Utiliza tu MSX#1

Armando Pérez Abad

24 de septiembre de 2015

Índice general

Capítulo 1

Objetivo.

El objetivo del presente artículo es la creación de una ROM desde nuestro MSX, con:

⇒ **AS.COM** Macro Ensamblador de Egor Voznesensky (SOLID).

⇒ **LD.COM** Linkador de Egor Voznesensky.

Con las utilidades de apoyo, de creación propia:

⇒ **TOAS.COM** Exportador de formato ensamblador a compatible con `AS.COM`.

⇒ **SMAKE.COM** Pequeño y simple *Make*.

⇒ **ASIZE.COM** Alineador de tamaño ROM.

Y apoyándonos en una utilidad creada hace mucho tiempo:

⇒ **D2R.COM** Conversor de ficheros binarios a relocatables.

Cuando se me propuso la programación del videojuego ***Booming Boy*** y al conocer las características del mismo vi que se me presentaba una gran oportunidad para *volver a los orígenes*. Es decir, volver a programar desde el MSX, por puro placer.

Pero también me di cuenta que, después de tanto años sin hacerlo, que mis gustos y maneras de programar habían cambiado bastante. Y aunque la biblioteca de programas dedicados al desarrollo en MSX es inmensa no encontré la manera de montar con ellas mi *kit de desarrollo ideal*. Tocaba, para variar, pringar y hacerse uno mismo gran parte de esas utilidades o comandos.

Todas las utilidades, fuentes y ficheros necesarios que se han utilizado para este artículo, incluido el presente texto, los puedes encontrar aquí:
https://www.dropbox.com/s/irc5fh7racapb10/UTM.ZIP?dl=0

1.1. Qué se pretende.

Como toda la serie de artículos *Utiliza tu MSX* el único objetivo es animar a la gente a darle uso a su querida máquina. En **ningún momento** pretendemos convencer a nadie de que el MSX es mejor que ninguna máquina. De que es más rápido y cómodo hacerlo con MSX que con otra máquina. O, en definitiva, que nuestro MSX es lo que no es. No.

También queremos dejar claro que este artículo va dirigido a quien ya ha tenido contacto con la programación en ensamblador. En ningún momento nos encontramos ante un curso de programación en ensamblador.

1.2. Nomenclatura y síntaxis.

Para facilitar la comprensión del texto a partir de este momento cuando hablemos de *ensamblador* nos referiremos, a no ser que así se especifique en el texto, a `AS.COM`. Cuando hablemos de *linkador* estaremos hablando de `LD.COM`.

En cuanto a la sintaxis, los primeros ejemplos, y hasta que se diga lo contrario, utilizan la sintaxis que interpreta correctamente el ensamblador `AS.COM`. Seguidamente utilizaremos ya un formato propio que, gracias a `TOAS.COM`, exportaremos a la sintaxis necesaria para ser ensamblado por `AS.COM`.

Capítulo 2

El sistema modular.

Seguramente, los programadores acostumbrados a los ensambladores cruzados modernos, no se plantean, ni se han planteado jamás, el organizar el código en diferentes módulos. Normalmente se tiene *un único fichero* con todo el fuente.

Los más organizados seguramente separarán el fuente en varios ficheros, aunque con sentencias tipo `INCLUDE` que permiten estos ensambladores cruzados, al final se genera, de nuevo un único fichero que será ensamblado a la vez.

Y no es que esté mal. Aunque desde el punto de vista organizativo no es que sea lo mejor. Tiene varias desventajas:

- Cada vez que iniciamos un proyecto, debemos *copiar* todas esas partes (librerias, módulos...) a ese fichero único.
- Cada vez que se ensambla una versión, se ha de reensamblar, cuando no sería necesario. Esto *hoy* lógicamente carece de sentido ya que una máquina actual ensambla en microsegundos fuentes de miles de lineas.
- No podemos tener etiquetas locales para cada módulo. Tampoco podemos tener etiquetas privadas a un módulo.
- El código no es tan claro y limpio.

Después de muchos años cometiendo siempre errores (es la mejor manera de aprender), te das cuenta que la única manera de poder hacer asequible la programación directa en MSX es utilizar módulos.

Aunque sea pecar un poco de *abuelo cebolleta*, el programa más grande que he realizado íntegramente en el MSX fue ***KPI-Ball***. Tirando la vista atrás veo que aquello fue un poco suplicio. Utilicé, como casi todos los desarrolladores en esa época ***CompAss***, que, si, como development pack con IDE integrado no creo que exista nada mejor en MSX. Pero tiene un terrible problema con los fuentes y es que:

- El tamaño del fuente depende totalmente de la memoria del MSX.
- Para tener `includes` necesitas meterlos en otro buffer separado, que has de cargar.

En resumen, que no interactua con el disco, imagino que por cuestiones de velocidad. Y, claro, como editor es muy bueno, pero podría no ser del agrado de todo el mundo. Permite, eso si, .REL (de los que vamos a hablar ahora mismo), aunque desconozco si puede *linkarlos*.

Muchos disgustos tuve en aquel desarrollo... perdida de fuentes... lentitud al ensamblar... Todo venía dado por la necesidad de ampliar la memoria del MSX con una memoria externa para tener más tamaño del buffer. Ya que el código de ***KPI-Ball***, al estar TODO junto, era monstruoso. Toda una odisea, que podría haberse evitado de haber tenido, por un lado más experiencia y por otro lado las herramientas de las que ahora si disponemos.

Pero, al margen de las historias pasadas... ¿qué diantres es un módulo?

2.1. Los módulos.

Un módulo no deja ser una porción de nuestro programa, que puede contener código o datos, y que uniremos, junto con otros, para crear el programa final.

Los acostumbrados a otros lenguajes, como C, C++, habrán trabajado, seguramente con *bibliotecas* donde tenemos por un lado las cabeceras (.h) y por otro la biblioteca (.lib, .o, ...). En el .h teníamos los nombres de cada función (prototipos) para poder acceder a ellos desde nuestro programa principal.

Pues un módulo no deja de ser eso. Solo que aquí, trabajando al más bajo nivel el fichero de cabeceras se define de manera diferente y el acceso desde nuestro programa también. Mucho más sencillo, incluso.

Los módulos son ensamblados independientemente. Cada módulo debe de ser ensamblable por si mismo y sin necesidad de otros. Aunque decir que lo *ensamblamos* no sea del todo correcto ya que lo que hacemos, con cada módulo, es generar un fichero .REL.

