }

public:
    void EmulateOne();

public:
    BusComponent<0x0000,0x10000>* DataBus;
    BusComponent<0x0000,0x10000>* IOBus;
    Z80Registers regs;
    unsigned int tStates;
};

```

El procesador tiene dos buses, uno para la memoria (DataBus) y otro para I/O (apropiadamente denominado IOBus). Incluye un juego de registros (regs) y una variable que contará cuantos ciclos de reloj lleva emular cada instrucción (y no es coña). La función EmulateOne() se encargará de emular una instrucción en nuestro procesador.

El siguiente paso

En el próximo capítulo de éste tutorial comenzaremos a implementar el emulador de instrucciones de nuestro procesador Z80. Iremos instrucción por instrucción y tendremos una clase que encapsula, más o menos con precisión, el procesador que dará vida a nuestro Spectrum.



Hazte un Spectrum - Capítulo 2

En la [primera parte de este tutorial](#) establecimos una arquitectura básica para nuestro emulador de ZX Spectrum: teníamos un procesador y unos buses, incluso tenemos un componente que emula memoria RAM. Ahora toca ejecutar las instrucciones del procesador.

Y debo decir que tras la primera lectura del datasheet del procesador Z80 (que podéis [leer directamente aquí](#) en formato PDF) la cosa no pintaba tan bien como podía esperar en un principio.

Acostumbrado como estaba al sencillísimo código máquina de los procesadores 6502 que conocía de la época (Commodore Vic 20 y Commodore 64), la estructura del Z80 era extraordinariamente compleja. El 6502 tiene un juego de instrucciones codificado en 8 bits, e incluso no todas las combinaciones (de 0 a 255) eran instrucciones válidas. En cambio, el Z80 no solo emplea instrucciones de 8 bits, sino que reconoce hasta cuatro “prefijos” para ejecutar instrucciones adicionales. Los prefijos 0xDD y 0xFD hacen que el procesador cambie la interpretación de las instrucciones para emplear un direccionamiento basado en los registros IX e IY, mientras que los prefijos 0xCB y 0xED abren dos nuevas páginas completas de instrucciones. Para colmo, las instrucciones con prefijo 0xDD y 0xFD también permiten el uso del prefijo 0xCB, complicando más si cabe la interpretación de las instrucciones del Z80.

Necesito una plantilla

Como soy consciente de que ni seré el primero ni el último que va a escribir un emulador de un procesador Z80 (los hay a montones), me puse a buscar si ya había algún código que me sirviese de “plantilla” para mi emulador. Empezando por el del archiconocido [MAME](#) (“Multi Arcade Machine Emulator”), emulador de videojuegos que soporta montones de plataformas, rápidamente lo dejé de lado al estar fuertemente basado en macros que harían complicado reescribir el código y, sobre todo, lo harían difícilmente portable a otros lenguajes. Eché un vistazo a [FUSE](#), un emulador de Spectrum, pero tampoco me convenció su estructura, pero entre ambas opciones tenía una lista, una por una, de todas las instrucciones del procesador Z80... incluso de las no documentadas.

Hay más de lo que dice Zilog

De la lectura de montones de código fuente de emulación del Z80, me maravilla la precisión con la que emulan las instrucciones no documentadas del mismo. Si, éstas instrucciones son “accidentes” en el desarrollo del procesador.

En un lenguaje de programación, las instrucciones están perfectamente documentadas, son esas y no hay más. Luego hay gente con más o menos ingenio para combinarlas buscando efectos concretos, pero no hay nada oculto. En cambio, cuando hablamos de procesadores, cada instrucción lo que realmente hace es activar o desactivar ciertas partes de la electrónica interna, y los fabricantes documentan las combinaciones diseñadas a propósito. Pero hay muchas combinaciones que no están documentadas, y hay gente con un extraordinario talento dispuesta a descubrirlos, uno a uno, estudiando todos los comportamientos del procesador tras ejecutar todas y cada una de las instrucciones que no aparecen en la documentación oficial. Algunos ejemplos? Sólo hay una instrucción NOP oficial, pero los opcodes 0x77 y 0x7F también son NOPs con la curiosa diferencia que consumen 8 ciclos de reloj en lugar de los 4 habituales. O la instrucción NEG, que oficialmente es 0x44. Las instrucciones 0x4C, 0x54, 0x5C, 0x64, 0x6C, 0x74 y 0x7C hacen exactamente lo mismo.

Pero, e instrucciones nuevas? También las hay. La instrucción SLI no existe en el datasheet de Zilog, pero contiene todas las variantes de direccionamiento de cualquier instrucción oficial, y es una mutación de la instrucción oficial SLA. La instrucción SLA desplaza los contenidos del acumulador (registro A) a la izquierda, el bit que sobra se envía al flag de acarreo (carry) e inserta un cero por la derecha. La función SLI hace exactamente lo mismo, pero insertando un 1 a la derecha. Imagino que cuando Zilog estaba desarrollando el procesador, pensaron que ésta instrucción no tendría utilidad práctica alguna, y no se molestaron en documentarla... O ni siquiera se dieron cuenta de su existencia. Pero es realmente útil para “scrolls” horizontales en videojuegos...

Y precisamente por eso toca implementarlas: aunque no estén en la documentación oficial, los programadores se han acostumbrado a usarlas y, de hecho, muchas de ellas tienen una utilidad más que práctica.

Hay otra cosa que, la verdad, no he considerado para emular: los dos bits que “faltan” del registro F del procesador (los “flags”) que ya vimos en el primer capítulo:

Veis los dos “flags” con un guión en lugar de una letra? Bueno, pues también están documentados, gracias al trabajo de mucha gente que ha analizado con un detalle exquisito cómo se alteran tras diferentes operaciones en el procesador. Los llaman “Flag X” y “Flag Y” y en la mayoría de los casos son los bits colocados en esa posición del resultado de una operación aritmética.

Como mi intención es hacerme un Spectrum para, sobre todo, aprender cómo era por dentro, no voy a incluir esta funcionalidad (por el momento). Como ya podréis comprobar, simplemente implementando las instrucciones no documentadas basta para que funcione la mayoría de los programas. Y creedme: hay muchísimas cosas más para emular mucho más importantes que esos dos flags para que algunos programas funcionen!! Lo que sí tenía por seguro cuando empecé, es que la ROM original del Spectrum no usa ninguna de estas características... buen punto para comenzar.

Instrucciones primarias

Volviendo a nuestra implementación de Z80, el primer paso es decodificar las instrucciones “directas” (sin prefijos) del procesador Z80, y lo haremos en la función `EmulateOne()` que definimos en la primera parte de este tutorial. La función tiene más o menos este aspecto:

```
void Z80::EmulateOne()
{
    // Fetch next instruction
    unsigned char op = DataBus->Read(regs.PC++);
    // Increment instruction counter register
    regs.R60++;

    switch(op)
    {
        // nop
        case 0x00:
            tStates += 4;
            break;

        // ld bc,NN
        case 0x01:
            tStates += 10;
            regs.C = DataBus->Read(regs.PC++);
            regs.B = DataBus->Read(regs.PC++);
            break;

        // ld (bc),a
        case 0x02:
            tStates += 7;
            DataBus->Write(regs.BC,regs.A);
            break;

        // inc bc
        case 0x03:
            tStates += 6;
            regs.BC++;
            break;
        ...
        ...
    };
}
```

Lo primero que hace la función es leer (empleando el bus de datos del procesador) el siguiente opcode a interpretar. Para ello emplea el registro PC (“Program Counter”), que indica la posición de memoria de la siguiente instrucción a ejecutar e incrementa el contador:

```
unsigned char op = DataBus->Read(regs.PC++);
```

Una vez que tenemos el opcode a emular, incrementamos el registro R del procesador que consiste, básicamente, en un contador de instrucciones y que se emplea en el hardware real para refrescar memoria dinámica. El registro R realmente está dividido en dos secciones: el bit más significativo (“R7”) que siempre está a cero y los bits 6-0 que contienen el contador propiamente dicho. Gracias al uso de los bitfields de C++ incrementamos solamente la parte menos significativa del registro (“R60++”).

A continuación comenzamos el bloque switch que ejecuta cada una de las instrucciones. La primera es fácil: el opcode 0x00 del Z80 es la instrucción `NOP`:

```
// nop
case 0x00:
    tStates += 4;
    break;
```

La instrucción `NOP` en cualquier procesador no hace nada, simplemente consume tiempo. Y concretamente en el procesador Z80 consume cuatro ciclos de reloj, de ahí la instrucción `tStates += 4`.

Vamos a por el opcode 0x01, que equivale a la instrucción `LD BC,#valor`. Para emular esta instrucción usaremos el bus de datos del procesador para leer dos bytes que cargaremos en las mitades baja y alta del registro:

```

...
// ld bc,NN
case 0x01:
    tStates += 10;
    regs.C = DataBus->Read(regs.PC++);
    regs.B = DataBus->Read(regs.PC++);
    break;
...

```

El primer paso es acumular cuantos ciclos de reloj necesita la instrucción, para luego proceder a ejecutarla. la primera línea lee el byte siguiente en el flujo de instrucciones y lo asigna a la mitad C del registro BC. La siguiente línea hace lo mismo con la parte alta.

La primera ventaja obvia del uso de uniones a la hora de declarar la estructura de registros del procesador llega al emular el opcode 0x03, `INC BC`:

```

...
// inc bc
case 0x03:
    tStates += 6;
    regs.BC++;
    break;
...

```

Esta instrucción incrementa el valor de 16 bits contenido en el registro BC. Simplemente incrementando la variable `regs.BC` tenemos el trabajo hecho: las mitades B y C usan las mismas posiciones de memoria, así que su valor se verá instantáneamente actualizado.

Operaciones aritméticas

Aquí es donde la cosa se complica. Aunque pueda parecer sencillo que sumar dos valores no tiene la menor complicación, a nivel de una cpu sí lo tiene. La cpu debe mantener una serie de banderas (o flags) que afectan a la operación actual o que, una vez cambiadas por una operación, afectan a la operación siguiente. Pongamos el ejemplo del bit de acarreo (o “carry” en la jerga): el bit de acarreo es el equivalente en máquinas a nuestra tradición “y me llevo x”, aunque siendo máquinas y funcionando en binario, éste “x” sólo puede tener dos valores: 0 y 1.

Por ejemplo, si sumamos 5 y 7 el resultado (en base 10) sería algo así como “12” o “2 y me llevo 1”. Ese “1” se añade a la siguiente operación. En nuestro caso, el procesador Z80 puede sumar números de 8 bits. Si sumamos (por ejemplo) 170 (en hexadecimal 0xAA) y 131 (0x83) resulta en 301 (0x12D), pero que truncado a 8 bits (0x2D) resulta en 45 y el bit de acarreo en “1”.

Como hay montones de operaciones aritméticas en el Z80, prefería escribir funciones específicas para cada operación, como por ejemplo la de “suma con acumulador”:

```

void Z80::ADC_R8(unsigned char v)
{
    unsigned short aux = regs.A + v + (regs.CF ? 1 : 0);
    unsigned char idx = ((regs.A & 0x88) >> 3) | ((v & 0x88) >> 2) | ((aux & 0x88) >> 1);
    regs.A = (unsigned char)aux;
    regs.SF = (regs.A & 0x80) != 0;
    regs.ZF = (regs.A == 0);
    regs.HF = halfcarryTable[idx & 0x07];
    regs.PF = overflowTable[idx >> 4];
    regs.NF = 0;
    regs.CF = (aux & 0x100) != 0;
}

```

La función suma el valor v al acumulador del procesador, y añade “+1” si el acarreo estaba activo (regs.CF). A continuación actualiza el estado del resto de los flags del procesador con el resultado de la operación, como por ejemplo el flag “Z” (regs.ZF) que indica si el resultado de la operación ha sido cero.

Luego, desde la función `EmulateOne()` empleamos estas funciones para realizar las operaciones aritméticas. Por ejemplo, la instrucción `ADC A,C` (que suma el contenido del registro C al acumulador) la emulamos así:

```

...
// adc a,c
case 0x89:
    tStates += 4;
    ADC_R8(regs.C);
    break;
...

```

A la hora de implementar los bloques de instrucciones 0xDD y 0xFD, utilicé una sola función que emplea un registro “extra” inventado en nuestro procesador. Los prefijos en cuestión activan un modo de direccionamiento que emplea el registro IX (para 0xDD) ó IY (para 0xFD). La función `EmulateOneXX()` procesa los dos juegos de instrucciones, pero en lugar de hacerlo sobre los registros IX ó IY lo hace sobre un registro ficticio llamado “XX”. En la función `EmulateOne()` el código es sencillo:

```

// IX register operations
case 0xDD:

```

```

        regs.XX = regs.IX;
        EmulateOneXX();
        regs.IX = regs.XX;
        break;
    ...
    // IY register operation prefix
    case 0xFD:
        regs.XX = regs.IY;
        EmulateOneXX();
        regs.IY = regs.XX;
        break;

```

Para el resto de prefijos, el trabajo es similar. Hay una función para emular las instrucciones con prefijo 0xCB (`EmulateOneCB()`), otra para emular las instrucciones con prefijo 0xED (`EmulateOneED()`) y además de la ya conocida `EmulateOneXX()` existe otra para emular los prefijos 0xDD+0xCB y 0xFD+0xCB llamada `EmulateOneXXCB()`. Lo dicho: una risa.

Avanzamos en el tiempo y...

Los fundamentos para emular las instrucciones de un procesador son básicos, aunque algunas cpus puedan ser más simples que otras. En el caso del Z80 el trabajo necesario para emular las instrucciones es significativo, pero sin mucha mayor complicación. Instrucciones para emular las operaciones aritméticas y lógicas del procesador, alguna que otra función “helper” para acceder a la memoria y ya casi, casi lo tenemos. Pero me gustaría probarlo!!

Para ello nada más fácil (hombre, visto lo visto) que escribir un pequeño programa de pruebas, que construya un pequeño “ordenador virtual”, cargue algunas instrucciones en su memoria y las ejecute.

Lo primero, vamos a crear la máquina que ejecutará nuestro código:

```

int _tmain(int argc, _TCHAR* argv[])
{
    printf("Z80 cpu tests\n");

    // Components for tests
    Z80 cpu;           // processor
    RAM<0,1024> ram;   // 1KB of RAM
    Bus16 dataBus;     // Main data bus

    // attach components to bus
    dataBus.AddBusComponent(&ram);

    // attach bus to cpu
    cpu.DataBus = &dataBus;
}

```

Ya tenemos una máquina con un procesador Z80, 1 kilobyte de RAM (a que se hace raro?) en la dirección 0, todo ello unido por su bus de datos. De momento el bus de IO se queda sin inicializar, así que si queréis probar opcodes de entrada/salida, tendréis que instanciar un nuevo bus y asignarlo al miembro `IOBus` del procesador.

Ahora vamos a cargar un minúsculo programa en código máquina en nuestro ordenador virtual. Y, por cierto, si alguien es tan carca como yo, y esto le suena a los típicos “DATA”s de listados en código máquina de los ‘80, bienvenido al club.

