

## Content

<b>1. Prologo .....</b>	<b>two</b>
<b>2. What you need to mount the IDE .....</b>	<b>3</b>
<b>3. Installing the emulator ZEsarUX .....</b>	<b>6</b>
Step 1 - Download the source code ZEsarUX .....	6
Step 2 - Installation and configuration of the MinGW .....	7
Step 3 - Compilation and configuration ZEsarUX .....	8
<b>4. Installing and configuring the VS Code .....</b>	<b>eleven</b>
Step 1 - Installing basic plugins .....	12
Step 2 - Installing the plugin Z80 Debugger .....	14
Step 3 - Install node.js and assembler SJASMPLUS .....	fifteen
<b>6. Creating our integrated IDE to debug .....</b>	<b>fifteen</b>
6.1 - Open the folder with our sample program .....	fifteen
6.2 - Files tasks.json and launch.json .....	16
6.2 - Assembly of the ASM code SJASMPLUS .....	19
6.3 - Launch of the debugging session .....	twenty
<b>6. Introduction to debug .....</b>	<b>twenty</b>
<b>7. Now what? .....</b>	<b>2. 3</b>
<b>7. Acknowledgments .....</b>	<b>2. 3</b>

## 1. Prologo

I'm getting old.

In a few days I fall 48 chestnuts and, as happens when you get old, this summer during the holidays I got to look back and review those things that have been happening to me throughout my life and how I have influenced to become who I am.

And I realized I had to settle a historical debt.

My passion for computers started way back in the 80s when my father brought a brand new Spectrum 16K which gave him the bank with which I discovered the exciting world of video games. From there I went to do a program in Basic and already in high school, I got head with C and UNIX. When finished COU, I had only one thing in mind: to study Computer Science.

University and, after much effort 6 years old, I got my new title of Computer Engineer, thanks to which I make my living today, making information systems for hospitals.

However, there is something I never did and was a thorn that had stuck. I never learned ASM Z80 (or machine code, as he called the Microhobby) and managed to take full advantage of this wonderful machine without whose entry into my life, perhaps it would be a degree in history unemployed (my other passion).

So after thinking about decided days to pay off my old debt, entering an unknown world for me, and I decided to make a game for Spectrum in ASM pure and simple, since the challenge was not so much finish a game and publish it but "dominate the beast "and learn to do what I was not able to accomplish when I was a kid.

Excitedly, I began my journey to learn ASM Z80 impressive and indispensable reading my course **Compiler Soft**, you can find in

<https://wiki.speccy.org/cursos/ensamblador/indice> , Examples typed in **context** and tested in **Spectaculator 8.0**, but I made it very difficult, because with this "IDE" debugging possibilities are not very complete.

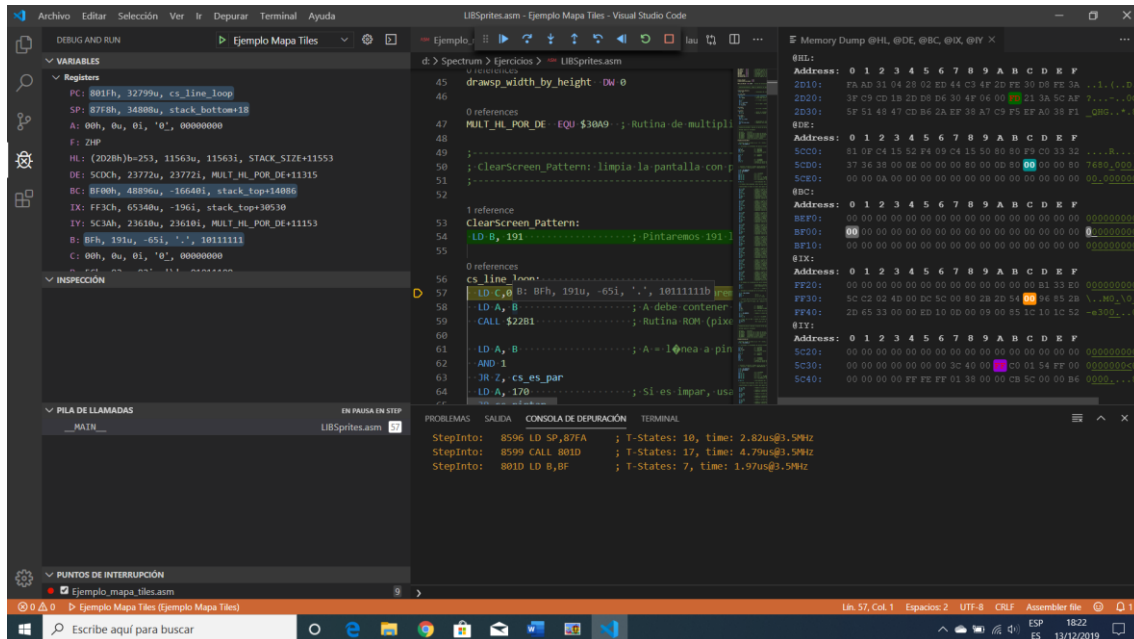
Around the summer, I went into several groups Telegram where people met a very experienced group to which raise doubts and were contributing their bit to progresase on my way. In one of these groups met César Hernández, developer of the wonderful **ZEsarUX**, Spectrum emulator spectacular and growing, offering many of the tools that he missed, but even without an IDE as integrated as was looking for.

One day, Caesar spoke in one group of a plugin for VS Code, the **Z80 Debugger**

Thomas Busse, allowing debugging assembler in a very simple way from code running on **ZEsarUX** and providing the tools they had found so far. It was what I was looking for.

Caesar told me he was working on UNIX and really had not had a beta tester of the IDE on Windows. When I contacted Thomas, the answer was the same: things got interesting ...

After weeks of tests, ranting, headaches and hundreds of emails and messages with Cesar and Thomas, and some other version they sent me correcting bugs, I got it going and ... is spectacular.



My effort would not be complete without this document that I hope will serve to other developers without having to suffer what I have suffered, can leverage the work of Caesar and Thomas with the sum of these tools so powerful to its creators dedicate so much time, effort and enthusiasm.

If the follow tutorial someone gets stuck at any point, or have any specific questions, you can contact me by sending me an email to [cesar.wagener@gmail.com](mailto:cesar.wagener@gmail.com) and, if possible, I will try to lend a hand.

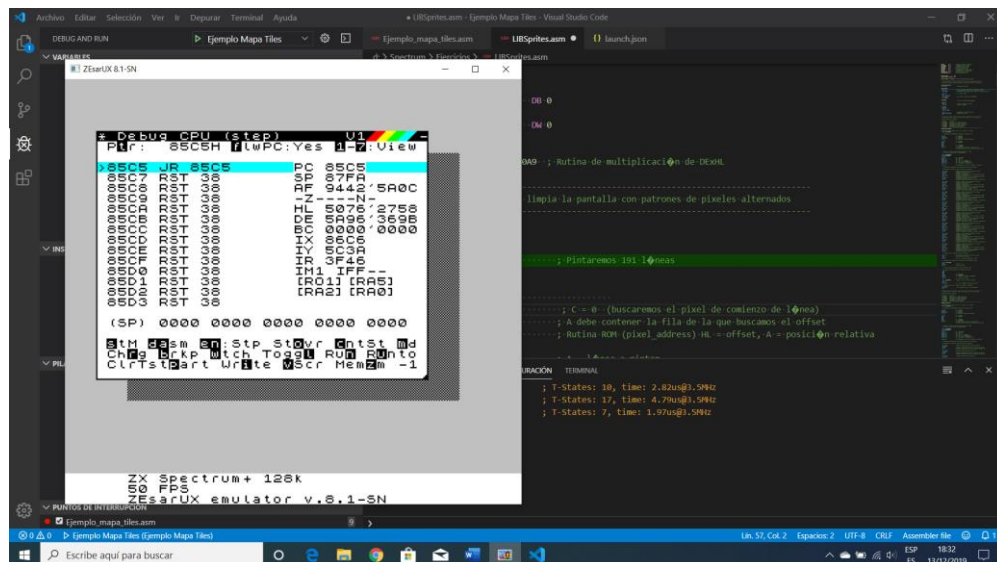
And now, let's get into job!

## 2. What you need to mount the IDE

Well, first I have to say is that everything I'll explain what I tried on a W10 Professional, but I imagine it should work without problems on other Windows systems, and even move to Linux or Mac without too much trouble.

