

## Contenido

<b>1.- Prologo.....</b>	<b>2</b>
<b>2.- Elementos necesarios para montar el IDE.....</b>	<b>3</b>
<b>3.- Instalación del emulador ZEsarUX.....</b>	<b>6</b>
Paso 1 – Descarga del código fuente de ZEsarUX .....	6
Paso 2 – Instalación y configuración del MinGW .....	7
Paso 3 – Compilación y configuración de ZEsarUX .....	8
<b>4.- Instalación y configuración del VS Code.....</b>	<b>11</b>
Paso 1 – Instalación de plugins básicos.....	12
Paso 2 – Instalación del plugin Z80 Debugger.....	14
Paso 3 – Instalación de node.js y el ensamblador SJASMPLUS.....	15
<b>5.- Creando nuestro IDE integrado para depurar .....</b>	<b>15</b>
5.1 – Abriendo la carpeta con nuestro programa de ejemplo .....	15
5.2 – Ficheros tasks.json y launch.json .....	16
5.3 – Ensamblado del código ASM con SJASMPLUS.....	21
5.4 – Lanzamiento de la sesión de depuración .....	22
<b>6.- Introducción a la depuración .....</b>	<b>23</b>
<b>7.- ¿Y ahora qué? .....</b>	<b>26</b>
<b>7.- Agradecimientos.....</b>	<b>26</b>

## 1.- Prologo

Me hago viejo.

En unos días me caen 48 castañas y, como pasa cuando uno se hace viejo, este verano durante las vacaciones me dio por echar la vista atrás y repasar aquellas cosas que me han ido pasando a lo largo de mi vida, y como me han influido para llegar a ser quien soy.

Y me di cuenta de que tenía una deuda histórica que zanjar.

Mi pasión por la informática comenzó allá por los años 80 cuando mi padre trajo un flamante Spectrum 16K que le regalaron en el banco con el que descubrí el apasionante mundo de los videojuegos. De ahí pasé a hacer algún programa en Basic y, ya en el instituto, me metí de cabeza con el C y el UNIX. Cuando terminé el COU, sólo tenía una cosa en la cabeza: estudiar Informática.

Ya en la Universidad, tras 6 añitos de mucho esfuerzo, conseguí mi flamante título de Ingeniero Informático, gracias a lo cual me gano la vida hoy en día, haciendo sistemas de información para Hospitales.

Sin embargo, hay algo que nunca hice y era una espinita que tenía clavada. Nunca aprendí ASM del Z80 (o código máquina, como le llamaba el Microhobby) ni conseguí sacarle todo el partido a esa maravillosa máquina sin cuya entrada en mi vida, quizás sería un licenciado en Historia en paro (mi otra gran pasión).

Así que tras pensarlo unos días decidí saldar mi vieja deuda, entrando en un mundo desconocido para mí, y me propuse hacer un juego para Spectrum en ASM puro y duro, ya que el reto no era tanto terminar un juego y publicarlo sino “dominar a la bestia” y aprender a hacer aquello que no fui capaz de lograr cuando era un chaval.

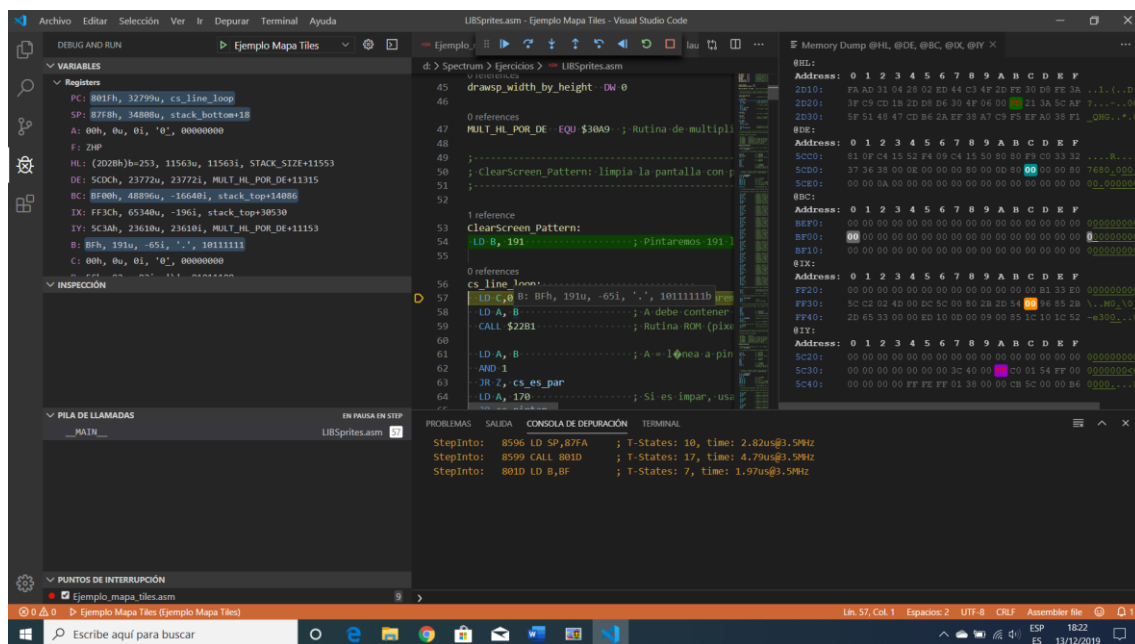
Con mucho ánimo, comencé mi periplo para aprender ASM del z80 leyéndome el impresionante e indispensable curso de **Compiler Soft**, que podéis encontrar en <https://wiki.speccy.org/cursos/ensamblador/indice>, cuyos ejemplos tecleaba en **Context** y probaba en **Spectaculator 8.0**, pero se me hacía muy cuesta arriba, ya que con este “IDE”, las posibilidades de depuración no son muy completas.

A la vuelta del verano, entré en varios grupos de Telegram donde conocí a un grupo de personas con mucha experiencia a las que plantear dudas y que fueron aportando su granito de arena para que progresase en mi camino. En uno de estos grupos conocí a César Hernández, desarrollador del maravilloso **ZEsaUX**, un emulador de Spectrum espectacular y en pleno crecimiento, que ofrece muchas de las herramientas que echaba en falta, aunque aún sin un IDE tan integrado como andaba buscando.

Un día, César habló en uno de los grupos de un plugin para VS Code, el **Z80 Debugger** de Thomas Busse, que permitía depurar ensamblador de una manera muy sencilla desde el código, corriendo sobre **ZEsaUX** y proporcionando las herramientas que no había encontrado hasta ese momento. Era lo que andaba buscando.

César me comentó que él trabajaba en UNIX y que realmente, no había tenido un beta tester del IDE en Windows. Cuando contacté con Thomas, la respuesta fue la misma: la cosa se ponía interesante...

Tras unas semanas de pruebas, cabreos, dolores de cabeza y cientos de correos y mensajes con Cesar y Thomas, y alguna que otra versión que me enviaron corrigiendo bugs, conseguí echarlo a andar y...es espectacular.



Mi esfuerzo no estaría terminado sin este documento que espero que sirva para que otros desarrolladores, sin tener que sufrir todo lo que yo he sufrido, puedan aprovechar el trabajo de César y Thomas con la suma de estas herramientas tan potentes a las que sus creadores dedican tanto tiempo, esfuerzo e ilusión.

Si al seguir el tutorial alguien se atasca en algún paso, o tiene alguna duda concreta, puede contactar conmigo mandándome un correo a [cesar.wagener@gmail.com](mailto:cesar.wagener@gmail.com) y, si es posible, trataré de echarle una mano.

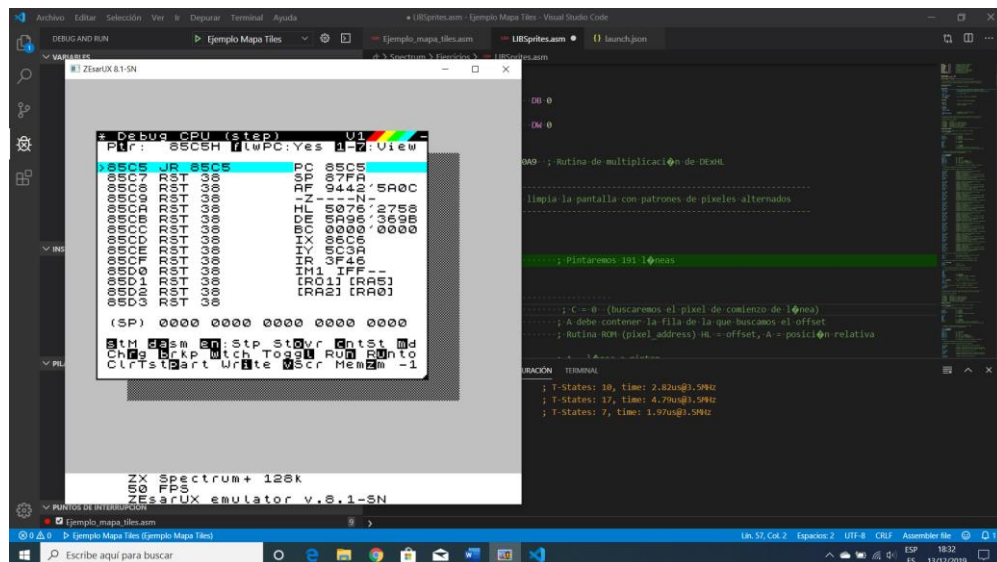
Y ahora, ¡vamos a meternos en faena!

## 2.- Elementos necesarios para montar el IDE

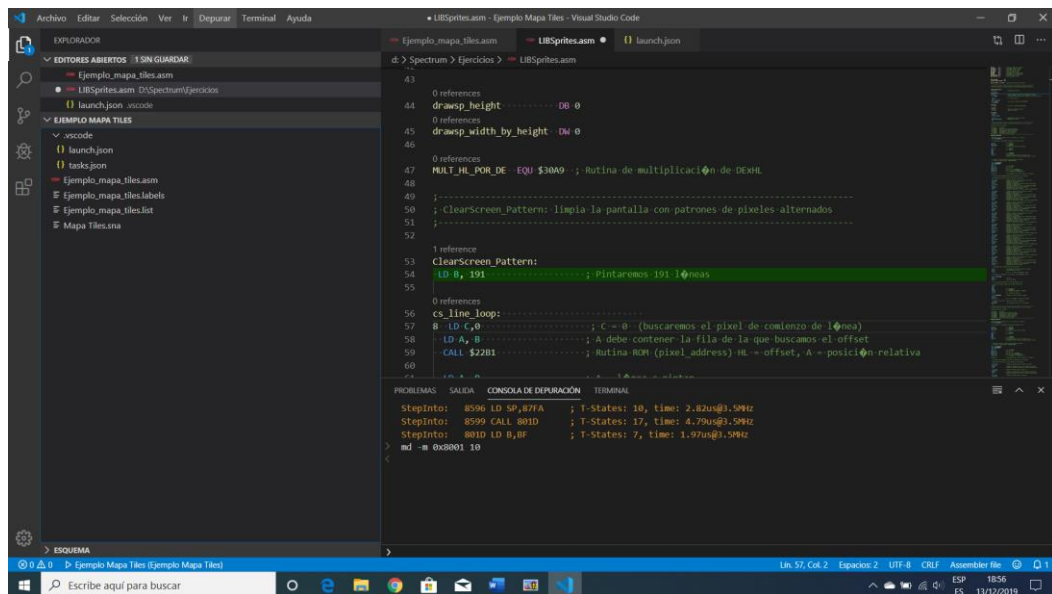
Bueno, lo primero que tengo que deciros es que todo lo que os voy a explicar lo he probado sobre un W10 Profesional, pero imagino que debe funcionar sin mayores problemas sobre otros sistemas Windows, e incluso trasladarse a Linux o Mac sin demasiados problemas.

