# Ensamblador para ZX Spectrum

# Tres en raya

Ensamblador para ZX Spectrum Tres en raya por Juan Antonio Rubio García

Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional License.

# Contenido

# Introducción

Tres en raya nació de la lectura del libro “Club de programación de juegos de ZX Spectrum”, de Gary Plowman. Al terminar de teclear el listado Basic que viene en el libro, nos propone como ejercicio añadir el modo de un jugador.

Una vez que llegué a la conclusión de que el Basic no es lo mío, decidí desarrollar el juego desde cero en ensamblador, con las opciones siguientes:

- Modo de uno y dos jugadores.
- Posibilidad de especificar los nombres de los jugadores.
- Puntos a conseguir para terminar la partida.
- Tiempo máximo por movimiento.

En Batalla espacial vimos el uso de la rutina de la ROM para el manejo de teclado, pero es en Tres en raya donde se utiliza de manera más intensa.

También vamos a hacer uso de las interrupciones, para controlar el tiempo máximo por movimiento e informar al usuario, con una cuenta atrás, de los segundos que le quedan para mover; perderá el turno si los consume sin haber realizado el movimiento.

Llegados a este punto del libro, ya se debería tener cierto nivel, por lo que no se explicarán las instrucciones una por una; se explicará lo que hace la rutina en su conjunto, que junto con los comentarios debería ser suficiente para su comprensión.

# Paso 1: tablero de juego

Lo primero que vamos a hacer es diseñar y pintar el tablero de juego, que se compone de una cuadrícula con nueve celdas.

Creamos la carpeta TresEnRaya y los archivos:

- Main.asm
- Rom.asm
- Screen.asm
- Sprite.asm
- Var.asm

Esta vez no voy a ir creando carpetas por cada paso, aunque si lo preferís podéis seguir haciéndolo así.

## Sprites

Definimos los gráficos en Sprite.asm: la X del jugador uno, la O del jugador dos, la cruceta del tablero y las líneas vertical y horizontal. Al igual que en Batalla espacial, vamos usar los UDG.

El aspecto de los gráficos en Sprite.asm es el siguiente:

```
; ------------------------------------------------------------------
; Fichero: Sprite.asm
;
; Definición de gráficos.
; ------------------------------------------------------------------
; Sprite jugador 1
Sprite_P1:
db $c0, $e0, $70, $38, $1c, $0e, $07, $03    ; $90
db $03, $07, $0e, $1c, $38, $70, $e0, $c0    ; $91

; Sprite jugador 2
Sprite_P2:
db $03, $0f, $1c, $30, $60, $60, $c0, $c0    ; $92 Arriba/Izquierda
db $c0, $f0, $38, $0c, $06, $06, $03, $03    ; $93 Arriba/Derecha
db $c0, $c0, $60, $60, $30, $1c, $0f, $03    ; $94 Abajo/Izquierda
db $03, $03, $06, $06, $0c, $38, $f0, $c0    ; $95 Abajo/Derecha

; Sprite cruceta
Sprite_CROSS:
db $18, $18, $18, $ff, $ff, $18, $18, $18    ; $96

; Sprite línea horizontal
Sprite_SLASH:
db $18, $18, $18, $18, $18, $18, $18, $18    ; $97

; Sprite línea vertical
Sprite_MINUS:
db $00, $00, $00, $ff, $ff, $00, $00, $00    ; $98
```

Con las prácticas que realizasteis en Batalla espacial, deberíais ser capaces de dibujar los sprites en un papel para ver cómo quedan.

El siguiente paso es pintar el tablero. Son necesarias constantes para referenciar la ROM, definir posiciones, datos a pintar y rutinas para pintarlos.

## Rutinas y variables de la ROM

Las constantes de la ROM las definimos en ROM.asm.

```
; ------------------------------------------------------------------
; Fichero: Rom.asm
;
; Rutinas de la ROM y variables de sistema.
; ------------------------------------------------------------------

; Atributos permanentes de la pantalla 2, principal.
ATTR_S:        equ $5c8d

; Atributos actuales de la pantalla 2.
ATTR_T:        equ $5c8f

; Atributo del borde y pantalla 1. Usada por BEEPER.
BORDCR:        equ $5c48

; Dirección de las coordenadas de inicio X e Y (PLOT)
COORDX:        equ $5c7d
COORDY:        equ $5c7e

;-------------------------------------------------------------------
; Dirección donde están los flags de estado del teclado cuando
; no están activas las interrupciones.
;
; Bit 3 = 1 entrada en modo L, 0 entrada en modo K.
; Bit 5 = 1 se ha pulsado una tecla, 0 no se ha pulsado.
; Bit 6 = 1 carácter numérico, 0 alfanumérico.
;-------------------------------------------------------------------
FLAGS_KEY:     equ $5c3b

;-------------------------------------------------------------------
; Dirección dónde está la última tecla pulsada
; cuando no están activas las interrupciones.
;-------------------------------------------------------------------
LAST_KEY:      equ $5c08

; Dirección de los gráficos definidos por el usuario.
UDG:           equ $5c7b

; Dirección de inicio del área de gráficos de la VideoRAM.
VIDEORAM:      equ $4000

; Longitud del área de gráficos de la VideoRAM.
VIDEORAM_L:    equ $1800

; Dirección de inicio del área de atributos de la VideoRAM.
VIDEOATT:      equ $5800

; Longitud del área de atributos de la VideoRAM.
VIDEOATT_L:    equ $0300

;-------------------------------------------------------------------
; Rutina beeper de la ROM.
;
; Entrada:      HL -> Nota.
;               DE -> Duración.
;
; Altera el valor de los registros AF, BC, DE, HL e IX.
;-------------------------------------------------------------------
BEEPER:        equ $03b5

;-------------------------------------------------------------------
; Dibuja una línea desde las coordenadas COORDS.
;
; Entrada: B -> Desplazamiento vertical de la línea.
;          C -> Desplazamiento horizontal de la línea.
;          D -> Orientación vertical de la línea:
;               $01 = Arriba, $FF = Abajo.
;          E -> Orientación horizontal de la línea:
;               $01 = Izquierda, $FF = Derecha.
; Altera el valor de los registros AF, BC y HL.
;-------------------------------------------------------------------
```

```
DRAW:          equ $24ba

;--------------------------------------------------------------------
; Rutina AT de la ROM. Posiciona el cursor.
;
; Entrada:      B -> Coordenada Y.
;               C -> Coordenada X.
;
; La esquina superior izquierda de la pantalla es 24, 33.
; (0-21) (0-31)
;--------------------------------------------------------------------
LOCATE:        equ $0dd9

;--------------------------------------------------------------------
; Rutina de la ROM que abre el canal de salida.
;
; Entrada:      A -> Canal (1 = pantalla inferior, 2 = superior).
;--------------------------------------------------------------------
OPENCHAN:      equ $1601
```

Como veréis, hemos añadido variables y rutinas no vistas hasta ahora. No os preocupéis, las iremos viendo según avanzamos.

## Variables

En Var.asm añadimos los datos necesarios para pintar el tablero.

```
; --------------------------------------------------------------------
; Fichero: Var.asm
;
; Declaraciones de variables y constantes.
; --------------------------------------------------------------------

; Tinta del tablero
INKBOARD:   equ $04

; Posición Y = 0 para LOCATE.
INI_TOP:    equ $18
; Posición X = 0 para LOCATE.
INI_LEFT:   equ $21
; Posiciones para pintar los elementos de OXO.
POS1_TOP:   equ INI_TOP - $07
POS2_TOP:   equ POS1_TOP - $05
POS3_TOP:   equ POS2_TOP - $05

POS1_LEFT:  equ INI_LEFT - $0a
POS2_LEFT:  equ POS1_LEFT - $05
POS3_LEFT:  equ POS2_LEFT - $05
; --------------------------------------------------------------------
; Gráficos.
; --------------------------------------------------------------------
; Líneas verticales del tablero.
Board_1:
db $20, $20, $20, $20, $97, $20, $20, $20, $20, $97, $20, $20, $20
db $20, $ff

; Líneas horizontales del tablero.
Board_2:
db $98, $98, $98, $98, $96, $98, $98, $98, $98, $96, $98, $98, $98
db $98, $ff

; --------------------------------------------------------------------
; Partes de la pantalla.
; --------------------------------------------------------------------
; Números de guía para que el jugador sepa que tecla pulsar
; para poner las fichas (O X O)
Board_Helper:
; Tinta magenta
db $10, $03
```

```
; AT,Y,X,número
db $16, $08, $0a, "1", $16, $08, $10, "2", $16, $08, $15, "3"
db $16, $0d, $0a, "4", $16, $0d, $10, "5", $16, $0d, $15, "6"
db $16, $12, $0a, "7", $16, $12, $10, "8", $16, $12, $15, "9"
db $ff
```

## Pantalla

En Screen.asm vamos a implementar las rutinas necesarias para pintar los elementos en la pantalla.

```
; ------------------------------------------------------------------
; Posiciona el cursor. La esquina superior está en 24,33.
;
; Entrada:     B -> Y.
;              C -> X.
; ------------------------------------------------------------------
AT:
push    af
push    bc
push    de
push    hl      ; Preserva registros

call    LOCATE  ; Posiciona cursor.

pop     hl
pop     de
pop     bc
pop     af      ; Recupera registros.

ret
```

La rutina AT recibe en los registro B y C las coordenadas Y y X simultáneamente (son coordenadas invertidas). Llama a la rutina de la ROM que posiciona el cursor.

Vamos a administrar los distintos atributos de color de manera diferenciada, con una rutina para cada atributo.

```
; ------------------------------------------------------------------
; Cambia el color de borde.
;
; Entrada: A -> Color para el borde.
;
; Altera el valor de los registros AF.
; ------------------------------------------------------------------
BORDER:
push    bc              ; Preserva BC
and     $07             ; A = color
out     ($fe), a        ; Cambia color borde

rlca
rlca
rlca                    ; Rota tres bits a izquierda para poner
                        ; color en bits de paper/border
ld      b, a            ; B = A
ld      a, (BORDCR)     ; A = variable sistema BORDCR.
and     $c7             ; Quita los bits de paper/border
or      b               ; Agrega el color de paper/border

ld      (BORDCR), a     ; BORDCR = A, para que BEEPER no cambie
pop     bc              ; Recupera BC

ret
```

La rutina BORDER recibe en A el color a asignar al borde. Para asegurar que es un color correcto, nos quedamos con los bits 0, 1 y 2. Cambiamos el color de borde y luego ponemos el valor en la variable de sistema BORDCR para que BEEPER no lo cambie.

```
; ------------------------------------------------------------------
; Asigna el color de la tinta.
;
; Entrada: A -> Color de la tinta.
;               FBPPPIII
;
; Altera el valor de los registros AF.
; ------------------------------------------------------------------
INK:
push    bc              ; Preserva BC
and     $07             ; A = INK
ld      b, a            ; B = A

ld      a, (ATTR_T)     ; A = atributo actual
and     $f8             ; A = FLASH, BRIGHT y PAPER
or      b               ; Añade INK

ld      (ATTR_T), a     ; Actualiza atributo actual
ld      (ATTR_S), a     ; Actualiza atributo permanente
pop     bc              ; Recupera BC

ret
```

La rutina INK recibe en A el color de la tinta. Nos aseguramos de quedarnos solo con el color de la tinta, obtenemos los atributos actuales, dejamos el parpadeo, brillo y fondo, y le añadimos la tinta. Actualizamos las variables de sistema, que son consultadas por RST $10 y al cambiar el canal, para aplicar los atributos de color.

```
; ------------------------------------------------------------------
; Asigna el color de fondo.
;
; Entrada: A -> Color de fondo.
;
; Altera el valor de los registros AF.
; ------------------------------------------------------------------
PAPER:
push    bc              ; Preserva BC
and     $07             ; A = color
rlca
rlca
rlca                    ; Rota tres bits a izquierda para poner
                        ; color en bits de paper/border
ld      b, a            ; B = A
ld      a, (ATTR_T)     ; A = atributo actual
and     $c7             ; Quita fondo

or      b               ; Añade fondo
ld      (ATTR_T), a     ; Actualiza atributo actual
ld      (ATTR_S), a     ; Actualiza atributo permanente
pop     bc              ; Recupera BC

ret
```

La rutina PAPER recibe el color en A. Nos quedamos con el color, rotamos tres bits a la izquierda para ponerlo en los bits del fondo, y lo guardamos en B. Obtenemos los atributos actuales, quitamos el fondo y le agregamos el que venía a A. Actualizamos las variables de sistema.

```
; ------------------------------------------------------------------
; Pinta cadenas terminadas en $FF.
;
; Entrada: HL -> Dirección de la cadena.
;
; Altera el valor de los registros AF y HL.
; ------------------------------------------------------------------
PrintString:
ld      a, (hl)     ; A = carácter a pintar
cp      $ff         ; ¿Es $FF?
```

```
ret     z           ; Sí, sale
rst     $10         ; Pinta carácter
inc     hl          ; HL = siguiente carácter
jr      PrintString ; Bucle hasta fin de cadena
```

La rutina PrintString ya la vimos en Batalla espacial, por lo que no necesita explicación.

Vamos a implementar rutinas para limpiar la pantalla completa, una línea y los atributos.

```
; ------------------------------------------------------------------
; Cambia los atributos de la VideoRAM con el atributo especificado
;
; Entrada:    A -> Atributo especificado (FBPPPIII)
;      Bits 0-2       Color de tinta (0-7).
;      Bits 3-5       Color de papel (0-7).
;      Bit  6         Brillo (0/1).
;      Bit  7         Parpadeo (0/1).
;
; Altera el valor de los registros BC, DE y HL.
; ------------------------------------------------------------------
CLA:
ld              hl, VIDEOATT          ; HL = inicio atributos
ld              (hl), a               ; Cambia atributo
ld              de, VIDEOATT+1        ; DE = 2ª dirección atributos VideoRAM
ld              bc, VIDEOATT_L-1      ; BC = longitud atributos - 1
ldir                                  ; Cambia los atributos

ld              (ATTR_T), a           ; Actualiza variable de sistema
ld              (ATTR_S), a           ; atributo actual, permanente
ld              (BORDCR), a           ; y borde
ret
```

La rutina CLA recibe en A los atributos a asignar a la pantalla, en el formato que ya hemos visto anteriormente. Los cambia y actualiza las variables de sistema de atributos actuales, permanentes y borde.

```
; ------------------------------------------------------------------
; Borra la línea de pantalla especificada.
;
; Entrada: B -> Línea de pantalla a borrar.
;               Coordenadas invertidas.
;
; Altera el valor de los registros AF.
; ------------------------------------------------------------------
CLL:
ld      a, (ATTR_T)     ; A = atributos actuales
and     $3f             ; A = PAPER + INK
ld      (ATTR_T), a     ; Actualiza en memoria

push    bc              ; Preserva BC

ld      c, INI_LEFT     ; C = 1ª columna
call    AT              ; Posiciona cursor.
ld      b, $20          ; B = 32 columnas*línea.
CLL_loop:
ld      a, $20          ; A = espacio
rst     $10             ; Lo imprime
djnz    CLL_loop        ; Bucle mientras B > 0

pop     bc              ; Recupera BC

ret
```

La rutina CLL recibe en A la línea a borrar (con las coordenadas invertidas) y la borra. Antes quita el brillo y el parpadeo de los atributos actuales.

```
; ------------------------------------------------------------------
; Borra los gráficos de la VideoRAM.
```

```
;
; Altera el valor de los registros BC, DE y HL.
; ------------------------------------------------------------------
CLS:
ld      hl, VIDEORAM        ; HL = dirección VideoRAM
ld      de, VIDEORAM+1      ; DE = 2ª dirección VideoRAM
ld      bc, VIDEORAM_L-1    ; BC = longitud VideoRAM - 1
ld      (hl), $00           ; Limpia 1ª posición
ldir                        ; Limpia el resto

ret
```

La rutina CLS borra el área gráfica de la pantalla, los píxeles.

Al igual que hicimos en Batalla espacial, vamos a pintar diversos datos numéricos en formato BDC, pero esta vez, debido a que el rango de los números va estar entre cero y diez, controlamos si la decena es cero para pintar un espacio en su lugar.

```
; ------------------------------------------------------------------
; Pinta el valor de números BCD.
;
; Sólo pinta números del 0 al 99.
;
; Entrada: HL -> Puntero al número a pintar.
;
; Altera el valor del registro AF.
; ------------------------------------------------------------------
PrintBCD:
ld      a, (hl)             ; A = número a pintar
and     $f0                 ; A = decenas
rrca
rrca
rrca
rrca                        ; Decenas a bits del 0 al 3
or      a                   ; ¿Decena = 0?
jr      nz, PrintBCD_ascii  ; No, salta
ld      a, ' '              ; Decena = 0
jr      PrintBCD_continue   ; Pinta espacio

PrintBCD_ascii:
add     a, '0'              ; A = A + código Ascii del 0

PrintBCD_continue:
rst     $10                 ; Pinta 1er dígito
ld      a, (hl)             ; A = número a pintar
and     $0f                 ; Se queda con las unidades
add     a, '0'              ; A = A + código Ascii del 0
rst     $10                 ; Imprime 2ª dígito
ret
```

PrintBCD recibe el HL el puntero al número a pintar, lo carga en A, se queda con las decenas y si son cero, pinta un espacio. El resto de la rutina es igual a lo visto en Batalla espacial.

Tenemos casi todo listo para pintar el tablero de juego, sólo nos falta la rutina que lo hace.

```
; ------------------------------------------------------------------
; Pinta el tablero.
;
; Altera el valor de los registros AF, BC, D y HL.
; ------------------------------------------------------------------
PrintBoard:
ld      a, INKBOARD         ; A = tinta
call    INK                 ; Cambia tinta

; Coordenadas iniciales del tablero
ld      b, INI_TOP - $06    ; B = coord Y
```

```
ld      c, INI_LEFT - $09   ; C = coord X
ld      d, $0e              ; D = nº líneas tablero

printBoard_1:
call    AT                  ; Posiciona cursor
ld      hl, Board_1         ; HL = línea vertical
call    PrintString         ; Pinta caracteres línea vertical
dec     b                   ; B-=1, siguiente línea
dec     d                   ; D-=1
jr      nz, printBoard_1    ; Bucle mientras D > 0

printBoard_2:
; Coordenadas de la primera línea horizontal
ld      b, INI_TOP - $0a    ; B = coord Y
ld      c, INI_LEFT - $09   ; C = coord X
call    AT                  ; Posiciona cursor
ld      hl, Board_2         ; HL = línea horizontal
call    PrintString         ; Pinta línea horizontal

; Coordenadas de la segunda línea horizontal, la coordenada X no cambia
ld      b, INI_TOP - $0f    ; B = coord Y
call    AT                  ; Posiciona cursor
ld      hl, Board_2
call    PrintString         ; Pinta línea horizontal

printBoard_3:
; Pinta los números de guía para que el usuario sepa que tecla pulsar
; para poner las fichas (O X O)
Ld      hl, Board_Helper    ; HL = helper
jp      PrintString         ; Pinta helper y sale
```

PrintBoard pinta el tablero de juego, cambia la tinta y prepara todo para pintar la línea vertical, que es pintada en printBoard_1. La línea horizontal la pinta printBoard_2, y la guía para que sepa el jugador que teclas pulsar la pinta printBoard_3.

Al pintar las líneas verticales y las horizontales, donde se cruzan la vertical se borra. Fijaos bien en la definición de Board2 y veréis como no se pinta el mismo carácter en toda ella.

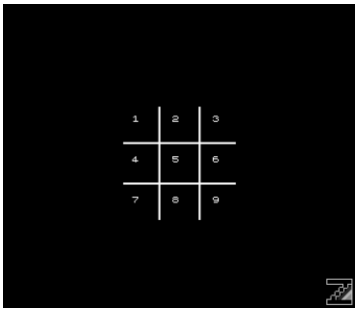
## Main

Para ver si todo funciona, implementamos la primera versión del archivo Main.asm.

```
org     $5e88

Main:
ld      hl, Sprite_P1
ld      (UDG), hl
ld      a, $02
call    OPENCHAN

xor     a
call    BORDER
call    CLA
call    CLS

Init:
call    PrintBoard

Loop:
jr      Loop

include "Rom.asm"
include "Screen.asm"
include "Sprite.asm"
include "Var.asm"

end     Main
```

Ponemos la dirección de inicio, teniendo en cuenta que haremos un cargador personificado. Indicamos dónde están los UDG, abrimos el canal dos, cambiamos borde, atributos, borramos la pantalla y pintamos el tablero. Nos quedamos en un bucle infinito que será el principal.

Compilamos, cargamos en el emulador y vemos los resultados.




Ya tenemos el tablero sobre el que se va a desarrollar la partida.

Descarga el código fuente desde aquí

https://tinyurl.com/2lmnm5e7

# Paso 2: marcadores y partida a dos

El siguiente paso es implementar los marcadores y la partida a dos jugadores, teniendo así gran parte del programa desarrollado.

## Información

En primer lugar añadiremos el nombre del programa, jugadores, marcadores y la ficha que mueve.