To mount the environment, we will need the following tools:

- **ZEsarUX emulator (Emulator ZX Second And Released for Unix):** First linchpin. It is the emulator on which we will execute our code. Has many configuration options and own good debugging tools, you can offer to external systems via sockets with ZRCF protocol. You can find the source code, executable versions and documentation on it in <https://github.com/chemandezba/zesarux>



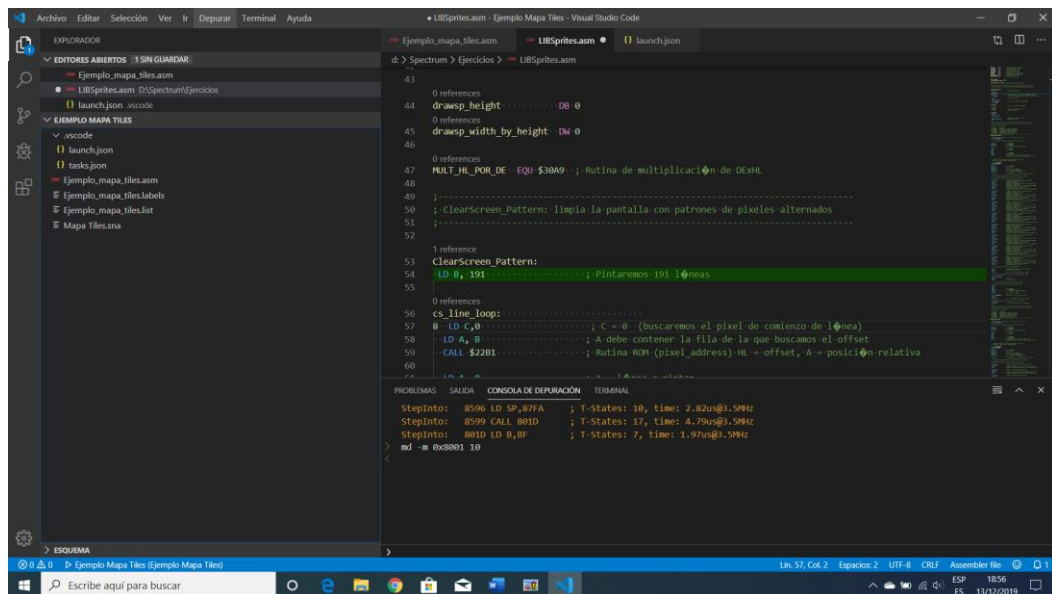
- **MinGW compiler:** I do not know if Caesar will have already uploaded a new official version, in my case I had to unburden ZEsarUX the source and compile it to access the latest developments and bug fixes released by Caesar. You can descargaros the latest version from the following link: <https://sourceforge.net/projects/mingw/> . Additionally need version 1.2.15 of the unloaded the SDL (fichero **SDL-devel-**

**1.2.15-mingw32.tar.gz** you can download it from this address:

<https://www.libsdl.org/download-1.2.php> ).

- **VS Code Editor:** Tool from which to write the code will assemble and launch debugging. It is a tool widely used by Microsoft developers, although I acknowledge that I did not know. We will use version 1.41 that you can download from this link:

<https://code.visualstudio.com/download>



We also have to get settled (if you have not already) the Node.js, which can be found here: <https://nodejs.org/es/download/>

- **Z80 Plugin Debug:** Plugin that will install on VS Code and, using the ZRCF protocol, we allow debug visually. You can find the source code and extensive documentation about this plugin at:

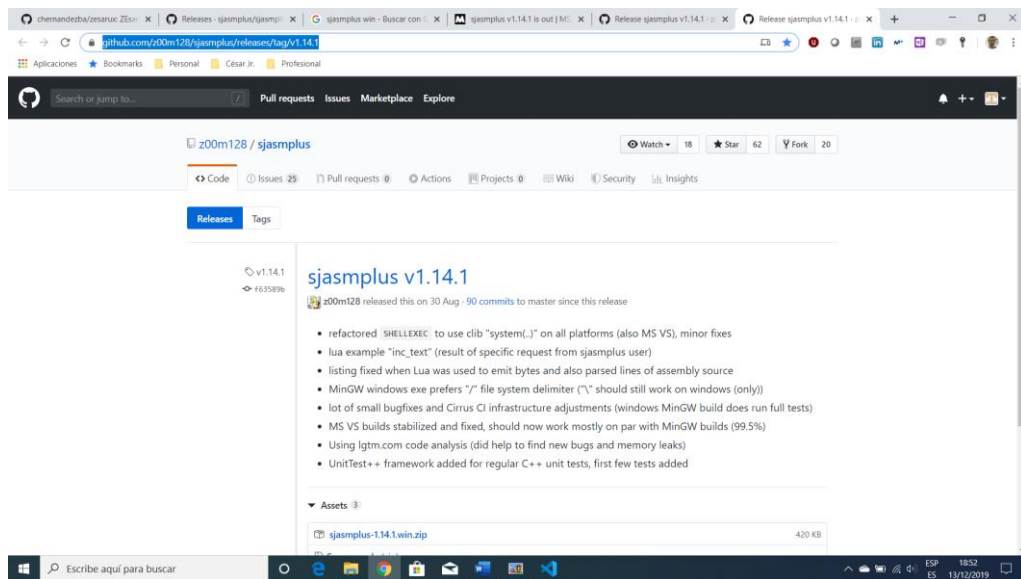
<https://github.com/maziac/z80-debug>

Although you can install the plugin directly in a simple way from VS Code, Thomas sent me a couple of versions to install manually from VSCode that correct things that do not work in the "published" version. Until Thomas does not release official version 0.9.3, you have to install the .VSIX file with the version 0.9.3-2 has prepared me specifically to correct a bug that prevented properly inspect the value of records or view the contents of variables and memory locations aimed at runtime.

- **Assembler sjasmpplus:** It will be the tool with which, from the IDE, will assemble our programs. Although there are more advanced betas, I'm working with version v1.14.1, which was released in August and can be downloaded from this address: <https://github.com/z00m128/sjasmpplus/releases/tag/v1.14.1>. You have complete documentation on it in these other directions:

<http://z00m128.github.io/sjasmpplus/documentation.html> or

<https://github.com/sjasmpplus/sjasmpplus/wiki>



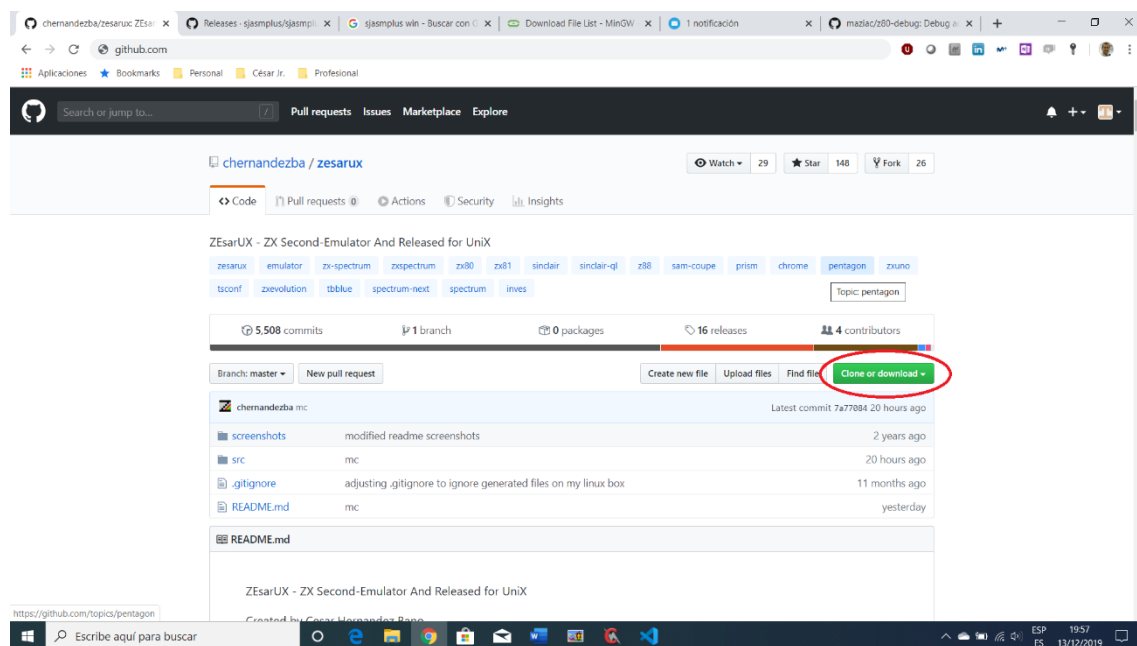
With all these elements downloaded and ready to be installed, we can get down to work!

### 3. Installing the emulator ZEsarUX

We begin with the "Heart of the Beast", the emulator ZEsarUX.

#### Step 1 - Download source code ZEsarUX

From the right page, click on the button **Clone or download** and download the file in the directory `zesarux-master.zip` where we go to stop the emulator (in my case, `D: \ Spectrum \ ZEsarUX`)



After downloading the file, unzip it on the same directory and move on to the next point, which is to install the MinGW to compile the source code ZEsarUX and get the .exe file of the emulator.

#### Step 2 - Installation and configuration of the MinGW

Once downloaded file mingw-get-setup.exe on the address, execute it and installed in the directory **C: \ MinGW** with all the default options. At the installation is complete, we will open the **MinGW Installation Manager**

where we have to choose to install the following packages:

In **Basic Setup**:

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

In **All packages**, you will have to add the following packages:

- mingw-pthreads (all packages that you see with this name)
- msys-bash

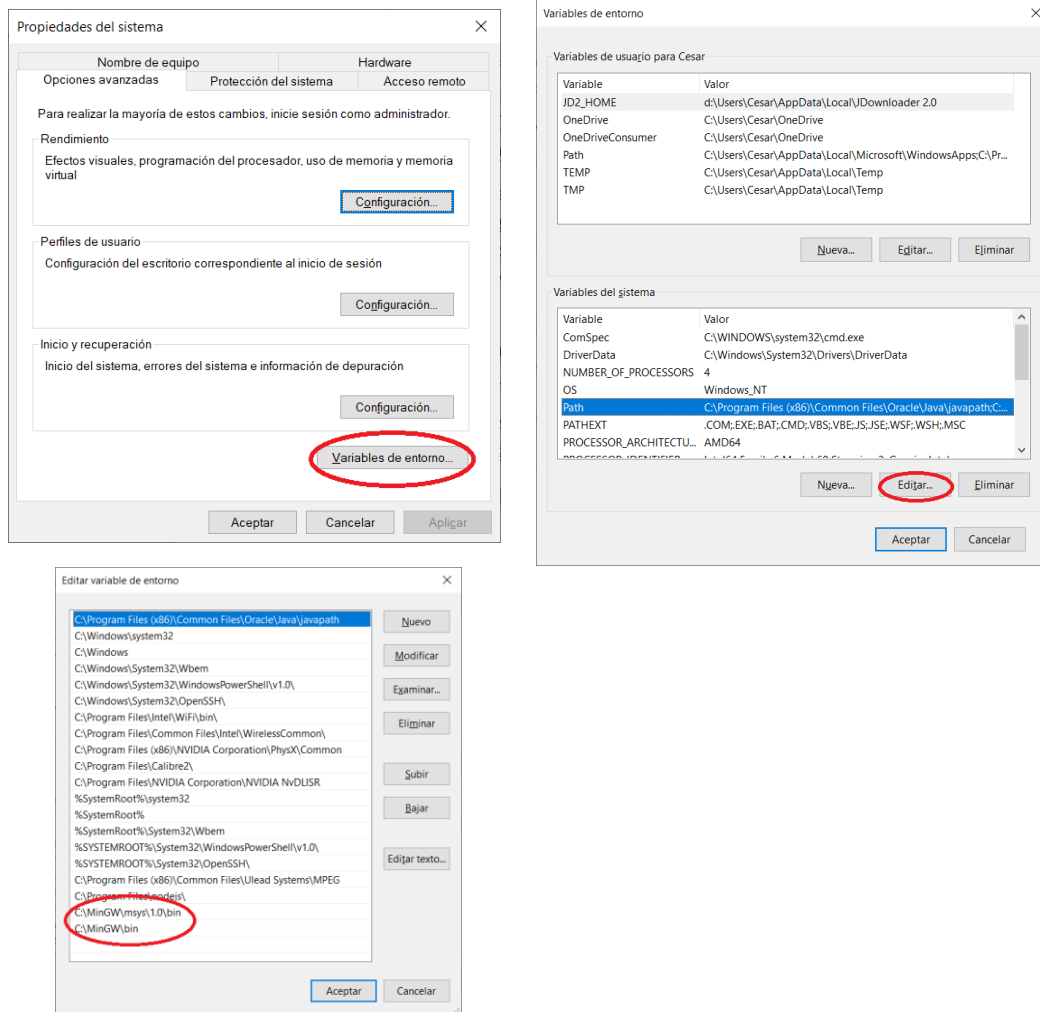
After selecting these packages, we apply the changes and we can close the **MinGW Installation Manager**. Now you have to go away to the browser, copy to **C: \ MinGW** the file downloaded earlier with SDL libraries ( **SDL-devel-1.2.15mingw32.tar.gz**) and uncompress. This will create a directory called us **SDL-1.2.15**

to rename simply as **sdl**.

If all went well, in **c: \ mingw \ sdl \ lib** We have several files **libSDL.dll \*** and in **c: \ mingw \ sdl \ include \** we will have a subfolder **SDL** within which there will be several files. **h**.

Our last step before compiling the .exe file will be added to the PATH system directories **c: \ mingw \ bin** and **c: \ mingw \ msys \ 1.0 \ bin**.

### Option a) from the control panel:



### Option a) from the command line:

Run cmd.exe and write **set PATH=% PATH%; c: \ mingw \ bin; c: \ mingw \ msys \ 1.0 \ bin**

### Step 3 - Compilation and configuration ZEsarUX

we are ready to compile our ZEsarUX. To do this, we will have to run cmd.exe, move to the directory where you have unzipped the source code ZEsarUX (in my case d: \ spectrum \ ZesarUX \ src) and run the command:

**bash**

we will display a prompt "bash.3.1 \$" from which to launch the following command