Para montar el entorno, vamos a necesitar las siguientes herramientas:

- **Emulador ZEsarUX (ZX Second Emulator And Released for Unix):** Primera pieza clave. Es el emulador sobre el que ejecutaremos nuestro código. Tiene muchísimas posibilidades de configuración y buenas herramientas propias de depuración, que puede ofrecer a sistemas externos a través de sockets con el protocolo ZRCP. Podéis encontrar el código fuente, versiones ejecutables y documentación sobre el mismo en <https://github.com/chernandezba/zesarux>

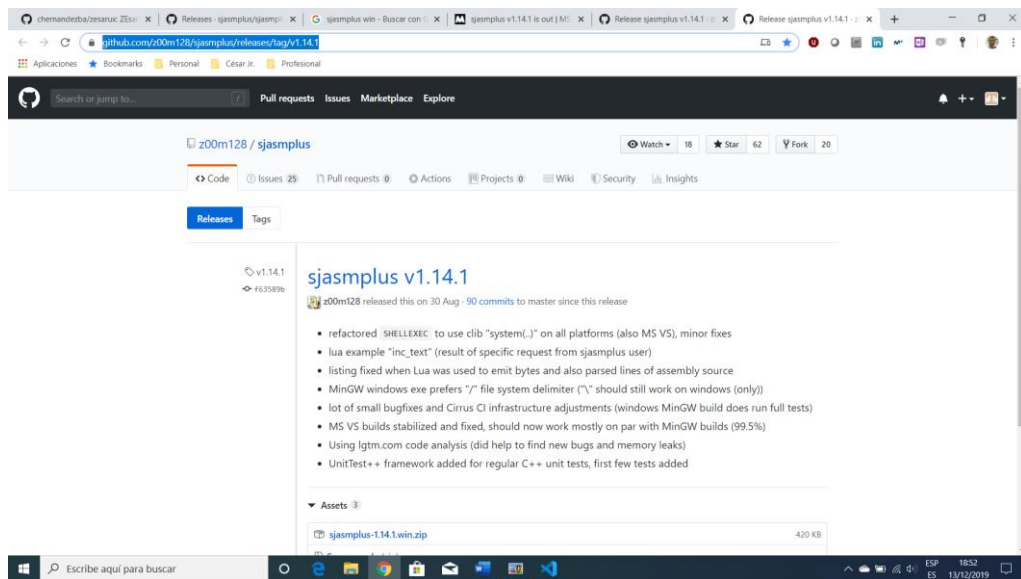


- **Compilador MinGW:** Aunque no sé si César ya habrá subido una nueva versión oficial, en mi caso he tenido que descargarme el fuente de ZEsarUX y compilarlo para tener acceso a los últimos desarrollos y correcciones de bugs liberadas por César. Podéis descargaros la última versión desde el siguiente enlace: <https://sourceforge.net/projects/mingw/>. Adicionalmente necesitaremos bajarnos la versión 1.2.15 de la librería SDL (fichero **SDL-devel-1.2.15-mingw32.tar.gz** que podéis descargar de esta dirección: <https://www.libsdl.org/download-1.2.php>).
- **Editor VS Code:** Herramienta desde la que escribiremos el código, ensamblaremos y lanzaremos la depuración. Es una herramienta de Microsoft muy usada por los desarrolladores, aunque os reconozco que yo no la conocía. Usaremos la versión 1.41 que podéis descargar desde este enlace: <https://code.visualstudio.com/download>



También tenéis que instalaros (si no lo tenéis ya) el Node JS, que podéis encontrar aquí: <https://nodejs.org/es/download/>

- Plugin Z80 Debug:** Plugin que vamos a instalar sobre VS Code y que, haciendo uso del protocolo ZRCP, nos permitirá depurar visualmente. Podéis encontrar el código fuente y mucha documentación sobre este plugin en esta dirección: <https://github.com/maziac/z80-debug>  
 Aunque puedes instalar el plugin directamente de una manera sencilla desde VS Code, Thomas me mandó un par de versiones para instalar manualmente desde VSCode que corrigen cosas que no funcionan en la versión “publicada”. Hasta que Thomas no libere la versión 0.9.3 oficial, tendréis que instalar el fichero .VSIX con la versión 0.9.3-2 que me ha preparado específicamente para corregir algún bug que no permitía inspeccionar correctamente el valor de los registros ni ver el contenido de las variables y las posiciones de memoria a las que apuntan en tiempo de ejecución.
- Ensamblador sjasmplus:** Será la herramienta con la que, desde el entorno IDE, ensamblaremos nuestros programas. Aunque hay betas más avanzadas, estoy trabajando con la versión v1.14.1, que se liberó en agosto y podéis descargar desde esta dirección: <https://github.com/z00m128/sjasmplus/releases/tag/v1.14.1>. Tenéis documentación completa sobre el mismo en estas otras direcciones: <http://z00m128.github.io/sjasmplus/documentation.html> ó <https://github.com/sjasmplus/sjasmplus/wiki>



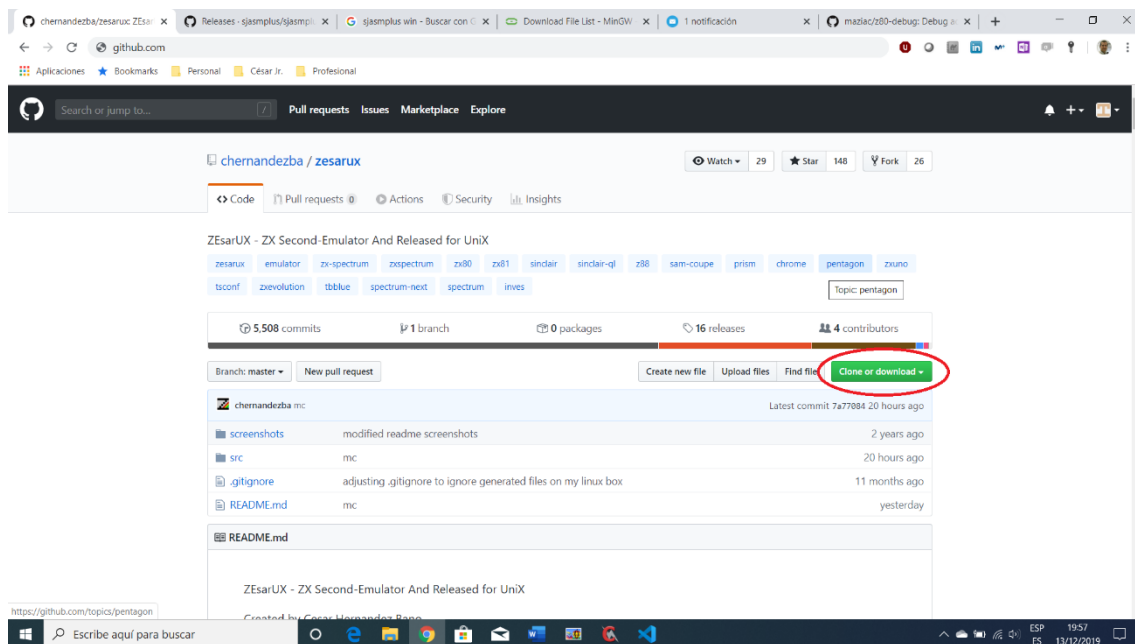
Con todos estos elementos descargados y preparados para ser instalados, ¡podemos ponernos manos a la obra!

### 3.- Instalación del emulador ZEsarUX

Comenzamos con el “Corazón de la bestia”, el emulador ZEsarUX.

#### Paso 1 – Descarga del código fuente de ZEsarUX

Desde la página indicada, pulsamos el botón **Clone or download** y descargamos el fichero zesarux-master.zip en el directorio donde vayamos a dejar el emulador (en mi caso, D:\Spectrum\ZEsarUX)



Una vez descargado el fichero, lo descomprimos sobre el mismo directorio y pasamos al siguiente punto, que es instalar el MinGW para compilar el código fuente de ZEsarUX y obtener el fichero .exe del emulador.

## Paso 2 – Instalación y configuración del MinGW

Una vez descargado el fichero mingw-get-setup.exe de la dirección indicada, lo ejecutamos e instalamos en el directorio **C:\MinGW** con todas las opciones por defecto. Al terminar la instalación, se nos abrirá el **MinGW Installation Manager** donde tendremos que seleccionar los siguientes paquetes para instalar:

En **Basic Setup**:

- Mingw-developer-toolkit
- mingw32-base
- mingw32-gcc-g++
- msys-base

En **All packages**, deberéis añadir los siguientes paquetes:

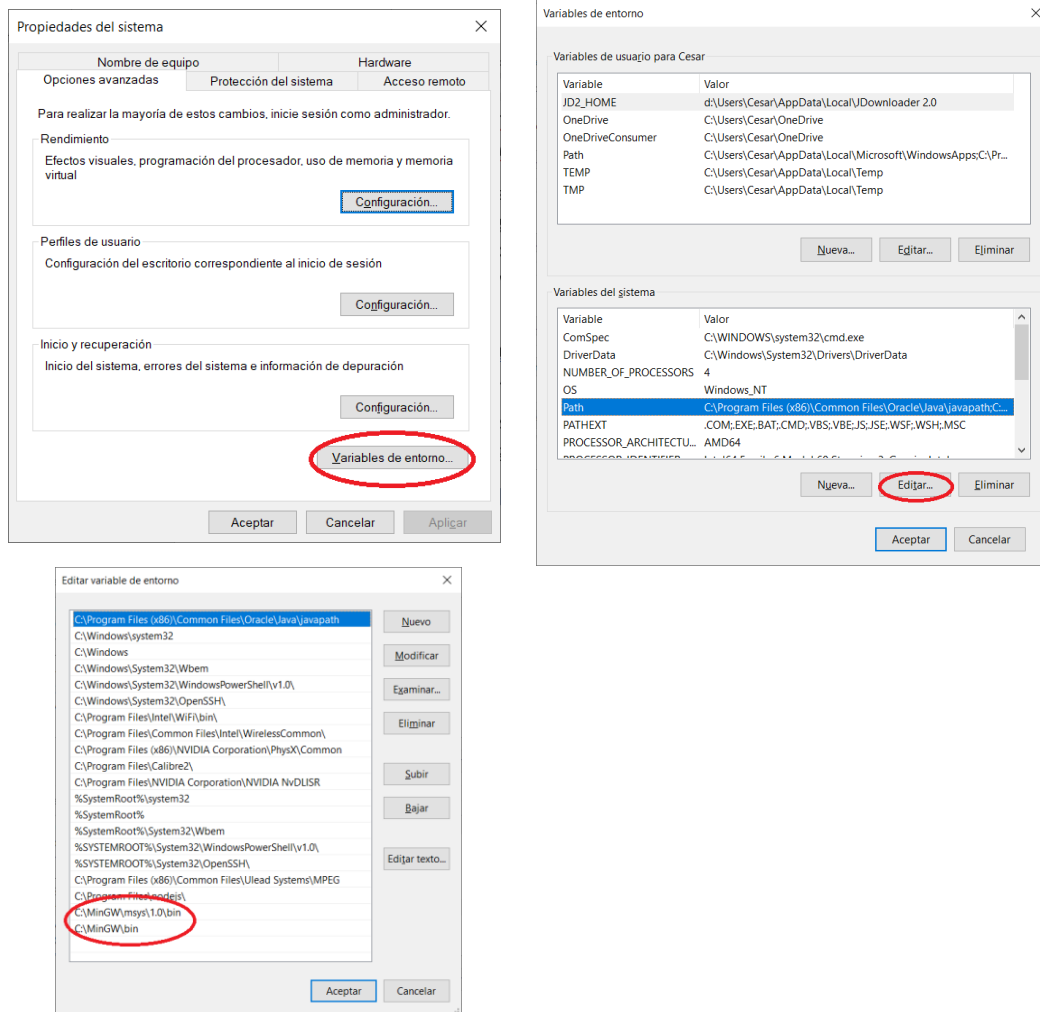
- mingw-pthreads (todos los paquetes que veáis con este nombre)
- msys-bash

Tras seleccionar estos paquetes, aplicamos los cambios y ya podemos cerrar el **MinGW Installation Manager**. Ahora tendréis que ir al explorador, copiar en **C:\MinGW** el fichero que descargamos anteriormente con las librerías SDL (**SDL-devel-1.2.15-mingw32.tar.gz**) y descomprimirlo. Esto nos creará un directorio llamado **SDL-1.2.15** que renombraremos simplemente como **sdl**.