```

unsigned char testProg[] =
{
    0x21, 0x00, 0x02,      // 0x0000 LD HL,#0200
    0x34,                 // 0x0003 INC (HL)
    0xC3, 0x03, 0x00      // 0x0004 JP 0003
};

// Load test program at RAM start
// This test program simply updates contents of
// memory location 0x0200
unsigned short ptr = 0;
for (int dd=0;dd<sizeof(testProg);dd++)
    cpu.DataBus->Write(ptr++,testProg[dd]);

```

Ya puestos, y en un afán de seguir probando cosas, inicializamos el contenido de la posición de memoria 0x200 a “algo”:

```

cpu.DataBus->Write(0x0200,0x23); // Set initial value

```

Si os fijáis, en lugar de escribir directamente en el bloque de memoria RAM (con `ram.Write()`), escribimos el programa usando el bus de datos del procesador directamente. Esto, en el mundo real, se llama DMA. La utilidad práctica es que no necesitamos saber “qué hay” en la posición de memoria, o cómo se usa, o si tiene hardware asociado que haya que modificar: la estructura de buses reenvía la información al dispositivo encargado de manejarla y éste puede optar por hacer cualquier cosa con él.

Volviendo al programa de pruebas, no hace gran cosa (normal con ese tamaño, no?). Simplemente carga el registro HL con 0x200 y luego comienza un bucle que incrementa esa posición de memoria. Para hacerlo más divertido, mostraremos por pantalla el estado de algunos registros (y el contenido de la dirección de memoria 0x200) a medida que el procesador ejecuta las instrucciones.

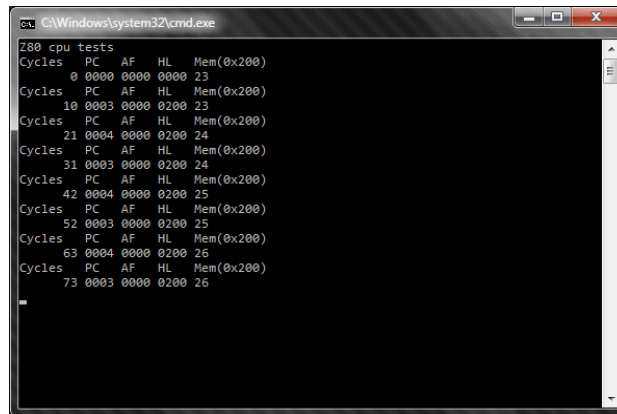

```

cpu.regs.PC = 0x0000;
int key = 0;
do
{
    printf("PC   AF   HL   Mem(0x200)\n");
    printf("%04X %04X %04X %02X\n",
        cpu.regs.PC,cpu.regs.AF,
        cpu.regs.HL,ram.Read(0x0200));

    key = _getch();
    if (key == 0x20)
        cpu.EmulateOne();
} while(key == 0x20);           // repeat while Space key pressed

```

Cuando ejecutamos nuestro flamante “emulador” (ejem) veremos algo parecido a lo siguiente:



```

Z80 cpu tests
Cycles PC AF HL Mem(0x200)
0 0000 0000 0000 23
Cycles PC AF HL Mem(0x200)
10 0003 0000 0200 23
Cycles PC AF HL Mem(0x200)
21 0004 0000 0200 24
Cycles PC AF HL Mem(0x200)
31 0003 0000 0200 24
Cycles PC AF HL Mem(0x200)
42 0004 0000 0200 25
Cycles PC AF HL Mem(0x200)
52 0003 0000 0200 25
Cycles PC AF HL Mem(0x200)
63 0004 0000 0200 25
Cycles PC AF HL Mem(0x200)
73 0003 0000 0200 25

```

La primera sección muestra el estado de la máquina antes de ejecutar nada: el contador de ciclos es cero, al igual que el contenido de los registros. La posición de memoria 0x200 contiene 0x23, que es el valor inicial que asignamos al inicializar la máquina.

Si vamos pulsado la barra espaciadora, la cpu irá ejecutando instrucciones con cada llamada a `EmulateOne()`. Con la primera de ellas veremos como el registro HL pasa a tener el valor 0x200 (`LD HL,#0200`), y en ello ha consumido 10 ciclos de reloj y el contador de programa (PC) ha avanzado hasta la posición de memoria 0x0003. La siguiente instrucción ejecuta `INC (HL)`, y lo veremos reflejado en la columna donde aparece el contenido de la memoria, mientras que el contador de programa avanza a la dirección 0x0004. La siguiente instrucción ejecuta un salto de nuevo a la posición 0x0003 para repetir el proceso. El registro AF aparentemente siempre está a cero: no es el caso. Si seguimos simulando instrucciones y (eventualmente) el contenido de la posición de memoria 0x200 sigue aumentando, llegará a 0x2F y luego pasará a 0x30: el flag ‘H’ (HalfCarry) indica que ha habido acarreo en los cuatro bits menos significativos del valor (de ‘F’ hexadecimal ha pasado a ‘0’). Esto ocurre cada pasada de los cuatro bits inferiores de ‘F’ a ‘0’: de 0x2F a 0x30, de 0x3F a 0x40, etc.

Si continuamos incluso más, el contenido de la posición de memoria 0x200 llegará a 0x7F y se incrementará a 0x80. Realmente 0x80 puede interpretarse de dos modos: sin signo, como “128” o con signo, que equivale a “-128”. En la transición de 0x7F a 0x80 varios flags se activarán en el registro F (parte baja de AF), concretamente los bits 7 (flag ‘S’), 4 (flag ‘H’) y 2 (flag ‘P’) dando un resultado de 0x94. El flag ‘S’ indica que el número es negativo, el flag ‘H’ indica el ya conocido efecto de que los 4 bits menos significativos del valor han desbordado y el flag ‘P’ (o ‘P/V’ para ser mas exactos) indica overflow en la operación. Si avanzamos hasta que el valor de la posición de memoria avanza hasta 0x81 veremos como el flag ‘S’ sigue activo, pero los flags ‘H’ y ‘P’ se ponen a cero: ya no hay overflow, ni acarreo en los 4 bits menos significativos del valor.

Todo este baile de banderas es la base de la programación en código máquina, en cualquier procesador: en código máquina no se “comparan variables con valores”, sino que la comparación es una operación en sí, que afecta a los flags del procesador. Luego hay otras instrucciones que actúan (o no) dependiendo del valor de dichos flags. Y si, también se puede “comparar” (en caso del Z80 con la instrucción `CP`) pero lo que realmente hace es “restar un valor al acumulador” y activar el flag ‘Z’ si el resultado es cero.

Más ejemplos: para hacer un bucle de 100 a 0 (los bucles descendentes son más efectivos en código máquina), se escribiría como “LD B,100; (bucle) DEC B; JP NZ,(bucle)”. No hay una comparación expresa: la instrucción JP NZ sigue saltando mientras el flag ‘Z’ no indique que el resultado de la operación fue cero. Un bucle de 0 a 127 sería algo así como “LD B,0; (bucle) INC B; JP P,(bucle)”, donde “JP P” significa “salta si el signo (flag ‘S’) es cero” o lo que es lo mismo “salta si el número es positivo”.

Siguiente paso

Ya tenemos un procesador que funciona y los buses necesarios para conectar componentes al mismo. Empieza el trabajo de emular otra cosa: el hardware del Spectrum! Mientras tanto, el código fuente tanto de nuestro flamante procesador Z80 y del programa de pruebas lo podéis [descargar desde aquí](#).



Hazte un Spectrum - Capítulo 3

En las dos primeras partes de éste tutorial ha quedado definida una arquitectura de un procesador Z80 simulado y de los componentes que permiten la conexión de nuestro procesador a diversos componentes a través de buses. Hoy empezamos a construir el componente hardware que convierte nuestra plataforma virtual en un Spectrum: el chip ULA.

El chip ULA del Spectrum es lo que da la “personalidad” de la máquina: define cómo se gestionan los gráficos, cómo se genera la señal de vídeo que vemos en el monitor (perdón... “televisor”), cómo se genera el sonido, cómo leer el teclado... Esta es, con diferencia, la parte que más me atraía de éste apasionante rato de aprendizaje.

Para empezar, lo primero es hacer un poco de memoria y recordar cómo estaba construido un Spectrum. En la época era impensable diseñar algo ni remotamente parecido a lo que hoy se considera una estructura de ordenador “moderna”: una cpu que habla con un chip “northbridge” que se encarga de redistribuir las señales a los distintos componentes del hardware, ajustando sus diferentes velocidades, anchos de bus y, a ser posible, sin atascar al procesador esperando a que dispositivos lentos terminen su trabajo.

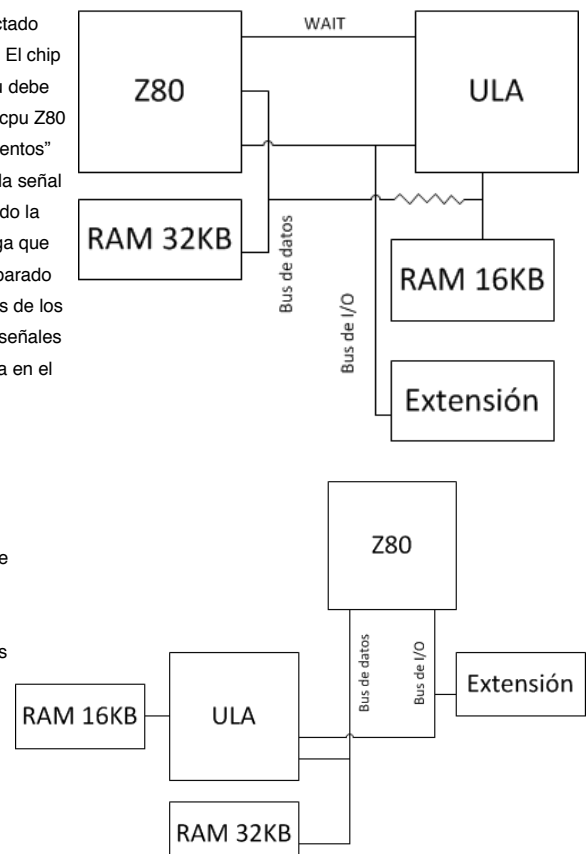
En la época del Spectrum tuvieron una idea económica, sencilla, y facilísima de implementar: en lugar de emplear un costosísimo (y complejísimo para la época) chip que se encargase de éstas funciones, lo que hicieron fue unir directamente y sin escalas el procesador principal y el chip ULA del Spectrum al mismo bus. El problema es que el chip ULA tiene que compartir la memoria con el procesador principal, para leer sus contenidos y generar la imagen que vemos en el televisor. Cómo podríamos hacer que dos componentes hardware activos, que necesitan leer la memoria en el mismo momento, puedan convivir usando los mismos buses?

En el diagrama (escandalosamente simplificado) se puede ver cómo el chip ULA está conectado directamente al primer banco de memoria de 16KB (y único en el caso del Spectrum 16KB). El chip ULA necesita leer la memoria para generar la imagen de vídeo, pero al mismo tiempo la cpu debe acceder a la memoria también para seguir ejecutando programas. La solución es simple: la cpu Z80 tiene una señal hardware (“WAIT”) que normalmente se emplea por dispositivos hardware “lentos” para permitirles terminar su trabajo cuando son accedidos por el procesador. En éste caso, la señal WAIT está conectada al chip ULA que, cuando necesita acceder a la memoria RAM (o cuando la cpu accede a sus registros) simplemente mantiene activa la señal, haciendo que la cpu tenga que esperar. La conexión del bus de datos del bloque de RAM de 16KB y el procesador está separado por simples resistencias, así que si la cpu lee memoria fuera del rango de la ULA las señales de los chips de memoria llegarán directamente a la CPU (las resistencias bloquean el paso de las señales al bus de la ULA). Cuando la cpu lee los 16KB inferiores y dado que ningún chip de memoria en el bus principal forzará señal alguna en el bus, llegará al bloque de memoria a través de las resistencias. Francamente muy ingenioso (y económico de implementar, por cierto!).

Para nuestra implementación vamos a intentar hacer algo parecido, pero en nuestro caso, y dado que el software nos permite emular el hardware “gratis”, si vamos a emplear el concepto de “northbridge” de una cpu moderna: vamos a implementar una ULA que contiene su propio bloque de 16KB y a los que la cpu accede a través de la misma, independientemente de otros componentes (32KB extra, por ejemplo) que pueda haber conectados. Esta arquitectura nos facilita otra emulación que vendrá en un futuro: emular los retardos que introduce la ULA en la ejecución de programas – y os sorprendería la cantidad de software que “cuenta ciclos” para realizar curiosos efectos gráficos.

ULA – implementación básica

Según podemos ver en el gráfico, el chip ULA se conecta al mismo tiempo tanto al bus de



datos como al bus de I/O del procesador. Hasta ahora nuestros componentes (como por ejemplo la RAM) podían conectarse a uno u a otro, pero no a los dos a la vez. Tipo de bus nuevo? Nuevas clases de dispositivos? O cómo hacemos para que un componente herede de dos clases bases iguales a la vez? Nuevamente, C++ viene al rescate: herencia múltiple.

Necesitamos que nuestro componente ULA distinga entre los accesos al bus de datos y accesos al bus de direcciones, pero la cpu usa la misma función en ambos casos: `Read()` o `Write()` de las clases base `BusComponentBase`. Podríamos pensar en que la hemos pifiado con el diseño original, pero no es el caso: implementar la ULA derivando de dos buses a la vez es tan “simple” como ésto:

```
class ULAMemory : public RAM<16384,16*1024>
{
protected:
    void Write(unsigned int address,unsigned char value)
    {
        // First, store data
        __super::Write(address,value);
        // and forward to ULA
        MemoryWrite(address,value);
    }
    virtual void MemoryWrite(unsigned int address,
                            unsigned char value) = 0;
};

class ULAIO : public BusComponent<0xFE,1>
{
protected:
    unsigned char Read(unsigned int address)
    {
        return(IORead(address));
    }

    void Write(unsigned int address,unsigned char value)
    {
        IOWrite(address,value);
    }

    virtual unsigned char IORead(unsigned int address) = 0;
    virtual void IOWrite(unsigned int address,
                        unsigned char value) = 0;
};

class ULA : public ULAMemory, ULAIO
{
    ....
    void MemoryWrite(unsigned int address,unsigned char value);
    void IOWrite(unsigned int address,unsigned char value);
    unsigned char IORead(unsigned int address);
    ....
};
```

Definimos dos clases base (virtuales) que cuando reciben las llamadas desde la cpu `Read()` y `Write()` invocan a una correspondiente función `MemoryWrite()`, `IORead()` e `IOWrite()` de una futura clase derivada, implementada por la clase ULA, que deriva de ambas. En éste caso no implementamos `MemoryRead()`, porque la implementación base (de la clase RAM) hace precisamente lo que necesitamos: devolver el contenido de la posición de memoria emulada a la cpu. En cambio la función `MemoryWrite()` la emplearemos para actualizar la pantalla simulada. Para más efectividad, la clase `ULAMemory` deriva de `RAM<16384,16384>`, con lo que el almacenamiento de la ULA ya está implementado. No ha sido tan difícil, verdad?

Pero cómo conectamos cada uno de los dos buses de la ULA a los buses correspondientes del procesador? Fácil, casteando la instancia de la ULA al bus correspondiente:

```
dataBus.AddBusComponent((ULAMemory*)&ula);
....
ioBus.AddBusComponent((ULAIO*)&ula);
```

La pantalla del Spectrum

Ya que estamos rehaciendo la ULA, el primer paso será analizar la pantalla del Spectrum y analizar su emulación.

El Spectrum tenía una pantalla de 256x192 píxeles (increíble que se viese algo, verdad?) y emplea dos zonas de memoria para la generación de la imagen. La primera zona de memoria de 6 kilobytes, comprendida entre las posiciones de memoria 16384 (0x4000) a 22527 (0x57FF) contiene el bitmap en sí, donde cada byte define una secuencia de 8 píxeles horizontales. Una segunda sección inmediatamente después de 768 bytes, desde 22528 (0x5800) a 23295 (0x5AFF) contiene un byte de atributo por cada bloque de 8x8 píxeles del bitmap. Cada byte de atributos, que se aplica a cada bloque de 8x8 píxeles en pantalla, tiene el formato siguiente:

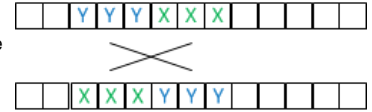
F	P	P	P	B	I	I	I	I
---	---	---	---	---	---	---	---	---

El bit marcado como 'F' indica que el carácter parpadea, esto es, intercambia cada cierto tiempo los colores de fondo y principal. Los bits 'P' indican el color de fondo (o “PAPER” usando nomenclatura del Spectrum), Los bits 'I' a su vez indican el color principal (“INK”) del carácter, y el bit 'B' indica que el color es brillante.

Parece fácil, no? Bueno, casi.

El “problema” es que, internamente, la líneas que componen el bitmap de pantalla del Spectrum no están ordenados. De hecho, el truco consiste en que “cruzaron” líneas de direcciones entre los bloques superior e inferior de direcciones, de tal forma que podían usar la parte alta de los registros de 16 bits (H en el registro HL, por ejemplo) para ir “bajando” líneas a medida que dibujaban caracteres. Para que ésto funcione, cada línea tiene que estar 256 bytes más adelante en memoria, en lugar de los 32 (32 bytes x 8 pixeles son los 192 pixeles horizontales de pantalla) que serían necesarios de usar un bitmap correlativo. Se ahorran un montón de cálculos, porque para avanzar líneas basta con hacer “INC H” en lugar de otras operaciones aritméticas.

Dado que vamos a emular la pantalla en un bitmap de 320x240 (tamaño francamente estándar y que además nos da margen para emular el borde de la pantalla), era necesario “linealizar” la memoria de video del Spectrum. Resulta que es tan fácil como intercambiar dos bloques de tres bits en la dirección para obtener otra dirección, lineal en un bitmap de 256x192.



Como tenía que definir el formato en que iba a general el bitmap, y para que fuese lo más portable posible, opté por emplear BGRA en 32 bits. Lo mejor del caso es que, dado que los colores que puede generar la ULA del Spectrum son limitados a 8 (con otros 8 iguales pero más brillantes) y lo mejor es hacerlo por tabla, adaptar de un formato a otro es trivial en la mayoría de los casos. Así que manos a la obra e incluimos el siguiente código en la función `MemoryWrite()` de la clase ULA:

```
void MemoryWrite(unsigned int address,unsigned char value)
{
    if (nativeBitmap == NULL)
        return;

    address &= 0x3FFF;
    if (address > 0x1AFF)    // Above bitmap+attrs?
        return;

    // Video memory: 0x0000->0x17FF bitmap graphics,
    //                0x1800->0x1AFF attributes
    if (address < 0x1800)    // Bitmap graphics
    {
        // Convert from Spectrum memory video to plain buffer
        unsigned int rowbase = address >> 5;
        rowbase = ((rowbase & 0xC0) |
            ((rowbase & 0x38) >> 3) |
            ((rowbase & 0x07) << 3));

        LPDWORD scr = nativeBitmap;
        // Center pixel video (256x192) in the whole
        //bitmap (320x240) == offset 32 horz, 24 vert
        scr += (320 * 24);
        scr += 32;
        scr += (rowbase * dwwordsPerRow);
        scr += ((address & 0x001F) * 8);

        // Fetch attribute
        unsigned char attr = data[0x1800 + ((rowbase / 8) * 32) + (address & 0x001F)];
        DWORD dwInk;
        DWORD dwPaper;
        if ((attr & 0x80) && blinkState)
        {
            dwPaper = dwColorTable[((attr & 0x40) >> 3) | (attr & 0x07)];
            dwInk = dwColorTable[(attr & 0x78) >> 3];
        }
        else
        {
            dwInk = dwColorTable[((attr & 0x40) >> 3) | (attr & 0x07)];
            dwPaper = dwColorTable[(attr & 0x78) >> 3];
        }

        for (int dd=7;dd >= 0;dd--)
        {
            *scr++ = (value & (1 << dd)) ? dwInk : dwPaper;
        }
        IsDirty = true;
    }
    else    // Attribute memory
    {
        // ToDo: Redraw whole affected "character block"
    }
}
```

La variable miembro `memoryBitmap` se inicializará (por otros medios) al buffer real que hay que manipular y que nos será provisto desde una instancia superior. En la clase está definido como un `DWORD*`, para poder acceder a cada pixel BGRA (32 bits) como un solo elemento del array. La función accede al miembro `data` de la clase base `ULAMemory` para determinar el atributo (los colores) a emplear para “pintar” la pantalla. Y finalmente, la variable `blinkState` determina en qué fase del “parpadeo” se encuentra el carácter y que será actualizada más adelante (a “0” ó “1” alternativamente).

La cosa se complica un poco cuando lo que se escribe en la memoria de la ULA es un atributo de color: hay que actualizar un carácter completo de 8x8 pixeles, así que vamos a escribir una función especializada en ello. La función simplemente recibe el offset del carácter a redibujar, de 0 a 767:

```
void UpdateChar(unsigned int nChar)
{
    int memOffset = ((nChar & 0x300) << 3) | (nChar & 0xFF);
    unsigned char value = data[nChar + 0x1800];
```

```

LPDWORD scr = nativeBitmap;
// Center pixel video (256x192) in the whole
// bitmap (320x240) == offset 32 horz, 24 vert
scr += (320 * 24);
scr += 32;
scr += (((nChar / 32) * 8) * dwordsPerRow);
scr += ((nChar & 0x001F) * 8);

DWORD dwInk;
DWORD dwPaper;
if ((value & 0x80) && blinkState)
{
    dwPaper = dwColorTable[((value & 0x40) >> 3) | (value & 0x07)];
    dwInk = dwColorTable[(value & 0x78) >> 3];
}
else
{
    dwInk = dwColorTable[((value & 0x40) >> 3) | (value & 0x07)];
    dwPaper = dwColorTable[(value & 0x78) >> 3];
}

// Redraw 8x8 pixels
for (int yy=0;yy<8;yy++)
{
    unsigned char scanData = data[memOffset];
    LPDWORD pixel = scr;
    for (int dd=7;dd >= 0;dd--)
        *pixel++ = (scanData & (1 << dd)) ? dwInk : dwPaper;
    memOffset += 256; // Next scanline on Spectrums memory
    scr += dwordsPerRow; // Next scanline on native bitmap
}
}

```

Ahora podemos completar la función `MemoryWrite()`, para que actualice el bloque de 8x8 píxeles cuando se cambia un atributo:

```

else // Attribute memory
{
    // Redraw whole affected "character block"
    UpdateChar(address - 0x1800);
}

```

Finalmente, sólo nos queda definir la paleta de colores (`dwColorTable`) del Spectrum:

```

static const unsigned int dwColorTable[] =
{
    0xFF000000,
    0xFF0000CD,
    0xFFCD0000,
    0xFFCD00CD,
    0xFF00CD00,
    0xFF00CDCD,
    0xFFCDCD00,
    0xFFCDCDCD,

    0xFF000000,
    0xFF0000FF,
    0xFFFF0000,
    0xFFFF00FF,
    0xFF00FF00,
    0xFF00FFFF,
    0xFFFFFF00,
    0xFFFFFFFF
};

```

El siguiente paso

Qué sigue ahora? Ahora toca hacer un banco de pruebas para todos estos componentes, y verificar que podemos traer un Spectrum a la vida. Para hacernos la vida más facil vamos a usar un aliado que nos hará casi, casi trivial la implementación de una aplicación que una todas las piezas que hemos desarrollado: WPF. Y para facilitar la unión de un entorno de tan alto nivel y nuestros componentes en C++, nada mejor que seguir desarrollando en C++... pero gestionado y soportado por .NET.



Hazte un Spectrum - Capítulo 4

En la anterior parte de este largo tutorial, la ULA del Spectrum estaba empezando a tomar cuerpo. A partir de ahora empieza la integración práctica de todos los componentes gracias a la inestimable ayuda de .NET y WPF.

Ya empieza a ver muchos componentes sueltos, y toca empezar a desarrollar el armazón que sujetará todo. Ya que toda la base está desarrollada en C++, bien podríamos seguir en éste lenguaje y completar un emulador totalmente nativo – solo que da como pereza, sobre todo teniendo en cuenta que, para completarlo acabaríamos interactuando con el GDI de Windows “de toda la vida”, y que está más que obsoleto. En cambio pensé en usar .NET y WPF que ofrecen una infinita mayor versatilidad y, sobre todo, una impresionante facilidad de migración hacia el futuro – léase WinRT. Pero no adelantemos acontecimientos.

Para un componente de bajo nivel como nuestro emulador de Spectrum, C++ es sin duda el lenguaje ideal. Pero para completarlo necesitamos un “front-end”, una “aplicación emulador” y pensé cómo convertir todo el código creado hasta ahora como un componente reutilizable. Simplemente, convirtamos el emulador en un control WPF, que podremos emplear desde una sencillísima aplicación .NET y C#.

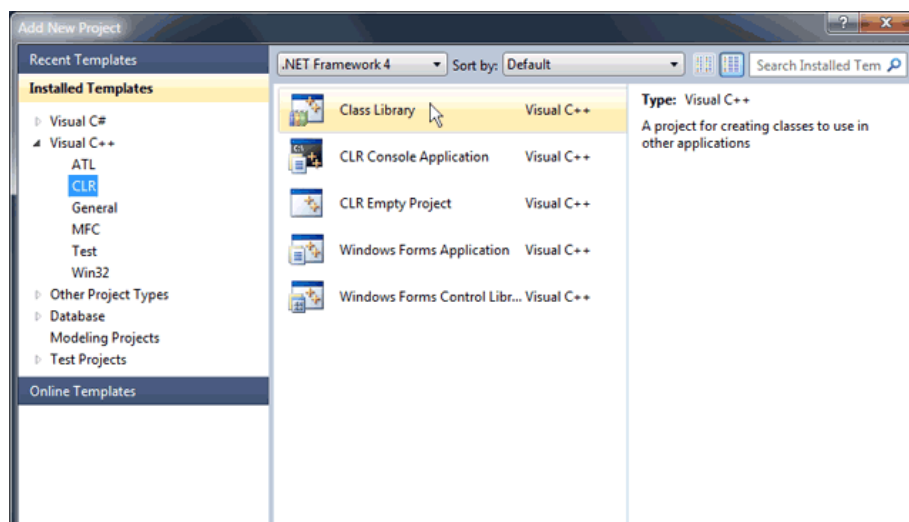
C++ Gestionado

Parece que existe la convicción de que “todo” se puede (e incluso, “debe”) hacerse en C# para ser molón, para ser “del siglo XXI”, pero para mí no es el caso. C# es una maravilla de lenguaje, encuentro su estructura absolutamente preciosa, pero no me da todo lo que necesito. Lo que la mayoría de la gente pasa por alto es que hay “otro C++” el C++ Gestionado (ó “Managed C++”) que es la combinación perfecta de los dos entornos: la flexibilidad, la potencia de C++, unido a poder emplear un runtime moderno como .NET. En C++ Gestionado se puede mezclar el código nativo, con acceso a bajo nivel a la máquina junto con clases de altísimo nivel de abstracción, todo en un sólo lenguaje, todo en una sola librería. El truco? Las clases “ref”.

Las clases en C++ se declaran como “class NombreDeLaClase”. Esa clase será nativa, con acceso total a la máquina y al API. Pero, y si una clase en C++ desea acceder (y ser accedida) desde .NET? Basta con declarar la clase como “ref class NombreDeLaClase”. Una “ref class” es una clase .NET a todos los efectos, solo que escrita en C++. Eso es justo lo que queremos.

La clase “Emulation.Spectrum”

Vamos a escribir una clase que, vista desde .NET es un control WPF más, pero que internamente implementa un Sinclair ZX Spectrum. El primer paso es crear un nuevo proyecto, una “class library” en C++ Gestionado. Y si, apuesto a que muchos de vosotros ni siquiera os habíais fijado en esa parte de los proyectos que se pueden crear con Visual Studio:





En qué se diferencia programar en C++ en un entorno gestionado? Absolutamente en nada. Simplemente se abre la puerta a la posibilidad de crear clases “visibles” desde .NET y que soporten toda su inmensa librería de componentes – lo cual, frente a tener que hacerlo usando al API clásico de Windows, es toda una ventaja. El cambio llega al crear clases gestionadas, donde la sintaxis clásica de C++ se extiende. Si, hay una curva de aprendizaje, pero considero que vale la pena. Y no es tan duro como podría parecer en un principio.

Una vez creado el proyecto, simplemente copié dentro todos los fuentes C++ que ya había diseñado. Y compila, aunque claro, no genera nada visible desde .NET. Así que ahora toca implementar el control. Para ello, escogí el control `Border` (nombre completo, “`System.Windows.Controls.Border`”) como clase base de la que derivarme porque necesito lo más básico posible. De hecho, me vale cualquier control WPF que tenga un “`Background`” o un “`Fill`”... Así que la clase es más o menos como sigue (y nótese que es una “ref class”):

```
using namespace System;
using namespace System::Threading;
using namespace System::Windows::Media::Imaging;

namespace Emulation
{
    public ref class Spectrum : System::Windows::Controls::Border, IDisposable
    {
    public:
        Spectrum();
        ~Spectrum()
        {
            Stop();
            GC::SuppressFinalize(this);
        };
        !Spectrum()
        {
            Stop();
            emulatorScreen = nullptr;
        }

    public:
        void Start();
        void Stop() { };

    protected:
        void emulatorMain();
        void OnRenderTick(Object^ sender, EventArgs^ e);

    private:
        Thread^ emulatorThread;
        WriteableBitmap^ emulatorScreen;
        LPDWORD screenData;
        bool screenDataDirty;
        int bytesPerScreenLine;
        int quitEmulation;
    };
}
```

Lo primero que llama la atención a un programador C++ (e incluso a uno de C#) es el uso de “^” para representar un puntero. Realmente no es un puntero, es un “handle” a un objeto .NET. Pronto descubriremos como podemos mezclar, en la misma clase, handles a otros objetos .NET y punteros a instancias de clases nativas (como el “`LPDWORD`” al buffer nativo del bitmap). Todo al mismo tiempo.

La implementación es muy simple: la emulación va a correr en un thread independiente (para simplificar incluso más la programación desde un entorno .NET). La emulación de la pantalla voy a resolverla usando un componente `WriteableBitmap`, que no es más que una “superficie pintable” (si es que eso significa algo) que voy a emplear como “`Background`” de mi control.

El método `Start` de la clase es simple: instanciamos el bitmap (`WriteableBitmap`) para tener dónde “pintar” la pantalla del `Spectrum` y arrancamos el thread de emulación:

```
Spectrum::Spectrum()
{
    emulatorScreen = nullptr;
    quitEmulation = false;
    screenData = nullptr;
    screenDataDirty = false;
    m_RenderDelegate = gcnew EventHandler(this, &Spectrum::OnRenderTick);
}

void Spectrum::Start()
{
    if (emulatorThread != nullptr)
        return; // Already running

    quitEmulation = false;
```

```

// Attach to WPF screen refresh to update
CompositionTarget::Rendering += m_RenderDelegate;

// Build the emulated screen and WPF render surface.
if (emulatorScreen == nullptr)
{
    // Emulator will draw on this bitmap
    emulatorScreen = gcnew WriteableBitmap(320,240,72,72,PixelFormat::Bgra32,nullptr);
    bytesPerScreenLine = emulatorScreen->BackBufferStride;

    emulatorScreen->Lock();
    screenData = (LPDWORD)emulatorScreen->BackBuffer.ToPointer();

    for (int dd=0;dd<320*240;dd++)
        screenData[dd] = 0xFFC0C0C0;
    emulatorScreen->Unlock();

    // ... and the bitmap is attached as the contents of the control's background.
    ImageBrush^ bkgnd = gcnew ImageBrush();
    bkgnd->ImageSource = emulatorScreen;
    Background = bkgnd;
}

emulatorThread = gcnew Thread(gcnew ThreadStart(this,&Spectrum::emulatorMain));
emulatorThread->Start();
}

```

Fijaos en otra diferencia al programar en C++ gestionado: las clases nativas se instancian con “new”, mientras que las clases gestionadas se instancian con “gcnew”. Nunca he entendido el por qué de emplear dos comandos para hacer algo que el compilador podría distinguir él solito...

Lo único reseñable de éste fragmento de código es cómo usamos la clase `WriteableBitmap` como elemento de unión entre WPF y C++: por un lado obtenemos un puntero “clásico de toda la vida” a los contenidos físicos del bitmap, mientras que a la vez lo usamos para construir una textura que usamos como “fondo” (“Background”) del control. El único trabajo extra es que necesitamos avisar a WPF cada vez que el bitmap cambie, y para eso nos “enganchamos” al delegado de repintado (`CompositionTarget.Rendering`) del sistema.

Por cierto, sí que necesitamos una cosa más: un módulo que contenga la ROM del Spectrum: nuevamente, empleando nuestros buses la implementación es sencilla:

```

template<unsigned int B,unsigned int S> class ROM : public BusComponent<B,S>
{
public:
    int Load(char *romName)
    {
        FILE *file = NULL;
        int err = fopen_s(&file,romName,"rb");
        if ((err == 0) && (file != NULL))
        {
            int got = fread(data,1,S,file);
            fclose(file);
        }
        return(err);
    };

public:
    void Write(unsigned int address,unsigned char value)
    {
        // Do nothing... this is ROM!!!
    }
    unsigned char Read(unsigned int address)
    {
        return(data[address-B]);
    }

protected:
    unsigned char data[S];
};

```

Para la implementación de la función “Load()” he usado el código más “ANSI” posible – la portabilidad primero (y por eso no hago más que usar “unsigned short”, “unsigned int”, etc, en lugar de los términos habituales de WIN32 como `USHORT` o `DWORD`).

Ahora, echemos un vistazo a la función “emulatorMain()”, que es donde se realiza el trabajo de verdad. Observad como para ser una clase .NET, maneja tanto objetos nativos como gestionados con la misma soltura:

```

void Spectrum::emulatorMain()
{
    // Components
    Bus16 bus;           // Main Z80 data bus
    ROM<0,16384> rom;     // Main system ROM
    ULA ula;             // Spectrum's ULA chip + 16KB
    RAM<32768,32*1024> ram; // Remaining RAM (32KB)
    Z80 cpu;             // and finally, the CPU core

    // Load ROM contents
    if (rom.Load("48.rom"))
        throw gcnew System::IO::FileNotFoundException("Unable to load rom '48.ROM'");
}

```



```

// Configure the ULA with the native Windows bitmap used to emulate the screen
ula.SetNativeBitmap((LPBYTE)screenData,bytesPerScreenLine);

// Populate busses
bus.AddBusComponent(&rom);
bus.AddBusComponent((ULAMemory*)&ula);
bus.AddBusComponent(&ram);

// And attach busses to cpu core
cpu.DataBus = &bus;
cpu.IOBUS = (ULAIOW*)&ula;

cpu.regs.PC = 0;