El ensamblador que vamos a utilizar, `AS.COM`, *solo genera ficheros .REL* que es un formato intermedio entre el fichero fuente y el programa final, denominados también ficheros objeto. A diferencia que los ensambladores cruzados modernos u otros como GEN80 o Compass (aunque también son capaces de generar .REL) no se genera el ejecutable (.COM) o programa final, cuando ensamblamos. Para ello se necesita un último paso, enlazar todos los ficheros .REL, que los ensambladores anteriores realizan automáticamente.

2.2. Ficheros .REL (RELOCATABLE FILES).

Un fichero .REL, como hemos dicho en el capítulo anterior, es un fichero que contiene información para el linkador: sus etiquetas públicas, externas, información de si es datos o código (o ambos)... No vamos aquí a definir el formato del mismo ya que no viene a cuento y no nos interesa.

Su nombre, .REL, viene de la palabra *relocatable*, o reubicable. Como puede adivinar el ávido lector un módulo puede ubicarse *en cualquier parte* del programa final. Y eso se lo hemos de decir nosotros con a la hora de enlazar.

2.3. El Linker.

Hemos hablado del mismo en los capítulos anteriores. Como hemos explicado un linker no deja de ser un programa que, a partir de unos módulos compilados ya en formato .REL (objetos) genera un programa final. Al mismo podemos indicarle normalmente el nombre del programa final, el **orden** de los módulos

y algunos datos que más adelante explicaremos.

Para entenderlo mejor: El linker lo que hace es casi lo mismo que haría un COPY /B de MSX-DOS. Concatenar uno detrás de otro todos los módulos para generar un programa. Esto a grandes rasgos, claro, ya que realmente realiza muchas más funciones.

Capítulo 3

Entendiendo los módulos.

Creo que no hay mejor manera de entender tanta teoría que con un pequeño ejemplo. Y a eso nos diponemos en este capítulo. Creemos dos ficheros fuente.

```
                LD      A,10
```

Listado 3.1: Fichero A.Z8A

```
                RET
```

Listado 3.2: Fichero B.Z8A

Simples, ¿verdad? Los hemos separado en dos, para entender mejor los módulos. Seguidamente ejecutemos `AS.COM` con cada uno de ellos.

```
A> AS A.Z8A
A> AS B.Z8A
```

Si todo ha ido bien, `AS.COM`, nos habrá generado dos ficheros:

```
A.REL
B.REL
```

Fantástico. Vamos a generar el programa final, vamos a *linkar*:

```
A> LD FINAL.COM=A.REL, B.REL
```

Y si todo ha ido como toca ya tendremos un flamante `FINAL.COM`. ¿Y qué tiene ese fichero? Pues si lo observamos con un editor hexadecimal encontraremos esto:

```
3A 0A           LD      A,10
C9              RET
```

Hagamos otra prueba:

```
A> LD FINAL.COM=B.REL, A.REL
```

Y de nuevo veamos el resultado de `FINAL.COM` en un editor hexadecimal:

```
C9              RET
3A 0A           LD      A,10
```

¿Os fijáis? Creo que queda bastante claro lo que ha pasado y lo que hace el linkador. Pero vamos a seguir profundizando.

Vamos a modificar ambos ficheros.

```
                CALL    MYFUNC
                RET
```

Listado 3.3: Fichero A.Z8A

```
MYFUNC:         LD      A,10
                RET
```

Listado 3.4: Fichero B.Z8A

Vamos a ensamblar primero `B.Z8A`:

```
A> AS B.Z8A
```

Y seguidamente `A.Z8A`:

```
A> AS A.Z8A
```

¡¡¡ Opppps !!! ¿Qué ha pasado? `AS.COM` nos ha escupido un mensaje de error:

```
Undefined symbol.
```

Y nos marca la linea donde se llama a `MYFUNC`. Pero... ¿por qué? Si tenemos esa rutina definida en `B.Z8A`...

Aunque claro, si lo pensamos... ¿cómo puede `AS.COM` saber eso? No se lo hemos indicado en ningún sitio.

3.1. Público o privado, externo o global.

Pues aquí hemos de introducir otro concepto nuevo para quien no ha trabajado con bibliotecas. Y, de nuevo, a ser en bajo nivel no representa mucha dificultad.

Empezemos por lo más sencillo, siguiendo con el ejemplo. Necesitamos que `A.Z8A` sepa que estamos llamando a una rutina ubicada en el otro módulo. Digamos que necesitamos llamar a una dirección *EXTERNA* denominada `MYFUNC`. Pues simple, modifiquemos `A.Z8A`, tal que así:

```
                EXTRN   MYFUNC
                CALL    MYFUNC
                RET
```

Listado 3.5: Fichero A.Z8A

Y ahora, si que si, ensamblamos:

```
A> AS A.Z8A
```

Ya está. Ahora si que hemos generado `A.REL` sin problemas, sin quejas. Pasemos, pues a linkar el programa final:

```
A> LD FINAL.COM=A.REL, B.REL
```

¡¡¡ No puede ser !!! ¡ Otra vez ! Esto es con lo que nos deleita `LD.COM`:

```
Undefined symbols:

MYFUNC
Code segment 0007 (start address 0100)
Free memory  6826
```

¿Por qué? Pues bastante sencillo de entender. `A.Z8A` se ha podido ensamblar sin problemas ya que le hemos indicado que `MYFUNC` era una dirección (rutina, función) externa. Pero, pongámonos en la piel del linkador: *Si, vale, es externa pero... ¿dónde está? ¿Qué valor tiene?.*

Ese es el problema. El linkador `desconoce` la ubicación de `MYFUNC`. Si. Podríamos decir que realmente *lo conoce* ya que se encuentra en `B.Z8A` por lo tanto en `B.REL`. Si, pero no. No le hemos indicado en `B.Z8A` que esa rutina va a poder ser utilizada (llamada) por otro módulo. No le hemos dicho, en definitiva, que es *PUBLICA* o *GLOBAL*.

Modifiquemos, por tanto, `B.Z8A`:

```
                GLOBAL  MYFUNC
MYFUNC:         LD      A,10
                RET
```

Listado 3.6: Fichero B.Z8A

Ahora generemos el fichero `B.REL`:

```
A> AS B.Z8A
```

Y, finalmente, volvamos a linkar:

```
A> LD FINAL.COM=A.REL, B.REL
```

¡Ya está! Ni un solo error ni advertencia (warning). Si desensamblamos el fichero resultante, `FINAL.COM` veremos algo así:

```
0100 CD 04 01    CALL    #0104
0103 C9          RET
0104 3E 0A       LD      A,#10
0106 C9          RET
```

Y aquí, viendo esto, se nos plantean bastantes dudas:

- ¿Por qué se ha ensamblado el programa en `#100`? ¿Y si quiero ensamblarlo en otra dirección? Yo no he puesto `ORG`.
- ¿Por qué hay que indicar como `GLOBAL` la rutina `MYFUNC`?

