

---

# TASM for VS Code

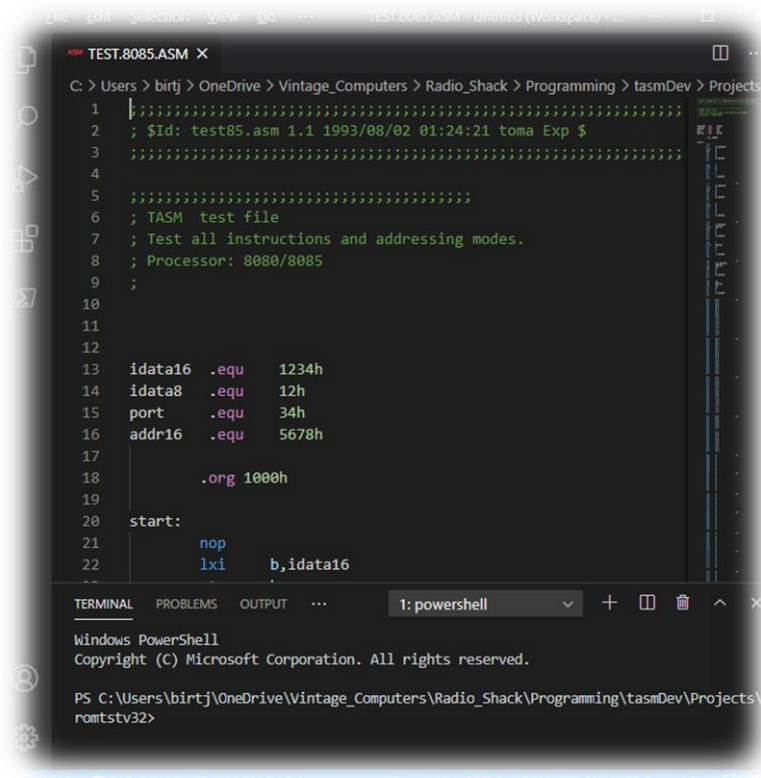
---

Visual Studio Code  
Extension for the  
Telemark Assembler,  
Soigeneris (Hey Birt!)

---

User's Manual V1.1

---



```
TEST.8085.ASM X
C:\Users\birtj\OneDrive\Vintage_Computers\Radio_Shack\Programming\tasmDev\Projects
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ; $Id: test85.asm 1.1 1993/08/02 01:24:21 toma Exp $
3  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
4
5  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
6  ; TASM test file
7  ; Test all instructions and addressing modes.
8  ; Processor: 8080/8085
9  ;
10
11
12
13  idata16 .equ 1234h
14  idata8 .equ 12h
15  port .equ 34h
16  addr16 .equ 5678h
17
18  .org 1000h
19
20  start:
21      nop
22      lxi b, idata16
```

TERMINAL PROBLEMS OUTPUT ... 1: powershell

Windows PowerShell  
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\birtj\OneDrive\Vintage\_Computers\Radio\_Shack\Programming\tasmDev\Projects> romtstv32>

## **Introduction**

The Telemark Assembler is a table-driven assembler which originated in the late 1980s and was maintained until about 2002 with the release of V3.2. A table-driven assembler can be used with many different microprocessors as the relationship between mnemonics and opcode values is given by a table of such values. You specify the correct table to use for your target microprocessor when you ask Telemark to assemble your code.

This not only has the advantage of working with many different processors it also lets you use the same familiar assembler syntax for every target processor. This syntax might look slightly different than the manufactures own assembler syntax in some cases, but the differences are small in comparison to the advantage of a uniform assembler syntax across a wide range of supported processors.

This extension provides syntax highlighting of the source code. Each supported processor as its own syntax highlighting definitions. Telemark assembler also provides a uniform assembly error report which has been integrated to the 'problem matching' capabilities in VS Code so you can click on an a reported error and be taken to the exact line in the proper source code file. The following three processors are presently supported: 6502, Motorola 6800 series, 8080, 8085 and Z80. More processor will be added as time allows. All bug reports are welcome.