// Ready to roll!!
DWORD dwFrameStartTime = GetTickCount();
do
{
    // Emulate next instruction
    cpu.EmulateOne();
} while(quitEmulation == false);
}

```

Dos cosas: la primera es que si os fijáis, he conectado la parte de I/O del chip ULA “a capón” al procesador (otra ventaja más de nuestra estructura de buses). Para que ésto funcione, sólo tengo que crear las dos funciones `IORead()` e `IOWrite()` de la ULA:

```

unsigned char IORead(unsigned int address)
{
    return(0xFF); // pull ups on bus
}

void IOWrite(unsigned int address,unsigned char value)
{
    // ULA is selected when reading an even address
    if (address & 0x01)
        return;

    // Update border color member variable.
    dwBorderRGBColor = dwColorTable[value & 0x07];
}

```

La arquitectura de la ULA – y de los buses de I/O del Spectrum en general están simplificados al máximo. Si bien la dirección de I/O “estándar, documentada” de la ULA es 0xFE, cualquier dirección con el bit 0 de direcciones de “0” (o sea, cualquier dirección de I/O par) también sirve. De momento aprovecho para quedarme con el color para el borde de la pantalla, que se configura en los tres bits menos significativos del único puerto de I/O de la ULA. Pero de eso toca hablar un poco más adelante.

En segundo lugar, y si recordáis el capítulo anterior, nuestra implementación de ULA estaba a la espera de que nos diesen un bitmap sobre el que dibujar. Como ya lo tenemos – gracias al `WriteableBitmap` e instanciado en el método `start()` de la clase, ha llegado el momento de hacérselo saber a la ULA para que lo use, por medio de la función “`SetNativeBitmap()`”. La implementación es trivial:

```

void SetNativeBitmap(LPBYTE pBitmap,int bytesPerScanLine)
{
    nativeBitmap = (LPDWORD)pBitmap;
    dwordsPerRow = bytesPerScanLine / 4;
}

```

Sólo queda una cosa más por hacer: WPF nos llamará 50 veces por segundo para saber si tenemos algo que pintar. De momento, vamos a lo bestia:

```

void Spectrum::OnRenderTick(Object^ sender,EventArgs ^e)
{
    emulatorScreen->Lock();
    emulatorScreen->AddDirtyRect(Int32Rect(0,0,320,240));
    emulatorScreen->Unlock();
}

```

Si, poco optimizado (invalida los contenidos del `WriteableBitmap` 50 veces por segundo), pero por lo menos nos sirve para ver hasta dónde llegamos.

Compilamos todo y obtenemos una bonita DLL en C++ gestionado. Ahora, habrá que usarla en alguna parte, verdad?

Vuelta a C#

Ya tocaba? Bien, esto será muchísimo más familiar para la mayoría. Vamos a hacer una aplicación WPF en C#, aunque VB sería exactamente igual de válido. Cread un nuevo proyecto de “aplicación WPF” (yo le he puesto “Emulator” de nombre – original que es uno) y agregad la referencia a nuestro flamante control y abrid “`MainWindow.xaml`”.

Lo primero, es necesario definir un “namespace” XML para referirnos a nuestro control. Simplemente agregadlo en la cabecera del XAML, indicando la DLL que contenga la implementación (“`Spectrum.DLL`” en mi caso). Para el namespace, escogí (como no podía ser de otro modo) “zx”:

```
<Window x:Class="Emulator.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="YASS - Yet another Spectrum Simulator"
        Height="350"
        Width="525"
        xmlns:zx="clr-namespace:Emulation;assembly=Spectrum">
```

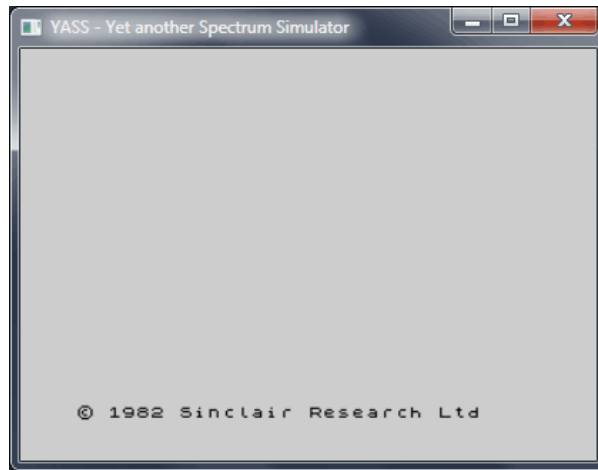
Quitamos todo lo que sobra que VisualStudio tiene a bien colocar de forma predeterminada en el cuerpo de la ventana y lo reemplazamos por ésto:

```
<Grid>
    <zx:Spectrum x:Name="emulator" />
</Grid>
```

Sólo queda arrancar el emulador al inicio del programa. En el constructor de la clase basta con invocar el método Start() del control:

```
public MainWindow()
{
    InitializeComponent();
    emulator.Start();
}
```

Nos aseguramos que el proyecto "Emulator" está configurado como predeterminado (VisualStudio tiene tendencia a dejar como predeterminado el primer proyecto que se agrega a una solución), pulsamos F5 (o sea: "Debug/Start"), cruzamos los dedos y nuestro trabajo se verá recompensado con ésto:



Es el Spectrum más "sordo" del universo (no tiene teclado, no tiene sonido, no tiene nada de nada), pero la secuencia de arranque la completó a la perfección y nuestra implementación de ULA es capaz de actualizar un bitmap de WPF recibiendo las escrituras de la cpu a la memoria de vídeo emulada. Vamos bien...

El siguiente paso será darle a nuestro Spectrum un estupendo teclado, para poder comunicarnos con él. También descubriremos cómo necesitaremos recrear las temporizaciones de un televisor para sincronizarlo todo... Tranquilos, es más fácil de lo que parece.



Hazte un Spectrum - Capítulo 5

En el capítulo anterior conseguimos arrancar por primera vez nuestro nuevo emulador de Spectrum, pero sin ninguna forma de recibir entradas del mundo exterior su utilidad es francamente nula. Hoy implementaremos la simulación del teclado y las temporizaciones de la máquina.

La implementación del teclado tiene dos partes realmente: la primera, el interface hardware entre la cpu y las teclas, misión encargada (obviamente) al chip ULA. Por otra parte, el Spectrum necesita un reloj que periódicamente le haga “visitar” el teclado y leerlo. Todos los ordenadores necesitan este tipo de relojes y la mayoría incluyen relojes internos programables que facilitan la tarea. En los ‘80 éste no era el caso y la mayoría empleaban la temporización de la pantalla para sincronizar sus operaciones. Y cuando digo “pantalla” me refiero a “televisor” y su conocido refresco de 50 cuadros por segundo.

El chip ULA del Spectrum es el responsable (como ya hemos visto) de generar la imagen que aparece en la pantalla. Para hacerlo, el chip ULA cuenta meticulosamente ciclos de la máquina para determinar dónde está el cañón del televisor en cada momento. Durante unas cuantas líneas (el borde superior de la pantalla) el chip simplemente genera el color del borde y periódicamente la señal de sincronismo horizontal para que el cañón vuelva al flanco izquierdo de la pantalla. Cuando la ULA determina que ha llegado a la zona de pantalla propiamente dicha, comienza a leer la memoria de vídeo y convertir los contenidos en imágenes durante 256 píxeles, luego genera un poco más de borde, un sincronismo horizontal, un poco más de borde (en el lado izquierdo) y luego otra línea de pantalla, así hasta que completa las 192 líneas. Finalmente, vuelve a generar color de borde hasta que se completa el cuadro de pantalla (312 líneas) y genera la señal de sincronismo vertical, para que le haz de electrones del cañón vuelva a la esquina superior izquierda de la pantalla. Y esto, 50 veces por segundo. Es en éste momento cuando el chip ULA envía una señal a la cpu (la señal INT) que interrumpe el proceso de la cpu cada cuadro y que el Spectrum emplea para sincronizar sus procesos.

En nuestra emulación no existe un cañón de electrones de una televisión, ni nada que se le parezca, pero necesitamos simular esa interrupción 50 veces por segundo. Podríamos simularla con un simple temporizador, pero su precisión sería desastrosa. Os recuerdo que muchos juegos del Spectrum hacían simpáticos efectos con el borde de la pantalla (cambiando el color muchas más veces que 50 por segundo). También quiero simular ese efecto.

Afortunadamente, el chip ULA emplea los mismos relojes que la cpu Z80 y los usa para cronometrar sus operaciones. El chip ULA empieza a contar ciclos de reloj y considera que los 16384 primeros ciclos son el borde superior de la pantalla, justo hasta el borde izquierdo del contenido de pantalla de la primera línea de visualización. Luego cada línea son 224 ciclos exactos: 128 ciclos para presentar los 256 píxeles de pantalla (esto es, 2 píxeles por cada ciclo), mas 96 ciclos para el borde derecho, el sincronismo horizontal y el borde izquierdo de la siguiente línea. En total, cada fotograma de pantalla son 224 ciclos por 312 líneas, esto es, 69888 ciclos (o tStates) y si multiplicáis esta cifra por 50 fotogramas por segundo obtenemos 3494400 ciclos (los famosos 3.5 MHz a los que funciona el Spectrum). De hecho, el chip ULA realmente funciona a 3.5 MHz, lo cual resulta en una visualización en pantalla de 50.08 cuadros por segundo... afortunadamente, a ningún televisor le afecta esta diferencia.

Así que vamos a implementar la función de la ULA que cuenta los ciclos de reloj a medida que avanza la visualización, en la función `AddCycles()`:

```
#define TSTATES_PER_SCANLINE (224)
#define TVSCANLINES (312)
#define TSTATES_PER_FRAME (TSTATES_PER_SCANLINE * TVSCANLINES)

void AddCycles(unsigned int cycles, bool& IRQ)
{
    dwFrameTStates += cycles;
    dwScanLineTStates += cycles;

    if (dwScanLineTStates > TSTATES_PER_SCANLINE)
        ScanLine(IRQ);
}
```

Empleamos dos contadores porque es más simple que hacer operaciones sobre uno solo, y mantenemos un contador de líneas y un contador de cuadros. Cuando detectamos que se ha completado una línea de visualización completa, invocamos a la función `ScanLine()`:

```
void ScanLine(bool &IRQ)
```

```

{
    dwScanLineTStates %= TSTATES_PER_SCANLINE;
    dwScanLine++;
    if (dwScanLine >= TVSCANLINES)
    {
        // Frame complete - trigger IRQ
        IRQ = true;
        dwFrameTStates %= TSTATES_PER_FRAME;
        dwScanLine = 0;
        if (++dwFrameCount > 16)    // each 16 full frames...
        {
            dwFrameCount = 0;        // reset counter...
            blinkState = !blinkState; // ... invert blink ...
            UpdateBlink();           // ... and update screen contents
        }
    }

    // Check if current scanline (0-311) lies within the "visible portion"
    // of screen bitmap (lines 0-239), i.e., skip first and last 36 lines (front/back porches)
    if ((dwScanLine < 36) || (dwScanLine > (239+36))) return;

    // Now check if current background color for scanline is different
    // from the required one
    DWORD bitmapLine = dwScanLine-36;
    if (dwCurrentScanLineBackColor[bitmapLine] == dwBorderRGBColor) // Already right color
        return;

    // Redraw current scanline with proper color
    LPDWORD scr = nativeBitmap;
    scr += (bitmapLine * 320);    // Get pointer to start of line

    // top/bottom borders?
    if ((bitmapLine < 24) || (bitmapLine > 24+191))
    {
        // Draw whole line
        unsigned int bCount = 320;
        do
        {
            *scr++ = dwBorderRGBColor;
        } while (--bCount);
    }
    else
    // screen contents area
    {
        // First 32 pixels
        unsigned int bCount = 32;
        do
        {
            *scr++ = dwBorderRGBColor;
        } while (--bCount);

        // Last 32 pixels
        scr += 256;
        bCount = 32;
        do
        {
            *scr++ = dwBorderRGBColor;
        } while (--bCount);
    }

    // and remember this scanline's colour
    dwCurrentScanLineBackColor[bitmapLine] = dwBorderRGBColor;
    IsDirty = true;
}

```

La función realiza tres operaciones: cuenta los ciclos de reloj tanto para la línea en curso como para la pantalla completa, y si determina que ésta se ha dibujado totalmente indica que es necesario invocar la interrupción del procesador poniendo a verdadero la variable `IRQ`. También cuenta bloques de 16 fotogramas para hacer parpadear los caracteres de pantalla que tengan el atributo "blink". Por último la función comprueba si ha cambiado el color del borde y repinta cada línea de la pantalla (si es que ha cambiado) con el nuevo color.

Para añadir la interrupción, basta un pequeño cambio en la función `emulatorMain()` que ya vimos en un capítulo anterior:

```

DWORD dwFrameStartTime = GetTickCount();
do
{
    // Emulate next instruction
    cpu.tStates = 0;
    cpu.EmulateOne();

    // After each instruction, report the ULA the number of cycles we've used
    bool irq = false;
    ula.AddCycles(cpu.tStates, irq);

    // As in the real Spectrum, the ULA will trigger an IRQ for every frame. This
    // implementation uses cpu clock cycles to know where the screen beam is.
    if (irq)    // Ula signals a frame interrupt
    {
        cpu.INT();    // Generate system interrupt

        // If screen contents have been modified, set a flag for the WPF rendering event.
        if (ula.IsDirty)
        {
            screenDataDirty = true;
            ula.IsDirty = false;
        }
    }
}

```

```

    }

    // The PC executes code a lot faster than the original Z80.
    // As we now that "20ms" (1 frame) have ellapsed, pause execution to
    // match host PC and emulated computer
    DWORD dwNow = GetTickCount();
    DWORD dwEllapsed = dwNow - dwFrameStartTime;
    if (dwEllapsed < 20)    // Running above real time?
    {
        Sleep(20 - dwEllapsed);
        dwNow = GetTickCount();
    }
    dwFrameStartTime = dwNow;
} while (quitEmulation == false);

```

Utilizando el contador de ciclos de la clase Z80 vamos actualizando la ULA para que mantenga la posición de pantalla. Si `AddCycles()` activa la variable local que le pasamos como referencia (`irq`), simplemente invocamos el método `INT()` del Z80 para que dispare una interrupción. Fijaos que también podemos aprovecharnos de esta característica para hacer que nuestro emulador corra más o menos a la misma velocidad que un Spectrum: dado que sabemos que la máquina ha estado corriendo durante un cuadro de pantalla completo (1/50 de segundo = 20 milisegundos), y que nosotros sabemos cuanto tiempo hemos empleado en hacerlo (`dwFrameStartTime - dwNow`), basta con hacer un `sleep()` del tiempo restante hasta cumplir los 20 milisegundos. No es exacto, pero el efecto es muy, muy aproximado.

Finalmente, añadimos una variable (`screenDataDirty`) para saber si los contenidos de pantalla han cambiado o no y optimizar la función de refresco de WPF (y no repintar cada fotograma completo como hasta ahora):

