Inhalt

1.- Prolog

Ich werde alt.

In ein paar Tagen bekomme ich 48 Kastanien und, wie es im Alter der Fall ist, gab mir dieser Sommer in den Ferien die Möglichkeit, zurückzublicken und die Dinge zu überprüfen, die mir während meines ganzen Lebens passiert sind, und wie sie mich beeinflusst haben, zu werden, wer ich bin.

Und ich erkannte, dass ich eine historische Schuld zu begleichen hatte.

Meine Leidenschaft für das Computerspiel begann bereits in den 80er Jahren, als mein Vater ein brandneues Spectrum 16K brachte, das ihm in der Bank geschenkt wurde, mit der ich die aufregende Welt der Videospiele entdeckte. Von dort aus absolvierte ich ein Programm in Basic und schon in der High School kam ich mit dem C und UNIX voran. Als ich die COU beendete, hatte ich nur eine Sache im Kopf: das Studium der Informatik.

Bereits an der Universität habe ich nach 6 Jahren harter Arbeit meinen neuen Abschluss in Computertechnik gemacht, dank dem ich heute meinen Lebensunterhalt verdiene, indem ich Informationssysteme für Krankenhäuser herstelle.

Aber es gibt etwas, das ich nie getan habe, und es war ein Dorn in meiner Seite. Ich habe ASM nie von der Z80 gelernt (oder Maschinencode, wie er von der Microhobby genannt wurde), noch habe ich den vollen Nutzen aus dieser wunderbaren Maschine gezogen, ohne deren Eintritt in mein Leben ich vielleicht einen Abschluss in Arbeitslosengeschichte (meine andere große Leidenschaft) machen würde.

Nachdem ich einige Tage darüber nachgedacht hatte, entschied ich mich, meine alten Schulden zu begleichen, indem ich in eine mir unbekannte Welt eintrat, und ich machte mich daran, ein Spiel für Spectrum in ASM rein und hart zu machen, denn die Herausforderung bestand nicht so sehr darin, ein Spiel zu beenden und zu veröffentlichen, sondern "das Tier zu dominieren" und zu lernen, das zu tun, was ich als Kind nicht erreichen konnte.

Mit viel Mut begann ich meine Reise, um ASM vom z80 zu lernen und las mir den beeindruckenden und unverzichtbaren Kurs von **Compiler Soft**, den Sie unter https://wiki.speccy.org/cursos/ensamblador/indice finden können, dessen Beispiele ich im **Kontext** eingegeben und in **Spectaculator 8.0** getestet habe, aber es war sehr schwierig für mich, denn mit dieser "IDE" sind die Möglichkeiten des Debuggens nicht sehr vollständig.

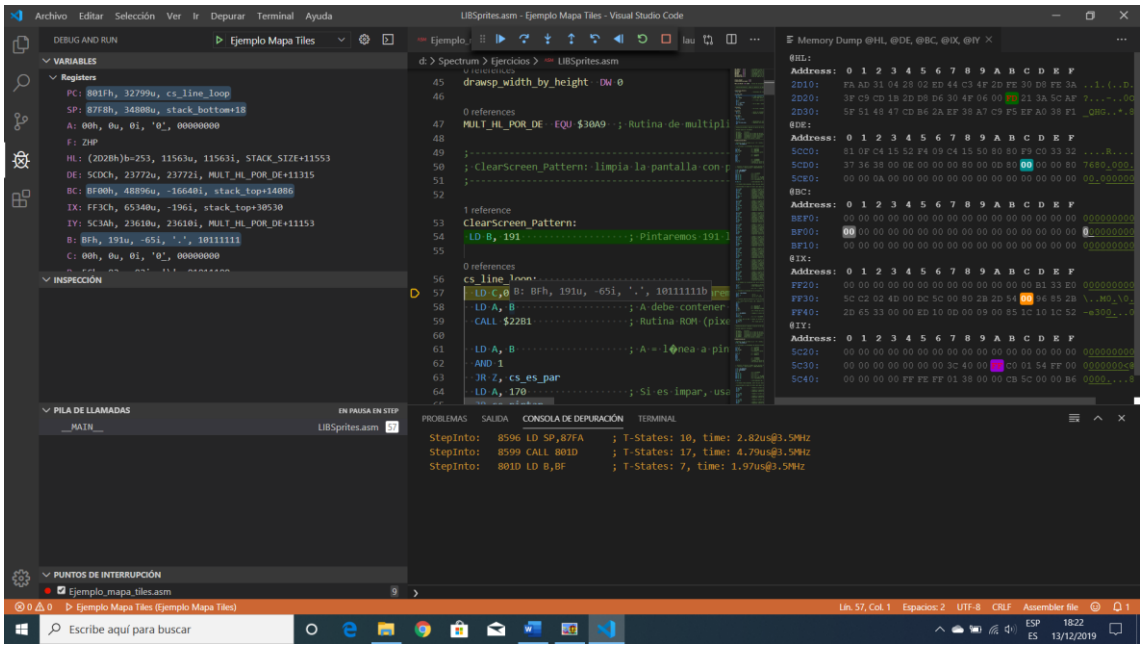
Als der Sommer wieder kam, trat ich in mehrere Telegrammgruppen ein, in denen ich eine Gruppe von Menschen mit viel Erfahrung traf, an denen ich Zweifel hegte und die ihr Sandkorn beisteuerten, damit ich auf meinem Weg weitermachen konnte. In einer dieser Gruppen traf ich Cesar Hernandez, Entwickler des wunderbaren **ZEsarUX**, einem spektakulären und wachsenden Spectrum-Emulator, der viele der Tools bietet, die ich

vermisst habe, aber immer noch ohne eine IDE, die so integriert ist, wie ich sie gesucht habe.

Eines Tages sprach Caesar in einer der Gruppen eines Plugins für VS-Code, Thomas Busses **Z80-Debugger**, der es ermöglichte, einen Assembler auf sehr einfache Weise aus dem Code heraus zu debuggen, der auf **ZEsarUX** lief und die Werkzeuge bereitstellte, die er bis dahin nicht gefunden hatte. Das ist es, wonach ich gesucht habe.

Cesar erzählte mir, dass er unter UNIX arbeitete und dass er nicht wirklich einen Beta-Tester der IDE unter Windows hatte. Als ich Thomas kontaktierte, war die Antwort die gleiche: Die Sache wurde interessant.....

Nach ein paar Wochen Tests, Wut, Kopfschmerzen und Hunderten von E-Mails und Nachrichten mit Cesar und Thomas, und einer anderen Version, die sie mir geschickt haben, um Fehler zu korrigieren, habe ich es in Gang gebracht und.... es ist spektakulär.



Meine Bemühungen wären nicht zu Ende ohne dieses Dokument, das, wie ich hoffe, anderen Entwicklern helfen wird, ohne dass sie alles erleiden müssen, was ich erlitten habe, die Arbeit von César und Thomas mit der Summe dieser mächtigen Werkzeuge zu nutzen, denen ihre Schöpfer so viel Zeit, Mühe und Illusion widmen.

Wenn jemand, während er dem Tutorial folgt, in einem Schritt stecken bleibt oder eine bestimmte Frage hat, kannst du mich kontaktieren, indem du mir eine E-Mail an cesar.wagener@gmail.com schickst, und wenn möglich, werde ich versuchen, dir zu helfen.

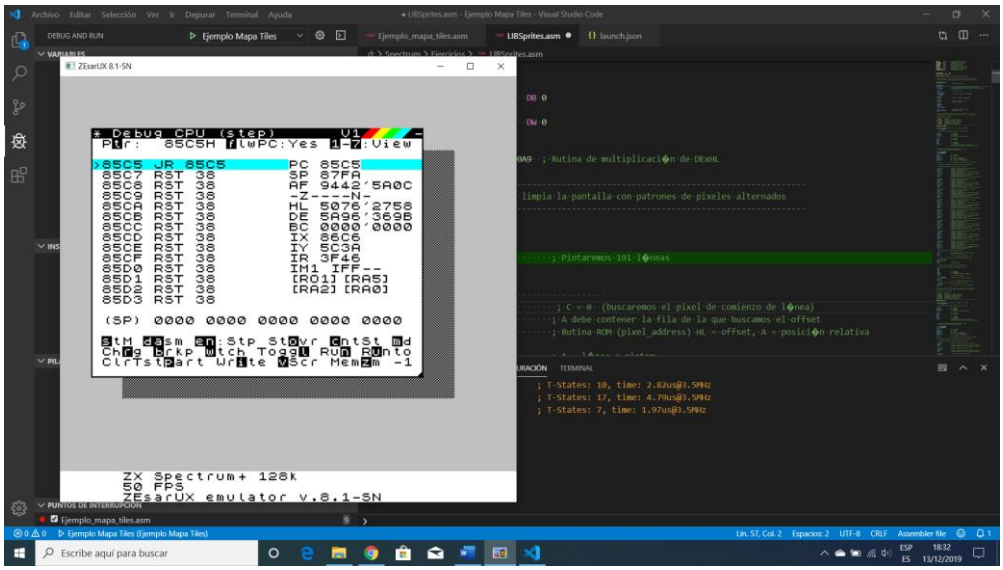
Und jetzt, kommen wir zur Sache!

2.- Elemente, die für die Montage der IDE benötigt werden

Nun, das erste, was ich Ihnen sagen muss, ist, dass alles, was ich Ihnen erklären werde, ist, dass ich es auf einer Professional W10 ausprobiert habe, aber ich stelle mir vor, dass es ohne größere Probleme auf anderen Windows-Systemen funktionieren sollte, und sogar ohne zu viele Probleme auf Linux oder Mac umsteigen sollte.

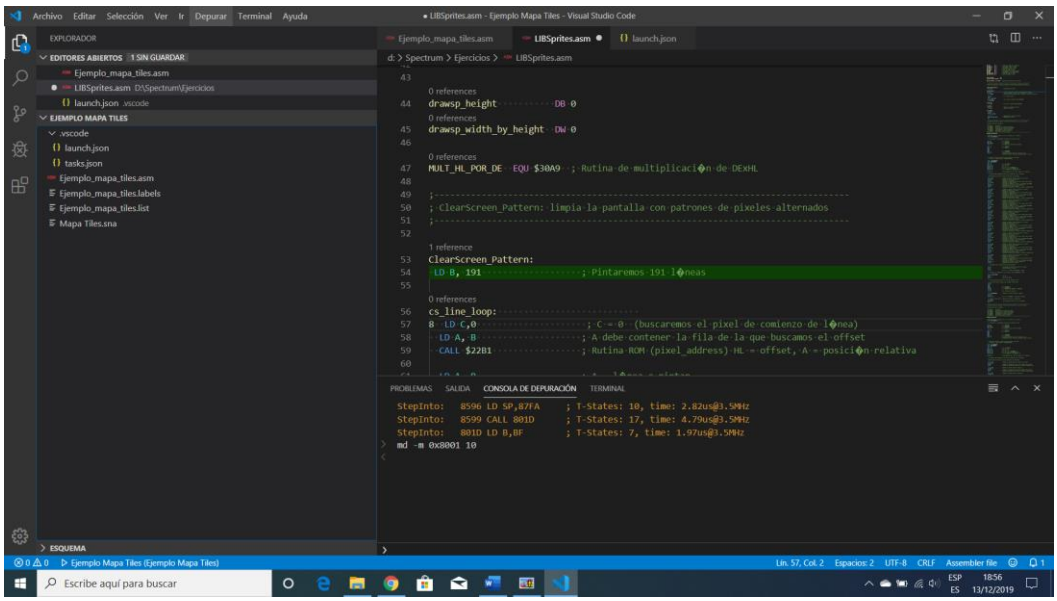
Um die Umgebung einzurichten, benötigen wir die folgenden Tools:

- **ZEsarUX (ZX Second Emulator And Released for Unix):** Erstes Kernstück. Es ist der Emulator, auf dem wir unseren Code ausführen werden. Es verfügt über viele Konfigurationsmöglichkeiten und eigene gute Debugging-Tools, die es externen Systemen über Sockets mit dem ZRCP-Protokoll anbieten kann. Den Quellcode, die ausführbaren Versionen und die Dokumentation dazu finden Sie unter https://github.com/chernandezba/zesarux



- **MinGW Compiler:** Obwohl ich nicht weiß, ob Cesar bereits eine neue offizielle Version hochgeladen hat, musste ich in meinem Fall den Quellcode von ZEsarUX herunterladen und kompilieren, um Zugang zu den neuesten Entwicklungen und Bugfixes von Cesar zu erhalten. Die aktuelle Version können Sie unter folgendem Link herunterladen: https://sourceforge.net/projects/mingw/. Zusätzlich müssen wir die Version 1.2.15 der SDL-Bibliothek (Datei **SDL-devel-1.2.15-mingw32.tar.gz**, die Sie von dieser Adresse herunterladen können: https://www.libsdl.org/download-1.2.php) herunterladen.