Si todo ha ido bien, en **c:\mingw\sdl\lib** tendremos varios ficheros **libSDL.dll\***, y en **c:\mingw\sdl\include\** tendremos una subcarpeta **SDL** en cuyo interior habrá varios ficheros **.h**.

Nuestro último paso antes de compilar el fichero .exe será añadir al PATH del sistema los directorios **c:\mingw\bin** y **c:\mingw\msys\1.0\bin**.

## Opción a) desde el panel de control:



## Opción a) desde la línea de comandos:

Ejecutamos cmd.exe y escribimos **set PATH=%PATH%;c:\mingw\bin;**  
**c:\mingw\msys\1.0\bin**

### Paso 3 – Compilación y configuración de ZEsarUX

Ya estamos preparados para compilar nuestro ZEsarUX. Para ello, tendremos que ejecutar cmd.exe, cambiarnos al directorio donde hayamos descomprimido el código fuente de ZEsarUX (en mi caso d:\spectrum\ZesarUX\src) y ejecutar el comando:

**bash**

se nos mostrará un prompt "bash.3.1\$" desde el que lanzaremos el siguiente comando

**./configure --enable-memptr --enable-visualmem --enable-cpustats**



```
Símbolo del sistema - bash
D:\Spectrum\Zesarux>cd src
D:\Spectrum\Zesarux\src>bash
bash-3.1$ ./configure --enable-memptr --enable-visualmem --enable-cpustats

Configuration script for ZesarUX

Initial CFLAGS=
Initial LDFLAGS=
Checking Operating system ... Msys
Checking for gcc compiler ... /mingw/bin/gcc.exe
Checking size of char ... 1
Checking size of short ... 2
Checking size of int ... 4
Checking Little Endian System ... ok
Checking for stdout functions ... not found
Checking for simpletext functions ... found
Checking for fbdev functions ... not found
Checking for cursesw libraries ... not found
Checking for curses libraries ... not found
Checking for aa libraries ... not found
Checking for caca libraries ... not found
Checking for SSL libraries ... disabled
Checking for xwindows libraries ... not found
Checking for xwindows extensions ... disabled
Checking for xwindows vidmode extensions ... disabled
Checking for posix threads ... found
Checking for realtime schedulling ... not found
Checking for audio dsp ... not found
Checking for audio alsa ... not found
Checking for audio pulse ... not found
Checking for coreaudio ... not found
Checking for Cocoa Mac OS X GUI ... not found
Checking for sdl libraries ... found
Checking for libsndfile ... not found
Checking for linux real joystick ... not found

Final CFLAGS= -Wall -Wextra -fsigned-char -DMINGW -I/c/mingw/SDL/include
Final LDFLAGS= -lwinmm -lpthread -lwsock32 -L/c/mingw/SDL/lib -lSDL
Creating Makefile
```

Al terminar, ejecutaremos **make clean** y finalmente **make**, con lo que se lanzará la compilación. Salimos de bash tecleando **exit** y, si todo ha ido bien, tendremos en nuestro directorio src el flamante fichero **zesarux.exe** con el ejecutable del emulador.

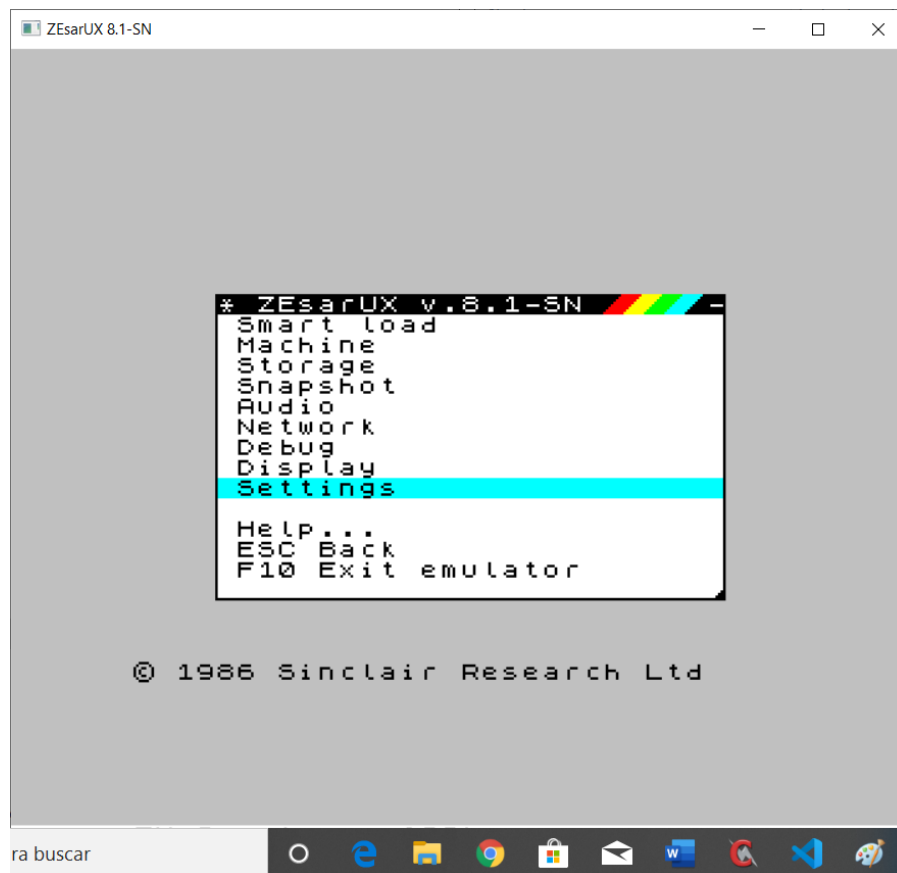
```
Símbolo del sistema
D:\Spectrum\Zesarux\src>make clean
rm -f *.o zesarux zesarux.exe smpatap sp_z80 tapabin bin_sprite_to_c leezx81 file_to_eprom bmp_to_prism_4_planar bmp_to_sprite spdetotxt install.sh
rm -fR bintargztemp/ sourcetargztemp/ ZEsarUX_win-8.1/
rm -fR macos/zesarux.app
rm -fR macos/zesarux.dmg
rm -fR macos/zesarux.dmg.gz
rm -fR macos/ZEsarUX_macos*.gz
rm -f ZEsarUX_bin-*.tar.gz
rm -f ZEsarUX_src-*.tar.gz
rm -f ZEsarUX_win-*.zip
rm -f ZEsarUX_extras-*.zip

D:\Spectrum\Zesarux\src>make
gcc -Wall -Wextra -fsigned-char -DMINGW -I/c/mingw/SDL/include -c charset.c
gcc -Wall -Wextra -fsigned-char -DMINGW -I/c/mingw/SDL/include -c scrsimpletext.c
gcc -Wall -Wextra -fsigned-char -DMINGW -I/c/mingw/SDL/include -c scrsdl.c
scrsdl.c: In function 'realjoystick_sdl_init':
scrsdl.c:1561:39: warning: passing argument 1 of 'menu_tape_settings_trunc_name' discards 'const' qualifier from pointer target type [-Wdiscarded-qualifiers]
    menu_tape_settings_trunc_name(SDL_JoystickName(0),realjoystick_joy_name,REALJOYSTICK_MAX_NAME);
                                     ^~~~~~
In file included from utils.h:27:0,
                  from z88.h:32,
                  from screen.h:26,
                  from scrsdl.c:38:
menu.h:532:13: note: expected 'char *' but argument is of type 'const char *'
extern void menu_tape_settings_trunc_name(char *orig,char *dest,int max);
                                     ^~~~~~
```

Nuestro último paso será copiar la dll **SDL.dll** desde MinGW al directorio actual con el comando:

**copy c:\mingw\sdl\bin\SDL.dll .**

Hecho esto... ¡¡¡¡Ya podemos ejecutar nuestro flamante ZesarUX.exe y terminar de configurarlo!!!! Nada más abrirlo, al hacer click con el ratón, veremos el siguiente menú:



Aquí seleccionaremos **settings**, en el siguiente menú **debug** y aquí marcaremos la opción **ZRCP Remote Protocol**, estableciendo el puerto por el que se comunicará ZesarUX con el plugin (por defecto, el 10000, aunque en mi caso me daba algún problema y lo tengo configurado por el 12000)

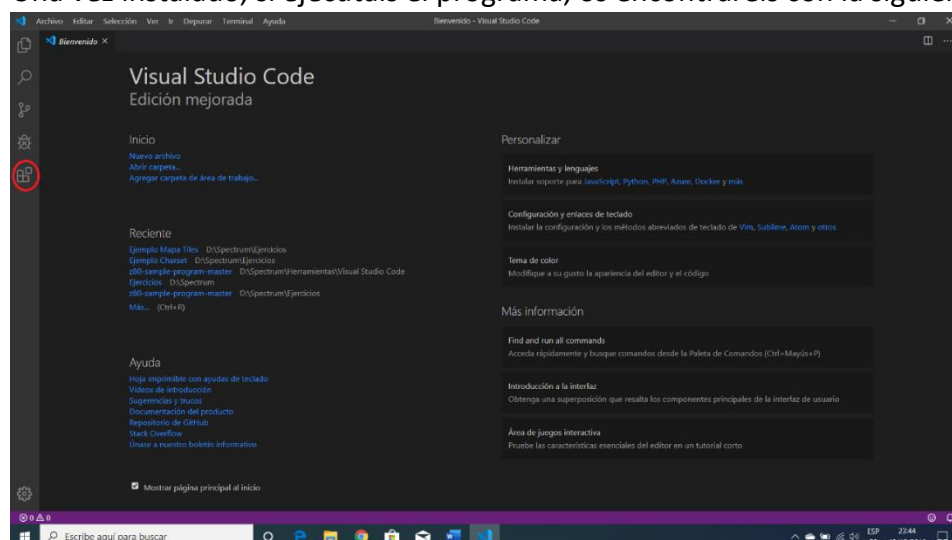


ZEsarUX tiene cientos de aspectos interesantes y configurables que podéis ver navegando desde sus menús. Para poder depurar, no necesitamos configurar nada más, y aunque no es objeto de este tutorial, desde aquí os animo a “bichear” con las opciones para que podáis ver toda la potencia de este magnífico emulador. Seguro que mi tocayo estará encantado de que trasteéis y le saquéis todo el partido posible a las opciones de su emulador.

#### 4.- Instalación y configuración del VS Code

Una vez descargado el fichero ejecutable del instalador del VS Code desde la dirección indicada en el apartado 2 del documento, bastará con ejecutarlo y seguir las instrucciones, aceptando las opciones por defecto a la hora de instalarlo.