```

void Spectrum::OnRenderTick(Object^ sender, EventArgs ^e)
{
    if (screenDataDirty)
    {
        emulatorScreen->Lock();
        emulatorScreen->AddDirtyRect(Int32Rect(0,0,320,240));
        screenDataDirty = false;
        emulatorScreen->Unlock();
    }
}

```

Ya tenemos interrupciones! Ahora viene la parte en la que emulamos el hardware de teclado de la ULA.

8 ó 16 bits?

Al que se haya fijado, comprobará como el bus de I/O que hemos definido para nuestro procesador Z80 es de 16 bits. En cambio, y según la documentación de Zilog, el bus de direcciones de I/O del procesador real es de sólo 8 bits. Y esto? Bueno, no tengo documentación de la época y afortunadamente el datasheet actualizado lo describe, pero es que realmente cuando el procesador hace una operación de I/O coloca direcciones en los 16 bits de direcciones, aprovechando el contenido de otros registros del procesador.

Cuando leemos un puerto con una instrucción como `IN A,(C)`, el procesador realmente coloca en el bus de direcciones el contenido completo (16 bits) del registro BC, y no sólo la parte baja del mismo (C) como cabría esperar. De igual forma, cuando hacemos un `IN A,#puerto`, el procesador tiene la necesidad de colocar "algo" en los 8 bits más significativos del bus de direcciones, y coloca los contenidos del acumulador (registro A) porque es lo que tiene más a mano. Por ejemplo, las instrucciones `LD A,#14`; `IN A,$FE` realmente leen el puerto de I/O \$14FE. Y lo más curioso es que el Spectrum hace uso de esta característica para leer el teclado.

El chip ULA sólo utiliza una línea de direcciones para saber si están hablando con él: el bit de dirección 0 del bus. Esto es, cualquier dirección par de I/O accede a la ULA. Por convención, siempre se usa la dirección 0xFE, pero cualquier dirección par obtiene el mismo resultado. Y para leer el teclado aprovecha los 8 bits superiores de la dirección de I/O para saber qué sección del teclado deseamos leer.

El teclado del Spectrum

Ahora veamos cómo se conecta el teclado al resto del sistema. Las 40 teclas del Spectrum (4 filas de 10 teclas) están divididas en 8 bloques de cinco teclas cada uno:

	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
Fila 3	1	2	3	4	5	6	7	8	9	0	Fila 4
Fila 2	Q	W	E	R	T	Y	U	I	O	P	Fila 5
Fila 1	A	S	D	F	G	H	J	K	L	Enter	Fila 6
Fila 0	Shift	Z	X	C	V	B	N	M	Symbol Shift	Space Break	Fila 7

Para leer una fila del teclado, hay que leer el puerto de I/O de la ULA (0xFE) colocando en el byte más significativo la fila en la que estamos interesados. La fila se

codifica poniendo a “cero” el bit que represente su posición. Por ejemplo, para leer la fila “0” (teclas Shift, Z, X, C y V) se coloca “11111110” (0xFE), mientras que para leer la fila “5” (con las teclas Y, U, I, O y P) se coloca “11011111” (o lo que es lo mismo: 0xDF) en la parte alta de la dirección del puerto. Tras cada lectura, el chip ULA devuelve el estado de las cinco teclas correspondientes a esa fila (los cinco bits menos significativos) estando a cero el bit que represente una tecla pulsada. Fijaos también que la ordenación de los bits de las semilíneas 0 a 3 es la inversa que para las semifilas 4 a 7.

Para la implementación del teclado del emulador, voy a seguir el mismo patrón: un array de 8 bytes que representa las ocho “semifilas” de teclas y que podremos manipular desde el exterior para simular las pulsaciones:

```
protected:
    unsigned char keyMatrix[8];

public:
    void PressKey(unsigned int keyRow,unsigned int keyCol,bool down)
    {
        if (keyCol > 9)
            return;
        if (keyRow > 3)
            return;

        // Spectrum keyboard layout
        //      D0 D1 D2 D3 D4      D4 D3 D2 D1 D0
        // A11      ROW3              ROW4      A12
        // A10      ROW2              ROW5      A13
        // A9       ROW1              ROW6      A14
        // A8       ROW0              ROW7      A15

        int rowNdx;
        int bitMask;
        if (keyCol < 5)    // Left bank
        {
            rowNdx = 3 - keyRow;
            bitMask = 0x01 << keyCol;
        }
        else               // Right bank
        {
            rowNdx = 4 + keyRow;
            bitMask = 0x01 << (9 - keyCol);
        }

        if (down)
            keyMatrix[rowNdx] |= bitMask;
        else
            keyMatrix[rowNdx] &= ~bitMask;
    }
}
```

La función recibe las coordenadas de una tecla en el formato físico del Spectrum como “fila/columna”, y modifica el bit correspondiente a la configuración física por medio de unas simples operaciones, dependiendo de si la tecla ha sido pulsada (`down == TRUE`) o liberada (`down == FALSE`). El array `keyMatrix` queda totalmente configurado para poder leerlo muy rápidamente (y con la misma facilidad que en el caso real) cuando el código del Spectrum lo necesite.

Las peticiones de lectura del teclado llegarán a la ULA desde el procesador por medio de la función `IORead()`, que hasta ahora no hacía nada salvo devolver 0xFF (valor habitual en buses vacíos):

```
unsigned char IORead(unsigned int address)
{
    // ULA is selected by A0 being low
    if ((address & 0x01) == 0)
    {
        // Get the scan codes
        unsigned char kData = 0xFF;    // Pull ups
        unsigned char row = (address >> 8) ^ 0xFF;
        for (int dd=0;dd<8;dd++)
        {
            if (row & (1 << dd))      // Select scanline?
                kData &= ~keyMatrix[dd]; // pull down bits representing "pressed" lines
        }
        return(kData);
    }
    return(0xFF);
}
```

La función decodifica la dirección de I/O de la misma forma que el chip ULA real: viendo si el bit menos significativo de la dirección es “0”. En tal caso, recorre el array de filas, comprobando si la posición coincide con la máscara requerida. Si es así, limpia los bits correspondientes a las teclas pulsadas en la fila. La función recorre las ocho filas de teclas: si se solicita un identificador de fila con varios bits a cero, simplemente combina las teclas pulsadas en todas las filas seleccionadas.

Publicando el API del teclado

Dado que nuestro emulador de Spectrum es un control, necesitamos publicar funciones que nos permitan interactuar con él desde nuestra aplicación host. Dado que la instancia de la clase ULA es interna a la función `emulatorMain()` (y que hace todo el trabajo), vamos a publicar un puntero en nuestra clase que nos permita acceder a la instancia desde el exterior. Nada más simple: en la declaración de la clase `Spectrum` añadimos lo siguiente:

```
private:
    ULA *pCurrentUla;
```

y en la función `emulatorMain()`...

```
void Spectrum::emulatorMain()
{
    // Components
    ...
    ULA ula;      // Spectrum's ULA chip + 16KB
    ...

    pCurrentUla = &ula;
    ...
```

Cuando la emulación termina la función limpia la variable. Ahora, nada más fácil ahora que implementar dos funciones públicas (`PressKey()` y `ReleaseKey()`) que nos permitan simular las pulsaciones de teclas desde la aplicación host:

```
public:
    void Spectrum::PressKey(int keyRow,int keyCol)
    {
        if (pCurrentUla != nullptr)
            pCurrentUla->PressKey(keyRow,keyCol,true);
    }

    void Spectrum::ReleaseKey(int keyRow,int keyCol)
    {
        if (pCurrentUla != nullptr)
            pCurrentUla->PressKey(keyRow,keyCol,false);
    }
```

Terminemos añadiendo la emulación de teclado a nuestra aplicación host en C#. Sobrecargando las funciones `OnKeyDown()` y `OnKeyUp()` de la clase base, podemos pulsar o liberar las teclas del Spectrum. Aquí sigue un fragmento de la función `OnKeyDown()` – como podréis imaginar la función `OnKeyUp()` es básicamente la misma, solo que invocando a la función `ReleaseKey()` del control:

```
protected override void OnKeyDown(System.Windows.Input.KeyEventArgs e)
{
    base.OnKeyDown(e);
    switch (e.Key)
    {
        ....
        case System.Windows.Input.Key.Q: emulator.PressKey(1, 0); break;
        case System.Windows.Input.Key.W: emulator.PressKey(1, 1); break;
        case System.Windows.Input.Key.E: emulator.PressKey(1, 2); break;
        case System.Windows.Input.Key.R: emulator.PressKey(1, 3); break;
        case System.Windows.Input.Key.T: emulator.PressKey(1, 4); break;
        case System.Windows.Input.Key.Y: emulator.PressKey(1, 5); break;
        case System.Windows.Input.Key.U: emulator.PressKey(1, 6); break;
        case System.Windows.Input.Key.I: emulator.PressKey(1, 7); break;
        case System.Windows.Input.Key.O: emulator.PressKey(1, 8); break;
        case System.Windows.Input.Key.P: emulator.PressKey(1, 9); break;
        ....
    }
```

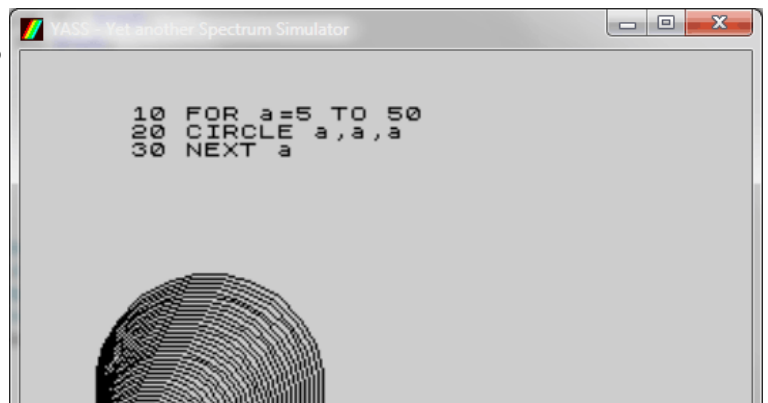
Podemos aprovechar también para fabricarnos “atajos” en el teclado del Spectrum. Por ejemplo, para borrar un carácter (tecla “Delete” del Spectrum) es necesario pulsar la combinación “mayúsculas” y “0”. Podemos mapear la tecla del PC a una combinación de ambas (y recordad hacer lo mismo en la función `OnKeyUp()`!!)

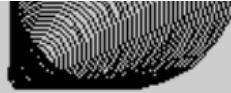
```
// Keyboard shortcuts:
case System.Windows.Input.Key.Back:
    emulator.PressKey(3, 0);    // Shift
    emulator.PressKey(0, 9);    // '0'
    break;
```

Ya podemos escribir!

Gracias a todo lo visto hoy nuestro control Spectrum está casi listo – ya podemos iniciar de nuevo nuestro emulador y disfrutar escribiendo código como se hacía en los ‘80... Quien no ha escrito algo así? Es lo primero que hicimos cuando, de chavales, descubríamos el maravilloso mundo de la informática, los ordenadores, la programación... y los videojuegos!

En la próxima entrega de este tutorial implementaremos la simulación de una unidad de cinta de cassette... porque todos queremos jugar a Head Over Heels o Skooldaze, no?





0 OK, 0:1



Hazte un Spectrum - Capítulo 6

Queda poco para completar un emulador de Sinclair ZX Spectrum, y hoy nos vamos a dedicar a emular una cinta de cassette para poder volver a disfrutar de esos videojuegos de los '80.

A ver, pregunta: cuántos de vosotros no recordabais o incluso sabíais que se usaban cintas de cassette como medio de almacenamiento masivo en los ordenadores personales de los '80? Yo me he dado cuenta de lo raro que suena hoy cuando al escribir "cassette" el corrector ortográfico se ha quejado porque no reconoce la palabra.... Hoy vamos a darle a nuestro emulador soporte para leer el formato más habitual: los archivos .TAP.

Nuevamente, y gracias a la información encontrada en WorldOfSpectrum.org he obtenido todo lo necesario para implementarlo. Lo primero, un poco de teoría...

Usar cintas de cassette como almacenamiento

Cualquier ordenador que use cintas de audio para almacenar o recuperar programas lo que realmente hace es generar sonidos con ciertos patrones que luego puede reconocer al reproducir la señal. En el caso del ZX Spectrum éste formato es totalmente conocido y se genera (cómo no) con el chip ULA. El chip ULA puede generar sonidos por dos vías diferentes, pero con exactamente las mismas herramientas: un sólo bit (0 ó 1) se emplea para generar una señal de alto nivel (1) o bajo nivel (0) en una salida de audio. Hay dos bits en el registro de I/O de la ULA, uno genera una señal más amplia que la otra. La primera se emplea para activar el altavoz interno del ordenador, mientras que la otra se envía al conector MIC que se enchufa a la cinta de cassette. Realmente las dos señales están unidas, siendo la única diferencia que la salida de la unidad de cassette pasa por una resistencia para atenuar la señal.

Para grabar datos en la cinta basta con modificar la señal de audio a 1 y a 0 contando ciclos entre cada "flanco". Por ejemplo, el tono de aviso previo a los datos de una cabecera consiste en 8063 pulsos (de 0 a 1 y vuelta a 0) con una duración de 2168 ciclos (tStates) entre flancos. Los bits se codifican igualmente con pulsos: un bit a "0" se codifica como dos pulsos de 855 ciclos entre flancos y un bit a "1" se codifica como dos pulsos de 1710 ciclos entre flancos. La recuperación es igual de simple: la señal de audio entra al chip ULA, que pone un bit a 0 o a 1 dependiendo del nivel de la señal. Los programas en la ROM del Spectrum simplemente leen el bit y calculan el tiempo que la señal está en "alto" o en "bajo".

El formato .TAP reproduce exactamente el formato binario de los bloques que genera la ROM del Spectrum, sin las señales de sincronismo. Simplemente añade dos bytes antes de cada uno de ellos para poder recuperarlos de un archivo secuencial.

Las rutinas en ROM del Spectrum graban la información como dos bloques separados. El primer bloque contiene una cabecera y mide 17 bytes. El bloque que sigue contiene la información propiamente dicha. Aprovechando el ejemplo más documentado del mundillo (grabar en cinta los dos primeros bytes de la ROM de un Spectrum con `SAVE "ROM" CODE 0,2`), un ZX Spectrum grabaría lo siguiente:

- 8063 pulsos de 2168 ciclos (tStates) cada uno para la señal de aviso de cabecera.
- Un pulso de sincronización de 667 ciclos.
- Un segundo pulso de sincronización de 735 ciclos.
- Un bloque de datos de 19 bytes con la cabecera: un byte con flags, más los datos de la cabecera que incluye tipo de bloque, longitud, ubicación en memoria, etc, y finalmente un byte de suma de control.

Tras una breve pausa, se grabarían los datos propiamente dichos:

- 3223 pulsos de 2168 ciclos para la señal de aviso de cabecera.
- Un pulso de sincronización de 667 ciclos.
- Un segundo pulso de sincronización de 735 ciclos.
- Un bloque de datos con un byte con flags, los dos bytes de datos y un byte con la suma de control del bloque.

Un fichero .TAP equivalente contendría lo siguiente:

- Dos bytes (0x13+0x00 == 0013h ó 19 decimal) con la longitud del bloque

- Los 19 bytes grabados físicamente en la cinta (flags+datos+suma de control)
- Dos bytes (0x03+0x00 == 0004h) con la longitud del bloque.
- Los 4 bytes grabados físicamente en la cinta (flags+datos+suma de control)

Parece obvio que los archivos .TAP contienen los datos grabados por la ROM del Spectrum "tal cual"... no sería posible leerlos de la misma forma?

Interceptando la ROM

Pues realmente sí. Tras leer bastante al respecto y gracias a la inestimable (e imprescindible) ayuda del libro "[The Complete ZX Spectrum ROM Dissassembly](#)" que encontraréis (cómo no) en WorldOfSpectrum.org/documentation.html, vamos a interceptar el curso normal de ejecución del código de la ROM y esperar tranquilamente hasta que ésta llegue al punto donde se leen bloques de cinta.

No hay que buscar mucho (nuevamente gracias a la ingente cantidad de información disponible). La función que queremos interceptar está a partir de la posición 056Bh de la ROM y su misión es leer un bloque (tanto de cabecera como de datos propiamente dicho) desde la cinta. Esta función (documentada como "LD-BREAK" en el desensamblado) es el primer (y frontal!!) paso en la lectura de un bloque.

Como la idea es hacer el emulador programable, en lugar de implementar los archivos .TAP directamente en el control lo vamos a implementar como un delegado normal y corriente de .NET. Cada vez que la ROM necesite un bloque de datos, la función que hayamos asignado a dicho delegado (escrita en C#) recibirá la petición y podrá devolver los contenidos. El cambio en la clase es mínimo:

```
namespace Emulation
{
    public delegate array<Byte>^ LoaderHandler(int bytesToLoad);

    public ref class Spectrum : System::Windows::Controls::Border, IDisposable
    {
    public:
        LoaderHandler^ OnLoad;

        ....
    }
}
```

La firma del delegado es la de una función que recibe un número de bytes a cargar y que devuelve un array de bytes con los datos. Ahora vamos a utilizar al delegado desde la emulación.

Dado que cada bucle que ejecuta el emulador consiste en una instrucción de la cpu Z80, para saber si hemos llegado al punto crítico para interceptar la carga basta con comparar el valor del registro PC de la cpu con la dirección que nos interesa antes de emularla:

```
// Ready to roll!!
do
{
    // Tape load trap
    if ((cpu.regs.PC == 0x056B) && (OnLoad))
    {
        try
        {
            LoadTrap(cpu);
            // set up registers for success
            cpu.regs.BC = 0xB001;
            cpu.regs.altAF = 0x0145;
            cpu.regs.CF = 1;           // Carry flag set: Success
        }
        catch(Exception^)
        {
            // set up registers for failure
            cpu.regs.CF = 0;           // Carry flag reset: Failure
        }
        cpu.regs.PC = 0x05e2; // "Return" from the "tape block load" routine
    }

    // And emulate next instruction
    cpu.tStates = 0;
    cpu.EmulateOne();
    ...
}
```

Fácil, no? Justo antes de emular cada instrucción comprobamos si el registro PC ("Program Counter") indica que hemos llegado a la dirección del programa de carga. Si es así y el delegado está inicializado en el emulador, invocamos una pequeña función que hará el trabajo sucio: pedir los datos y cargarlos en la memoria del Spectrum. El uso de C++ gestionado hace fácil hablar los dos "idiomas" (.NET y nativo) al mismo tiempo:

```
void Spectrum::LoadTrap(Z80& cpu)
{
    // Call the delegate to obtain the next tape block
    array<Byte>^ data = OnLoad(cpu.regs.DE);

    // First byte of data contains value for the A register on return. Last
    // byte is blocks checksum (not using it).
    int nBytes = data->Length-2;
    if (cpu.regs.DE < nBytes)
        nBytes = cpu.regs.DE;
}
```

```

// We must place data read from tape at IX base address onwards
// DE is the number of bytes to read, IX increments with each byte read.
for (int dd=0;dd<nBytes;dd++)
{
    // Write block using cpu's data bus and cpu's registers...
    cpu.DataBus->Write(cpu.regs.IX++,data[dd+1]);
    cpu.regs.DE--;
}
}

```

La implementación es tremendamente simple, pero funciona. Si la función delegado devuelve un array de bytes (lo esperado), simplemente usamos los buses del procesador (y sus registros!!) para ir escribiendo los contenidos en la memoria del Spectrum. Cualquier excepción en la función será interpretada como un error que llegará al Spectrum como un “error de cinta”. Una vez completado el trabajo, volvemos a colocar el registro PC de la cpu a la dirección de programa donde todo debería continuar normalmente de emplear una unidad de cinta “de verdad”.

Ficheros .TAP en C#

La parte pesada de todo ésto (la manipulación de los archivos y sus bloques propiamente dicha) la vamos a implementar en C#, porque precisamente para éstas cosas es el lenguaje ideal. La implementación es como sigue:

```

class TapBlock
{
    public TapBlock(BinaryReader stream)
    {
        // A TAP block consists of a two byte header plus data bytes.
        // Data is raw data as saved by Spectrum - and as expected
        // by ROM routines we are replacing.
        UInt16 size = stream.ReadUInt16();
        data = stream.ReadBytes(size);
    }

    public Byte[] Data
    {
        get
        {
            return (data);
        }
    }

    protected byte[] data = null;
}

class TapFile
{
    public void Open(String fileName)
    {
        System.Collections.Generic.List<TapBlock> blockCol = new System.Collections.Generic.List<TapBlock>()

        try
        {
            using (var file = File.OpenRead(fileName))
            {
                var stream = new BinaryReader(file);
                while (stream.BaseStream.Position < stream.BaseStream.Length)
                {
                    var newBlock = new TapBlock(stream);
                    blockCol.Add(newBlock);
                }
            }
        }
        catch (Exception) { }

        blocks = blockCol.ToArray();
    }

    public TapBlock[] Blocks
    {
        get
        {
            return (blocks);
        }
    }

    protected TapBlock[] blocks = null;
}

```

Realmente no es más que eso: un objeto `TapFile` abre el archivo en disco y lee todos los bloques, añadiéndolos a una colección de objetos `TapBlock` que obtenemos de su propiedad `Blocks`. Cada bloque simplemente contiene los datos en su miembro `Data`.

Cómo no, lo siguiente es que nuestro emulador pueda leer alguna cinta y empezar a hacer cosas verdaderamente divertidas con nuestro Spectrum. Para ello, vamos a añadir un menú a nuestro emulador con la opción “Load”. En el archivo `MainWindow.xaml` añadimos lo siguiente a nuestra ventana:

```

<Grid>
<Grid.RowDefinitions>
<RowDefinition Height="Auto"/>
<RowDefinition/>

```

```

</Grid.RowDefinitions>
<Menu IsMainMenu="True" Grid.Row="0">
  <MenuItem Header="File">
    <MenuItem Header="Load tape..." Click="OnSelectTape" />
  </MenuItem>
</Menu>
<zx:Spectrum x:Name="emulator" Grid.Row="1" />
</Grid>

```

Hemos añadido un simple menú con la opción de leer cintas. Ahora en el archivo de código `MainWindow.xaml.cs` añadimos lo siguiente:

```

public partial class MainWindow : Window
{
    TapFile currentTape = null;
    int currentTapeBlock = 0;
    ...

    private void OnSelectTape(object sender, RoutedEventArgs e)
    {
        OpenFileDialog dialog = new OpenFileDialog();
        dialog.AddExtension = true;
        dialog.CheckFileExists = true;
        dialog.CheckPathExists = true;
        dialog.Filter = "Tape files (*.tap)|*.tap";
        dialog.DefaultExt = "tap";
        bool? result = dialog.ShowDialog(this);
        if (result == null)
            return;
        if ((bool)result == false)
            return;

        try
        {
            var tape = new TapFile();
            tape.Open(dialog.FileName);
            currentTape = tape;
            currentTapeBlock = 0;
        }
        catch (Exception)
        {
            MessageBox.Show("Unable to load tape file");
        }
    }
}

```

En el delegado `OnSelectTape()` (invocado al seleccionar la opción de menú) creamos una instancia de la clase `TapFile` que obtiene los contenidos del archivo y la asignamos a una variable que leeremos bloque a bloque a medida que el emulador nos lo vaya pidiendo. Sólo queda activar la intercepción de la carga de bloques:

```

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

        // Set up load traps
        emulator.OnLoad = TapeLoad;

        emulator.Start();
    }

    protected Byte[] TapeLoad(int bytesToLoad)
    {
        TapBlock currentBlock = currentTape.Blocks[currentTapeBlock++];
        return (currentBlock.Data);
    }
}

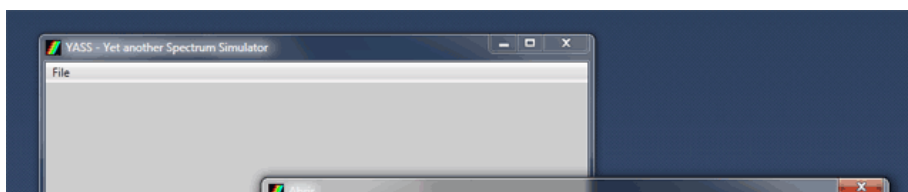
```

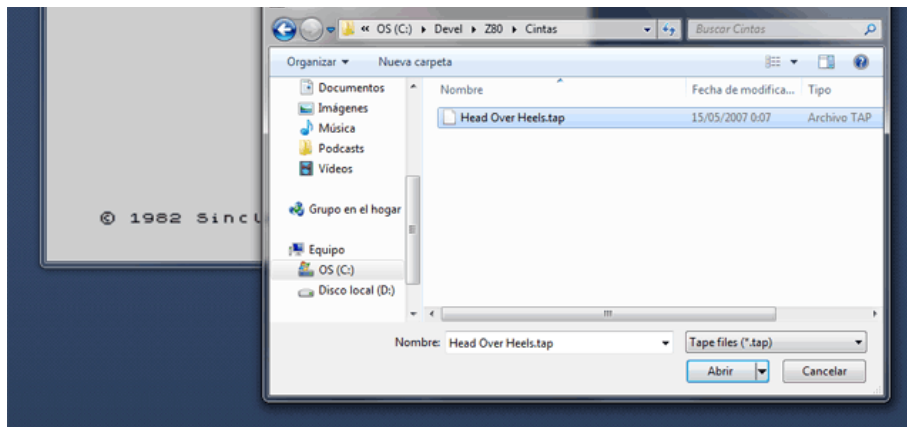
En el constructor conectamos el delegado del emulador a nuestra implementación (la función `TapeLoad()`). La función intentará leer los bloques secuencialmente (empleando la variable `currentTapeBlock` como índice) y devuelve directamente el array de bytes al control.

Lo siguiente es una nueva visita (cómo no!) al inmenso [archivo de programas de WorldOfSpectrum](#) y descargar nuestros juegos favoritos. En dos clicks encontramos el archivo TAP de *Head Over Heels*. Vamos a probar el emulador!!

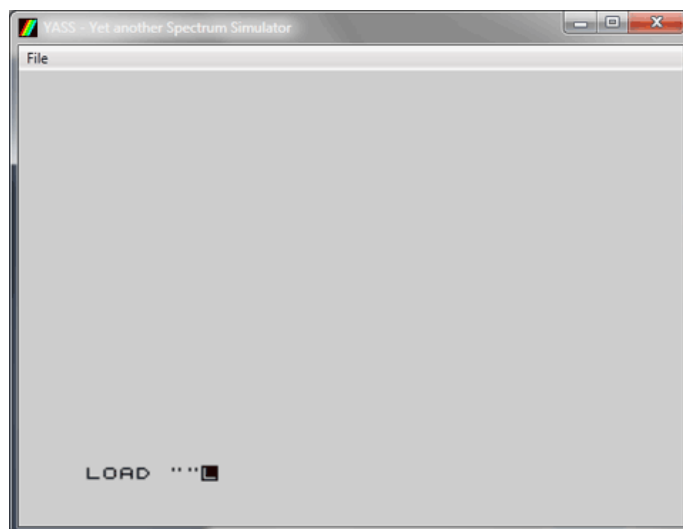
A jugar!

Una vez arrancado el emulador, abrimos el menú y seleccionamos la opción de cargar un archivo. Simplemente buscamos la ubicación del archivo TAP y lo abrimos.





El hecho de cargar una cinta en el programa no da ninguna indicación al emulador de que la lea – al igual que en un Spectrum real, tras insertar la cinta de cassette requiere que el usuario inicie la carga de la misma. Tecleamos `LOAD ""` en el emulador (tecla “J” para “LOAD” y CTRL+P para las comillas) y el emulador se pondrá a cargar la cinta que le hemos preparado:



Tras pulsar retorno de carro, nuestro emulador pedirá bloque a bloque los contenidos de la cinta... Y el resultado es éste:



Jugar con el emulador es divertido, pero he perdido (hace mucho!!) la costumbre de usar el teclado y más todavía empleando las combinaciones de teclas que se usan en el Spectrum. Como remate final a nuestro emulador, en el próximo capítulo vamos a añadir lo que creo que pueden ser los dos únicos componentes que quedan para completarlo: la emulación de un joystick Kempston y el sonido! Y no nos vamos a quedar en emular el “beep” tradicional del Spectrum, sino que vamos a soportar (y emular!) los programas que generaban audio multicanal usando un solo bit: el audio PWM.



Hazte un Spectrum - Capítulo 7

Ya casi llegamos al final y hoy vamos a añadir dos componentes a nuestro emulador: la capacidad de hacer “ruido” y la posibilidad de emular un joystick Kempston.

Y para comenzar, vamos con lo facilito primero: el joystick. El hardware de un joystick Kempston es lo más simple del mundo: solamente un registro de I/O que se direcciona poniendo el bit de dirección 5 a cero – muy parecido al direccionamiento de un bit de la ULA del Spectrum. Al leer el registro (típicamente de 0x1F == 00011111b por ejemplo) se obtiene un byte donde los cinco bits menos significativos son el estado de las cinco posiciones del joystick (arriba, abajo, izquierda, derecha y “fuego”):

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	FIRE	UP	DOWN	LEFT	RIGHT

Dado que hasta ahora estábamos conectado la ULA directamente al bus de I/O de la cpu Z80, ya no puede ser el caso, y necesitamos un decodificador intermedio. La clase hospeda los componentes (ULA y Kempston), decodifica la dirección y reenvía la petición al controlador apropiado:

```
class SpectrumIOBus : public BusComponent<0x00,0x10000>
{
public:
    SpectrumIOBus()
    {
        ULA = NULL;
        Kempston = NULL;
    }

    unsigned char Read(unsigned int address)
    {
        // Kempston interface is addressed by A5 == 0
        if (Kempston && ((address & 0x20) == 0))
            return(Kempston->Read(address));

        // ULA is addressed by A0 == 0
        if (ULA && ((address & 0x01) == 0))
            return(ULA->Read(address));

        // No device
        return(0xFF);
    }

    void Write(unsigned int address,unsigned char value)
    {
        // ULA is addressed by A0 == 0
        if (ULA && ((address & 0x01) == 0))
            return(ULA->Write(address,value));
    }

public:
    BusComponentBase *ULA;
    BusComponentBase *Kempston;
};
```

Si os fijáis, en el `write()` del bus ni siquiera nos molestamos en enviar las escrituras al controlador Kempston...

La implementación de la clase Kempston es igual de simple:

```
#define KEMPSTON_RIGHT    0x01
#define KEMPSTON_LEFT    0x02
#define KEMPSTON_DOWN    0x04
#define KEMPSTON_UP      0x08
#define KEMPSTON_FIRE    0x10
```

```

class Kempston : public BusComponent<0x1F,1>
{
public:
    Kempston()
    {
        kempstonData = 0;
    }

public:
    void SetKempstonData(unsigned char data)
    {
        kempstonData = data;
    }

public:
    unsigned char Read(unsigned int address)
    {
        return(kempstonData);
    }

    void Write(unsigned int address,unsigned char value)
    {
        // Do nothing
    }

    unsigned char kempstonData;
};

```

La variable miembro `kempstonData` (pública) es donde la aplicación host establece el estado del joystick, y la clase simplemente conserva el valor y lo devuelve a la cpu cada vez que nos lo pida. Finalmente, en la clase `Spectrum` vamos a modificar los buses de nuestro procesador para integrar el nuevo decodificador de I/O y agregar el interface `Kempston`:

```

void Spectrum::emulatorMain()
{
    // Components
    Bus16 bus;           // Main Z80 data bus
    SpectrumIOBus ioBus; // Main Z80 I/O bus
    ROM<0,16384> rom;     // Main system ROM
    ULA ula;             // Spectrum's ULA chip + 16KB
    RAM<32768,32*1024> ram; // Remaining RAM (32KB)
    Kempston kempston;   // Kempston joystick
    Z80 cpu;             // and finally, the CPU core

    // Load ROM contents
    if (rom.Load("48.rom"))
        throw gcnew System::IO::FileNotFoundException("Unable to load rom '48.ROM'");

    // Configure the ULA with the native Windows bitmap used to emulate the screen
    ula.SetNativeBitmap((LPBYTE)screenData,bytesPerScreenLine);

    // Populate buses
    bus.AddBusComponent(&rom);
    bus.AddBusComponent((ULAMemory*)&ula);
    bus.AddBusComponent(&ram);

    ioBus.ULA = (ULAIOW*)&ula;
    ioBus.Kempston = &kempston;

    // And attach busses to cpu core
    cpu.DataBus = &bus;
    cpu.IOBus = &ioBus;

    pCurrentUla = &ula;
    pCurrentKempston = &kempston;
}

```

Con una enumeración de .NET y un par de funciones, tenemos el API para poder usar el joystick desde una aplicación host:

```

namespace Emulation
{
    // Kempston joystick status values.
    [Flags]
    public enum class KempstonFlags
    {
        Right = KEMPSTON_RIGHT,
        Left = KEMPSTON_LEFT,
        Down = KEMPSTON_DOWN,
        Up = KEMPSTON_UP,
        Fire = KEMPSTON_FIRE
    };

    public ref class Spectrum : System::Windows::Controls::Border, IDisposable
    {
    ...
    public:
        void SetKempstonStatus(KempstonFlags status)
        {
            if (pCurrentKempston != nullptr)
                pCurrentKempston->SetKempstonData((unsigned char)status);
        }

    private:
        ULA *pCurrentUla;
    };
}

```

```

    Kempston *pCurrentKempston;
}

```

Sólo queda ampliar nuestra gestión del teclado en C# para mapear las teclas de cursor y "ctrl" (por ejemplo) al joystick Kempston (nuevamente, tanto en `OnKeyDown()` como en `OnKeyUp()`):

```

Emulation.KempstonFlags kempston;

protected override void OnKeyDown(System.Windows.Input.KeyEventArgs e)
{
    base.OnKeyDown(e);
    switch (e.Key)
    {
        ...
        // Kempston
        case System.Windows.Input.Key.Left: kempston |= Emulation.KempstonFlags.Left; break;
        case System.Windows.Input.Key.Right: kempston |= Emulation.KempstonFlags.Right; break;
        case System.Windows.Input.Key.Up: kempston |= Emulation.KempstonFlags.Up; break;
        case System.Windows.Input.Key.Down: kempston |= Emulation.KempstonFlags.Down; break;
        case System.Windows.Input.Key.LeftCtrl: kempston |= Emulation.KempstonFlags.Fire; break;
    }
    emulator.SetKempstonStatus(kempston);
}