Agregamos en Var.asm los datos necesarios.

```
LENNAME:              equ $0c ; Longitud nombre jugadores

; Información
Info:                 db $10, $07, $16, $00, $05, "Tres en raya a "
info_points:          db "5 punto"
info_gt1:             db "s"

player1_title:        db $10, $05, $16, $02, $00
player1_name:         defs LENNAME, " "
player1_figure:       db $16, $04, $00, $90, $91, $0d, $91, $90

player_tie:           db $10, $02, $16, $02, $0d, "Tablas"

player2_title:        db $10, $06, $16, $02, $14
player2_name:         defs LENNAME, " "
player2_figure:       db $16, $04, $1b, $92, $93, $16, $05, $1b
                      db $94, $95, $ff

; Títulos
TitleTurn:            db $16, $15, $05, $13, $01
                      db "Turno para "
TitleTurn_name:       defs LENNAME, " "
                      db $ff
TitleError:           db $10, $02, $16, $15, $09, $12, $01, $13, $01
                      defm "Casilla ocupada", $ff
TitleEspamatica:      defm "Espamatica 2019", $ff
TitleGameOver:        db $10, $07, $16, $15, $03, $12, $01, $13, $01
                      defm "Partida finalizada. Otra?", $ff
TitleLostMovement:    db $10, $02, $12, $01, $13, $01, $16, $15, $05
                      defm "Pierde turno "
TitleLostMov_name:    defs LENNAME, " "
                      db $ff
TitleOptionStart:     defm "0. Empezar", $ff
TitleOptionPlayer:    defm "1. Jugadores", $ff
TitleOptionPoint:     defm "2. Puntos", $ff
TitleOptionTime:      defm "3. Tiempo", $ff
TitlePlayerName:      defm "Nombre jugador "
TitlePlayerNumber:    db " : ", $ff
TitlePointFor:        db $10, $07, $16, $15, $05, $12, $01, $13, $01
                      defm "Punto para "
TitlePointName:       defs LENNAME, " "
                      db $ff
TitleTie:             db $10, $07, $16, $15, $0d, $12, $01, $13, $01
                      defm "Tablas", $ff

; ------------------------------------------------------------------
; Desarrollo de la partida.
; ------------------------------------------------------------------
; Casillas del tablero. Un byte por casilla, del 1 al 9.
; Bit 0, casilla ocupada por jugador 1.
; Bit 4, casilla ocupada por jugador 2.
Grid:          db $00, $00, $00, $00, $00, $00, $00, $00, $00
; Posiciones de las fichas en el tablero, 2 bytes por ficha X, Y
GridPos:       db POS1_LEFT, POS1_TOP ; 1
               db POS2_LEFT, POS1_TOP ; 2
```

```
              db POS3_LEFT, POS1_TOP ; 3
              db POS1_LEFT, POS2_TOP ; 4
              db POS2_LEFT, POS2_TOP ; 5
              db POS3_LEFT, POS2_TOP ; 6
              db POS1_LEFT, POS3_TOP ; 7
              db POS2_LEFT, POS3_TOP ; 8
              db POS3_LEFT, POS3_TOP ; 9

Points_p1:    db $00                 ; Puntos jugador 1
Points_p2:    db $00                 ; Puntos jugador 2
Points_tie:   db $00                 ; Puntos tablas
PlayerMoves:  db $00                 ; Jugador que mueve

MaxPlayers:   db $01                 ; Máximo jugadores
MaxPoints:    db $05                 ; Máximo puntos
MaxSeconds:   db $10                 ; Máximo segundos

Name_p1:       db "Amstrad CPC "     ; Nombre jugador 1
Name_p2:       db "ZX Spectrum "     ; Nombre jugador 2
Name_p2Default:db "ZX Spectrum "     ; Nombre por defecto
```

En la etiqueta Info definimos los literales que se van a mostrar en la parte superior de la pantalla. Aunque lo veáis raro, la división que se hace en varias etiquetas se debe a que hay valores que cambian por la selección que hagan los jugadores:

- Puntos necesarios para ganar la partida.
- Si son más de un punto, se pone en plural.
- Nombre de los jugadores.

También se definen las etiquetas que vamos a usar para llevar la cuenta de la puntuación de cada jugador y sus nombres.

Vamos a Screen.asm e implementamos una rutina para pintar la información y otra para pintar la puntuación.

```
; ------------------------------------------------------------------
; Pinta la información de la partida.
;
; Altera el valor de HL.
; ------------------------------------------------------------------
PrintInfo:
ld      hl, Info        ; HL = información
jp      PrintString     ; La pinta
```

PrintInfo pinta la información a mostrar en la parte superior de la pantalla. No asigna colores de tinta, ni posiciones, ya que todo eso está definido en la etiqueta Info.

```
; ------------------------------------------------------------------
; Pinta la puntuación.
;
; Altera el valor de AF, BC, DE y HL.
; ------------------------------------------------------------------
PrintPoints:
ld      a, $05          ; A = tinta cyan
call    INK             ; Cambia tinta
ld      b, INI_TOP-$04  ; B = coord Y
ld      c, INI_LEFT-$03 ; C = coord X
call    AT              ; Posiciona cursor
ld      hl, Points_p1   ; HL = puntos jugador 1
call    PrintBCD        ; Los pinta

ld      a, $02          ; A = tinta roja
call    INK             ; Cambia tinta
ld      b, INI_TOP-$04  ; B = coord Y
ld      c, INI_LEFT-$0f ; C = coord X
call    AT              ; Posiciona cursor
ld      hl, Points_tie  ; HL = puntos tablas
call    PrintBCD        ; Los pinta
```

```
ld      a, $06          ; A = tinta amarilla
call    INK             ; Cambia tinta
ld      b, INI_TOP-$04  ; B = coord Y
ld      c, INI_LEFT-$1e ; C = coord X
call    AT              ; Posiciona cursor
ld      hl, Points_p2   ; HL = puntos jugador 2
jp      PrintBCD        ; Los pinta y sale
```

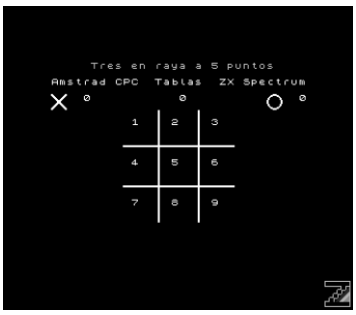
PrintPoints pinta la puntuación de los jugadores y las tablas. Por cada valor, cambia la tinta, posiciona el cursor y pinta el valor.

Ya lo tenemos todo listo. Vamos a Main.asm y tras la llamada a PrintBoard, añadimos el código que pinta la información y la puntuación.

```
ld      hl, Name_p1
ld      de, player1_name
ld      bc, LENNAME
ldir
ld      hl, Name_p2
ld      de, player2_name
ld      bc, LENNAME
ldir
call    PrintInfo

xor     a
ld      hl, Points_p1
ld      de, Points_p1+$01
ld      bc, $03
ld      (hl), a
ldir
call    PrintPoints
```

Los nombres de los jugadores están en las etiquetas Name_p1 y Name_p2, por lo que tenemos que pasarlos al lugar correspondiente de la información. Inicializamos la puntuación y la pintamos.

Compilamos, cargamos en el emulador y vemos los resultados.




## Partida a dos

Vamos a implementar la partida a dos jugadores: controles, movimientos de ficha, comprobaciones de que sean correctos y de consecución de tres en raya.

En Var.asm incluimos una serie de constantes con los colores de tinta, líneas en las que se producen las tres en raya, y códigos de teclas, que vamos a usar junto a la rutina de la ROM para verificar los controles.

```
INKWARNING:     equ $C2 ; Tinta advertencias
INKPLAYER1:     equ $05 ; Tinta jugador 1
INKPLAYER2:     equ $06 ; Tinta jugador 2
INKTIE:         equ $07 ; Tinta tablas
```

```
KEYDEL:         equ $0c ; Tecla Delete
KEYENT:         equ $0d ; Tecla Enter
KEYSPC:         equ $20 ; Tecla Space
KEY0:           equ $30 ; Tecla 0
KEY1:           equ $31 ; Tecla 1
KEY2:           equ $32 ; Tecla 2
KEY3:           equ $33 ; Tecla 3
KEY4:           equ $34 ; Tecla 4
KEY5:           equ $35 ; Tecla 5
KEY6:           equ $36 ; Tecla 6
KEY7:           equ $37 ; Tecla 7
KEY8:           equ $38 ; Tecla 8
KEY9:           equ $39 ; Tecla 9

WINNERLINE123:  equ $01 ; Línea ganadora 123
WINNERLINE456:  equ $02 ; Línea ganadora 456
WINNERLINE789:  equ $03 ; Línea ganadora 789
WINNERLINE147:  equ $04 ; Línea ganadora 147
WINNERLINE258:  equ $05 ; Línea ganadora 258
WINNERLINE369:  equ $06 ; Línea ganadora 369
WINNERLINE159:  equ $07 ; Línea ganadora 159
WINNERLINE357:  equ $08 ; Línea ganadora 357
```

La lógica la vamos a implementar en el archivo Game.asm. Lo creamos y añadimos el include en Main.asm.

```
; ------------------------------------------------------------------
; Espera a que se pulse alguna tecla del tablero.
; Durante el juego se activan las interrupciones para realizar la
; cuenta atrás.
; Con las interrupciones activadas no actualiza FLAGS_KEY/LAST_KEY.
;
; Salida:   C   ->  Tecla pulsada.
;
; Altera el valor de los registros AF y BC.
; ------------------------------------------------------------------
WaitKeyBoard:
ld      a, $f7          ; A = semifila 1-5
in      a, ($fe)        ; Lee semifila
cpl                     ; Invierte bits, teclas pulsadas a 1
and     $1F             ; A = bits 1 a 4
jr      z, waitKey_cont ; Ninguna tecla pulsada, salta

; Evalúa la tecla pulsada del 1 al 5
ld      c, KEY1         ; C = tecla 1
ld      b, $05          ; B = teclas a comprobar
waitKey_1_5:
rra                     ; ¿Tecla pulsada?
ret     c               ; Pulsada, salta
inc     c               ; C = código siguiente tecla
djnz    waitKey_1_5     ; Bucle para 5 teclas

waitKey_cont:
ld      a, $ef          ; A = semifila 0-6
in      a, ($fe)        ; Lee semifila
cpl                     ; Invierte bits, teclas pulsadas a 1
and     $1F             ; A = bits 1 a 4
jr      z, waitKey_end  ; Ninguna tecla pulsada, salta

; Evalúa la tecla pulsada 9 al 6
rra                     ; Se salta la tecla cero
ld      c, KEY9         ; C = tecla 9
ld      b, $04          ; B = teclas a comprobar
waitKey_9_6:
rra                     ; ¿Tecla pulsada?
ret     c               ; Pulsada, salta
dec     c               ; C = código siguiente tecla
djnz    waitKey_9_6     ; Bucle para 4 teclas

; No se ha pulsado ninguna tecla
waitKey_end:
```

```
ld      c, KEY0
ret
```

Con WaitKeyBoard vamos a comprobar si se ha pulsado alguna de las teclas del tablero, del uno al nueve. En este caso no vamos a usar la rutina de la ROM ya que, más adelante, tendremos las interrupciones en modo dos activas, y no se actualizarán ni FLAGS_KEY ni LAST_KEY, y la ROM no podrá decirnos la última tecla pulsada.

La comprobación de las teclas es muy parecida a la que hicimos en Batalla espacial, con las siguientes variantes:

- CPL: invertimos los bit y evaluamos si se pulsó alguna tecla con AND $1F; saltamos si no se pulsó ninguna.
- Lo que devolvemos, en este caso en C, es el código ASCII de la tecla pulsada; de la tecla cero si no se ha pulsado ninguna.

También se puede ver que antes de comprobar las teclas del 9 al 6 hacemos una rotación para ignorar la tecla 0.

Continuamos ahora con la lógica del movimiento de la ficha.

```
; ------------------------------------------------------------------
; Comprueba y realiza el movimiento si es correcto.
;
; Entrada:  C   ->  Tecla pulsada
; Salida:   Z   ->  Movimiento correcto
;           NZ  ->  Movimiento incorrecto
;
; Altera el valor de los registros AF y HL.
; ------------------------------------------------------------------
ToMove:
push        bc                ; Perserva BC
ld          hl, Grid          ; HL = dirección grid
ld          a, c              ; A = C (código tecla)
sub         $30               ; A = valor numérico tecla
dec         a                 ; A-=1, para que no sume uno más
ld          b, $00            ; B = 0
ld          c, a              ; C = A, BC = desplazamiento
add         hl, bc            ; HL = dirección casilla de tecla
pop         bc                ; Recupera BC

ld          a, (hl)           ; A = valor casilla
or          a                 ; ¿Está libre?
ret         nz                ; Ocupada, sale

ld          a, (PlayerMoves)  ; A = jugador que mueve
or          a                 ; ¿Jugador 1?
jr          nz, toMove_p2     ; Jugador 2, salta
set         $00, (hl)         ; Activa casilla jugador 1
jr          toMove_end        ; Salta

toMove_p2:
set        $04, (hl)          ; Activa casilla jugador 2

toMove_end:
xor         a                 ; Pone flag Z
ret                           ; Sale
```

ToMove recibe en C el código de la tecla pulsada. Calculamos el valor restando el código ASCII de cero, restamos uno para no sumar uno de más en el desplazamiento, y apuntamos a la celda correspondiente a la tecla pulsada. Comprobamos si la celda no está ocupada, si no lo está activamos los bits uno o cuatro de la celda según el jugador que mueve. Si la celda está ocupada sale con NZ, si no lo está sale con Z (XOR A).

Necesitamos una rutina que pinte la ficha en el lugar correcto; la implementamos en Screen.asm.

```
; ------------------------------------------------------------------
; Pinta la ficha.
;
; Entrada:  C   ->  Tecla pulsada.
;
; Altera el valor de los registros AF, BC y HL.
; ------------------------------------------------------------------
PrintOXO:
; Cálculo posición ficha
ld      a, c                ; A = tecla
sub     $30                 ; A = valor tecla
dec     a                   ; A-=1, para no sumar de más en desplazamiento
add     a, a                ; A+=A, desplazamiento, dos bytes por posición
ld      b, $00              ; B = 0
ld      c, a                ; C = A
ld      hl, GridPos         ; HL = dirección posiciones grid
add     hl, bc              ; HL+=BC, dirección coord X celda
ld      c, (hl)             ; C = coord X celda
inc     hl                  ; HL = dirección coord Y
ld      b, (hl)             ; B = coord Y celda
call    AT                  ; Posiciona cursor

; Cálculo ficha
ld      a, (PlayerMoves)    ; A = jugador que mueve
or      a                   ; Comprueba jugador
jr      nz, printOXO_Y      ; No cero, salta

printOXO_X:
ld      a, INKPLAYER1       ; A = tinta jugador 1
call    INK                 ; Cambia tinta
ld      a, $90              ; A = 1er sprite
rst     $10                 ; Lo pinta
ld      a, $91              ; A = 2º sprite
rst     $10                 ; Lo pinta
dec     b                   ; B = línea inferior
call    AT                  ; Posiciona cursor
ld      a, $91              ; A = 2º sprite
rst     $10                 ; Lo pinta
ld      a, $90              ; A = 1er sprite
rst     $10                 ; Lo pinta
ret                         ; Sale

printOXO_Y:
ld      a, INKPLAYER2       ; A = tinta jugador 2
call    INK                 ; Cambia tinta
ld      a, $92              ; A = 1er sprite
rst     $10                 ; Lo pinta
ld      a, $93              ; A = 2º sprite
rst     $10                 ; Lo pinta
dec     b                   ; B = línea inferior
call    AT                  ; Posiciona cursor
ld      a, $94              ; A = 3er sprite
rst     $10                 ; Lo pinta
ld      a, $95              ; A = 4º sprite
rst     $10                 ; Lo pinta
ret                         ; Sale
```

En la primera parte de PrintOXO se calcula el desplazamiento para obtener las coordenadas donde pintar la ficha, se obtienen y se posiciona el cursor. Tras esto se obtiene el jugador que mueve.

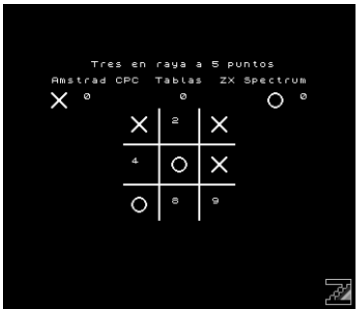
La forma de pintar una u otra ficha son casi idénticas: cambia la tinta, pinta la parte de arriba, posiciona el cursor en la línea inferior, y pinta la parte de abajo antes de salir.

Ya podemos empezar a probar como se ve en el programa todo lo implementado. Vamos a Main.asm y debajo de la etiqueta Loop, antes de JR Loop, añadimos las llamadas a las rutinas implementadas.

```
call    WaitKeyBoard
ld      a, c
cp      KEY0
jr      z, loop_cont
call    ToMove
jr      nz, loop_cont
call    PrintOXO
ld      a, (PlayerMoves)
xor     $01
ld      (PlayerMoves), a
loop_cont:
```

Esperamos a que el jugador pulse una tecla, si la ha pulsado se comprueba si el movimiento es correcto, y si es así pintamos la ficha y cambiamos de jugador.

Compilamos, cargamos en el emulador y vemos los resultados.




Vemos como las fichas se pintan y una vez que se ocupa todo el tablero sólo podemos reiniciar y volver a cargar, por lo que tenemos que implementar lo siguiente:

- Si el tablero está lleno, reiniciarlo.
- Mostrar mensajes de información.
- Comprobar si hay tres en raya o tablas.
- Actualizar el marcador.

## Reinicio del tablero

En Var.asm, antes de PlayersMoves, añadimos una etiqueta para llevar la cuenta del número de movimientos. El tablero estará lleno en el momento en el que el valor llegue a nueve.

La ponemos antes porque PlayersMoves sólo la vamos a reiniciar cuando si inicia una partida. Si la reiniciamos con cada punto, el jugador uno siempre movería primero.

```
MoveCounter:   db $00  ; Contador de movimientos
```

También en Var.asm añadimos una constante al inicio, en la que especificamos la longitud de datos a inicializar en cada partida.

```
LENDATA:       equ $04 ; Longitud de los datos a inicializar
```

Según sea necesario añadir más datos que debamos inicializar, cambiaremos el valor de esta constante.

En Main.asm localizamos la etiqueta Loop y cuatro líneas antes la línea LD BC, $03. Sustituimos está línea por la siguiente:

```
ld      bc, LENDATA
```

Compilamos, cargamos en el emulador y comprobamos que todo sigue funcionando.

Como hemos comentado, no siempre vamos a inicializar todos los datos, por lo que es buena idea implementar una rutina a la que se le pueda pasar la longitud de los datos a inicializar, y así poder llamarla desde distintos sitios, con distintos valores.

Los valores que vamos a inicializar son:

- Points_p1
- Points_p2
- Points_tie
- MoveCounter
- PlayerMoves (solo al iniciar la partida)

Hay más valores que debemos inicializar, pero están más arriba, por lo que, o los inicializamos en dos partes, o cambiamos la etiqueta de sitio. La etiqueta es Grid (movimientos de cada jugador), y debemos de poner todos los valores a cero al inicio de cada punto. Vamos a cambiar la etiqueta de lugar, justo por encima de Points_p1, quedando así:

```
; ------------------------------------------------------------------
; Desarrollo de la partida.
; ------------------------------------------------------------------
; Posiciones de las fichas en el tablero, 2 bytes por ficha Y, X
GridPos:       db POS1_LEFT, POS1_TOP ; 1
               db POS2_LEFT, POS1_TOP ; 2
               db POS3_LEFT, POS1_TOP ; 3
               db POS1_LEFT, POS2_TOP ; 4
               db POS2_LEFT, POS2_TOP ; 5
               db POS3_LEFT, POS2_TOP ; 6
               db POS1_LEFT, POS3_TOP ; 7
               db POS2_LEFT, POS3_TOP ; 8
               db POS3_LEFT, POS3_TOP ; 9

; Casillas del tablero. Un byte por casilla, del 1 al 9.
; Bit 0 a 1, casilla ocupada por jugador 1.
; Bit 4 a 1, casilla ocupada por jugador 2.
Grid:          db $00, $00, $00, $00, $00, $00, $00, $00, $00
Points_p1:     db $00                ; Puntos jugador 1
Points_p2:     db $00                ; Puntos jugador 2
Points_tie:    db $00                ; Puntos tablas
MoveCounter:   db $00                ; Contador de movimientos
PlayerMoves:   db $00                ; Jugador que mueve
```

El valor de LENDATA hay que cambiarlo por $0D.

Implementamos la rutina de inicialización en Game.asm.

```
; ------------------------------------------------------------------
; Inicializa los valores de la partida/punto.
;
; Entrada:  BC  ->  Longitud de los valores a inicializar.
;
; Altera el valor de los registros BC, DE y HL.
; ------------------------------------------------------------------
ResetValues:
ld      hl, Grid        ; HL = dirección de Grid
ld      de, Grid+$01    ; DE = dirección de Grid+1
ld      (hl), $00       ; Pone a cero la primera posición
ldir                    ; Pone a cero el resto (BC)

ret
```

ResetValues recibe en BC el número de bytes a limpiar (debe ser uno menos de la longitud total, por eso LENDATA es igual a $0D en lugar de $0E), apunta HL y DE a la primera y segunda posición de Grid, limpia la primera y luego el resto.

Si os fijáis bien, esta rutina os debe de sonar mucho, id y mirad la rutina CLS de Screen.asm. ¡Son casi iguales!

Podríamos implementar una rutina con las líneas LD (HL), $00, LDIR y RET, teniendo en cuenta que antes de llamarla habría que cargar los valores necesarios en HL y BC, y eliminar CLS y ResetValues. Lo dejo a vuestra elección, yo lo voy a dejar como está ahora.

Volvemos a Main.asm, localizamos Loop y unas líneas más arriba XOR A. Desde XOR A a CALL PrintPoint, sólo vamos a dejar las líneas LD BC, LENDATA y CALL PrintPoints, el resto las borramos.

Entre LD BC, LENDATA y CALL PrintPoints añadimos la llamada a la inicialización de los valores, quedando así:

```
call    PrintInfo

ld      bc, LENDATA
call    ResetValues
call    PrintPoints

Loop:
```

Tras cargar la longitud de los datos a limpiar en BC, llamamos a la inicialización de los valores y pintamos los puntos.

Compilamos, cargamos en el emulador y vemos que todo sigue funcionando.

Para poder saber si el tablero está lleno, tenemos que actualizar MoveCounter con cada movimiento y comprobar si ha llegado a nueve.