**./ 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
```

In the end, run **make clean** and finally **make** , So that the compilation will be launched. We left typing **bash exit** and if all went well, we in our directory the brand new file **src zesarux.exe** with the emulator executable.

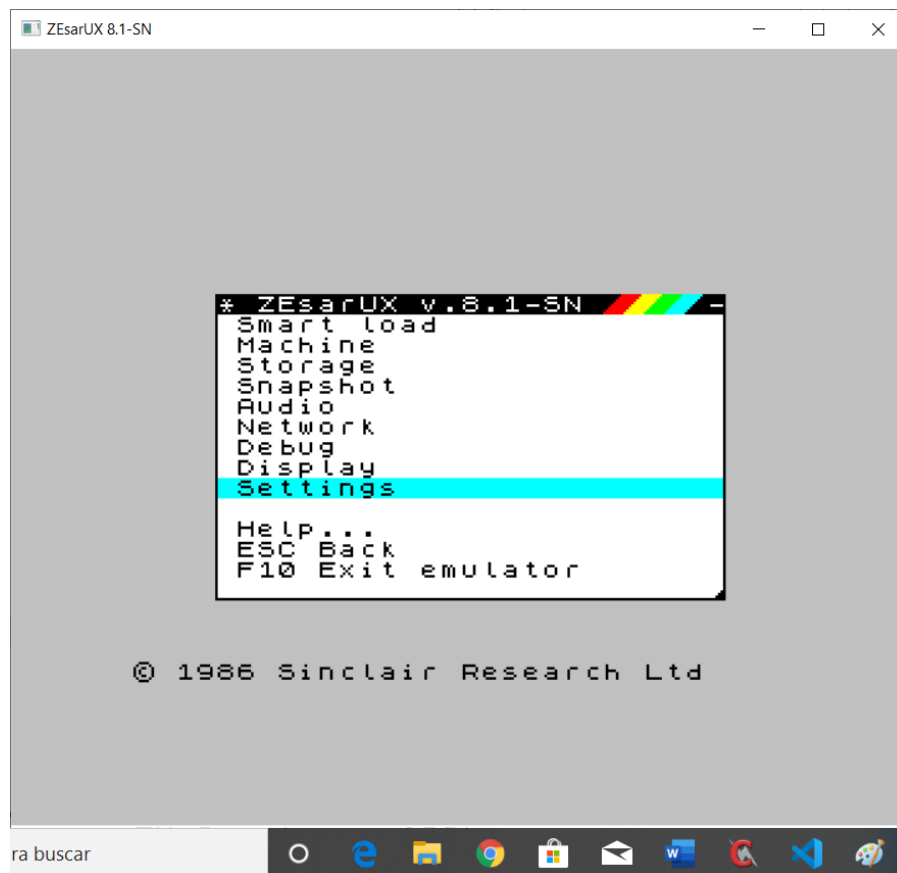
```
Símbolo del sistema
D:\Spectrum\Zesarux\src>make clean
rm -f *.o zesarux zesarux.exe smpatap sp_z80 tapabin bin_sprite_to_c leezx81 file_to_eprom bmp_to_prism_4_planar bmp_to_sprite spdetotxt install.sh
rm -fR bintargztemp/ sourcetargztemp/ ZEsarUX_win-8.1/
rm -fR macos/zesarux.app
rm -fR macos/zesarux.dmg
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 scrssimpletext.c
gcc -Wall -Wextra -fsigned-char -DMINGW -I/c/mingw/SDL/include -c scrssdl.c
scrssdl.c: In function 'realjoystick_sdl_init':
scrssdl.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 scrssdl.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);
                  ^~~~~~
```

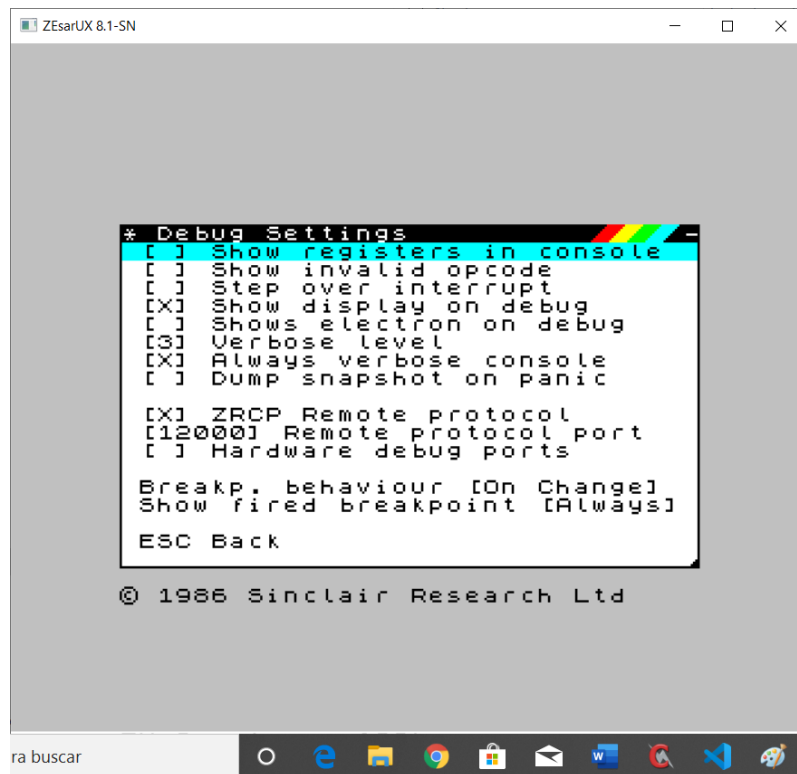
Our last step is to copy the dll **SDL.dll** from MinGW to the current directory with the command:

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

Done this ... " Now we can run our brand new set ZesarUX.exe and finish !!!! Nothing more open, click on the mouse, see the following menu:



here we will select **settings**, in the following menu **debug** and this will mark the option **ZRCP Remote Protocol**, setting the ZesarUX port that will communicate with the plugin (by default, 10000, although in my case gave me any problems and I have set for 12000)

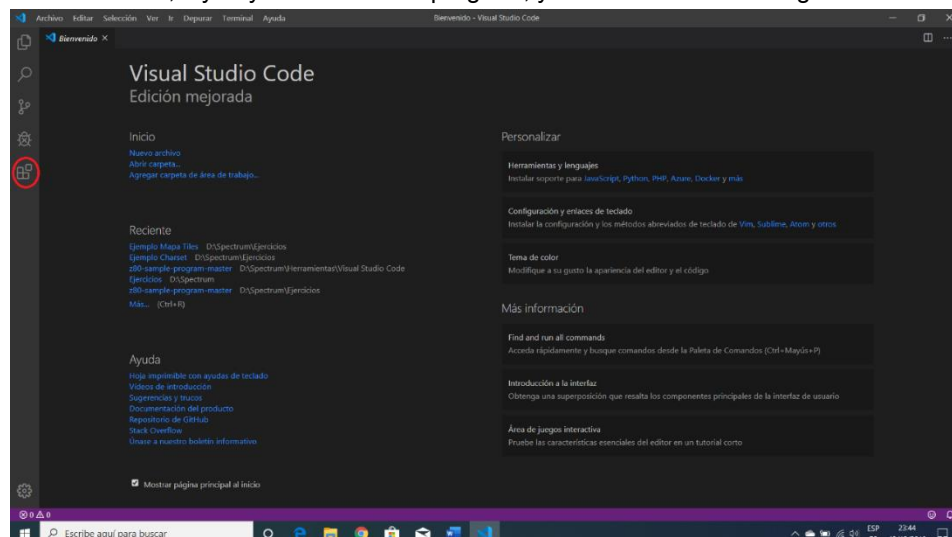


ZEsarUX has hundreds of interesting and configurable aspects that you see sailing from their menus. In order to debug, we do not need to configure anything else, and although not the subject of this tutorial, here I encourage you to "bichear" with options so you can see the power of this great emulator. Sure my namesake will be glad that you trasteéis and bring forth all possible options to match your emulator.

#### 4. Installing and configuring the VS Code

After downloading the executable file installer the Code VS from the address in paragraph 2 of the document, simply run it and follow the instructions, accepting the default options when installing.

Once installed, if you you execute the program, you will find the following window:

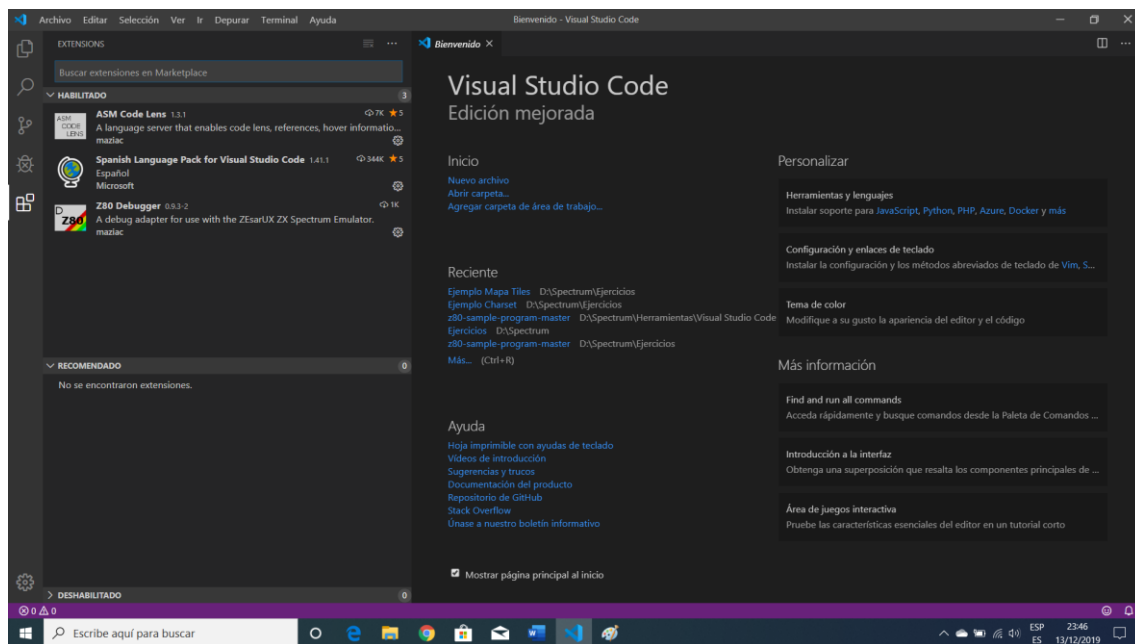


Yours will not be exactly the same because the system will be in English, but now we go with it.

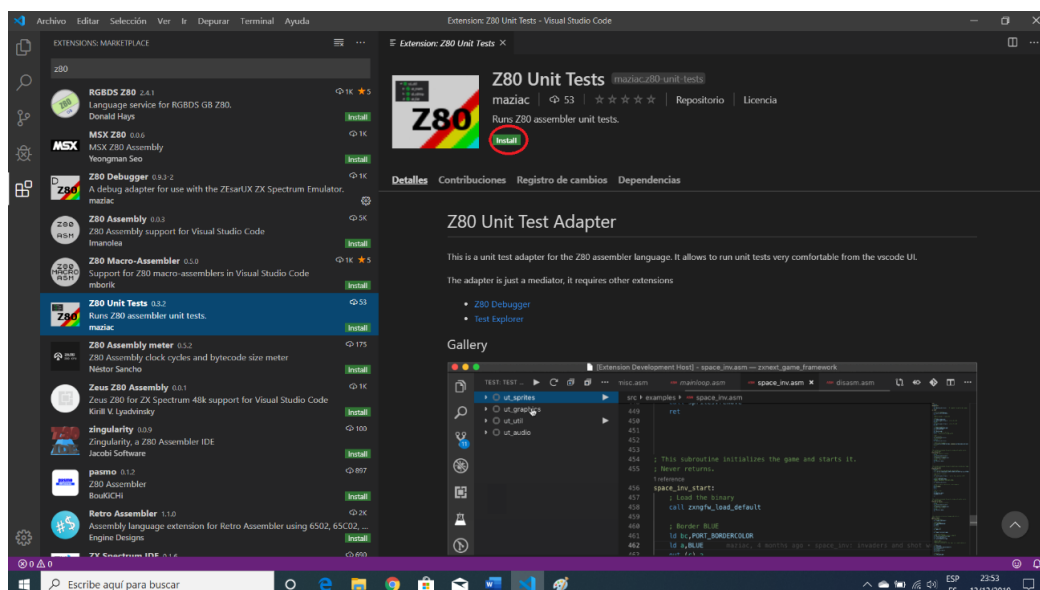
One more detail: I do not remember if I did the installer automatically, I advise checking that has been added to the system path of the subdirectory **bin** the folder where you installed the VS Code ye (in my case d: \ Microsoft VS Code \ bin)

## Step 1 - Installing basic plugins

The first thing we do is install the basic additional plugins. To do this, pincharemos on the icon that I have marked in a red circle, which will take us to the setup screen extensions:



To install a plugin simply look, click on it and in the right window, press the "Install" button



Now we will install the following additional plugins to Z80 Debug:

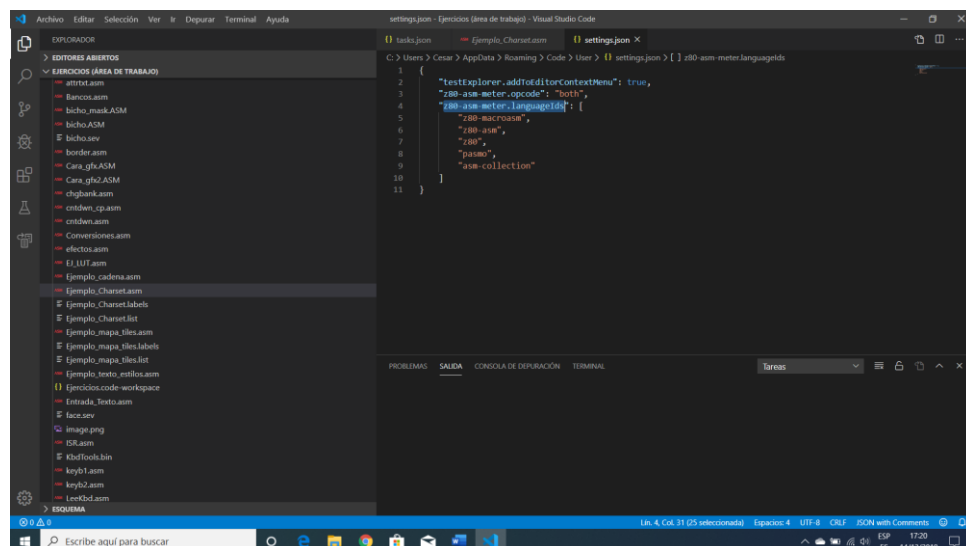
- **Spanish Language Pack for Visual Studio Code.** Will allow you to put your VS Code in Spanish, if you feel more comfortable working in our mother tongue.
- **ASM Code Lens**, that's like the main plugin Z80 Debug is developed by Thomas Busse (aka Maziac). This plugin will give us access to a number of very interesting tools when debugging (locating references, hovering variables and registers, see the number of references to a particular symbol, Assembler syntax highlighting ...).
- **Z80 Unit Tests.** Maziac also our friend. This plugin allows us to include in our ASM sources certain unit tests for testing in debugging time, stopping the execution if the test is not met. I have pending have a leisurely eye and see what you can do with it, but it looked great. If I see it's worth, I promise to make an extension to this tutorial use (if someone does not do it before me;).
- **Z80 Assembly Meter:** Grand plug Nestor Sancho, which determines the number of clock cycles that instructions consume we select from the publisher, and the size thereof bytecodes. In order for this plugin to work, you will have to make a small adjustment:

or We open the command console with **Ctrl + Shift + P**

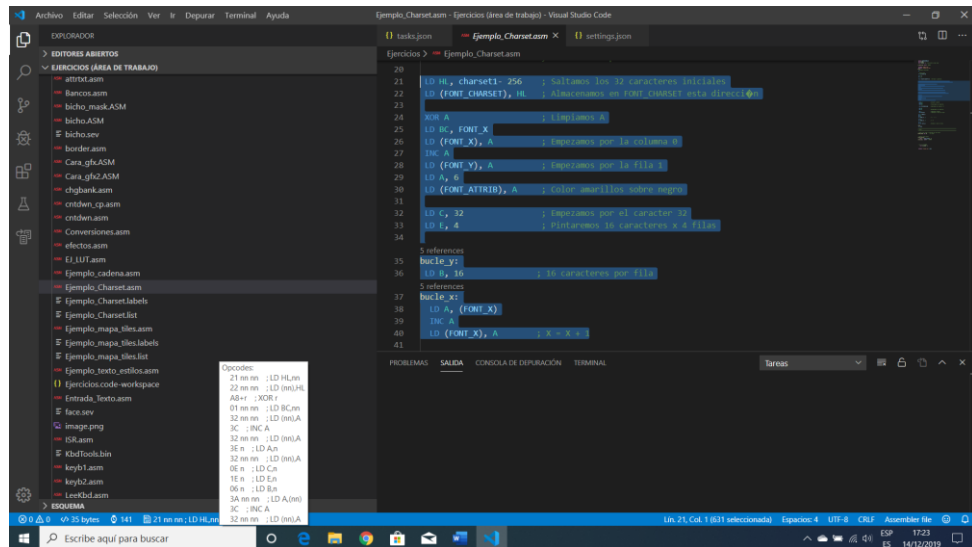
or select **Open Configuration Preferences (JSON)** You preferences

Open Settings JSON).

