



MB Maniax Next Debugger

User Manual

A ZX Spectrum Next emulator and debugger right inside Visual Studio Code. A program built with sjasmpplus runs in an accurate emulator and you debug it in its source: breakpoints, stepping, registers, memory, the emulator screen and views of the Next hardware – sprites, palettes, tilemap, Copper and more.

Extension version 0.1.0 · October 2026

MB Maniax · mb-maniax.cz

The extension is licensed under the GNU GPL 3 or later. JNEXT emulator core © Jorge González Villalonga (GPL 3).
ZX Spectrum 48K ROM © Amstrad, who allow it to be distributed with emulators.

Contents

1	Introduction	4
1.1	Features	4
1.2	Requirements	4
1.3	How to read this manual	4
2	Installation	6
2.1	Installing from a .vsix file	6
2.2	Updating to a new version	6
2.3	Reloading the window (Developer: Reload Window)	6
2.4	Checking the installation	6
2.5	Extension language	7
3	Finding your way around VS Code	8
3.1	Run and Debug	8
3.2	Debug toolbar	9
3.3	Variables – registers and machine state	9
3.4	Watch	10
3.5	Call Stack	10
3.6	Breakpoints	10
3.7	Debug Console	11
3.8	Command Palette	11
3.9	Terminal and Problems	11
4	Your first program	12
4.1	Building with debug information	12
4.2	Opening the folder	13
4.3	The launch.json file – how to debug	13
4.4	The tasks.json file – how to build	14
4.5	Starting: F5	15
4.6	Without launch.json	15
5	Debugging	17
5.1	Breakpoints	17
5.2	Stepping	17
5.3	Run to Cursor and Jump to Cursor	17
5.4	Conditional breakpoints	18
5.5	Hit counts	18
5.6	Logpoints	18
5.7	Machine event breakpoints	19
5.8	Watchpoints – stopping on memory access	20
5.9	Expressions in Watch and on hover	20
5.10	Debug Console and memory dump	20
5.11	Viewing and editing memory	21
5.12	Disassembly	21
5.13	Editing registers and flags	21
6	Expression language	22
6.1	Numbers	22
6.2	Registers and flags	22
6.3	Memory, Next registers and MMU	23
6.4	Time and raster	23
6.5	Labels	23
6.6	Operators	23
6.7	Names for events and watchpoints	24
6.8	Examples	24
7	sjasmplus keywords in comments	25
7.1	SLDOPT – enabling them in sjasmplus	25
7.2	WPMEM – watchpoint	25
7.3	LOGPOINT – debug print	25
7.4	ASSERTION – assertion	26

7.5	When keywords are evaluated	26
7.6	Expression syntax in comments	26
7.7	Turning them off	27
7.8	CSpect BREAK	27
8	Emulator screen	29
8.1	Keyboard	29
8.2	Sound	30
8.3	Pause, stops and the picture	30
9	Next panel	31
9.1	Opening and controls	31
9.2	Highlighting on the emulator screen	32
9.3	Sprites	32
9.4	Pattern RAM	34
9.5	Palettes	35
9.6	Layers	36
9.7	Layer 2	37
9.8	Tilemap	38
9.9	Copper	39
9.10	NEXTREG	40
10	Saved states	41
10.1	Saving and restoring	41
10.2	States are saved at the end of a frame	42
10.3	Export to and import from a file	42
11	Profiler	43
11.1	Starting	43
11.2	Reading the results	43
11.3	Most expensive functions	44
11.4	Live updates, stopping, clearing	44
11.5	Tips	45
12	Step Back	46
12.1	Enabling it	46
12.2	Using it	46
13	Settings and launch.json	47
13.1	launch.json configuration attributes	47
13.2	Extension settings	48
13.3	Command overview	48
14	Troubleshooting	49
15	Keyboard shortcuts	51

1 Introduction

MB Maniax Next Debugger is an extension for Visual Studio Code (VS Code from here on) that adds a ZX Spectrum Next emulator and a debugger to the editor. A program built with **sjasmpplus** runs in an emulator based on the **JNEXT** core and you debug it right in its source code: stop it on a line, inspect registers and memory, step instruction by instruction – with the emulator screen next to your code.

If you are used to CSpect or DeZog, much of this will feel familiar: the CSpect **BREAK** instruction works, and the special comments **WPMEM**, **LOGPOINT** and **ASSERTION** use the same syntax as DeZog. The difference is that there is nothing else to install or start – the emulator is part of the extension.

1.1 Features

- **Source breakpoints** based on the sjasmpplus debug information (the SLD file), including the Next's 8K pages – a breakpoint in paged code stops only when its page is mapped in.
- **Conditional breakpoints with no slowdown** – the condition (`A == 5`, `mem[counter] == 3`, `nextreg[$50] == 10`, `FRAME > 100` ...) is evaluated by the emulator itself. Labels from the source can be used directly.
- **Hit counts** (`3`, `>= 10`, `% 4`) and **logpoints**, which print a message with values instead of stopping.
- **Machine event breakpoints**: IM 1 / IM 2 interrupts, NMI, NEXTREG writes, port input/output, Copper, DMA, reset.
- **Watchpoints** – stop on a memory read or write.
- **Stepping** (Step Into / Over / Out), call stack with labels, registers and flags (editable), MMU pages, disassembly, memory viewer and editor, expressions in the Watch view and on hover.
- **Step Back** – the emulator remembers the last 10 seconds.
- **Emulator screen** in a VS Code panel, with sound and keyboard.
- **Next panel** with views of the hardware: sprites, Pattern RAM, palettes, layers, Layer 2, tilemap, the Copper program and all NEXTREG registers.
- **Saved states** of the whole machine, and export to a file.
- **Profiler** showing how many T-states the program spent on every source line.

1.2 Requirements

- **Windows (x64)** – the extension package `mbx-next-debugger-win32-x64-0.1.0.vsix` contains the debug server for 64-bit Windows.
- **Visual Studio Code 1.100 or newer.**
- **sjasmpplus** – the assembler you build the program with. It should be on the `PATH`, so that the command `sjasmpplus` works in a terminal.
- No SD card and no Next ROMs are needed: the extension has its own ZX Spectrum 48K ROM and starts the program directly. An SD card with ROMs is only needed to emulate machines other than the Next (see [13](#)).

1.3 How to read this manual

The manual does not assume you know VS Code. Whenever a part of the window is mentioned, it says where to find it and which keyboard shortcut gets you there. Conventions:

- `Ctrl` + `Shift` + `P` – a keyboard shortcut (press the keys together).
- **Run and Debug** – the name of a part of the VS Code window, a button or a tab, exactly as it appears on screen.
- **MB Maniax: Save State...** – a command run from the Command Palette (3.8).
- `mem[counter]` – text you type (an expression, code, a setting).

NOTE

Language. The manual assumes VS Code itself is in English (the default). Texts of the extension – the emulator screen, the Next panel, messages, the groups in the Variables view – follow the language of the extension (2.5).

Command names and the descriptions in Settings follow the VS Code display language instead.

2 Installation

The extension comes as a single file with the `.vsix` extension (for example `mbx-next-debugger-win32-x64-0.1.0.vsix`). It is a package VS Code installs by itself – do not unpack it.

2.1 Installing from a .vsix file

1. Start VS Code.
2. Open the Extensions view: in the narrow vertical strip of icons on the far left (the **Activity Bar**) click the icon with four squares, or press `Ctrl + Shift + X`. The **Extensions** list opens on the left.
3. In the header of the list (top right, above the search box) click the three dots ... (**Views and More Actions**) and choose **Install from VSIX...**
4. Select the `.vsix` file in the dialog and confirm with **Install**. A notification in the bottom right corner reports when installation is complete.

The same is available from the Command Palette: `Ctrl + Shift + P`, type `vsix` and choose **Extensions: Install from VSIX...** From the command line:

```
code --install-extension mbx-next-debugger-win32-x64-0.1.0.vsix
```

2.2 Updating to a new version

A new version is installed exactly the same way – install the new `.vsix`; there is no need to uninstall the old one first. If a debug session is running, stop it first (`Shift + F5`). After installation VS Code usually offers a button to restart extensions; if it does not, or you are not sure the new version is running, reload the window (next section).

2.3 Reloading the window (Developer: Reload Window)

After installing or updating the extension, the safest step is to reload the window – VS Code reloads all extensions, while open files and the window layout stay as they are:

1. Press `Ctrl + Shift + P` (or `F1`). The Command Palette input box opens at the top centre of the window.
2. Type `reload`.
3. Choose **Developer: Reload Window** and press `Enter`.

The window disappears for a moment and is drawn again. Closing and restarting VS Code works too.

2.4 Checking the installation

In the **Extensions** list (`Ctrl + Shift + X`) type `MB Maniax` into the search box. **MB Maniax Next Debugger** with its version number should be listed among the installed extensions (**Installed**). Click it to open its description page.

2.5 Extension language

The extension is available in English, Czech, Slovak, Spanish, German, French, Italian, Polish, Portuguese (Brazil), Russian, Ukrainian, Japanese and Simplified Chinese. The language is chosen by the `mbxNext.language` setting:

- `auto` (default, shown as **Automatic**) – the VS Code display language if it is not English; otherwise the Windows language; otherwise English. On, say, Czech Windows with English VS Code the extension therefore speaks Czech.
- or a fixed language: `en`, `cs`, `sk`, `es`, `de`, `fr`, `it`, `pl`, `pt-br`, `ru`, `uk`, `ja`, `zh-cn`.

To change it:

1. Open Settings: `Ctrl` + `,` (comma), or the menu **File** > **Preferences** > **Settings**.
2. Type `mbxNext` into the search box at the top.
3. Choose the language in the **Mbx Next: Language** drop-down.

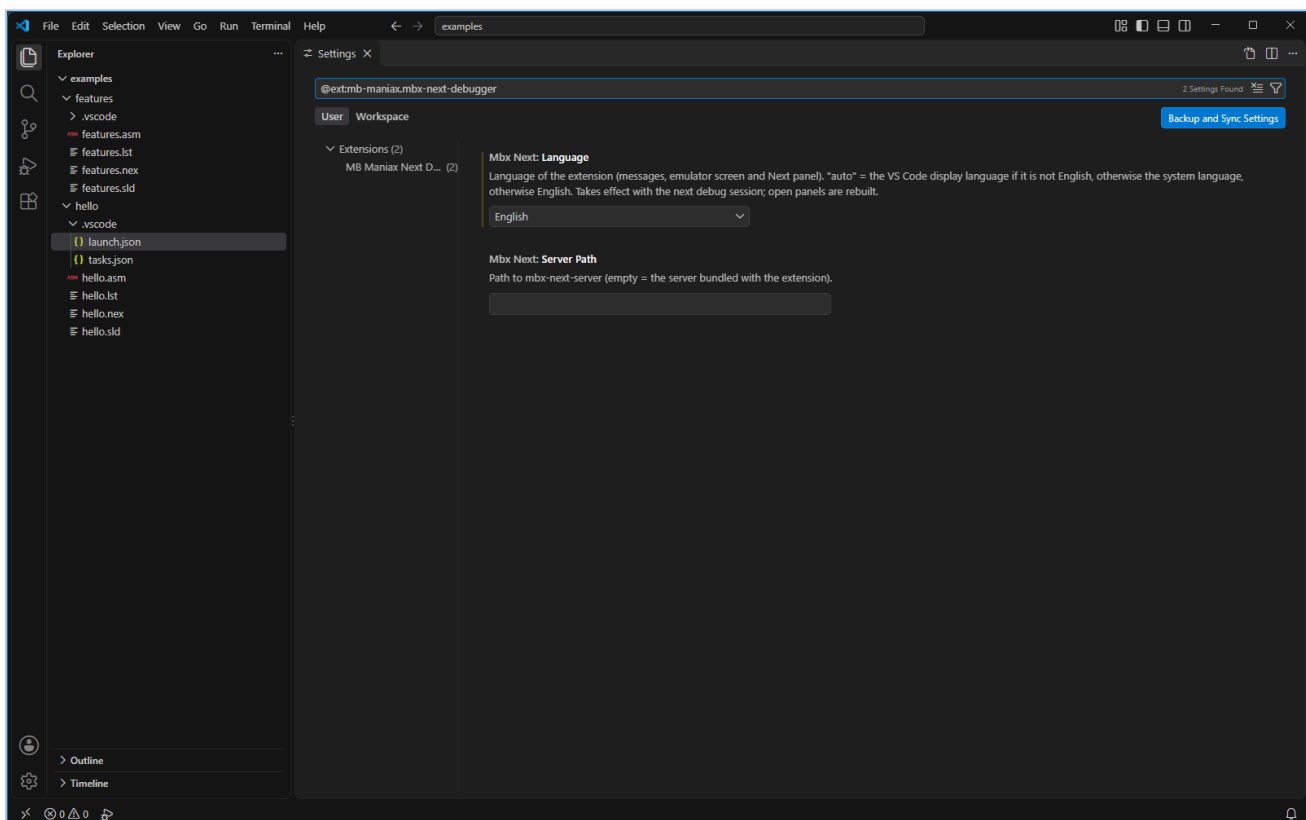


Figure 1 VS Code Settings filtered to `mbxNext`: the extension language and the path to the debug server.

A change takes effect with the next debug session; panels that are open (emulator screen, Next panel) are rebuilt in the new language right away. Command names and the descriptions in Settings and `launch.json` follow the VS Code display language, not this setting.

3 Finding your way around VS Code

This chapter is for those who are new to VS Code. It describes the parts of the window you will meet while debugging and where to find them. Figure 2 shows the window while debugging the example program.

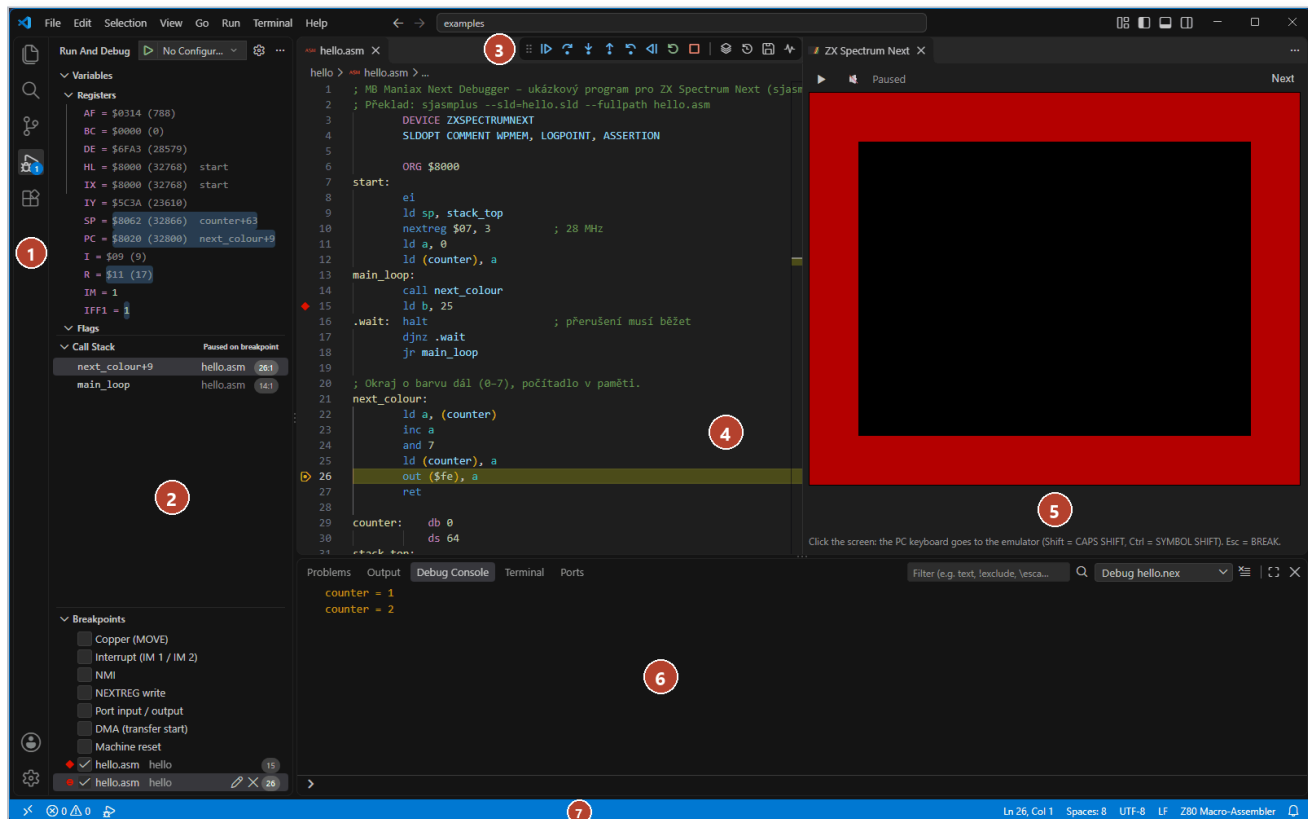


Figure 2 The VS Code window while debugging: the program is stopped at a breakpoint (yellow line), the debug views are on the left, the debug toolbar at the top, the emulator screen on the right.