Localizamos loop_cont y añadimos las líneas siguientes justo por encima, después de LD (PlayerMoves), A:

```
ld      hl, MoveCounter
inc     (hl)
ld      a, $09
cp      (hl)
jr      nz, loop_cont
ld      bc, LENDATA-$01
call    ResetValues
call    PrintBoard
```

Incrementamos el contador de movimientos, comprobamos si ha llegado a nueve, en cuyo caso cargamos en BC la longitud de los datos a limpiar menos uno (para no limpiar el jugador que mueve), inicializamos los datos y pintamos el tablero vacío.

Compilamos y cargamos en el emulador. Si vamos pulsando las teclas del uno al nueve, hasta que se llene el tablero, el jugador al que le toca mover no se ha limpiado y mueve al que le tocaba en el siguiente turno.

## Mensajes de información

Durante el transcurso de la partida se deben mostrar diversos mensajes a los jugadores: si el movimiento es erróneo, el jugador que mueve, quién consigue el punto, si hay tablas y si la partida a finalizado.

Los mensajes ya los definimos al inicio del capítulo, y en cada uno de ellos la tinta, brillo, parpadeo y localización, por lo que lo único que tendríamos que hacer es llamar a PrintString con la dirección del mensaje en HL.

Hay dos motivos para no poder hacerlo así:

- Los mensajes Turno para, Pierde turno y Punto para no están completos, se completan con el nombre del jugador en posesión del turno.
- Antes de escribir un mensaje, hay que borrar la línea.

Vamos a implementar dos rutinas, una en Game.asm y la otra en Screen.asm; estas rutinas nos harán de puente cuando llamemos a PrintString.

Empezamos por la rutina que implementamos en Screen.asm.

```
; ------------------------------------------------------------------
; Pinta los mensajes
;
; Entrada:     HL      -> dirección del mensaje
; ------------------------------------------------------------------
PrintMsg:
ld      b, INI_TOP-$15 ; B = coord Y línea a borrar (invertida)
call    CLL            ; Borra la línea
jp      PrintString    ; Pinta el mensaje
```

PrintMsg recibe en HL la dirección del mensaje, borra la línea de mensajes y salta a pintar el mensaje; sale por allí. Borrar la línea antes de pintar un nuevo mensaje es la única utilidad de esta rutina.

Implementamos la rutina de Game.asm, que va a completar los tres mensajes que pintan el nombre del jugador.

```
; ------------------------------------------------------------------
; Completa los mensajes en los que se muestra el nombre del jugador.
;
; Entrada:  HL  ->  dirección del mensaje
;           DE  ->  dirección donde debe ir el nombre
;
; Altera el valor de los registros AF, BC y DE.
; ------------------------------------------------------------------
DoMsg:
push    hl                  ; Perserva HL
ld      hl, player1_name    ; HL = nombre jugador 1
ld      a, (PlayerMoves)    ; A = jugador
or      a                   ; ¿Jugador 1?
ld      a, INKPLAYER1       ; A = tinta jugador 1
jr      z, doMsg_cont       ; Jugador 1, salta
ld      hl, player2_name    ; HL = nombre jugador 2
ld      a, INKPLAYER2       ; A = tinta jugador 2
doMsg_cont:
call    INK                 ; Cambia tinta
ld      bc, LENNAME         ; BC = longitud nombre
ldir                        ; Pasa nombre jugador a mensaje
pop     hl                  ; Recupera HL
jp      PrintMsg            ; Pinta mensaje
```

DoMsg recibe en HL la dirección de mensaje a pintar, y en DE la dirección dónde se debe poner el nombre del jugador. Comprueba que jugador tiene el turno, pone uno u otro nombre y la tinta del jugador. Pinta el mensaje y sale por allí.

Vamos a Main.asm y pintamos todos los mensajes que podemos pintar, por ahora, empezando por quién tiene el turno.

Localizamos Loop, y justo debajo añadimos las siguientes líneas:

```
ld      b, $19
loop_wait:
halt
djnz    loop_wait
ld      hl, TitleTurn
ld      de, TitleTurn_name
call    DoMsg
loop_key:
```

Cada vez que pasamos por el bucle pintamos el jugador al que le toca mover. Antes realizamos una pausa de como medio segundo para que dé tiempo a leer los mensajes.

Ahora localizamos la línea CP KEY0 y, justo debajo, en la línea JR Z, loop_cont, cambiamos loop_cont por loop_key.

Compilamos, cargamos en el emulador y vemos que ya pintamos a que jugador le toca mover.


Vamos a pintar también el mensaje de error de casilla ocupada. Localizamos la línea CALL ToMove, borramos la línea siguiente, JR NZ, loop_cont, y añadimos las siguientes:

```
jr      z, loop_print
ld      hl, TitleError
call    PrintMsg
jr      Loop
loop_print:
```

Si retorna de la rutina ToMove con el flag Z activo, significa que el movimiento es correcto y saltamos a loop_print, etiqueta que hemos añadido antes de CALL PrintOXO. Si no es el caso, el movimiento no es correcto y pintamos el mensaje de error y volvemos al inicio del bucle.

Compilad, cargad en el emulador e intentad mover a una casilla ocupada. Se debe pintar el mensaje de error, aunque se ve durante un corto espacio de tiempo; no os preocupéis, más adelante quitaremos la pausa al inicio de Loop y utilizaremos los efectos de sonido para poder temporizar.

## Comprobación de tablas

La comprobación de tablas es sencilla; si se ha llegado a nueve movimientos y no hay tres en raya, hay tablas.

Localizamos la etiqueta loop_cont y cuatro líneas por encima, entre JR NZ, loop_cont y LD BC, LENDATA-$01, añadimos lo siguiente:

```
ld      hl, Points_tie
inc     (hl)
ld      hl, TitleTie
call    PrintMsg
```

Con estas líneas, si se llega a nueve movimientos sin haber tres en raya, se suma uno al marcador de tablas.

Id a la etiqueta loop_key y encima añadid la siguiente línea:

```
call    PrintPoints
```

Ahora, en cada iteración del bucle pintamos las puntuaciones. El CALL PrintPoints que hay justo encima de Loop lo podemos quitar.

Compilad, cargad en el emulador y veréis como no todo funciona como debería, no se actualiza el marcador de tablas.

En realidad todo está funcionando como debe, el problema está en que ResetValues está reiniciando las puntuaciones, lo cual es fácil de solucionar, aunque lo haremos más adelante.

También observamos que al repintar el tablero el parpadeo está activo, debido a que el mensaje Tablas lo deja así.

Para solucionarlo podríamos implementar una rutina para quitar el parpadeo, pero optamos por solucionarlo en la definición del tablero. Vamos a Var.asm y justo debajo de la etiqueta Board_1 añadimos la siguiente línea para desactivar el parpadeo y brillo al pintar el tablero.

```
db $12, $00, $13, $00
```

Compilamos, cargamos en el emulador, movemos hasta llenar el tablero y comprobamos que al repintarse ya no hay parpadeo.

## Comprobación de tres en raya

La comprobación de tres en raya no tendríamos que realizarla hasta que se hayan realizado al menos tres movimientos. En principio, se deberían dar al menos cinco movimientos antes de poder haber tres en raya, pero dado que vamos a implementar la posibilidad de que el jugador pierda el turno si tarda más de lo establecido en realizarlo, es posible que con tres movimientos haya tres en raya. No obstante, vamos a realizar la operación cada vez que se mueva una ficha, y así siempre tardará lo mismo cada iteración del bucle (aproximadamente).

En Game.asm añadimos la rutina que comprueba si hay tres en raya.

```
; ------------------------------------------------------------------
; Comprueba si hay tres en raya.
;
; Retorno:  A    ->  línea de tres en raya
;           Z si hay tres en raya, NZ en el caso contrario.
;
; Altera el valor de los registros AF, B e IX.
; ------------------------------------------------------------------
CheckWinner:
ld      ix, Grid-$01            ; IX = dirección grid - 1
ld      b, $03                  ; B = suma celdas jugador 1
ld      a, (PlayerMoves)        ; A = jugador
or      a                       ; ¿Jugador 1?
jr      z, CheckWinner_check    ; Jugador 1, salta
ld      b, $30                  ; B = suma celdas jugador 2

CheckWinner_check:
ld      a, (ix+1)               ; A = celda 1
add     a, (ix+2)               ; A+= celda 2
```

```
add     a, (ix+3)               ; A+= celda 3
cp      b                       ; ¿Tres en raya?
ld      a, WINNERLINE123        ; A = indicador línea 123
ret     z                       ; Tres en raya, sale

ld      a, (ix+4)               ; A = celda 4
add     a, (ix+5)               ; A+= celda 5
add     a, (ix+6)               ; A+= celda 6
cp      b                       ; ¿Tres en raya?
ld      a, WINNERLINE456        ; A = indicador línea 456
ret     z                       ; Tres en raya, sale

ld      a, (ix+7)               ; A = celda 7
add     a, (ix+8)               ; A+= celda 8
add     a, (ix+9)               ; A+= celda 9
cp      b                       ; ¿Tres en raya?
ld      a, WINNERLINE789        ; A = indicador línea 789
ret     z                       ; Tres en raya, sale

ld      a, (ix+1)                 ; A = celda 1
add     a, (ix+4)                 ; A+= celda 4
add     a, (ix+7)                 ; A+= celda 7
cp      b                         ; ¿Tres en raya?
ld      a, WINNERLINE147          ; A = indicador línea 147
ret     z                         ; Tres en raya, sale

ld      a, (ix+2)                 ; A = celda 2
add     a, (ix+5)                 ; A+= celda 5
add     a, (ix+8)                 ; A+= celda 8
cp      b                         ; ¿Tres en raya?
ld      a, WINNERLINE258          ; A = indicador línea 258
ret     z                         ; Tres en raya, sale

ld      a, (ix+3)                 ; A = celda 3
add     a, (ix+6)                 ; A+= celda 6
add     a, (ix+9)                 ; A+= celda 9
cp      b                         ; ¿Tres en raya?
ld      a, WINNERLINE369          ; A = indicador línea 369
ret     z                         ; Tres en raya, sale

ld      a, (ix+1)                 ; A = celda 1
add     a, (ix+5)                 ; A+= celda 5
add     a, (ix+9)                 ; A+= celda 9
cp      b                         ; ¿Tres en raya?
ld      a, WINNERLINE159          ; A = indicador línea 159
ret     z                         ; Tres en raya, sale

ld      a, (ix+3)                 ; A = celda 3
add     a, (ix+5)                 ; A+= celda 5
add     a, (ix+7)                 ; A+= celda 7
cp      b                         ; ¿Tres en raya?
ld      a, WINNERLINE357          ; A = indicador línea 357
ret                               ; Ultima condición, siempre sale
```

En CheckWinner apuntamos IX a la dirección anterior a Grid, en B cargamos el valor que deben sumar las celdas en el caso de que gane uno u otro jugador: tres para el jugador uno y treinta para el dos.

Luego, vamos comprobando las posibles combinaciones de tres en raya que hay y salimos si se ha producido alguna, con la combinación ganadora en A y el flag Z activado. Para comprobar si ha habido tres en raya, sumamos los valores de las celdas en A y lo compramos con B.

En la última comprobación, si no se han logrado las tres en raya, se sale con el flag Z desactivado (NZ).

En Main.asm vamos a añadir las líneas para la comprobación de tres en raya, y para actuar según se haya conseguido o no. Añadimos las líneas siguientes justo debajo de CALL PrintOXO.

```
loop_checkWinner:
call    CheckWinner
jr      nz, loop_tie
ld      hl, TitlePointFor
ld      de, TitlePointName
call    DoMsg
ld      hl, Points_p1
ld      a, (PlayerMoves)
or      a
jr      z, loop_win
inc     hl
loop_win:
inc     (hl)
jr      loop_reset
loop_tie:
```

Llamamos a la comprobación de tres en raya y saltamos si no las hay. Si hay tres en raya, pintamos el mensaje de *Punto para*, obtenemos que jugador lo ha conseguido e incrementamos su marcador.

Justo debajo de loop_tie están estas líneas:

```
ld      a, (PlayerMoves)
xor     $01
ld      (PlayerMoves), a
```

Las quitamos y las ponemos debajo de la etiqueta loop_cont, de lo contrario el siguiente punto lo empieza el mismo jugador que ganó el anterior.

Por último, antes de LD BC, LENDATA-$01 añadimos la etiqueta loop_reset.

Compilamos, cargamos en el emulador y todo parece funcionar, excepto los marcadores que siguen sin actualizarse.

Vamos a marcar visualmente dónde se han producido las tres en raya, dibujando una diagonal que cruce las tres fichas. Implementamos la rutina en Screen.asm.

```
; ------------------------------------------------------------------
; Imprime la línea ganadora.
;
; Entrada:     A       ->      Línea ganadora
;
; Altera el valor de los registros AF, BC, DE y HL.
; ------------------------------------------------------------------
PrintWinnerLine:
ld      hl, COORDX              ; HL = coord X
ld      bc, $6c6c               ; BC = desplazamiento
ld      de, $01ff               ; DE = orientación
cp      WINNERLINE159           ; ¿Gana línea 159?
jr      z, printWinnerLine_159; Sí, salta
ld      e, $01                  ; DE = orientación
cp      WINNERLINE357           ; ¿Gana línea 357?
jr      z, printWinnerLine_357; Sí, salta

ld      c, $00                  ; Desplazamiento
cp      WINNERLINE147           ; ¿Gana línea 147?
jr      z, printWinnerLine_147; Sí, salta
cp      WINNERLINE258           ; ¿Gana línea 258?
jr      z, printWinnerLine_258; Sí, salta
cp      WINNERLINE369           ; ¿Gana línea 369?
jr      z, printWinnerLine_369; Sí, salta
```

```
ld      bc, $006c             ; Desplazamiento
ld      (hl), $48             ; Coord X
inc     hl                    ; Apunta HL a coord Y
cp      WINNERLINE123         ; ¿Gana línea 123?
jr      z, printWinnerLine_123; Sí, salta
cp      WINNERLINE456         ; ¿Gana línea 456?
jr      z, printWinnerLine_456; Sí, salta
cp      WINNERLINE789         ; ¿Gana línea 789?
jr      z, printWinnerLine_789; Si es así, salta a pintarla.

printWinnerLine_159:
ld      (hl), $b7             ; Coord X
jr      printWinnerLine_Y

printWinnerLine_357:
ld      (hl), $48             ; Coord X
jr      printWinnerLine_Y

printWinnerLine_147:
ld      (hl), $58             ; Coord X
jr      printWinnerLine_Y

printWinnerLine_258:
ld      (hl), $80             ; Coord X
jr      printWinnerLine_Y

printWinnerLine_369:
ld      (hl), $a8             ; Coord X

printWinnerLine_Y:
inc     hl                    ; Apunta HL a coord Y
ld      (hl), $10             ; Coord Y
jr      printWinnerLine_end   ; Pinta la línea

printWinnerLine_123:
ld      (hl), $70             ; Coord Y
jr      printWinnerLine_end   ; Pinta la línea

printWinnerLine_456:
ld      (hl), $47             ; Coord Y
jr      printWinnerLine_end   ; Pinta la línea

printWinnerLine_789:
ld      (hl), $20             ; Coord Y

printWinnerLine_end:
jp      DRAW                  ; Pinta la línea
```

PrintWinnerLine recibe en A la línea ganadora. En primer lugar evaluamos cual es esa línea para saltar a una parte u otra de la rutina. Dado que las distintas líneas a pintar comparten datos comunes, se van cambiando solo los datos que difieren y se llama a DRAW para pintar la línea y salir por allí (para más información ver comentarios de DRAW en ROM.asm).

En Main.asm, localizamos la etiqueta loop_checkWinner y debajo de JR NZ, loop_tie añadimos la línea siguiente:

```
call    PrintWinnerLine
```

Compilamos, cargamos en el emulador y vemos los resultados.


Como podemos comprobar se pinta una línea marcando las tres en raya, aunque es tan rápido que apenas podemos verlo. En el próximo capítulo usaremos el sonido para temporizar.

No obstante, si queréis comprobar que se pinta la línea, podéis poner debajo de CALL PrintWinnerLine estas dos líneas:

```
tmp:
jr      tmp
```

No olvidéis quitarlas luego.

## Actualización del marcador

Vamos a arreglar el error que arrastramos desde hace tiempo y que provoca que el marcador no se esté actualizando.

En Var.asm localizamos la etiqueta Grid y vemos que por debajo están las etiquetas Points_p1, Points_p2 y Point_tie. Movemos estas tres etiquetas a justo debajo de MoveCounter.

```
; ------------------------------------------------------------------
; Desarrollo de la partida.
; ------------------------------------------------------------------
; Posiciones de las fichas en el tablero, 2 bytes por ficha Y, X
GridPos:       db POS1_LEFT, POS1_TOP ; 1
               db POS2_LEFT, POS1_TOP ; 2
               db POS3_LEFT, POS1_TOP ; 3
               db POS1_LEFT, POS2_TOP ; 4
               db POS2_LEFT, POS2_TOP ; 5
               db POS3_LEFT, POS2_TOP ; 6
               db POS1_LEFT, POS3_TOP ; 7
               db POS2_LEFT, POS3_TOP ; 8
               db POS3_LEFT, POS3_TOP ; 9

; Casillas del tablero. Un byte por casilla, del 1 al 9.
; Bit 0 a 1, casilla ocupada por jugador 1.
; Bit 4 a 1, casilla ocupada por jugador 2.
Grid:          db $00, $00, $00, $00, $00, $00, $00, $00, $00
MoveCounter:   db $00 ; Contador de movimientos
Points_p1:     db $00 ; Puntos jugador 1
Points_p2:     db $00 ; Puntos jugador 2
Points_tie:    db $00 ; Puntos tablas
PlayerMoves:   db $00 ; Jugador que mueve
```

En Main.asm, localizamos la etiqueta loop_reset, y en la línea de más abajo, LD BC, LENDATA-$01, sustituimos $01 por $04.

Compilamos, cargamos en el emulador y, ahora sí, se actualizan los marcadores.


Descarga el código fuente desde aquí

https://tinyurl.com/2mzuh3j5

# Paso 3: sonido

Vamos a implementar distintos sonidos, y los vamos a usar para temporizar haciendo pausas entre distintos eventos. Creamos el archivo Sound.asm y añadimos el include en Main.asm.

En Sound.asm vamos añadir la definición de los distintos sonidos y melodías.

```
; ----------------------------------------------------------------
; Sonidos.
;
; Los sonidos acaban en $00, byte que indica el final.
; ----------------------------------------------------------------
; Cuenta atrás
SoundCountDown:
db $03, $8c, $3a, $00

; Error
SoundError:
db $0d, $c6, $1e, $16, $13, $13, $00

; Perdida de movimiento
SoundLostMovement:
db $0d, $07, $10, $0b, $96, $12, $0a, $4d, $14
db $0b, $96, $12, $1a, $2c, $08, $00

; Siguiente jugador
SoundNextPlayer:
db $01, $9d, $7b, $00

; Movimiento Spectrum
SoundSpectrum:
db $06, $6e, $20, $06, $6e, $20, $05, $b7, $24
db $06, $6e, $20, $05, $13, $29, $05, $b7, $24
db $06, $6e, $20, $00

; Tablas
SoundTie:
db $0d, $07, $10, $0b, $96, $12, $06, $d4, $1e
db $0b, $96, $12, $07, $a6, $1b, $0b, $96, $12
db $08, $a5, $18, $0b, $96, $12, $09, $b4, $15
db $0b, $96, $12, $0a, $4d, $14, $0b, $96, $12
db $0d, $07, $10, $00

; Punto
SoundWinGame:
db $06, $6e, $20, $05, $13, $29, $04, $40, $30
db $06, $6e, $20, $05, $13, $29, $04, $40, $30
db $06, $6e, $20, $05, $13, $29, $04, $c7, $2b
db $05, $13, $29, $05, $b7, $24, $05, $13, $29
db $06, $6e, $20, $05, $13, $29, $04, $40, $30
db $06, $6e, $20, $05, $13, $29, $04, $40, $30
db $06, $6e, $20, $05, $13, $29, $04, $c7, $2b
db $05, $13, $29, $05, $b7, $24, $06, $6e, $20
db $00
```

Todos los sonidos tienen como último byte cero, que vamos a usar en la rutina que los reproduce para saber cuándo se acaban.

En estas definiciones cada nota está compuesta por tres bytes en lugar de cuatro, los dos primeros son la nota y el tercero el byte bajo de la duración, siendo todas fusas y/o semifusas.

Implementamos la rutina que reproduce los sonidos.

```
; ----------------------------------------------------------------
```

```
; Reproduce una melodía o sonido.
;
; Entrada: BC = Dirección de inicio de la melodía.
;
; Para ahorrar se hace byte a byte, siendo la duración siempre fusa o
; semifusa; sólo se necesite un byte para la duración.
; Esta rutina está adaptada para este funcionamiento.
; ---------------------------------------------------------------
PlayMusic:
push    af
push    bc
push    de
push    hl
push    ix                  ; Preserva registros

playMusic_loop:
ld      a, (bc)             ; A = byte alto nota
ld      h, a                ; H = A
or      a                   ; ¿A = 0?
jr      z, playMusic_end    ; A = 0, fin

inc     bc                  ; BC = siguiente valor
ld      a, (bc)             ; A = byte bajo nota
ld      l, a                ; L = A

inc     bc                  ; BC = siguiente valor
ld      a, (bc)             ; A = duración nota
ld      e, a                ; E = A
ld      d, $00              ; D = 0 (fusa o semifusa)
inc     bc                  ; BC = siguiente valor

push    bc                  ; Preserva BC
call    BEEPER              ; Reproduce nota
pop     bc                  ; Recupera BC

jr      playMusic_loop      ; Bucle hasta que llegue al 0
                            ; fin melodía

playMusic_end:
pop     ix
pop     hl
pop     de
pop     bc
pop     af                  ; Recupera registros
ret
```

PlayMusic recibe en BC la dirección del sonido a emitir, carga la nota en HL, la duración en DE, la emite y sigue en bucle hasta llegar al byte cero, que marca el fin del sonido. D siempre es cero, sólo se usa un byte para la duración.