La primera cuestión es simple, aunque profundizaremos en ella más adelante, digamos que **por defecto** `AS` y `LD` están pensado para generar ejecutables para *MSX-DOS*, con lo que asumen por defecto la dirección #100. Y digo que adelante profundizaremos porque `LD` permite cualquier dirección, incluso nos genera, si se lo indicamos ficheros binarios para ejecutar en *BASIC*, formato *BLOAD*.

La segunda cuestión es donde quería llegar. Una de las grandes ventajas del sistema modular es que te permite tener etiquetas *PUBLICAS* y *PRIVADAS*. En `B.Z8A` hemos indicado al ensamblador que `MYFUNC` era pública y que podía ser llamada desde otro módulo. Pero hasta que lo hemos indicado esa etiqueta era *PRIVADA*. Esto significa que es posible tener `etiquetas locales a modulo` que pueden ser las mismas que las de otro módulo. Sigamos con el ejemplo:

```
                 EXTRN   MYFUNC
                 CALL    MYFUNC
                 RET
DATA:            DW      0DEADH
```

Listado 3.7: Fichero A.Z8A

```
                 GLOBAL  MYFUNC
MYFUNC:          LD      A,10
                 RET
DATA:            DW      0CAFEH
```

Listado 3.8: Fichero B.Z8A

Si ejecutamos los 3 pasos, ensamblar A y B, y linkar veremos que ni el ensamblador ni linker nos han dado ningún error. Y, como podemos observar tenemos **la misma etiqueta** en ambos fuentes (`DATA`). Si utilizásemos un sistema convencional, donde todo el fuente estuviese en el mismo fichero esto era **error garantizado**, ya que habría duplicidad en una etiqueta. Pero aquí no. Tenemos que `DATA` en `B.Z8A` es privada. Y también en `A.Z8A`. Y ni uno ni otro saben de la existencia de esta dirección. Ni el linker, ya que tampoco le hace falta.

¿Empezáis a notar el potencial que tenemos con los módulos? Sería sencillo intercambiar con otro programador un `.REL` indicándole las variables *GLOBALES* o *PUBLICAS* y el funcionaminento de

sus subrutinas, sin que el programador que utiliza el `.REL` tenga que preocuparse de como llamar a las mismas, ni el programador que ha creado la librería tenga que preocuparse porque le vean su código.

Pero el mayor potencial de los módulos, y que hemos podido demostrar en uno de los pasos del ejemplo, es que una vez ensamblados **no es necesario** volver a hacerlo nunca más, a no ser, claro que necesiten modificarse. Y esto es lo que, para nosotros, usuarios de un MSX, más nos interesa.

Los símbolos públicos y privados (globales y externos) que definimos `siempre` son un word, o lo que es lo mismo, direcciones de memoria. Nunca pueden ser un `EQU`, un `BYTE`...

Una cosa muy importante sobre ellos son las **operaciones** que podemos realizar con los mismos. Podemos hacer cosas así:

```
                LD      A,(MYFUNC + 2)
```

Así como utilizar cualquier otra operación. Ahora bien, lo que **no permite** `AS` (o igual no he sabido como) es operaciones **entre símbolos**:

```
                LD      HL,ENDVARS - INIVARS
```

Esto nos dará errores al linkar. Además el error que da `LD` es curioso: *Reserved Item*. Y para nuestra desgracia lo que hace es no interpretar la operación y se convierte en NUL, igual que lo que vamos a ver en el siguiente punto.

3.2. Cuidado con los Warning.

Ahora que tenemos fresco el ejemplo, es el mejor momento de hacer mención especial sobre los warning, o avisos que nos da ensamblador o linkador. Si nos fijamos en una de las partes del ejemplo del capítulo anterior, cuando el linkador no conocía la dirección de `MYFUNC`, el archivo `FINAL.COM`, igualmente, fue generado. Pero... ¿qué se generó? Pues aquí lo tenemos:

```
0100 CD 00 00   CALL    #0000
0103 C9         RET
0104 3E 0A      LD      A,#10
0106 C9         RET
```

¿Véis como está la instrucción `CALL`? Hace una llamada a `#0000`. Cualquier *SIMBOLO* que el linker desconoce y es utilizado se inicializa siempre a un valor nulo. Lo que es lo mismo `#0000`.

Así pues, **nada de warnings**. No son errores, pero mi recomendación es que nunca, jamás, os dejéis un warning en ningún programa. Por experiencia.

Después de esta pequeña introducción y para entender mejor otros conceptos sobre los módulos y el linker, pasaremos a la creación de un programa en formato ROM.

Capítulo 4

Creación de una ROM.

Ahora, si que si, después de esta pequeña introducción teórica, vamos al meollo del asunto, lo que nos ha traido hasta aquí: *La creación de una ROM*.

Sin querer profundizar, queremos recordar la estructura de una ROM. Normalmente (hay excepciones, si) se ejecutan en `#4000` u `#8000`, y utilizan como variables (RAM) la zona de `#C000` a `#F37F` o bien `#E000` a `#F37F`.

En el presente ejemplo, vamos a crear una ROM de 32K, compatible con cualquier MSX1 con 8K de RAM, que se ejecuta en `#4000` y utiliza la zona `#E000` para variables.

El objetivo que nos hemos marcado, ya que se trata de un ejemplo-demostración sencillo, es sencillamente cargar unos datos en la VRAM (patrones, sprites) y mostrarlos en pantalla, en Screen 2.

4.1. Linker: Conceptos Avanzados.

Seguramente, y en el punto que nos encontramos del artículo, es más que interesante hablar de algunos conceptos sobre el linkador, que tarde o temprano acabaremos utilizando.

4.1.1. Dirección de carga.

⚠ Pregunta: ¿Como ubico un programa en una dirección diferente a `#100`?

Es una de las primeras cosas que se nos vendrán a la cabeza a la hora de realizar un programa que no sea un ejecutable para *MSX-DOS*. Posicionar la dirección de carga (loading address) en otra posición.

Normalmente, cuando hemos trabajado con otros ensambladores, hemos utilizado la pseudoinstrucción `ORG` para tal fin. Y no es que aquí no podamos utilizarla si no que el linkador, en este caso hace un tratamiento especial para esa intrucción. En pocas palabras lo que hace es *rellenar* el espacio entre la dirección de carga (`#100` por defecto) y el ORG que pongamos. Así que si se nos ocurre hacer esto:

```
ORG     04000H
DB      041H, 042H
DW      INIT
DS      12
```

```
                ...
```

Y lo ensamblamos y linkamos con la dupla `AS-LD`, lo que vamos a obtener es un fichero que tendrá, por defecto un tamaño de `#4000 - #100` bytes, y a continuación nuestro código. Esos bytes extra, se rellenan con `NOPs`.