- **VS Code Editor:** Werkzeug, aus dem wir den Code schreiben, zusammensetzen und das Debugging starten. Es ist ein von Entwicklern weit verbreitetes Microsoft-Tool, obwohl ich weiß, dass ich es nicht kannte. Wir verwenden die Version 1.41, die Sie unter folgendem Link herunterladen können: https://code.visualstudio.com/download

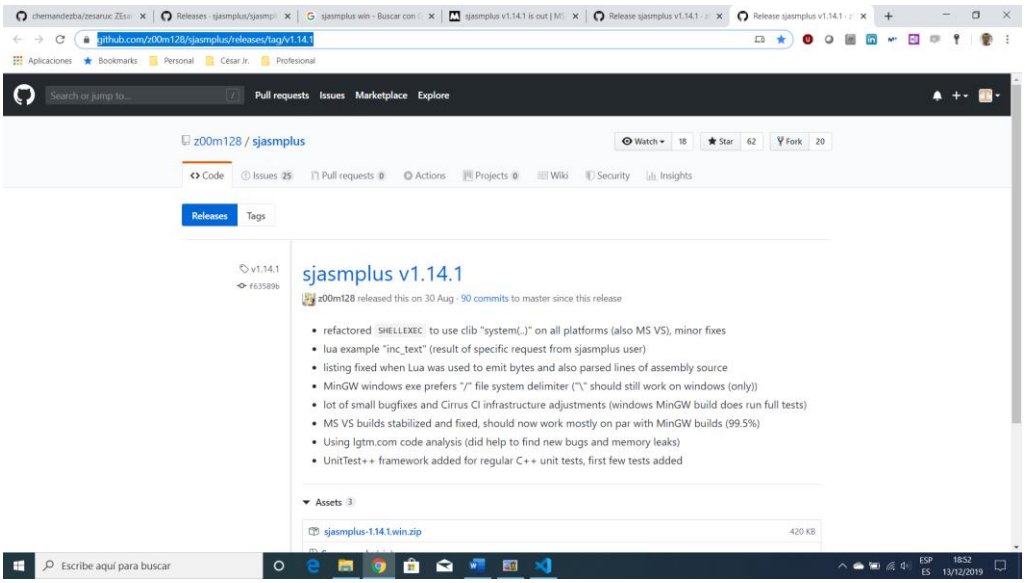


Du musst auch den JS-Knoten installieren (wenn du ihn noch nicht hast), den du hier finden kannst: https://nodejs.org/es/download/

- **Plugin Z80 Debug:** Plugin, das wir auf VS-Code installieren werden und das uns unter Verwendung des ZRCP-Protokolls die visuelle Fehlersuche ermöglicht. Den Quellcode und viele Dokumentationen zu diesem Plugin finden Sie unter dieser Adresse: https://github.com/maziac/z80-debug
  Obwohl Sie das Plugin auf einfache Weise direkt aus dem VS-Code installieren können, hat mir Thomas ein paar Versionen geschickt, um sie manuell aus dem VSCode zu installieren, die Dinge beheben, die in der "veröffentlichten" Version nicht funktionieren. Bis Thomas die offizielle Version 0.9.3.3 veröffentlicht, müssen Sie die.VSIX-Datei mit der Version 0.9.3-2 installieren, die mich speziell darauf vorbereitet hat, einen Fehler zu beheben, der es nicht erlaubte, den Wert der Register korrekt zu überprüfen, noch den Inhalt der Variablen und die Speicherpositionen zu sehen, auf die sie zum Zeitpunkt der Ausführung zeigen.

- **Sjasmplus Assembler:** Es wird das Werkzeug sein, mit dem wir aus der IDE-Umgebung unsere Programme zusammenstellen werden. Obwohl es fortgeschrittenere Betas gibt, arbeite ich mit der Version v1.14.1, die im August veröffentlicht wurde und die Sie von dieser Adresse herunterladen können: https://github.com/z00m128/sjasmplus/releases/tag/v1.14.1. Sie haben eine vollständige Dokumentation dazu unter diesen anderen Adressen:

http://z00m128.github.io/sjasmplus/documentation.html oder
https://github.com/sjasmplus/sjasmplus/wiki



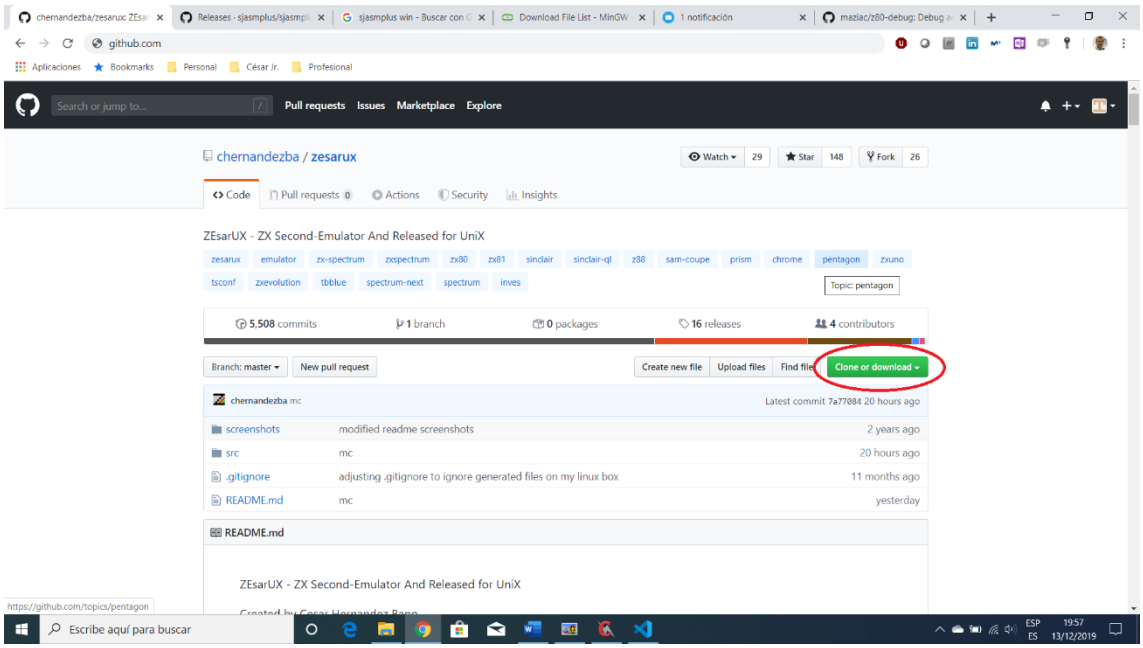
Wenn alle diese Elemente heruntergeladen und installiert sind, können wir mit der Arbeit beginnen!

3. Installation des ZEsarUX-Emulators

Wir beginnen mit dem "Herzen des Tieres", dem ZEsarUX-Emulator.

Schritt 1 - ZEsarUX Quellcode herunterladen

Klicken Sie auf der angegebenen Seite auf den Button **Clone oder Download** und laden Sie die Datei zesarux-master.zip in das Verzeichnis herunter, in dem wir den Emulator verlassen werden (in meinem Fall D:\Spectrum\ZEsarUX).



Sobald Sie die Datei heruntergeladen haben, entpacken Sie sie in das gleiche Verzeichnis und gehen Sie zum nächsten Punkt, nämlich der Installation der MinGW zur Kompilierung des ZEsarUX-Quellcodes und der .exe-Datei des Emulators.

Schritt 2 - Installation und Konfiguration der MinGW

Nach dem Herunterladen der Datei mingw-get-setup.exe der angegebenen Adresse führen Sie sie aus und installieren sie im Verzeichnis **C:\MinGW** mit allen Standardoptionen. Wenn die Installation abgeschlossen ist, öffnet sich der **MinGW Installation Manager** und Sie müssen die folgenden Pakete zur Installation auswählen:

In der **Grundeinstellung**:

- Mingw-Entwickler-Toolkit für Entwickler
- mingw32-Basis
- mingw32-gcc-gcc-g++++
- msys-base

In **allen Paketen** müssen Sie die folgenden Pakete hinzufügen:

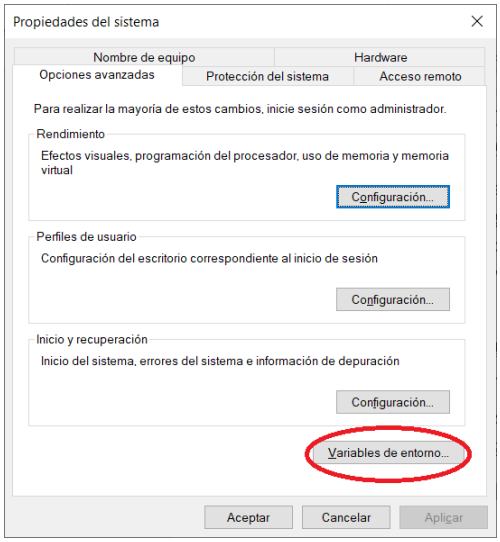
- mingw-pthreads (alle Pakete, die Sie unter diesem Namen sehen)
- msys-bash

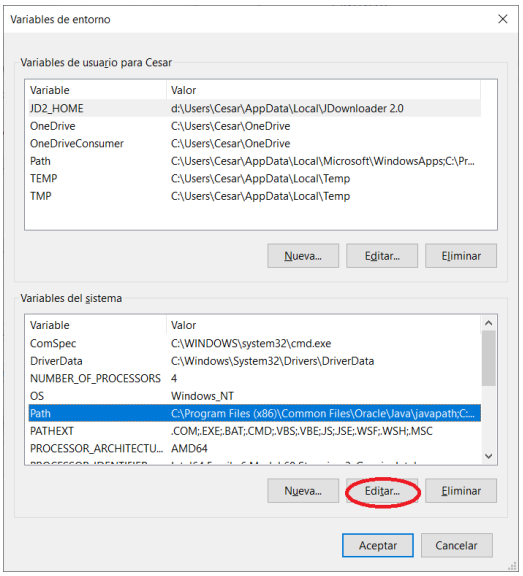
Nach der Auswahl dieser Pakete übernehmen wir die Änderungen und können den **MinGW Installation Manager** schließen. Jetzt müssen Sie zum Explorer gehen, die Datei, die wir zuvor mit den SDL-Bibliotheken heruntergeladen haben (**SDL-devel-1.2.15-mingw32.tar.gz**), in **C:\MinGW** kopieren und dekomprimieren. Dadurch wird ein Verzeichnis namens **SDL-1.2.15** erstellt, das wir einfach in **sdl** umbenennen werden.

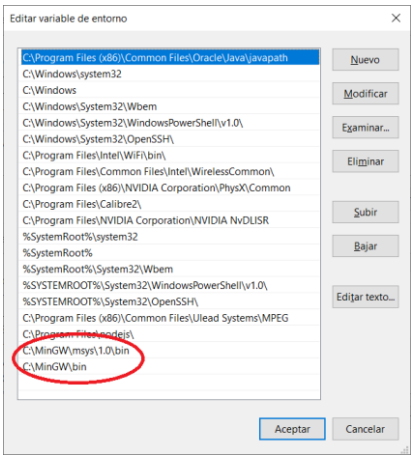
Wenn alles gut gelaufen ist, werden wir in **c:\mingw\sdl\lib** mehrere Dateien **libSDL.dll*** haben, und in **c:\mingw\sdl\include** werden wir einen Unterordner **SDL** haben, in dem es mehrere Dateien geben wird**. h**.

Unser letzter Schritt vor dem Kompilieren der.exe-Datei ist das Hinzufügen der Verzeichnisse **c:\mingw\bin** und **c:\mingw\msys\1.0\bin** zum System PATH.