Una vez instalado, si ejecutáis el programa, os encontraréis con la siguiente ventana:

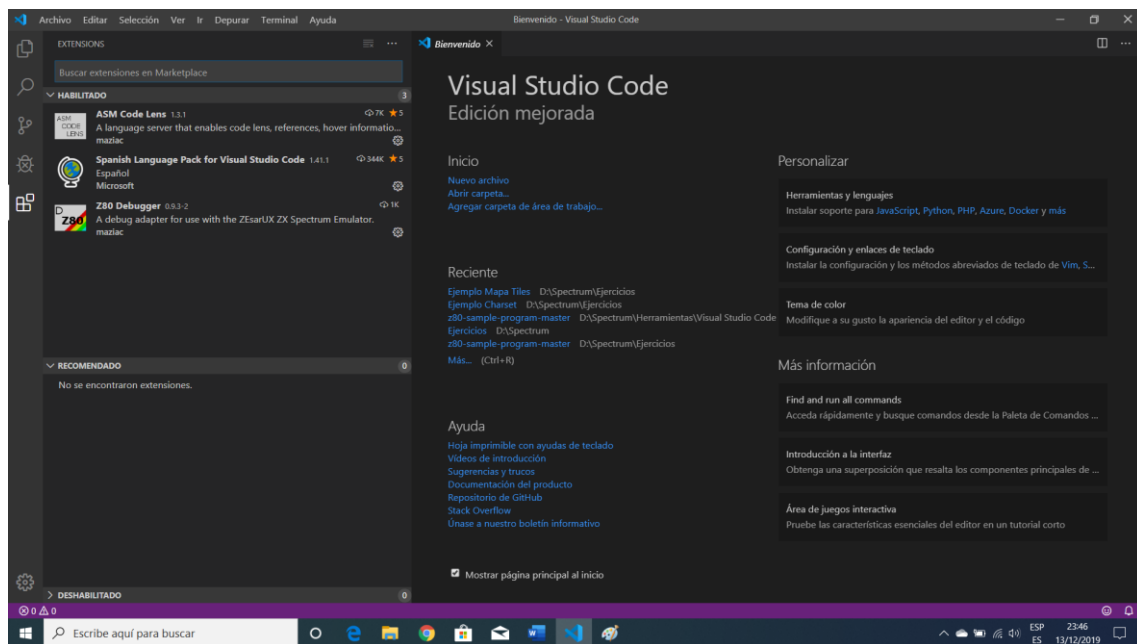


La vuestra no será exactamente igual porque el sistema estará en inglés, pero ahora iremos con ello.

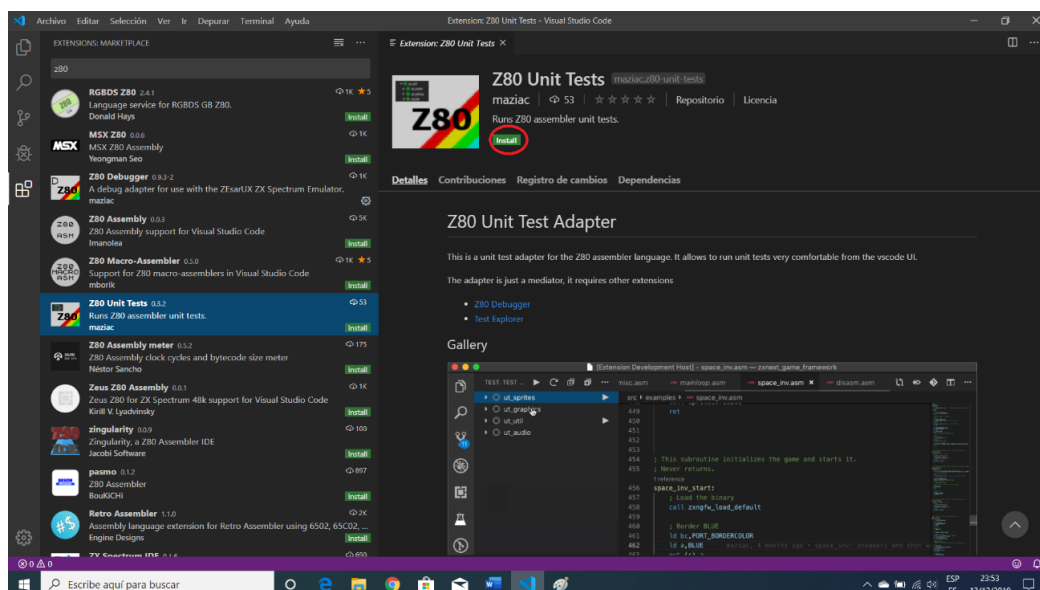
Un detalle más: aunque no recuerdo si lo hacía el instalador de forma automática, os aconsejo comprobar que se ha añadido al path del sistema el subdirectorio **bin** de la carpeta donde hayáis instalado el VS Code (en mi caso d:\Microsoft VS Code\bin)

## Paso 1 – Instalación de plugins básicos

Lo primero que haremos será instalar los plugins adicionales básicos. Para ello, pincharemos sobre el icono que os he marcado en un círculo rojo, que nos llevará a la pantalla de instalación de extensiones:



Para instalar un plugin basta con buscarlo, pinchar sobre él y en la ventana de la derecha, pulsar el botón “Install”

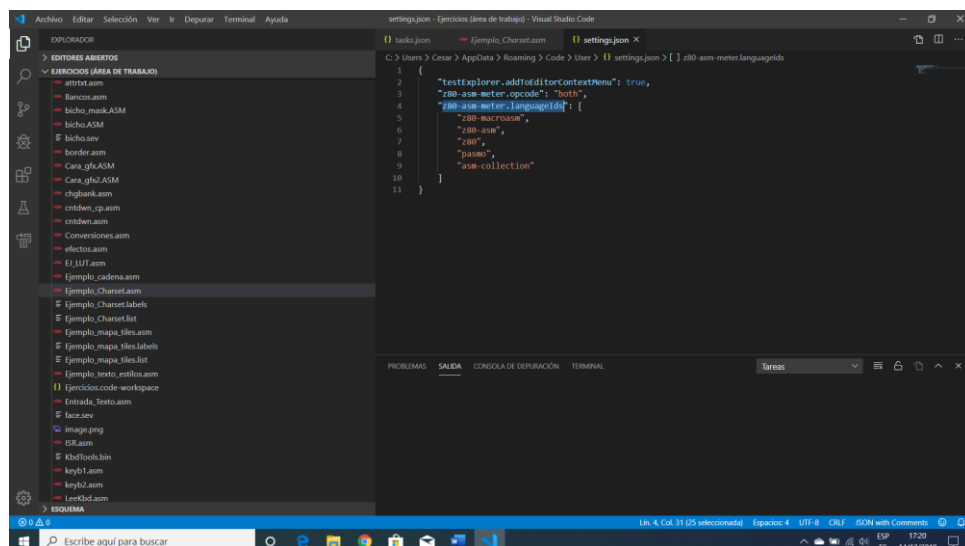


Ahora vamos a instalar los siguientes plugins complementarios a Z80 Debug:

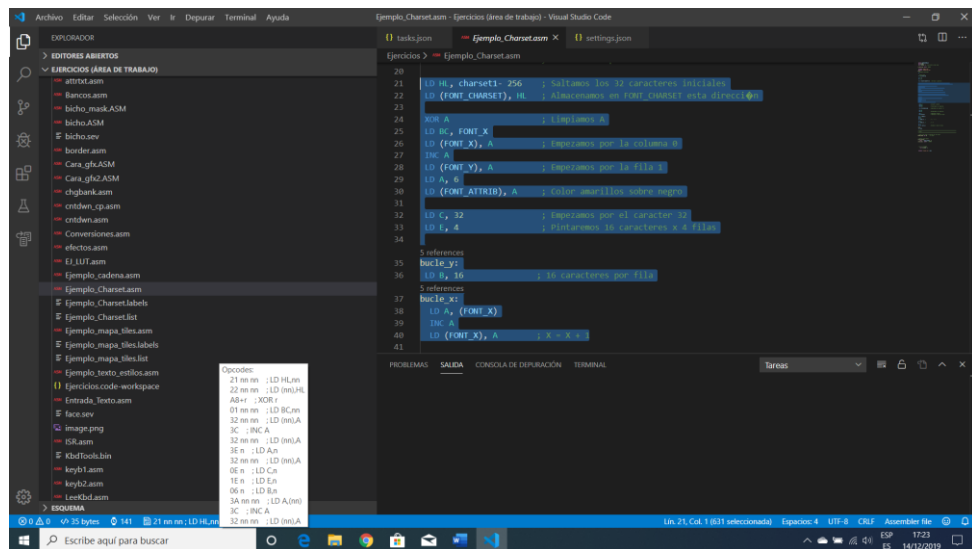
- **Spanish Language Pack for Visual Studio Code.** Os permitirá poner vuestro VS Code en español, si os sentís más cómodos trabajando en nuestra lengua materna.
- **ASM Code Lens**, que al igual que el plugin principal Z80 Debug está desarrollado por Thomas Busse (aka Maziac). Este plugin nos dará acceso a una serie de herramientas muy interesantes a la hora de depurar (localización de referencias, hovering de variables y registros, ver el número de referencias a un símbolo concreto, Assembler syntax highlighting...).
- **Z80 Unit Tests.** También de nuestro amigo Maziac. Este plugin nos permite incluir en nuestros fuentes ASM ciertos tests unitarios para realizar comprobaciones en tiempo de depuración, deteniendo la ejecución si el test no se cumple. Tengo pendiente echarle un ojo tranquilamente y ver qué se puede hacer con él, pero tiene una pinta estupenda. Si veo que merece la pena, prometo hacer una ampliación a este tutorial con su uso (si alguien no lo hace antes por mí ;).
- **Z80 Assembly Meter:** Magnífico plugin de Néstor Sancho, que determina el número de ciclos de reloj que consumirán las instrucciones que seleccionemos desde el editor, así como el tamaño en bytcodes de las mismas.

Para que este plugin funcione, deberéis hacer un pequeño ajuste:

- Abrimos la consola de comandos con **Ctrl+Shift+P**
- Seleccionamos **Preferencias Abrir Configuración JSON** (Preferences Open Settings JSON).
- En la entrada **z80-asm\_meter-languageIds** añadimos, **"asm-collection"**



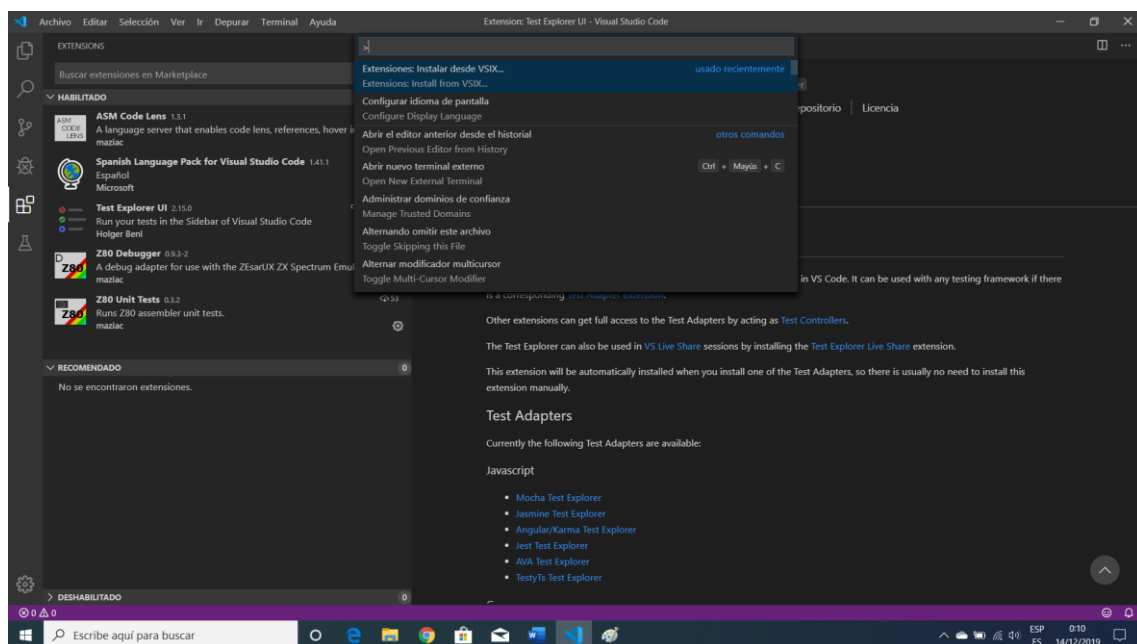
Ahora, cada vez que seleccionemos una o varias líneas de código, la barra de estado nos mostrará el tamaño en bytes, el número de ciclos de reloj que consumirá y el (o los) opcodes de las operaciones seleccionadas.