Pero claro, ¡eso no es lo que queremos nosotros! Queremos que, sencillamente el programa tenga la dirección de carga en `#4000` y ocupe exactamente los bytes que hemos programado.

El padre de `LD`, el *ensamblador M80* que es en el que está basado solucionaba el problema (y `LD` también permite hacerlo así) con las pseudoinstrucciones `PHASE-DEPHASE`. Que seguramente alguna vez hemos utilizado también. Esta sería la solución:

```
                PHASE   04000H
                DB      041H, 042H
                DW      INIT
                DS      12

                ...
                DEPHASE
```

Es decir, incluir todo nuestro programa entre estas instrucciones y eliminar la instrucción `ORG`. Pero esto no funciona con los modulos ya que, bien, podemos poner un módulo el `PHASE` pero.. ¿y al resto? ¿Cómo se en qué dirección va a quedar un módulo?

No, no es la solución. Pero tenemos suerte, `LD` permite hacer lo que queremos con un parámetro al linkar: **`/R#`**.

Volvamos al ejemplo:

```
                DB      041H, 042H
                DW      INIT
                DS      12
                ...
```

Hemos eliminado `PHASE` y `ORG`. Ensamblamos para generar el `.REL` y sencillamente al linkar especificamos la dirección de carga:

```
A> LD /R4000 PROG=PROG.REL,...
```

¡Ya está! `LD` nos generará el fichero del tamaño exacto a lo que hemos programado y con la dirección de carga en `#4000`. ¡Problema resuelto!

4.1.2. Dirección de carga. (Datos)

⚠ Pregunta Ahora ya sabemos como cambiar la dirección de carga pero... ¿y qué pasa con variables?

O lo que es lo mismo: Queremos especificar que uno de los módulos tiene una dirección de carga diferente,

para que sus etiquetas públicas apunten allí. Ej: Las variables de una ROM, normalmente ubicadas en `#C000` ó `#E000`.

Pues de nuevo el linker, junto al ensamblador nos da la solución. Parametro **`/D#`**, *locate data segment at address*. Pero claro, aquí aparece un concepto nuevo. Segmento de datos. Y no solo eso, es que tenemos dos conceptos más como éste: Segmento de código y Segmento Absoluto.

Ahora mismo no vamos a profundizar en estos conceptos pero, curiosamente, `CSEG` y `DSEG`, dos pseudo comandos del ensamblador fueron creados, en su día, precisamente para separar código que se cargaba en RAM (o ubicaba en ella) del código que estaba en la ROM.

Lo que nos interesa saber, aquí y para el ejemplo que llevamos entre manos, es como separar las variables del código, en cuanto a dirección de carga se refiere. Vamos con un miniejemplo:

```
                EXTRN   MYVAR

                DB      041H, 042H
                DW      INIT
                DS      12
INIT:           DI
                IM      1
                LD      SP,0F380H
                LD      A,(MYVAR)
                RET
```

Listado 4.1: Fichero CODE.Z8A

```
                DSEG
                GLOBAL  MYVAR
MYVAR:          DB      0
```

Listado 4.2: Fichero VARS.Z8A

⚠ ¡Atención! En `VARS.Z8A` hemos añadido el pseudocomando `DSEG`, para indicar al ensamblador (y al linkador más bien) que es un **segmento de datos**. Ensamblamos todo y linkamos:

```
A> AS CODE.Z8A
A> AS VARS.Z8A
A> LD /R4000 /DE000 ROM=CODE.REL,VARS.REL
```

Si analizamos, con un desensamblador lo que se nos ha generado veremos algo así:

```
4000: 41 42 10 40 00 00 00...
4010: 31 80 FE  LD      SP,#F380
4013: 3A 00 E0  LD      A,(#E000)
```

¡Perfecto! Justo lo que queríamos. Hemos creado la cabecera de la ROM en `#4000`, las variables apuntan a `#E000` y... **¡¡ tenemos un fichero de 40k y pico !!!**

¿Qué ha pasado? Pues como hemos comentado en la anterior cuestión el linkador se empeña siempre

en crear un programa completo, de inicio a fin. Y nos ha rellenado la zona de `#40xx` a `#E000`, que es donde empiezan las variables con `NOPS`. No. Eso tampoco lo queríamos. O si, pero a medias.

4.1.3. Exclusión de partes del código.

⚠ Pregunta: ¿Cómo evito que el linkador me incluya el segmento de datos de mi programa?

Si hubiésemos empezado por esta cuestión, sin habernos planteado las dos anteriores, seguramente andaríamos totalmente perdidos.

Pues si, `LD` permite no incluir en el programa el segmento de datos. Si señor. Parámetro **`/X`**.

Reutilicemos el ejemplo anterior pero cambiemos la orden del linkador:

```
LD /R4000 /DE000 /X ROM=CODE.REL,VARS.REL
```

Ahora mirad el fichero resultante. Cambia la cosa, ¿verdad? 29 bytes. Es eso lo que queríamos, ¿no? ¿No?

4.1.4. Tamaño del programa.

⚠ Pregunta: ¿Como consigo generar un fichero de un tamaño determinado?

Bueno. Esta duda nos asalta al ver el tamaño del programa generado en el punto anterior. Si queremos una ROM, 29 bytes no es algo muy ortodoxo... Buscamos 8K, 16K, 32K, 48K, 64K, 128K...

Otros ensambladores cuentan con pseudocomados que nos permiten ajustar el tamaño de nuestro programa, tales como:

1. **`ALIGN`**: Alinear.
2. **`SIZE`**: sjASM cuenta con la sentencia para generar el programa del tamaño que le indiquemos.

Luego están las que podemos hacer directamente con instrucciones clásicas de ensamblador, el clásico:

```
                DS 06000H - $
```

Que nos generaría una zona de bytes de relleno hasta 8k. Teniendo en cuenta que nuestra ROM se ejecuta en `#4000`, claro.

Pues bien. Las dos primeras **no existen** en `AS`. Y la tercera no se puede utilizar. ¿Cómo? Pues no. No se puede porque los módulos son reubicables. Así que sentencias de este tipo no están permitidas para los módulos. Es obvio, ¿no?