En Main.asm vamos a incluir las llamadas necesarias para que se emitan los sonidos.

Localizamos la etiqueta Loop y borramos desde LD B, $19 hasta DJNZ loop_wait, ya que vamos a temporizar con el sonido. Más abajo, tras CALL PrintPoints, añadimos el sonido de siguiente jugador:

```
ld      bc, SoundNextPlayer
call    PlayMusic
```

Localizamos la etiqueta loop_print y justo por encima la línea JR Loop. Encima de esta línea añadimos el sonido de error:

```
ld      bc, SoundError
call    PlayMusic
```

Localizamos la etiqueta loop_win y justo por debajo la línea INC (HL). Debajo de esta línea añadimos el sonido de ganador de punto:

```
ld      bc, SoundWinGame
call    PlayMusic
```

Localizamos la etiqueta loop_reset y justo por encima añadimos el sonido de tablas.

```
ld      bc, SoundTie
call    PlayMusic
```

Compilamos, cargamos en el emulador y oímos cada uno de los sonidos.

El tamaño de Main.asm ha crecido considerablemente desde que empezamos, es hora de ver su aspecto una vez comentado.

```
org     $5e88                ; Dirección de carga

Main:
ld      hl, Sprite_P1        ; HL = dirección Sprite_P1
ld      (UDG), hl            ; UDG = dirección Sprite_p1
ld      a, $02               ; A = 2
call    OPENCHAN             ; Abre canal 2

xor     a                    ; A = 0, Z = 0, C = 0
call    BORDER               ; Cambia borde
call    CLA                  ; Cambia atributos pantalla
call    CLS                  ; Borra pantalla

Init:
call    PrintBoard           ; Pinta tablero

ld      hl, Name_p1          ; HL = nombre jugador 1
ld      de, player1_name     ; DE = nombre jugador 1
ld      bc, LENNAME          ; BC = longitud nombre
ldir                         ; Pasa datos
ld      hl, Name_p2          ; HL = nombre jugador 2
ld      de, player2_name     ; DE = nombre jugador 2
ld      bc, LENNAME          ; BC = longitud nombre
ldir                         ; Pasa datos
call    PrintInfo            ; Pinta info

ld      bc, LENDATA          ; BC = longitud datos partida
call    ResetValues          ; Limpia datos

Loop:
ld      hl, TitleTurn        ; HL = dirección turno
ld      de, TitleTurn_name   ; DE = dirección nombre en turno
call    DoMsg                ; Compone y pinta turno
call    PrintPoints          ; Pinta puntos
ld      bc, SoundNextPlayer  ; BC = dirección sonido
call    PlayMusic            ; Emite sonido
loop_key:
call    WaitKeyBoard         ; Espera pulsación tecla
ld      a, c                 ; A = C
cp      KEY0                 ; ¿Pulsada tecla?
jr      z, loop_key          ; No, bucle
call    ToMove               ; Comprueba movimiento
jr      z, loop_print        ; Correcto, salta
ld      hl, TitleError       ; HL = dirección error
call    PrintMsg             ; Pinta error
ld      bc, SoundError       ; BC = dirección sonido
call    PlayMusic            ; Emite sonido
jr      Loop                 ; Bucle principal
loop_print:
call    PrintOXO             ; Pinta ficha
loop_checkWinner:
call    CheckWinner          ; ¿Algún ganador?
jr      nz, loop_tie         ; No, comprueba tablas
call    PrintWinnerLine      ; Pinta línea ganadora
ld      hl, TitlePointFor    ; HL = dirección ganador
ld      de, TitlePointName   ; HL = dirección nombre ganador
call    DoMsg                ; Compone y pinta ganador
```

```
ld      hl, Points_p1          ; HL = dirección puntos jugador 1
ld      a, (PlayerMoves)       ; A = jugador que ha movido
or      a                      ; ¿Jugador 1?
jr      z, loop_win            ; Sí, salta
inc     hl                     ; HL = dirección puntos jugador 2
loop_win:
inc     (hl)                   ; Incrementa puntuación
ld      bc, SoundWinGame       ; BC = dirección sonido
call    PlayMusic              ; Emite sonido
jr      loop_reset             ; Salta
loop_tie:
ld      hl, MoveCounter        ; HL = dirección contador movimientos
inc     (hl)                   ; Incrementa contador
ld      a, $09                 ; A = 9
cp      (hl)                   ; ¿Contador = 9?
jr      nz, loop_cont          ; No, salta
ld      hl, Points_tie         ; HL = dirección puntos tablas
inc     (hl)                   ; Incrementa puntuación
ld      hl, TitleTie           ; HL = dirección tablas
call    PrintMsg               ; Pinta tablas
ld      bc, SoundTie           ; BC = dirección sonido
call    PlayMusic              ; Emite sonido
loop_reset:
ld      bc, LENDATA-$04        ; BC = longitud datos limpiar
call    ResetValues            ; Limpia datos
call    PrintBoard             ; Pinta tablero
loop_cont:
ld      a, (PlayerMoves)       ; A = jugador que ha movido
xor     $01                    ; A = jugador siguiente
ld      (PlayerMoves), a       ; Actualiza en memoria
jr      Loop                   ; Bucle principal

include "Game.asm"
include "Rom.asm"
include "Screen.asm"
include "Sprite.asm"
include "Sound.asm"
include "Var.asm"

end     Main
```

Tenemos desarrollado y funcional gran parte del juego. Podemos jugar nuestras primeras partidas con amigos y/o familiares, y tenemos sonido.

En el próximo capítulo implementaremos las distintas opciones de la partida, y el fin de la misma.

Descarga el código fuente desde aquí

https://tinyurl.com/2q4kkgtv

# Paso 4: opciones y fin de partida

En este capítulo vamos a implementar las distintas opciones de la partida y el fin de la misma; según lo dejamos en el capítulo anterior la partida continúa indefinidamente. Las opciones las seleccionaremos a través de un menú de inicio.

En Var.asm vamos a añadir las siguientes constantes para los colores, para así evitar poner números y que cambiar los colores en todo el juego sea más sencillo.

```
INK0:   equ $00
INK1:   equ $01
INK2:   equ $02
INK3:   equ $03
INK4:   equ $04
INK5:   equ $05
INK6:   equ $06
INK7:   equ $07
```

Como ejercicio, revisad las declaraciones de títulos y textos y donde veías $10, cambiad el número que le sigue por la etiqueta INK que le corresponde.

De igual manera, buscad en el resto de archivos las llamadas a INK, y en lugar de cargar en A el número de la tinta, cargad la etiqueta INK correspondiente.

## Menú

Parte de los textos del menú de inicio ya los tenemos definidos en Var.asm: la etiqueta TitleEspamatica y desde TitleOptionStart hasta TitleOptionTime.

Vamos a borrar todas estas líneas pues la vamos a implementar de nuevo, añadiendo colores y posiciones, quedando así:

```
Title3EnRaya:          db $10, INK2, $13, $01, $16, $02, $0a
                       defm "Tres en raya"
TitleOptionStart:      db $10, INK1, $13, $01, $16, $08, $08, "0. "
                       db $10, INK5
                       defm "Empezar"
TitleOptionPlayer:     db $10, INK7, $13, $01, $16, $0a, $08, "1. "
                       db $10, INK6
                       defm "Jugadores"
TitleOptionPoint:      db $10, INK7, $16, $0c, $08, "2. ", $10, INK6
                       defm "Puntos"
TitleOptionTime:       db $10, INK7, $16, $0e, $08, "3. ", $10, INK6
                       defm "Tiempo"
TitleEspamatica:       db $16, $14, $08
                       db $10, INK2, "E", $10, INK6, "s"
                       db $10, INK4, "p", $10, INK5, "a"
       db $10, INK2, "m", $10, INK6, "a"
                       db $10, INK4, "t", $10, INK5, "i"
                       db $10, INK2, "c", $10, INK6, "a"
                       db $10, INK7, " 2019", $ff
```

Ya tenemos la definición del menú de inicio casi lista para poder pintarlo, sólo nos falta una rutina para pintar los valores de las opciones. La implementamos en Screen.asm.

```
; ------------------------------------------------------------------
; Pinta los valores de las opciones.
;
; Altera el valor de los registros AF, BC y HL.
; ------------------------------------------------------------------
PrintOptions:
ld      a, INK4               ; A = tinta verde
call    INK                   ; Cambia tinta
ld      b, INI_TOP-$0a        ; B = coord Y
```

```
ld      c, INI_LEFT-$15       ; C = coord X
call    AT                    ; Posiciona cursor
ld      hl, MaxPlayers        ; HL = valor jugadores
call    PrintBCD              ; Lo pinta
ld      b, INI_TOP-$0c        ; B = coord Y
call    AT                    ; Posiciona cursor
ld      hl, MaxPoints         ; HL = valor puntos
call    PrintBCD              ; Lo pinta
ld      b, INI_TOP-$0e        ; B = coord Y
call    AT                    ; Posiciona cursor
ld      hl, MaxSeconds        ; HL = valor tiempo
jp      PrintBCD              ; Lo pinta y sale
```

PrintOptions pinta los valores de las opciones en verde. Posiciona el cursor y va pintando cada uno de los valores.

Vamos a ver qué tal queda el menú de inicio. Localizamos Main en Main.asm, y debajo de CALL OPENCHAN añadimos estas líneas:

```
Menu:
Di                            ; Desactiva interrupciones
im      $01                   ; Interrupciones = modo 1
ei                            ; Activa interrupciones
```

Hemos añadido una etiqueta para la parte del menú, desactivado las interrupciones para cambiar al modo uno y volverlas a activar. Luego usaremos las interrupciones en modo dos, y es por eso que estás líneas son necesarias aquí.

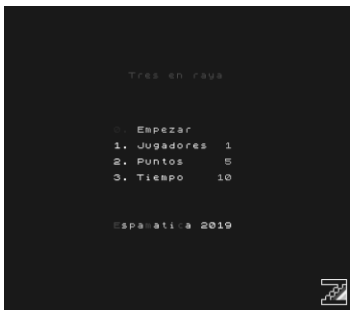
Más abajo localizamos CALL CLS y a continuación, añadimos las líneas siguientes:

```
ld      hl, Title3EnRaya      ; HL = dirección tres en raya
call    PrintString           ; Pinta el menú
menu_op:
call    PrintOptions          ; Pinta las opciones
jr      menu_op               ; Bucle menú
```

Pintamos la pantalla de inicio y los valores de las opciones. Nos quedamos en un bucle infinito, ya que ahí vamos a implementar la lógica del menú.

Con la definición del menú y una pocas líneas lo pintamos, pero falta hacerlo funcionar.

Compilamos, cargamos en el emulador y vemos como queda.




Antes de implementar la lógica del menú, en Game.asm vamos a implementar una rutina que utilice la ROM para leer el teclado y que nos devuelva el código ASCII de la última tecla pulsada. Esta rutina la vamos a usar para las opciones del menú y para la solicitud del nombre de los jugadores.

```
; ------------------------------------------------------------------
; Espera a que se pulse una tecla y devuelve su código Ascii.
;
; Salida: A -> Código Ascii de la tecla.
```

```
;
; Altera el valor de los registros AF y HL.
; ------------------------------------------------------------------
WaitKeyAlpha:
ld      hl, FLAGS_KEY           ; HL = dirección flag teclado
set     $03, (hl)               ; Entrada modo L

; Bucle hasta que se obtenga una tecla.
WaitKeyAlpha_loop:
bit     $05, (hl)               ; ¿Tecla pulsada?
jr      z, WaitKeyAlpha_loop    ; No pulsada, bucle
res     $05, (hl)               ; Bit a 0 para futuras inspecciones

; Obtiene el Ascii de la tecla pulsada
; Ascii válidos 12 ($0C), 13 ($0D) y de 32 ($20) a 127 ($7F)
; Si la tecla pulsada es Space, carga ' ' en A
WaitKeyAlpha_loadKey:
ld      hl, LAST_KEY            ; HL = dirección última tecla pulsada
ld      a, (hl)                 ; A = última tecla pulsada
cp      $80                     ; ¿Ascii > 127?
jr      nc, WaitKeyAlpha        ; Sí, tecla no válida, bucle
cp      KEYDEL                  ; ¿Pulsado Delete?
ret     z                       ; Sí, sale
cp      KEYENT                  ; ¿Pulsado Enter?
ret     z                       ; Sí, sale
cp      KEYSPC                  ; ¿Pulsado Espacio?
jr      c, WaitKeyAlpha         ; Ascii < espacio, inválida, bucle
ret                             ; Sale
```

WaitKeyAlpha espera hasta que se haya pulsado una tecla que sea válida para nosotros, esto es: delete, enter o un código ASCII entre treinta y dos y ciento veintisiete. Una vez se ha pulsado una tecla válida, devuelve el código ASCII de la misma en A.

Implementamos en Main.asm la rutina del menú, justo debajo de CALL PrintOptions, siendo el aspecto final este:

```
menu_op:
call    PrintOptions        ; Pinta las opciones
call    WaitKeyAlpha        ; Espera pulsación tecla
cp      KEY0                ; ¿Pulsado 0?
jr      z, Init             ; Sí, inicia partida
menu_Players:
cp      KEY1                ; ¿Pulsado 1?
jr      nz, menu_Points     ; No, salta
ld      a, (MaxPlayers)     ; A = jugadores
xor     $03                 ; Alterna entre 1 y 2
ld      (MaxPlayers), a     ; Actualiza en memoria
jr      menu_op             ; Bucle
menu_Points:
cp      KEY2                ; ¿Pulsado 2?
jr      nz, menu_Time       ; No, salta
ld      a, (MaxPoints)      ; A = puntos
inc     a                   ; A += 1
cp      $06                 ; ¿A = 6?
jr      nz, menu_PointsDo   ; No, salta
ld      a, $01              ; A = 1
menu_PointsDo:
ld      (MaxPoints), a      ; Actualiza en memoria
add     a, '0'              ; A = ascii de valor
ld      (info_points), a    ; Actualiza en memoria
cp      '1'                 ; ¿Puntos 1?
jr      z, menu_Points1     ; Sí, salta
ld      a, 's'              ; Plural
jr      menu_PointsEnd      ; Salta
menu_Points1:
ld      a, ' '              ; Singular
menu_PointsEnd:
ld      (info_gt1), a       ; Actualiza en memoria
jr      menu_op             ; Bucle
```

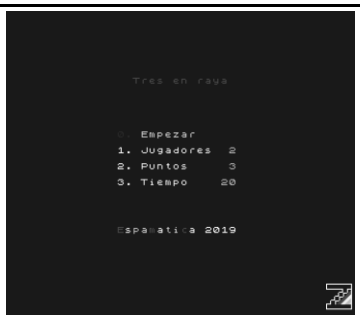
```
menu_Time:
cp      KEY3                ; ¿Pulsado 3?
jr      nz, menu_op         ; No, bucle
ld      a, (MaxSeconds)     ; A = segundos
add     a, $05              ; A += 5
daa                         ; Ajuste decimal
cp      $35                 ; ¿ A = 35 BCD?
jr      nz, menu_TimeDo     ; No, salta
ld      a, $05              ; A = 5
menu_TimeDo:
ld      (MaxSeconds), a     ; Actualiza en memoria
jr      menu_op             ; Bucle
```

Tras llamar a la rutina que comprueba si se ha pulsado alguna tecla válida, evalúa si es alguna entre la tecla cero y la tres, y según cual sea actúa de una manera u otra. Observad que sólo hace el ajuste decimal al incrementar los segundos; es la única opción cuyo valor pasa de nueve.

Compilamos, cargamos en el emulador y vemos los resultados. Todo parece ir bien hasta que pulsamos el cero, iniciamos la partida y quedan restos del menú.

Para solucionar esto, debajo de la etiqueta Init añadimos la línea siguiente:

```
call    CLS                 ; Limpia pantalla
```




## Inicio de partida

Al iniciar la partida, lo primero que vamos a hacer es solicitar los nombres de los jugadores. Antes de nada, añadimos otra constante en ROM.asm; en esta dirección están las coordenadas del cursor.

```
; Posición del cursor en pantalla 2.
; Si se carga en BC -> B = Y, C = X.
CURSOR:     equ $5c88
```

Implementamos la rutina encargada de solicitar los nombres de los jugadores en Game.asm. Aunque seguimos sin explicar instrucción a instrucción, debido al tamaño de la rutina la vamos a ir implementando por bloques.

```
GetPlayersName:
ld      hl, Name_p1
ld      de, Name_p1+$01
ld      bc, LENNAME*2-$01
ld      (hl), " "
ldir
```

Limpiamos los nombres de los jugadores.

```
ld      e, $01
getPlayersName_loop:
ld      a, INK4
call    INK
ld      b, INI_TOP - $0f
```

```
ld      c, INI_LEFT - $01
call    CLL
call    AT

ld      hl, TitlePlayerNumber
ld      (hl), "1"
ld      a, $01
cp      e
jr      z, getPlayersName_cont
ld      (hl), "2"
getPlayersName_cont:
ld      hl, TitlePlayerName
call    PrintString

ld      hl, Name_p1
ld      a, $01
cp      e
jr      z, getPlayersName_cont2
ld      hl, Name_p2
```

Hacemos un bucle de dos iteraciones máximo, una por jugador, cambiamos la tinta, borramos la línea en la que se solicitan los nombres, posicionamos el cursor, preparamos el título dependiendo del jugador, y apuntamos HL al nombre del jugador.

```
getPlayersName_cont2:
ld      d, $00
ld      a, INK3
call    INK
call    getPlayersName_getName

ld      a, (MaxPlayers)
cp      $02
jr      nz, getPlayersName_onlyOne
inc     e
cp      e
jr      z, getPlayersName_loop
ret
```

Usamos D para controlar la longitud del nombre, cambiamos la tinta y solicitamos el nombre del jugador. Obtenemos los jugadores, vemos si son dos, y si no es así ponemos el nombre por defecto al dos. Si son dos jugadores, comprobamos si el nombre introducido es el del jugador dos, y si no es así bucle para solicitarlo.

```
getPlayersName_onlyOne:
ld      hl, Name_p2Default
ld      de, Name_p2
ld      bc, LENNAME
ldir
ret
```

Si es un solo jugador, asigna al segundo el nombre por defecto.

```
getPlayersName_getName:
push    hl
call    WaitKeyAlpha
pop     hl

cp      KEYDEL
jr      z, getPlayersName_delete
cp      KEYENT
jr      z, getPlayersName_enter
push    de
ld      e, a
ld      a, LENNAME
cp      d
ld      a, e
pop     de
jr      z, getPlayersName_getName
```

```
ld      (hl), a
inc     hl
rst     $10
inc     d
jr      getPlayersName_getName
```

Esperamos a la pulsación de una tecla válida, comprobamos si es delete y de serlo saltamos a su manejo. Comprobamos si es enter y de serlo saltamos a su manejo.

Si no es delete, ni es enter, vemos si hemos llegado a la longitud máxima, y si no es así añadimos el carácter al nombre y lo pintamos.

```
getPlayersName_delete:
ld      a, $00
cp      d
jr      z, getPlayersName_getName

dec     d
dec     hl
ld      a, ' '
ld      (hl), a
ld      bc, (CURSOR)
inc     c
call    AT
rst     $10
call    AT
jr      getPlayersName_getName
```

Si la tecla pulsada es delete y la longitud del nombre no es cero, borramos el carácter anterior del nombre y de la pantalla.

```
getPlayersName_enter:
ld      a, 0
cp      d
jr      z, getPlayersName_getName
ret
```

Si la tecla pulsada es enter y la longitud del nombre no es cero, se finaliza la solicitud del nombre.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------
; Solicita el nombre de los jugadores.
;
; Altera el valor de los registros AF, BC, DE y HL.
; ------------------------------------------------------------------
GetPlayersName:
ld      hl, Name_p1                 ; HL = dirección nombre jugador 1
ld      de, Name_p1+$01             ; DE = HL+1
ld      bc, LENNAME*2-$01           ; BC = longitud nombres - 1
ld      (hl), " "                   ; Limpia primera posición
ldir                                ; Limpia el resto

ld      e, $01                     ; E = 1
getPlayersName_loop:
ld      a, INK4                     ; A = tinta 4
call    INK                         ; Cambia tinta
ld      b, INI_TOP - $0f            ; B = coord Y
ld      c, INI_LEFT - $01           ; X = coord X
call    CLL                         ; Borra la línea
call    AT                          ; Posiciona cursor

ld      hl, TitlePlayerNumber       ; HL = número de jugador
ld      (hl), "1"                   ; Jugador 1
ld      a, $01                      ; A = 1
cp      e                           ; ¿Jugador 1?
jr      z, getPlayersName_cont      ; Sí, salta
ld      (hl), "2"                   ; Jugador 2
```

```
getPlayersName_cont:
ld      hl, TitlePlayerName         ; HL = título nombre jugador
call    PrintString                 ; Lo pinta

ld      hl, Name_p1                 ; HL = dirección nombre jugador 1
ld      a, $01                      ; A = 1
cp      e                           ; ¿Jugador 1?
jr      z, getPlayersName_cont2     ; Sí, salta
ld      hl, Name_p2                 ; HL = dirección nombre jugador 2
getPlayersName_cont2:
ld      d, $00                      ; D = contador longitud nombre
ld      a, INK3                     ; A = tinta 3
call    INK                         ; Cambia tinta
call    getPlayersName_getName      ; Pide nombre jugador

ld      a, (MaxPlayers)             ; A = jugadores
cp      $02                         ; ¿Dos jugadores?
jr      nz, getPlayersName_onlyOne  ; Un jugador, nombre por defecto
inc     e                           ; E+=1
cp      e                           ; Compara con jugadores
jr      z, getPlayersName_loop      ; Iguales, salta
ret

; Un solo jugador
; Copia el nombre por defecto del jugador 2
getPlayersName_onlyOne:
ld      hl, Name_p2Default          ; HL = nombre por defecto jugador 2
ld      de, Name_p2                 ; DE = nombre jugador 2
ld      bc, LENNAME                 ; Longitud nombre
ldir                                ; Copia nombre por defecto
ret                                 ; Sale

; Pide el nombre del jugador
getPlayersName_getName:
push    hl                          ; Preserva HL
call    WaitKeyAlpha                ; Espera tecla válida
pop     hl                          ; Recupera HL

cp      KEYDEL                      ; ¿Delete?
jr      z, getPlayersName_delete    ; Sí, salta
cp      KEYENT                      ; ¿Enter?
jr      z, getPlayersName_enter     ; Sí, salta
push    de                          ; Preserva DE
ld      e, a                        ; E = código Ascii
ld      a, LENNAME                  ; A = longitud máxima nombre
cp      d                           ; ¿D = longitud máxima?
ld      a, e                        ; A = código Ascii
pop     de                          ; Recupera DE
jr      z, getPlayersName_getName   ; D = longitud máxima
                                    ; otro carácter
                                    ; Enter o Delete

ld      (hl), a                     ; Añade carácter a nombre
inc     hl                          ; HL = siguiente posición
rst     $10                         ; Imprime carácter
inc     d                           ; D+=1
jr      getPlayersName_getName      ; Solicitar otro carácter

getPlayersName_delete:
ld      a, $00                      ; A = 0
cp      d                           ; ¿Longitud 0?
jr      z, getPlayersName_getName   ; Sí, otro carácter

dec     d                           ; D-=1
dec     hl                          ; HL-=1, carácter anterior
ld      a, ' '                      ; A = espacio
ld      (hl), a                     ; Limpia carácter anterior
ld      bc, (CURSOR)                ; BC = posición cursor
inc     c                           ; BC = columna anterior para AT
call    AT                          ; Posiciona cursor
rst     $10                         ; Borra el carácter pantalla
```

```
call    AT                          ; Posiciona cursor
jr      getPlayersName_getName      ; Otro carácter

getPlayersName_enter:
ld      a, 0                        ; A = 0
cp      d                           ; ¿Longitud 0?
jr      z, getPlayersName_getName   ; Sí, otro carácter
ret                                 ; Fin nombre
```

Vamos a ver si todo funciona. En Main.asm localizamos Init, y debajo de CALL CLS añadimos la llamada a la solicitud de los nombres, y otra llamada a CLS.