## Paso 2 – Instalación del plugin Z80 Debugger

Aunque este plugin también se puede instalar desde la pantalla de instalación de extensiones, si lo buscáis allí encontraréis la versión v.0.9.2, y lamentablemente, esta versión contiene un bug que genera muchos problemas durante la depuración, así que tendréis que instalar la beta v.0.9.3-2 que me facilitó el propio Thomas Busse (y que encontraréis en el mismo zip en el que está este documento) que tiene un proceso de instalación ligeramente diferente.

El primer paso es descomprimir el fichero **z80-debug-0.9.3-2.vsix** sobre una carpeta de trabajo. Ahora, desde VS Code, nos iremos al menú “Ver” y allí seleccionaremos la opción **Paleta de comandos**.



Al hacerlo, se abrirá una ventana donde deberemos buscar “**Extensiones: instalar desde fichero VSIX**”. Ahora seleccionaremos el fichero **z80-debug-0.9.3-2.vsix**,

aceptaremos y veremos cómo nuestro plugin queda perfectamente instalado. Ya estamos terminando.

### Paso 3 – Instalación de node.js y el ensamblador SJASMPLUS

Las dos últimas herramientas que tendremos que instalar son el node.js y SJASMPLUS. Para el primero de ellos, bastará con ejecutar el fichero **node-v12.13.1-x64.msi** que habremos descargado, ejecutarlo e instalar con las opciones por defecto.

La instalación de SJASMPLUS es aún más sencilla. Tras descargar el fichero sjasmpplus-1.14.3.win.zip de la dirección indicada, sólo tenéis que descomprimirlo sobre el directorio que queráis (en mi caso d:\Spectrum\herramientas\sjasmpplus-1.14.3.win) y añadir al path del sistema dicho directorio tal como explicamos durante el paso de instalación del MinGW.

Realizado este último paso, ya podemos abrir nuestro VS Code y prepararnos para empezar a depurar. ¡¡Vamos al lío!!

## 5.- Creando nuestro IDE integrado para depurar

### 5.1 – Abriendo la carpeta con nuestro programa de ejemplo

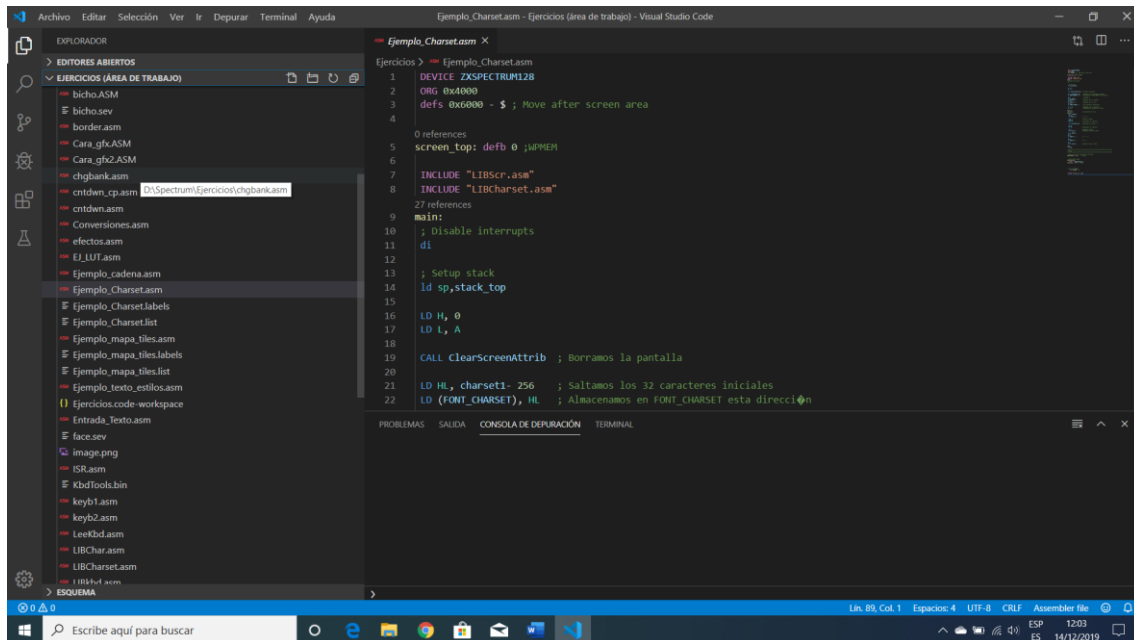
Para ilustrar la depuración con nuestro nuevo IDE, usaré un par de programas de ejemplo incluidos en el curso de **Compiler soft**: uno muestra un charset completo y otro que pinta el mapa de un nivel de su programa sokoban. Podéis encontrar los fuentes (y ficheros **tasks.json** y **launch.json**) en el fichero **Ejemplos.rar** que anexo al presente tutorial.

Al descomprimir el fichero, encontraréis el siguiente contenido:

- **LIBSprites.asm**: Librería con rutinas para el manejo de sprites.
- **LIBScr.asm**: Librería con rutinas para el manejo de pantalla.
- **LIBCharset.asm**: Librería con rutinas para el manejo de charsets.
- **Ejemplo\_Charset.asm**: Fichero con la rutina principal del primer ejemplo
- **Ejemplo\_charset.list y .labels**: generados durante el ensamblado y necesarios para la depuración
- **Ejemplo\_mapa\_tiles.asm**: Fichero con la rutina principal del segundo ejemplo
- **Ejemplo\_mapa\_tiles.list y .labels**: generados durante el ensamblado y necesarios para la depuración
- **Carpeta .vscode** que contiene los ficheros **tasks.json** y **launch.json**, necesarios para lanzar el ensamblado desde VS Code y comenzar la depuración respectivamente.

Descomprimos el zip en una carpeta de trabajo y arrancamos el VS Code. Lo ideal es generar una nueva área de trabajo (opción **Archivo\Guardar área de trabajo como...**) e incluir en la misma la carpeta donde hayamos descargado el código, desde la opción

archivo\agregar carpeta al espacio de trabajo. Si lo hemos hecho bien, veremos una pantalla similar a esta:



En la barra que tenemos a la izquierda, utilizaremos principalmente dos opciones:

- Explorador (primer icono empezando por arriba) para navegar por los ficheros.
- Debug and run (icono del “bicho”) para lanzar la depuración

## 5.2 – Ficheros tasks.json y launch.json

En todos nuestros proyectos, deberemos tener un directorio llamado **.vscode** colgando del directorio raíz donde tendremos dos ficheros muy importantes: el **tasks.json** y el **launch.json**. Aunque se pueden crear por otros medios, mi consejo es que tengáis una copia maestra de este directorio y para cada proyecto, la copiéis y modifiquéis los ficheros como sea oportuno, ya que es la opción más cómoda.

Vamos a hablar ahora de estos ficheros, para qué sirven y cómo se usan.

### FICHERO TASKS.JSON

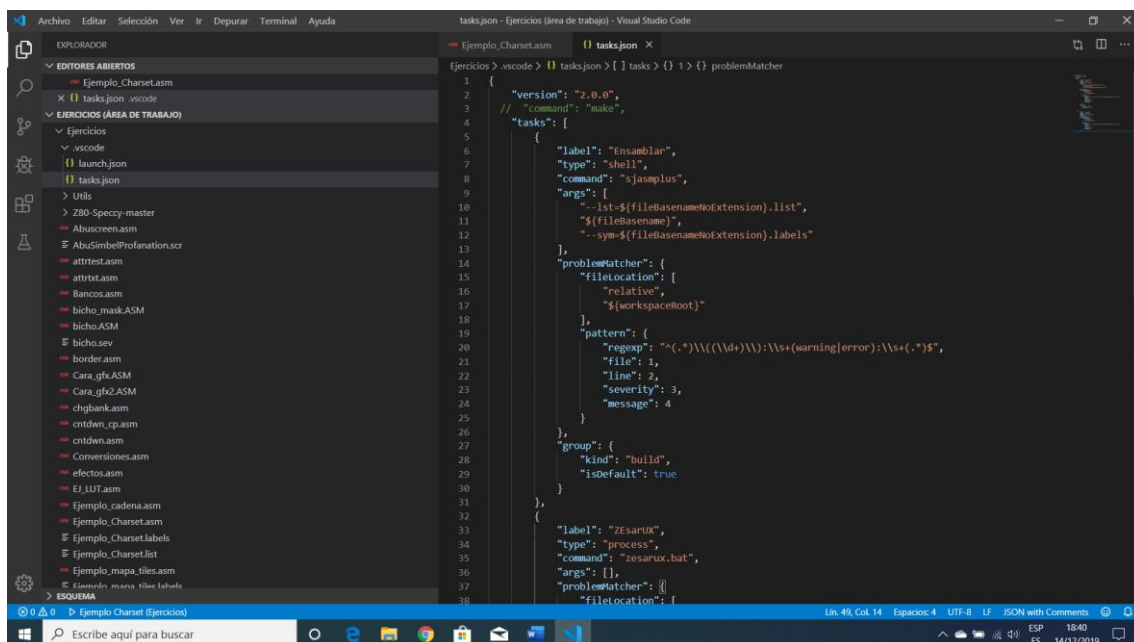
A través de este fichero, podremos definir comandos o tareas que podremos lanzar desde VS Code. En nuestro caso, vamos a crear dos tareas:

- Una que invocará al **sjasmplus** para ensamblar nuestro código y generará los ficheros **.lst** y **.labels** que necesita el debugger para la depuración paso a paso.
- Otra para llamar a ZEsarUX antes de lanzar la depuración. Para este segundo caso, nos crearemos un fichero llamado **zesarux.bat** que se posicionará en el directorio donde esté ZEsarUX y lo ejecutará. En mi caso, este fichero contiene los siguientes comandos:

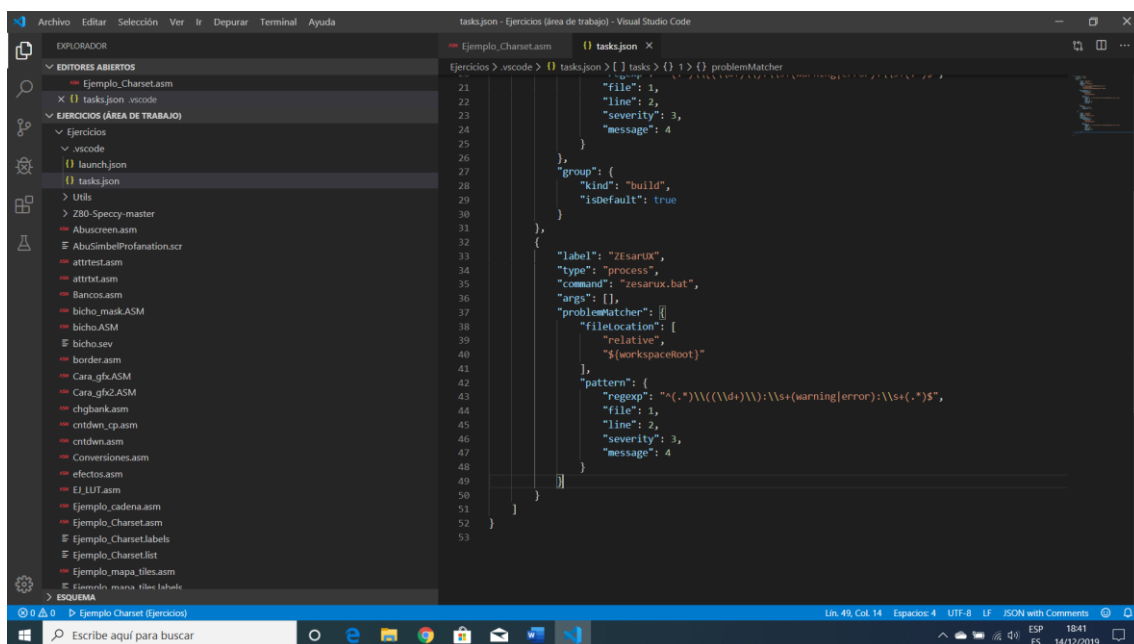


```
d:
cd \
cd spectrum
cd zesarux
cd src
zesarux
exit
```