Bueno, no es tan obvio. Realmente se podría solucionar y tener en cuenta, pero `AS-LD` no lo permiten. Así que no le demos más vueltas. O, mejor dicho, yo no he conseguido hacerlo.

Esta vez, no, no hay solucion con parámetro de linker. Pero la tenemos en una tool externa: *ASIZE.COM*. Lo que hace esta utilidad no es otra cosa que alinear el tamaño al siguiente tamaño de ROM conocido. Ni

más ni menos. Si el programa ya tiene un tamaño conocido, pues no lo toca. Simple, ¿verdad? Veremos un ejemplo de uso en los siguientes capítulos.

4.2. Módulos, fuentes y estructura.

Antes de enumerar los módulos en los que hemos dividido nuestra ROM, vamos a repasar primero nuestras necesidades:

- Generamos una ROM de 32k, que nos muestra una pantalla en Screen 2.
- La pantalla muestra sprites de 16x16.
- Los datos a mostrar en pantalla, gráficos y sprites, están en la propia ROM, no se autogeneran. Se adjuntan en un fichero binario, sin compresión, denominado `VRAM.DAT`.
- Al ser una ROM de 32k, que se ejecuta en `#4000`, necesitaremos posicionar el slot de nuestra ROM en la pagina 2.
- Al utilizar sprites de 16x16, además de poner Screen 2, hemos de configurar el modo para ello.
- Para configurar el modo, vamos a tratar directamente con el hardware, con lo que necesitamos la dirección de los puertos del VDP y para su fácil tratamiento los meteremos en variables.

Así, a bote pronto, podemos apreciar los siguientes módulos:

- **DATOS**. Los datos que deseamos volcar a la pantalla. La información gráfica.
- **MAIN**, o principal. Donde tendremos la cabecera de la ROM y el código genérico.
- **SLOT**. El módulo que contendrá todas las rutinas asociadas al manejo de slots y subslots.
- **VARS**. Que contendrá las variables de la ROM.

⚠ Pregunta: ¿Y los DATOS de la pantalla?

NOTA SOBRE LA SINTAXIS: A partir de este momento, y hasta finalizar el artículo, la sintaxis utilizada para el código será **no compatible** con `AS`. Utilizaremos, como veremos más adelante, la herramienta *TOAS*, junto con *SMAKE* para la conversión.

4.2.1. Variables.

El primer módulo que vamos a explicar, quizás el más sencillo, van a ser las variables. Lo almacenaremos en un fichero con nombre `VARS.Z8A`.

```
                ;************************************************
                ; VARS                                          *
                ;************************************************

                DSEG            MANDATORY, DATA SEGMENT

--- GLOBAL SYMBOLS      ----------------------------------------


                PUBL    SLOTVAR
                PUBL    VPORT98W, VPORT99W, VPORT98R, VPORT99R


--- VARS                ----------------------------------------


- SLOT

SLOTVAR         DEFB    0

- VDP

VPORT98W        DEFB    0
VPORT99W        DEFB    0
VPORT98R        DEFB    0
VPORT99R        DEFB    0
```

Listado 4.3: Fichero VARS.Z8A

Como hemos explicado en el punto 5.1.2, las variables las asignamos al segmento de datos, por ello la pseudoinstruccion `DSEG`.

Seguidamente aparece la sentencia `PUBL`, se convertirá a `GLOBAL` para `AS`. Todas las variables que utilizamos son públicas.

Y finalmente las variables. Hemos añadido `SLOTVAR`, que contendrá el slot-subslot donde se ubica la ROM al arrancar, para completar. Seguidamente vemos las variables donde almacenaremos los puertos del VDP, tanto de escritura como de lectura.

4.2.2. Slot.

El módulo SLOT contendrá, como hemos comentado en el primer punto de la sección, las rutinas correspondientes al manejo de slots-subslots. Lo almacenaremos en un fichero con nombre `SLOT.Z8A`.

```
                ;************************************************
                ; SLOT ROUTINES                                 *
                ;************************************************
```

```
--- INCLUDES            ----------------------------------------

                INCL    HARD.MAC                SYSTEM VARS, BIOS...

--- PUBLIC SYMBOLS      ----------------------------------------

                PUBL    SEARSLOT, SETSLOT

--- ROUTINES            ----------------------------------------

- SEARSLOT ------------
- SEARCH ROM SLOT
- ROM IN PAGE 1 (#4000 - #7FFF)

SEARSLOT        CALL    RSLREG
                RRCA
                RRCA
                AND     3
                LD      C,A
                LD      B,0
                LD      HL,EXPTBL
                ADD     HL,BC
                LD      A,(HL)
                AND     #80
                JP      Z,$NOEXP
                OR      C
                LD      C,A
                INC     HL
                INC     HL
                INC     HL
                INC     HL
                LD      A,(HL)
                AND     %00001100
$NOEXP          OR      C
                LD      H,#80
                RET

- SSETSLOT ------------
- SEARCH AND SET SLOT
- OUT   A SLOT

SETSLOT         CALL    SEARSLOT
                PUSH    AF
```

```
                CALL    ENASLT
                POP     AF
                RET
```

Listado 4.4: Fichero SLOT.Z8A

Nada que comentar en este módulo. No tiene nada especial. Y las rutinas son conocidas, sin duda, por todos.

¿Nada de especial? ¿Qué es la sentencia `INCL`? Pues otra palabra reservada de `TOAS` que se convierte en `INCLUDE` para `AS.COM`. Sobre los `INCLUDES` hablaremos más adelante.