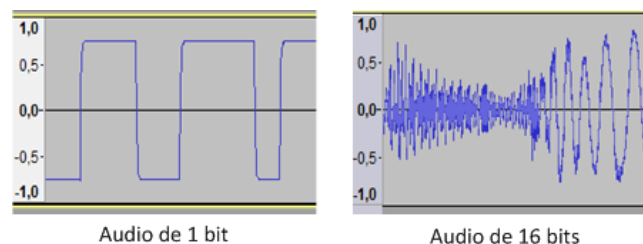
protected override void OnKeyUp(System.Windows.Input.KeyEventArgs e)
{
    base.OnKeyUp(e);
    switch (e.Key)
    {
        ...
        // Kempston
        case System.Windows.Input.Key.Left: kempston &= ~Emulation.KempstonFlags.Left; break;
        case System.Windows.Input.Key.Right: kempston &= ~Emulation.KempstonFlags.Right; break;
        case System.Windows.Input.Key.Up: kempston &= ~Emulation.KempstonFlags.Up; break;
        case System.Windows.Input.Key.Down: kempston &= ~Emulation.KempstonFlags.Down; break;
        case System.Windows.Input.Key.LeftCtrl: kempston &= ~Emulation.KempstonFlags.Fire; break;
    }
    emulator.SetKempstonStatus(kempston);
}

```

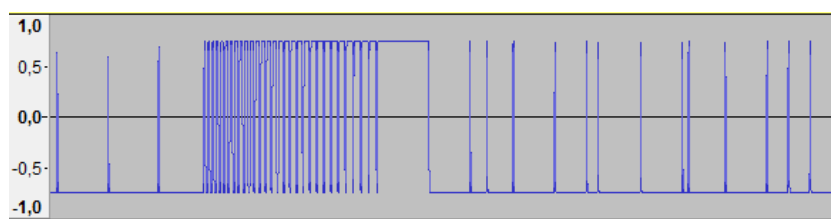
Vamos finalmente con la parte de audio que, lamentablemente, es algo más compleja que la implementación del interface Kempston. Veamos primero un poco de la teoría

Sonido de 1 bit?

Aunque siendo pragmáticos el sonido del Spectrum realmente son "2" bits: uno para manipular el altavoz y otro para la generación de audio para la unidad de cassette, la única diferencia entre ambos es una resistencia para limitar la amplitud de la segunda señal. El audio de "1 bit" sólo permite establecer dos niveles de audio, frente a los 65536 que se pueden definir con (por ejemplo) una moderna tarjeta de sonido de 16 bits. Usando Audacity (un estupendo editor de audio open source) podemos comparar audio de 1 bit con audio de 16 bits:



Con una limitación como ésta, parecería imposible que el Spectrum pudiese producir nada más que tristes pitidos, verdad? Pues no: utilizando pulsos escrupulosamente medidos (codificación PWM o "Pulse Width Modulation") se puede obtener un audio que simule características muy avanzadas: voces múltiples, control de volumen, etc. El truco del audio PWM es que, ya que no puedo "colocar el altavoz en la posición que yo quiero", lo que sí puedo hacer es dar "empujoncitos" para aproximarlos. El símil que se me ocurre es lo que hacemos en los juegos de conducción: aunque los controles "izquierda" o "derecha" sean botones y un volante real sea "analógico", puedo dar toques, más o menos largos, a los controles de izquierda o derecha para aproximar la posición del volante a la deseada. Fijaos en el siguiente gráfico de una demo del Spectrum ("Garfield"):



Audio PWM en estado puro!! El programa cuenta ciclos para subir o bajar la señal de audio y controlar el ancho de los pulsos... Luego nuestro cerebro hace el

resto – y creedme que el resultado es sorprendente!!

Generación de audio

Ya conocemos un poco la teoría, vamos ahora a la práctica. En el ZX Spectrum, para generar audio se usan los bits 4 y 3 de la ULA, escribiendo (por ejemplo) en la posición 0xFE de I/O. Cuando se manipula el bit 4 el nivel de la señal es muy amplio (más “alta”) mientras que el bit 3 se usa para modular la señal que se dirige a la cinta de cassette. Parece simple entonces que, cada vez que nos llegue una escritura a la ULA podamos calcular el nivel de la señal. Pero no nos basta con una muestra: necesitamos generar una señal en el tiempo, o sea que necesitamos muchas muestras. Pero cuantas?

Recordemos que la emulación deja correr al procesador durante un cuadro de pantalla (1/50 de segundo, o lo que es lo mismo 20 milisegundos) para luego hacer una pausa y “sincronizar” la velocidad del Spectrum emulado en nuestro rápido PC. Podemos trabajar en bloques de audio de 1/50 de segundo aprovechando el contador que usamos en la ULA para saber nuestra posición en pantalla.

Ahora sabemos que necesitamos 20 milisegundos de audio por cada cuadro de pantalla. Cuantas muestras son esas? un par de cálculos nos lo dirán: vamos a generar audio a 48KHz, esto es, 48000 muestras por segundo. Eso son $(48000/50) = 960$ muestras cada 20 milisegundos. Pero qué valor le damos a esas muestras?

No podemos predecir cuándo van a escribir los programas en los controles de audio del chip ULA, pero sí sabemos la posición (en ciclos) con respecto a la pantalla. Aprovechando la función `AddCycles()` de la ULA, podemos actualizar la muestra que corresponda con la posición del tubo CRT:

```
#define AUDIO_SAMPLE_RATE      (48000)
#define SAMPLES_PER_FRAME     (AUDIO_SAMPLE_RATE / 50)

class ULA : public ULAMemory, ULAIO
{
    unsigned char ULAIOData;
    short FrameAudio[SAMPLES_PER_FRAME];

    ...
    void AddCycles(unsigned int cycles, bool& IRQ)
    {
        dwFrameTStates += cycles;
        dwScanLineTStates += cycles;

        // Update the analog audio output from ULA
        // First, compute audio output value for this cycle
        int signal = 0;
        signal = (ULAIOData & 0x10) ? +16384 : -16384;
        signal += (ULAIOData & 0x08) ? +8192 : -8192;

        // Now, add audio output over an 8 tap filter:
        // 1: Maintain 7/8ths of the original signal
        audioOutput -= (audioOutput / 8);
        // 2: ...and add 1/8th of the new one
        audioOutput += (signal / 8);

        // Update the audio sample corresponding to this screen tState
        unsigned int offset = (dwFrameTStates * SAMPLES_PER_FRAME) / (TSTATES_PER_SCANLINE * TVSCANLINES);
        // As clocks don't match and this is a quick approximation, limit offset output
        if (offset < SAMPLES_PER_FRAME)
            FrameAudio[offset] = audioOutput;

        if (dwScanLineTStates > TSTATES_PER_SCANLINE)
            ScanLine(IRQ);
    }

    void IOWrite(unsigned int address, unsigned char value)
    {
        ...
        // save a copy of ULA's state
        ULAIOData = value;
        ...
    }
}
```

La función `AddCycles()` se invoca después de cada instrucción ejecutada por el procesador, y recorrerá de forma más o menos uniforme los casi 69000 ciclos que forman la pantalla. Dado que 69000 ciclos entre 960 muestras dan para muchas repeticiones en cada muestra de audio, la función integra repetidamente la señal con un filtro de 8 taps, esto es, sólo 1/8 de la señal de agrega en cada interacción, algo así como un “condensador en software”. De esta forma, los rápidos cambios en los controles de audio de la ULA se convierten en señales un poco más “analógicas” que, posteriormente, la tarjeta de sonido terminará de digerir. Para la amplitud de la señal he usado +/-16384 para la señal del altavoz y +/- 8192 para la del cassette. Arbitrario? Totalmente.

Ahora que ya tenemos la muestra de 20 milisegundos de audio, es necesario enviarla a la tarjeta de sonido para que la reproduzca, y es aquí donde aparece el mayor problema (con diferencia) para la emulación: cómo sincronizar a la perfección el Spectrum con el audio para que no dé tirones ni cortes.

Para mí la solución consiste en emplear el cronómetro más exacto que se pueda conseguir en un PC.

El temporizador más preciso de un PC: la tarjeta de sonido

Hasta ahora, la solución para “ralentizar” el Spectrum a su velocidad real era ver el tiempo que habíamos necesitados para correr durante 20 milisegundos y

complementarlo con un `sleep()` del sistema operativo. Aunque funciona (y es válido si no hay sonido), hay que tener en cuenta que la mayoría de los sistemas operativos no son realmente “de tiempo real”: una instrucción `sleep(17)` no dura realmente 17 milisegundos, sino cualquier cosa entre 1 (o 0!!) y 30 (o más), dependiendo de la resolución de los “timers” del hardware, de lo ocupada que está la cpu, etc. Cómo conseguimos sincronizar entonces nuestro audio?

Aunque parezca mentira, el mejor cronómetro de un PC moderno es su tarjeta de sonido. En la tarjeta de sonido, 20 milisegundos son realmente 20 milisegundos. De hecho, los reproductores multimedia sincronizan los streams de audio y vídeo empleando el audio como reloj maestro. Nosotros vamos a hacer lo mismo.

Dado que generamos bloques de audio de 20 milisegundos, en lugar de intentar cronometrar nosotros mismos el retardo vamos a emplear el evento de la tarjeta de sonido, enviándole bloques de 20 milisegundos y “congelando” la emulación hasta que nos avise que ya se han terminado de reproducir. Obviamente, tenemos el problema de que cuando “termine de reproducirlos” vamos a tardar un poco en generar otros 20 milisegundos de audio, así que para solucionarlo vamos a utilizar tres buffers de 20 milisegundos que enviaremos al sistema de audio de golpe. A medida que se vayan consumiendo, tendremos 40 milisegundos de audio todavía pendiente de reproducir para seguir emulando instrucciones y alimentando al sistema de sonido. El resultado? Audio de bajísima latencia sin cortes ni chasquidos.

Cualquiera que haya programado el sistema de audio de Windows sabe las complicaciones que conlleva la reserva de los bloques de memoria, las cabeceras, abrir y cerrar los dispositivos de audio... Vamos a hacerlo encapsulando toda la funcionalidad en una sola clase (bueno, en un par) que nos evite perder el enfoque en nuestro principal problema, que es la emulación. Aquí va el código completo del sistema de audio:

```
#pragma comment(lib, "WinMM.lib")

template<unsigned int sampleRate> struct AudioBlock
{
    WAVEHDR Header;
    short AudioData[sampleRate / 50];
};

#define AUDIOBLOCK_COUNT (3)

template<unsigned int sampleRate> class WaveOutAudioManager
{
public:
    WaveOutAudioManager()
    {
        hWaveDev = NULL;
        hWaveEvent = INVALID_HANDLE_VALUE;
    }
    virtual ~WaveOutAudioManager()
    {
        Close();
    }

public:
    int Open()
    {
        if (hWaveDev != NULL)
            return(ERROR_INVALID_STATE);

        if (hWaveEvent != INVALID_HANDLE_VALUE)
            CloseHandle(hWaveEvent);
        hWaveEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

        WAVEFORMATEX wf;
        ZeroMemory(&wf, sizeof(wf));
        wf.cbSize = sizeof(wf);
        wf.wFormatTag = WAVE_FORMAT_PCM;
        wf.nChannels = 1; // mono audio
        wf.nSamplesPerSec = sampleRate;
        wf.nAvgBytesPerSec = sampleRate * sizeof(short);
        wf.nBlockAlign = sizeof(short);
        wf.wBitsPerSample = 8 * sizeof(short);
        int result = waveOutOpen(&hWaveDev, WAVE_MAPPER, &wf, (DWORD_PTR)hWaveEvent, NULL, CALLBACK_EVENT);
        if (result != 0)
        {
            Close();
            return(result);
        }

        // Prepare audio buffers. Each one will be 20ms long
        for (int dd=0; dd<AUDIOBLOCK_COUNT; dd++)
        {
            ZeroMemory(&audioBlocks[dd], sizeof(AudioBlock<sampleRate>));
            audioBlocks[dd].Header.lpData = (LPSTR)&audioBlocks[dd].AudioData[0];
            audioBlocks[dd].Header.dwBufferLength = sizeof(audioBlocks[dd].AudioData);
            result = waveOutPrepareHeader(hWaveDev, &audioBlocks[dd].Header, sizeof(audioBlocks[dd].Header));
            if (result != MMSYSERR_NOERROR)
            {
                Close();
                return(result);
            }
        }

        // Everything ready...
        return(0);
    }

    void Close()
    {
        if (hWaveDev != NULL)
```

```

    {
        waveOutReset(hWaveDev);
        waveOutClose(hWaveDev);
        hWaveDev = NULL;
    }

    if (hWaveEvent != INVALID_HANDLE_VALUE)
    {
        CloseHandle(hWaveEvent);
        hWaveEvent = INVALID_HANDLE_VALUE;
    }
}

protected:
AudioBlock<sampleRate> *FindAvailableBlock()
{
    for (int dd=0;dd<AUDIOBLOCK_COUNT;dd++)
    {
        if ((audioBlocks[dd].Header.dwFlags & WHDR_INQUEUE) == 0)
            return(&audioBlocks[dd]);
    }
    return(NULL);
}

public:
bool QueueAudio(short* audioData)
{
    // Check if any block is available right now
    AudioBlock<sampleRate> *block = FindAvailableBlock();
    // If no audio block is available, just wait for one (max 20ms)
    if (block == NULL)
    {
        WaitForSingleObject(hWaveEvent,1000);    // Wait for 1 second max
        block = FindAvailableBlock();
    }

    // This is weird... something really strange happened.
    if (block == NULL)
        return(false);

    // Fill block with audio data
    for (int dd=0;dd<sampleRate/50;dd++)
        block->AudioData[dd] = audioData[dd];
    // and send again to audio subsystem
    int result = waveOutWrite(hWaveDev,&block->Header,sizeof(block->Header));
    return(result == MMSYSERR_NOERROR ? true : false);
}

protected:
HANDLE hWaveEvent;
HWAVEOUT hWaveDev;
AudioBlock<sampleRate> audioBlocks[AUDIOBLOCK_COUNT];
};

```

La clase WaveOutAudioManager<> inicializa el sistema de sonido y tres buffers (de 20 milisegundos de audio). La función que nos interesa es QueueAudio(), que recibe 20 milisegundos de audio desde el bucle principal de la emulación. Si encuentra buffers disponibles (que no estén en ejecución) la función envía el bloque al sistema de audio para su reproducción y vuelve de inmediato. Si todos los buffers están encolados, la función espera pacientemente a que el sistema de audio nos notifique por medio de un evento que uno de los bloques se ha completado, encolando un nuevo bloque.

Para los buffers y cabeceras empleo una clase template (AudioBlock<>) que agrega las dos estructuras como una sola, simplificando enormemente el código y haciendo innecesario el uso de reserva dinámica de memoria.

Volviendo ahora a la función principal de emulación (emulatorMain), el código queda así:

```

void Spectrum::emulatorMain()
{
    ...
    // Native audio driver
    WaveOutAudioManager<AUDIO_SAMPLE_RATE> audMgr;

    ...

    audMgr.Open();

    // Ready to roll!!
    do
    {
        // Tape load trap
        if ((cpu.regs.PC == 0x056B) && (OnLoad))
        {
            ....
        }

        // And emulate next instruction
        cpu.tStates = 0;
        cpu.EmulateOne();

        // After each instruction, report the ULA the number of cycles we've used
        bool irq = false;
        ula.AddCycles(cpu.tStates,irq);

        // As in the real Spectrum, the ULA will trigger an IRQ for every frame. This

```

```

// implementation uses cpu clock cycles to know where the screen beam is.
if (irq) // Ula signals a frame interrupt
{
    cpu.INT(); // Generate system interrupt

    // If screen contents have been modified, set a flag for the WPF rendering event.
    if (ula.IsDirty)
    {
        screenDataDirty = true;
        ula.IsDirty = false;
    }

    // Submit audio to driver. The "driver" will hold the loop until an audio buffer
    // is available - that is, 20 ms. This will match the speed of our emulated
    // Spectrum to the real one.
    audMgr.QueueAudio(ula.FrameAudio);
}
} while(quitEmulation == false);