Aunque podemos definir más tareas, de entrada nuestro fichero tasks.json tendrá la siguiente estructura:



```
{
  "version": "2.0.0",
  "command": "make",
  "tasks": [
    {
      "label": "Ensamblar",
      "type": "shell",
      "command": "sjasplus",
      "args": [
        "-lst-${fileBasenameNoExtension}.list",
        "${fileBasename}",
        "-sym-${fileBasenameNoExtension}.labels"
      ],
      "problemMatcher": {
        "fileLocation": [
          "relative",
          "${workspaceRoot}"
        ],
        "pattern": {
          "regexp": "^(.*)\\((\\d+\\.\\d+\\.\\d+)(warning|error):\\s+(.*)$",
          "file": 1,
          "line": 2,
          "severity": 3,
          "message": 4
        }
      },
      "group": {
        "kind": "build",
        "isDefault": true
      }
    },
    {
      "label": "ZesarUX",
      "type": "process",
      "command": "zesarux.bat",
      "args": [],
      "problemMatcher": [
        {
          "fileLocation": [
            "relative",
            "${workspaceRoot}"
          ],
          "pattern": {
            "regexp": "^(.*)\\((\\d+\\.\\d+\\.\\d+)(warning|error):\\s+(.*)$",
            "file": 1,
            "line": 2,
            "severity": 3,
            "message": 4
          }
        }
      ]
    }
  ]
}
```



```
{
  "version": "2.0.0",
  "command": "make",
  "tasks": [
    {
      "label": "Ensamblar",
      "type": "shell",
      "command": "sjasplus",
      "args": [
        "-lst-${fileBasenameNoExtension}.list",
        "${fileBasename}",
        "-sym-${fileBasenameNoExtension}.labels"
      ],
      "problemMatcher": {
        "fileLocation": [
          "relative",
          "${workspaceRoot}"
        ],
        "pattern": {
          "regexp": "^(.*)\\((\\d+\\.\\d+\\.\\d+)(warning|error):\\s+(.*)$",
          "file": 1,
          "line": 2,
          "severity": 3,
          "message": 4
        }
      },
      "group": {
        "kind": "build",
        "isDefault": true
      }
    },
    {
      "label": "ZesarUX",
      "type": "process",
      "command": "zesarux.bat",
      "args": [],
      "problemMatcher": [
        {
          "fileLocation": [
            "relative",
            "${workspaceRoot}"
          ],
          "pattern": {
            "regexp": "^(.*)\\((\\d+\\.\\d+\\.\\d+)(warning|error):\\s+(.*)$",
            "file": 1,
            "line": 2,
            "severity": 3,
            "message": 4
          }
        }
      ]
    }
  ]
}
```

Vamos a ver los principales tags de este fichero:

- **Label:** Contendrá el nombre con el que veremos la tarea si tratamos de ejecutarla desde la opción **terminal\ejecutar tarea...**
- **Type:** Aquí indicamos que la tarea se ejecutará sobre una Shell
- **Command:** Nombre del comando que lanzaremos desde la Shell, en nuestro caso, **sjasmplus**.
- **Args:** contendrá una lista con los diferentes parámetros que se pasarán al comando que se ejecutará desde la Shell. En nuestro caso serán el nombre del fichero **.list** a generar (nuestro archivo principal sin su extensión, añadiéndole **.list**), el fichero a ensamblar y el nombre del fichero de etiquetas a generar (de nuevo, el nombre de nuestro archivo principal sin extensión añadiéndole **.labels**):

```

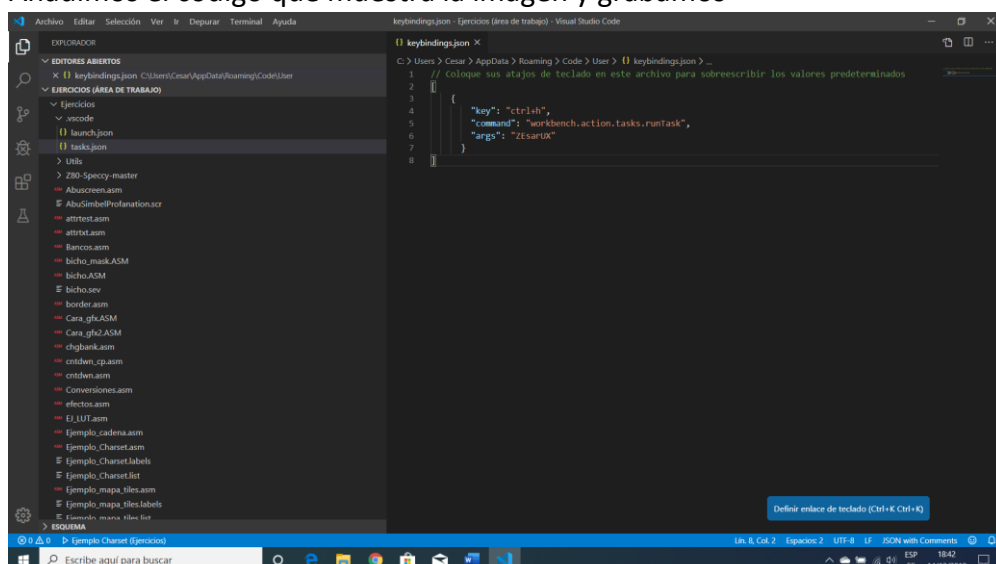
•      "--lst=${fileBasenameNoExtension}.list",
•      "${fileBasename}",
•      "--sym=${fileBasenameNoExtension}.labels"

```

- El resto de tags son estándar. Entre ellos destacar **group**, donde estableceremos que nuestra tarea es una tarea de compilación y que además será la tarea por defecto, para poder lanzarla pulsando **mayus+ctrl+b**

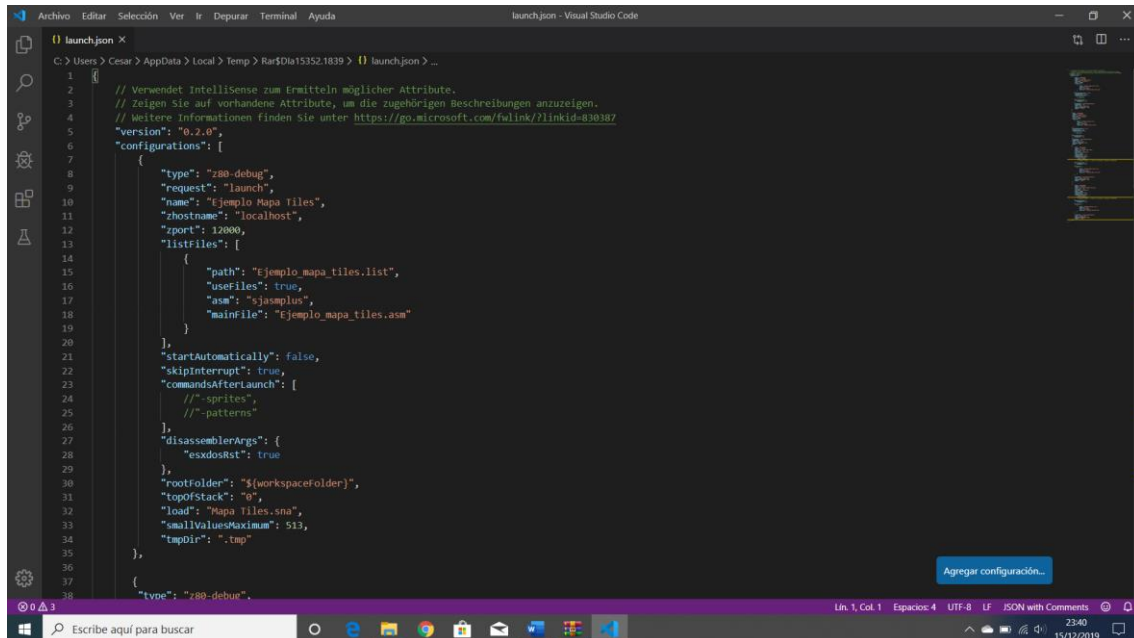
En el fichero tasks que se muestra en la imagen, creamos las dos tareas: una para invocar el ensamblador que se ejecutará con **mayus+ctrl+b** y otra para llamar a ZEsarUX . Para lanzarlo, vamos a asignar una combinación de teclas: **ctrl+z**:

- Desde el menú **Ver\Paleta de comandos...** seleccionamos **Abrir métodos abreviados de teclado (JSON)**
- Añadimos el código que muestra la imagen y grabamos



## FICHERO LAUNCH.JSON

Es el fichero que utiliza el plugin para lanzar la depuración. Este fichero deberá tener una entrada para cada programa que queramos depurar. Aunque hay información detallada en la página del plugin sobre este fichero, os paso unas explicaciones básicas que serán suficientes para echarlo a andar:



Después del tag inicial de versión, hay un tag llamado **“configurations”** donde deberemos tener una entrada por cada programa que queramos depurar. La estructura de cada bloque es la siguiente:

```
{  
  "type": "z80-debug",  
  "request": "launch",  
  "name": "Ejemplo Mapa Tiles",  
  "zhostname": "localhost",  
  "zport": 12000,  
  "listFiles": [  
    {  
      "path": "Ejemplo_mapa_tiles.list",  
      "useFiles": true,  
      "asm": "sjasplus",  
      "mainFile": "Ejemplo_mapa_tiles.asm"  
    }  
  ],  
  "startAutomatically": false,  
  "skipInterrupt": true,  
  "commandsAfterLaunch": [  
    //"-sprites",  
    //"-patterns"  
  ],  
  "disassemblerArgs": {
```