4.2.3. Main.

El módulo principal de la aplicación, el código que realizará la tarea. Lo almacenaremos con nombre `ROM.Z8A`.

```
;*****************************************************************
; ROM                                                            *
; EXAMPLE ROM                                                    *
; DEMOSTRATION:                                                  *
;       AS.COM                                                   *
;       LD.COM                                                   *
;       D2R.COM                                                  *
;       TOAS.COM                                                 *
;       SMAKE.COM                                                *
;       ASIZE.COM                                                *
; (C) 2014 ARMANDO PEREZ ABAD                                    *
;*****************************************************************


--- INCLUDES            ----------------------------------------


                INCL    HARD.MAC


--- EXTERNAL SYMBOLS    ----------------------------------------


- DATA

                EXTR    VRAM

- SLOT

                EXTR    SEARSLOT, SETSLOT

- VARS
```

```
                EXTR    SLOTVAR
                EXTR    VPORT98W, VPORT99W, VPORT98R, VPORT99R


--- ROM HEADER          ----------------------------------------


                DEFB    #41, #42
                DEFW    INIT
                DEFS    12


--- PROGRAM INIT        ----------------------------------------


- INIT ----------------
- INIT ROM

INIT            DI
                IM      1
                LD      SP,#F380                MANDATORY
                CALL    SETSLOT                 SET SLOT
                LD      (SLOTVAR),A             SAVE SLOT

-               INIT VDP PORT

                LD      A,(#07)                 WRITE
                LD      (VPORT98W),A
                INC     A
                LD      (VPORT98W),A

                LD      A,(#06)                 READ
                LD      (VPORT98R),A
                INC     A
                LD      (VPORT99R),A


-               SCREEN 2 COLOR 0,0,0

                LD      HL,0
                XOR     A
                LD      (FORCLR),A
                LD      (BAKCLR),HL
                LD      A,2
                CALL    CHGMOD

-               SET VDP SPRITES 16X16 NOMAG
```

```
                LD      A,(VPORT98W)
                LD      C,A
                LD      A,(RG0SAV + 1)
                AND     %11111100
                OR      %10
                DI
                OUT     (C),A
                LD      (RG0SAV + 1),A
                LD      A,#80 + 1               VDP REG#1
                EI
                OUT     (C),A

-               VRAM TRANSFER

                LD      HL,VRAM
                LD      DE,#0000
                LD      BC,#4000
                CALL    LDIRVM

-               WAIT

                DI
                JP      $
```

Listado 4.5: Fichero ROM.Z8A

Este módulo tampoco tiene mucho de especial, excepto el uso del pseudocomando `EXTR`, que `TOAS` convertirá como `EXTRN`, y del que hablamos en el punto 3.1 para la definición de un símbolo externo para el linker.

4.2.4. Includes.

Podríamos pasar ya directamente a generar la ROM, pero si cogiésemos los archivos módulo que hemos expuesto y los intentásemos ensamblar y linkar tendríamos bastantes errores. Nos faltan, cosas. *Constantes*. Si nos fijamos en los fuentes presentados vemos símbolos que no hemos definido en ningún sitio:

- RSLREG
- EXPTBL
- ENASLT
- CHGMOD
- RG0SAV
- ...

Todos estas constantes las encontramos en el fichero `HARD.Z8A`. Que es, precisamente, el que **incluimos** en los módulos. No dejan de ser constantes definidas en el formato de todos los ensambladores conocidos, `EQU`. Y si, aquí `AS` funciona como cualquier otro ensamblador. Al encontrar un `INCLUDE` abre el fichero en cuestión y inserta en la posición donde se encuentra el include, antes de realizar el ensamblado. Aquí lo tenemos.

```
                ************************************************
                * HARDWARE CONSTANTS                             *
                ************************************************

--- GENERIC             ----------------------------------------


WORD            EQU     2
BYTE            EQU     1


--- BIOS                ----------------------------------------


ENASCR          EQU     #44             ENABLE SCREEN
DISSCR          EQU     #41             DISABLE SCREEN
WRTVDP          EQU     #47             WRITE VDP C->REG B->DATA
WRTVRM          EQU     #4D             HL ADDRESS A DATA
FILVRM          EQU     #56             FILL VRAM
LDIRVM          EQU     #5C             TRANSFER TO VRAM FROM MEMORY
LDIRMV          EQU     #59             TRANSFER TO MEMORY FROM VRAM
CHGMOD          EQU     #5F             A -> SCREEN MODE
WRTPSG          EQU     #93             A -> REG E -> DATA
RDPSG           EQU     #96             A -> REG
GTSTCK          EQU     #D5             A -> JOYPORT (1/2)
GTTRIG          EQU     #D8             A -> JOYTRIG 1,3 - 2,4
SNSMAT          EQU     #141            A -> ROW
ENASLT          EQU     #24             ENABLES A SLOT (FxxxSSPP)
RSLREG          EQU     #138            IN A,(#A8)
RDSLT           EQU     #0C
WRSLT           EQU     #14
CHPUT           EQU     #A2
CALSLT          EQU     #1C
DCOMPR          EQU     #20


--- SYSTEM VARS         ----------------------------------------


HTIMI           EQU     #FD9F           TIMER INTERRUPT HANDLER
HKEYI           EQU     #FD9A           I/O TIMER INTERRUPT HANDLER
CLIKSW          EQU     #F3DB           0 OFF KEY CLICK
FORCLR          EQU     #F3E9           FOREGROUND COLOR
```

```
BAKCLR          EQU     #F3EA           BACKGROUND COLOR
BDRCLR          EQU     #F3EB           BORDER COLOR
SCRMOD          EQU     #FCAF           CURRENT SCREEN MODE
RG0SAV          EQU     #F3DF           VDP #0
RG1SAV          EQU     RG0SAV + BYTE   VDP #1
RG7SAV          EQU     RG0SAV + 7      VDP #7
JIFFY           EQU     #FC9E           JIFFY
EXPTBL          EQU     #FCC1           SLOT INFO TABLE
LINL40          EQU     #F3AE
EXTVDP          EQU     #FFE7           EXT VDP REGS
STATFL          EQU     #F3E7           VDP REG STATUS 0


--- BDOS POINTER        ----------------------------------------


BDOSCALLE       EQU     #05


--- DOS CONSTANTS       ----------------------------------------


_PARAMS         EQU     #80
_FCB1           EQU     #5C
_FCB2           EQU     #6C


--- DOS 1 FUNCTIONS     ----------------------------------------


_CONOUT         EQU     #02
_STROUT         EQU     #09
_FOPEN          EQU     #0F
_FCLOSE         EQU     #10
_FMAKE          EQU     #16
_SETDTA         EQU     #1A
_WRBLK          EQU     #26
_RDBLK          EQU     #27
_TERM0          EQU     #00


--- DOS 2 FUNCTIONS     ----------------------------------------


_OPEN           EQU     #43
_CREATE         EQU     #44
_CLOSE          EQU     #45
_READ           EQU     #48
_WRITE          EQU     #49
```

```
_SEEK           EQU     #4A
_DEFAB          EQU     #63
_DEFER          EQU     #64
_DOSVER         EQU     #6F
_FFIRST         EQU     #40


--- DOS FCB OFFSETS     ----------------------------------------


FDRIVE          EQU     0
FFILNAME        EQU     1
FFILEXT         EQU     9
FCURBLK         EQU     12
FRECSIZ         EQU     14
FFILSIZ         EQU     16
FDATE           EQU     20
FTIME           EQU     22
FDEVID          EQU     24
FDIRLOC         EQU     25
FTOPCLUS        EQU     26
FLSTCLUS        EQU     28
FRELLOC         EQU     30
FCURREC         EQU     32
FRNDREC         EQU     33
SIZEFCB         EQU     37


--- PSG                 ----------------------------------------


PSGSEL          EQU     #A0
PSGWRT          EQU     #A1
PSGREAD         EQU     #A2


--- SR2 BASE ADDRESS    ----------------------------------------


SCRPAT          EQU     #0000           PATTERN GENERATOR TABLE
SCRCOL          EQU     #2000           PATTERN COLOR TABLE
SCRATT          EQU     #1800           PATTERN NAME TABLE
SPRPAT          EQU     #7800           SPRITE GENERATOR TABLE
SPRATT          EQU     #7600           SPRITE NAME TABLE
SPRCOL          EQU     #7400           SPRITE COLOR TABLE
SPRATTS         EQU     #0080           SPRITENAM SIZE


--- VDP COMMANDS        ----------------------------------------
```

```
- COMMANDS

VDPCHMMM        EQU     #D0             HIGH SPEED MOVE VRAM TO VRAM
VDPCLMMM        EQU     #98             LOGICAL SPEED VRAM VRAM TIMP

- VDP COMMAND OFFSETS

VDPOFFSX        EQU     0
VDPOFFSY        EQU     2
VDPOFFDX        EQU     4
VDPOFFDY        EQU     6
VDPOFFNX        EQU     8
VDPOFFNY        EQU     10
VDPOFFNU        EQU     12
VDPOFFDI        EQU     13
VDPOFFCM        EQU     14
```