or At the entrance **z80-asm\_meter-languageIds** add, **"Asm-collection"**



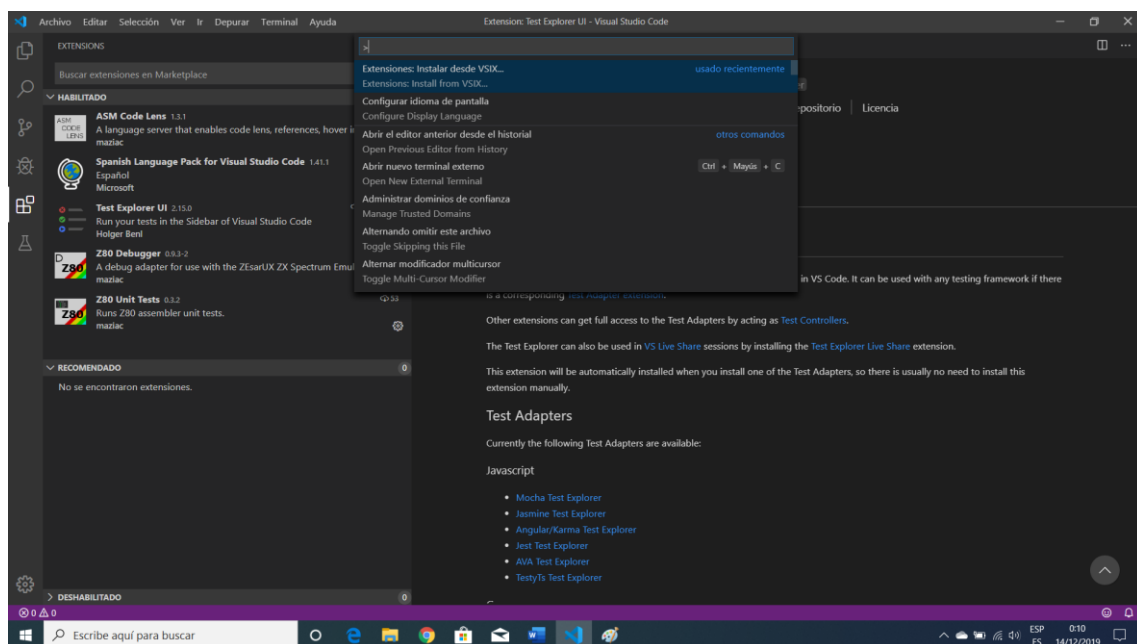
Now, whenever we select one or more lines of code, the status bar will show the size in bytes, the number of clock cycles that consume and (or) opcodes of the selected operations.



## Step 2 - Installing the plugin Z80 Debugger

Although this plugin can also be installed from the installation screen extensions, if you seek there you will find the v.0.9.2 version, and unfortunately, this version contains a bug that causes many problems during debugging, so you'll have to install the v.0.9.3-2 beta facilitated by my own Thomas Busse (and which you will find in the same zip where is this document) having a slightly different installation process.

The first step is to decompress the file **z80-debug-0.9.3-2.vsix** on a working folder. Now from VS Code, we will go to the "View" menu and then select the option **Command Palette**.



In doing so, a window will open where we find " **Extensions: Install from file VSIX** ". Now select the file **z80-debug-0.9.3-2.vsix**,

We accept and see how our plugin is installed perfectly. We are already ending.

### Step 3 - Install node.js and assembler SJASMPLUS

The last two tools you have to install are the node.js and SJASMPLUS. For the first, simply run the file **node-v12.13.1-x64.msi** we will have downloaded, run it and install with default options.

SJASMPLUS installation is even simpler. After downloading the file sjasmpus-1.14.3.win.zip of the address, just have to unpack to the directory ye (in my case d: \ Spectrum \ tools \ sjasmpus-1.14.3.win) and add to the path of the directory system as explained during the passage of the MinGW installation.

Made this last step, we can open our VS Code and prepare to start debugging. 'Let's mess !!

## 6. Creating our integrated IDE to debug

### 6.1 - Open the folder with our sample program

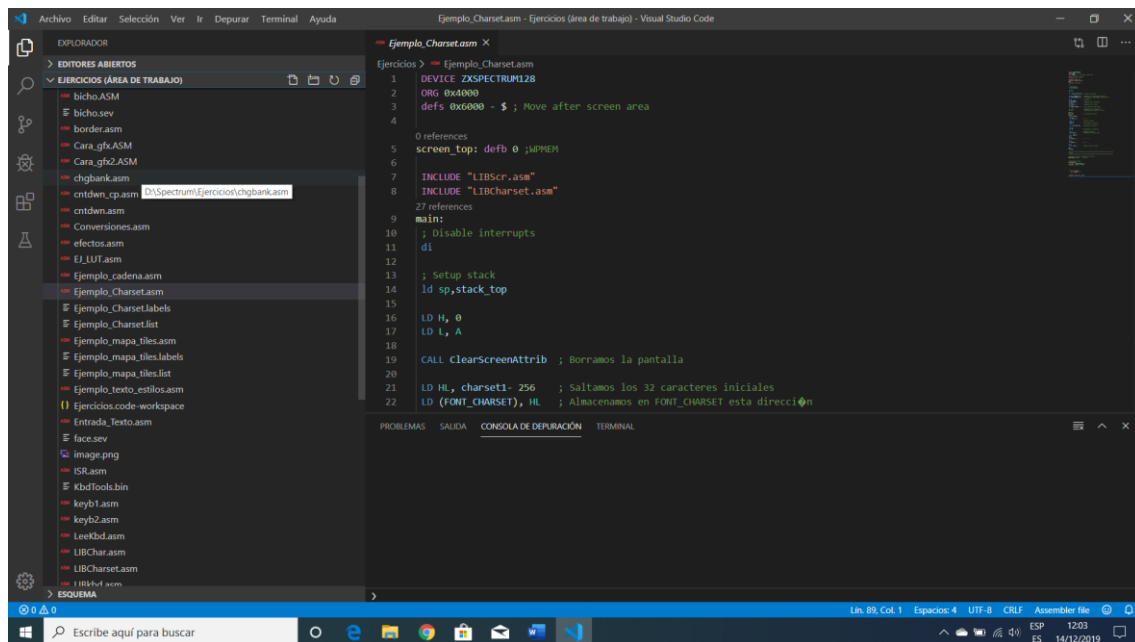
To illustrate debugging with our new IDE, I'll use a couple of sample programs included in the course **Compiler soft**: One shows a complete charset and another painting a map of a level of sokoban program. You can find the source code (and files **tasks.json** and **launch.json**) in the file **Ejemplos.rar** that Annex to this tutorial.

A decompressing the file, find the following content:

- **LIBSprites.asm**: Library routines for handling sprites.
- **LIBScr.asm**: Library routines for handling screen.
- **LIBCharset.asm**: Library routines for handling charsets.
- **Ejemplo\_Charset.asm**: File with the main routine of the first example
- **Ejemplo\_charset.list and .labels**: generated during assembly and necessary for debugging
- **Ejemplo\_mapa\_tiles.asm**: File with the main routine of the second example
- **Ejemplo\_mapa\_tiles.list and .labels**: generated during assembly and necessary for debugging
- **folder .vscode** It is containing files **tasks.json** and **launch.json**, necessary to launch the assembly from VS and begin debugging Code respectively.

Unzip the zip file in a working folder and start the VS Code. Ideally, create a new workspace (option **File are listed \ Save Workspace as ...** ) and include in the same folder where you have downloaded the code from the option

\ File add folder to the workspace. If we have done well, we will see a screen similar to this:



In the bar we have left, mainly we use two options:

- Explorer (first icon from the top) to browse files.
- Debug and run (icon of the "bug") to launch debugging

## 6.2 - Files tasks.json and launch.json

In all our projects, we have a directory called **vscode** hanging from the root directory where we have two very important files: the **tasks.json** and the **launch.json**. While you can create by other means, my advice is that you have a master copy of this directory and for each project, the modifiqueis files and copy them as appropriate, since it is the most comfortable option.

Let's talk now about these files, what they are and how they are used.

### FILE TASKS.JSON

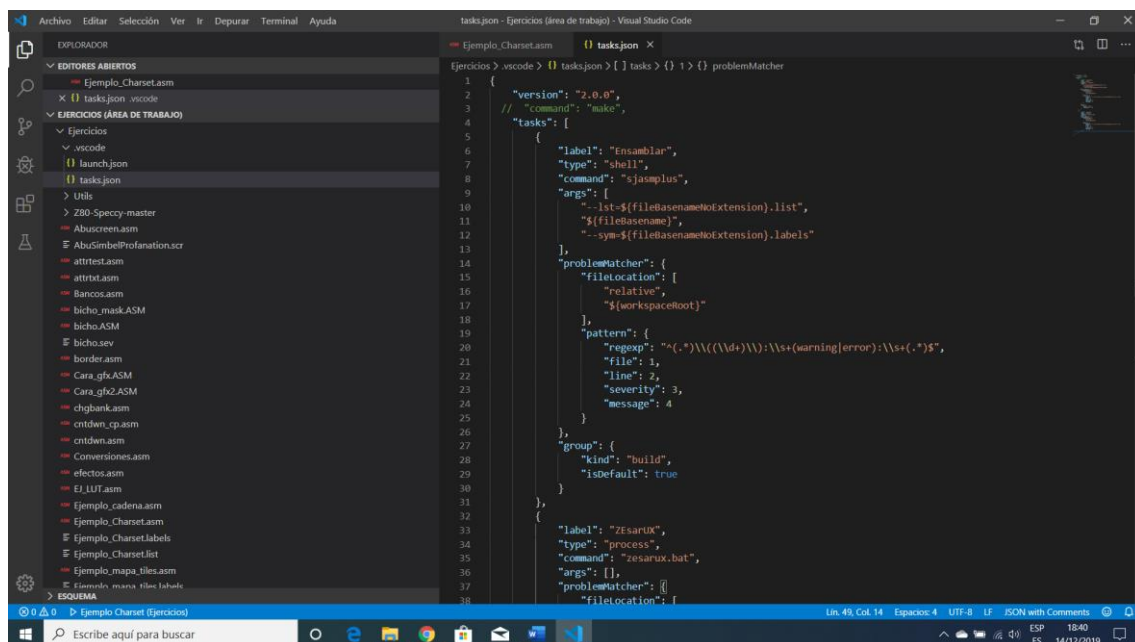
Through this file, we can define commands or tasks that can be launched from VS Code. In our case, we will create two tasks:

- One that will invoke the **sjasmplu**s to join our code and generate files. **lst** and **. labels** you need the debugger for debugging step by step.
- Another ZEsarUX to call before launching debugging. For this second case, we will create a file called zesarux.bat to be positioned in the directory where is ZEsarUX and run. In my case, this file contains the following commands:

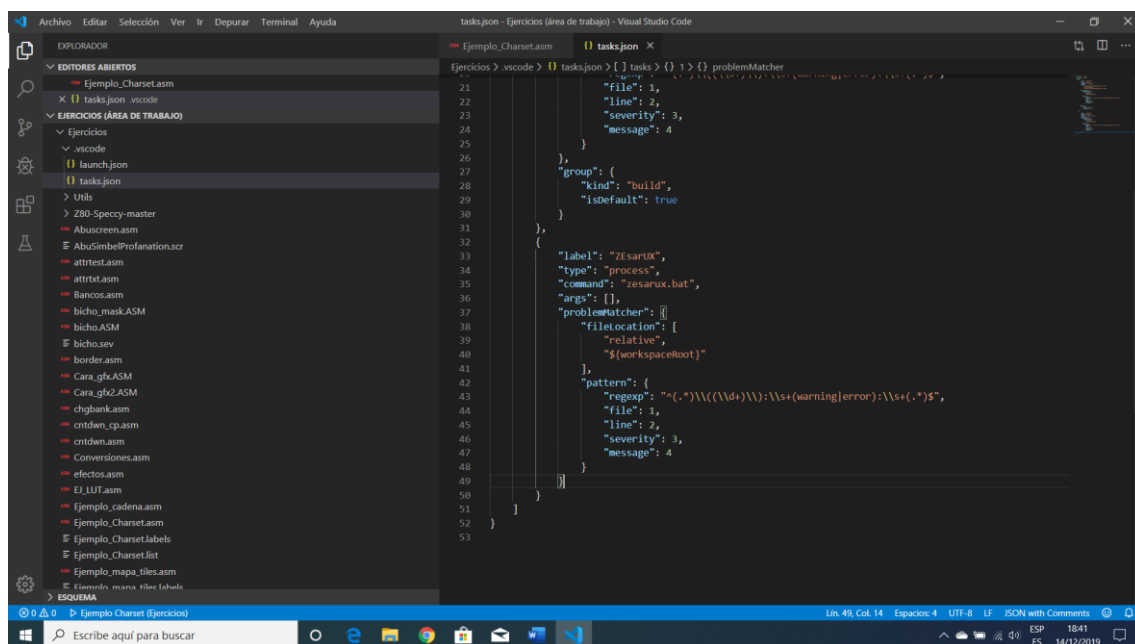