Option a) über das Bedienfeld:







Option a) von der Kommandozeile aus:

Wir führen cmd.exe aus und schreiben **set PATH=%PATH%;c:\mingw\bin; c:\mingw\msys\1.0\bin**

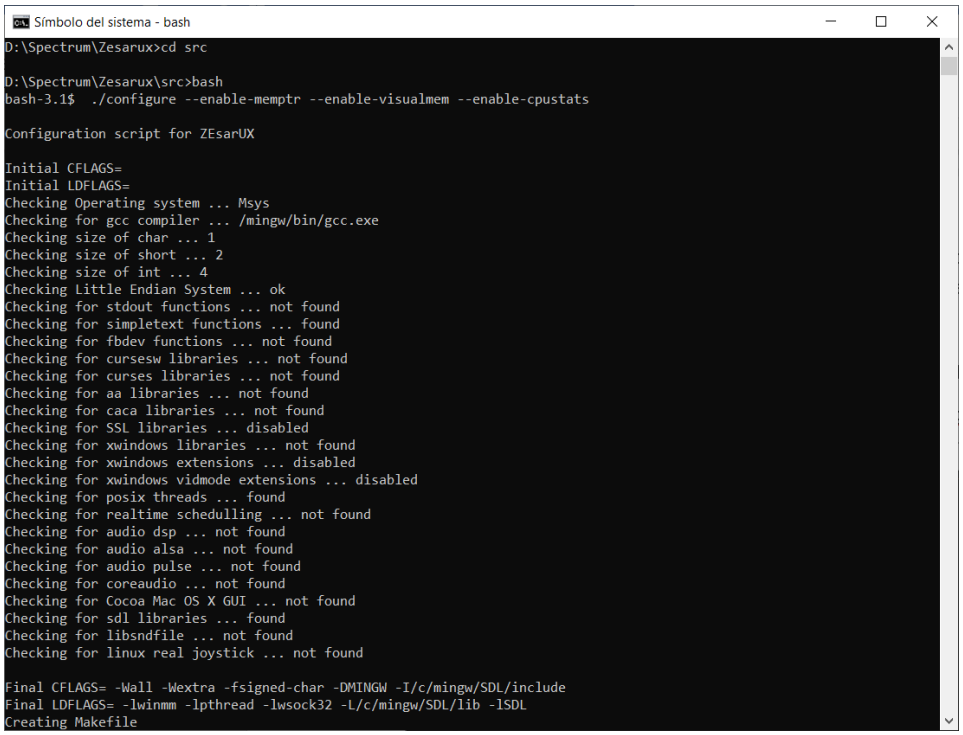
Schritt 3 - Kompilieren und Konfigurieren von ZEsarUX

Wir sind nun bereit, unsere ZEsarUX zu erstellen. Um dies zu tun, müssen wir cmd.exe ausführen, in das Verzeichnis wechseln, in dem wir den ZEsarUX-Quellcode entpackt haben (in meinem Fall d:\spectrum\ZesarUX\src) und den Befehl ausführen:

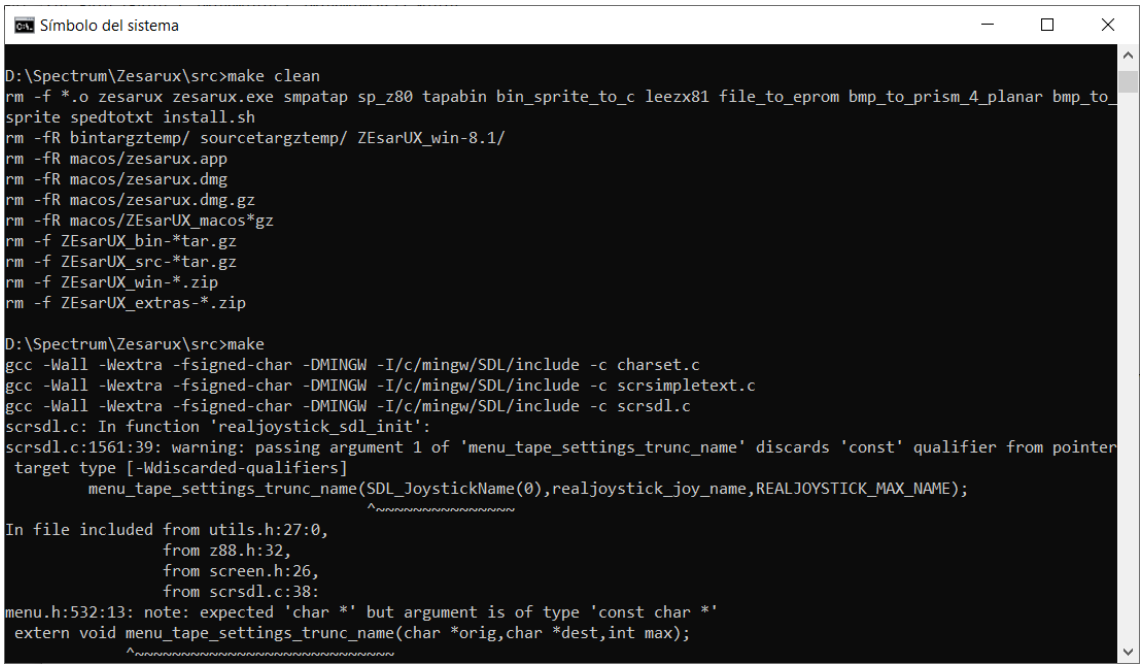
schlagen

uns wird eine Eingabeaufforderung "bash.3.1$ angezeigt, von der aus wir den folgenden Befehl starten werden

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



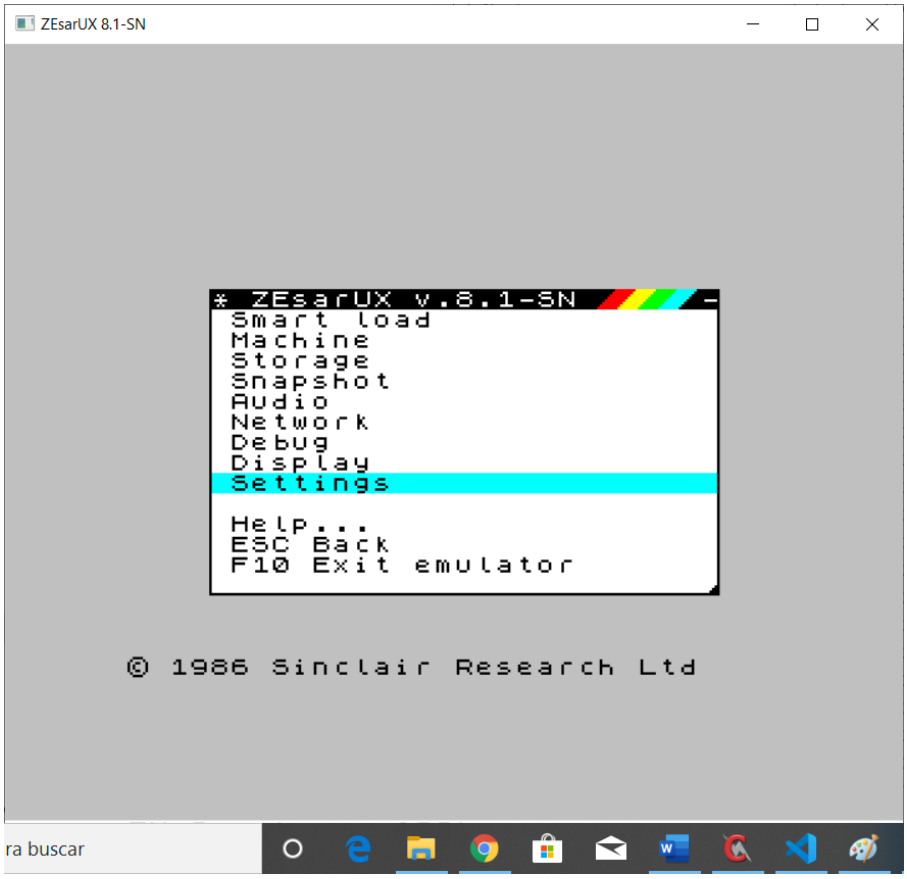
Wenn wir fertig sind, werden wir **make clean** und schließlich **make** ausführen, wodurch die Kompilierung gestartet wird. Wir lassen bash typing **exit** und, wenn alles gut gelaufen ist, werden wir in unserem src-Verzeichnis die brandneue Datei **zesarux.exe** mit der ausführbaren Emulator-Datei haben.



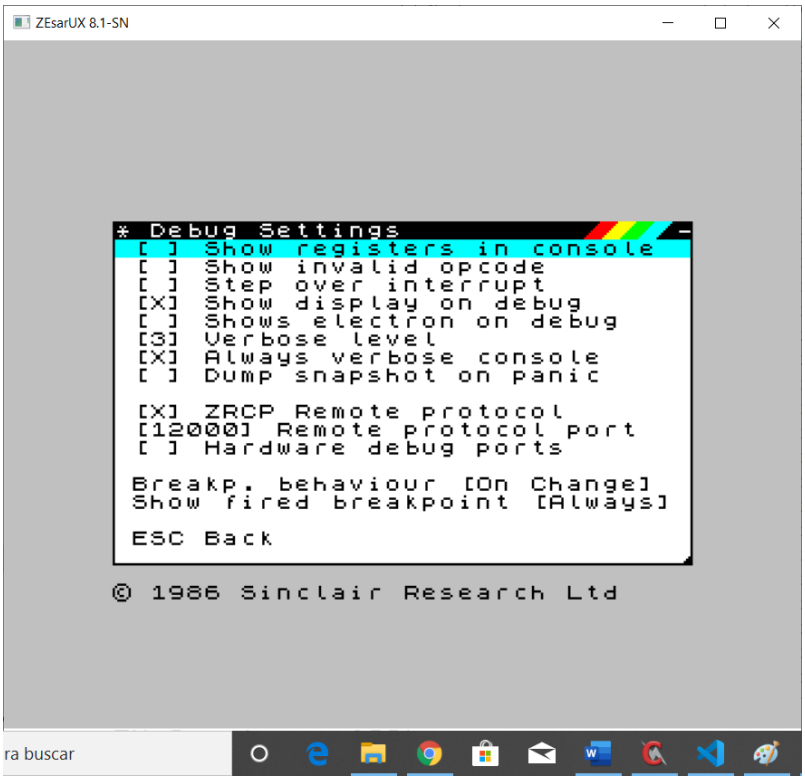
Unser letzter Schritt ist das Kopieren der **SDL.dll** von MinGW in das aktuelle Verzeichnis mit dem Befehl:

copy c:\mingw\sdl\sdl\sdl\dll.dll .

Erledigt.... wir können nun unsere brandneue ZesarUX.exe ausführen und die Konfiguration abschließen..... Sobald Sie es öffnen, wenn Sie mit der Maus klicken, sehen wir das folgende Menü:



Hier werden wir **Einstellungen** auswählen, im folgenden Menü **debuggen** und hier die Option **ZRCP Remote Protocol** markieren, die den Port festlegt, über den ZEsarUX mit dem Plugin kommunizieren soll (standardmäßig das 10000, obwohl es in meinem Fall zu einem Problem führte und ich es bis zum 12000 konfiguriert habe).

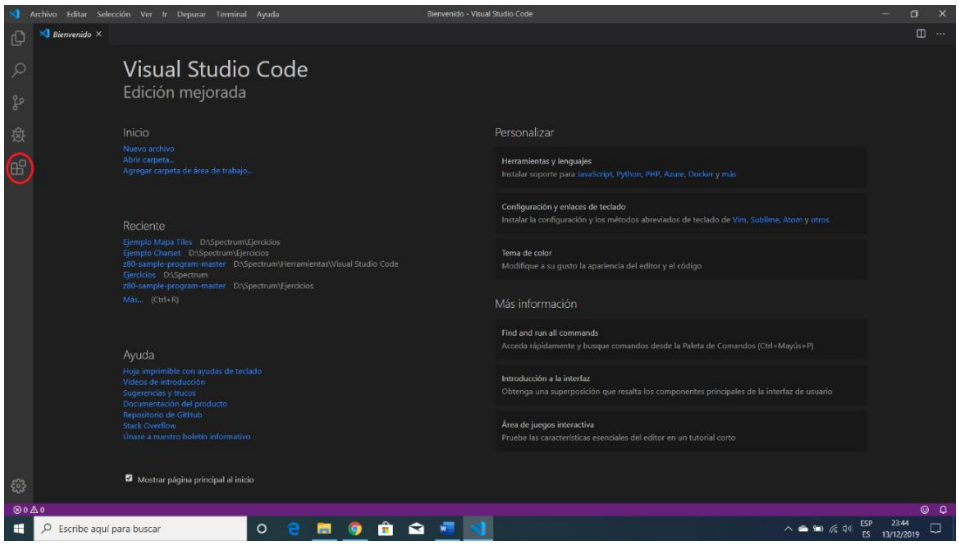


ZEsarUX hat Hunderte von interessanten und konfigurierbaren Aspekten, die Sie bei der Navigation über die Menüs sehen können. Um zu debuggen, müssen wir nichts anderes konfigurieren, und obwohl es nicht das Ziel dieses Tutorials ist, ermutige ich Sie von hier aus, "bichear" mit den Optionen zu wählen, damit Sie die ganze Leistung dieses großartigen Emulators sehen können. Ich bin sicher, dass mein Namensgeber erfreut sein wird, dass Sie sich Sorgen machen und das Beste aus seinen Emulatoroptionen machen.