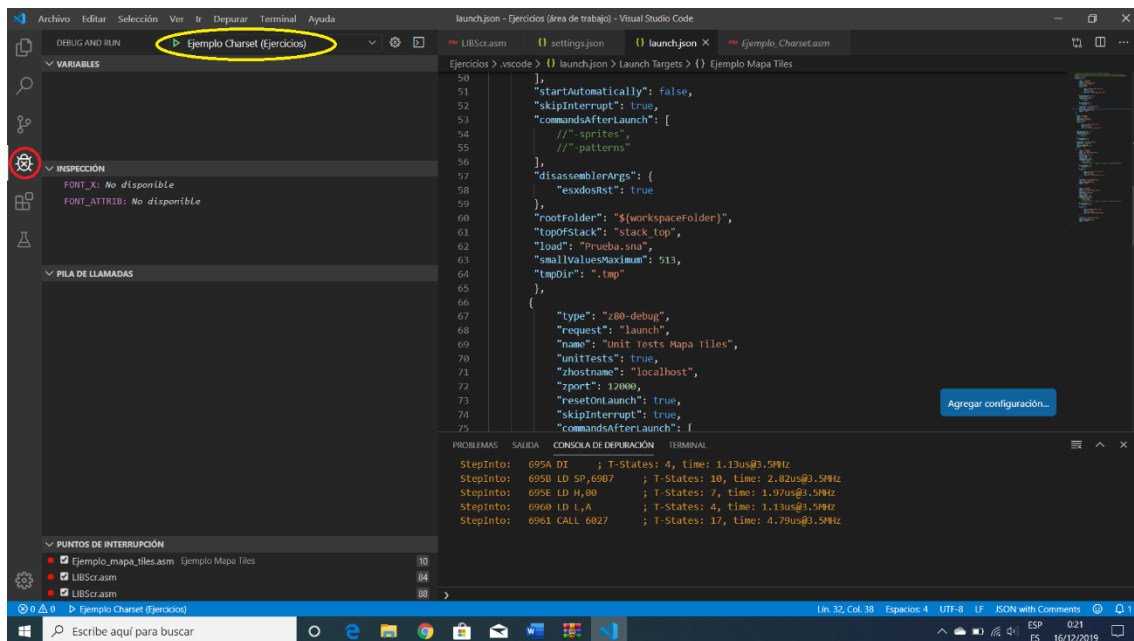
```
        "esxdosRst": true
    },
    "rootFolder": "${workspaceFolder}",
    "topOfStack": "stack_top",
    "load": "Mapa Tiles.sna",
    "smallValuesMaximum": 513,
    "tmpDir": ".tmp"
},
```

Para cada nueva entrada, tendremos que hacer las siguientes modificaciones:

- En el tag **"name"** pondremos el nombre con que queremos que aparezca en la lista de tareas de depuración (que veremos más adelante).
- A no ser que estemos depurando contra un servidor remoto, en **"zhostname"** dejaremos siempre **"localhost"**.
- En **"zport"** pondremos el número de puerto que configuramos en ZEsarUX para comunicarnos a través del protocolo ZRCP (en mi caso, si recordáis, 12000).
- En la estructura **"listfiles"** sólo hay que cambiar los nombres de las entradas **"path"** (para que apunte al fichero .list que generaremos con sjasmplus) y **"mainfile"**, que contendrá el nombre del fichero principal a depurar.
- En el tag **"topOfStack"** pondremos **"stack\_top"**, que es una variable que pondremos en nuestro ASM, con el código que explico más adelante, para que el plugin pueda mostrar correctamente la pila de llamadas. (OJO: En el fichero de ejemplo que acompaña al tutorial, por error viene un **"0"**. Debéis cambiar este valor por **"stack\_top"** si queréis que funcione correctamente la pila de llamadas)-
- Del resto de la estructura, sólo tenéis que cambiar el tag **"load"** con el nombre del fichero .SNA que generaremos al ensamblar.

Si miráis el **launch.json** del ejemplo que acompaña al tutorial, veréis que tiene realmente cuatro entradas en **"configurations"**: las dos primeras son las necesarias para compilar los programas de ejemplo; las otras dos son las configuraciones necesarias para llamar a los unit tests, que están copiadas del ejemplo que acompaña al plugin, pero no he sabido terminar de configurarlas aún así que podéis ignorarlas.

Una vez listo el fichero **launch.json**, podemos irnos a la ventana de depurar, pulsando la opción correspondiente en la barra de la izquierda (marcada con un círculo rojo):

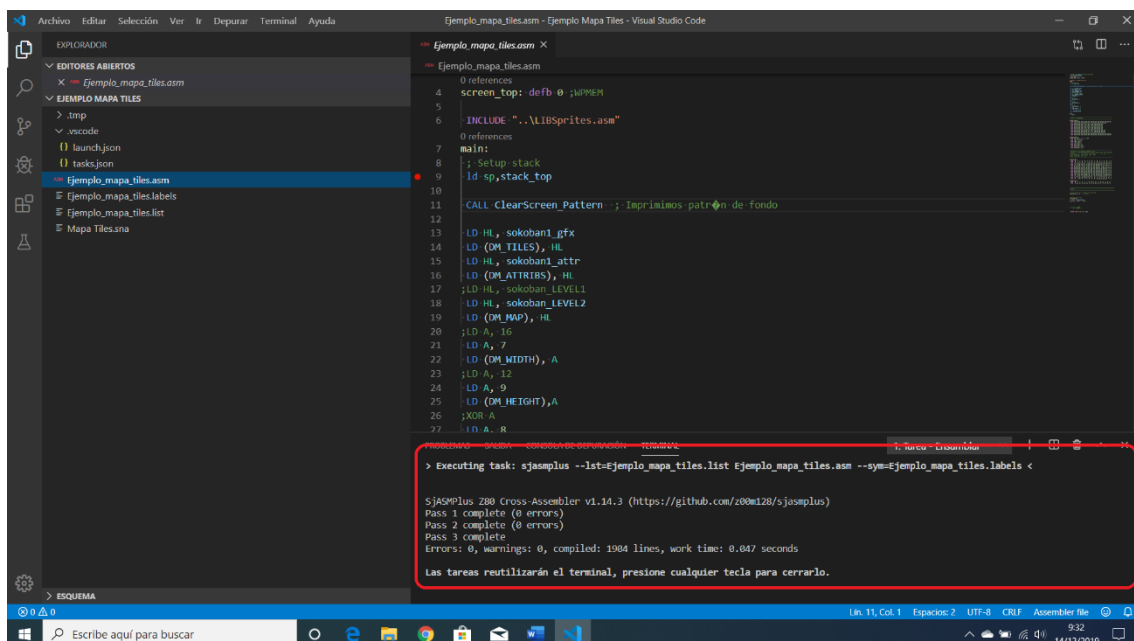


En esta pantalla, en la parte superior izquierda (marcado con un círculo amarillo), tenemos una flecha verde, que permite lanzar la depuración y a su derecha un combo donde aparecerán todas las configuraciones del fichero **launch.json**, identificadas por el valor que pusiéramos en cada una de ellas en el tag **"name"**.

Si lanzáis desde aquí la depuración, es importante que hayáis abierto antes ZEsarUX (recordad, ctrl+z), ya que sino os dará un error de conexión fallida con el emulador.

### 5.3 – Ensamblado del código ASM con SJASMPPLUS

Para ver que todo está bien definido, desde el explorador seleccionaremos el fichero **ejemplo\_mapa\_tiles.asm** y pulsando la combinación de teclas indicadas, lanzaremos el ensamblado, cuyo resultado veremos en la consola de depuración:



Para los que vengáis del mundo PASMO, os paso algunos cambios de sintaxis necesarios entre PASMO y sjasmpplus que tendréis que hacer en vuestros programas si no queréis tener un buen montón de errores de ensamblado:

- Todas las etiquetas deberán empezar en la primera columna, sin dejar ningún espacio ya que en caso contrario se considerarán directivas o instrucciones, con el consiguiente error.
- Todas las instrucciones o directivas deberán dejar al menos un espacio para no ser consideradas etiquetas.
- Nuestro programa deberá empezar con la directiva DEVICE indicando el tipo de máquina para el que vamos a generar el fichero SNA (ZXSPECTRUM48, ZXSPECTRUM128, etc.)
- Deberéis definir una etiqueta para la rutina de entrada, que luego pasaremos a la directiva que genera el fichero SNA que usaremos para depurar.
- Definición de la pila: aunque no he profundizado lo suficiente, parece que hay que definir las zonas de comienzo y fin de la pila. Para ello es necesario añadir el siguiente código:

```
;=====
=====
; Stack.
;=====
=====

; Stack: this area is reserved for the stack
STACK_SIZE: equ 10      ; in words

; Reserve stack space
stack_bottom:
    defs    STACK_SIZE*2, 0
stack_top:   defb 0      ; WPMEM
```

Además de hacer la siguiente carga en SP al comienzo de la rutina principal:

```
; Setup stack
ld sp,stack_top
```

#### 5.4 – Lanzamiento de la sesión de depuración

Bueno, pues ha llegado el gran momento: vamos a depurar el primer ejemplo. Para ello:

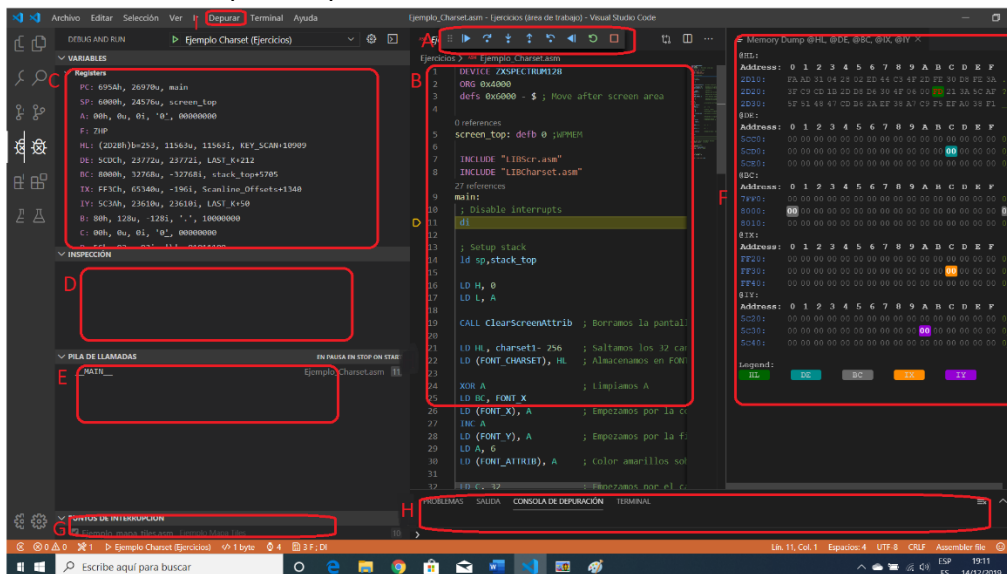
- abriremos Ejemplo\_charset.asm
- Lo ensamblaremos pulsando **ctrl+mayus+b**
- arrancamos ZEsarUX con **ctrl+z**

- Pulsamos F5 para lanzar la depuración, o nos vamos a la ventana de depuración, seleccionamos la prueba a lanzar y pulsamos la flecha verde.

## 6.- Introducción a la depuración

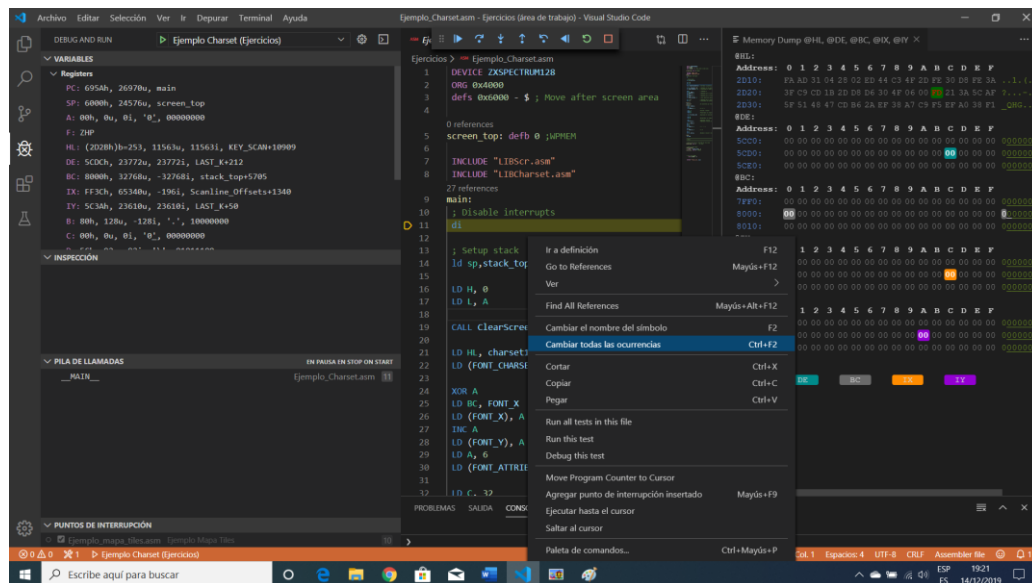
Aunque la herramienta permite opciones avanzadas de depuración (breakpoints condicionales, inspección de zonas de memoria concretas, etc.), aún no me ha dado tiempo a profundizar en ellas y poco os puedo contar en este sentido, así que tendréis que investigar un poco la documentación que mencioné al principio del documento y probar, probar y probar.