## **Installation**

The TASM extension for VS Code can be installed via the marketplace just as any other extension. It can also be installed by downloading the .vsix file from the GitHub repository (link at end of this document) for the project and installing it manually. It should be noted that additional support files, including this PDF are available from the repository that do not come as a part of the installation through the Marketplace.

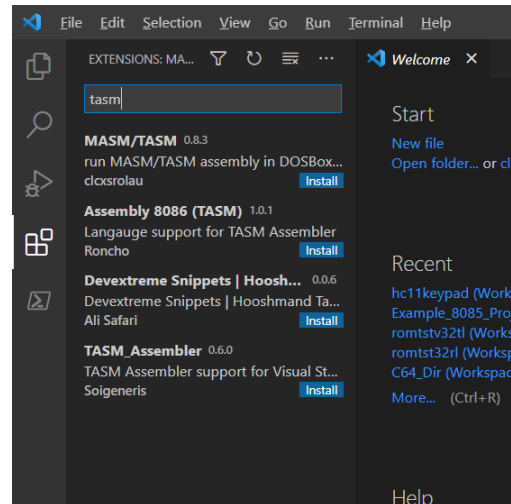
We'll first show the steps required to install the extension and then how to set things up to use it. Finally, we'll assemble an example project.

## Marketplace installation

In VS Code open the Extension menu as shown and type 'tasm' in the search box. Click the 'Install' button for 'TASM\_Assembler' by Soigeneris.

You will then be presented with the 'readme.md' for the extension in the main window to the right. It will also show up in your list of installed extension.

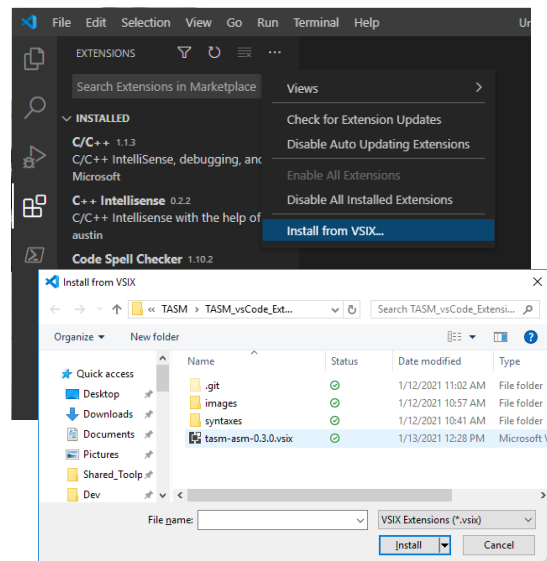
You should also now be notified when an update is available.



## Local installation

Find the 'tasm-asm-#.#.#.vsix' from the file set you downloaded from the GitHub repository. The "#.#.#" is version number which will change as the extension is updated.

In VS Code open the Extension menu as shown. Click the ellipsis ... and then click on 'Install from VSIX' and navigate to the VSIX file you downloaded.



## Suggested directory structure

The suggested directory structure is as shown to the right. The batch file that automates the assembly/build process expects this layout. This lets you keep the Telemark Assembler files (TASM) separated from your source code files (Projects).

In the TASM folder we have a subfolder for the latest version of TASM, V3.2, called "tasm32". There is also a separate subfolder called 'tasmTab' for keeping new or updated TASM assembler tables. For example, I have a modified table for the 8085 that includes some undocumented opcodes in my 'tasmTab' folder.