```
d: cd
\
zesarux cd cd cd
src spectrum
exit zesarux
```

Although we can define more tasks, our file input tasks.json have the following structure:



```
{
  "version": "2.0.0",
  "command": "make",
  "tasks": [
    {
      "label": "Ensamblar",
      "type": "shell",
      "command": "sjasplus",
      "args": [
        "-l=${fileBasenameNoExtension}.list",
        "${fileBasename}",
        "-s=${fileBasenameNoExtension}.labels"
      ],
      "problemMatcher": {
        "fileLocation": [
          "relative",
          "${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": [
        "-l=${fileBasenameNoExtension}.list",
        "${fileBasename}",
        "-s=${fileBasenameNoExtension}.labels"
      ],
      "problemMatcher": {
        "fileLocation": [
          "relative",
          "${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
          }
        }
      ]
    }
  ]
}
```

Let's see the main tags this file:

- **Label:** Contain the name you will see the task if we try to run it from the option **terminal \ perform task ...**
- **type:** Here we indicate that the task will run on a Shell
- **Command:** Name of the command that will launch from Shell, in our case, **sjasmplus**.
- **args:** contain a list of different parameters to be passed to the command to be executed from the shell. In our case will be the name of the file. **list** generate (our main file without its extension, adding **.list**), assemble the file and the file name tags to generate (again, the name of our main file without adding extension

**.labels):**