4. Installation und Konfiguration des VS-Codes

Sobald die ausführbare Datei des VS Code Installers von der in Abschnitt 2 des Dokuments angegebenen Adresse heruntergeladen wurde, führen Sie sie einfach aus und folgen Sie den Anweisungen, wobei Sie bei der Installation die Standardoptionen übernehmen.

Nach der Installation, wenn Sie das Programm ausführen, finden Sie das folgende Fenster:

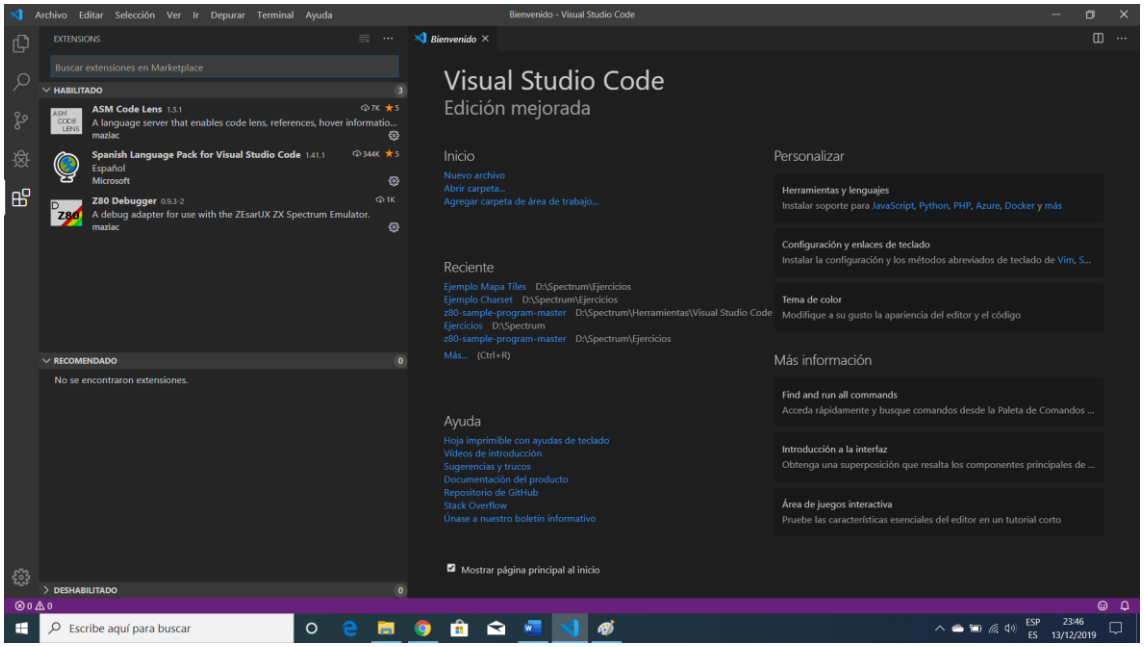


Deines wird nicht genau das gleiche sein, denn das System wird auf Englisch sein, aber jetzt werden wir damit weitermachen.

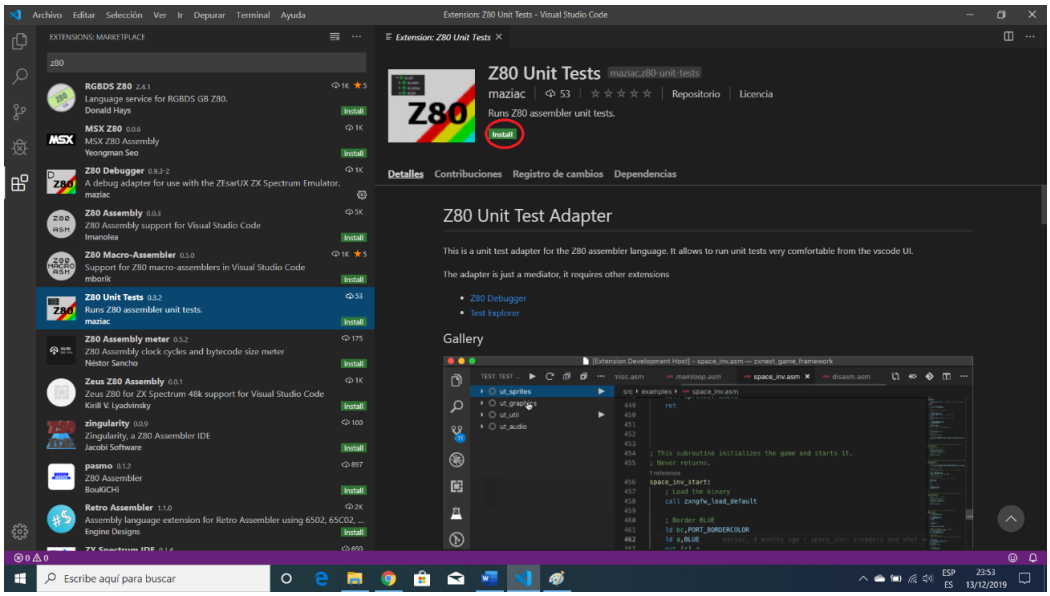
Noch ein Detail: Obwohl ich mich nicht erinnere, ob der Installer es automatisch gemacht hat, rate ich Ihnen, zu überprüfen, ob das Unterverzeichnis **bin** des Ordners, in dem Sie den VS-Code installiert haben (in meinem Fall d:\Microsoft VS Code\bin), dem Systempfad hinzugefügt wurde.

Schritt 1 - Installieren Sie die grundlegenden Plugins.

Das erste, was wir tun werden, ist, die zusätzlichen Basis-Plugins zu installieren. Klicken Sie dazu auf das Symbol, das ich in einem roten Kreis markiert habe, der uns zum Bildschirm für die Installation von Erweiterungen führt:

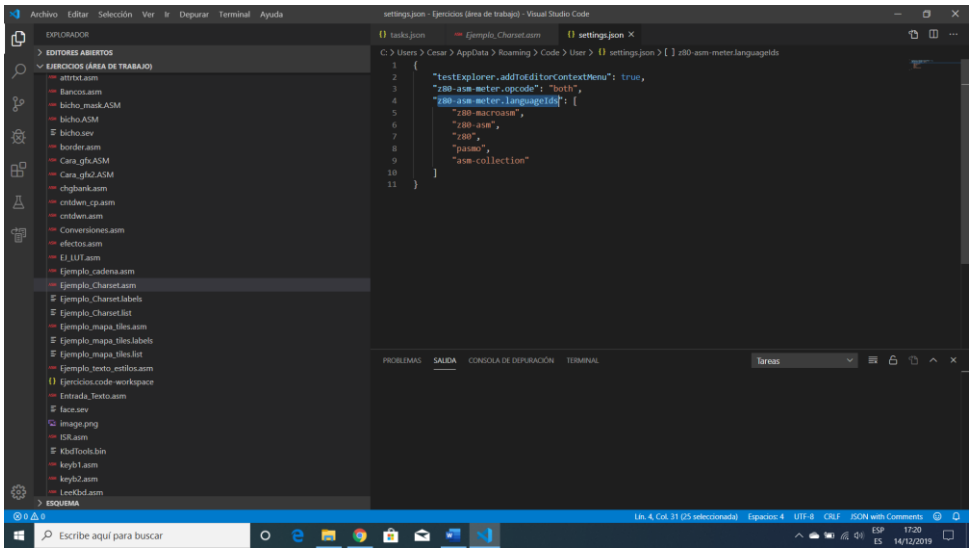


Um ein Plugin zu installieren, suchen Sie es einfach, klicken Sie darauf und klicken Sie im rechten Fenster auf die Schaltfläche "Install".

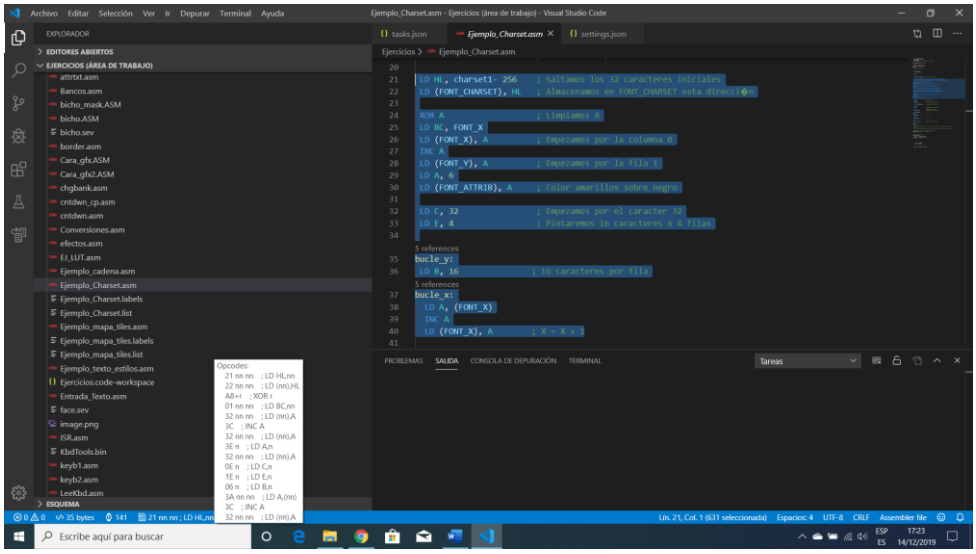


Nun werden wir die folgenden Plugins zusätzlich zum Z80 Debug installieren:

- **Spanisches Sprachpaket für Visual Studio Code.** So können Sie Ihren VS-Code auf Spanisch eingeben, wenn Sie sich bei der Arbeit in unserer Muttersprache wohler fühlen.
- **ASM Code Lens**, das wie das Haupt-Plugin Z80 Debug von Thomas Busse (alias Maziac) entwickelt wird. Dieses Plugin gibt uns Zugang zu einer Reihe von sehr interessanten Werkzeugen beim Debuggen (Position von Referenzen, Schweben von Variablen und Datensätzen, siehe die Anzahl der Referenzen auf ein bestimmtes Symbol, Assembler-Syntax-Hervorhebung....).
- **Z80 Unit Tests**. Auch von unserem Freund Maziac. Dieses Plugin ermöglicht es uns, bestimmte Komponententests in unsere ASM-Quellen aufzunehmen, um Prüfungen während der Debug-Zeit durchzuführen und die Ausführung zu stoppen, wenn der Test nicht erfüllt ist. Ich muss ihn ruhig im Auge behalten und sehen, was mit ihm gemacht werden kann, aber er sieht toll aus. Wenn ich sehe, dass es sich lohnt, verspreche ich, eine Erweiterung dieses Tutorials mit seiner Verwendung zu machen (wenn es jemand vorher für mich nicht tut;).
- **Z80 Assembly Meter**: Großartiges Plugin von Néstor Sancho, das die Anzahl der Taktzyklen bestimmt, die die Anweisungen verbrauchen, die wir aus dem Editor auswählen, sowie die Größe in Bytecodes der gleichen.
  Damit dieses Plugin funktioniert, müssen Sie eine kleine Anpassung vornehmen:
  - Öffnen Sie die Bedienkonsole mit **Strg+Shift+P'.**
  - Wählen Sie Einstellungen Öffnen Einstellungen **JSON**.
  - Im Eintrag **z80-asm_meter-languageIds** fügen wir**, "asm-collection"** hinzu.