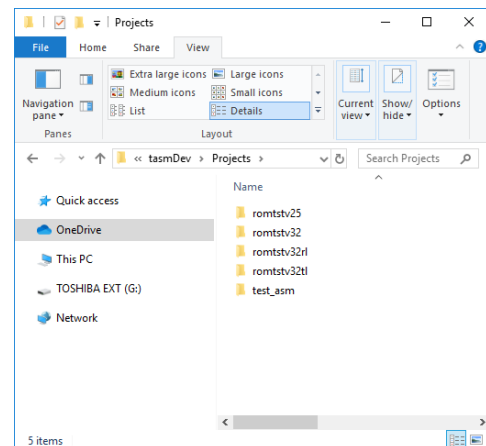
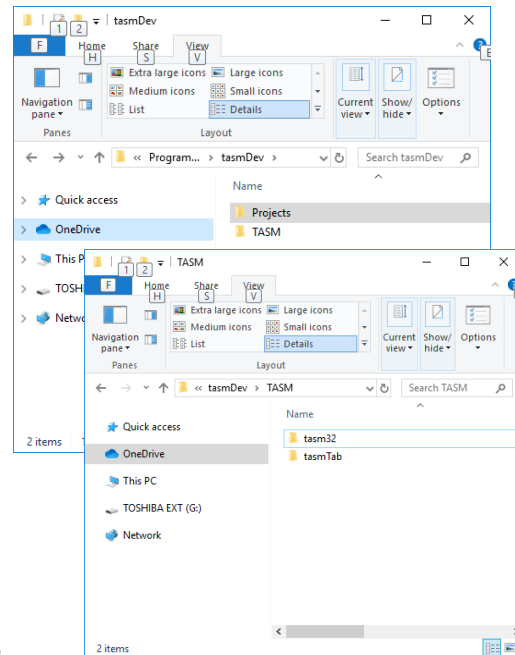
The "Projects" folder is arranged as shown, with a separate folder for each project. Here I have multiple variations of firmware for the M100 Test harness.

This also allows us to set up a separate 'Workspace' in VS Code for each separate project, this will be covered shortly.

The project repository on GitHub includes a ZIPed directory structure which also includes TASM3.2 and the modified 8085 table which includes undocumented opcodes.

## Setting up a workspace

VS Code uses the idea of a 'Workspace' to organize projects. By setting up a workspace you are telling VS Code that your project lives in a certain folder which allows VS Code to keep track of everything. VS Code will then set that folder as the Current Working



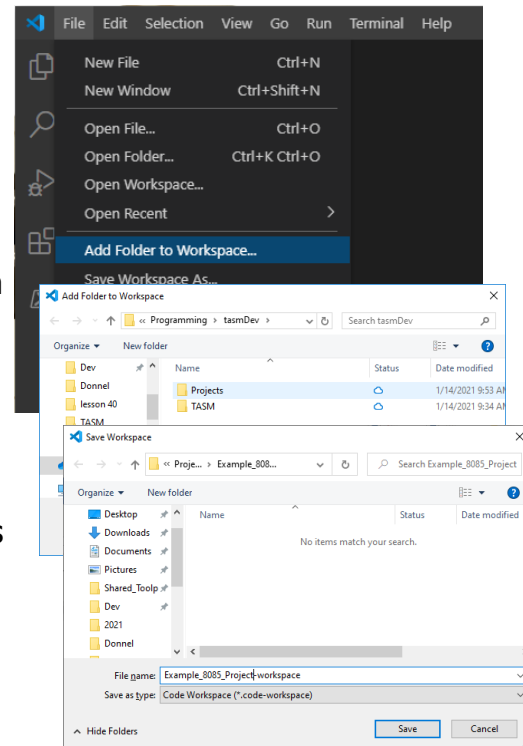
Directory (CWD). Commands which are run from the VS Code terminal, including our 'build.bat' file we'll cover in a bit, will be run from the CWD.

Let's set up a workspace for the 'Example\_8085\_Project' which is included in the GitHub repository example directory structure folder set.

In VS Code, from the File menu close any open workspace, the select 'Add folder to Workspace' and navigate to the 'Example\_8085\_Project' folder.

Next select 'File->Save Workspace As...'. I like to name my workspace with the same name as the containing folder as shown.

Your workspace for this project is now set up.