```

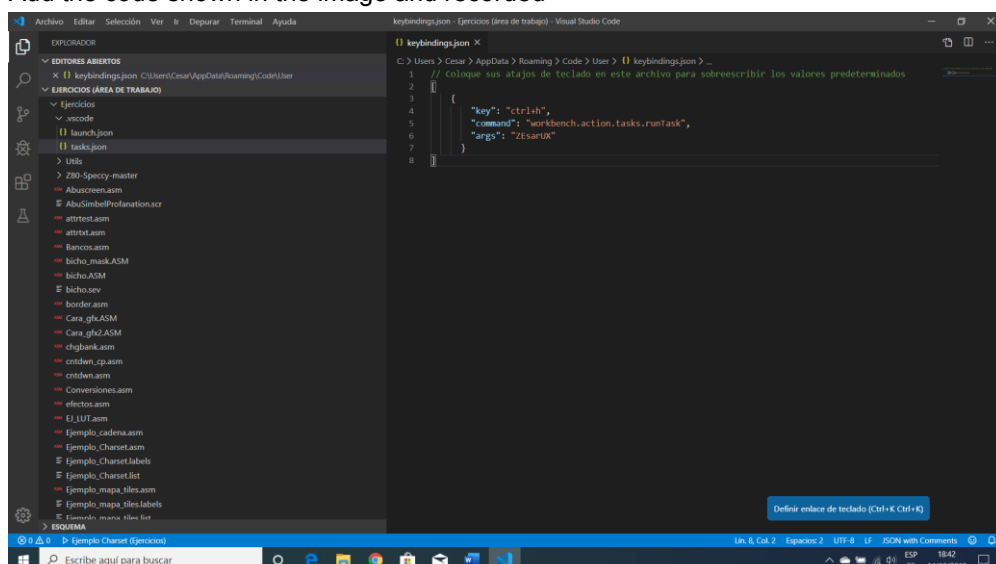
•      "--Lst = $ {fileBasenameNoExtension} .list" ,
•      "$ {FileBasename}" ,
•      "--Sym = $ {fileBasenameNoExtension} .labels"

```

- The remaining tags are standard. Including highlighting **group**, where we will establish our task is a task of compiling and also be the default task, to launch pressing **b Shift + Ctrl +**

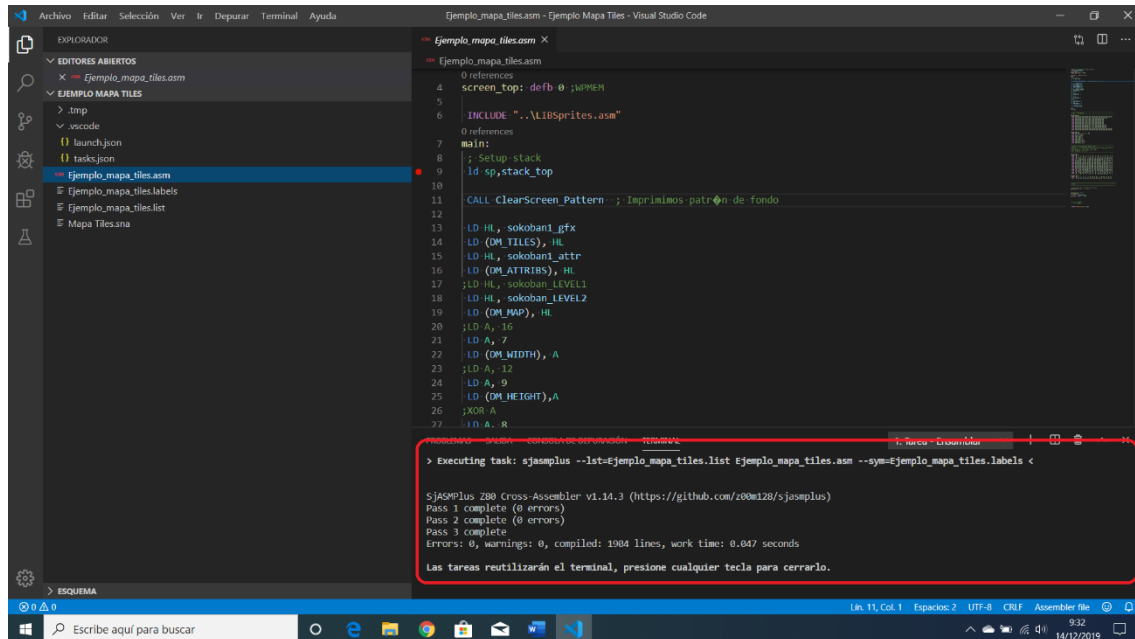
In the tasks file shown in the image, we create the two tasks: one to invoke the assembler will run **b Shift + Ctrl +** and another to call ZEsarUX. To launch it, we will assign a key combination: **ctrl + z**

- From the menu **View \ Command Palette ...** select **Open keyboard shortcuts (JSON)**
- Add the code shown in the image and recorded



## 6.2 - Assembly of the ASM code SJASMPPLUS

To see that all is well defined, since the browser will select the file ejemplo\_mapa\_tiles.asm and pressing the combination of keys indicated, launch assembly whose result will see in the debug console:



```
 Ejemplo_mapa_tiles.asm X
 Ejemplo_mapa_tiles.asm
0 references
4 screen_top: defb 0 ;WIPEN
5
6 INCLUDE "..\LIBSprites.asm"
7
8 references
7 main:
8 ; Setup stack
9 ld sp,stack_top
10
11 CALL ClearScreen_Pattern ; Imprimimos patrón de fondo
12
13 ld hl, sokoban1_gfx
14 ld (DM_TILES), hl
15 ld hl, sokoban1_attr
16 ld (DM_ATTRIBS), hl
17 ;LD HL, sokoban_LEVEL1
18 ld hl, sokoban_LEVEL2
19 ld (DM_MAP), hl
20 ;LD A, 16
21 ;LD A, 7
22 ;ld (DM_WIDTH), A
23 ;LD A, 12
24 ;LD A, 9
25 ;ld (DM_HEIGHT),A
26 ;XOR A
27 ;ld A, R

> Executing task: sjasmpplus --lst=ejemplo_mapa_tiles.list Ejemplo_mapa_tiles.asm --sym=ejemplo_mapa_tiles.labels <

SjASMPplus Z80 Cross-Assembler v1.14.3 (https://github.com/z00m128/sjasmpplus)
Pass 1 complete (0 errors)
Pass 2 complete (0 errors)
Pass 3 complete
Errors: 0, warnings: 0, compiled: 1904 lines, work time: 0.047 seconds
Las tareas reutilizarán el terminal, presione cualquier tecla para cerrarlo.
```

For you to come to the world PASMO, you step needed some syntax changes between PASMO and sjasmpplus you have to do in your program if you will not have a bunch of assembly errors:

- All tags should begin in the first column, leaving no space because otherwise be considered directives or instructions, resulting in error.
- All instructions or directives must leave the least space not to be considered labels.
- Our program will start with the directive DEVICE indicating the type of machine that will generate the SNA file (ZXSPECTRUM48, ZXSPECTRUM128, etc.)
- Will have to define a label for the input routine, which then pass to the SNA it generates directive file use to debug.
- Definition battery life: although I have not deep enough, seems to be defined zones start and end of the stack. For this you need to add the following code:

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

```

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

; Book stack space
stack_bottom :
    defs          STACK_SIZE * two , 0
stack_top : DEFB 0 ; WPMEM

```

In addition to making the next load in SP to the start of the main routine:

```

; Setup stack
ld sp , stack_top

```

### 6.3 - Launching a debugging session

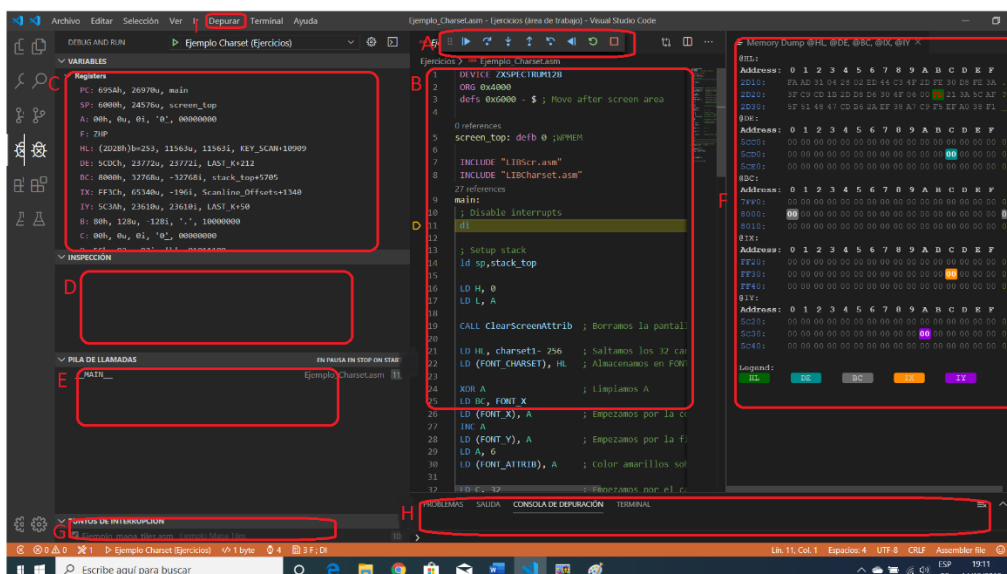
Well, the great moment has come: let's debug the first example. For it:

- we will open Ejemplo\_charset.asm
- Assemble it by pressing **CTRL + SHIFT + b**
- we started with ZEsarUX **ctrl + z**
- Press F5 to launch debugging

## 6. Introduction to debug

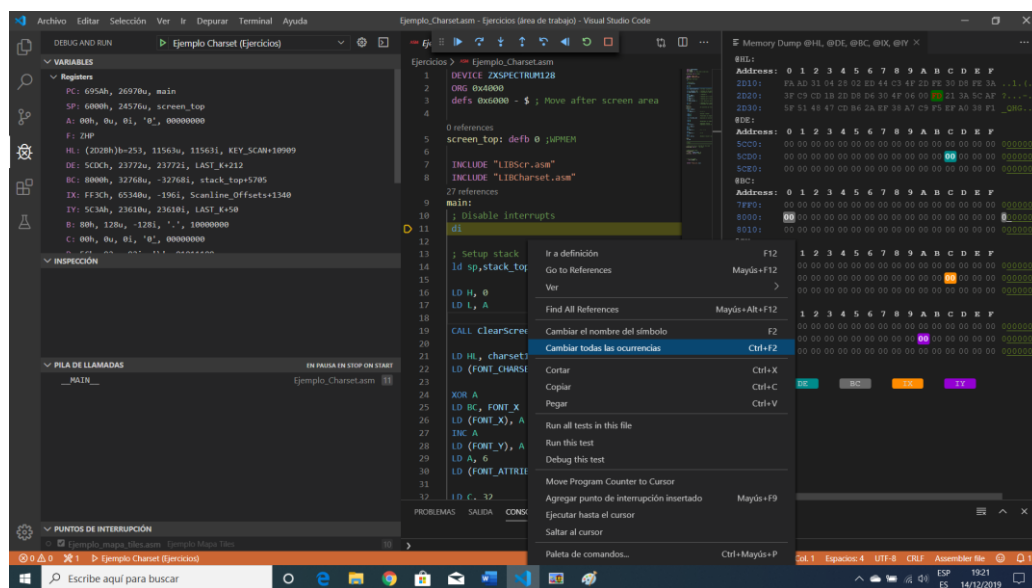
Although the tool allows Advanced Options debugging (conditional breakpoints, inspecting areas of specific memory, etc.), it has not yet had time to delve into them and little I can tell you in this regard, so you'll have to research a little documentation mentioned at the beginning of the document and try, try and try.

In any case, if I can give a small outline of what basic tools we offer the IDE to debug:

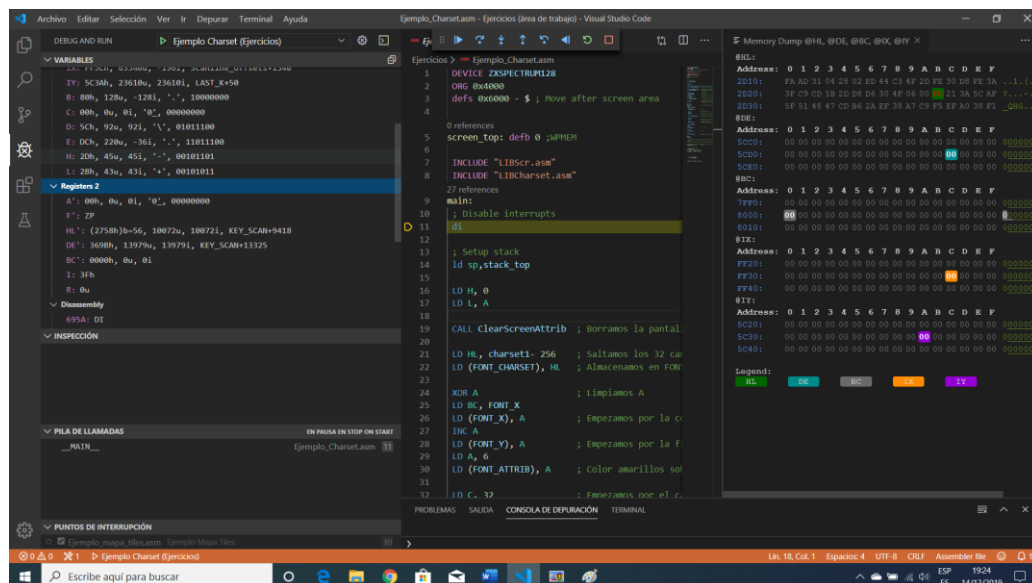


Areas I checked are:

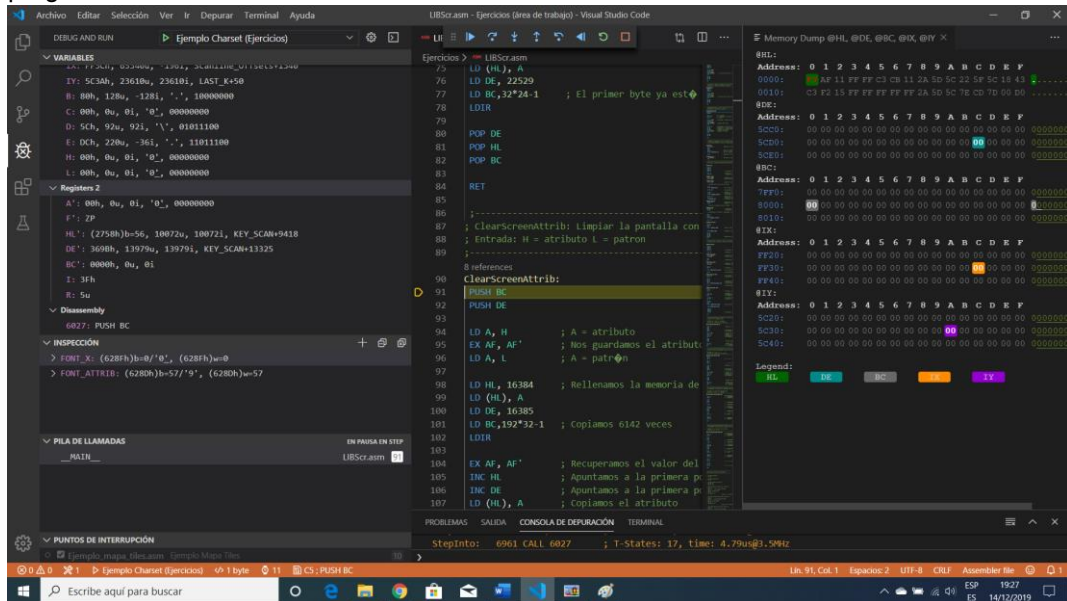
- **A - Command Bar:** It offers the usual options for debugging, as they are to continue, step by step procedures, step by step instructions, out of the routine, back in the running, reverse, restart and finish debugging. In the code window (labeled B) are marked in green executed instructions.
- **B - Area code:** Shows the code that we are running. Putting the cursor on a variable or label, we can see its value and content of the memory area to which it points. By pressing the right button, we can see and browse the definition of a routine or variable references in the code has run to that position, etc.



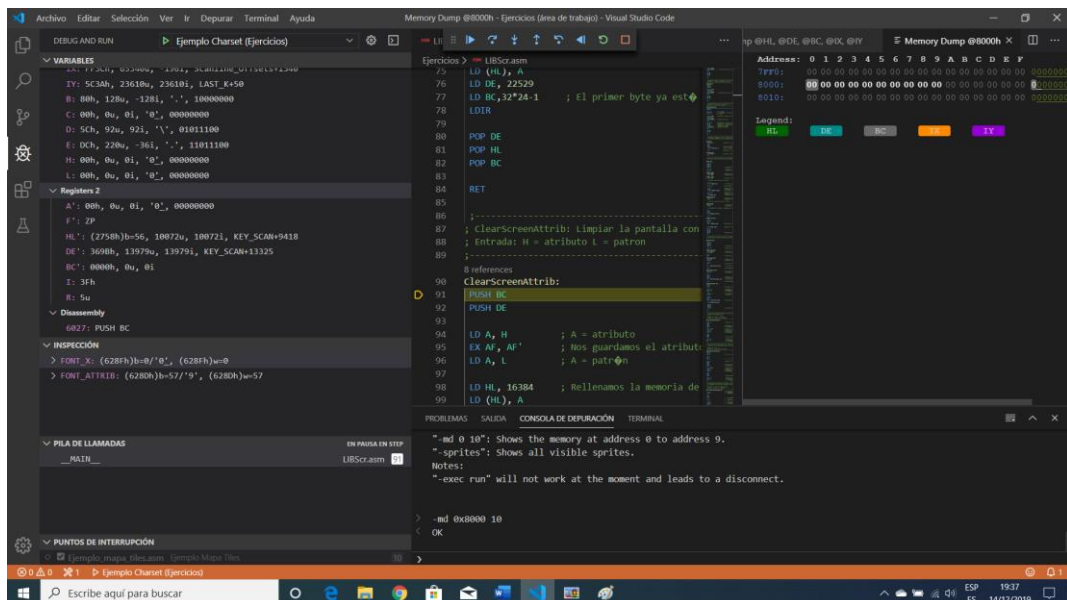
- **C - Zone variables:** View and modify the values of the records, shadow, see the disassembled code, memory areas or content of the stack.



- **D - Inspection Zone:** Lets you add variables to be inspected during execution of the program code



- **E - Call Stack:** It indicates that point we find exactly the execution.
- **F - Inspection Zone memory:** We can view (and change) the values contained in specific memory areas. By default displays the areas targeted by HL, DE, BC, IX and IY registers. However, if from the Debug Console (zone H) write `-MD address bytes` (-MD eg 0x8000 10), it will show a new window with that area of memory and the number of bytes marked.



- **G - Breakpoints:** It allows you to add breakpoints. We can also do this by double clicking on the left of any line of code.

- **H - Debug Console:** From here we can launch specific commands during debugging. I recommend launching a - **help** to see a list of available commands.
- **I - Debug Menu:** It lets you access the main debugging options.

## 7. Now what?

Well, at this point and will only put you to program is like madmen in ASM and to discover all the possibilities offered by the combination of these tools.

For my part, I will continue researching / studying the following aspects:

- Syntax and possibilities **sjasmplus**
- And additional configuration possibilities **ZEsaUX**
- Debugging capabilities offered by the plugin **Z80 Debug**
- Using Test Unit Z80 in my programs

As you progress along those lines, I will publish updates to the tutorial with everything that you discover.

## 7. Acknowledgments

Finally, I have to finish with the public thanks to several people who have been helping me in every problem I have encountered along the way and who have put their bit to have completed this tutorial.

Specifically, I want to mention especially:

- **César Hernández**, for making an emulator like ZEsaUX, which offers a world of possibilities, but mostly because of its proximity, cordiality and interest in helping me at every step. 'THANK YOU !!! Namesake
- **Thomas Busse**, for giving us such a brutal plugin that squeezes all the possibilities ZEsaUX and your kindness when answering my questions, issues, problems and, above all, for giving a specific version for Windows that corrected all the bugs I did not allow debug as I was looking for.
- **Nestor Sancho**, for their sympathy and quick help to run your plugin without colliding with other plugins I've had to install.
- **My fellow groups "Assembler ZX Spectrum" and "Retrodevs"**.  
Your closeness, help, comments and others are key to progress in my dream "tame the beast" ....

And I will end with this tutorial serve for a single person to use the development environment and make the most, the effort has cost me

learn (and later write) everything you need to mount it will have been worthwhile.

I would like, if you use this tutorial, I contactéis me by mail or telegram (my Nick is @Metaprime) to share questions, problems, progress and others.

See you on the forums!

César Wagener Moriana in Seville December 14, 2019.

**Randomize usr 0**