```
call    GetPlayersName      ; Solicita nombres jugadores
call    CLS                 ; Limpia pantalla
```

Compilamos, cargamos en el emulador y vemos los resultados. Ya podemos proporcionar los nombres de los jugadores, y aparecen en la información de la partida. Si la partida es a un solo jugador, jugamos contra ZX Spectrum.


## Tiempo del turno

Unas de las opciones son los segundos de los que dispone cada jugador para realizar el movimiento, si pasados esos segundos no se ha realizado el movimiento, pierde el turno y pasa al otro jugador.

Vamos a hacer uso de las interrupciones para poder controlar el tiempo. Creamos el archivo Int.asm.

Al inicio de Main.asm, justo debajo de ORG $5E88, añadimos las líneas siguientes (ya lo hicimos en Batalla espacial):

```
; ------------------------------------------------------------------
; Indicadores
;       bit 0   ->  Reiniciar cuenta atrás
;       bit 1   ->  Pierde turno
;       bit 2   ->  Pintar cuenta atrás
;       bit 3   ->  Sonido de advertencia, acaba cuenta atrás
; ------------------------------------------------------------------
flags:      db $00
; Valor cuenta atrás
countdown:  db $00
; Segundos por turno
seconds:    db $00
```

Seconds la vamos a usar para que las interrupciones sepan el número de segundos por turno que han seleccionado los jugadores. Con seconds ya no es necesaria la variable MaxSeconds, la quitamos.

Localizamos Main, y tras la línea CALL OPENCHAN, inicializamos seconds:

```
ld      a, $10              ; A = $10 BCD
ld      (seconds), a        ; Actualiza segundos
```

```
ld      (countdown), a      ; Actualiza cuenta atrás
```

Localizamos menu_Time y tras JR NZ, menu_op modificamos la línea LD A, (MaxSeconds), quedando así:

```
ld      a, (seconds)        ; A = segundos
```

Localizamos menu_TimeDo, borramos LD, (MaxSecond), A y en su lugar añadimos las líneas siguientes:

```
ld      (seconds), a        ; Actualiza segundos
ld      (countdown), a      ; Actualiza cuenta atrás
```

En Screen.asm, al final de la rutina PrintOptions, reemplazamos LD HL, MaxSeconds por:

```
ld      hl, seconds         ; HL = valor tiempo
```

Por último, en Var.asm, borramos la definición de MaxSeconds.

La rutina de las interrupciones cambiará el valor de flags y el de countdown, debemos tenerlo en cuenta en el bucle principal.

Para que la rutina de interrupciones se ejecute cincuenta veces por segundo (en PAL, sesenta en NTSC) hay que activar el modo dos de las mismas. Localizamos Loop en Main.asm y encima añadimos:

```
di                          ; Desactiva interrupciones
ld      a, $28              ; A = $28
ld      i, a                ; I = A (interrupciones en $7e5c)
im      $02                 ; Interrupciones = modo 2
ei                          ; Actualiza interrupciones
```

Desactivamos las interrupciones, cargamos $28 en el registro de interrupciones, las ponemos en modo dos y las activamos.

Buscamos loop_key y justo debajo implementamos la lógica del manejo de flags.

```
ld      a, (flags)          ; A = Flags
bit     $02, a              ; ¿Bit 2 activo?
res     $02, a              ; Desactiva bit 2
jr      z, loop_warning     ; No activo, salta
push    af                  ; Preserva AF
call    PrintCountDown      ; Pinta cuenta atrás
pop     af                  ; Recupera AF
```

Cargamos flags en A y evaluamos si el bit dos está activo y lo desactivamos. Si no está activo saltamos, si lo está pintamos la cuenta atrás.

```
loop_warning:
bit     $03, a              ; ¿Bit 3 activo?
res     $03, a              ; Desactiva bit 3
jr      z, loop_lostMov     ; No activo, salta
ld      bc, SoundCountDown  ; BC = dirección sonido
call    PlayMusic           ; Emite sonido
```

Evaluamos si el bit tres está activo y lo desactivamos. Si está activo saltamos, si no lo está emitimos el sonido de advertencia.

```
loop_lostMov:
bit     $01, a              ; ¿Bit 1 activo?
res     $01, a              ; Desactiva bit 1
ld      (flags), a          ; Actualiza en memoria
halt                        ; Sincroniza con interrupciones
jr      z, loop_keyCont     ; No activo, salta
ld      hl, TitleLostMovement   ; HL = dirección mov perdido
```

```
ld      de, TitleLostMov_name   ; DE = dirección nombre
call    DoMsg               ; Pinta mensaje
ld      bc, SoundLostMovement   ; BC = dirección sonido
call    PlayMusic           ; Emite sonido
jr      loop_cont           ; Salta
loop_keyCont:
```

Evaluamos si el bit uno está activo, desactivamos, actualizamos flags y sincronizamos con las interrupciones. Si no está activo saltamos, Si lo está, pintamos el mensaje de movimiento perdido y emitimos el sonido. La etiqueta loop_keyCont queda encima de CALL WaitKeyBoard.

Por último, localizamos loop_cont, borramos JR Loop y añadimos en su lugar las líneas siguientes:

```
ld      a, $01              ; A = reiniciar cuenta atrás
ld      (flags), a          ; Actualiza flags
jp      Loop                ; Bucle principal
```

Actualizamos flags para que la rutina de interrupciones reinicie la cuenta atrás. También hemos sustituido JR por JP debido a las líneas que hemos añadido, que hacen que JR esté fuera de rango.

En Screen.asm vamos implementar la rutina que pinta la cuenta atrás.

```
; ------------------------------------------------------------------
; Pinta la cuenta atrás.
;
; Altera el valor de los registros AF, BC y HL.
; ------------------------------------------------------------------
PrintCountDown:
ld      a, INK3         ; A = tinta 3
call    INK             ; Cambia tinta
ld      b, INI_TOP-$0c  ; B = coord Y
ld      c, INI_LEFT     ; C = coord X
call    AT              ; Posiciona cursor
ld      hl, countdown   ; HL = dirección cuenta atrás
call    PrintBCD        ; Pinta cuenta atrás izquierda
ld      c, INI_LEFT-$1e ; C = coord X
call    AT              ; Posiciona cursor
jp      PrintBCD        ; Pinta cuenta atrás derecha y sale
```

La cuenta atrás la pintamos en dos posiciones, a la izquierda y la derecha del tablero. Ponemos la tinta en magenta, ponemos el cursor a la izquierda, pintamos la cuenta atrás, ponemos el cursor a la derecha y pintamos la cuenta atrás.

La rutina de manejo de las interrupciones la implementamos en Int.asm (no incluyáis este archivo en Main.asm). También vamos a ver esta implementación por bloques.

```
; ------------------------------------------------------------------
; Int.asm
;
; Manejo de interrupciones en modo 2
; ------------------------------------------------------------------
org     $7e5c

; ------------------------------------------------------------------
; Indicadores
;       bit 0   ->  Reiniciar cuenta atrás
;       bit 1   ->  Pierde turno
;       bit 2   ->  Pintar cuenta atrás
;       bit 3   ->  Sonido de advertencia, acaba cuenta atrás
; ------------------------------------------------------------------
FLAGS:      equ $5e88

COUNTDOWN:  equ FLAGS+$01       ; Valor cuenta atrás
```

```
SECONDS:    equ FLAGS+$02       ; Segundos por turno
```

La rutina de interrupciones carga en la dirección $7E5C. Después añadimos las constantes con las direcciones de memoria que usaremos para el intercambio de información entre Main.asm e Int.asm.

```
CountDownISR:
push    af
push    bc
push    de
push    hl
push    ix                      ; Preserva registros
```

El primer paso de la rutina es preservar el valor de los registros.

```
countDown_flags:
ld      a, (FLAGS)              ; A = flags
and     $01                     ; ¿Reiniciar cuenta atrás?
jr      z, countDown_cont       ; No, salta
ld      a, (SECONDS)            ; A = SECONDS
ld      (COUNTDOWN), a          ; Actualiza en memoria
ld      a, $04                  ; A = pintar cuenta atrás
ld      (FLAGS), a              ; Actualiza en memoria
jr      countDownISR_end        ; Fin
```

Comprobamos si desde Main.asm se indica que se debe reiniciar la cuenta atrás (cambio de turno) y salta si no es así. Si hay cambio de turno, reinicia la cuenta atrás al valor especificado en el menú, indica en flags que se debe pintar la cuenta atrás y salta para salir de la rutina.

```
countDown_cont:
ld      hl, countDownTicks      ; HL = contador ticks
inc     (hl)                    ; Contador ticks+=1
ld      a, $32                  ; A = 50
cp      (hl)                    ; ¿Contador ticks = 50?
jr      nz, countDownISR_end    ; No, salta.
xor     a                       ; A = 0, Z = 1, Carry = 0
ld      (hl), a                 ; Contador ticks = 0
```

En countDownTicks vamos sumando uno en cada interrupción, y cuando llegamos a cincuenta es señal de que ha pasado un segundo, lo que comprobamos en este bloque. Si no se ha llegado a cincuenta salta para salir de la rutina, si ha llegado ponemos el contador de ticks a cero y seguimos con la rutina.

```
; Ha llegado a 50, ha pasado un segundo
ld      a, (COUNTDOWN)          ; A = valor cuenta atrás
dec     a                       ; A-=1
daa                             ; Ajuste decimal
ld      (COUNTDOWN), a          ; Actualiza en memoria
ld      b, $04                  ; B = pintar cuenta atrás / 4 seg
cp      b                       ; ¿Menos de 4 segundos?
jr      nc, countDownISR_reset  ; A >= 4, salta
set     $03, b                  ; Sonido de advertencia
or      a                       ; ¿A = 0?
jr      nz, countDownISR_reset  ; No, salta
set     $01, b                  ; Pierde turno
```

En el caso de que haya pasado un segundo, calculamos que tipo de información hay que pasar a Main.asm. Restamos un segundo de la cuenta atrás, vemos si está por debajo de cuatro (también comunica a Main.asm que hay que pintar la cuenta atrás), y saltamos si no lo está. Si lo está, activamos el bit para el sonido de la advertencia, evaluamos si la cuenta atrás ha llegado a cero, y saltamos si no lo ha hecho. Activa el bit de perdida de turno si ha llegado a cero.

```
countDownISR_reset:
```

```
ld      a, b                    ; A = B
ld      (FLAGS), a              ; Actualiza en memoria
```

Actualiza el valor de flags con la información que hay que pasar a Main.asm.

```
countDownISR_end:
pop     ix
pop     hl
pop     de
pop     bc
pop     af                      ; Recupera registros


ei                              ; Reactiva interrupciones
reti                            ; Sale

countDownTicks: db $00          ; Ticks (50*seg)
```

Recuperamos el valor de los registros, activamos interrupciones y salimos. Por último, declaramos el contador de ticks.

Recordad que es necesario compilar ahora dos .tap por separado y hacer nuestro propio cargador.

El cargador Basic es el siguiente:

```
10 CLEAR 24200
20 LOAD ""CODE 24200
30 LOAD ""CODE 32348
40 RANDOMIZE USR 24200
```

El script para compilar en Windows queda así:

```
echo off
cls
echo Compilando oxo
pasmo --name TresEnRaya --tap Main.asm oxo.tap oxo.log
echo Compilando int
pasmo --name Int --tap Int.asm int.tap int.log
echo Generando Tres en raya
copy /y /b cargador.tap+oxo.tap+int.tap TresEnRaya.tap
echo Proceso finalizado
```

La versión en Linux queda así:

```
clear
echo Compilando oxo
pasmo --name TresEnRaya --tap Main.asm oxo.tap oxo.log
echo Compilando int
pasmo --name Int --tap Int.asm int.tap int.log
echo Generando Tres en raya
cat cargador.tap oxo.tap int.tap > TresEnRaya.tap
echo Proceso finalizado
```

Compilamos, cargamos en el emulador y comprobamos que se pinta la cuenta atrás, suena una alarma al bajar de cuatro segundos y pierde el turno al llegar a cero. También veremos el mensaje de perdida de turno.


## Fin de partida

Para finalizar el capítulo vamos a implementar el fin de partida, que se da cuando uno de los jugadores llega a los puntos definidos en el menú, o cuando se llega al máximo tablas.

Al acabar la partida se pregunta si queremos otra; añadimos las teclas de la respuesta como constantes en Var.asm. También añadimos el número máximo de tablas.

```
KEYN:           equ $4e ; Tecla N
KEYn:           equ $6e ; Tecla n
KEYS:           equ $53 ; Tecla S
KEYs:           equ $73 ; Tecla s
MAXTIES:        equ $05 ; Número máximo tablas
```

Dependiendo de si se opta o no por jugar otra partida, se saltará a un lugar u otro de Main.asm. Localizamos la etiqueta Init y después de CALL GetPlayersName añadimos la siguiente etiqueta:

```
Start:
```

Localizamos la etiqueta loop_reset y justo debajo de ella vamos a implementar la verificación de si alguien gana la partida.

```
call    PrintPoints         ; Pinta los puntos
ld      a, (MaxPoints)      ; A = máximo puntos
ld      b, a                ; B = A
ld      a, (Points_p1)      ; A = puntos jugador 1
cp      b                   ; ¿Jugador 1 gana?
jr      z, EndPlay          ; Sí, salta
ld      a, (Points_p2)      ; A = puntos jugador 2
cp      b                   ; ¿Jugador 2 gana?
jr      z, EndPlay          ; Sí, salta
ld      b, MAXTIES          ; B = máximo tablas
ld      a, (Points_tie)     ; A = puntos tablas
cp      b                   ; ¿A = B?
jr      z, EndPlay          ; Sí, salta
```

Pintamos los puntos y comparamos los puntos de los jugadores con el máximo de puntos definidos. Si alguno de los jugadores tiene los puntos necesarios, gana la partida.

Al final de Main.asm, antes de los include, añadimos la rutina del fin partida.

```
EndPlay:
di                          ; Desactiva interrupciones
im      $01                 ; Interrupciones en modo 1
ei                          ; Activa interrupciones
ld      hl, TitleGameOver   ; HL = título game over
call    PrintMsg            ; Pinta el mensaje
endPlay_waitKey:
call    WaitKeyAlpha        ; Espera tecla
cp      KEYN                ; ¿Pulsada N?
jp      z, Menu             ; Sí, menú
cp      KEYn                ; ¿Pulsada n?
jp      z, Menu             ; Sí, menú
```

```
cp      KEYS                ; ¿Pulsada S?
jp      z, Start            ; Sí, inicio
cp      KEYs                ; ¿Pulsada s?
jp      z, Start            ; Sí, inicio
jr      endPlay_waitKey     ; No pulsadas, bucle
```

Desactivamos interrupciones, pasamos a modo uno, activamos interrupciones, esperamos a que se responda a la pregunta, y según la respuesta saltamos a un sitio u otro.

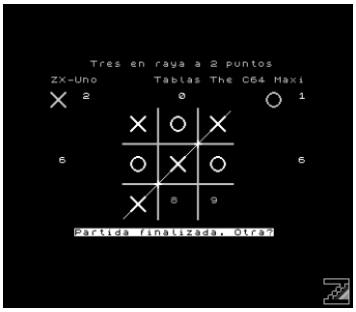
- Menu: volvemos al menú principal. Podemos volver a seleccionar las distintas opciones e introducir nuevos nombres de jugadores.
- Start: es la etiqueta que hemos añadido. Limpia la pantalla, la pinta, reinicia los datos de la partida y pasa las interrupciones a modo dos.
  Si no se ha pulsado ninguna de las teclas esperadas, bucle hasta que se pulse alguna.

Si vais a la etiqueta Menu veréis que las tres primeras líneas son para pasar a modo uno de interrupciones, cosa que también hacemos en EndPlay. Borrad las tres líneas que están debajo de la etiqueta Menu.

Compilamos, cargamos en el emulador y vemos los resultados. Todo parece funcionar correctamente, pero al finalizar la partida la puntuación se muestra parpadeando. Esto lo arrastramos desde el principio, aunque no nos hayamos percatado hasta ahora.

Para cambiar el comportamiento, localizamos PrintPoints dentro de Screen.asm, y justo debajo añadimos las líneas necesarias para desactivar el parpadeo.

```
ld      a, (ATTR_T)     ; A = atributos temporales
and     $7f             ; Quita parpadeo
ld      (ATTR_T), a     ; Actualiza en memoria
```




Llegados aquí, podemos jugar las primeras partidas completas a dos jugadores.

¿Queréis jugar contra el Spectrum y ganarle? Lo implementamos en el siguiente capítulo.

Descarga el código fuente desde aquí

https://tinyurl.com/2n6yjst8

# Paso 5: yo contra el Spectrum

Una vez que podemos jugar contra amigos y familiares, queda la posibilidad de que no tengamos con quién, de ahí la necesidad de que se pueda jugar contra el Spectrum.

El grueso de esta implementación lo escribimos en Game.asm, empezando por una rutina que genera números semialeatorios entre uno y nueve, para que cuando el Spectrum empieza la partida, lo haga de distinta manera.

```
; ------------------------------------------------------------------
; Obtiene un número semialeatorio entre 1 y 9.
;
; Retorno:  A   ->  Número obtenido.
; ------------------------------------------------------------------
GetRandomN:
ld      a, r                ; A = R
and     $0f                 ; Deja bits 0 a 3
inc     a                   ; A+=1
cp      $0a                 ; ¿A > 9?
ret     c                   ; No, sale
rra                         ; A/=2, porque Carry = 0, si no SRL A
ret                         ; Sale
```

GetRandomN obtiene un número semialeatorio entre el uno y el nueve, apoyándose en el registro R. En la línea RRA, tal y como se ve en los comentarios, dividimos A entre dos porque el acarreo está a cero y porque solo lo hacemos una vez, si no fuera así deberíamos hacerlo con SRL A. RRA ocupa un byte y tarda cuatro ciclos de reloj, SRL A es justo el doble. Sólo se divide entre dos si el número obtenido es mayor de 9.

La parte que realiza los movimientos del Spectrum es muy larga, ya que son diversas las combinaciones que se evalúan, por lo que vamos a verla por bloques. Aun así, y dado que las semejanzas entre distintos bloques son muchas, vamos a explicar sólo los más significativos.

```
; ------------------------------------------------------------------
; Mueve ZX Spectrum.
;
; Altera el valor de los registros AF, BC e IX.
; ------------------------------------------------------------------
ZxMove:
ld      a, (MoveCounter)    ; A = movimientos
or      a                   ; ¿Movimientos = 0?
jr      nz, zxMove_center   ; No, salta
call    GetRandomN          ; A = número entre 1 y 9
add     a, '0'              ; A = código ascii
ld      c, a                ; C = A
call    ToMove              ; Mueve a celda
ret                         ; Sale
zxMove_center:
cp      $01                 ; ¿Movimientos > 1?
jr      nz, zxMove_cont     ; Sí, salta
ld      c, KEY5             ; C = tecla 5
call    ToMove              ; Mueve a celda 5
ret     z                   ; Si es correcto, sale
```

En esta primera parte evaluamos si es el primer movimiento de la partida, y si es así obtenemos un número aleatorio entre uno y nueve y movemos a esa celda. Si es el segundo movimiento intentamos mover a la celda cinco (centro). Si no se dan estas condiciones, seguimos con las comprobaciones de si el Spectrum puede ganar o perder.