- 1 **Activity Bar** – the narrow vertical strip of icons on the far left. It switches what the side bar shows: files (**Explorer**, **Ctrl** + **Shift** + **E**), search, extensions and, above all, **Run and Debug** (the “play” triangle with a bug, **Ctrl** + **Shift** + **D**).
- 2 **Run and Debug side bar** – to the right of the Activity Bar. At the top is the green ► button with the name of the debug configuration (starts debugging), below it the **Variables**, **Watch**, **Call Stack** and **Breakpoints** sections. Click a section header to expand or collapse it.
- 3 **Debug toolbar** – a floating row of buttons at the top centre of the window; it appears only while debugging.
- 4 **Editor** – in the middle. Left of the line numbers is a narrow margin (the gutter) where you click to set breakpoints. The line the program is stopped on is highlighted in yellow with a yellow arrow in the margin.
- 5 **ZX Spectrum Next panel** – the emulator screen; it opens to the right of the editor when debugging starts (8).
- 6 **Bottom panel (Panel)** – tabs **Problems**, **Output**, **Debug Console**, **Terminal**. Show and hide it with **Ctrl** + **J**.
- 7 **Status Bar** – at the very bottom. It changes colour while debugging (usually to orange). On its right side is, among other things, the language of the open file (for example **Plain Text**) – chapter 14 comes back to it.

3.1 Run and Debug

Start debugging with **F5**, or with the green ► button at the top of the **Run and Debug** side bar (**Ctrl** + **Shift** + **D**). Next to the button is a drop-down list of debug configurations from `.vscode/launch.json` – if you have more than one, this is where you pick which one to start. If the folder has

no launch.json, the side bar shows a **create a launch.json file** link (4.3).

3.2 Debug toolbar

While debugging, a floating toolbar appears at the top centre of the window. You can drag it by its left edge. The buttons from the left:

Button	Key	What it does
Continue / Pause	F5 / F6	Runs a stopped program / stops a running one.
Step Over	F10	Executes one instruction; steps over CALL, RST and DJNZ (the subroutine or loop runs to completion).
Step Into	F11	Executes exactly one instruction (enters subroutines).
Step Out	Shift + F11	Runs until the current subroutine returns.
Step Back, Reverse	–	Step Back (12); Reverse Continue is not available yet.
Restart	Ctrl + Shift + F5	Loads the program again and starts it from the beginning.
Stop	Shift + F5	Ends debugging and the emulator.

They are followed by the buttons of this extension (icons; hover to see the name): **Next Panel (Sprites, Palettes, ...)** – opens the Next panel, **Save State...** and **Restore State...** – saved states, **Start Profiler / Stop Profiler** – the profiler. If not all buttons fit, VS Code hides some of them behind an arrow or three dots at the end of the toolbar.

3.3 Variables – registers and machine state

The **Variables** section of the Run and Debug side bar shows the machine state whenever the program is stopped. It has four groups:

- **Registers** – AF BC DE HL IX IY SP PC in hex and decimal (\$8000 (32768)), followed by the label the value points to (start, main_loop+3); then I, R, the interrupt mode IM and IFF1.
- **Flags** – S Z H P/V N C as 0 or 1.
- **Shadow registers** – AF' BC' DE' HL' (collapsed by default).
- **Machine (MMU, time)** – **Frame** (frame number since start), **Total T-states** and the eight MMU slots: MMU0 (\$0000) ... MMU7 (\$E000) with the number of the 8K page mapped into the slot (page 10 (\$0A)).

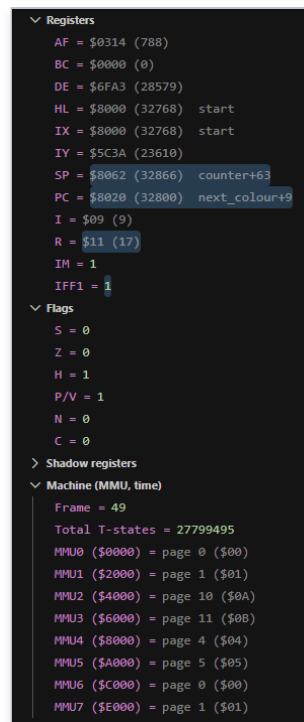


Figure 3 The Variables section: registers with labels, flags and the Machine (MMU, time) group.

Register and flag values can be changed ([5.13](#)).

3.4 Watch

The **Watch** section holds expressions you want to keep an eye on – for example `mem[counter]` or `HL`. Add them with the **+** button in the section header (it appears when you hover over the header), or select text in the editor, right-click and choose **Add to Watch**. The values are recalculated every time the program stops. Details in [5.9](#).

3.5 Call Stack

The **Call Stack** section shows where the program is stopped and how it got there: the current location at the top (the name of the nearest label), below it the places subroutines were called from. Click a line to move the editor to that place in the source. Locations outside your source (ROM, code without debug information) are greyed out and show their address. The section header also shows the **reason for stopping** – for example **Paused on breakpoint**, **Paused on step**, **Paused on BREAK** or the name of an event.

3.6 Breakpoints

The **Breakpoints** section lists all breakpoints and logpoints; the check box disables one temporarily, a double-click takes you to its line. At the top of the list are the **machine event** check boxes of this extension (**Interrupt (IM 1 / IM 2)**, **NMI**, **NEXTREG write ...**), see [5.7](#). The section header has buttons to deactivate and remove all breakpoints.

3.7 Debug Console

Debug Console is a tab of the bottom panel. Open it with **Ctrl** + **Shift** + **Y**, the menu **View > Debug Console**, or by clicking the **Debug Console** tab in the bottom panel (**Ctrl** + **J** shows the panel). Logpoint output, messages of the extension and emulator output appear here. At the very bottom is an input line with a **>** prompt: type an expression (for example **HL**) and press **Enter** – its value is printed. The command **m address** dumps memory (5.10). The **↑** **↓** arrows browse previously entered commands.

3.8 Command Palette

The **Command Palette** is where you can run any command of VS Code and its extensions. Open it with **Ctrl** + **Shift** + **P** or **F1**, or from the menu **View > Command Palette....** An input box appears at the top centre of the window; type part of a command name (for example **mb maniax**) and pick it with the arrow keys. Commands of this extension start with **MB Maniax**: (the emulator screen with **MB Maniax Next**). Some of them are listed only while a debug session is running.

3.9 Terminal and Problems

The **Terminal** (**Ctrl** + **`**, the key below **Esc**) is a command line in the bottom panel – this is where **sjsmplus** prints its output while building. Build errors also appear in the **Problems** tab (**Ctrl** + **Shift** + **M**); click an error to jump to its source line.

4 Your first program

In this chapter you build and run the example program **hello** from the `examples/hello` folder. The program changes the border colour and counts in memory how many times it did so. The procedure is the same for your own project.

```
; examples/hello/hello.asm (shortened)
DEVICE ZXSPCTRUMNEXT
SLDOPT COMMENT WPMEM, LOGPOINT, ASSERTION
ORG $8000
start: ei
      ld sp, stack_top
      nextreg $07, 3      ; 28 MHz
      ld a, 0
      ld (counter), a
main_loop:
      call next_colour
      ld b, 25
.wait: halt
      djnz .wait
      jr main_loop
next_colour:
      ld a, (counter)
      inc a
      and 7
      ld (counter), a
      out ($fe), a
      ret
counter:  db 0
          ds 64
stack_top:
      SAVENEX OPEN "hello.nex", start, stack_top
      SAVENEX CORE 3, 0, 0
      SAVENEX AUTO
      SAVENEX CLOSE
```

4.1 Building with debug information

For the debugger to know which source line corresponds to which address, it needs an **SLD** file (Source Level Debugging) from `sjasmpplus`. `sjasmpplus` writes it with the `--sld` option:

```
sjasmpplus --sld=hello.sld --fullpath hello.asm
```

- `--sld=hello.sld` – writes the debug information (lines, addresses, 8K pages, labels, keywords from comments).
- `--fullpath` – writes full source paths into the SLD, so the extension reliably matches lines with the files open in the editor (even when the source is split into several files with `INCLUDE`).

The program itself (`hello.nex`) is written by the `SAVENEX` directive in the source.

TIP

Check that `sjasmpplus` can be started: open a terminal (`Ctrl` + ```) and type `sjasmpplus --version`. If Windows does not know the command, add the `sjasmpplus` folder to the `PATH` variable (then close and restart VS Code), or put the full path to `sjasmpplus.exe` into `tasks.json`.

4.2 Opening the folder

VS Code works with **folders** (workspaces), not single files – debug and task settings are looked up in the `.vscode` subfolder of the open folder.

1. Menu **File** > **Open Folder...** (`Ctrl` + `K` `Ctrl` + `O`) and select the `examples\hello` folder.
2. If VS Code asks whether you trust the authors of the folder (**Do you trust the authors...**), confirm with **Yes, I trust the authors** – otherwise it would not be allowed to run tasks (the build).
3. In the **Explorer** on the left (`Ctrl` + `Shift` + `E`) click `hello.asm`; it opens in the editor.

4.3 The launch.json file – how to debug

The `.vscode/launch.json` file tells VS Code what to start on `F5`. The example already has one:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "mbx-next",
      "request": "launch",
      "name": "Ladit hello.nex",
      "program": "${workspaceFolder}/hello.nex",
      "sld": "${workspaceFolder}/hello.sld",
      "stopOnEntry": true,
      "preLaunchTask": "sjasmpus"
    }
  ]
}
```

- `"type": "mbx-next"` – debug with this extension.
- `"name"` – the name shown in the list next to the green ► button (in the example it is Czech, “Debug hello.nex”; use any name you like).
- `"program"` and `"sld"` – the built program and its debug information; `${workspaceFolder}` is the open folder.
- `"stopOnEntry": true` – stop at the program's first instruction.
- `"preLaunchTask": "sjasmpus"` – before starting, build the program with the `sjasmpus` task from `tasks.json` (next section).

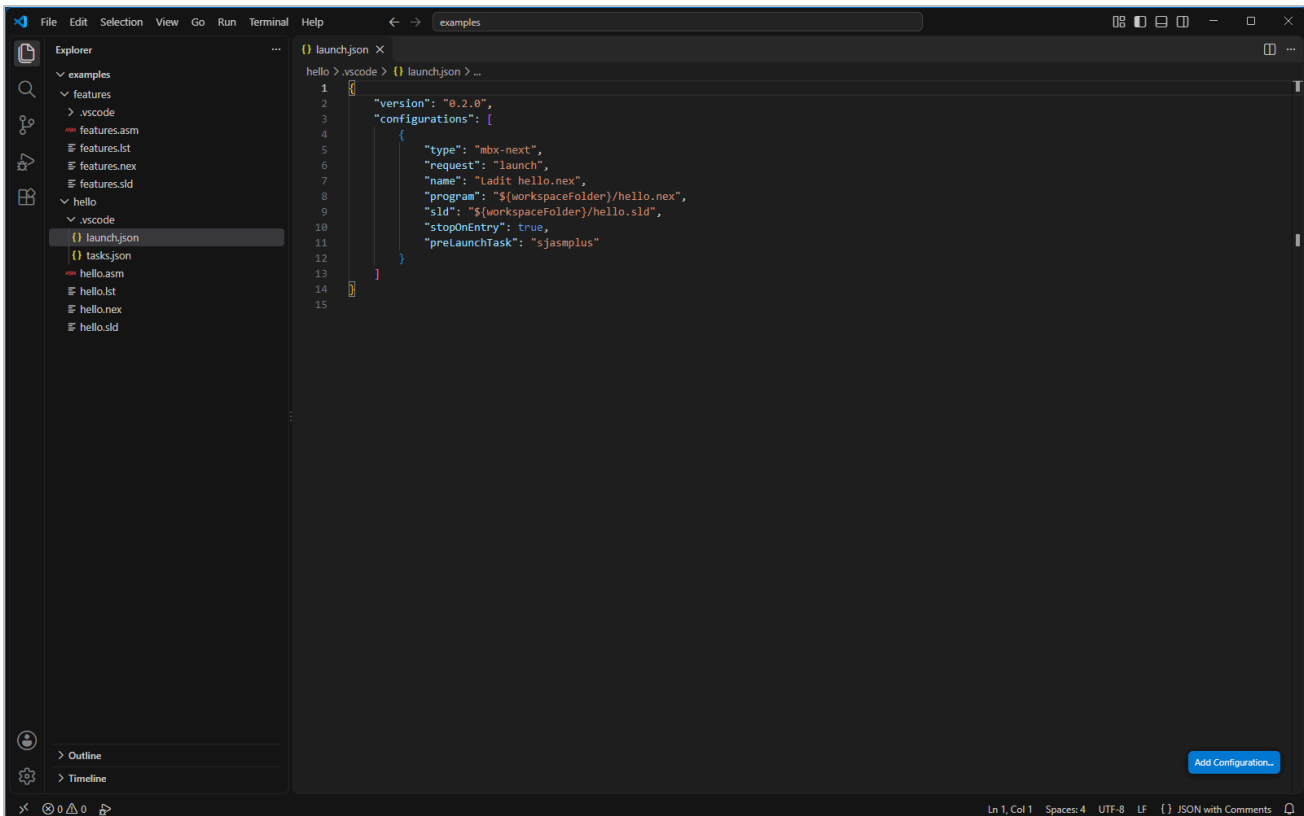


Figure 4 The .vscode/launch.json file with an mbx-next configuration in the editor.

For your own project create launch.json like this: in the **Run and Debug** side bar click **create a launch.json file** and choose **MB Maniax Next Debugger**. You get a configuration “Debug on ZX Spectrum Next” with `program.nex` and `program.sld` – replace the names with yours. To add another configuration to an existing launch.json, click **Add Configuration...** at the bottom right of the launch.json editor and choose **MB Maniax Next Debugger: Next program (sjasmplius)**. All attributes are described in 13.

4.4 The tasks.json file – how to build

A task is a command VS Code runs for you – here the sjasmplius build. The example's `.vscode/tasks.json`:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "sjasmpplus",
      "type": "shell",
      "command": "sjasmpplus",
      "args": ["--sld=hello.sld", "--fullpath", "--lst=hello.lst", "hello.asm"],
      "options": { "cwd": "${workspaceFolder}" },
      "group": { "kind": "build", "isDefault": true },
      "problemMatcher": {
        "owner": "sjasmpplus",
        "fileLocation": ["autoDetect", "${workspaceFolder}"],
        "pattern": {
          "regexp": "^(.*)\\((\\d+)\\):\\s+(error|warning)[^:]*:\\s+(.*)$",
          "file": 1, "line": 2, "severity": 3, "message": 4
        }
      }
    }
  ]
}
```

- "label" – the task name; `preLaunchTask` in `launch.json` refers to it.
- "args" – sjasmpplus options (`--lst` also writes a listing).
- "group": ... "isDefault": true – the default build task: run it any time with **Ctrl** + **Shift** + **B**, even without debugging.
- "problemMatcher" – sjasmpplus errors and warnings appear in the **Problems** tab and as squiggles in the source.

4.5 Starting: F5

1. Press **F5**. The **Terminal** opens in the bottom panel and the build runs. (If the build fails, VS Code offers **Debug Anyway** or **Show Errors** – look into **Problems**.)
2. The emulator starts and the **ZX Spectrum Next** panel with the screen opens to the right of the editor.
3. Because the example has `"stopOnEntry": true`, the program stops right at its first instruction (`ei` after the `start` label) – the line is yellow and the Variables and Call Stack sections on the left fill in.
4. Click in the margin left of the line number of `call next_colour` – a red dot appears: a breakpoint.
5. Press **F5** (Continue). The program runs and stops at the breakpoint.
6. Press **F11** (Step Into) – you are inside `next_colour`. Keep stepping with **F11** and watch `A` change in the Variables section; after `out ($fe)`, `a` the border colour on the emulator screen changes.
7. Hover over the word `counter` in the source – a tooltip shows its address and the memory content. In the Debug Console (**Ctrl** + **Shift** + **Y**) type `m counter` and press **Enter**.
8. End debugging with **Shift** + **F5** or the **Stop** button on the debug toolbar.