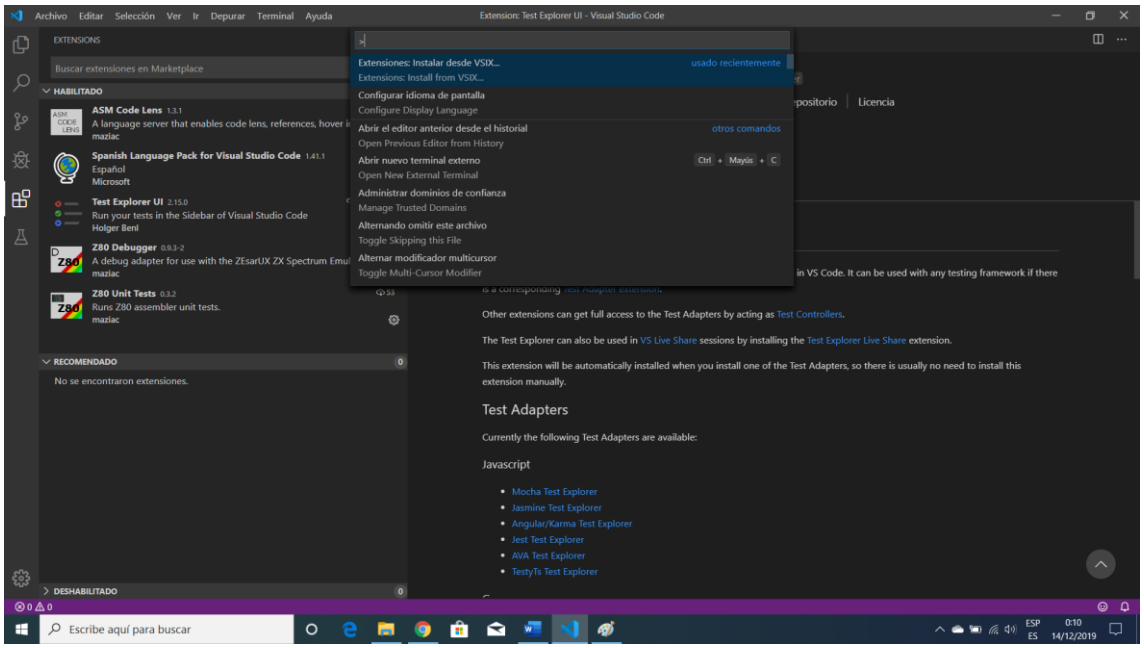
Jedes Mal, wenn wir eine oder mehrere Codezeilen auswählen, zeigt uns die Statusleiste die Größe in Bytes, die Anzahl der Taktzyklen, die sie verbrauchen wird, und die Opcodes (oder Opcodes) der ausgewählten Operationen an.



Schritt 2 - Installieren des Z80 Debugger Plugins

Obwohl dieses Plugin auch über den Installationsbildschirm der Erweiterungen installiert werden kann, finden Sie dort die Version v.0.9.2, und leider enthält diese Version einen Fehler, der viele Probleme beim Debuggen verursacht, so dass Sie die Beta v.0.9.3-2 installieren müssen, die Thomas Busse mir gegeben hat (und die Sie im gleichen zip finden werden, in dem sich dieses Dokument befindet), das einen etwas anderen Installationsprozess hat.

Der erste Schritt ist das Entpacken der Datei **z80-debug-0.9.3-2.vsix** über einen Arbeitsordner. Nun, aus dem VS-Code, gehen wir zum Menü "Ansicht" und wählen dort die Option **Befehlspalette**.



Dabei öffnet sich ein Fenster, in dem Sie nach "**Erweiterungen: Installation aus VSIX-Datei**" suchen sollten. Wählen Sie nun die Datei **z80-debug-0.9.3-2.vsix,** akzeptieren und sehen Sie, wie unser Plugin perfekt installiert ist. Wir sind gerade dabei, es zu beenden.

Schritt 3 - Installieren Sie node.js und den SJASMPLUS-Assembler.

Die letzten beiden Tools, die wir installieren müssen, sind node.js und SJASMPLUS. Für den ersten von ihnen reicht es aus, den **Datei-Node-v12.13.1-x64.msi** auszuführen, den wir heruntergeladen, ausgeführt und mit den Standardoptionen installiert haben.

Die Installation von SJASMPLUS ist noch einfacher. Nach dem Herunterladen der Datei sjasmplus-1.14.3.win.zip der angegebenen Adresse müssen Sie sie nur noch auf das gewünschte Verzeichnis entpacken (in meinem Fall d:\Spectrum\tools\sjasmplus-1.14.3.win) und dieses Verzeichnis in den Pfad des Systems aufnehmen, wie im Installationsschritt von MinGW erläutert.

Nach diesem letzten Schritt können wir unseren VS-Code öffnen und uns auf das Debuggen vorbereiten - gehen wir zur Sache!

5. erstellen unserer integrierten IDE für das Debugging

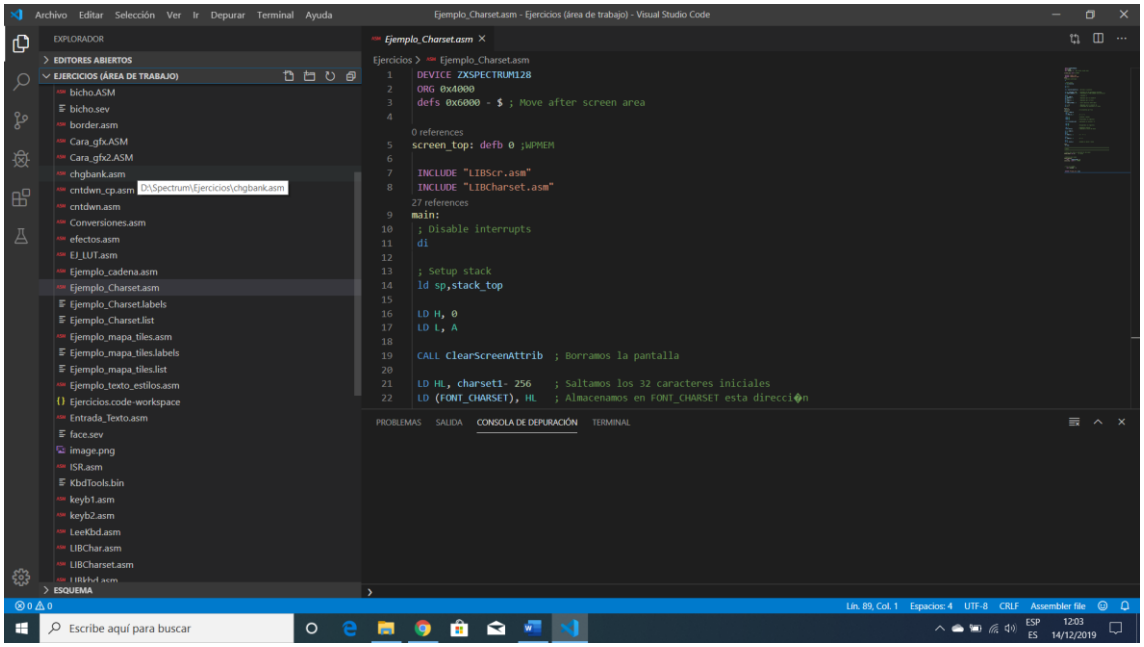
5.1 - Öffnen des Ordners mit unserem Beispielprogramm

Um das Debugging mit unserer neuen IDE zu veranschaulichen, werde ich ein paar Beispielprogramme verwenden, die im **Compiler-Softkurs** enthalten sind: eines zeigt einen vollständigen Zeichensatz und ein weiteres, das die Map einer Ebene Ihres Sokoban-Programms zeichnet. Sie finden die Quellen (und die Dateien **tasks.json** und **launch.json**) in der Datei **Examples.rar im** Anhang dieses Tutorials.

Wenn Sie die Datei entpacken, finden Sie den folgenden Inhalt:

- **LIBSprites.asm**: Bibliothek mit Routinen zum Umgang mit Sprites.
- **LIBScr.asm**: Bibliothek mit Routinen zur Bildschirmverwaltung.
- **LIBCharset.asm**: Bibliothek mit Routinen zur Behandlung von Zeichensätzen.
- **Beispiel_Charset.asm**: Datei mit der Hauptroutine des ersten Beispiels
- **Example_charset.list und . labels***:* werden während der Assembly generiert und sind für das Debugging notwendig.
- **Beispiel_map_tiles.asm**: Datei mit der Hauptroutine des zweiten Beispiels
- **Beispiel_map_tiles.list und . labels***:* werden während der Assembly generiert und sind für das Debugging notwendig.
- **. vscode Ordner** mit den Dateien **tasks.json** und **launch.json**, notwendig, um die Assembly aus dem VS-Code zu starten und mit dem Debuggen zu beginnen.

Entpacken Sie die Zip-Datei in einem Arbeitsordner und starten Sie den VS-Code. Das Ideal ist es, einen neuen Arbeitsbereich zu erstellen (Option **Archvo\Arbeitsbereich speichern unter...**) und darin den Ordner aufzunehmen, in den wir den Code heruntergeladen haben, von der Option **Datei\add Ordner zu Arbeitsbereich**. Wenn wir es gut gemacht haben, sehen wir einen ähnlichen Bildschirm wie diesen:



In der Leiste, die wir auf der linken Seite haben, werden wir hauptsächlich zwei Optionen verwenden:

- Explorer (erstes Symbol oben beginnend), um durch die Dateien zu navigieren.
- Debuggen und Ausführen (Symbol des "Fehlers"), um das Debugging zu starten.

5.2 - Tasks.json und launch.json Dateien

In all unseren Projekten müssen wir ein Verzeichnis namens **. vscode** haben, das vom Stammverzeichnis hängt, wo wir zwei sehr wichtige Dateien haben werden: die **tasks.json** und die **launch.json.** Obwohl sie auf andere Weise erstellt werden können, rate ich Ihnen, dass Sie eine Masterkopie dieses Verzeichnisses haben und für jedes Projekt kopieren und die Dateien entsprechend ändern, da es die bequemste Option ist.

Lassen Sie uns nun über diese Dateien sprechen, wofür sie bestimmt sind und wie sie verwendet werden.