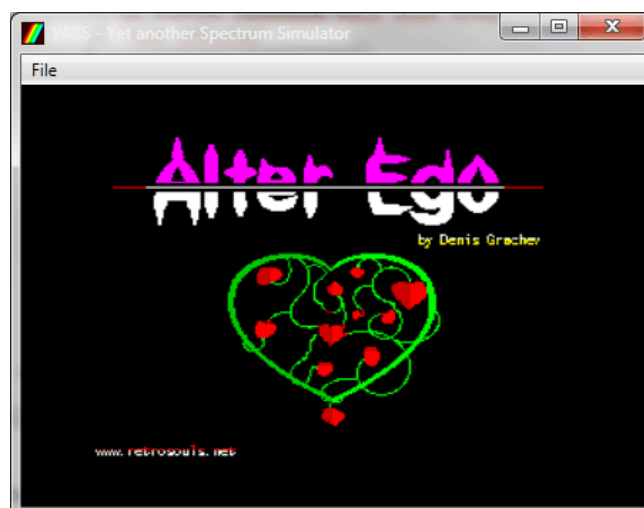
pCurrentUla = nullptr;
pCurrentKempston = nullptr;
}

```

El resultado? Probad simplemente a arrancar el emulador, escribir un programa de un par de líneas y salvarlo con SAVE "test"... veréis las familiares líneas en el borde de la pantalla, junto con el igualmente familiar sonido de la grabación, ese sonido que nos acompañó durante largas tardes de juego.



Pero si queréis ver realmente de lo que era capaz el Spectrum, os recomiendo como un ejemplo absolutamente encantador la música de Alter Ego y de lo que se puede hacer un audio "de un solo bit".



Pues creo que eso es todo...

Nuestro emulador está completo, y tenemos un flamante ZX Spectrum de 48KB, capaz de generar sonido (y recuerdos) y un estupendo joystick que podéis adaptar a todas vuestras necesidades. En el último capítulo de este tutorial daremos los últimos toques a la emulación... y podréis descargar el código fuente completo de la misma.



Hazte un Spectrum - Capítulo 8

Todos los componentes están realizados y en su sitio, el código ha ido tomando forma con una maravillosa fluidez y he aprendido lo indecible durante estas breves semanas. Aprovecho que mañana vuelvo a la dura vida cotidiana para cerrar esta serie de artículos – unos remates finales y completar el código fuente.

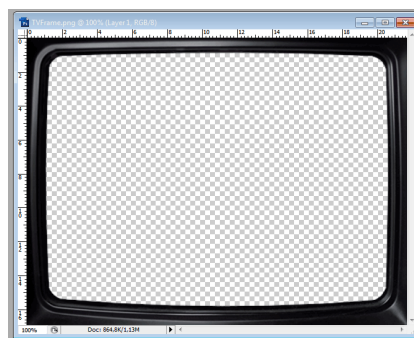
Hemos llegado al final del camino – unas semanas que me han servido para conocer un poco más uno de los mitos de la informática personal de los años '80 y que para mí, hasta hace unas semanas, era casi un completo desconocido. Hemos empezado por diseñar una máquina virtual, ir aprendiendo cómo funcionaban los diferentes componentes de la misma y emulando su comportamiento. Hoy concluye este fascinante viaje. Vamos a divertirnos un poco y darle incluso un aire más “retro” al emulador, aprendiendo de paso cómo aprovechar las capacidades de aceleración hardware de gráficos de WPF.

Vamos a empezar con una herramienta imprescindible para diseñar interfaces de usuario en aplicaciones modernas: Photoshop. Con ella vamos a realizar una capa gráfica (“overlay”) que podremos activar a voluntad para darle ese toque retro, imprescindible para mi: el “efecto TV” o, como me ha dado por llamarlo en el emulador, el “modo Telefunken” (y apuesto a que mi hermano estará de acuerdo conmigo a que el nombre le viene que ni pintado gracias a nuestras experiencias de chavales).

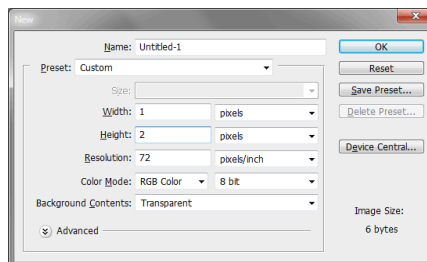
Una nueva herramienta de desarrollo: Photoshop

Comencemos (sorprendentemente) buscando en Google Images por un fondo apropiado. Nada más fácil que navegar <http://images.google.com> y buscar por “TV frame”. Busco un “marco de televisor”, y he escogido el que me ha parecido más simpático, que incluso ya viene con el hueco transparente. Veamos los pasos siguientes, uno a uno:

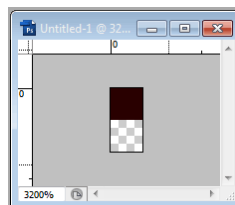
El primer paso es abrir la imagen que más nos guste en Photoshop y ajustar su tamaño a 640x480. Invocando el menú Image/Image Size podremos ajustarlo. Si habéis cargado la imagen como “Background”, haced doble click sobre ella para convertirla en una capa normal (Layer). Ponedle como nombre “TV Frame” por ejemplo y borrad todo lo que pudiese verse dentro de la imagen para que quede transparente:



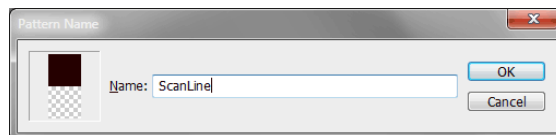
Vamos a crear una nueva imagen, con un tamaño algo peculiar: 1 pixel de ancho por 2 de alto. Esta imagen nos servirá como “relleno” para el familiar efecto de las líneas en pantalla



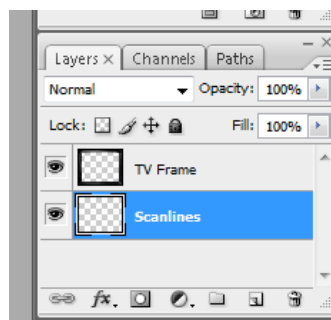
Tras crear la imagen, seleccionamos la herramienta “lápiz” y rellenamos el pixel de arriba en negro, dejando el pixel de abajo transparente



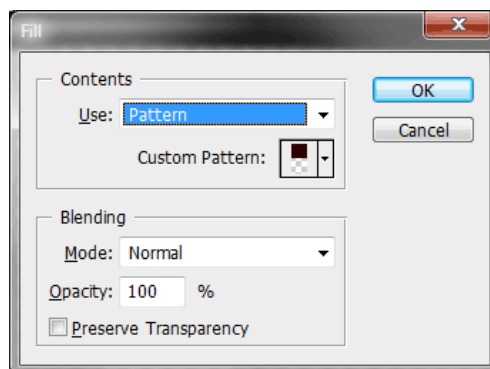
Seleccionad la imagen con Ctrl+A y luego Edit/Define Pattern. Yo le puse de nombre "ScanLine".



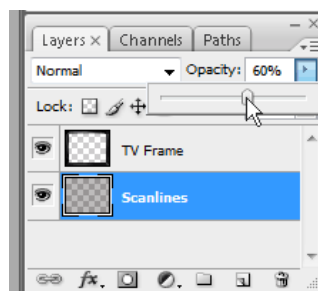
Volved ahora a la imagen principal con nuestro televisor. Añadid una nueva capa y aseguraos que queda debajo del marco del televisor. Ponedle de nombre por ejemplo "Scanlines":



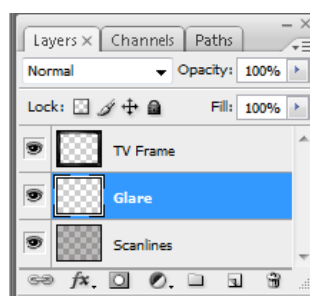
Aseguraos que tenéis la capa Scanlines seleccionada y rellenadla con el patrón que diseñamos antes. Pulsad mayúsculas+BackSpace, seleccionad "Pattern" de la lista de opciones y el patrón Scanlines como "Custom Pattern":



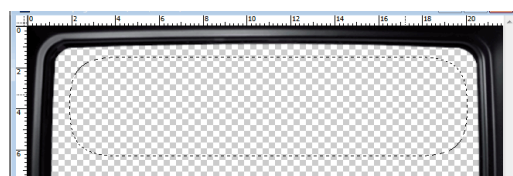
Ya tenemos nuestro familiar efecto de líneas de televisor... simplemente ajustad la opacidad de la capa "al gusto" (40-60%)



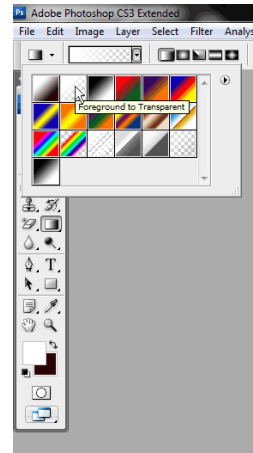
Agregad una nueva capa al diseño, y colocadla entre el marco del televisor y la capa de líneas. Yo la he llamado "Glare":



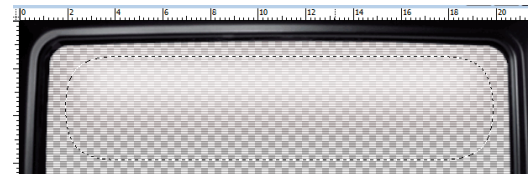
Seleccionad la capa y empleando la herramienta de selección, dibujad un rectángulo que ocupe más o menos el tercio superior de la pantalla. Luego redondead la selección con la herramienta Select/Modify/Feather y algo así como 25 píxeles:



Comprobad que vuestros colores son el blanco para la tinta y el negro para el fondo, seleccionad la herramienta de gradientes y en la barra superior escoged la configuración “Foreground to transparent” (esto es, de blanco – nuestro color actual – a transparente).



Ahora pulsad con el botón más o menos en la zona superior de la selección y arrastradla hasta un poco antes de la zona inferior. Luego ajustad un poco la opacidad de la capa, quizá entre 60-70%:



Ya tenemos el overlay listo – simplemente guardadlo dentro del proyecto del emulador como PNG de 32 bits (con transparencia!!) y ya podemos importarlo en el proyecto. Para ello pulsad con el botón derecho del ratón sobre el proyecto en el que queráis añadir la imagen y seleccionad “Add/Existing Item”. Buscad la imagen PNG, aceptad el diálogo y aseguraos que en las propiedades de la imagen, la entrada Build Action aparece configurada como “Resource”: el compilador “incrustará” la imagen dentro del binario, permitiéndonos olvidarnos de distribuir los ficheros sueltos junto a la aplicación.

Ahora abrid el archivo MainWindow.xaml, y añadid una nueva imagen justo debajo del control zx:Spectrum:

```
<zx:Spectrum x:Name="emulator" Grid.Row="1" />
<Image x:Name="tvOverlay" Grid.Row="1" Source="/YASS;component/Resources/TVEffect.png"
        Stretch="Fill" Visibility="Collapsed" />
```

La propiedad `source` especifica la imagen que debe emplear el control, en este caso `Source="/YASS;component/Resources/TVEffect.png"`. La primera parte (“/YASS;”) indica el nombre del ensamblado que lleva el archivo, en este caso “YASS.exe” que será el nombre del ejecutable. Lo que sigue es el camino (o “path”) hasta el recurso: en mi caso el archivo se llama “TVEffect.png” y está dentro de la carpeta “Resources” del proyecto. Si no queréis tener que averiguar el camino al archivo, insertad una imagen sin campo “Source” y luego simplemente usad las propiedades del control en Visual Studio. Navegad hasta la propiedad `Source` en la lista y seleccionad directamente la imagen con el menú desplegable – mucho más fácil que ponerlo a mano y sin errores!!! Fijaos que los dos elementos (el control `zx:Spectrum` y la imagen “`tvOverlay`”) están situados en la misma celda del Grid que los contiene. El efecto es que la imagen cubrirá permanentemente al control, sin tener que hacer absolutamente nada en código – solo cambiar su visibilidad.

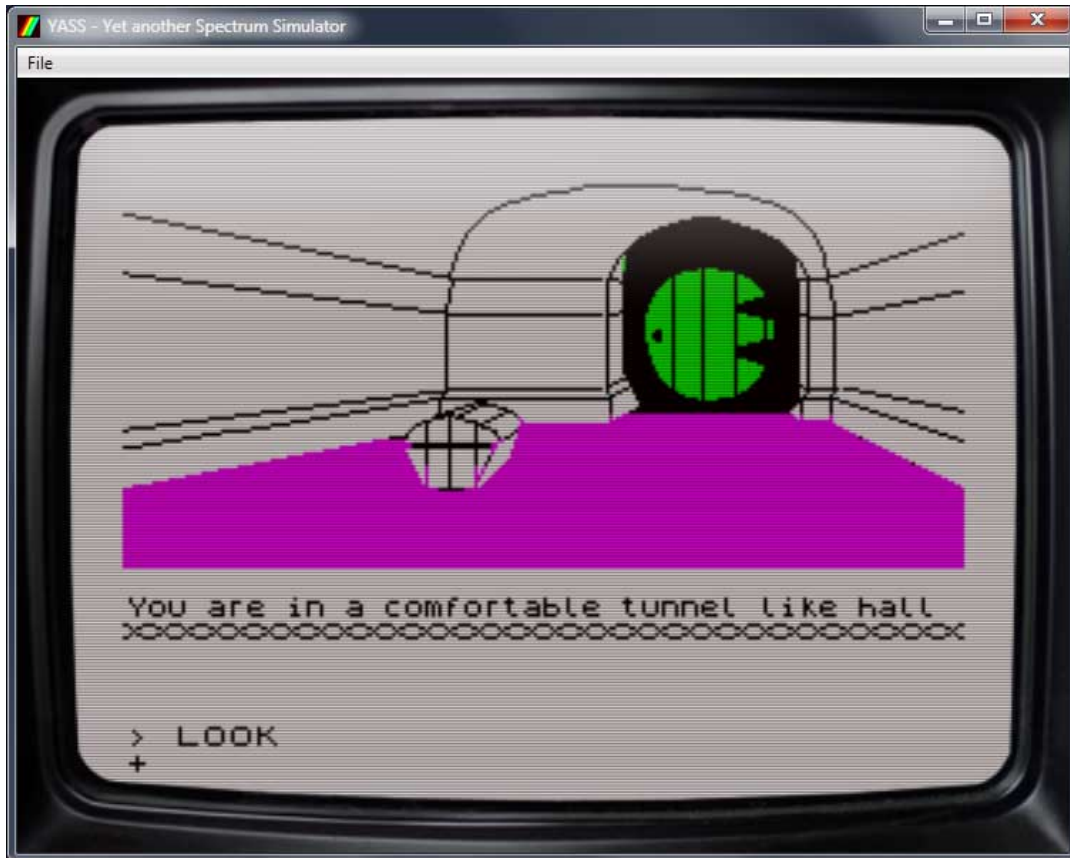
Aprovechad también para añadir al menú de la aplicación una entrada para activar o desactivar el “modo Telefunken”:

```
<Menu>
...
<MenuItem Header="Load tape..." Click="OnSelectTape"/>
<Separator/>
<MenuItem x:Name="tvModeMenu" Header="Telefunken mode" Click="OnTVMode" />
...
</Menu>
```

Finalmente, basta con agregar la función “`OnTVMode()`” al programa, que alternará la visibilidad de la capa:

```
private void OnTVMode(object sender, RoutedEventArgs e)
{
    // Invert visibility of tv overlay
    tvOverlay.Visibility = tvOverlay.Visibility == Visibility.Collapsed ? Visibility.Visible : Visibility.Collapsed;
    tvModeMenu.IsChecked = tvOverlay.Visibility == Visibility.Visible;
}
```

Si ejecutamos el emulador y seleccionamos el menú “Telefunken Mode”, el efecto será un emulador incluso más retro:



El overlay está acelerado por hardware, así que la capa no afecta en lo más mínimo a la velocidad de ejecución del emulador, ni incrementa el consumo de la cpu – la tarjeta gráfica hace todo el trabajo.

Alguna cosita más?

Pues si, un par de cosas que he añadido al código final antes de publicarlo. Lo más significativo es que al seleccionar un archivo de cinta ya no es necesario escribir `LOAD ""` para iniciar la carga. La solución, rápida y simple: una función que emula la pulsación de las teclas para escribir el comando tras abrir el archivo TAP. Muy simple, pero cumple con su cometido – con la pega de que necesitamos que el Spectrum esté dispuesto a aceptar el comando para que funcione. También encontraréis opciones para parar, arrancar y resetear el emulador – funciones que he empleado para depurar algunas cosas y asegurarme de que no me dejaba nada “tirado” por ahí.

Bueno... creo que eso es todo

El emulador está completo y el viaje ha llegado a su fin. Me he divertido mucho, realmente mucho durante estas semanas y me ha encantado intentar documentar todo el proceso. El emulador que os dejo está lejos de ser completo o de ejecutar “absolutamente todos los programas del Spectrum” – no había tiempo para tanto, siendo más bien mi intención la de ofreceros un tutorial completo sobre cómo desarrollar un emulador “desde cero”, cómo combinar varios lenguajes de programación para obtener los mejores resultados y, sobre todo, mostraros cómo era la informática de los años 80. He intentado mantener el código fuente lo más limpio y estructurado posible y creo que es bastante fácil de leer y, sobre todo, ampliar.

Podréis encontrar todos los contenidos de éste tutorial en la [zona de descargas](#).

Como os decía al principio, hace unas semanas no conocía apenas nada del Sinclair ZX Spectrum, y seguro que algún error habré cometido al documentar o emular alguna funcionalidad – espero que sepáis disculparme y, sobre todo, dejadme saber dónde he metido la pata – quiero seguir aprendiendo!

Mañana me toca volver a los quehaceres diarios, pero esta “retromanía” me sigue afectando... cual podría ser el próximo proyecto?