4.6 Without launch.json

For a quick try, `launch.json` is not required: open an `.asm` file in the editor (also `.a80`, `.z80` or `.s`) and press **F5**. The extension starts the `.nex` and `.sld` of the same name from the same folder (for `game.asm` that is `game.nex` and `game.sld`). In this mode the program is **not built** – build it first (**Ctrl** + **Shift** + **B** or in the

terminal). If VS Code asks which debugger to use (**Select debugger**), choose **MB Maniax Next Debugger**.

NOTE

Program data and esxDOS. Without an SD card image, the program sees its own folder through esxDOS (loading data, saved games). A different folder or an SD card image is set in launch.json with the `esxdosRoot` and `sdImage` attributes.

5 Debugging

5.1 Breakpoints

A breakpoint stops the program just **before** the instruction on its line is executed.

- **With the mouse:** move the pointer into the narrow margin left of the line numbers – a faint red dot appears. Click to set it, click again to remove the breakpoint.
- **With the keyboard:** put the cursor on the line and press **F9** (toggle).
- **From the context menu:** right-clicking the margin offers **Add Breakpoint**, **Add Conditional Breakpoint...** and **Add Logpoint...**; right-clicking an existing dot offers **Edit Breakpoint...**, **Disable Breakpoint** and **Remove Breakpoint**.

You can set breakpoints before debugging starts; they are sent to the emulator at startup. If there is no instruction on the line (a comment, an empty line, a label on its own), the breakpoint moves to the nearest following line with an instruction (at most 20 lines down). If there is none there either, the dot stays **grey and hollow** (unverified breakpoint) and its tooltip says why – for example *No instruction on this line or below it (according to SLD)* or *SLD from sjasmpplus is missing ("sld" attribute)*.

NOTE

Paged memory. The sjasmpplus SLD knows the 8K page of every line (when the source uses `DEVICE ZXPECTRUMNEXT`). The breakpoint then stops only when that page is actually mapped into the corresponding MMU slot – code of another page at the same address does not trigger it. When the same line is assembled into several pages, the breakpoint applies to all of them.

TIP

If you cannot set a dot in the margin at all (nothing appears on hover), VS Code does not consider the file a language the extension reports breakpoints for. See chapter 14.

5.2 Stepping

While the program is stopped, control it with the debug toolbar buttons or keys:

Key	Action	Description
F5	Continue	Runs until the next breakpoint, event or pause.
F6	Pause	Stops the running program at the next instruction boundary.
F11	Step Into	One instruction. On <code>HALT</code> it waits until an interrupt ends it.
F10	Step Over	Like Step Into, but <code>CALL</code> , <code>RST</code> and <code>DJNZ</code> run to completion and the machine stops at the following instruction.
Shift + F11	Step Out	Runs until the program returns from the current subroutine (one level up the stack).

You can also pause and resume the program with the **|| / ▶** button in the emulator screen panel.

5.3 Run to Cursor and Jump to Cursor

While the machine is stopped, right-click a source line:

- **Run to Cursor** – the program runs until it gets to this line (like a one-shot breakpoint).

- **Jump to Cursor** – moves PC to this line **without running the code in between**. Useful for skipping a piece of code or repeating an instruction. If the line is assembled at several addresses (several pages), VS Code offers a choice – the items look like `$C000 (page 12)`, and pages that are not mapped at the moment are marked *not mapped*.

5.4 Conditional breakpoints

A conditional breakpoint stops only when its condition holds. The condition is evaluated by the emulator itself on every pass, so it **does not slow the program down** – feel free to put one into a loop that runs thousands of times per frame.

1. Right-click the margin left of the line and choose **Add Conditional Breakpoint...** (for an existing breakpoint **Edit Breakpoint...**).
2. A small input box opens below the line. It has a drop-down on the left: leave it at **Expression**.
3. Type the condition and press **Enter**. The breakpoint dot gets an **=** sign.

Example conditions:

```
A == 5
HL > $4000 and not ZF
mem[counter] == 3
mem16[score] >= 1000
nextreg[$50] == 10
FRAME > 100
B == 0 or (IX >= enemies and IX < enemies_end)
```

Labels from the source (`counter`, `score`) are written directly. The full expression language is described in chapter 6. An error in the condition (a typo, an unknown name) is caught by the emulator right away: the breakpoint stays grey and its tooltip shows the error. A run-time error (e.g. division by zero) is printed to the Debug Console.

5.5 Hit counts

In the breakpoint edit box choose **Hit Count** in the drop-down and enter which pass should stop:

Entry	Stops
3 or == 3	exactly on the 3rd pass
>= 10	on the 10th pass and every one after it
> 10, < 5, <= 5	likewise
% 4	on every 4th pass (4th, 8th, 12th ...)

A condition and a hit count can be combined – passes are then counted only when the condition holds. The counter starts from zero whenever breakpoints change.

5.6 Logpoints

A logpoint **does not stop** the program; it only prints a message to the Debug Console – a debug print you do not have to write into the program. Right-click the margin and choose **Add Logpoint...** (or **Log Message** in the breakpoint edit box) and type the message. Expressions in curly braces are replaced by their values:

```
colour {A}, score {mem16[score]}
HL={HL:hex} DE={DE:hex} flags {F:flags}
```

A format may follow a colon: `hex`, `unsigned` (default), `signed`, `char` (ASCII character), `bits` (binary), `flags` (flags as `SZHPNC`, cleared ones as `-`), `labels` (the label at the address). A logpoint has a diamond shape in the margin. The message is printed on every pass; with a hit count, only when the count matches.

5.7 Machine event breakpoints

Some bugs are not tied to a particular line: “who keeps overwriting register \$50?”, “when did the interrupt come?”. For these the extension adds event check boxes to the **Breakpoints** section. Ticking a box enables the event; when you hover over the item, a pencil (**Edit Condition**) appears on the right for entering a condition – without a condition every event of that kind stops.

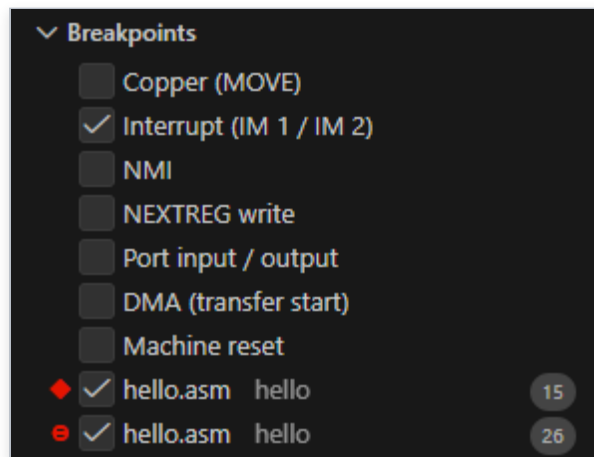


Figure 5 The Breakpoints section: machine event check boxes, a breakpoint with the condition `A == 3` and a logpoint.

Event	Stops	Extra names for the condition
Interrupt (IM 1 / IM 2)	at the start of a maskable interrupt handler	– (e.g. <code>IM == 2</code> , <code>FRAME > 50</code>)
NMI	non-maskable interrupt (Multiface, DivMMC)	–
NEXTREG write	on a write to a Next register (with the NEXTREG instruction, through ports \$243B/\$253B, or by the Copper)	REG register number, VALUE value written, PREV previous value, SOURCE who writes (CPU, COPPER, DMA)
Port input / output	on an IN or OUT instruction	PORT port number, VALUE value, SOURCE
Copper (MOVE)	when the Copper executes a MOVE	CPC address in the Copper program (0–1023), CVC line, HC_UA horizontal position
DMA (transfer start)	at the start of a DMA transfer	SRC source address, DST destination address
Machine reset	on reset	(no condition)

All event conditions can also use registers, memory and the other names from chapter 6; here `PC` is the address of the instruction that caused the event. The machine stops right after that instruction. Examples: `REG == $50` or `REG == $51`, `(PORT & $FF) == $FE`, `REG == $15` and `SOURCE == CPU`. After stopping on an event, its name is shown in the Call Stack header.

NOTE

PORT is the full 16-bit port address as seen on the bus – for out (\$fe), a the high byte is the value of A (port \$01FE, \$02FE ...). The condition `PORT == $FE` would therefore stop only when `A = 0`. For ports decoded by the low byte only (ULA \$FE, Kempston \$1F ...) write `(PORT & $FF) == $FE`; compare the full number for 16-bit ports (`PORT == $243B`).

5.8 Watchpoints – stopping on memory access

A watchpoint (a “data breakpoint” in VS Code) stops the program when an instruction reads or writes the chosen memory. The machine stops right after the instruction that made the access.

- **On a register:** in the **Variables** section right-click BC, DE, HL, IX, IY or SP and choose **Break on Value Change** (write), **Break on Value Read** (read) or **Break on Value Access** (both). It watches the byte **the register currently points to** – not the register itself.
- **On a label or address:** add an expression to the **Watch** section that is just a label name (`counter`) or a number (`$5B00`), right-click it and choose **Break on Value Change**.
- **With a comment in the source:** ; WPMEM (7.2) – handy for areas longer than a byte and for watchpoints you want to keep.

Watchpoints are listed in the **Breakpoints** section, where you also disable or remove them. Newer VS Code versions also offer **Add Data Breakpoint at Address** in the Breakpoints section header, where you enter an address and length directly.

5.9 Expressions in Watch and on hover

You can type any expression from chapter 6 into the **Watch** section: `mem[counter]`, `HL`, `nextreg[$07]`, `mem16[stack_top - 2]`, `mmu[3]`. The result is shown in hex and decimal (`$03 (3)`). If the expression is just a label name, its address is shown together with the memory content there: `$8023 (32803) → [$03] [$0003]` – the byte and the 16-bit word in square brackets.

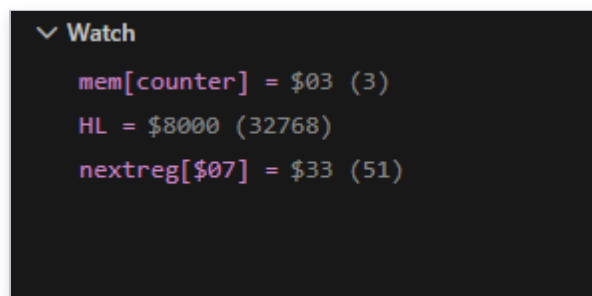


Figure 6 The Watch section with the expressions `mem[counter]`, `HL` and `nextreg[$07]`.

While the program is stopped, just **hover** over a register or label in the editor – a tooltip shows the value and the memory it points to. The tooltip appears only for words that are a valid expression.

5.10 Debug Console and memory dump

Type an expression into the Debug Console input line (`Ctrl` + `Shift` + `Y`, the line with `>` at the very bottom) and press `Enter`; its value is printed just like in Watch. In addition, there is a memory dump command:

```
m address
m address, length
```

Both the address and the length are expressions (a number, label, register...). Without a length, 128 bytes are dumped; the maximum is 4096. Each line has the address, 16 bytes in hex, the same bytes as text (non-printable ones as `.`) and the label that starts at or precedes the address:

```
> m start, 32
$8000 FB 31 64 80 ED 91 07 03 3E 00 32 23 80 CD 17 80 .1d.....>.2#.... start
$8010 06 19 76 10 FD 18 F6 3A 23 80 3C E6 07 32 23 80 ..v.....:#. <..2#. main_loop+3
> m HL, 16
> m stack_top - 16, 16
```

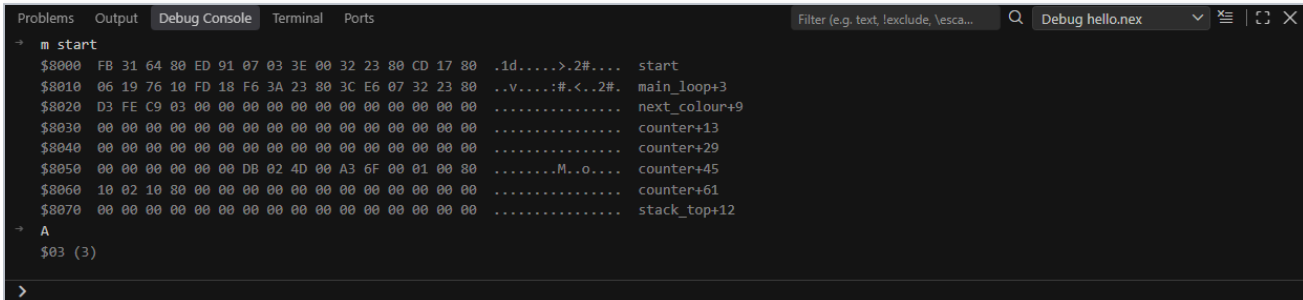


Figure 7 Debug Console: a memory dump with `m start` and the value of the expression `A`.

5.11 Viewing and editing memory

When you hover over a 16-bit register in the **Variables** section, a small **View Binary Data** icon appears on the right. It opens memory from the address in the register in the VS Code hex editor (the Microsoft *Hex Editor* extension – if you do not have it, VS Code offers to install it). Bytes can be overwritten in the hex editor; the write goes straight into the emulator's memory. The memory is shown as the CPU sees it (64 KB according to the current MMU mapping).

5.12 Disassembly

When the program stops outside your source (in ROM, in code without SLD, in data), or when you want to see the actual instructions, open the disassembly: right-click in the editor and choose **Open Disassembly View**, or run **Debug: Open Disassembly View** from the Command Palette. Instructions are shown with labels from the SLD and source lines. In the disassembly too you can click left of an address to set a breakpoint, and step by instruction. Without an SLD file this is the only place to debug.

5.13 Editing registers and flags

In the **Variables** section double-click a register value (or right-click **Set Value**), type a new value and press **Enter**. Numbers can be written as `$FF`, `0xFF`, `#FF`, `FFh`, `255` or in binary `%1010`. Flags (`S` `Z` `H` `P/V` `N` `C`) take 0 or 1. To move PC to another line, **Jump to Cursor** (5.3) is more convenient – the editor follows it.

6 Expression language

The same expression language is used everywhere: in breakpoint and watchpoint conditions, in event conditions, in logpoints (inside curly braces), in the Watch section, on hover, in the Debug Console and in keywords in comments (7). Expressions are evaluated by the emulator in 32-bit signed integers. A condition holds when the result is non-zero.

6.1 Numbers

Written as	Meaning
255	decimal
\$FF, 0xFF	hexadecimal
0b1010	binary
true, false	1 and 0

CAUTION

The forms #FF, 0xFF and %1010 do **not** work in conditions entered in VS Code (% is the remainder operator). They work only in WPMEM / LOGPOINT / ASSERTION comments and when editing a register value.

6.2 Registers and flags

Name	Meaning
A F B C D E H L I R	8-bit registers
AF BC DE HL IX IY SP PC	16-bit registers
AF' BC' DE' HL'	shadow registers (also written AF2 BC2 DE2 HL2)
IXH IXL IYH IYL	high and low halves of IX and IY
CF ZF SF PF HF NF	flags carry, zero, sign, parity/overflow, half-carry, subtract (0/1)
IFF1 IFF2 IM HALTED	interrupt flip-flops, interrupt mode, CPU halted on HALT

Register names from the first four rows (including AF' and IXH) and IM, IFF1, IFF2 may also be written in lower case (the extension converts them); write all other names (AF2, the flags, HALTED, the names in the following tables) in **upper case** as in the table.

6.3 Memory, Next registers and MMU

Written as	Value
<code>mem[a]</code>	byte at address <code>a</code> (CPU memory 0–\$FFFF with the current mapping)
<code>mem16[a]</code>	16-bit word at address <code>a</code> (little endian)
<code>stack[n]</code>	word on the stack: <code>stack[0]</code> = at <code>SP</code> , <code>stack[1]</code> = at <code>SP+2</code> ...
<code>phys[page, offset]</code>	byte in physical 8K page (offset 0–\$1FFF), regardless of mapping
<code>nextreg[r]</code>	value of Next register <code>r</code> (0–255)
<code>mmu[s]</code>	8K page set for slot <code>s</code> (0–7) by the MMU registers
<code>page[s]</code>	8K page slot <code>s</code> actually uses

Reads in expressions have no side effects: they do not trigger watchpoints and do not change the state of registers whose reads have side effects.