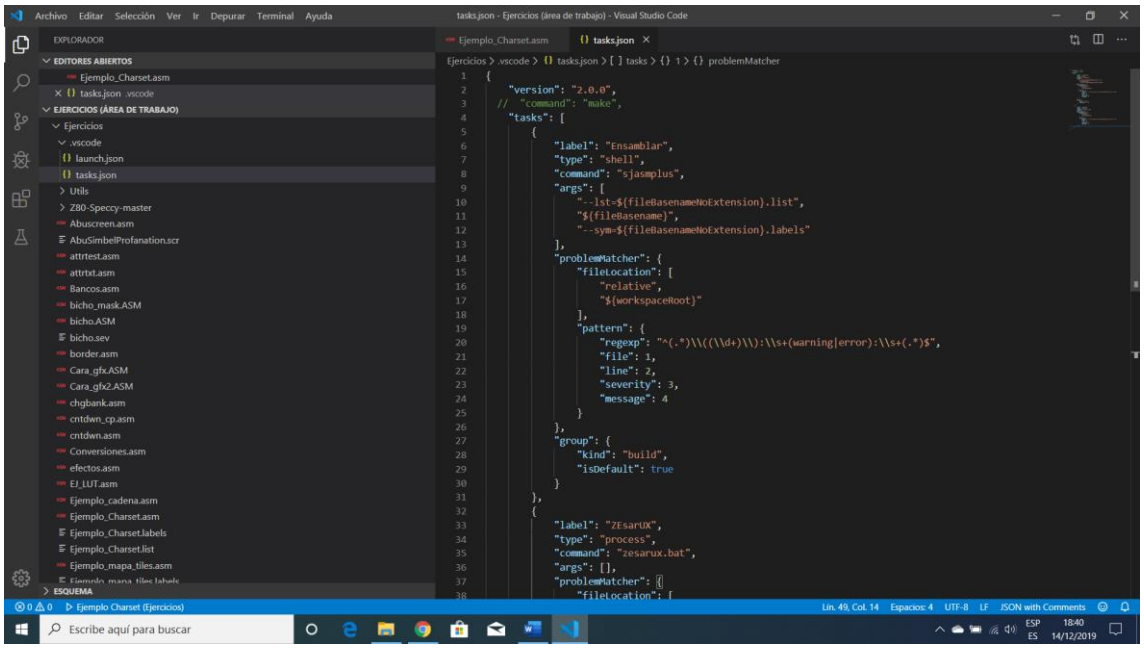
TASKS.JSON DATEI

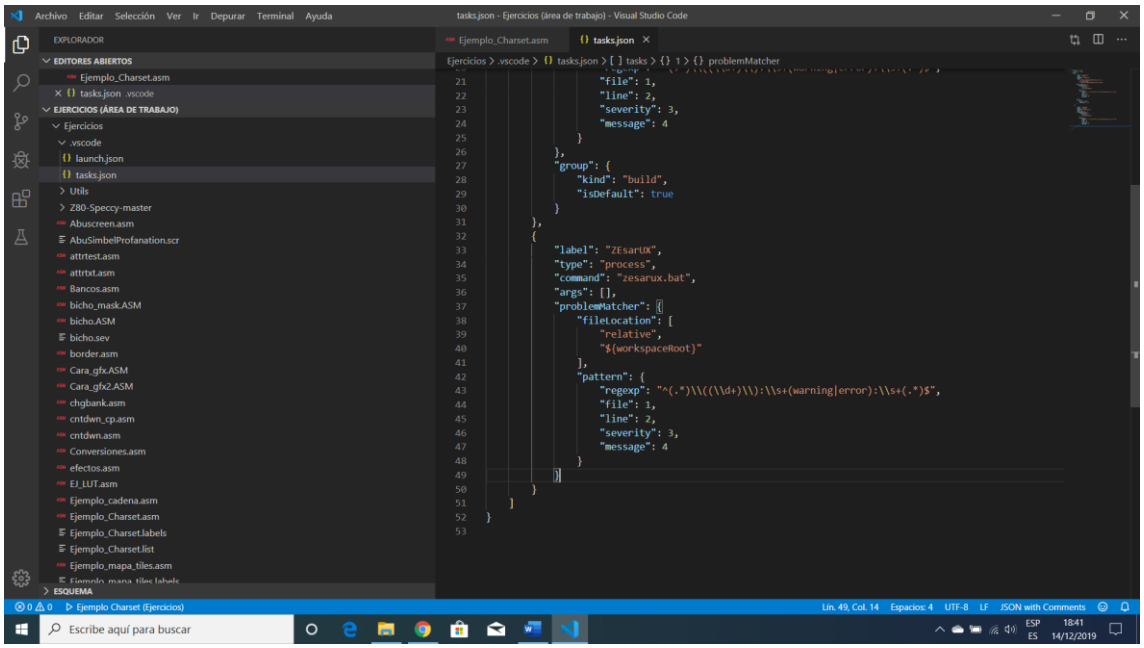
Durch diese Datei können wir Befehle oder Aufgaben definieren, die wir aus dem VS-Code starten können. In unserem Fall werden wir zwei Aufgaben schaffen:

- Eine, die **sjasmplus** aufruft, um unseren Code zu assemblieren und die Dateien mit **den Bezeichnungen . lst** und **. zu** erzeugen, die der Debugger für das Debuggen benötigt.
- Ein weiterer, der ZEsarUX anruft, bevor er mit dem Debugging beginnt. Für diesen zweiten Fall erstellen wir eine Datei namens zesarux.bat, die im Verzeichnis von ZEsarUX abgelegt wird und führen sie aus. In meinem Fall enthält diese Datei die folgenden Befehle:

  d:
  cd
  cd-spektrum
  cd zesarux
  cd src
  zesarux
  Ausgang

Obwohl wir mehr Aufgaben definieren können, wird unsere Datei tasks.json die folgende Struktur haben:





Lassen Sie uns die Haupt-Tags dieser Datei sehen:

- **Label**: Es enthält den Namen, unter dem wir die Aufgabe sehen werden, wenn wir versuchen, sie von der **Terminal-Option\ausführen Aufgabe** auszuführen**.....**
- **Typ**: Hier geben wir an, dass die Aufgabe auf einer Shell ausgeführt wird.
- **Befehl**: Name des Befehls, den wir von der Shell aus starten werden, in unserem Fall **sjasmplus.**
- **Args:** Es wird eine Liste mit den verschiedenen Parametern enthalten, die an den Befehl übergeben werden, der von der Shell ausgeführt wird. In unserem Fall sind dies der Name der zu erzeugenden **Listendatei** (unsere Hauptdatei ohne Erweiterung, indem wir ihr **eine Liste** hinzufügen), die zu erstellende Datei und der Name der zu erzeugenden Etikettendatei (wiederum der Name

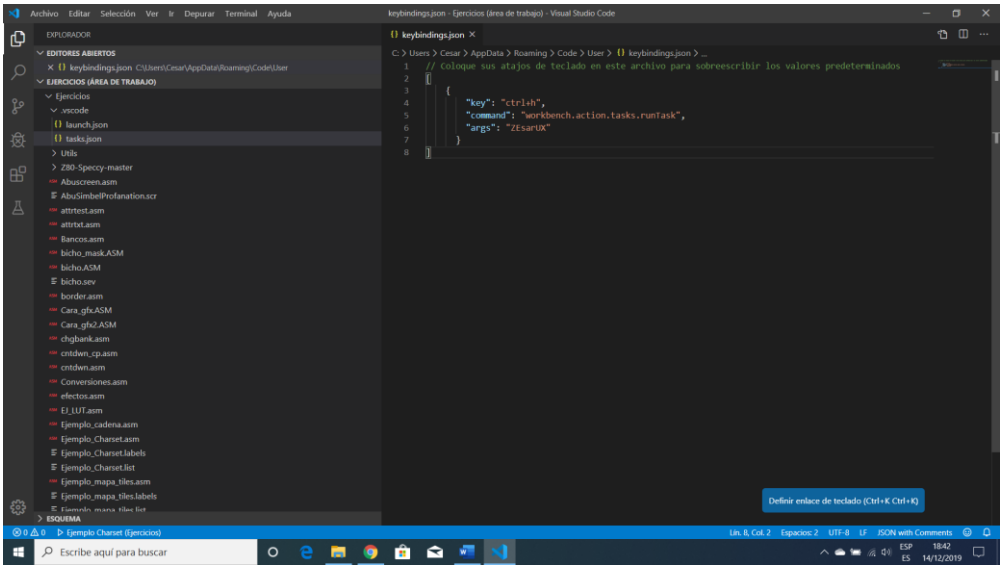
unserer Hauptdatei ohne Erweiterung, indem wir ihr **eine Bezeichnung** hinzufügen):

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

- Alle anderen Tags sind Standard. Unter ihnen, um **die Gruppe** zu betonen, wo wir feststellen werden, dass unsere Aufgabe eine Aufgabe der Zusammenstellung ist und dass es zusätzlich die Standardaufgabe sein wird, um sie starten zu können, indem wir **capital+ctrl+b** drücken.

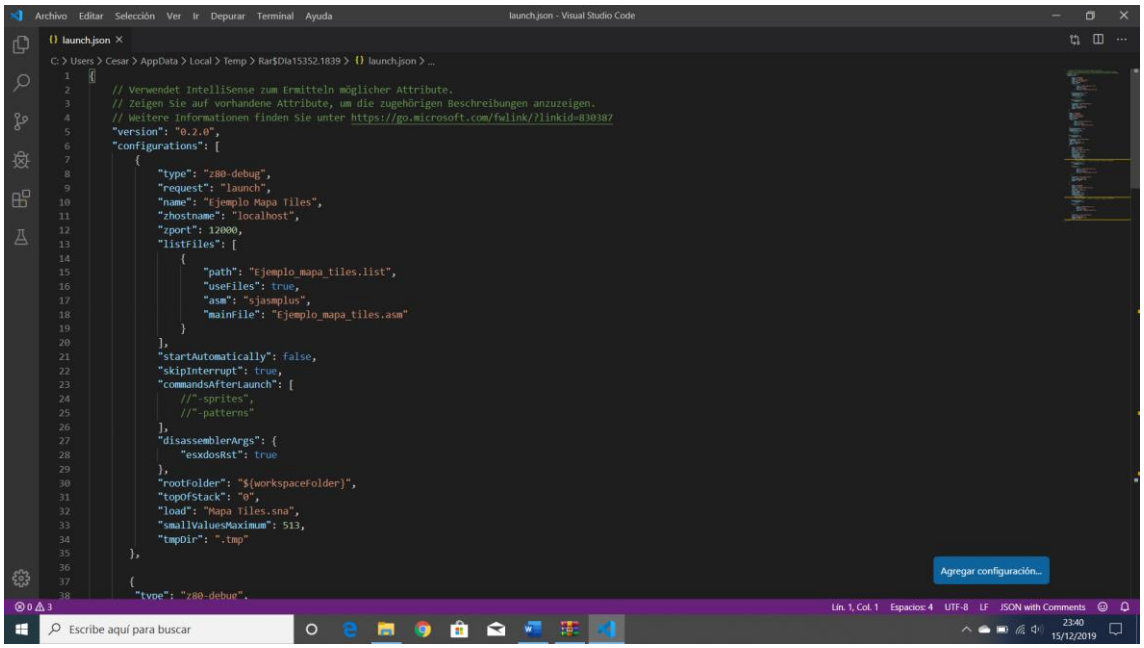
In der im Bild gezeigten Tasks-Datei erstellen wir die beiden Tasks: eine zum Aufruf des Assemblers, der mit **capital+ctrl+b** ausgeführt wird, und eine andere zum Aufruf von ZEsarUX. Um es zu starten, werden wir eine Kombination von Schlüsseln zuweisen: **ctrl+z**:

- Wählen Sie im Menü **View\Commands Palette.... die** Option **Open keyboard shortcut (JSON).**
- Wir fügen den Code hinzu, der das Bild und die Aufzeichnung anzeigt.



DATEI LAUNCH.JSON

Dies ist die Datei, mit der das Plugin das Debugging startet. Diese Datei muss einen Eintrag für jedes Programm haben, das wir debuggen möchten. Obwohl es auf der Seite des Plugins detaillierte Informationen über diese Datei gibt, gibt es hier einige grundlegende Erklärungen, die ausreichen werden, um Ihnen den Einstieg zu erleichtern:



Nach dem ersten Version-Tag gibt es ein Tag namens **"configurations", in** dem wir einen Eintrag für jedes Programm haben müssen, das wir debuggen wollen. Die Struktur jedes Blocks ist wie folgt:

```
{
            "type": "z80-debug",
            "Anfrage: Start,
            " Name": "Beispiel Kacheln Karte",
            "Zhostname": "localhost",
            "zport": 12000,
            " listFiles": [[
                {
                    " Pfad": "Beispiel_Karte_Kachelliste",
                    "useFiles: true,
                    "asm": "sjasmplus",
                    "mainFile": "Beispiel_map_tiles.asm"
                }
            ],
            "startAutomatically: false,
            "skipInterrupt: wahr,
            "commandsAfterLaunch": [
                /"-sprites",
                / /"-Muster"
            ],
            " DisassemblerArgs": {
                "esxdosRst": true
            },
            "rootFolder: ${workspaceFolder},
            "topOfStack": "stack_top",
            " Laden": "Mapa Tiles.sna",
            "smallValuesMaximum": 513,
```

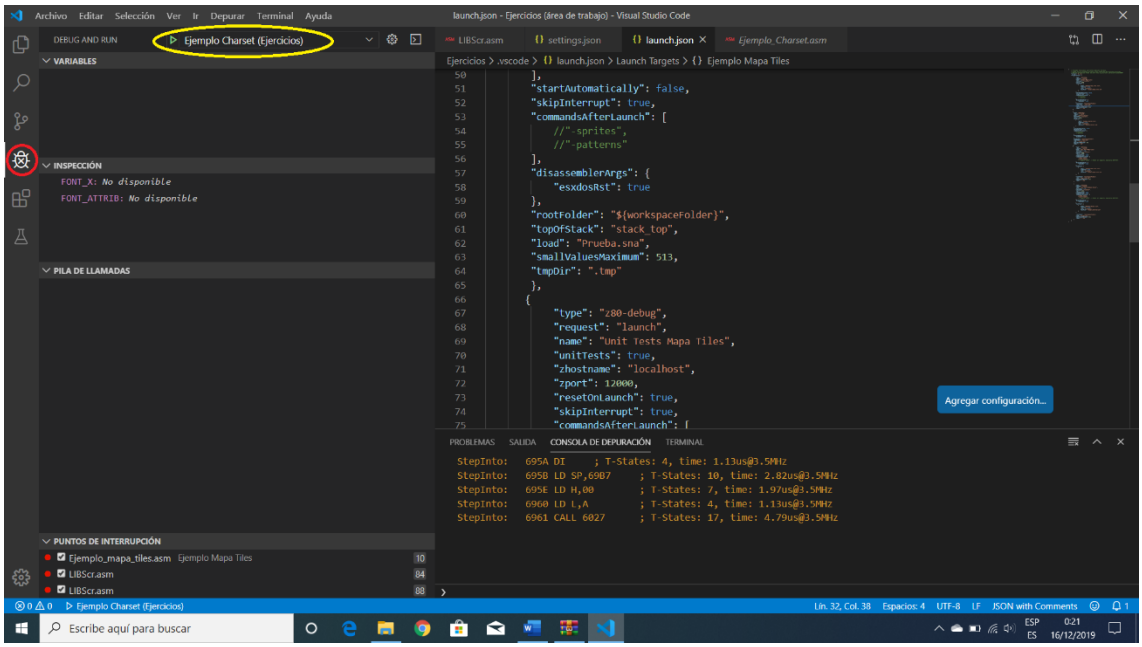
```
            " tmpDir": ". tmp"
        },
```

Für jeden neuen Eintrag müssen wir die folgenden Änderungen vornehmen:

- Im Tag **"name"** tragen wir den Namen ein, mit dem er in der Liste der Debugging-Tasks erscheinen soll (die wir später sehen werden).
- Wenn wir nicht gegen einen Remote-Server debuggen, werden wir in **"zhostname"** immer **"localhost"** lassen**.**
- In **"zport"** geben wir die Portnummer an, die wir in ZEsarUX für die Kommunikation über das ZRCP-Protokoll konfigurieren (in meinem Fall, wenn Sie sich erinnern, 12000).
- In der Struktur **"listfiles" müssen Sie** nur die Namen der Einträge **"path"** (so dass sie auf die mit sjasmplus erzeugte Listendatei zeigen) und **"mainfile"** ändern, die den Namen der zu debuggenden Hauptdatei enthalten.
- Im Tag **"topOfStack"** denken wir über **"stack_top"** nach, das ist eine Variable, die wir in unseren ASM einfügen werden, mit dem Code, den ich später erkläre, damit das Plugin den Stapel von Aufrufen korrekt anzeigen kann (WARNUNG: In der Beispieldatei, die dem Tutorial beigefügt ist, kommt versehentlich eine **"0".** Du musst diesen Wert auf **"stack_top"** ändern, wenn du willst, dass der Aufrufstapel korrekt funktioniert)-
- Aus dem Rest der Struktur müssen Sie nur das "load"-Tag mit dem Namen der .SNA-Datei ändern, die wir beim Assemblieren erzeugen werden.