```
zxMove_cont:
ld      ix, Grid-$01        ; IX = dir Grid-1
ld      b, $20              ; B = valor Spectrum puede ganar
```

```
call    zxMoveToWin_123     ; Movimiento para ganar Spectrum
ret     z                   ; Si es válido, sale
ld      b, $02              ; B = valor jugador 1 puede ganar
call    zxMoveToWin_123     ; Movimiento para evitarlo
ret     z                   ; Si es válido sale
jp      zxMoveDefence_diagonally ; Movimientos defensivos
```

Recordad que los movimientos del jugador dos, en este caso el Spectrum, se señalan en el bit cuatro de las celdas, por eso cargamos $20 en B, que sería el valor si hubiera dos celdas ocupadas en una línea por el Spectrum.

Llamamos a zxMoveToWin_123 para que el Spectrum, si puede ganar, realice el movimiento.

Si no ha ganado el Spectrum, cargamos dos en B para evaluar si el jugador uno tiene movimiento para ganar. Llamamos por segunda vez a zxMoveToWin_123 para evitar que gane.

Si no se ha dado ninguno de los casos, pasamos a la defensiva.

```
; ------------------------------------------------------------------
; Evalúa si el Spectrum tiene movimiento para ganar.
; ------------------------------------------------------------------
zxMoveToWin_123:
ld      a, (ix+$01)         ; A = valor celda 1
add     a, (ix+$02)         ; A+= valor celda 2
add     a, (ix+$03)         ; A+= valor celda 3
cp      b                   ; ¿A = B?
jp      nz, zxMoveToWin_456 ; No, salta
; Spectrum puede ganar
ld      c, KEY1             ; C = tecla 1
call    ToMove              ; Mueve a celda 1
ret     z                   ; Si es correcto, sale
inc     c                   ; C = tecla 2
call    ToMove              ; Mueve a celda 2
ret     z                   ; Si es correcto, sale
inc     c                   ; C = tecla 3
call    ToMove              ; Mueve a celda 3
ret                         ; Sale
```

Si el Spectrum ocupa dos casillas, intentamos mover a la casilla uno, si no es correcto a la dos y si no a la tres. Para ir de una casilla a la siguiente incrementamos C, numéricamente son contiguas.

Las comprobaciones para las combinaciones cuatro, cinco, seis y siete, ocho, nueve son iguales a la anterior.

```
zxMoveToWin_456:
ld      a, (ix+$04)         ; A = valor celda 4
add     a, (ix+$05)         ; A+= valor celda 5
add     a, (ix+$06)         ; A+= valor celda 6
cp      b                   ; ¿A = B?
jr      nz, zxMoveToWin_789 ; No, salta
; Spectrum puede ganar
ld      c, KEY4             ; C = tecla 4
call    ToMove              ; Mueve a celda 4
ret     z                   ; Si es correcto, sale
inc     c                   ; C = tecla 5
call    ToMove              ; Mueve a celda 5
ret     z                   ; Si es correcto, sale
inc     c                   ; C = tecla 6
call    ToMove              ; Mueve a celda 6
ret                         ; Sale

zxMoveToWin_789:
ld      a, (ix+$07)         ; A = valor celda 7
add     a, (ix+$08)         ; A+= valor celda 8
add     a, (ix+$09)         ; A+= valor celda 9
```

```
cp      b                   ; ¿A = B?
jr      nz, zxMoveToWin_147 ; No, salta
; Spectrum puede ganar
ld      c, KEY7             ; C = tecla 7
call    ToMove              ; Mueve a celda 7
ret     z                   ; Si es correcto, sale
inc     c                   ; C = tecla 8
call    ToMove              ; Mueve a celda 8
ret     z                   ; Si es correcto, sale
inc     c                   ; C = tecla 9
call    ToMove              ; Mueve a celda 9
ret                         ; Sale
```

El resto de comprobaciones cambian ligeramente.

```
zxMoveToWin_147:
ld      a, (ix+$01)         ; A = valor celda 1
add     a, (ix+$04)         ; A+= valor celda 4
add     a, (ix+$07)         ; A+= valor celda 7
cp      b                   ; ¿A=B?
jr      nz, zxMoveToWin_258 ; No, salta
ld      c, KEY1             ; C = tecla 1
call    ToMove              ; Mueve a celda 1
ret     z                   ; Si es correcto, sale
ld      c, KEY4             ; C = tecla 4
call    ToMove              ; Mueve a celda 4
ret     z                   ; Si es correcto, sale
ld      c, KEY7             ; C = tecla 7
call    ToMove              ; Mueve a celda 7
ret                         ; Sale
```

Al no estar las casillas contiguas numéricamente, en la parte del final en lugar de incrementar C cargamos cada tecla. Desde aquí, todas las comprobaciones para saber si el Spectrum puede ganar o perder, son iguales.

```
zxMoveToWin_258:
ld      a, (ix+$02)         ; A = valor celda 2
add     a, (ix+$05)         ; A+= valor celda 5
add     a, (ix+$08)         ; A+= valor celda 8
cp      b                   ; ¿A=B?
jr      nz, zxMoveToWin_369 ; No, salta
ld      c, KEY2             ; C = tecla 2
call    ToMove              ; Mueve a celda 2
ret     z                   ; Si es correcto, sale
ld      c, KEY5             ; C = tecla 5
call    ToMove              ; Mueve a celda 5
ret     z                   ; Si es correcto, sale
ld      c, KEY8             ; C = tecla 8
call    ToMove              ; Mueve a celda 8
ret                         ; Sale

zxMoveToWin_369:
ld      a, (ix+$03)         ; A = valor celda 3
add     a, (ix+$06)         ; A+= valor celda 6
add     a, (ix+$09)         ; A+= valor celda 9
cp      b                   ; ¿A=B?
jr      nz, zxMoveToWin_159 ; No, salta
ld      c, KEY3             ; C = tecla 3
call    ToMove              ; Mueve a celda 3
ret     z                   ; Si es correcto, sale
ld      c, KEY6             ; C = tecla 6
call    ToMove              ; Mueve a celda 6
ret     z                   ; Si es correcto, sale
ld      c, KEY9             ; C = tecla 9
call    ToMove              ; Mueve a celda 9
ret                         ; Sale

zxMoveToWin_159:
ld      a, (ix+$01)         ; A = valor celda 1
add     a, (ix+$05)         ; A+= valor celda 5
```

```
add     a, (ix+$09)         ; A+= valor celda 9
cp      b                   ; ¿A=B?
jr      nz, zxMoveToWin_357 ; No, salta
ld      c, KEY1             ; C = tecla 1
call    ToMove              ; Mueve a celda 1
ret     z                   ; Si es correcto, sale
ld      c, KEY5             ; C = tecla 5
call    ToMove              ; Mueve a celda 5
ret     z                   ; Si es correcto, sale
ld      c, KEY9             ; C = tecla 9
call    ToMove              ; Mueve a celda 9
ret                         ; Sale

zxMoveToWin_357:
ld      a, (ix+$03)         ; A = valor celda 3
add     a, (ix+$05)         ; A+= valor celda 5
add     a, (ix+$07)         ; A+= valor celda 7
cp      b                   ; ¿A=B?
ret     nz
ld      c, KEY3             ; C = tecla 3
call    ToMove              ; Mueve a celda 3
ret     z                   ; Si es correcto, sale
ld      c, KEY5             ; C = tecla 5
call    ToMove              ; Mueve a celda 5
ret     z                   ; Si es correcto, sale
ld      c, KEY7             ; C = tecla 7
call    ToMove              ; Mueve a celda 7
ret                         ; Sale
```

El conjunto de rutinas zxMoveToWin nos sirve tanto para saber si el Spectrum puede ganar y así realizar el movimiento para ello, como para saber si el Spectrum puede perder y mover para evitarlo.

Si tanto el jugador uno como el Spectrum no tienen posibilidad de ganar, nos intentamos anticipar a la jugada del jugador uno. Primero comprobamos si el jugador hace una jugada diagonal ocupando los dos vértices, y si es así realizamos un bloqueo en cruz.

```
; ------------------------------------------------------------------
; Movimiento defensivo diagonal.
; ------------------------------------------------------------------
zxMoveDefence_diagonally:
; Comprueba si jugador 1 tiene fichas en diagonal
ld      a, (ix+$01)         ; A = valor celda 1
add     a, (ix+$09)         ; A+= valor celda 9
cp      b                   ; ¿A = B?
jr      z, zxMoveDefence_crossBlock   ; Sí, mov diagonal, salta
ld      a, (ix+$03)         ; A = valor celda 3
add     a, (ix+$07)         ; A+= valor celda 7
cp      b                   ; ¿A = B?
jr      nz, zxMoveDefence_crossBlock1 ; No, salta

zxMoveDefence_crossBlock:
; Bloqueo en cruz
ld      c, KEY4             ; C = tecla 4
call    ToMove              ; Mueve a celda 4
ret     z                   ; Si es correcto, sale
ld      c, KEY6             ; C = tecla 6
call    ToMove              ; Mueve a celda 6
ret     z                   ; Si es correcto, sale
ld      c, KEY2             ; C = tecla 2
call    ToMove              ; Mueve a celda 2
ret     z                   ; Si es correcto, sale
ld      c, KEY8             ; C = tecla 8
call    ToMove              ; Mueve a celda 8
ret     z                   ; Si es correcto, sale
```

Luego comprobamos si ha intentado una jugada diagonal con el centro ocupado, y si es así, dependiendo de que vértice tenga ocupado, bloqueamos la vertical.

```
; ------------------------------------------------------------------
; Movimiento defensivo de diagonal con el centro ocupado.
; ------------------------------------------------------------------
zxMoveDefence_crossBlock1:
ld      a, (ix+$05)         ; A = valor celda 5
and     $0f                 ; A = valor jugador 1
jr      z, zxMoveDefence_cornerBlock16  ; Z = no ocupada, salta
ld      a, (ix+$01)         ; A = valor celda 1
and     $0f                 ; A = valor jugador 1
jr      z, zxMoveDefence_crossBlock3    ; Z = no ocupada, salta
ld      c, KEY7             ; C = tecla 7
call    ToMove              ; Mueve a celda 7
ret     z                   ; Si es correcto, sale

zxMoveDefence_crossBlock3:
ld      a, (ix+$03)         ; A = valor celda 3
and     $0f                 ; A = valor jugador 1
jr      z, zxMoveDefence_crossBlock7    ; Z = no ocupada, salta
ld      c, KEY9             ; C = tecla 9
call    ToMove              ; Mueve a celda 9
ret     z                   ; Si es correcto, sale

zxMoveDefence_crossBlock7:
ld      a, (ix+$07)         ; A = valor celda 7
and     $0f                 ; A = valor jugador 1
jr      z, zxMoveDefence_crossBlock9    ; Z = no ocupada, salta
ld      c, KEY1             ; C = tecla 1
call    ToMove              ; Mueve a celda 1
ret     z                   ; Si es correcto, sale

zxMoveDefence_crossBlock9:
ld      a, (ix+$09)         ; A = valor celda 9
and     $0f                 ; A = valor jugador 1
jr      z, zxMoveDefence_cornerBlock16  ; Z = no ocupada, salta
ld      c, KEY3             ; C = tecla 3
call    ToMove              ; Mueve a celda 3
ret     z                   ; Si es correcto, sale
```

Si el jugador uno no intentó un movimiento diagonal, vemos si lo intentó en cruz, y procuramos bloquearlo.

```
; ------------------------------------------------------------------
; Movimiento defensivo en cruz.
; ------------------------------------------------------------------
zxMoveDefence_cornerBlock16:
ld      a, (ix+$01)         ; A = valor celda 1
add     a, (ix+$06)         ; A+= valor celda 6
cp      b                   ; ¿A = B?
jr      nz, zxMoveDefence_cornerBlock34 ; No, no mov cruz, salta
ld      c, KEY3             ; C = tecla 3
call    ToMove              ; Mueve a celda 3
ret     z                   ; Si es correcto, sale

zxMoveDefence_cornerBlock34:
ld      a, (ix+$03)         ; A = valor celda 3
add     a, (ix+$04)         ; A+= valor celda 4
cp      b                   ; ¿A = B?
jr      nz, zxMoveDefence_cornerBlock67 ; No, no mov cruz, salta
ld      c, KEY1             ; C = tecla 1
call    ToMove              ; Mueve a celda 1
ret     z                   ; Si es correcto, sale

zxMoveDefence_cornerBlock67:
ld      a, (ix+$06)         ; A = valor celda 6
add     a, (ix+$07)         ; A+= valor celda 7
cp      b                   ; ¿A = B?
jr      nz, zxMoveDefence_cornerBlock49 ; No, no mov cruz, salta
ld      c, KEY9             ; C = tecla 9
call    ToMove              ; Mueve a celda 9
ret     z                   ; Si es correcto, sale
```

```
zxMoveDefence_cornerBlock49:
ld      a, (ix+$04)         ; A = valor celda 4
add     a, (ix+$09)         ; A+= valor celda 9
cp      b                   ; ¿A = B?
jr      nz, zxMoveDefence_cornerBlock1827 ; No, no mov cruz, salta
ld      c, KEY7             ; C = tecla 7
call    ToMove              ; Mueve a celda 7
ret     z                   ; Si es correcto, sale

zxMoveDefence_cornerBlock1827:
ld      a, (ix+$01)         ; A = valor celda 1
add     a, (ix+$08)         ; A+= valor celda 8
cp      b                   ; ¿A = B?
jr      z, zxMoveDefence_cornerBlock1827Cont ; Sí, mov cruz, bloquear
ld      a, (ix+$02)         ; A = valor celda 2
add     a, (ix+$07)         ; A+= Valor celda 7
cp      b                   ; ¿ A= B?
jr      nz, zxMoveDefence_cornerBlock2938 ; No, no mov cruz, salta

zxMoveDefence_cornerBlock1827Cont:
ld      c, KEY4             ; C = tecla 4
call    ToMove              ; Mueve a celda 4
ret     z                   ; Si es correcto, sale

zxMoveDefence_cornerBlock2938:
ld      a, (ix+$02)         ; A = valor celda 2
add     a, (ix+$09)         ; A+= valor celda 9
cp      b                   ; ¿A = B?
jr      z, zxMoveDefence_cornerBlock2938Cont ; Sí, mov cruz, bloquear
ld      a, (ix+$03)         ; A = valor celda 3
add     a, (ix+$08)         ; A+= valor celda 8
cp      b                   ; ¿A = B?
jr      nz, zxMoveAttack_123 ; No, no mov cruz, salta

zxMoveDefence_cornerBlock2938Cont:
ld      c, KEY6             ; C = tecla 6
call    ToMove              ; Mueve a celda 6
ret     z                   ; Si es correcto, sale
```

Si llegamos hasta aquí, hemos comprobado que el Spectrum no tiene movimiento para ganar, que el jugador uno tampoco, que no corre peligro por movimientos de ataque del jugador uno en diagonal o en cruz; es hora de que el Spectrum pase al ataque.

Esta última parte sólo se ejecutará en los primeros movimientos, según se vaya llenando el tablero estaremos más ocupados en defensa que en ataque.

```
; ------------------------------------------------------------------
; Movimiento ofensivo horizontal, vertical y diagonal.
; ------------------------------------------------------------------
zxMoveAttack_123:
ld      b, $20              ; B = valor Spectrum con dos casillas
ld      a, (ix+$01)         ; A = valor celda 1
add     a, (ix+$02)         ; A+= valor celda 2
add     a, (ix+$03)         ; A+= valor celda 3
ld      c, a                ; Preserva A en C
and     $03                 ; A = casillas jugador 1
jr      nz, zxMoveAttack_456 ; ¿Alguna? Sí, salta
ld      a, c                ; Recupera A
and     $30                 ; A = casillas Spectrum
cp      b                   ; ¿Dos ocupadas?
jr      z, zxMoveAttack_456 ; Sí, salta
ld      c, KEY1             ; C = tecla 1
call    ToMove              ; Mueve a celda 1
ret     z                   ; Si es correcto, sale
inc     c                   ; C = tecla 2
call    ToMove              ; Mueve a celda 2
ret     z                   ; Si es correcto, sale
inc     c                   ; C = tecla 3
call    ToMove              ; Mueve a celda 3
```

```
ret     z                   ; Si es correcto, sale
```

Para el ataque buscamos combinaciones en las que el jugador uno no tenga ninguna celda ocupada, y el Spectrum solo una.

El resto de comprobaciones son muy parecidas a la anterior.

```
zxMoveAttack_456:
ld      a, (ix+$04)         ; A = valor celda 4
add     a, (ix+$05)         ; A+= valor celda 5
add     a, (ix+$06)         ; A+= valor celda 6
ld      c, a                ; Preserva A en C
and     $03                 ; A = casillas jugador 1
jr      nz, zxMoveAttack_789 ; ¿Alguna? Sí, salta
ld      a, c                ; Recupera A
and     $30                 ; A = casillas Spectrum
cp      b                   ; ¿Dos ocupadas?
jr      z, zxMoveAttack_789  ; Sí, salta
ld      c, KEY4             ; C = tecla 4
call    ToMove              ; Mueve a celda 4
ret     z                   ; Si es correcto, sale
inc     c                   ; C = tecla 5
call    ToMove              ; Mueve a celda 5
ret     z                   ; Si es correcto, sale
inc     c                   ; C = tecla 6
call    ToMove              ; Mueve a celda 6
ret     z                   ; Si es correcto, sale

zxMoveAttack_789:
ld      a, (ix+$07)         ; A = valor celda 7
add     a, (ix+$08)         ; A+= valor celda 8
add     a, (ix+$09)         ; A+= valor celda 9
ld      c, a                ; Preserva A en C
and     $03                 ; A = casillas jugador 1
jr      nz, zxMoveAttack_147 ; ¿Alguna? Sí, salta
ld      a, c                ; Recupera A
and     $30                 ; A = casillas Spectrum
cp      b                   ; ¿Dos ocupadas?
jr      z, zxMoveAttack_147  ; Sí, salta
ld      c, KEY7             ; C = tecla 7
call    ToMove              ; Mueve a celda 7
ret     z                   ; Si es correcto, sale
inc     c                   ; C = tecla 8
call    ToMove              ; Mueve a celda 8
ret     z                   ; Si es correcto, sale
inc     c                   ; C = tecla 9
call    ToMove              ; Mueve a celda 9
ret     z                   ; Si es correcto, sale

zxMoveAttack_147:
ld      a, (ix+$01)         ; A = valor celda 1
add     a, (ix+$04)         ; A+= valor celda 4
add     a, (ix+$07)         ; A+= valor celda 7
ld      c, a                ; Preserva A en C
and     $03                 ; A = casillas jugador 1
jr      nz, zxMoveAttack_258 ; ¿Alguna? Sí, salta
ld      a, c                ; Recupera A
and     $30                 ; A = casillas Spectrum
cp      b                   ; ¿Dos ocupadas?
jr      z, zxMoveAttack_258  ; Sí, salta
ld      c, KEY1             ; C = tecla 1
call    ToMove              ; Mueve a celda 1
ret     z                   ; Si es correcto, sale
ld      c, KEY4             ; C = tecla 4
call    ToMove              ; Mueve a celda 4
ret     z                   ; Si es correcto, sale
ld      c, KEY7             ; C = tecla 7
call    ToMove              ; Mueve a celda 7
ret     z                   ; Si es correcto, sale
```

```
zxMoveAttack_258:
ld      a, (ix+$02)         ; A = valor celda 2
add     a, (ix+$05)         ; A+= valor celda 5
add     a, (ix+$08)         ; A+= valor celda 8
ld      c, a                ; Preserva A en C
and     $03                 ; A = casillas jugador 1
jr      nz, zxMoveAttack_369 ; ¿Alguna? Sí, salta
ld      a, c                ; Recupera A
and     $30                 ; A = casillas Spectrum
cp      b                   ; ¿Dos ocupadas?
jr      z, zxMoveAttack_369  ; Sí, salta
ld      c, KEY2             ; C = tecla 2
call    ToMove              ; Mueve a celda 2
ret     z                   ; Si es correcto, sale
ld      c, KEY5             ; C = tecla 5
call    ToMove              ; Mueve a celda 5
ret     z                   ; Si es correcto, sale
ld      c, KEY8             ; C = tecla 8
call    ToMove              ; Mueve a celda 8
ret     z                   ; Si es correcto, sale

zxMoveAttack_369:
ld      a, (ix+$03)         ; A = valor celda 3
add     a, (ix+$06)         ; A+= valor celda 6
add     a, (ix+$09)         ; A+= valor celda 9
ld      c, a                ; Preserva A en C
and     $03                 ; A = casillas jugador 1
jr      nz, zxMoveAttack_159 ; ¿Alguna? Sí, salta
ld      a, c                ; Recupera A
and     $30                 ; A = casillas Spectrum
cp      b                   ; ¿Dos ocupadas?
jr      z, zxMoveAttack_159  ; Sí, salta
ld      c, KEY3             ; C = tecla 3
call    ToMove              ; Mueve a celda 3
ret     z                   ; Si es correcto, sale
ld      c, KEY6             ; C = tecla 6
call    ToMove              ; Mueve a celda 6
ret     z                   ; Si es correcto, sale
ld      c, KEY9             ; C = tecla 9
call    ToMove              ; Mueve a celda 9
ret     z                   ; Si es correcto, sale

zxMoveAttack_159:
ld      a, (ix+$01)         ; A = valor celda 1
add     a, (ix+$05)         ; A+= valor celda 5
add     a, (ix+$09)         ; A+= valor celda 9
ld      c, a                ; Preserva A en C
and     $03                 ; A = casillas jugador 1
jr      nz, zxMoveAttack_357 ; ¿Alguna? Sí, salta
ld      a, c                ; Recupera A
and     $30                 ; A = casillas Spectrum
cp      b                   ; ¿Dos ocupadas?
jr      z, zxMoveAttack_357  ; Sí, salta
ld      c, KEY1             ; C = tecla 1
call    ToMove              ; Mueve a celda 1
ret     z                   ; Si es correcto, sale
ld      c, KEY5             ; C = tecla 5
call    ToMove              ; Mueve a celda 5
ret     z                   ; Si es correcto, sale
ld      c, KEY9             ; C = tecla 9
call    ToMove              ; Mueve a celda 9
ret     z                   ; Si es correcto, sale

zxMoveAttack_357:
ld      a, (ix+$03)         ; A = valor celda 3
add     a, (ix+$05)         ; A+= valor celda 5
add     a, (ix+$07)         ; A+= valor celda 7
ld      c, a                ; Preserva A en C
and     $03                 ; A = casillas jugador 1
jr      nz, zxMoveGeneric   ; ¿Alguna? Sí, salta
ld      a, c                ; Recupera A
```

```
and     $30                 ; A = casillas Spectrum
cp      b                   ; ¿Dos ocupadas?
jr      z, zxMoveGeneric    ; Sí, salta
ld      c, KEY3             ; C = tecla 3
call    ToMove              ; Mueve a celda 3
ret     z                   ; Si es correcto, sale
ld      c, KEY5             ; C = tecla 5
call    ToMove              ; Mueve a celda 5
ret     z                   ; Si es correcto, sale
ld      c, KEY7             ; C = tecla 7
call    ToMove              ; Mueve a celda 7
ret     z                   ; Si es correcto, sale
```

Si llegamos aquí, el Spectrum no ha encontrado dónde mover y mueve a la primera celda que esté libre.