6.4 Time and raster

Name	Value
<code>FRAME</code>	frame number since start
<code>CYCLE</code>	time since start in 28 MHz master clock cycles
<code>TFRAME</code>	time since the start of the frame in 28 MHz cycles
<code>VC_ULA</code>	display line (0 = first paper line, 0–191 paper)
<code>PHC</code>	pixel column (0 = first paper pixel)
<code>CVC</code>	line as compared by the Copper and the line interrupt ($\text{NR } \$1\text{E}/\$1\text{F}$); $= \text{VC_ULA} + \text{NR } \$64$
<code>HC_ULA</code>	7 MHz horizontal counter used by the Copper
<code>RAW_HC</code> , <code>RAW_VC</code>	raw raster counters
<code>MACHINE</code>	machine type: 0 = 48K, 1 = 128K, 2 = +3, 3 = Pentagon, 4 = Next

6.5 Labels

Global labels from the source (from the SLD) are written directly and evaluate to their address:

`mem[counter]`, `HL == next_colour`, `mem16[score] > 100`. The case must match the source. If a label has the same name as a register, the register wins; force the label with an at sign: `@c`. **Local labels with a dot** work too, in their full form: `main_loop.wait` evaluates to the address of the local label `.wait` inside `main_loop`.

6.6 Operators

From the lowest precedence to the highest (operators on one line have the same precedence):

Operators	Meaning
<code>or</code>	logical OR (short-circuit)
<code>and</code>	logical AND (short-circuit)
<code>not</code>	logical negation
<code>== != < > <= >=</code>	comparison (result 1 or 0, signed)
<code> ^</code>	bitwise OR, XOR
<code>&</code>	bitwise AND
<code><< >></code>	shifts
<code>+ -</code>	addition, subtraction
<code>* / %</code>	multiplication, integer division, remainder
<code>-x ~x</code>	unary minus, bitwise NOT

Unlike C, the bitwise operators bind **tighter** than comparisons, so `A & $80 == $80` means `(A & $80) == $80`. Parentheses never hurt, though. Write `and`, `or` and `not` instead of `&&`, `||` and `!`.

6.7 Names for events and watchpoints

In event conditions (5.7) data about the event are available as well: `REG`, `VALUE`, `PREV`, `PORT`, `SOURCE` (with the constants `CPU`, `COPPER`, `DMA`), `CPC`, `SRC`, `DST` – the table in the events section shows which of them each event has. In a watchpoint condition (the fourth `WPMEM` parameter) there are `ADDR` (the accessed address) and `VALUE` (the byte transferred). A name the event does not carry is rejected by the emulator when the condition is entered.

6.8 Examples

Expression	True when...
<code>A == 5</code>	register A is 5
<code>HL >= \$4000 and HL < \$5800</code>	HL points into the ULA screen
<code>not ZF</code>	the Z flag is clear
<code>mem[counter] == 3</code>	the byte at label <code>counter</code> is 3
<code>stack[0] == main_loop</code>	the return address on top of the stack is <code>main_loop</code>
<code>nextreg[\$07] & 3 == 3</code>	the CPU runs at 28 MHz
<code>mmu[6] == 20</code>	page 20 is in slot 6 (\$C000)
<code>FRAME % 50 == 0</code>	every fiftieth frame
<code>VC_ULA >= 192</code>	the beam is below the paper (bottom border)

7 sjasmpplus keywords in comments

Watchpoints, logpoints and assertions can be written right into the source as **special comments** (the same syntax as in DeZog). The advantage: they are part of the code, do not get lost, and work in every debug session without clicking.

7.1 SLDOPT – enabling them in sjasmpplus

sjasmpplus writes these comments into the SLD file only when the source contains (preferably at the beginning) the line:

```
SLDOPT COMMENT WPMEM, LOGPOINT, ASSERTION
```

List just the words you use. sjasmpplus looks for the word anywhere in the comment and is case-sensitive (write them in upper case); it does not write block comments `/* */` into the SLD. A comment on a line of its own applies to the next instruction, a comment after an instruction to the instruction on the same line. In a macro the comment refers to the place the macro is invoked from.

7.2 WPMEM – watchpoint

```
; WPMEM [address [, size [, access [, condition]]]]
```

- **address** – an expression; without it, the address of the line the comment is on (typically a data label).
- **size** – the number of bytes watched, 1 by default.
- **access** – `r` read, `w` write, `rw` both (default).
- **condition** – optional; stops only when it holds (`ADDR` and `VALUE` can be used).

```
score:  dw 0           ; WPMEM           - watch the 1 byte at "score" (read and write)
buffer: ds 16         ; WPMEM buffer,16,w - stop on a write into the 16 bytes
lives:  db 3          ; WPMEM lives,1,w,VALUE == 0 - stop when zero is written
           ; WPMEM $4000, $1800, w - any write into the ULA bitmap
```

After stopping, the Call Stack header shows the reason **WPMEM** and the Debug Console a line with the address range and the comment location, for example `WPMEM $8030-$803F (game.asm:120) - WPMEM buffer,16,w`.

7.3 LOGPOINT – debug print

```
; LOGPOINT [GROUP] text ${expression} text ${expression:format}
```

Prints the text to the Debug Console on every pass; `${...}` is replaced by the value of the expression. The formats are the same as for VS Code logpoints: `hex`, `unsigned`, `signed`, `char`, `bits`, `flags`, `labels`. An optional group in square brackets right after the word is printed at the start of the line (`[AI] ...`). The output in the Debug Console links to the source line; click it to go there.

```
ld a,(hl)
; LOGPOINT [MAP] read ${A:hex} from ${HL:hex}
ld (de),a
```

7.4 ASSERTION – assertion

```
; ASSERTION expression
```

Stops the program when the expression is **false** (zero). An empty **ASSERTION** always stops. The Call Stack header shows the reason **assertion** with the description *ASSERTION failed: A < 10*; the same text with the source location is printed to the Debug Console.

```
cp 10          ; ASSERTION A < 10 - stop when the expression is false
; ASSERTION SP > stack_bottom and SP <= stack_top
ret
```

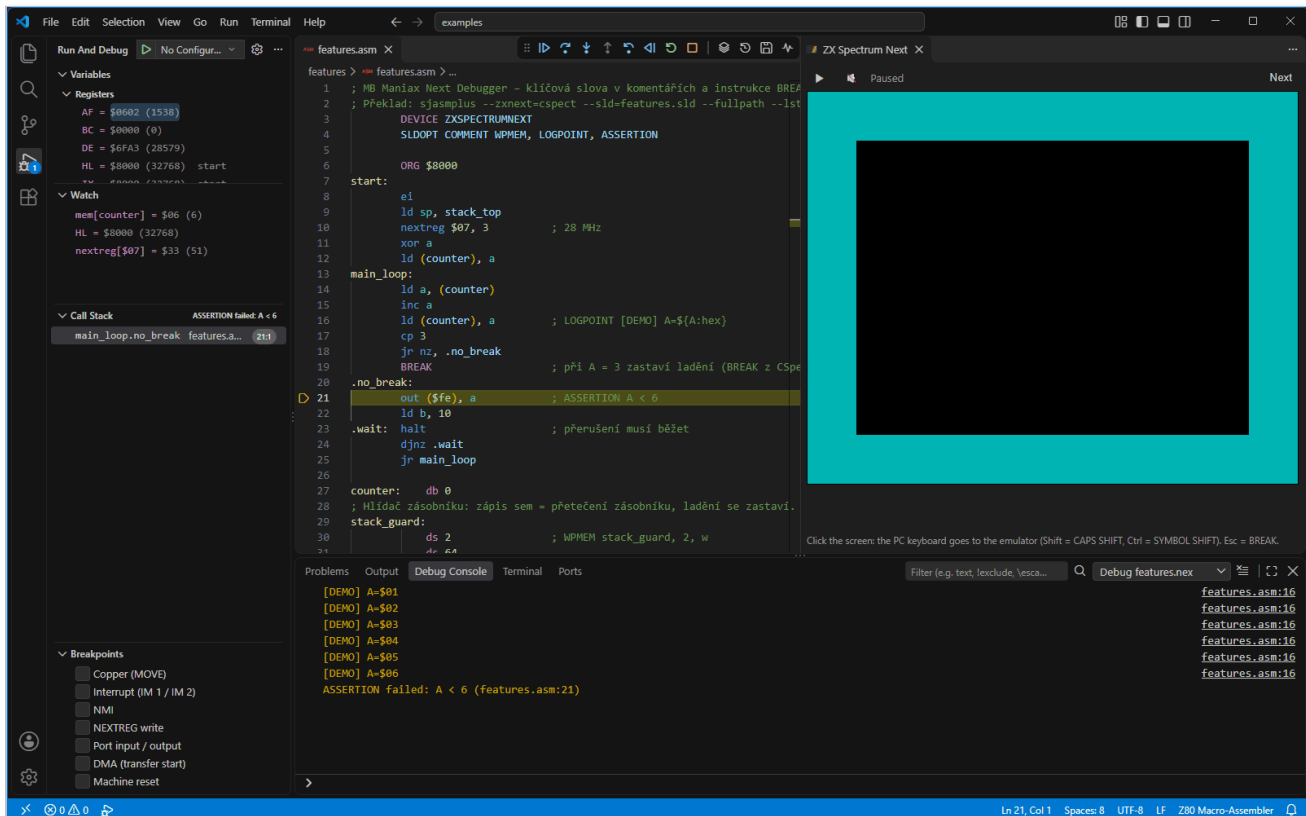


Figure 8 Stopped on an ASSERTION: the reason in Call Stack, LOGPOINT output in the Debug Console, and the source with the comments.

7.5 When keywords are evaluated

LOGPOINT and ASSERTION are breakpoints at the address of their line – they are evaluated **before** the instruction on that line is executed. In the example `ld a, (hl) ; LOGPOINT ... ${A}`, A would be printed before the load. To see the result of an instruction, put the comment on a line of its own below it (it then applies to the next instruction), as in the example above.

7.6 Expression syntax in comments

Comments accept the language from chapter 6 plus the DeZog forms, which the extension converts:

DeZog	The same in the expression language
<code>b@(x), w@(x)</code>	<code>mem[x], mem16[x]</code>
<code>&&, , !</code>	<code>and, or, not</code>
<code>=</code>	<code>==</code>
<code>#FF, 0FFh</code>	<code>\$FF</code>
<code>%1010</code> (where a number is expected)	<code>0b1010</code>

Errors in comments (unknown label, invalid size, unknown access or format) are reported by the extension as a notification in the bottom right corner, in the form *WPMEM in a comment (game.asm:120): ...*; the other keywords keep working.

7.7 Turning them off

Each kind can be turned off in `launch.json` without changing the source: `"wpmem": false, "logpoints": false, "assertions": false` (all default to `true`).

7.8 CSpect BREAK

The **BREAK** instruction from the CSpect emulator stops the program like a breakpoint – handy where writing a stop into the code is easier than clicking.

- `sjasmpplus` with `--zxnext=cspect` assembles **BREAK** as `FD 00`. The older CSpect `DD 01` and ZEsarUX `ED FF` also work.
- The machine stops just **after** the instruction (`PC = BREAK address + 2`): the yellow line is on the instruction after **BREAK**, the Call Stack description gives the address of the **BREAK**.

The Call Stack header shows the reason **BREAK** with the description *Paused on BREAK at \$8003*. Continue as usual (`F5`, steps). Turn stopping off in `launch.json`: `"magicBreakpoint": false`.

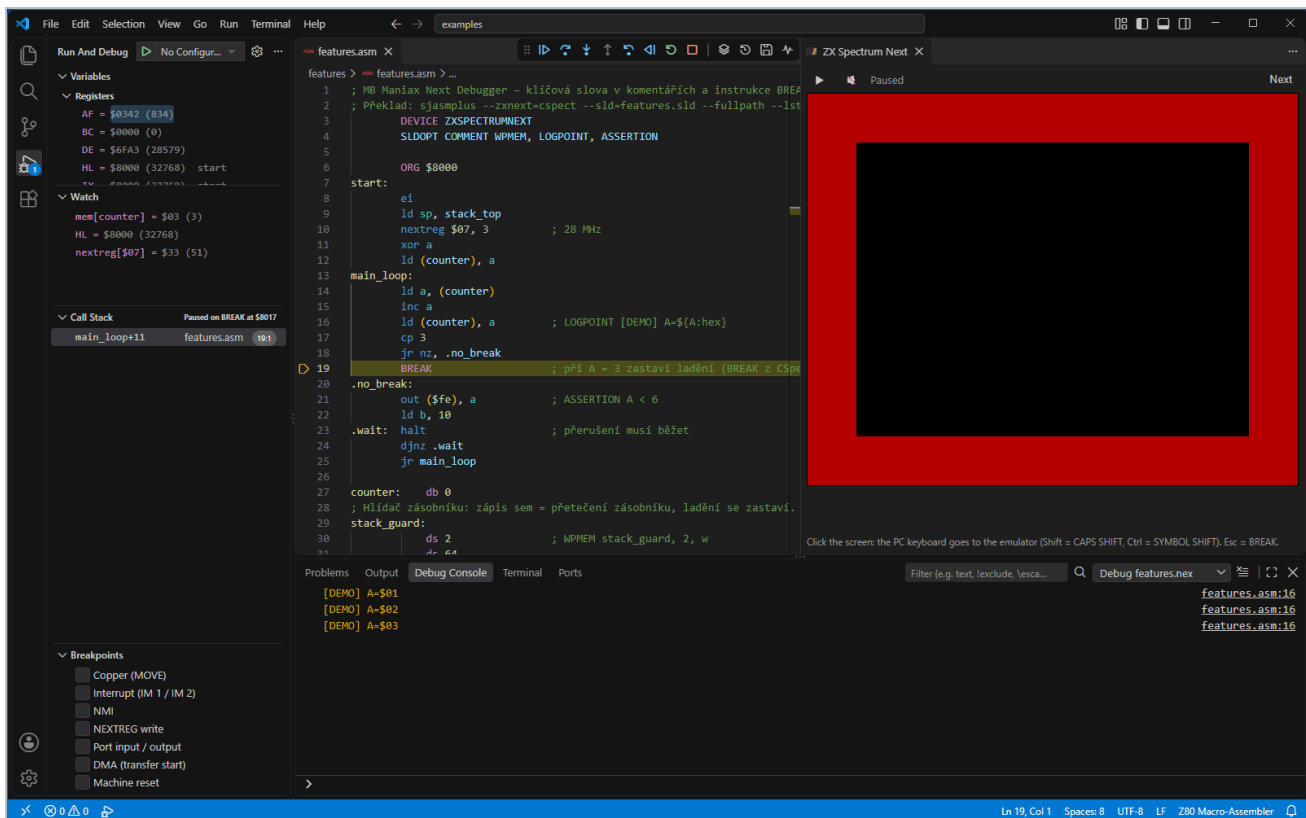


Figure 9 Stopped on a BREAK instruction: the reason and the BREAK address in Call Stack.

CAUTION

On real hardware FD 00 runs as a NOP, but DD 01 runs as LD BC,nn and swallows the next two bytes. Don't leave DD 01 in release builds. With magicBreakpoint off, DD 01 behaves exactly as on a real Next.

8 Emulator screen

When debugging starts, the **ZX Spectrum Next** panel with the emulator screen opens to the right of the editor. If you close it, reopen it with **MB Maniax Next: Show Emulator Screen** from the Command Palette. The panel is an ordinary editor tab – drag it by its tab elsewhere (to the bottom, into its own window, onto a second monitor: right-click the tab and choose **Move into New Window**).



Figure 10 The ZX Spectrum Next panel: the bar at the top with pause, sound, status and the Next button.

The bar at the top of the panel:

- **II / ▶** (*Pause / continue*) – pauses or resumes the program, like **F6** / **F5**.
- **🔊 / 🔇** (*Sound*) – turns sound off and on.
- **Status** – *Waiting for a debug session...* (no debug session), *Paused*, or *Running · 50 frames/s* – the actual emulation speed. 50 frames/s is normal; a markedly lower number means the computer cannot keep up with the emulation.
- **Next** – opens the Next panel (9).

8.1 Keyboard

Click the screen: while it has the focus, the PC keyboard goes to the emulator. **Shift** is CAPS SHIFT, **Ctrl** is SYMBOL SHIFT and **Esc** is BREAK. Other PC keys (arrows, **Backspace** ...) are translated by the emulator's key map. Clicking outside the screen (for example into the editor) gives the keyboard back to VS Code and releases all keys held in the emulator. This hint is also shown below the screen in the panel.

NOTE

Debugging shortcuts (**F5**, **F10** ...) work only when VS Code has the focus, not the emulator screen. If **F5** “does nothing”, click into the editor first.

8.2 Sound

The browser inside VS Code may play sound only after a user action – sound therefore starts the first time you click the screen (or the  button). Sound plays only while running; it is silent while paused and stepping. If the sound is choppy, the emulation cannot keep up – check the frames per second in the bar and [chapter 14](#).