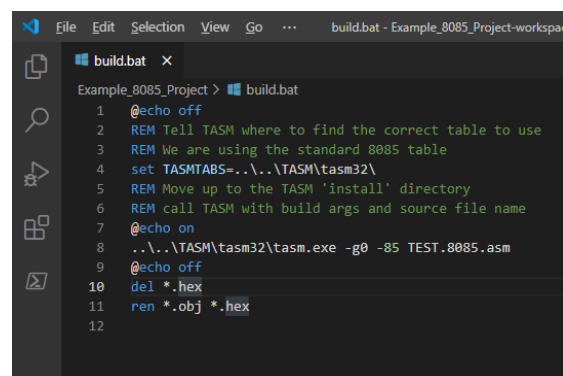


## The 'build.bat' file

The 'build.bat' file is used to automate the build/assembly process. Each project will have its own 'build.bat' which is easy to customize for that project's needs. This can be also be used with the VS Code build shortcut which we will discuss below.

Let's take a look at the 'build.bat' for our 'Example\_8085\_Project'. First, in Line #3 we are setting the environment variable 'TASMTABS' which tells the Telemark Assembler the path to the table for the processor being targeted. In this case we are pointing it to the default TASM directory.

Line #8 calls the assembler. The '..\..\\" navigates up two levels from the project directory to where we can find the tasm32 folder and thus the assembler executable.



The arguments used are as follows: `-g0` sets file output to Intel Hex format, next `-85` is the name of table to use which in case is for the 8085 processor, finally `TEST.8085.asm` is the name of the file to assemble. Line 10 deletes any existing `.hex` file in the directory and line 11 changes the file extension of the output file from `.obj` to `.hex`.

Now that we know how the build.bat works let's see how we go about assembling a source code file.

## Building/Assembling and VS Code Build Shortcut

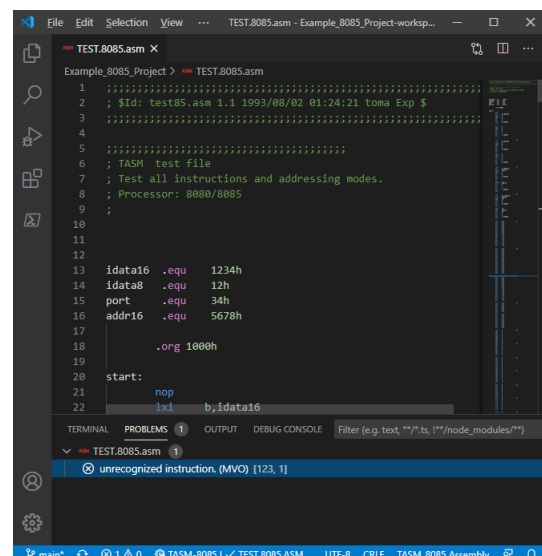
At this point we could assemble our project by typing `./build.bat` into the terminal window. Notice that the terminal window opened up with the Workspace folder as the Current Working Directory (CWD).

We can also set up a build shortcut in VS Code by modifying our `tasks.json` file. To do so select 'Terminal->Configure Tasks' and the current `tasks.json` file will be displayed. You will want to modify it as shown below, note that there is an `example.tasks.json` file in the GitHub repository file set so you can just copy and paste the changes.

After configuring the build shortcut, you can assemble your file by pressing `'Ctrl+Shift+B'`.

While I'm sure there is a method to add this functionality automatically via the extension, I was not able to figure out how to do so. Modifying the 'tasks.json' is easy enough to do though.

If you go ahead and build the project you will notice that you will get a build error which we will over in the next section on Problem Matching.



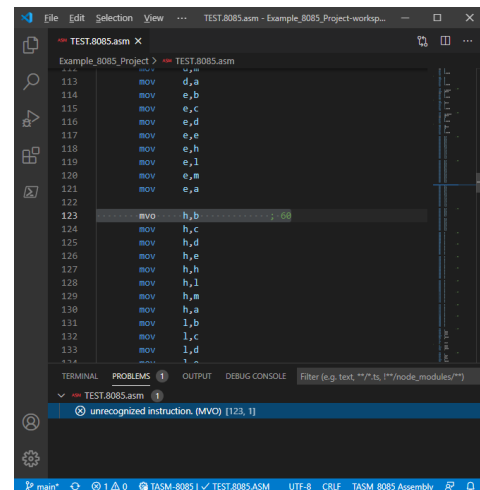
## Problem matching

The Telemark Assembler provides a uniform assembly error output. This allows us to use the 'problem matching' capabilities of VS Code which let us click on a reported assembler error and be taken to the correct line of the file that generated that error.