Wenn Sie sich die **launch.json** des Beispiels ansehen, das dem Tutorial beiliegt, werden Sie sehen, dass es wirklich vier Einträge in "**Konfigurationen"** hat: die ersten beiden sind die notwendigen, um die Beispielprogramme zu kompilieren; die anderen beiden sind die notwendigen Konfigurationen, um die Unit-Tests aufzurufen, die aus dem Beispiel kopiert werden, das dem Plugin beiliegt, aber ich konnte sie nicht fertig konfigurieren, selbst wenn Sie sie ignorieren können.

Sobald die Datei **launch.json** fertig ist**,** können wir zum Debug-Fenster gehen und die entsprechende Option in der linken Leiste drücken (mit einem roten Kreis markiert):

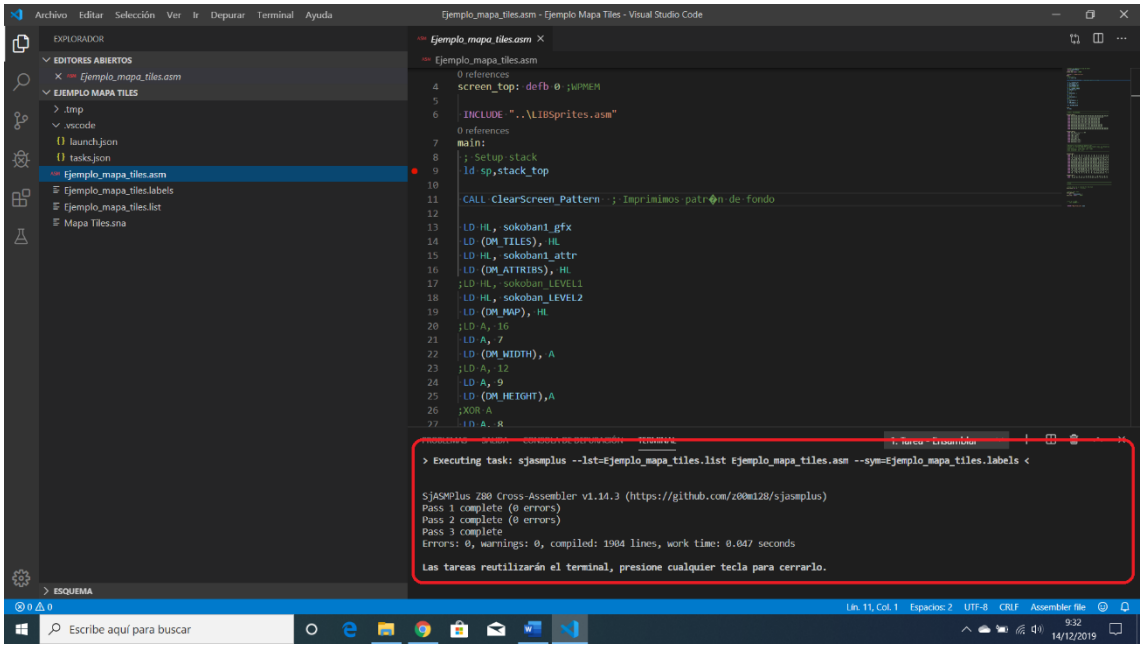


In diesem Bildschirm haben wir oben links (mit einem gelben Kreis markiert) einen grünen Pfeil, der es ermöglicht, das Debugging zu starten, und rechts davon eine Kombination, in der alle Konfigurationen der Datei **launch.json** erscheinen, identifiziert durch den Wert, den wir in jeden von ihnen im Tag **"name"** eingeben**.**

Wenn Sie das Debugging von hier aus starten, ist es wichtig, dass Sie ZEsarUX zuvor geöffnet haben (erinnern Sie sich, ctrl+z), sonst erhalten Sie einen fehlgeschlagenen Verbindungsfehler mit dem Emulator.

5.3 - Zusammenbau des ASM-Codes mit SJASMPLUS

Um zu sehen, dass alles gut definiert ist, wählen wir im Explorer die Datei example_map_tiles.asm und drücken die angegebene Tastenkombination, wir starten die Assembly, deren Ergebnis wir in der Debug-Konsole sehen werden:



Für diejenigen unter Ihnen, die aus der PASMO-Welt kommen, gebe ich Ihnen einige notwendige Syntaxänderungen zwischen PASMO und sjasmplus, die Sie in Ihren Programmen vornehmen müssen, wenn Sie nicht viele Montagefehler haben wollen:

- Alle Etiketten müssen in der ersten Spalte beginnen und keinen Platz lassen, da sie sonst als Anweisungen oder Anweisungen betrachtet werden, mit dem daraus resultierenden Fehler.
- Alle Anweisungen oder Richtlinien müssen mindestens einen Raum lassen, um nicht als Etiketten betrachtet zu werden.
- Unser Programm sollte mit der DEVICE-Direktive beginnen, die den Maschinentyp angibt, für den wir die SNA-Datei erzeugen werden (ZXSPECTRUM48, ZXSPECTRUM128, etc.).
- Sie müssen ein Label für die Eingaberoutine definieren, das wir dann an die Direktive übergeben, die die SNA-Datei erzeugt, mit der wir debuggen werden.
- Batteriedefinition: Obwohl ich nicht tief genug gegangen bin, scheint es, dass man die Start- und Endzonen der Batterie definieren muss. Es ist notwendig, den folgenden Code hinzuzufügen:

```
;=================================================================
==========
Stapeln.
;=================================================================
==========

Stapel: Dieser Bereich ist für den Stapel reserviert.
STACK_SIZE: equ 10 ; in Worten

Stackplatz reservieren
stack_bottom:
    defs STACK_SIZE*2, 0
stack_top: defb 0 ; WPMEM
```

Zusätzlich zum nächsten Laden in SP zu Beginn der Hauptroutine:

```
Einrichten des Stapels
ld sp,stack_top
```

5.4 - Start der Debug-Sitzung

Nun, der große Moment ist gekommen: Lasst uns das erste Beispiel reinigen. Dafür:

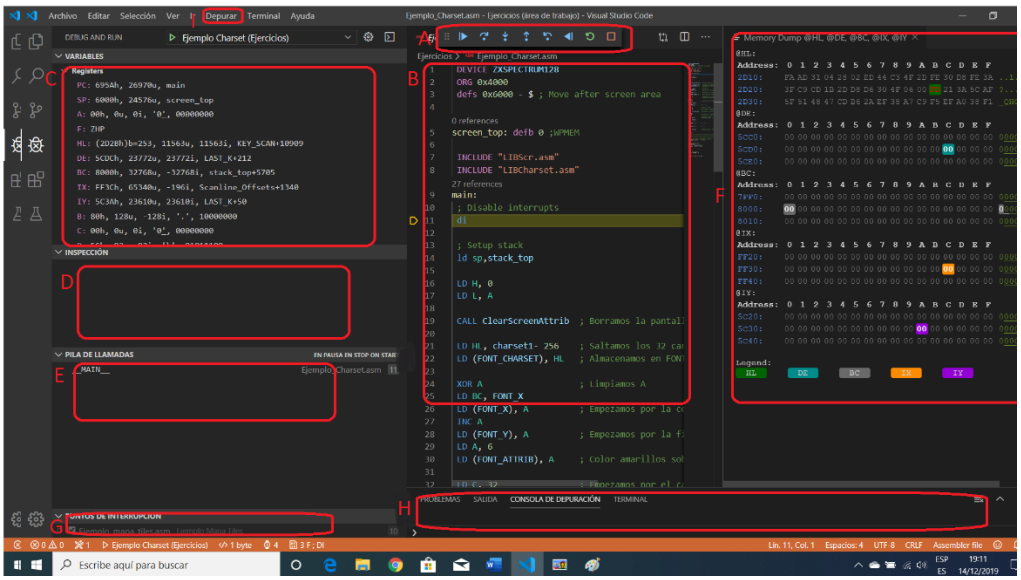
- open Beispiel_charset.asm
- Wir werden es zusammenbauen, indem wir **Strg+Mayus+b** drücken.
- wir starten ZEsarUX mit **ctrl+z**

- Drücken Sie F5, um das Debugging zu starten, oder gehen Sie zum Debugging-Fenster, wählen Sie den zu startenden Test aus und drücken Sie den grünen Pfeil.

6. Einführung in das Debugging

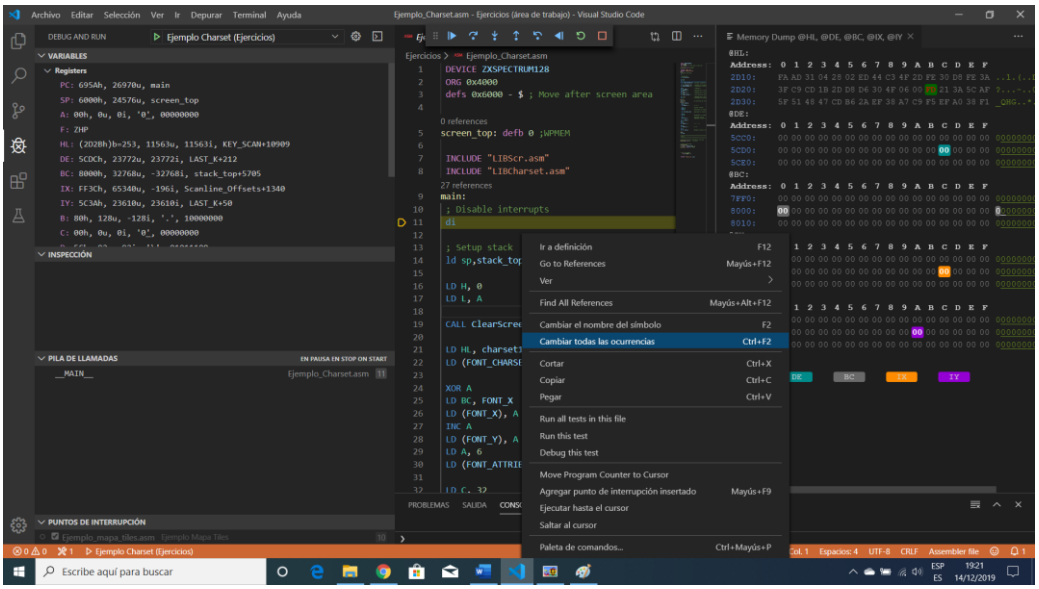
Obwohl das Tool erweiterte Debuggingoptionen (bedingte Haltepunkte, Inspektion bestimmter Speicherbereiche usw.) zulässt, hatte ich immer noch keine Zeit, mich mit ihnen zu befassen, und ich kann Ihnen in dieser Hinsicht wenig sagen, so dass Sie ein wenig die Dokumentation untersuchen müssen, die ich am Anfang des Dokuments erwähnt habe, und versuchen, versuchen und versuchen.