8.3 Pause, stops and the picture

While the program is stopped (breakpoint, step, pause), the screen shows the state at the moment it stopped. If the machine stops in the middle of a frame, the top part of the picture is already new and the bottom part still from the previous frame – just like the beam. The Next panel can draw a highlight of the selected sprite, tile or line onto the screen ([9.2](#)).

9 Next panel

The Next panel shows the state of the Next hardware that CPU registers don't show: sprites, patterns, palettes, the individual display layers, Layer 2, the tilemap, the Copper program and all NEXTREG registers.

9.1 Opening and controls

Open the panel in one of three ways:

- with the button on the debug toolbar (layers icon, tooltip **Next Panel (Sprites, Palettes, ...)**),
- with the **Next** button in the bar of the emulator screen panel,
- with the command **MB Maniax: Next Panel (Sprites, Palettes, ...)** from the Command Palette.

The panel opens next to the editor as another tab named **Next**; like any tab, it can be dragged elsewhere.

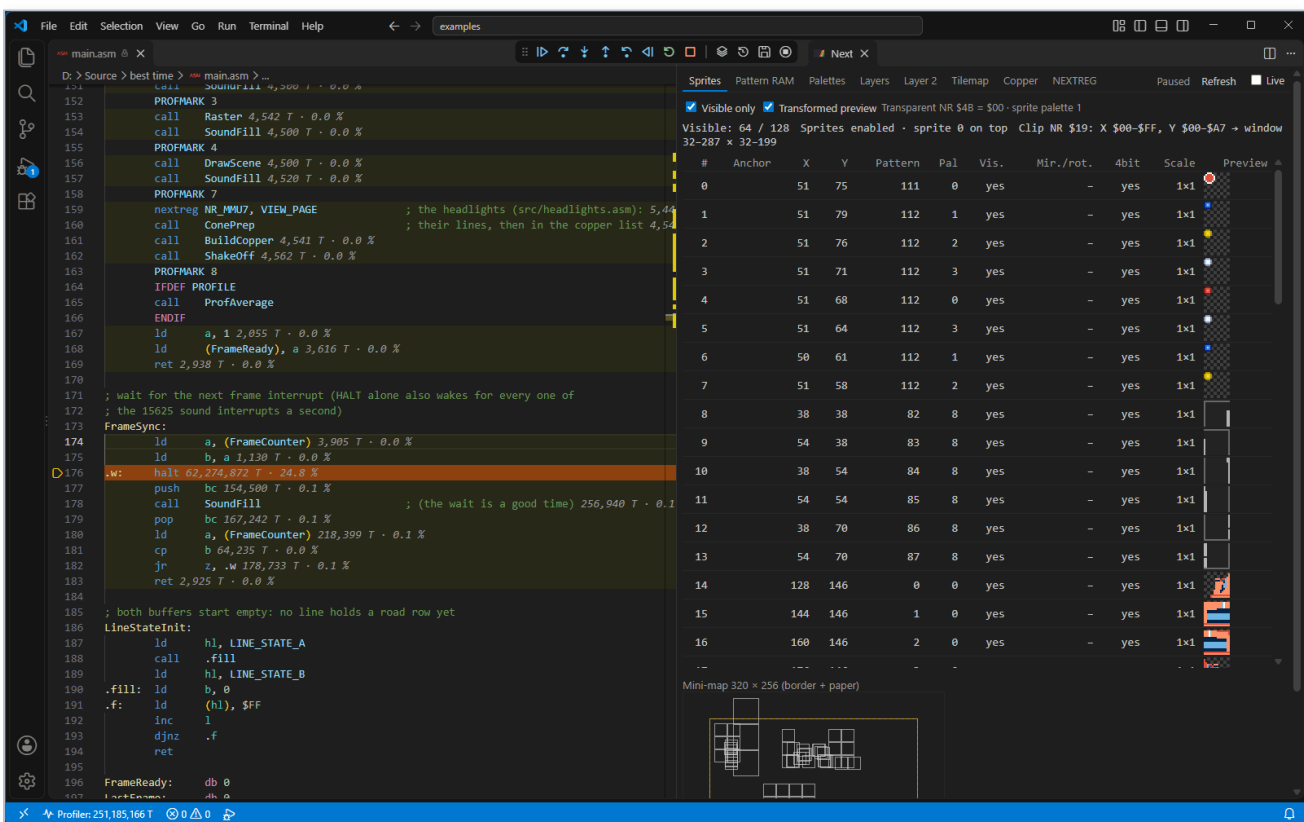


Figure 11 The Next panel inside VS Code next to the source, Sprites tab.

At the top of the panel is a bar:

- **Tabs** on the left – **Sprites**, **Pattern RAM**, **Palettes**, **Layers**, **Layer 2**, **Tilemap**, **Copper**, **NEXTREG**. The panel remembers the last open tab.
- On the right the **status** – *Waiting for a debug session...*, *Running* or *Paused*,
- **Refresh** – reloads the data of the displayed tab,
- **Live** – refreshes the displayed tab while running too, twice per second.

Without Live, the displayed tab is refreshed every time the machine stops (breakpoint, step, pause) and with the Refresh button. Only the visible tab is refreshed, so hidden tabs cost nothing.

9.2 Highlighting on the emulator screen

Some tabs can point out the selected item right on the emulator screen (the ZX Spectrum Next panel): the Sprites tab the sprite under the mouse or the selected sprite, the Tilemap tab the tile under the mouse, the Copper tab the line the WAIT under the mouse waits for. The highlight rectangle is drawn over the picture; it disappears when the mouse leaves the item or you switch tabs.

The following sections describe the individual tabs. Register numbers are written as NR \$xx; almost every label in the panel has a tooltip with an explanation (hover over it).

9.3 Sprites

Visible only ☒ Transformed preview ☒ Transparent NR \$4B = \$00 · sprite palette 1

Visible: 79 / 128 Sprites enabled · sprite 0 on top Clip NR \$19: X \$00-\$FF, Y \$00-\$A7 → window 32-287 x 32-199

#	Anchor	X	Y	Pattern	Pal	Vis.	Mir./rot.	4bit	Scale	Preview
0		50	75	80	0	yes	-	yes	1x1	
1		51	77	81	1	yes	-	yes	1x1	
2		51	74	81	2	yes	-	yes	1x1	
3		50	70	81	3	yes	-	yes	1x1	
4		49	68	81	0	yes	-	yes	1x1	
5		47	66	81	3	yes	-	yes	1x1	
6		46	64	81	1	yes	-	yes	1x1	
7		44	62	81	2	yes	-	yes	1x1	
8		38	38	82	8	yes	-	yes	1x1	
9		54	38	83	8	yes	-	yes	1x1	
10		38	54	84	8	yes	-	yes	1x1	
11		54	54	85	8	yes	-	yes	1x1	
12		38	70	86	8	yes	-	yes	1x1	
13		54	70	87	8	yes	-	yes	1x1	
14		26	161	36	5	yes	-	yes	2x2	
15		58	161	37	5	yes	-	yes	2x2	
16		90	161	38	5	yes	-	yes	2x2	

Mini-map 320 × 256 (border + paper)

Figure 12 The Sprites tab: a table of 128 sprites with previews and a screen mini-map on the right.

Toolbar: **Visible only** hides sprites with the visibility bit cleared (on by default); **Transformed preview** draws previews with mirroring and rotation as they appear on screen. To the right is the status: the transparent colour (Transparent NR \$4B = \$E3) and which sprite palette is used.

Summary line: the number of visible sprites (Visible: 12 / 128), whether sprites are enabled (*Sprites disabled* (NR \$15 b0) is highlighted as a warning), whether they are drawn over the border (*over border*) and whether sprite 0 is on top (*sprite 0 on top*), and the sprite clip window – the raw NR \$19 values and the resulting window in screen coordinates.

The **table** has a row for each sprite:

Column	Meaning
#	sprite number 0–127
Anchor	for a relative sprite <code>rel</code> → <code>#n</code> : the number of its anchor
X, Y	position in 320×256 coordinates including the border ($X\ 0$ = left border edge, the paper starts at 32, 32)
Pattern	the pattern number the hardware reads
Pal	palette offset (the upper 4 bits of the colour index)
Vis.	visibility bit
Mir./rot.	<code>X</code> , <code>Y</code> = mirroring, <code>R</code> = rotation
4bit	4-bit sprite (128 B per pattern) instead of 8-bit (256 B)
Scale	magnification $X \times Y$ (1, 2, 4, 8)
Preview	the pattern in the colours of the active palette; checkerboard = transparent pixels

For relative sprites (anchor + relative sprites) the table already shows the **resolved** values – position, pattern, palette and transformations as the hardware draws them. The tooltip over a row shows the raw attributes (5 bytes), the pattern address in Pattern RAM and the sprite's rectangle on screen. Invisible sprites are greyed out.

The **mini-map** on the right is the whole 320×256 area (border + paper): the 256×192 paper is outlined, the clip window dashed, each visible sprite drawn as a rectangle. Hovering over a table row or a sprite in the mini-map highlights the sprite in the mini-map and on the emulator screen. Click to select it – the highlight then stays even when the mouse moves away (click again to deselect). Clicking the mini-map selects the topmost sprite at that point and scrolls the table to its row.

9.4 Pattern RAM

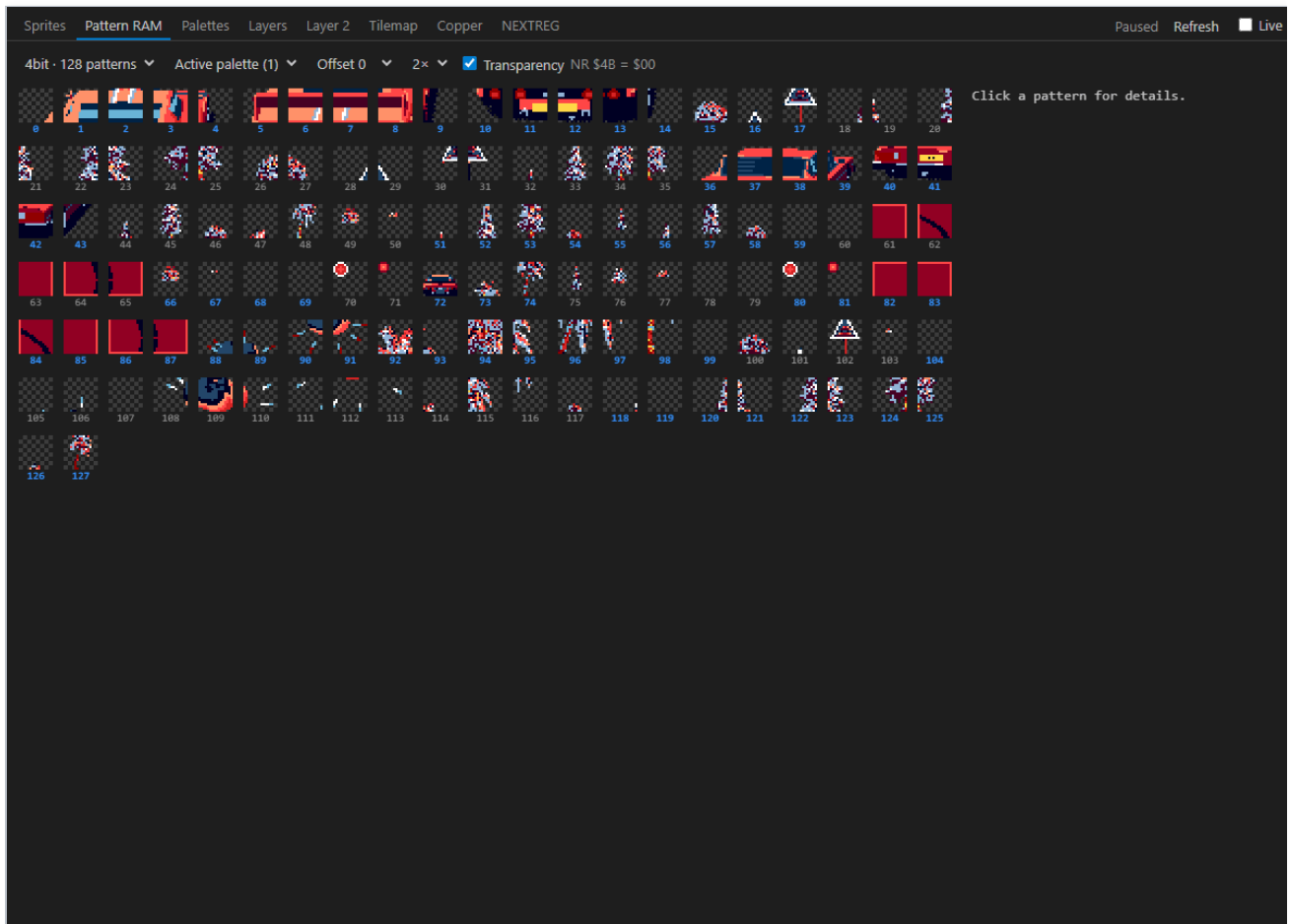


Figure 13 The Pattern RAM tab: a grid of all patterns and the detail of the selected pattern.

The whole sprite pattern memory (16 KB) as a grid. **Toolbar:**

- **Pattern format** – 8bit · 64 patterns or 4bit · 128 patterns.
- **Sprite palette** – Active palette (per NR \$43 bit 3, which one in parentheses), or fixed *Sprites 1* / *Sprites 2*.
- **Offset** 0–15 – the palette offset for the preview (for 4-bit patterns the upper nibble of the index, for 8-bit ones added to the upper nibble).
- **Grid zoom** and **Transparency** – draw pixels with the NR \$4B value as transparent (checkerboard).

The tooltip over a pattern shows its number and address and which sprites use it – visible and hidden ones separately. **Click a pattern:** an enlarged detail appears below the grid with the addresses, the list of sprites using it and, when you hover over a pixel, its coordinates, value (for 4-bit patterns also which nibble), palette index and colour in RGB333 and RGB888.

9.5 Palettes

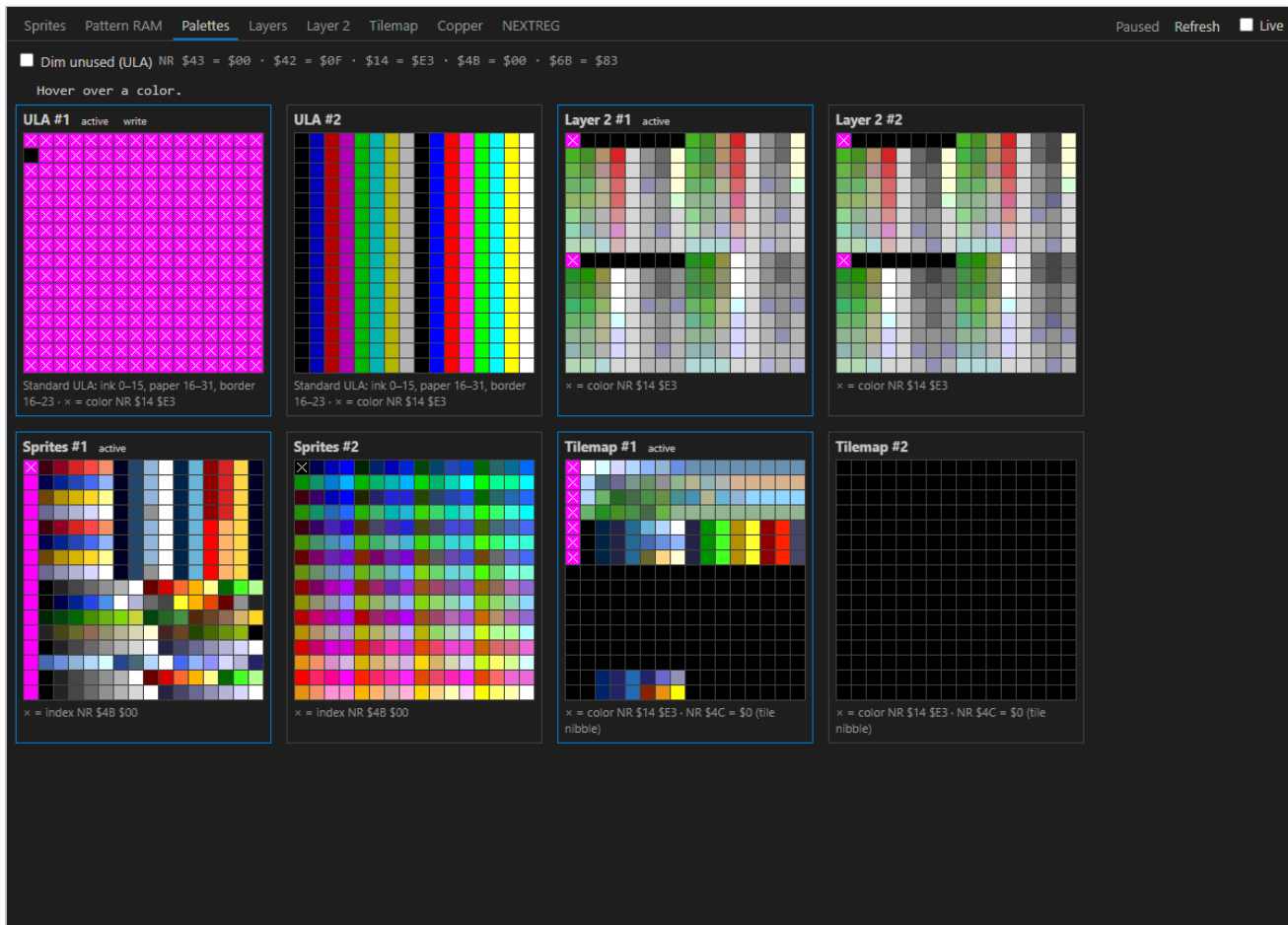


Figure 14 The Palettes tab: eight palettes of 256 colours with the active palette and the write target marked.