```
; ------------------------------------------------------------------
; Movimiento genérico.
; Si con todo lo anterior, no ha hecho movimiento
; mueve a la primera celda libre.
; ------------------------------------------------------------------
zxMoveGeneric:
ld      c, '1'              ; C = ascii 1 (celda 1)
ld      b, $09              ; B = celdas totales
ld      a, $00              ; A = celda vacía
ld      hl, Grid            ; HL = dirección 1ª celda

zxMoveGeneric_loop:
cp      (hl)                ; ¿Celda libre?
jr      z, zxMoveGeneric_end    ; Sí, salta
inc     c                   ; C = ascii siguiente celda
inc     hl                  ; HL = siguiente celda
djnz    zxMoveGeneric_loop  ; Bucle hasta que B = 0

zxMoveGeneric_end:
call    ToMove              ; Mueve a celda libre
ret                         ; Sale
```

Ya hemos implementado la forma en la que se comporta en las partidas a un jugador el Spectrum. No es perfecta, tiene varias lagunas y en ocasiones el Spectrum realiza algún movimiento errático. Eso está bien, si lo hiciéramos perfecto no podríamos ganar nunca al Spectrum, y eso no gusta a nadie.

Ahora tenemos que integrar lo que hemos implementado dentro del bucle principal en Main.asm.

Localizamos la etiqueta loop_lostMov, y cinco líneas por debajo sustituimos JR Z, loop_keyCont por:

```
jr      z, loop_players     ; No activo, salta
```

Localizamos la etiqueta loop_keyCont y la línea que tenemos por encima, JR loop_cont, la sustituimos por:

```
jp      loop_cont           ; Salta
```

Entre esta línea y loop_keyCont vamos añadir el movimiento del Spectrum en el caso de partidas a un jugador, esto hace que JR de un error de fuera de rango, por eso lo hemos sustituido por JP.

Por último, entre esta línea y loop_keyCont añadimos las líneas siguientes:

```
loop_players:
ld      a, (PlayerMoves)    ; A = jugador que mueve
or      a                   ; ¿Jugador 1?
jr      z, loop_keyCont     ; Sí, salta
```

```
ld      a, (MaxPlayers)     ; A = jugadores
cp      $02                 ; ¿2 jugadores?
jr      z, loop_keyCont     ; Sí, salta
call    ZxMove              ; Mueve Spectrum
jr      nz, loop_players    ; NZ = incorrecto, bucle
push    bc                  ; Preserva BC
ld      bc, SoundSpectrum   ; BC = dirección sonido
call    PlayMusic           ; Emite sonido
pop     bc                  ; Recupera BC
jr      loop_print          ; Pinta movimiento
```

Obtenemos que jugador mueve, si es el dos obtenemos cuántos jugadores son y si es un solo jugador, mueve el Spectrum.

Compilamos, cargamos en el emulador y vemos el resultado. Si todo ha ido bien ya podemos jugar contra el Spectrum. Si os parece que es demasiado fácil, podéis pulir más los movimientos del Spectrum; os adelanto que lo haremos en el próximo capítulo.

Descarga el código fuente desde aquí

https://tinyurl.com/2hyodh73

# Paso 6: ajustes finales

Hemos llegado al capítulo final. Vamos a realizar unos ajustes que, si bien no son del todo necesarios, creo que pueden ser de interés.

Añadiremos una opción en el menú para que puedan seleccionar los jugadores el número máximo de tablas que puede haber.

Siguiendo con las tablas, actualmente para finalizar el punto es necesario realizar todos los movimientos posibles (ocupar la totalidad de las celdas), aunque en ocasiones ya sabemos que no es posible ganar y son tablas. Implementaremos la detección de tablas para que detecte si ya no es posible ganar y finalizar el punto.

Propondremos unas modificaciones que nos harán ahorrar unos cuantos bytes y ciclos de reloj.

Modificaremos los movimientos del Spectrum para que ya no sea tan sencillo ganarle.

Por último, vamos a añadir la pantalla de carga.

## Menú tablas

Actualmente, el número máximo de tablas lo tenemos declarado en la constante MAXTIES en Var.asm; la borramos.

Localizamos MaxPoints y debajo añadimos la nueva variable para las tablas:

```
MaxTies:        db $05              ; Máximo tablas
```

Localizamos TitleEspamatica y justo encima añadimos la opción de menú para las tablas:

```
TitleOptionTies:    db $10, INK7, $16, $10, $08, "4. ", $10, INK6
                    defm "Tablas"
```

Para que quede más simétrico, localizamos Title3EnRaya y en la parte final de la línea sustituimos $16, $02, $0A por $16, $04, $0A.

En Main.asm vamos a añadir el tratamiento de la nueva opción del menú. Localizamos menu_Time y dos líneas más abajo sustituimos JR NZ, menu_op por:

```
jr      nz, menu_Ties       ; No, salta
```

Localizamos menu_TimeDo y tres líneas más abajo, después de JR menu_op, añadimos el tratamiento de la nueva opción de menú:

```
menu_Ties:
cp      KEY4                ; ¿Pulsado 4?
jr      nz, menu_op         ; No, bucle
ld      a, (MaxTies)        ; A = tablas
add     a, $02              ; A += 2
cp      $0a                 ; ¿A < 10?
jr      c, menu_TiesDo      ; Sí, salta
ld      a, $03              ; A = 3
menu_TiesDo:
ld      (MaxTies), a        ; Actualiza tablas
jr      menu_op             ; Bucle
```

Si se ha pulsado la tecla cuatro, se añade dos al número máximo de tablas, si es mayor de nueve se pone a tres, y se carga en memoria.

Tanto este rango de valores, como la diferencia entre uno y otro valor lo podéis modificar a vuestro gusto. También lo podéis cambiar a vuestro gusto en puntos y tiempo.

Una vez que tenemos el número máximo de tablas guardado en memoria, hay que usarlo. Localizamos la etiqueta loop_reset, la línea LD B, MAXTIES y la sustituimos por las siguientes:

```
ld      a, (MaxTies)        ; A = máximo tablas
ld      b, a                ; B = A
```

Ya solo nos queda pintar el valor seleccionado. En Screen.asm, localizamos PrintOptions y, justo por encima de JP PrintBCD, añadimos las líneas siguientes:

```
call    PrintBCD        ; Lo pinta
ld      b, INI_TOP-$10  ; B = coord Y
call    AT              ; Posiciona cursor
ld      hl, MaxTies     ; HL = valor tiempo
```

Compilamos, cargamos en el emulador y comprobamos que ya podemos definir el número de tablas, y que al alcanzarse se finaliza la partida.


## Detección de tablas

Actualmente, para que se detecten tablas tienen que estar todas las celdas ocupadas y que no haya tres en raya. En realidad, es posible saber si el punto terminará en tablas antes de que el tablero este lleno.

Vamos a Game.asm e implementamos la detección de tablas.

Para saber si hay posibilidad de algún movimiento, sólo hay que saber si queda alguna combinación en la que las celdas estén ocupadas por un solo jugador, o por ninguno.

```
; ------------------------------------------------------------------
; Comprueba si hay tablas.
; Para poder comprobar si hay tablas.
;
; Salida:   Z   ->  No hay tablas.
;           NZ  ->  Hay tablas.
;
; Altera el valor de los registros AF, BC, DE, HL e IX.
; ------------------------------------------------------------------
CheckTies:
ld      b, $f0              ; B = máscara celdas un jugador
ld      c, $0f              ; B = máscara celdas otro jugador
```

Cargamos en B la máscara para quedarnos con las celdas que ocupa un jugador, y en C la máscara para el otro.

```
checkTies_check123:
ld      hl, Grid            ; HL = dirección grid
ld      a, (hl)             ; A = valor celda 1
inc     hl                  ; HL = dirección celda 2
```

```
add     a, (hl)             ; A+= valor celda 2
inc     hl                  ; HL = dirección celda 3
add     a, (hl)             ; A+= valor celda 2
ld      d, a                ; Preserva A en D
and     b                   ; A = celdas ocupadas por un jugador
ret     z                   ; Ninguna, sale
ld      a, d                ; Recupera A desde D
and     c                   ; A = celdas ocupadas por otro jugador
ret     z                   ; Ninguna, sale
```

Apuntando HL a Grid, sumamos en A el valor de las tres celdas, incrementando HL para pasar de una celda a otra. Comprobamos si hay alguna celda ocupada por un jugador, y de no ser así salimos. Si sí las hay, comprobamos si hay celdas ocupadas por el otro jugador, y si no las hay salimos.

Las dos comprobaciones siguientes tiene la misma estructura.

```
checkTies_check456:
inc     hl                  ; A = dirección celda 4
ld      a, (hl)             ; A = valor celda 4
inc     hl                  ; HL = dirección celda 5
add     a, (hl)             ; A+= valor celda 5
inc     hl                  ; HL = dirección celda 6
add     a, (hl)             ; A+= valor celda 6
ld      d, a                ; Preserva A en D
and     b                   ; A = celdas ocupadas por un jugador
ret     z                   ; Ninguna, sale
ld      a, d                ; Recupera A desde D
and     c                   ; A = celdas ocupadas por otro jugador
ret     z                   ; Ninguna, sale

checkTies_check789:
inc     hl                  ; A = dirección celda 7
ld      a, (hl)             ; A = valor celda 7
inc     hl                  ; HL = dirección celda 8
add     a, (hl)             ; A+= valor celda 8
inc     hl                  ; HL = dirección celda 9
add     a, (hl)             ; A+= valor celda 9
ld      d, a                ; Preserva A en D
and     b                   ; A = celdas ocupadas por un jugador
ret     z                   ; Ninguna, sale
ld      a, d                ; Recupera A desde D
and     c                   ; A = celdas ocupadas por otro jugador
ret     z                   ; Ninguna, sale
```

Fijaos bien en cómo vamos recorriendo las celdas con HL, esto nos ayuda para ahorrar unos bytes en una implementación que hicimos anteriormente.

Las siguientes comprobaciones no las podemos hacer cambiando de celda con HL, ya que no son numéricamente contiguas; usamos IX.

```
checkTies_check147:
ld      ix, Grid-$01        ; IX = dirección  Grid - 1
ld      a, (ix+$01)         ; A = valor celda 1
add     a, (ix+$04)         ; A+= valor celda 4
add     a, (ix+$07)         ; A+= valor celda 7
ld      d, a                ; Preserva A en D
and     b                   ; A = celdas ocupadas por un jugador
ret     z                   ; Ninguna, sale
ld      a, d                ; Recupera A desde D
and     c                   ; A = celdas ocupadas por otro jugador
ret     z                   ; Ninguna, sale
```

El resto de comprobaciones tienen la misma estructura.

```
checkTies_check258:
ld      a, (ix+$02)         ; A = valor celda 2
add     a, (ix+$05)         ; A+= valor celda 5
```

```
add     a, (ix+$08)         ; A+= valor celda 8
ld      d, a                ; Preserva A en D
and     b                   ; A = celdas ocupadas por un jugador
ret     z                   ; Ninguna, sale
ld      a, d                ; Recupera A desde D
and     c                   ; A = celdas ocupadas por otro jugador
ret     z                   ; Ninguna, sale

checkTies_check369:
ld      a, (ix+$03)         ; A = valor celda 3
add     a, (ix+$06)         ; A+= valor celda 6
add     a, (ix+$09)         ; A+= valor celda 9
ld      d, a                ; Preserva A en D
and     b                   ; A = celdas ocupadas por un jugador
ret     z                   ; Ninguna, sale
ld      a, d                ; Recupera A desde D
and     c                   ; A = celdas ocupadas por otro jugador
ret     z                   ; Ninguna, sale

checkTies_check159:
ld      a, (ix+$01)         ; A = valor celda 1
add     a, (ix+$05)         ; A+= valor celda 5
add     a, (ix+$09)         ; A+= valor celda 9
ld      d, a                ; Preserva A en D
and     b                   ; A = celdas ocupadas por un jugador
ret     z                   ; Ninguna, sale
ld      a, d                ; Recupera A desde D
and     c                   ; A = celdas ocupadas por otro jugador
ret     z                   ; Ninguna, sale

checkTies_check357:
ld      a, (ix+$03)         ; A = valor celda 3
add     a, (ix+$05)         ; A+= valor celda 5
add     a, (ix+$07)         ; A+= valor celda 7
ld      d, a                ; Preserva A en D
and     b                   ; A = celdas ocupadas por un jugador
ret     z                   ; Ninguna, sale
ld      a, d                ; Recupera A desde D
and     c                   ; A = celdas ocupadas por otro jugador
ret                         ; Sale con Z en estado correcto
```

Con esto ya tenemos la predicción de si va a haber tablas, sólo queda ir a Main.asm y usarla.

Localizamos loop_tie y tres líneas más abajo sustituimos:

```
ld      a, $09              ; A = 9
cp      (hl)                ; ¿Contador = 9?
jr      nz, loop_cont       ; No, salta
```

Por:

```
call    CheckTies           ; ¿Algún movimiento posible?
jr      z, loop_cont        ; No, salta
```

Ya no hace falta esperar a que el tablero esté lleno para saber si hay o no tablas. Llamamos a CheckTies y si no hay tablas seguimos con el punto.

Compilamos, cargamos en el emulador y vemos los resultados.


Quizá veáis un comportamiento que os hace pensar que algo no funciona. Veis la línea uno, dos, tres libre, pero sabes que el Spectrum va a ocupar una celda, luego tú otra y por eso debería marcar tablas.

El sistema predice pero no adivina. Imaginaos que en una fila hay dos celdas libres y le toca mover al jugador que no tiene la celda ocupada. Bueno, va a ocupar una de las celdas así que ya sabemos que son tablas, pero imaginad que el jugador se duerme en los laureles y pierde el turno, el otro jugador ocupa otra celda y solo queda una libre, y sigue sin poderse predecir tablas, si el otro jugador se vuelve a dormir, pierde turno otra vez y el primer jugador consigue tres en raya.

## Ahorramos bytes y ciclos de reloj

Vamos a implementar varias modificaciones para ahorrar bytes y ciclos de reloj.

Las primeras de ellas van a ser en Game.asm, sobre el conjunto de rutinas CheckWinner y ZxMove, aplicando la forma en la que hemos implementado CheckTies en las líneas horizontales, las numéricamente contiguas.

Vamos a la etiqueta CheckWinner_check y localizamos las líneas siguientes:

```
ld      a, (ix+1)               ; A = celda 1
add     a, (ix+2)               ; A+= celda 2
add     a, (ix+3)               ; A+= celda 3
```

Y las sustituimos por:

```
ld      hl, Grid                ; HL = dirección celda 1
ld      a, (hl)                 ; A = valor celda 1
inc     hl                      ; HL = dirección celda 2
add     a, (hl)                 ; A+= valor celda 2
inc     hl                      ; HL = dirección celda 3
add     a, (hl)                 ; A+= valor celda 3
```

Localizamos las líneas:

```
ld      a, (ix+4)               ; A = celda 4
add     a, (ix+5)               ; A+= celda 5
add     a, (ix+6)               ; A+= celda 6
```

Y las sustituimos por:

```
inc     hl                      ; HL = dirección celda 4
ld      a, (hl)                 ; A = valor celda 4
inc     hl                      ; HL = dirección celda 5
add     a, (hl)                 ; A+= valor celda 5
inc     hl                      ; HL = dirección celda 6
add     a, (hl)                 ; A+= valor celda 6
```

Localizamos las líneas:

```
ld      a, (ix+7)               ; A = celda 7
```

```
add     a, (ix+8)               ; A+= celda 8
add     a, (ix+9)               ; A+= celda 9
```

Y las sustituimos por:

```
inc     hl                      ; HL = dirección celda 7
ld      a, (hl)                 ; A = valor celda 7
inc     hl                      ; HL = dirección celda 8
add     a, (hl)                 ; A+= valor celda 8
inc     hl                      ; HL = dirección celda 9
add     a, (hl)                 ; A+= valor celda 9
```

Esta parte está lista, añadid en los comentarios de la rutina a HL como registro afectado.

Esta es la comparativa entre ambas versiones:

| Versión | Ciclos | Bytes |
|---|---|---|
| 1 | 688/641 | 118 |
| 2 | 638/591 | 111 |

Como se observa, estamos ahorrando cincuenta ciclos de reloj y siete bytes. Compilamos, cargamos en el emulador y vemos como todo sigue funcionando.

Procedemos a modificar ZxMove, en concreto dos partes de este conjunto de rutinas. Localizamos zxMoveToWin_123 y sustituimos:

```
ld      a, (ix+$01)         ; A = valor celda 1
add     a, (ix+$02)         ; A+= valor celda 2
add     a, (ix+$03)         ; A+= valor celda 3
```

Por:

```
ld      hl, Grid            ; HL = dirección celda 1
ld      a, (hl)             ; A = valor celda 1
inc     hl                  ; HL = dirección celda 2
add     a, (hl)             ; A+= valor celda 2
inc     hl                  ; HL = dirección celda 3
add     a, (hl)             ; A+= valor celda 3
```

Localizamos zxMoveToWin_456 y sustituimos:

```
ld      a, (ix+$04)         ; A = valor celda 4
add     a, (ix+$05)         ; A+= valor celda 5
add     a, (ix+$06)         ; A+= valor celda 6
```

Por:

```
ld      hl, Grid+$03        ; HL = dirección celda 4
ld      a, (hl)             ; A = valor celda 4
inc     hl                  ; HL = dirección celda 5
add     a, (hl)             ; A+= valor celda 5
inc     hl                  ; HL = dirección celda 6
add     a, (hl)             ; A+= valor celda 6
```

Localizamos zxMoveToWin_789 y sustituimos:

```
ld      a, (ix+$07)         ; A = valor celda 7
add     a, (ix+$08)         ; A+= valor celda 8
add     a, (ix+$09)         ; A+= valor celda 9
```

Por:

```
ld      hl, Grid+$06        ; HL = dirección celda 7
ld      a, (hl)             ; A = valor celda 7
inc     hl                  ; HL = dirección celda 8
```

```
add     a, (hl)             ; A+= valor celda 8
inc     hl                  ; HL = dirección celda 9
add     a, (hl)             ; A+= valor celda 9
```

Ya hemos terminado con la primera parte. La diferencia principal con CheckTies y CheckWinner es que, aunque son celdas numéricamente contiguas, el paso de la celda tres a la cuatro y de la seis a la siete no es con INC HL, ya que ToMove altera el valor de HL.

Localizamos zxMoveAttack_123 y sustituimos:

```
ld      a, (ix+$01)         ; A = valor celda 1
add     a, (ix+$02)         ; A+= valor celda 2
add     a, (ix+$03)         ; A+= valor celda 3
```

Por:

```
ld      hl, Grid            ; HL = dirección celda 1
ld      a, (hl)             ; A = valor celda 1
inc     hl                  ; HL = dirección celda 2
add     a, (hl)             ; A+= valor celda 2
inc     hl                  ; HL = dirección celda 3
add     a, (hl)             ; A+= valor celda 3
```

Localizamos zxMoveAttack_456 y sustituimos:

```
ld      a, (ix+$04)         ; A = valor celda 4
add     a, (ix+$05)         ; A+= valor celda 5
add     a, (ix+$06)         ; A+= valor celda 6
```

Por:

```
ld      hl, Grid+$03        ; HL = dirección celda 4
ld      a, (hl)             ; A = valor celda 4
inc     hl                  ; HL = dirección celda 5
add     a, (hl)             ; A+= valor celda 5
inc     hl                  ; HL = dirección celda 6
add     a, (hl)             ; A+= valor celda 6
```

Localizamos zxMoveAttack_789 y sustituimos:

```
ld      a, (ix+$07)         ; A = valor celda 7
add     a, (ix+$08)         ; A+= valor celda 8
add     a, (ix+$09)         ; A+= valor celda 9
```

Por:

```
ld      hl, Grid+$06        ; HL = dirección celda 7
ld      a, (hl)             ; A = valor celda 7
inc     hl                  ; HL = dirección celda 8
add     a, (hl)             ; A+= valor celda 8
inc     hl                  ; HL = dirección celda 9
add     a, (hl)             ; A+= valor celda 9
```

Hacemos otra modificación, en zxMoveDefence_cornerBlock34 y en zxMoveDefence_cornerBlock67, que nos hace ahorrar ciclos de reloj, pero ningún byte.

Sustituimos en zxMoveDefence_cornerBlock34:

```
ld      a, (ix+$03)         ; A = valor celda 3
add     a, (ix+$04)         ; A+= valor celda 4
```

Por:

```
ld      hl, Grid+$02        ; A = dirección celda 3
ld      a, (hl)             ; A = valor celda 3
inc     hl                  ; HL = dirección celda 4
add     a, (hl)             ; A+= valor celda 4
```

En zxMoveDefence_cornerBlock67 sustituimos:

```
ld      a, (ix+$06)         ; A = valor celda 6
add     a, (ix+$07)         ; A+= valor celda 7
```

Por:

```
ld      hl, Grid+$05        ; A = dirección celda 6
ld      a, (hl)             ; A = valor celda 6
inc     hl                  ; HL = dirección celda 7
add     a, (hl)             ; A+= valor celda 7
```

Las modificaciones en Game.asm ya están. Como ya comenté, añadid en los comentarios de la rutina a HL como registro afectado, ya se nos olvidó antes, pues el registro HL ya se veía afectado por la rutina zxMoveGeneric.

Esta es la comparativa entre las dos versiones de ZxMove:

| Versión | Ciclos | Bytes |
|---|---|---|
| 1 | 4632/4079 | 808 |
| 2 | 4532/3979 | 802 |

La versión dos ocupa seis bytes menos y tarda cien ciclos menos que la uno, lo que parece poco si lo comparamos con el ahorro en bytes de CheckWinner, con menos modificaciones. Recordad que allí el paso de la celda tres a la celda cuatro y de la seis a la siete lo hacemos con INC HL y aquí no podemos.

La diferencia de INC HL con LD HL, Grid es de cuatro ciclos y dos bytes, y eso nos limita el ahorro.

Si tuviéramos problemas de capacidad, que no es así, podemos ahorrar otro buen puñado de bytes pintando la misma ficha, con distinto color, para ambos jugadores.

Vamos a Sprite.asm, comentamos la definición Sprite_P2 y las de Sprite_CROSS, Sprite_SLASH y Sprite_MINUS las ponemos al inicio del archivo, quedando así:

```
; ------------------------------------------------------------------
; Fichero: Sprite.asm
;
; Definición de los gráficos.
; ------------------------------------------------------------------
; Sprite de la cruceta
Sprite_CROSS:
db $18, $18, $18, $ff, $ff, $18, $18, $18   ; $90

; Sprite de la línea vertical
Sprite_SLASH:
db $18, $18, $18, $18, $18, $18, $18, $18   ; $91

; Sprite de la línea horizontal
Sprite_MINUS:
db $00, $00, $00, $ff, $ff, $00, $00, $00   ; $92

; Sprite del jugador 1
Sprite_P1:
db $c0, $e0, $70, $38, $1c, $0e, $07, $03   ; $93
db $03, $07, $0e, $1c, $38, $70, $e0, $c0   ; $94

; ; Sprite del jugador 2
; Sprite_P2:
; db $03, $0f, $1c, $30, $60, $60, $c0, $c0 ; $95 Arriba/Izquierda
; db $c0, $f0, $38, $0c, $06, $06, $03, $03 ; $96 Arriba/Derecha
; db $c0, $c0, $60, $60, $30, $1c, $0f, $03 ; $97 Abajo/Izquierda
; db $03, $03, $06, $06, $0c, $38, $f0, $c0 ; $98 Abajo/Derecha
```

Comentar el sprite del jugador dos no sólo obliga a modificar la rutina que pinta las fichas, también obliga a modificar la rutina que pinta el tablero, ya que cambian los UDG. Hemos subido a la parte de arriba los sprites del tablero para sólo modificar una vez la rutina que lo pinta, y para que si más adelante decidimos volver a pintar las dos fichas, no se vea afectada.

Al comentar Sprite_P2 ahorramos treinta y dos bytes. No parece mucho, pero en ciertas situaciones nos puede salvar.

Vamos a Var.asm y modificamos la definición de Board_1 y _2 y la dejamos así:

```
 ; Líneas verticales del tablero.
Board_1:
db $12, $00, $13, $00
db $20, $20, $20, $20, $91, $20, $20, $20, $20, $91, $20, $20, $20
db $20, $ff

; Líneas horizontales del tablero.
Board_2:
db $92, $92, $92, $92, $90, $92, $92, $92, $92, $90, $92, $92, $92
db $92, $ff
```

Compilamos, cargamos en el emulador y vemos los resultados, concretamente el desastre que hemos organizado.


Que la ficha del jugador dos se pinte mal lo esperábamos, pero ¿qué pasa con el tablero? Hemos cambiado la posición de la definición de los sprites, pero UDG sigue apuntando a Sprite_P1.

Vamos a Main.asm, y debajo de la etiqueta Main sustituimos:

```
ld      hl, Sprite_P1       ; HL = dirección Sprite_P1
```

Por:

```
ld      hl, Sprite_CROSS    ; HL = dirección Sprite_CROSS
```

Cambiad también el comentario de la línea de abajo.

Compilamos, cargamos en el emulador y vemos que el tablero se pinta bien, pero las fichas no. Tranquilos, era lo esperado.


Vamos a modificar la rutina que pinta las fichas para que lo haga bien, o no, lo podéis dejar así; tres en raya malditas.

Localizamos en Game.asm printOXO_X y las líneas en las que se carga el UDG en A. Le sumamos tres al valor que carga:

```
ld      a, $90          pasa a          ld      a, $93
ld      a, $91          pasa a          ld      a, $94
```

Localizamos printOXO_Y y las líneas en las que se carga el UDG en A. Le sumamos tres al valor que carga:

```
ld      a, $92          pasa a          ld      a, $95
ld      a, $93          pasa a          ld      a, $96
ld      a, $94          pasa a          ld      a, $97
ld      a, $95          pasa a          ld      a, $98
```

Con esto, si decidimos pintar dos fichas distintas, se pintaría el círculo.

Como ahora estamos en la situación de pintar una sola ficha, comentamos las líneas:

```
ld      a, $95              ; A = 1er sprite
ld      a, $98              ; A = 4º sprite
```

Y debajo de las mismas añadimos la línea:

```
ld      a, $93              ; A = 1er sprite
```

Comentamos las líneas:

```
ld      a, $96              ; A = 2º sprite
ld      a, $97              ; A = 3er sprite
```

Y debajo de las mismas añadimos la línea:

```
ld      a, $94              ; A = 2º sprite
```

Un poco de lío, ¿verdad? El aspecto final es este:

```
printOXO_X:
ld      a, INKPLAYER1       ; A = tinta jugador 1
call    INK                 ; Cambia tinta
ld      a, $93              ; A = 1er sprite
rst     $10                 ; Lo pinta
ld      a, $94              ; A = 2º sprite
rst     $10                 ; Lo pinta
dec     b                   ; B = línea inferior
call    AT                  ; Posiciona cursor
ld      a, $94              ; A = 2º sprite
rst     $10                 ; Lo pinta
ld      a, $93              ; A = 2º sprite
rst     $10                 ; Lo pinta
ret                         ; Sale

printOXO_Y:
ld      a, INKPLAYER2       ; A = tinta jugador 2
call    INK                 ; Cambia tinta
```

```
;ld      a, $95              ; A = 1er sprite
ld      a, $93              ; A = 1er sprite
rst     $10                 ; Lo pinta
;ld      a, $96             ; A = 2º sprite
ld      a, $94              ; A = 2º sprite
rst     $10                 ; Lo pinta
dec     b                   ; B = línea inferior
call    AT                  ; Posiciona cursor
;ld      a, $97             ; A = 3er sprite
ld      a, $94              ; A = 2º sprite
rst     $10                 ; Lo pinta
;ld      a, $98             ; A = 4º sprite
ld      a, $93              ; A = 1er sprite
rst     $10                 ; Lo pinta
ret                         ; Sale
```

De esta manera, si queremos volver a dibujar las dos fichas, en Sprite.asm descomentamos Sprite_P2, y en printOXO_Y alternamos los comentarios en las líneas LD A, …

Con esto ya se pintan bien las fichas en el tablero, pero no en la parte de información de la partida.

Volvemos a Var.asm y modificamos player1_figure que ahora es así:

```
player1_figure:     db $16, $04, $00, $90, $91, $0d, $91, $90
```

Y la dejamos así:

```
player1_figure:     db $16, $04, $00, $93, $94, $0d, $94, $93
```

Modificamos player2_figure que ahora es así:

```
player2_figure:     db $16, $04, $1b, $92, $93
                    db $16, $05, $1b, $94, $95, $ff
```

Y la dejamos así:

```
player2_figure:     db $16, $04, $1b, $93, $94
                    db $16, $05, $1b, $94, $93, $ff
```

Por último, añadimos, comentada, la definición para pintar fichas distintas.

```
; player2_figure:     db $16, $04, $1b, $95, $96
;                     db $16, $05, $1b, $97, $98, $ff
```

Compilamos, cargamos en el emulador y vemos los resultados.


Es decisión vuestra pintar la misma ficha en distinto color para los dos jugadores, o pintar distintas para cada jugador. Si utilizáramos una televisión en blanco y negro no habría ninguna duda.

Si mantenéis todas las modificaciones, hemos ahorrado cuarenta y cinco bytes y ciento cincuenta ciclos de reloj. Si decidís pintar distintas fichas para cada jugador, el ahorro de bytes se queda en trece.

## Movimiento del Spectrum

Si ya habéis jugado varias partidas contra el Spectrum, habréis averiguado la forma de ganarle siempre al iniciar vosotros la partida, y es que hay al menos un movimiento que el Spectrum no sabe defender, no se lo hemos programado.

El movimiento en concreto es ocupar dos celdas de la esquina: la dos y la seis o la seis y o la ocho. Si ocupamos la celda dos, el Spectrum ocupa la cinco, movemos a la celda seis y el Spectrum mueve a la siete, movemos a la tres y ya tenemos dos jugadas de tres en raya: celdas uno, dos y tres y celdas tres, seis y nueve.

En Game.asm, localizamos zxMoveAttack_123 y por encima de ella implementamos las líneas que hacen que ya no se pueda ganar al Spectrum realizando ese movimiento.

```
; ------------------------------------------------------------------
; Movimiento defensivo de esquina.
; ------------------------------------------------------------------
zxMoveDefence_corner24:
ld      a, (ix+$02)         ; A = valor celda 2
add     a, (ix+$04)         ; A+= valor celda 4
cp      b                   ; ¿A = B?
jr      nz, zxMoveDefence_corner26  ; No, salta
ld      c, KEY1             ; C = tecla 1
call    ToMove              ; Mueve a celda 1
ret     z                   ; Si es correcto, sale
```

Comprobamos si el jugador uno tiene ocupadas las casillas dos y cuatro, en cuyo caso movemos a la casilla uno si es posible.

El resto de comprobaciones tienen la misma estructura.

```
zxMoveDefence_corner26:
ld      a, (ix+$02)         ; A = valor celda 2
add     a, (ix+$06)         ; A+= valor celda 6
cp      b                   ; ¿A = B?
jr      nz, zxMoveDefence_corner84  ; No, salta
ld      c, KEY3             ; C = tecla 3
call    ToMove              ; Mueve a celda 3
ret     z                   ; Si es correcto, sale
zxMoveDefence_corner84:
ld      a, (ix+$08)         ; A = valor celda 8
add     a, (ix+$04)         ; A+= valor celda 4
cp      b                   ; ¿A = B?
jr      nz, zxMoveDefence_corner86  ; No, salta
ld      c, KEY7             ; C = tecla 7
call    ToMove              ; Mueve a celda 3
ret     z                   ; Si es correcto, sale
zxMoveDefence_corner86:
ld      a, (ix+$08)         ; A = valor celda 8
add     a, (ix+$06)         ; A+= valor celda 6
cp      b                   ; ¿A = B?
jr      nz, zxMoveAttack_123    ; No, salta
ld      c, KEY9             ; C = tecla 9
call    ToMove              ; Mueve a celda nueve
ret     z                   ; Si es correcto, sale
```

Aunque la jugada se da en dos esquinas, cubrimos las cuatro.

Justo encima de zxMoveDefence_cornerBlock2938Cont está la línea JR NZ, zxMoveAttack_123, la modificamos y la dejamos así:

```
jr      nz, zxMoveDefence_corner24  ; No, no mov cruz, salta
```

Si compiláis y jugáis algunas partidas contra el Spectrum veréis que ahora no es posible ganarle con esa jugada, no obstante sigue siendo posible ganarle, sigue habiendo jugadas que no sabe defender.


## Dificultad

Ahora es más difícil ganar al Spectrum, debemos esperar a que sea él el que empiece la partida y, dependiendo del movimiento que haga, podremos ganarle.

Vamos a añadir otra opción en el menú para poder seleccionar el nivel de dificultad: uno para que no cubra la jugada en la esquina, dos para que si lo haga.

En Var.asm, después de MaxTies, añadimos la variable para el nivel de dificultad:

```
Level:          db $02                  ; Nivel de dificultad
```

Al añadir una nueva opción de menú, está última queda pegada a la línea Espamatica. Subimos todas las opciones del menú una línea, quedando así:

```
TitleOptionStart:   db $10, INK1, $13, $01, $16, $07, $08, "0. "
                    db $10, INK5
                    defm "Empezar"
TitleOptionPlayer:  db $10, INK7, $13, $01, $16, $09, $08, "1. "
                    db $10, INK6
                    defm "Jugadores"
TitleOptionPoint:   db $10, INK7, $16, $0b, $08, "2. ", $10, INK6
                    defm "Puntos"
TitleOptionTime:    db $10, INK7, $16, $0d, $08, "3. ", $10, INK6
                    defm "Tiempo"
TitleOptionTies:    db $10, INK7, $16, $0f, $08, "4. ", $10, INK6
                    defm "Tablas"
```

Tras la definición de TitleOptionTies añadimos la de la dificultad:

```
TitleOptionLevel:   db $10, INK7, $16, $11, $08, "5. ", $10, INK6
                    defm "Dificultad"
```

En Screen.Asm localizamos PrintOptions y restamos una a todas las asignaciones de la coordenada Y:

```
LD B, INI_TOP-$0A           pasa a        LD B,INI_TOP-$09
LD B, INI_TOP-$0C           pasa a        LD B,INI_TOP-$0B
LD B, INI_TOP-$0E           pasa a        LD B,INI_TOP-$0D
LD B, INI_TOP-$10           pasa a        LD B,INI_TOP-$0F
```

Por encima de JP PrintBCD añadimos las líneas siguientes:

```
call    PrintBCD        ; Lo pinta
ld      b, INI_TOP-$11  ; B = coord Y
call    AT              ; Posiciona cursor
ld      hl, Level       ; HL = valor dificultad
```

Siendo el aspecto final de la rutina el siguiente:

```
; ------------------------------------------------------------------
; Pinta los valores de las opciones.
;
; Altera el valor de los registros AF, BC y HL.
; ------------------------------------------------------------------
PrintOptions:
ld      a, INK4         ; A = tinta verde
call    INK             ; Cambia tinta
ld      b, INI_TOP-$09  ; B = coord Y
ld      c, INI_LEFT-$15 ; C = coord X
call    AT              ; Posiciona cursor
ld      hl, MaxPlayers  ; HL = valor jugadores
call    PrintBCD        ; Lo pinta
ld      b, INI_TOP-$0b  ; B = coord Y
call    AT              ; Posiciona cursor
ld      hl, MaxPoints   ; HL = valor puntos
call    PrintBCD        ; Lo pinta
ld      b, INI_TOP-$0d  ; B = coord Y
call    AT              ; Posiciona cursor
ld      hl, seconds     ; HL = valor tiempo
call    PrintBCD        ; Lo pinta
ld      b, INI_TOP-$0f  ; B = coord Y
call    AT              ; Posiciona cursor
ld      hl, MaxTies     ; HL = valor tiempo
call    PrintBCD        ; Lo pinta
ld      b, INI_TOP-$11  ; B = coord Y
call    AT              ; Posiciona cursor
ld      hl, Level       ; HL = valor dificultad
jp      PrintBCD        ; Lo pinta y sale
```

En Main.asm, localizamos menu_Ties y debajo la línea CP KEY4. Tras esta línea sustituimos JR NZ, menu_op por:

```
jr      nz, menu_Level      ; No, bucle
```

Después de menu_TiesDo, tras la línea JR menu_op, añadimos las líneas de tratamiento de la nueva opción de menú:

```
menu_Level:
cp      KEY5                ; ¿Pulsado 5?
jr      nz, menu_op         ; No, bucle
ld      a, (Level)          ; A = dificultad
xor     $03                 ; Alterna entre 1 y 2
ld      (Level), a          ; Actualiza en memoria
jr      menu_op             ; Bucle
```

Por último, en Game.asm localizamos zxMoveDefence_corner24, y justo debajo de ella añadimos:

```
ld      a, (Level)          ; A = dificultad
cp      $01                 ; ¿Dificultad = 1?
jr      z, zxMoveAttack_123 ; Sí, salta
```

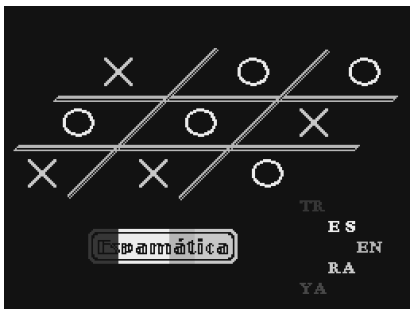
Con estas líneas, si el nivel de dificultad es uno no se comprueba la jugada de esquina. Compilad y probad.

## Pantalla de carga

Por último, vamos a añadir la pantalla de carga. ¿Y qué aporta si ya hemos visto como se hace en ZX-Pong y Batalla espacial? Pues esta vez lo vamos a hacer distinto, ya que no vamos a cargar la pantalla en VideoRAM, la cargaremos en otra dirección de memoria y luego haremos que aparezca de golpe.

Esta es la pantalla de carga, que podéis descargar desde:

https://tinyurl.com/2qe2txpz




Por favor, sed benévolos, el arte nunca ha sido lo mío. Veré con buenos ojos que diseñéis vuestra propia pantalla de carga, que seguro que es mejor que está; difícil no debe ser.

Para poder cargar la pantalla de carga en un área de memoria y después volcarla de golpe en la VideoRAM, necesitamos implementar la rutina que lo haga en un archivo aparte y compilarlo por separado.

El proceso de carga hará lo siguiente:

- Carga el cargador.
- Carga la rutina que vuelca la pantalla de carga en la dirección de memoria 24200.
- Carga la pantalla de carga en la dirección 24250.
- Ejecuta la rutina que vuelca la pantalla de carga a la VideoRAM.
- Carga el programa de tres en raya en la dirección 24200.
- Carga la rutina de interrupciones en la dirección 32348.
- Ejecuta el programa de tres en raya.

Lo primero que vamos a ver es el cargador que desarrollamos en Basic para hacer todo lo expuesto arriba.

```
 10 CLEAR 24200
 20 INK 0: PAPER 4: BORDER 4: CLS
 30 POKE 23610,255: POKE 23739,111
 40 LOAD ""CODE 24200
 50 LOAD ""CODE 24250
 60 RANDOMIZE USR 24200
 70 LOAD ""CODE 24200
 80 LOAD ""CODE 32348
 90 CLS
100 RANDOMIZE USR 24200
```

No olvidéis guardarlo con SAVE ”TresEnRaya” LINE 10.

El siguiente paso es implementar la rutina que vuelca la pantalla de carga a la VideoRAM. Creamos el archivo LoadScr.asm y agregamos las siguientes líneas:

```
; ---------------------------------------------------------------
; LoadScr.asm
;
; La pantalla de carga se cargará en $5eba, y esta rutina la pasará
```

```
; a la VideoRAM para que aparezca de golpe y luego limpiará la zona
; dónde se había cargado inicialmente.
; La limpieza de esa área de memoria no es necesaria, pero lo hacemos
; por si tenemos que depurar, no encontrar código que en realidad
; sean restos de la pantalla de carga.
; ------------------------------------------------------------------
org     $5e88               ; Dirección de carga

ld      hl, $5eba           ; HL = dirección dónde está la pantalla
ld      de, $4000           ; DE = dirección VideoRAM
ld      bc, $1b00           ; Longitud de la pantalla
ldir                        ; Vuelva la pantalla

ld      hl, $5eba           ; HL = dirección dónde está la pantalla
ld      de, $5ebb           ; Dirección siguiente
ld      bc, $1aff           ; Longitud de la pantalla - 1
ld      (hl), $00           ; Limpia la primera posición
ldir                        ; Limpia el resto

ret
```

La pantalla de carga la cargamos en $5eba. Esta rutina copia a la VideoRAM desde esa dirección $1B00 posiciones (6912 bytes), todo el área de píxeles y atributos.

Una vez copiado, limpia el área de memoria en la que se cargó la pantalla; no es necesario, pero si tenemos que depurar evitamos que quede código residual que nos pueda confundir.

Ya solo nos queda modificar el script que tenemos para compilar y generar el .tap final. La versión para Windows queda así:

```
echo off
cls
echo Compilando oxo
pasmo --name TresEnRaya --tap Main.asm oxo.tap oxo.log
echo Compilando int
pasmo --name Int --tap Int.asm int.tap int.log
echo Compilando loadscr
pasmo --name LoadScr --tap LoadScr.asm loadscr.tap loadscr.log
echo Generando TresEnRaya
copy /b /y cargador.tap+loadscr.tap+TresEnRayaScr.Tap+oxo.tap+int.tap TresEnRaya.tap
```

El aspecto de la versión para Linux sería éste:

```
clear
echo Compilando oxo
pasmo --name TresEnRaya --tap Main.asm oxo.tap oxo.log
echo Compilando int
pasmo --name Int --tap Int.asm int.tap int.log
echo Compilando loadscr
pasmo --name LoadScr --tap LoadScr.asm loadscr.tap loadscr.log
echo Generando TresEnRaya
cat cargador.tap loadscr.tap TresEnRayaScr.Tap oxo.tap int.tap > TresEnRaya.tap
```

Y con esto hemos terminado.

Podéis descargar el código fuente desde aquí

https://tinyurl.com/2e4dq2da