Listado 4.6: Fichero HARD.Z8A

Si, hay más constantes de las que realmente hemos utilizado en el ejemplo. Normalmente voy añadiendo al fichero las cosas que voy utilizando, de ahí que tenga más constantes de las utilizadas.

4.3. Generando la ROM.

Ahora que ya tenemos todos los módulos, tenemos los datos binarios y constantes solo nos falta generar la ROM.

A grandes rasgos estos son los pasos que hemos de seguir para generar la ROM:

⇒ `D2R VRAM.DAT VRAM` Generamos el `.REL` con los datos binarios, con nombre VRAM de símbolo público.

⇒ `TOAS HARD.Z8A HARD.MAC` Exportamos `HARD.Z8A`, las constantes, a formato compatible con `AS`.

⇒ `TOAS VARS.Z8A VARS.MAC` Exportamos `VARS.Z8A` a formato compatible con `AS`.

⇒ `AS VARS` Generamos el fichero `.REL` de las variables, segmento de datos.

⇒ `TOAS SLOT.Z8A SLOT.MAC` Exportamos `SLOT.Z8A` a formato compatible con `AS`.

⇒ `AS SLOT` Generamos objeto del módulo slot.

⇒ `TOAS ROM.Z8A ROM.MAC` Como con el resto, generamos el fichero compatible con `AS`.

⇒ `AS ROM` Ensamblamos el módulo `MAIN`.

⇒ `LD /R4000 /DE000 /X /A ROM.ROM=ROM.REL,SLOT.REL,VRAM.REL,VARS.REL` Linkamos.

⇒ `ASIZE ROM.ROM` Alineamos la ROM a 32k.

Como vemos, son muchos pasos que, por suerte, no hemos de repetir todas las veces. Pero... ¿qué pasa si modifico `VARS.Z8A` para añadir una variable? ¿Y si añado una rutina nueva a `SLOT.Z8A`?

En ambos casos debería, obligatoriamente, exportar el fuente y luego ensamblarlo, para terminar linkándolo. Pero.. ¿y si modifico todos o algunos y no se qué he tocado? Pues que, en este caso, debería de hacer todos los pasos. Y esto, precisamente, *no* es la ganancia de la que estábamos presumiendo con el trabajo con módulos, si no, más bien, todo lo contrario.

Para el colmo, este último caso que hemos expuesto, es el que más se da cuando estás trabajando en un proyecto. No, no es de recibo. Y, claro, como podemos intuir la solución no es meter todos estos pasos en un `.BAT`.

Capítulo 5

Smake.

Aquí es donde entra en juego la tool `SMAKE.COM`, que va en el paquete. `SMAKE`, no es otra cosa que un *BAT avanzado*. Porque llamarlo *MAKE* como el comando de **NIX* no me parece apropiado, aunque su nombre sea ese *STUPID MAKE*.

Lo único que hace `SMAKE` es, mediante una sencilla regla donde especificamos un *target* y unas *dependencias*, ejecutar *en caso de que alguna de las dependencias tenga una fecha superior al target* los comandos que seguidamente especificamos.

Realmente un make de **NIX* hace eso. Pero claro... con muchas más posibilidades, patrones, variables, no de forma secuencial... Y en esto hay que hacer hincapié: `SMAKE` funciona de manera secuencial. Si, por ejemplo, cuando llega a una regla, ve que ese target tiene una dependencia *no vuelve* a recorrer todas las reglas mirando si esa regla ha sido ejecutada o no.

Pero no nos liemos y tampoco os preocupeis. Para la generación de nuestros proyectos nos sobra con la tarea que hace `SMAKE`. Solo hemos de seguir una serie de normas.

`SMAKE` procesa un archivo *Makefile* que le pasaremos por parámetro. Una vez procesado generará, de forma automática, un `.BAT` llamado `SMAKEGEN.BAT` que contiene los comandos que se requieren en esa compilación.

Como podemos intuir, para ejecutar `SMAKE` hacen falta dos comandos:

```
A> SMAKE MAKEFILE
A> SMAKEGEN
```

Así pues, lo más rápido y cómodo es generarse un `.BAT` llamado `MAKE`.

Pero vayamos por pasos, pongamos aquí el `MAKEFILE` que vamos a utilizar:

```
VRAM.REL: VRAM.DAT
        ECHO VRAM...
        D2R VRAM.DAT VRAM
HARD.MAC: HARD.Z8A
        ECHO TOAS HARD.Z8A HARD.MAC
        TOAS HARD.Z8A HARD.MAC
VARS.REL: VARS.Z8A
        ECHO ASSEMBLING VARS...
        TOAS VARS.Z8A VARS.MAC
```

```
        AS VARS
SLOT.REL: SLOT.Z8A HARD.Z8A
        ECHO ASSEMBLING SLOT...
        TOAS SLOT.Z8A SLOT.MAC
        AS SLOT
ROM.REL: ROM.Z8A HARD.Z8A VARS.Z8A
        ECHO ASSEMBLING ROM...
        TOAS ROM.Z8A ROM.MAC
        AS ROM
ROM.ROM:
        ECHO LINKING...
        LD /R4000 /DE000 /X /A ROM.ROM=ROM.REL,SLOT.REL,VRAM.REL,VARS.REL
        ECHO ALIGN SIZE...
        ASIZE ROM.ROM
```

Vamos a explicar cada una de las reglas.