All eight Next palettes, each as a 16×16 grid: **ULA #1** and **#2**, **Layer 2 #1** and **#2**, **Sprites #1** and **#2**, **Tilemap #1** and **#2**. The colour index increases row by row from left to right (the first row is 0–15).

- The **active** badge – this palette is used for rendering.
- The **write** badge – writes through NR \$41/\$44 currently go here (per NR \$43 bits 6:4).
- A cross **x** marks the transparent colour: for ULA, Layer 2 and tilemap the colour NR \$14, for sprites the index NR \$4B. The legend below the palette says so and, for ULA, describes which indexes are ink, paper and border (standard ULA or ULANext with the format from NR \$42).
- **Dim unused (ULA)** – in the ULA palettes, dims the entries the current mode does not use.

The line next to the check box shows the registers NR \$43 · \$42 · \$14 · \$4B · \$6B. Hovering over a colour prints at the top: the palette, index (hex and dec), RGB333 and the R G B components (0–7), the 8-bit RRRGGGBB value, the colour as #rrggb and the role of the colour (ink, paper, border, transparent, unused...).

9.6 Layers

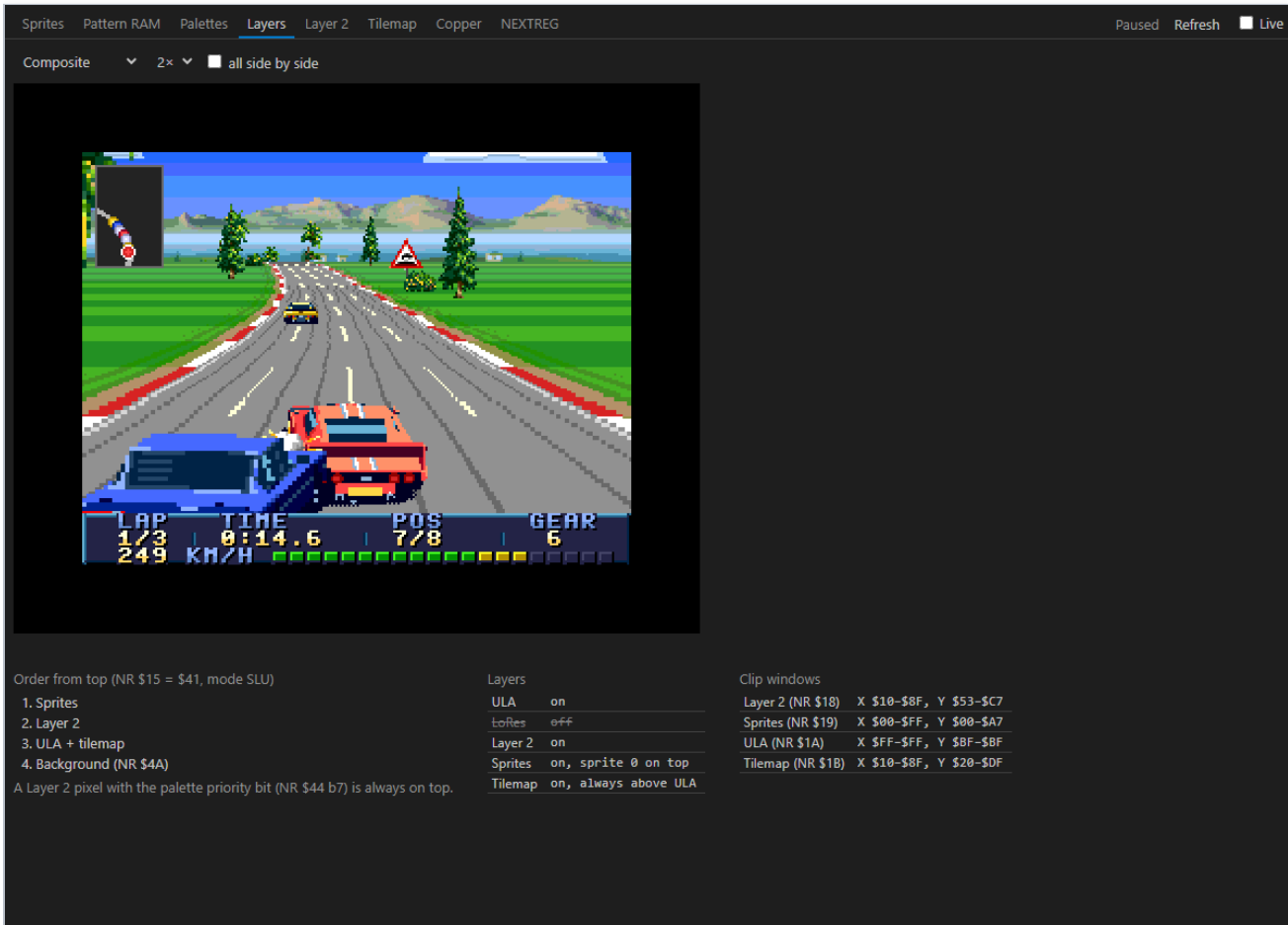


Figure 15 The Layers tab: the selected display layer, layer order and clip windows.

Splits the picture into the layers the Next composes it from. The drop-down on the left selects what to see: *Composite*, *ULA*, *ULA shadow*, *Layer 2*, *Layer 2 shadow*, *Sprites*, *Tilemap*, *Background*; next to it is the 1× / 2× zoom. The **all side by side** check box shows thumbnails of all layers at once – click a thumbnail to show it large.

A checkerboard means the layer draws nothing there (transparent). Hovering prints the coordinates below the picture (x, y in the 320 × 256 space and the column in the 640 pixel wide picture) and the pixel colour (#rrggbb and RGB333). Layers are drawn for the whole frame with the registers replayed line by line, so raster splits and Copper effects show up too.

Below the picture are three blocks of information:

- **Order from top** (NR \$15, mode SLU, LSU ...) – the layers from the top down, with the background (NR \$4A) at the end. For the colour blending modes (L+U add / subtract) it describes how ULA and tilemap are blended with Layer 2. Disabled layers are struck through. A note reminds you that a Layer 2 pixel with the palette priority bit (NR \$44 b7) is always on top.
- **Layers** – whether ULA (NR \$68 b7), LoRes, Layer 2 (NR \$69 b7 / port \$123B), sprites (also in border, sprite 0 on top) and tilemap (always above ULA, or by attribute) are enabled.
- **Clip windows** – the Layer 2, sprite, ULA and tilemap windows.

NOTE

If the machine stops in the middle of a frame, lines below the beam are drawn with the current register state – on the emulator screen the previous frame is still there, so they may differ.

9.7 Layer 2

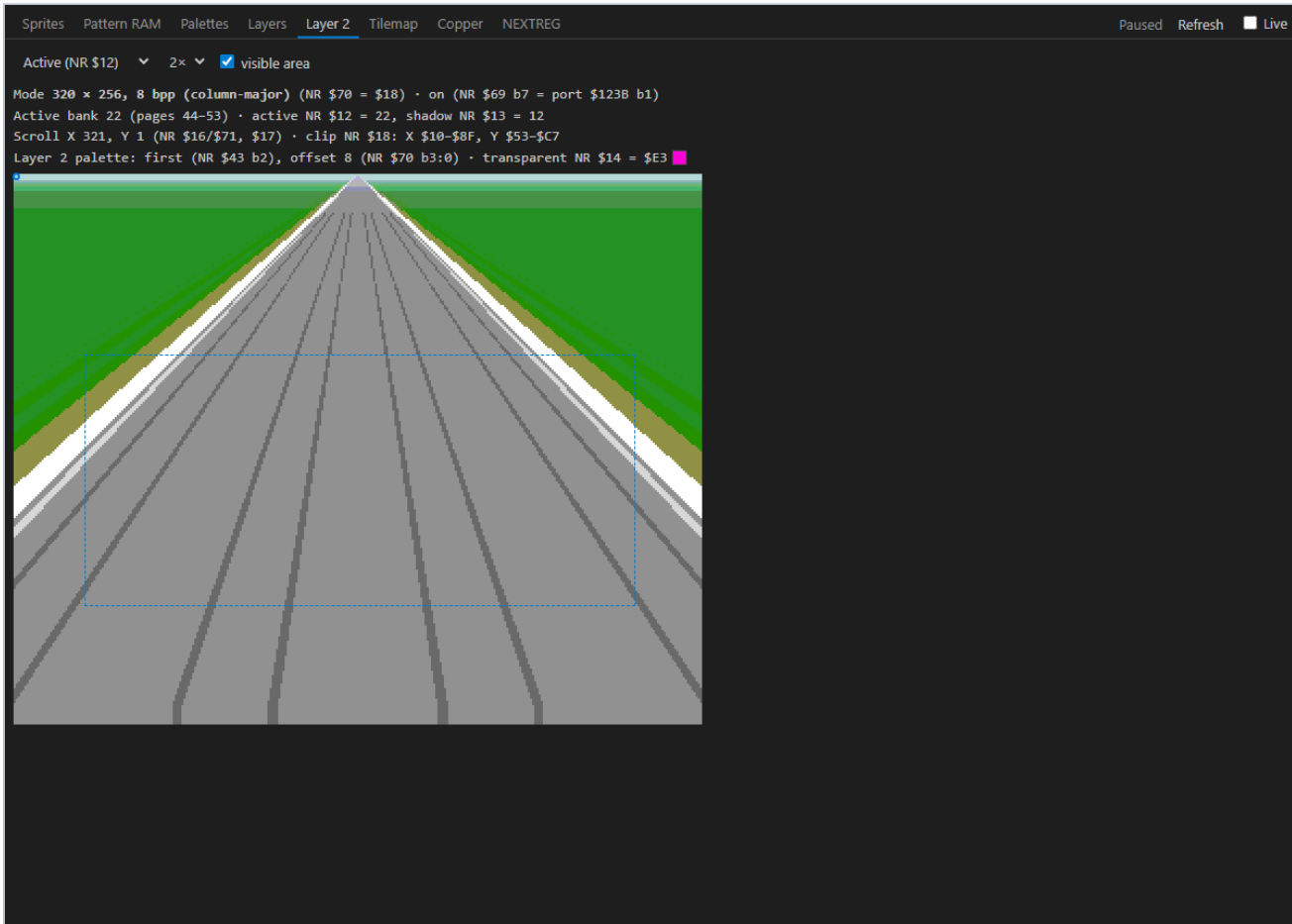


Figure 16 The Layer 2 tab: the whole Layer 2 content from memory with the visible part marked.

The whole Layer 2 content from memory – not just what is visible. The drop-down on the left selects the bank: *Active (NR \$12)* or *Shadow (NR \$13)*; then the zoom and **visible area**: a dashed rectangle shows what is on screen after scroll and clip, a circle marks the top-left corner of the screen.

Info lines at the top: the mode (NR \$70 – $256 \times 192, 8 \text{ bpp}$, $320 \times 256, 8 \text{ bpp}$ (column-major) or $640 \times 256, 4 \text{ bpp}$ (column-major)) and whether Layer 2 is on; the bank and its pages; scroll X/Y (NR \$16/\$71, \$17) and clip NR \$18; the palette (first/second per NR \$43 b2), palette offset (NR \$70 b3:0) and the transparent colour NR \$14. Hovering prints the bank and offset (and the 8K page), the byte at that point (for 4 bpp which nibble), the colour index and, if so, (transparent).

9.8 Tilemap

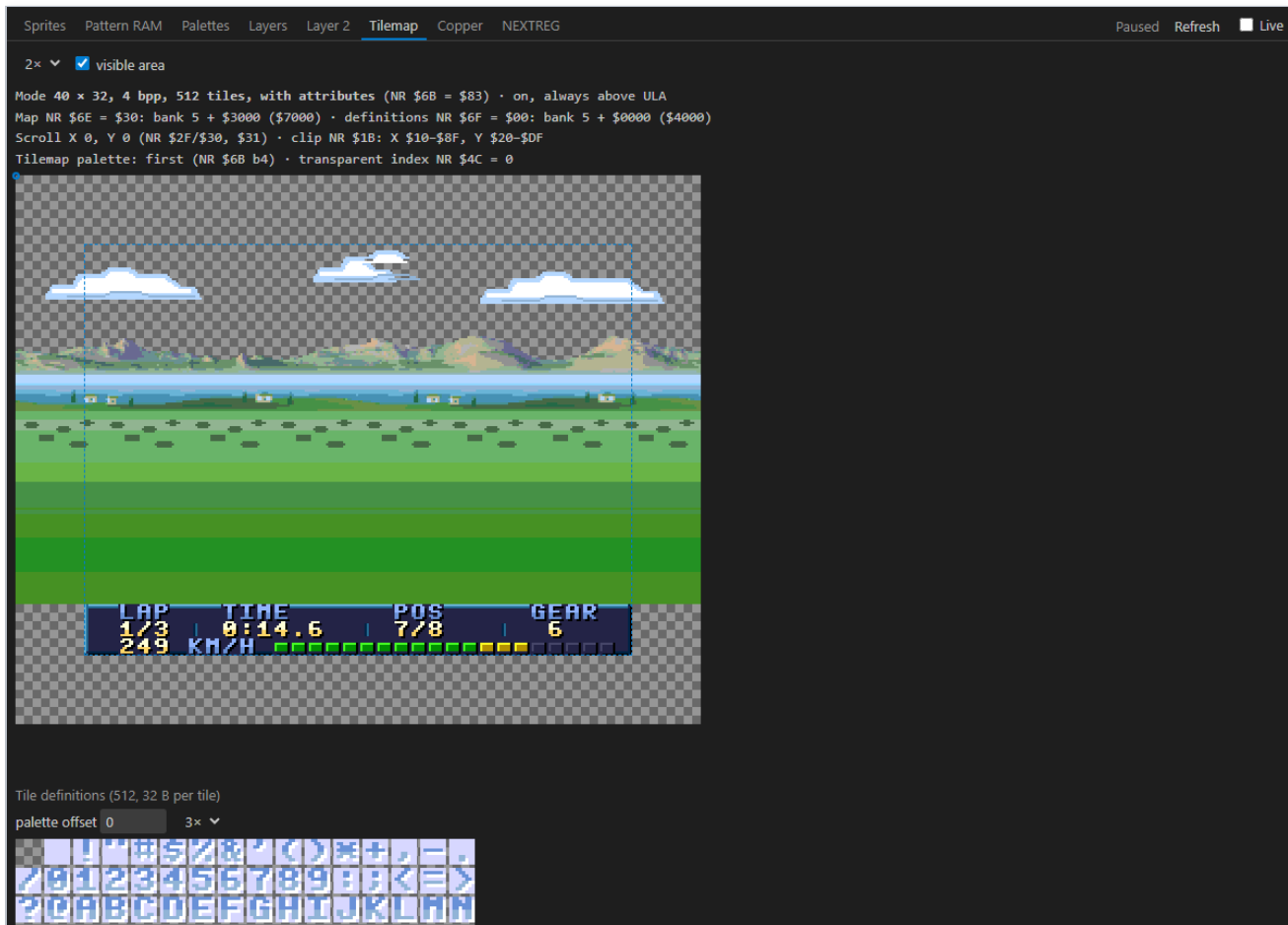


Figure 17 The Tilemap tab: the whole map with the visible part marked, and the tile definitions below it.