After building the 'test.8085.asm' file as described about you will receive an error. In the terminal window click on the 'Problems' tab and you will see an error like this "unrecognized instruction. (MVO) 123,1". This tells us that an unrecognized instruction of 'MVO' was used on line 123, column 1. Since Telemark does not provide an error column output the default value of 1 is shown.

Click on the reported error and you will be taken to the line of the file that caused the error. For this example, we are taken to line #123 of the file 'TEST.8085.asm', where we see we have a typo which we should have noticed previously due to syntax highlighting.

Change 'mvo' to 'mov', save the file and build again. Now we see that the file was assembled with no errors.



## Special output file types

If you want to use TASM to write programs to load onto a vintage computer there may be other considerations. For example, on a Commodore 64 one might load an '.prg' program off of disk. The first two bytes in this type of file give the loading address of the program in low byte, hi byte order. The rest of the file is the program.

If we want to use TASM to create a '.prg' file we first need to make sure our starting address is output first. Let's say we want to write a 'Hello World' program for our C64 that begins at the BASIC starting address of \$0801. Typically, we would put "\*"=\$0801" at the beginning of our source code to tell TASM that is the starting address of our code.

We want to make the two bytes of our '.prg' file '\$01, \$08' but how do we do that? If we just insert a ".byte \$01, \$08" as the first line of our source code these two bytes will



indeed be output first in the object file. However, something curious happens, we have not stated where in memory these two bytes are supposed to live, and not knowing any better, TASM assumes \$0000 and will fill the file with \$00s between \$0002 and \$0800. That is not what we wanted.

What we can do is use a 'adjust' our starting address like this `".org $0801 - 2"`, then specify our two bytes we want at the beginning of the file `".byte $01, $08"`. Now TASM will happily output our two-byte header and our first actual line of code will still be assembled at the correct address of \$0801. By specifying our starting address as `"$0801 - 2"` we are hinting at what we are trying to accomplish.

There is an example project called 'Example\_C64\_Project' included in the GitHub repository that will create a 'Hello World.prg' for the C64 that demonstrates this.

Another challenge for the Commodore 64 is that it uses PETSCII rather than ASCII, and if we are poking values directly to screen memory we need to use 'screen codes'. Think of a screen code as the index in the character ROM where a particular character's data can be found. This means we cannot use the assembler directive `".text"` as a convenient way to fill memory with the byte values that represent text. Instead we need to look up the numerical values, PETSCII or screen codes, and use the assembler directive `".byte"` instead.

There are platform specific development tools for example, for the C64 'CBM Prg Studio' knows all about how C64s work and can make development on the platforms it supports easier. Other vintage computer systems may have similar 'domain specific' development tool available for them.

A tool like TASM is more general purpose and can be used for embedded systems, homemade devices and vintage computers like the C64 or TRS-80 Model 100. One advantage of a general-purpose tool is that it could be used to assemble the same source code for multiple computer systems that use the same processor by making clever use of other assembler directives like 'if def' etc. For example, you might be able to use the same code base to assemble a program for the C64 and BBC computers.

These tips not only apply to the Commodore 64. For example, the TRS-80 Model 100 .CO file format uses a 6-byte header that provides the starting address, code length and entry address and we can use the same techniques to write those sorts of program files.





Your Resource for Hi-Tech Hobbies

304 Fox Creek Road  
Rolla, MO 65401 US  
573-647-9294

## Information and links

This is an open-source project created by Jeffrey T. Birt, a.k.a. 'Hey Birt!' Project files can be downloaded from the GitHub link below.

VS Code: <https://code.visualstudio.com/>

GitHub repository: [https://github.com/Jeff-Birt/TASM\\_vsCode\\_Extension](https://github.com/Jeff-Birt/TASM_vsCode_Extension)

Revision history: V1.1 clarified difference between C64 PETSCII and screen codes.  
Thanks to lagomorph for pointing this out.