Los comandos de cada regla (objetivo, target) deben de estar obligatoriamente separados por `TAB`. En caso contrario `SMAKE` no funcionará correctamente.

5.1. VRAM.

```
VRAM.REL: VRAM.DAT
        ECHO VRAM...
        D2R VRAM.DAT VRAM
```

Lo primero que generamos es el módulo correspondiente al archivo binario de los datos gráficos. `SMAKE` comparará la fecha del fichero `VRAM.REL` con `VRAM.DAT`. En caso de que `VRAM.DAT` tenga una fecha mayor *o bien* *`VRAM.REL` no exista* añadirá a `SMAKEGEN.BAT` los comandos de abajo de la regla.

5.2. HARD.

```
HARD.MAC: HARD.Z8A
        ECHO TOAS HARD.Z8A HARD.MAC
        TOAS HARD.Z8A HARD.MAC
```

Dado que dos de los módulos (y seguramente todos los que hagamos) utilizan el `INCLUDE` de `HARD.Z8A` es conveniente que sea lo primero que se exporte a formato AS, con TOAS. El funcionamiento, no voy a repetirlo, idéntico a la primera regla.

5.3. VARS.

```
VARS.REL: VARS.Z8A
        ECHO ASSEMBLING VARS...
        TOAS VARS.Z8A VARS.MAC
        AS VARS
```

Modulo de variables. Las variables se utilizan, también, en el módulo principal, así que es conveniente revisarlas antes.

5.4. SLOT.

```
SLOT.REL: SLOT.Z8A HARD.Z8A
        ECHO ASSEMBLING SLOT...
        TOAS SLOT.Z8A SLOT.MAC
        AS SLOT
```

Aquí ya vemos algo nuevo. Además de propio `SLOT.Z8A`, el módulo `SLOT.REL` depende de `HARD.Z8A`. ¿Y por qué no de `HARD.MAC` que es lo que se genera y se compila? Pues porque, como habíamos dicho, `SMAKE` trabaja sobre un `.BAT` y de manera secuencial.

Si, `HARD.MAC` no ha cambiado y si ha cambiado `HARD.Z8A`, aunque se hubiese metido en SMAKEGEN.BAT para generarse *no se daría cuenta*. Sin embargo, poniendo `HARD.Z8A` que es el fichero principal, todo queda solucionado.

5.5. MAIN.

```
ROM.REL: ROM.Z8A HARD.Z8A VARS.Z8A
        ECHO ASSEMBLING ROM...
        TOAS ROM.Z8A ROM.MAC
        AS ROM
```

Una regla más compleja todavía. Tiene 3 dependencias. Además de la suya propia, `HARD` (igual que hemos explicado en la anterior) y `VARS`. De nuevo utilizamos `VARS.Z8A` por lo mismo que hemos explicado en la regla anterior con `HARD.Z8A`.

5.6. ROM.

```
ROM.ROM:
        ECHO LINKING...
        LD /R4000 /DE000 /X /A ROM.ROM=ROM.REL,SLOT.REL,VRAM.REL,VARS.REL
        ECHO ALIGN SIZE...
        ASIZE ROM.ROM
```

Finalmente la regla-objetivo que genera la ROM. Como podemos ver, esta regla *no tiene dependencias*. Como hemos dicho anteriormente, si `SMAKE` encuentra una regla sin dependencias ejecuta obligatoriamente sus comandos.

Capítulo 6

Paquete preparado.

La mejor manera de entender esto es, con el paquete que hemos preparado debidamente descomprimido en un disquete o en tu directorio favorito en tu MSX.

El paquete lo podeis descargar desde la dirección que os hemos puesto en el punto 1.

Ejecutaremos:

```
A> MAKE.BAT (o MAKE si estáis en DOS1)
```

Como vemos ejecuta todos los pasos.

Volvamos a ejecutar:

```
A> MAKE.BAT (o MAKE si estáis en DOS1)
```

Ejecuta, sencillamente el linkado de la ROM. Nada del resto ha cambiado, nada tiene que hacer. Hemos puesto la regla de linkado así para comprender las posibilidades de `SMAKE`. Lo suyo sería, claro está, que `ROM.ROM` tuviese de dependencias todos los ficheros fuente `Z8A` y `VRAM.DAT`. En ese caso no hubiese ejecutado nada.

Ahora abrid, por ejemplo, el fichero `SLOT.Z8A`, meted un intro, un espacio o siplemente grabar el fichero (para que se actualice su fecha). Ejecutad, de nuevo:

```
A> MAKE.BAT (o MAKE si estáis en DOS1)
```

¿Veis lo que ha pasado? Ahora se habrá generado `SLOT.REL` y luego se habrá linkado.

Entendemos que, con este sencillo ejemplo, quedan patentes las ventajas de `SMAKE` y la necesidad del mismo para trabajar con proyectos del estilo. Si, existe un *MAKE* mejor en MSX, pero es algo más complejo para iniciados y no cumplía todos los objetivos que nos habíamos propuesto.

También hay que dejar claro que `SMAKE` funciona bien si las fechas están bien. Vamos que tener el reloj de vuestro MSX en hora es necesario. Si, casi ningún MSX1 lleva reloj interno. Pues bien, acordaros de poner la fecha cada vez que trabajeis.

Capítulo 7

Conclusiones.

No vamos a engañar a nadie: Trabajar con el MSX, aunque sea un Turbo R (recomendado) es lento. Nunca se podrá igualar la velocidad de ensamblado y proceso de cualquier máquina de hoy en día de las que tenemos, sin duda, en nuestros hogares.

Si recomendamos este método de trabajo es, principalmente, por tres razones:

1. ¡Le damos uso al MSX! Eso que seguramente no hacíamos hace años.
2. Si aprendemos a desenvolvernos con el sistema modular, saldremos, a la larga, ganando. Como decíamos todo el código queda más claro. Y se aprende a estructurar los programas.
3. Trabajar en el MSX tiene una ventaja fundamental: No te despistas. Si bien perdemos más tiempo (aunque eso es discutible) por la lentitud del proceso, lo vamos a recuperar y con creces al no tener distracciones. Cualquier PC/Mac de hoy en día está conectado a internet... mensajes, notificaciones, ... todo eso al final nos hace perder un tiempo precioso y evita concentrarnos en nuestro proyecto.

Y eso es todo. En el siguiente número, con todo esto aprendido, explicaremos como hacer, con este sistema, ROMs de 48k y... ¿MegaRoms?