At the top is the whole map (40×32 or 80×32 tiles) with $1\times$ / $2\times$ zoom and the **visible area** option (dashed visible part after scroll and clip, circle = top-left corner of the screen). Info lines: the mode (NR \$6B – number of columns, 4 bpp or text 1 bpp, 256/512 tiles, with or without attributes – then the default attribute NR \$6C applies), whether the tilemap is on and whether it is always above ULA; the map (NR \$6E) and definitions (NR \$6F) addresses in bank 5 or 7; scroll (NR \$2F/\$30, \$31) and clip NR \$1B; the tilemap palette (NR \$6B b4) and the transparent index NR \$4C (in text mode the colour NR \$14).

Hover over a tile in the map: its column and row, the tile index, the attribute broken down into palette, mirroring, rotation and priority relative to ULA, and the addresses of the map entry and the definition are printed. The tile is also framed in the definitions below the map and **highlighted on the emulator screen** where it lies after scrolling.

Tile definitions below the map: all defined tiles in a grid of 16 per row; above them the **palette offset** for the preview (0–15) and $2\times$ – $4\times$ zoom. Hovering shows the tile number and the address of its definition.

9.9 Copper

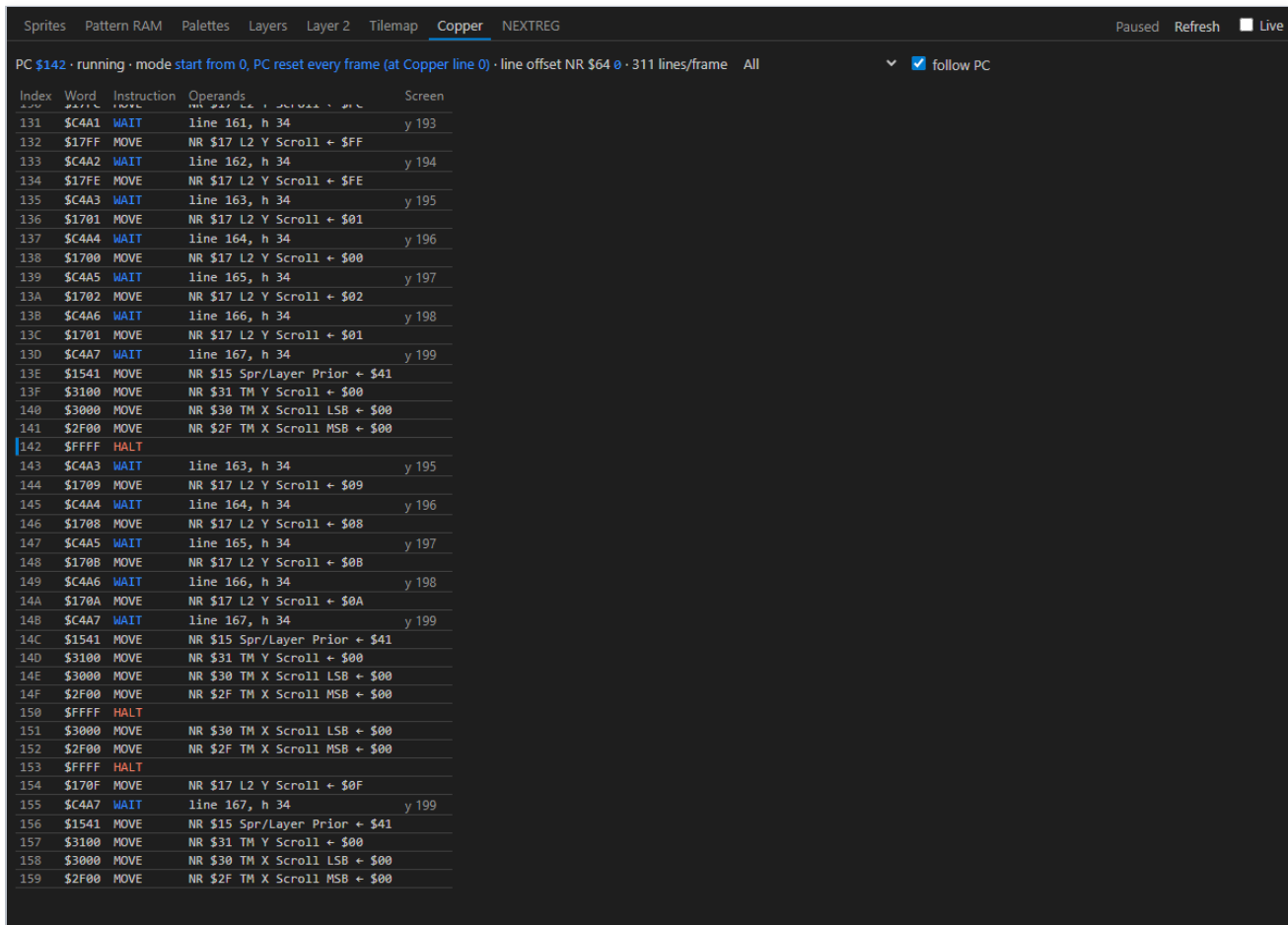


Figure 18 The Copper tab: the Copper program with decoded instructions; the row with PC is highlighted.

The whole Copper program (1024 instructions). The status line at the top: PC \$xxx (where the Copper is now), *running / stopped*, the mode (NR \$62 bits 7:6: *stop, start from 0 (PC = 0 when enabled), resume from current PC, start from 0, PC reset every frame*), the line offset NR \$64 and the number of lines per frame. The **Which instructions to show** drop-down: *All, Without trailing NOPs, Up to first HALT*; **follow PC** scrolls the table so the row with PC is visible.

Column	Meaning
Index	position in the program (0–3FF)
Word	the 16-bit instruction in hex
Instruction	MOVE, WAIT, NOP, HALT
Operands	for MOVE NR \$40 Palette Index ← \$10 (the tooltip breaks down the bits of the value), for WAIT line 100, h 0
Screen	for WAIT the screen line (y 132 in the 320 × 256 area including the border), <i>off screen</i> , or <i>never</i> when the frame has no such line

Hover over a WAIT row – the line it waits for is highlighted with a horizontal line on the emulator screen. The tooltip of a WAIT explains which line and horizontal position it waits for; HALT is a WAIT for line 511, which never happens.

9.10 NEXTREG

Sprites Pattern RAM Palettes Layers Layer 2 Tilemap Copper NEXTREG										Paused Refresh ☐ Live
Search (number, name, bit)...										
All groups ☐ all 256 ☐ changed only 141 shown · 0 changed										
Reg	Name	Group	Access	Hex	Dec	Bin	Bits			
\$00	Machine ID Machine identification	Machine	R	\$0A	10	0000 1010	ID: ZX Spectrum Next			
\$01	Core Version Core version (sub-minor in NR \$0E)	Machine	R	\$32	50	0011 0010	major: 3 minor: 2			
\$02	Reset Reset and NMI generation	Machine	R/W*	\$00	0	0000 0000	expansion bus reset: no NMI from I/O trap: no Multiface NMI pending: no DivMMC NMI pending: no last reset was hard: no last reset was soft: no			
\$03	Machine Type Machine type and timing	Machine	R/W*	\$33	51	0011 0011	NR \$44 colour write pending: no timing: +3 timing lock: no machine type: +3			
\$04	Config Mapping 16K bank in the ROM area in config mode	MMU	W*	\$00	0	0000 0000	16K bank: 0			
\$05	Peripheral 1 Joysticks, 50/60 Hz, scandoubler	Machine	R/W	\$41	65	0100 0001	joystick 1: Kempston 1 joystick 2: Sinclair 2 joystick 1 bit 2: 0 refresh rate: 50 Hz joystick 2 bit 2: 0 scandoubler: yes			
\$06	Peripheral 2 Hotkeys, PS/2, sound chip	Machine	R/W	\$A9	169	1010 1001	F8 hotkey (CPU speed): yes BEEP to internal speaker only: no F3 hotkey (50/60 Hz): yes DivMMC NMI by DRIVE button: no Multiface NMI by M1 button: yes PS/2: keyboard sound chip: AY			
\$07	CPU Speed CPU speed	Machine	R/W*	\$33	51	0011 0011	actual: 28 MHz requested: 28 MHz			
\$08	Peripheral 3 128K paging, contention, sound	Machine	R/W*	\$0E	22	1101 1110	\$7FFD paging unlocked: yes contention disabled: yes AY stereo: ABC internal speaker: yes 8-bit DACs: yes port \$FF read (Timex): yes TurboSound: yes Issue 2 keyboard: no			
\$09	Peripheral 4 AY mono, sprites, scanlines	Machine	R/W*	\$00	0	0000 0000	AY 2 mono: no AY 1 mono: no AY 0 mono: no NR \$34 locked to port \$303B: no HDMI audio disabled: no scanlines: none			
\$0A	Peripheral 5 Multiface, SD, DivMMC, mouse	Machine	R/W	\$01	1	0000 0001	Multiface: +3 swap SD cards: no DivMMC automap: no swap mouse buttons: no mouse DPI: default			
\$0B	Joystick Joystick I/O mode	Machine	R/W	\$01	1	0000 0001	I/O mode: no mode: bit-bang iomode 0: 1			
\$0E	Core Version Sub Core version – sub-minor	Machine	R	\$03	3	0000 0011	sub-minor: 3			
\$0F	Board Issue Board revision	Machine	R	\$00	0	0000 0000	revision: 0			
\$10	Core Boot Core ID and buttons	Machine	R/W*	\$04	4	0000 0100	core ID: 1 DRIVE button: no NMI button (M1): no			
\$11	Video Timing Video timing (VGA 0–6, HDMI)	Video	R/W*	\$03	3	0000 0011	mode: VGA 3			
\$12	L2 Active Bank Layer 2 active (displayed) bank (16K)	Layer 2	R/W	\$16	22	0001 0110	16K bank: 22			
\$13	L2 Shadow Bank Layer 2 shadow bank (16K)	Layer 2	R/W	\$0C	12	0000 1100	16K bank: 12			
\$14	Transparency Col Global transparency colour	Video	R/W	\$E3	227	1110 0011	RRRRGGGBB: R7 G0 B3			
\$15	Spr/Layer Prior Sprites, LoRes and layer priority	Video	R/W	\$41	65	0100 0001	LoRes: no sprite 0 on top: yes sprite clipping in border: no layer priority: SLU sprites over border: no sprites visible: yes			
\$16	L2 X Scroll LSB Layer 2 X scroll (low 8 bits)	Layer 2	R/W	\$41	65	0100 0001	X: 65			
\$17	L2 Y Scroll Layer 2 Y scroll	Layer 2	R/W	\$01	1	0000 0001	Y: 1			
\$18	Clip L2 Layer 2 clip window: successive writes X1, X2, Y1, Y2	Layer 2	R/W*	\$10	16	0001 0000	value at index (NR \$1C): 16			
\$19	Clip Sprite Sprite clip window: successive writes X1, X2, Y1, Y2	Sprites	R/W*	\$00	0	0000 0000	value at index (NR \$1C): 0			
\$1A	Clip ULA ULA/LoRes clip window: successive writes X1, X2, Y1, Y2	Video	R/W*	\$FF	255	1111 1111	value at index (NR \$1C): 255			
\$1B	Clip Tilemap Tilemap clip window: successive writes X1, X2, Y1, Y2	Tilemap	R/W*	\$10	16	0001 0000	value at index (NR \$1C): 16			
\$1C	Clip Control Clip window indexes	Video	R/W*	\$00	0	0000 0000	Tilemap index: 0 ULA index: 0 sprite index: 0 Layer 2 index: 0			
\$1E	Active Line MSB Active line (Copper counter), bit 8	Video	R	\$00	0	0000 0000	line bit 8: 0			
\$1F	Active Line LSB Active line (Copper counter), bits 7:0	Video	R	\$F6	246	1111 0110	line: 246			
\$20	Int Generate Interrupt status / generation	Machine	R/W*	\$40	64	0100 0000	line: no ULA: yes CTC 3: no CTC 2: no CTC 1: no CTC 0: no			
\$22	Line IRQ Enable Line interrupt control	Video	R/W*	\$00	0	0000 0000	/INT active: no ULA interrupt disabled: no line interrupt: no line bit 8: 0			
\$23	Line IRQ Line Line interrupt line, bits 7:0	Video	R/W	\$00	0	0000 0000	line: 0			
\$26	ULA X Scroll ULA X scroll	Video	R/W	\$00	0	0000 0000	X: 0			

Figure 19 The NEXTREG tab: Next registers with values, bit breakdown and highlighted changes.

A table of the Next registers with name, group, access, the value in hex, decimal and binary, and a **bit breakdown** into named fields (for example the CPU speed for NR \$07, the layer order for NR \$15). Toolbar:

- **Search (number, name, bit)...** – filters by register number, name or bit field name.
- **All groups** – limits the table to one group: Machine, Video, Layer 2, Sprites, Tilemap, Copper, Palettes, MMU, Sound, Other.
- **all 256** – also shows numbers the Next does not define.
- **changed only** – only registers changed since the last refresh.

Registers that changed since the last refresh have a highlighted row and the changed bits are emphasised – just step and watch what changes. The count is on the right (84 shown · 3 changed). The Access column: R/W read and write, R read only, W write only – for write-only registers the value shown is the last write (or 0 if the emulator does not keep it); an asterisk and a tooltip warn when the value read differs from the value written.

10 Saved states

A saved state is a snapshot of the whole machine – memory, registers, hardware. It is useful when you want to return to the same place in a game repeatedly (before the boss fight, before the bug) without playing your way there every time.

10.1 Saving and restoring

- **MB Maniax: Save State...** – the floppy disk button on the debug toolbar or the command from the palette. An input box *Save Emulator State* opens at the top; enter a name and press **Enter**. The same name overwrites an older state (the extension asks first).
- **MB Maniax: Restore State...** – the clock button. Pick a state from the list; the machine returns to it and **stays stopped** (the Call Stack shows the reason *state restored*). Continue with **F5**.

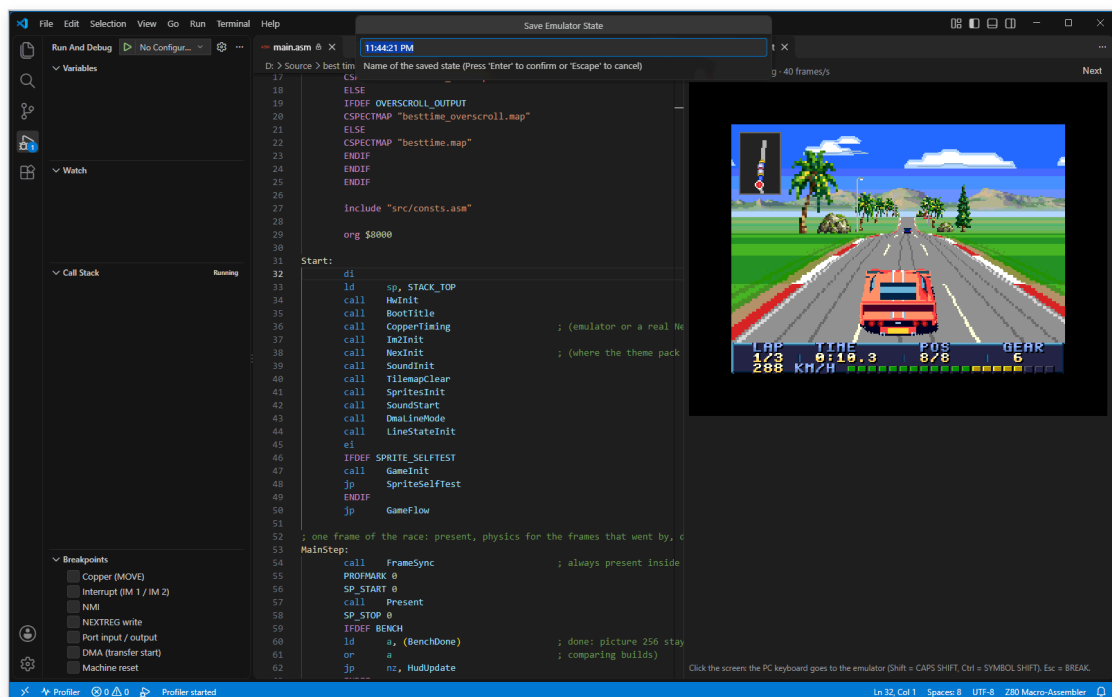


Figure 20 Saving a state: entering the name.