En cualquier caso, si os puedo dar un pequeño esbozo de qué herramientas básicas nos ofrece el IDE para depurar:

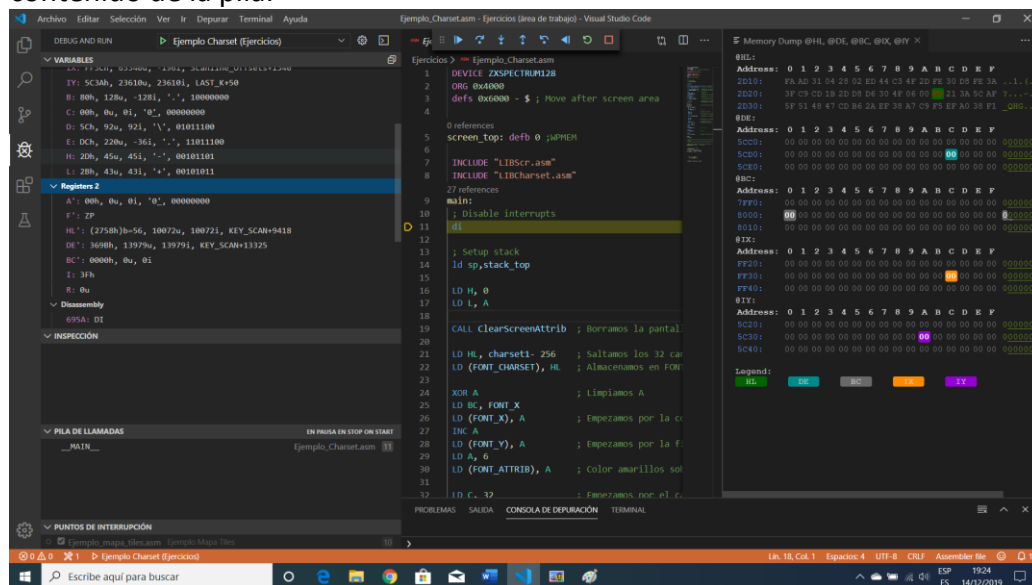


Las zonas que he marcado son:

- **A - Barra de comandos:** Ofrece las opciones habituales para depurar, como son continuar, paso a paso por procedimientos, paso a paso por instrucciones, salir de la rutina, retroceder en la ejecución, invertir, reiniciar y terminar la depuración. En la ventana de código (marcada como B) se marcarán en verde las instrucciones ejecutadas.
- **B – Zona de código:** Muestra el código que estamos ejecutando. Poniendo el cursor sobre una variable o etiqueta, podremos ver su valor y el contenido de la zona de memoria a la que apunta. Pulsando el botón derecho, podremos ver y navegar por la definición de una rutina o variable, las referencias que tiene en el código, ejecutar hasta esa posición, etc.

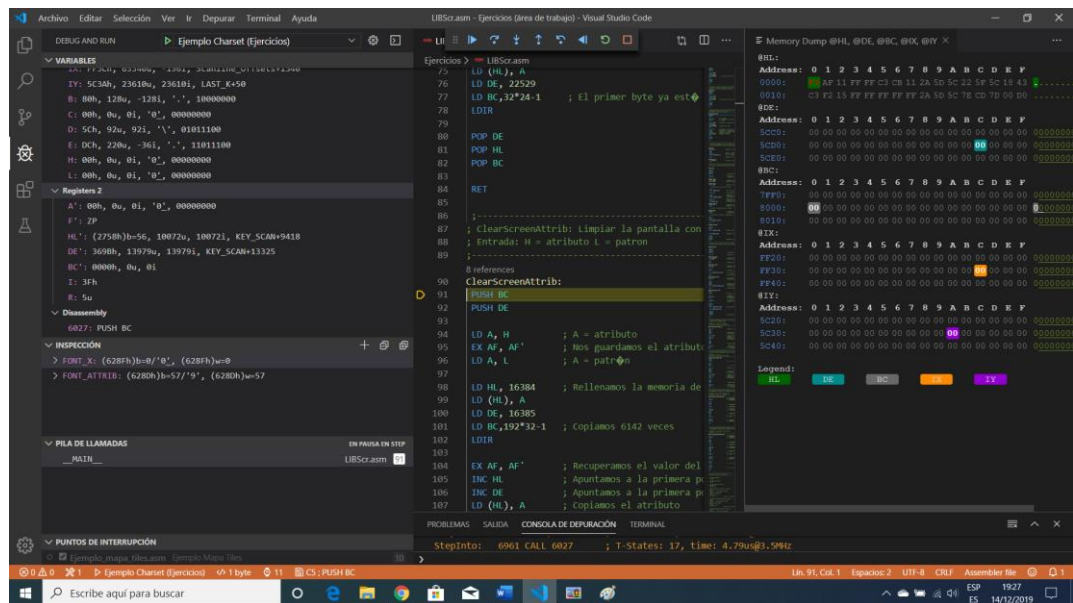


- **C – Zona de variables:** Permite ver y modificar los valores de los registros, registros shadow, ver el código desensamblado, las zonas de memoria o el contenido de la pila.

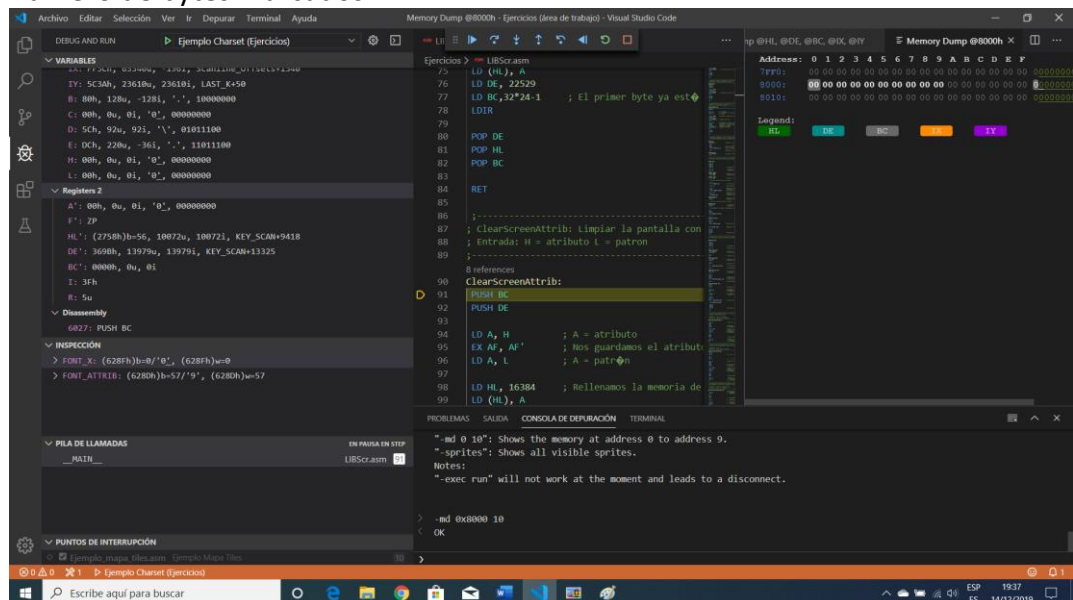


- **D – Zona de Inspección:** Permite añadir variables a inspeccionar durante la ejecución del código del programa





- **E – Pila de llamadas:** Nos indica en que punto de la ejecución nos encontramos exactamente.
- **F – Zona de inspección de memoria:** Nos permite visualizar (y cambiar) los valores contenidos en zonas de memoria concretas. Por defecto muestra las zonas apuntadas por los registros HL, DE, BC, IX e IY. Sin embargo, si desde la consola de depuración (zona H) escribimos `-md dirección bytes` (p.e. `-md 0x8000 10`), nos mostrará una nueva ventana con esa zona de memoria y ese número de bytes marcados.



- **G – Puntos de interrupción:** Permite añadir puntos de interrupción. También podemos hacerlo mediante un doble click a la izquierda de cualquier línea de código.
- **H – Consola de depuración:** Desde aquí podremos lanzar comandos específicos durante la depuración. Os recomiendo lanzar un `-help` para ver toda la lista de comandos disponibles.

- **I – Menú de depuración:** Permite acceder a las opciones principales de depuración.

## 7.- ¿Y ahora qué?

Bueno, pues llegados a este punto ya sólo os queda poneros a programar como poseos en ASM e ir descubriendo todas las posibilidades que nos ofrecen la combinación de estas herramientas.

Por mi lado, yo seguiré investigando/estudiando los siguientes aspectos:

- Sintaxis y posibilidades de **sjasmplus**
- Configuración y posibilidades adicionales de **ZEsarUX**
- Capacidades de depuración que ofrece el plugin **Z80 Debug**
- Uso de Z80 Unit Test en mis programas

Conforme vaya progresando en esas líneas, iré publicando actualizaciones al tutorial con todo lo que vaya descubriendo.

## 7.- Agradecimientos

Por último, tengo que terminar con el agradecimiento público a varias personas que me han ido ayudando en cada problema que me he encontrado en el camino y que han puesto su granito de arena para tener terminado este tutorial.

En concreto, quiero mencionar especialmente a:

- **César Hernández**, por haber hecho un emulador como ZEsarUX, que nos ofrece un mundo de posibilidades, pero sobre todo por su cercanía, cordialidad e interés en ayudarme en cada paso. ¡¡¡MUCHAS GRACIAS TOCAYO!!!
- **Thomas Busse**, por regalarnos un plugin tan brutal que exprime todas las posibilidades que permite ZEsarUX y por su amabilidad a la hora de responder mis dudas, cuestiones, problemas y, sobre todo, por facilitarme una versión específica para Windows que corregía todos los bugs que no me permitían depurar como estaba buscando.
- **Nestor Sancho**, por su simpatía y su rápida ayuda para hacer funcionar a su plugin sin colisionar con el resto de plugins que he tenido que instalar.
- **A mi compañero y amigo Alejandro Ortíz Carmona, aka Faliorn**, por haberme dado ese empujoncito que necesitaba para lanzarme a retomar mi contacto con el mundo del Spectrum. Sin nuestras charlas en los cafés y sus pedidos de material retro que siempre me enseña cuando los recibe en la oficina, este tutorial probablemente nunca hubiera llegado a escribirse. ¡¡Gracias Ale!!
- **A mis compañeros de los grupos “Ensamblador ZX Spectrum” y “Retrodevs”**. Vuestra cercanía, ayuda, comentarios y demás son una pieza clave para seguir avanzando en mi sueño de “dominar a la bestia”....

Y con esto termino, con que este tutorial sirva para que una sola persona utilice el entorno de desarrollo y le saque el máximo partido, el esfuerzo que me ha costado aprender (y posteriormente redactar) todo lo necesario para montarlo habrá merecido la pena.

Me gustaría que, si usáis este tutorial, contactéis conmigo por correo o por telegram (mi Nick es @Metaprime) para compartir dudas, problemas, avances y demás.

¡Nos vemos en los foros!

César Wagener Moriana, en Sevilla a 14 de diciembre de 2019.

**Randomize usr 0**