Auf jeden Fall, wenn ich Ihnen einen kleinen Überblick darüber geben kann, welche grundlegenden Tools die IDE uns zum Debuggen anbietet:

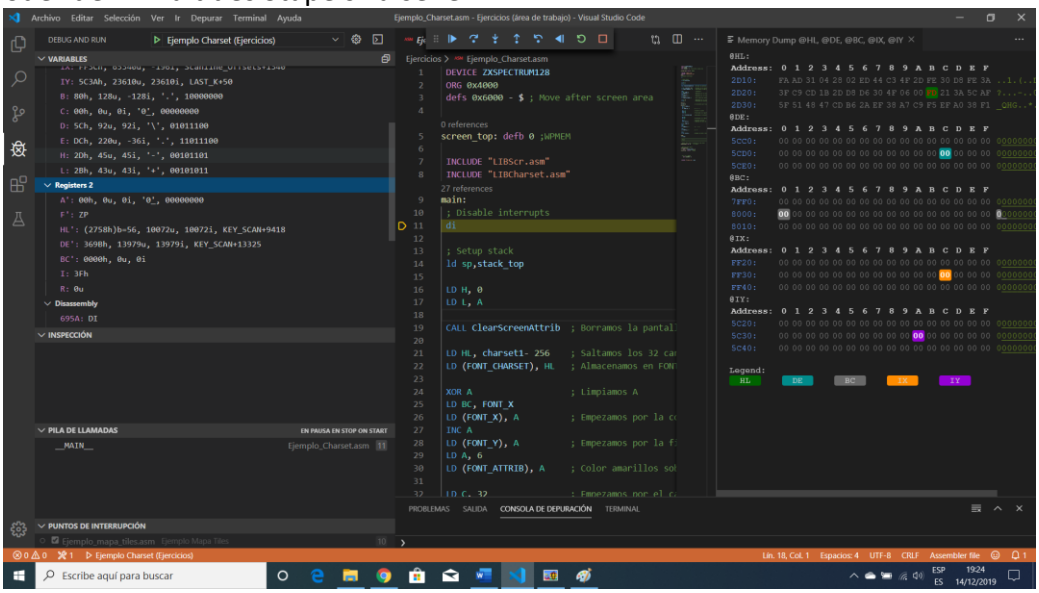


Die Bereiche, die ich markiert habe, sind:

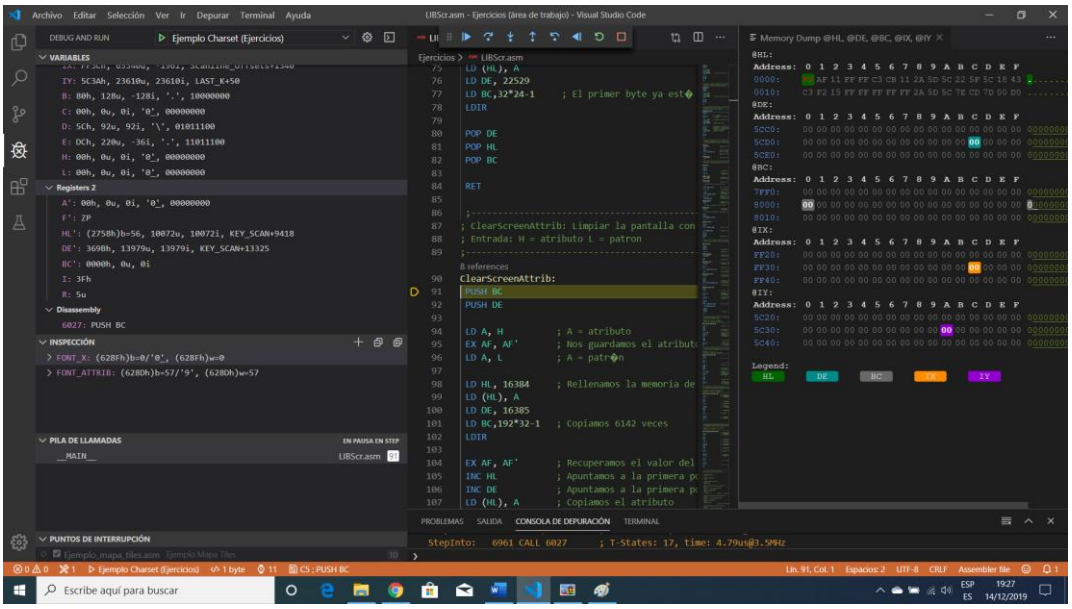
- **A - Befehlsleiste**: Bietet die üblichen Optionen zum Debuggen, wie z.B. Fortfahren, Schritt-für-Schritt-Anleitungen, Schritt-für-Schritt-Anleitungen, Verlassen der Routine, Sichern der Ausführung, Rückwärtsfahren, Neustarten und Beenden des Debuggens. Im Codefenster (markiert B) sind die ausgeführten Anweisungen grün markiert.
- **B - Codezone**: Zeigt den Code an, den wir ausführen. Wenn wir den Mauszeiger über eine Variable oder ein Label bewegen, sehen wir ihren Wert und den Inhalt des Speicherbereichs, auf den sie zeigt. Durch Drücken der rechten Taste können wir die Definition einer Routine oder Variable, die Referenzen, die sie im Code hat, sehen und durchlaufen, bis zu dieser Position ausführen, etc.



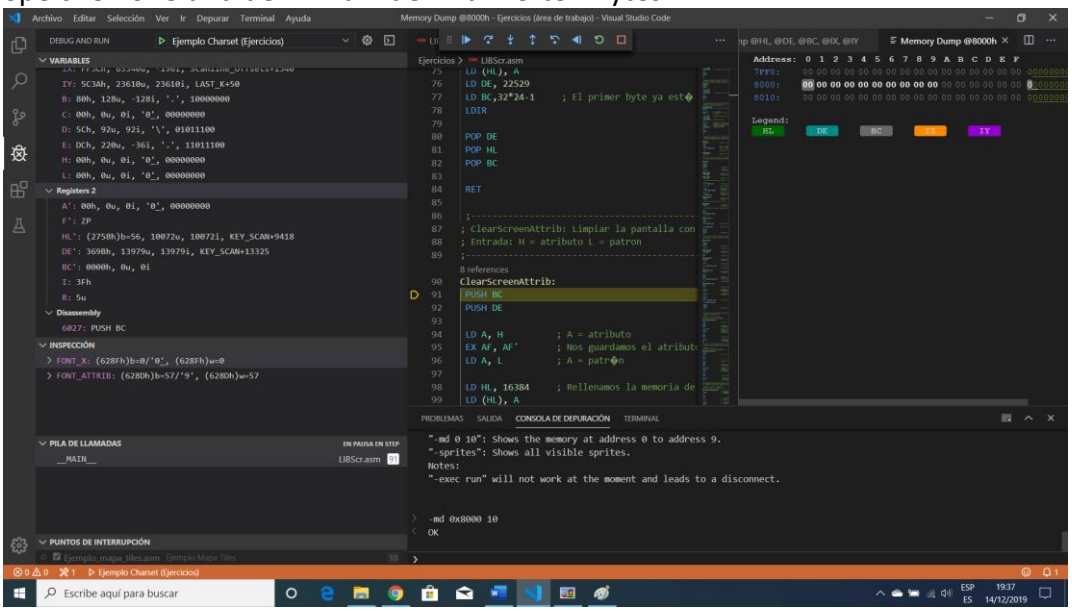
- **C - Zone der Variablen:** Es erlaubt, die Werte der Register zu sehen und zu ändern, Register Schatten, um den disassemblierten Code, die Speicherzonen oder den Inhalt des Stapels zu sehen.



- **D - Prüfzone**: Ermöglicht das Hinzufügen von Variablen, die während der Ausführung des Programmcodes überprüft werden sollen.



- **E - Stapel von Aufrufen**: Es zeigt uns an, in welchem Punkt der Ausführung wir uns genau befinden.
- **F - Speicherprüfzone:** Ermöglicht es uns, die in bestimmten Speicherzonen enthaltenen Werte anzuzeigen (und zu ändern). Standardmäßig werden die Zonen angezeigt, die von den Registern HL, DE, BC, IX und IY angezeigt werden. Wenn wir jedoch von der Debug-Konsole (Zone H) aus -md *Adressbytes* schreiben (z.B. -md 0x8000 10), zeigt es uns ein neues Fenster mit dieser Speicherzone und der Anzahl der markierten Bytes.



- **G - Haltepunkte:** Ermöglicht es Ihnen, Haltepunkte hinzuzufügen. Wir können dies auch tun, indem wir auf die linke Seite einer beliebigen Codezeile doppelklicken.
- **H - Debug-Konsole**: Von hier aus können wir während des Debuggens spezifische Befehle starten. Ich empfehle dir, eine **-Hilfe zu** starten, um die gesamte Liste der verfügbaren Befehle anzuzeigen.

- **I - Debug-Menü**: Ermöglicht den Zugriff auf die wichtigsten Debuggingoptionen.

7. Was jetzt?

Nun, an dieser Stelle müssen Sie sich nur noch als ASM-Besitzer programmieren und alle Möglichkeiten entdecken, die die Kombination dieser Werkzeuge bietet.

Ich meinerseits werde die folgenden Aspekte weiter untersuchen/studieren:

- Syntax und Möglichkeiten von **sjasmplus**
- Konfiguration und zusätzliche Möglichkeiten von **ZEsarUX**
- Debugging-Funktionen, die das **Z80 Debug-Plugin** bietet.
- Z80 Unit Test in meinen Programmen verwenden

Während Sie in diese Richtung voranschreiten, werde ich Aktualisierungen des Tutorials mit allem, was ich entdecke, veröffentlichen.

7.- Danksagungen

Abschließend möchte ich mich bei der Öffentlichkeit bedanken, dank mehrerer Personen, die mir bei jedem Problem, auf das ich dabei gestoßen bin, geholfen haben und die ihren Teil dazu beigetragen haben, dieses Tutorial zu beenden.

Insbesondere möchte ich besonders erwähnen:

- **César Hernández**, dafür, dass er einen Emulator wie ZEsarUX entwickelt hat, der uns eine Welt der Möglichkeiten bietet, aber vor allem für seine Nähe, Herzlichkeit und sein Interesse, mir bei jedem Schritt zu helfen: DANKE VIELEN VIELEN VIELEN DANK!
- **Thomas Busse**, dafür, dass er uns ein so brutales Plugin zur Verfügung gestellt hat, das alle Möglichkeiten, die ZEsarUX bietet, zusammenstößt, und für seine Freundlichkeit bei der Beantwortung meiner Zweifel, Fragen und Probleme und vor allem dafür, dass er mir eine spezielle Version für Windows zur Verfügung gestellt hat, die alle Fehler korrigiert hat, die es mir nicht erlaubten, wie gewünscht zu debuggen.
- **Nestor Sancho,** für seine Sympathie und seine schnelle Hilfe, damit dein Plugin funktioniert, ohne mit dem Rest der Plugins zu kollidieren, die ich installieren musste.
- **An meinen Kollegen und Freund Alejandro Ortíz Carmona, alias Faliorn,** für den kleinen Schub, den ich brauchte, um mich wieder in Kontakt mit der Welt von Spectrum zu bringen. Ohne unsere Gespräche in den Cafés und Ihre Bestellungen für die Retro-Ausstattung zeigen Sie mir immer, wenn Sie sie im Büro erhalten, dieses Tutorial wäre wahrscheinlich nie geschrieben worden. danke Ale!
- **An meine Kollegen in den Gruppen "ZX Spectrum Assembler" und "Retrodevs".** Deine Nähe, Hilfe, Kommentare und so weiter sind ein

Schlüsselstück, um in meinem Traum von der "Dominanz des Tieres" weiter voranzukommen.....

Und damit komme ich zu diesem Tutorial, das einer einzelnen Person dient, um die Entwicklungsumgebung zu nutzen und das Beste aus ihr herauszuholen, der Aufwand, der mich gekostet hat, alles Notwendige zu lernen (und später zu schreiben), um sie zusammenzubauen, wird es wert gewesen sein.

Ich möchte, dass du mich, wenn du dieses Tutorial verwendest, per Mail oder Telegramm kontaktierst (mein Nick ist @Metaprime), um Zweifel, Probleme, Fortschritte und so weiter zu teilen.

Wir sehen uns im Forum!

César Wagener Moriana, am 14. Dezember 2019 in Sevilla.

Zufallsgenerator usr 0