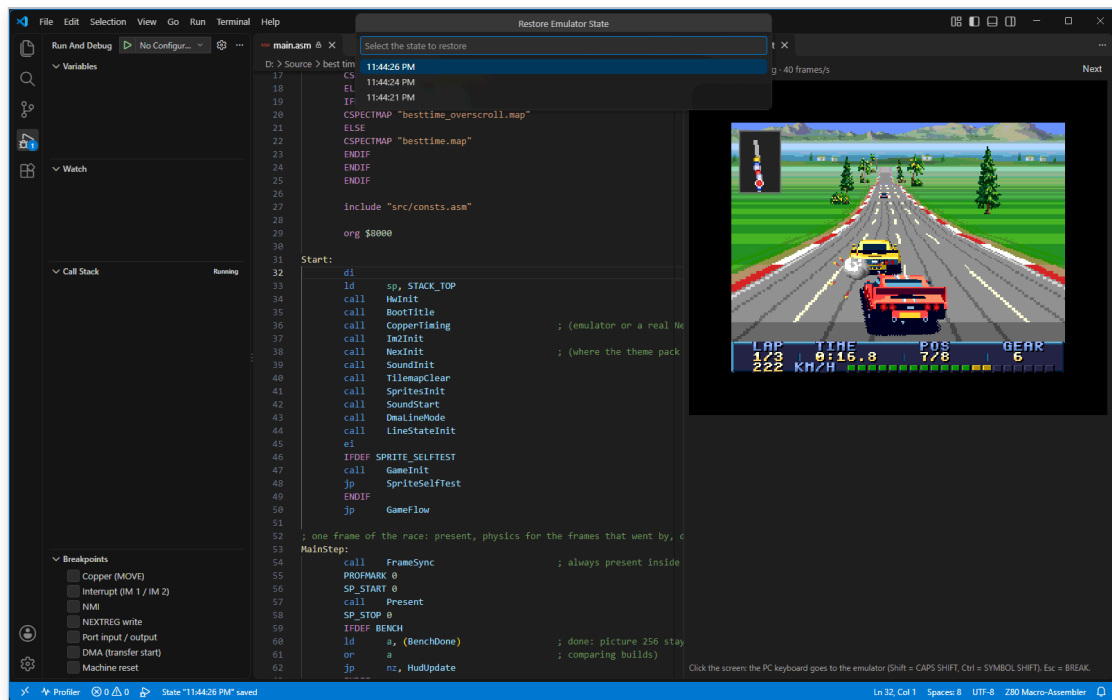


Figure 21 Restoring a state: choosing from the saved states.

States belong to the current debug session (until **Stop**) – they survive a Restart. There can be at most **8** of them per session. Restoring does not change breakpoints or the profiler.

10.2 States are saved at the end of a frame

The emulator can save a state only at a frame boundary. A running program is always at one, so saving while running happens immediately. If the machine is stopped in the middle of a frame (breakpoint, step), the extension asks: *The machine is stopped in the middle of a frame. A state can only be saved at the end of a frame. Run to the end of the current frame and save the state there?* The **Run to End of Frame and Save** button runs the program to the end of the frame (breakpoints are ignored meanwhile), stops (reason *frame boundary*) and saves the state.

10.3 Export to and import from a file

MB Maniax: Export State to File... saves the current state to a `.mbxstate` file, **MB Maniax: Import State from File...** loads it – also in another debug session. Both commands are available only in the Command Palette and only while debugging. A file can be loaded **only by the same version of the extension** and for the same machine type; after updating the extension, old files cannot be loaded.

11 Profiler

The profiler measures where the program spends its time. It adds up the T-states (CPU clock cycles) spent on every instruction and shows the result right in the source: for each line how much it cost, and with a colour how expensive it is compared to the others.

11.1 Starting

1. While debugging, click the **Start Profiler** button (pulse icon) on the debug toolbar, or run **MB Maniax: Start Profiler** from the Command Palette. **Profiler** appears in the Status Bar at the bottom left.
2. Let the program run through the part you want to measure (for example a few seconds of the game).
3. Stop the program (F6, a breakpoint). Every time it stops, the result is shown in the open editors and the Status Bar shows the total: Profiler: 3,512,000 T.

11.2 Reading the results

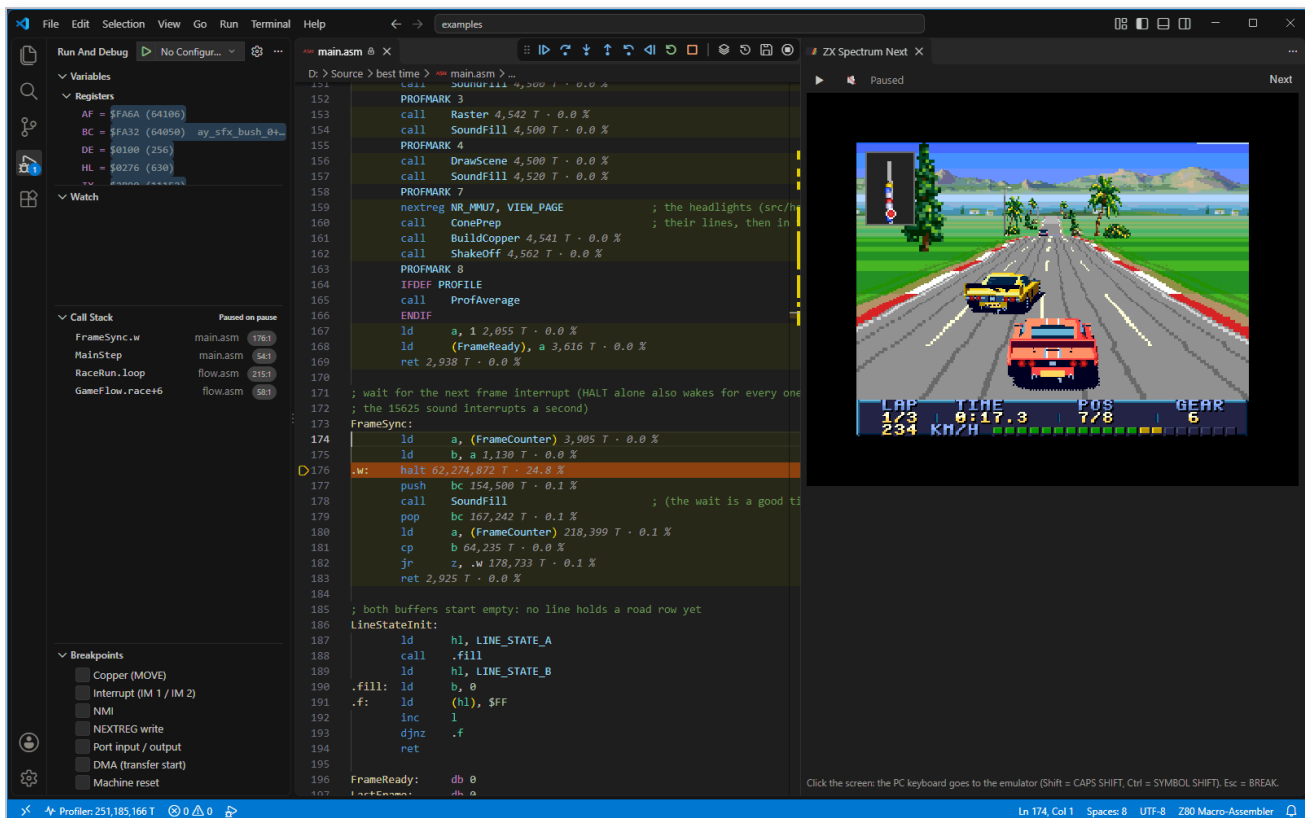


Figure 22 Profiler results in the source: T-states and share after the end of lines, background colour by the line's cost, marks in the scroll bar.

- **Grey text after the end of a line**, e.g. 1,234 T · 12.3 %: the line used 1,234 T-states, which is 12.3 % of all T-states during the measurement (including time outside your source). Hovering over the line repeats it in a tooltip.
- **Line background** from light yellow to red: the line's cost relative to the **most expensive line** (8 levels). The most expensive lines are the reddest – look at them first.
- **Marks in the scroll bar** on the right (overview ruler): the same colours, so you can find the expensive places even in a long file without scrolling.

A line executed many times is expensive even if the instruction itself is fast – the profiler shows the **total** time, not the time of one pass. Everything the CPU executes counts: waiting in HALT (every repetition) and accepting an interrupt (attributed to the instruction before which it came). Time when the CPU is held by DMA and time while paused are not counted. Instructions in a macro are attributed to the line the macro is invoked from.

11.3 Most expensive functions

MB Maniax: Show Profiler Results – from the Command Palette, or by clicking **Profiler** in the Status Bar – opens the list *Most expensive functions*, sorted from the most expensive. A function is the code from one global label to the next global label (local labels with a dot belong to their parent). Each shows the total T-states with the share, and the file and line; select one to go there. The item *Outside the source code (ROM, code without SLD)* at the end adds up the time spent outside your source.

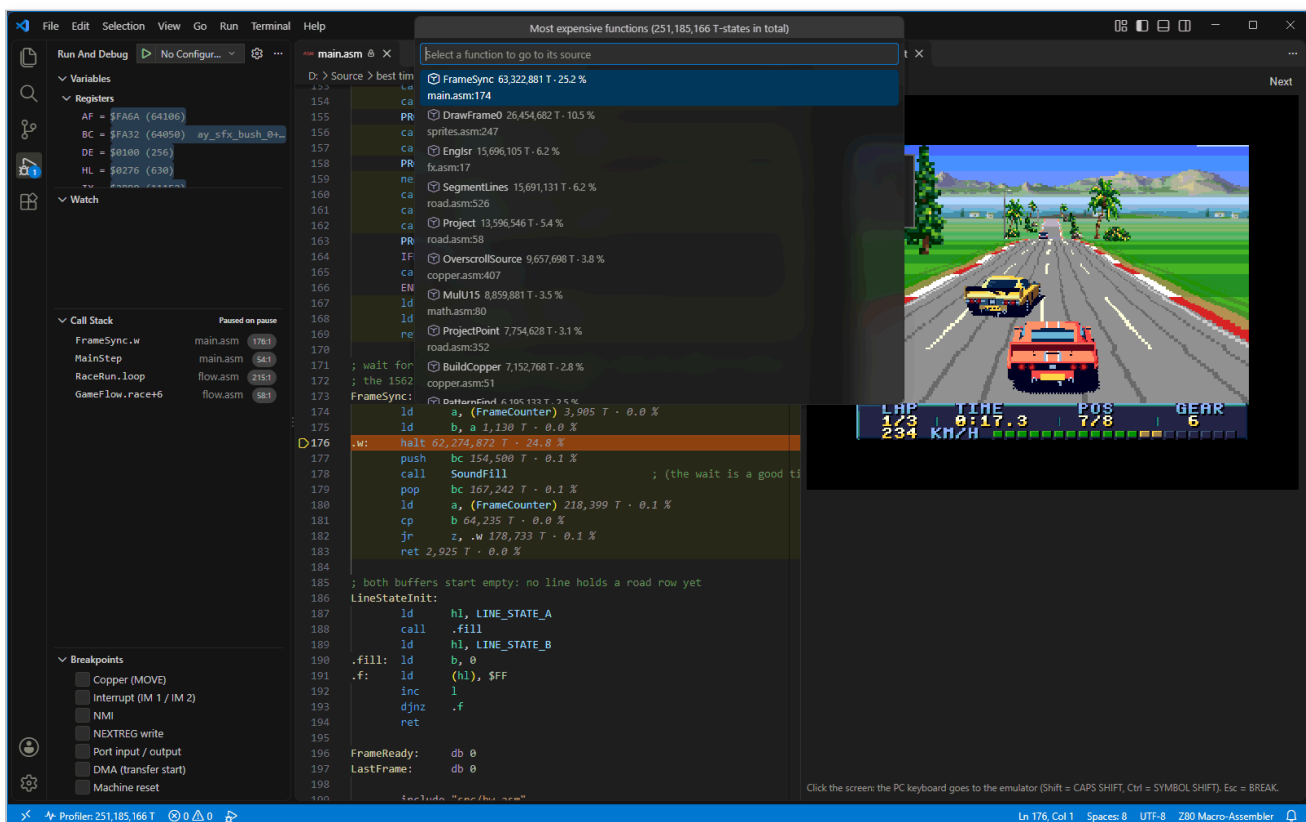


Figure 23 The list of the most expensive functions.

11.4 Live updates, stopping, clearing

- **MB Maniax: Toggle Live Profiler Updates** – results are refreshed while running too, every second. Handy when you want to see the numbers change while playing.
- **MB Maniax: Stop Profiler** (button on the debug toolbar) – the measurement ends; the last results stay in the editor (even after debugging ends). A notification shows the measured total with a **Show Results** button; the total also stays in the status bar, and clicking it opens the list of the most expensive functions again at any time.
- **MB Maniax: Clear Profiler Data** – starts counting from zero and removes the colouring. The profiler keeps running.

11.5 Tips

- Measure only what you are interested in: run the program, get to the place, run **Clear Profiler Data**, let it run and stop.
- A main loop waiting on HALT shows a large number right at the HALT – that is the free time of the frame, not a bug. The smaller HALT's share, the less headroom you have.
- Restart and reloading the program reset the counters; the profiler stays on.
- The profiler counts by physical address (page + offset), so paged code does not get mixed up even when it runs at the same address.

12 Step Back

Step Back moves the machine back by one instruction – repeatedly if you like. For this the emulator remembers the last **10 seconds** of execution. It is useful when the program stops on a bug and you want to see how it got there.

12.1 Enabling it

Step Back is off by default because recording the instruction trace slows emulation down – by about a quarter on a Next at 28 MHz. Enable it in `launch.json`:

```
"rewind": true
```

and start debugging again. Without it, the Step Back button reports *Step Back is turned off – set "rewind": true in launch.json (emulation then runs slower)*.

12.2 Using it

While the program is stopped, click **Step Back** (an arrow curving backwards) on the debug toolbar. Each click goes back one instruction; registers, memory and the picture correspond to the state before it. You can then step forward again. The **Reverse** button (run backwards to a breakpoint) is not available yet – it reports *Reverse Continue is not available yet – use Step Back*.

TIP

If the slowdown bothers you only sometimes, keep two configurations in `launch.json` – one with `"rewind": true` and one without – and pick between them in the list next to the green button in Run and Debug.

13 Settings and launch.json

13.1 launch.json configuration attributes

A configuration of type "mbx-next" in `.vscode/launch.json`. While typing, VS Code suggests attribute names and their descriptions (`Ctrl` + `Space`).

Attribute	Default	Meaning
type	–	Always "mbx-next".
request	–	Always "launch".
name	–	Configuration name in the Run and Debug list.
program	(required)	Program to run (<code>.nex</code> , <code>.sna</code> , <code>.tap</code> , ...), e.g. <code>\${workspaceFolder}/game.nex</code> .
sld	–	Debug information file from sjasmplus (<code>--sld</code>). Without it, debugging is limited to the disassembly and the extension warns about it.
sourceRoot	SLD folder	Folder that source paths in the SLD are relative to (when the SLD has no full paths).
stopOnEntry	false	Stop at the program's first instruction.
machine	"next"	Emulated machine: <code>next</code> , <code>48k</code> , <code>128k</code> , <code>plus3</code> , <code>pentagon</code> . Machines other than the Next need an SD card with ROMs (<code>sdImage</code>).
sdImage	–	SD card image (<code>.img</code>). Without it, the program sees its own folder through esxDOS.
esxdosRoot	program folder	Folder the program sees through esxDOS when there is no SD card.
rewind	false	Step Back – the emulator remembers the last 10 seconds; emulation is slower (12).
magicBreakpoint	true	Stop on the CSpect BREAK instruction (<code>FD 00</code> , <code>DD 01</code>) and on the ZEsarUX <code>ED FF</code> . When off, <code>DD 01</code> behaves as <code>LD BC,nn</code> .
wpmem	true	Watchpoints from WPMEM comments.
logpoints	true	Logpoints from LOGPOINT comments.
assertions	true	Assertions from ASSERTION comments.
preLaunchTask	–	(VS Code attribute) A task from tasks.json run before debugging – typically the build.

Paths may contain VS Code variables: `${workspaceFolder}` (the open folder), `${fileDirname}` (the folder of the open file) and others. A complete example:

```
{
  "type": "mbx-next",
  "request": "launch",
  "name": "Debug the game (with Step Back)",
  "program": "${workspaceFolder}/build/game.nex",
  "sld": "${workspaceFolder}/build/game.sld",
  "stopOnEntry": false,
  "rewind": true,
  "esxdosRoot": "${workspaceFolder}/data",
  "preLaunchTask": "sjasmplus"
}
```

13.2 Extension settings

Settings (**Ctrl** + **,** , search for `mbxNext`):

Setting	Default	Meaning
<code>mbxNext.language</code>	auto	Language of the extension (messages, emulator screen, Next panel), see 2.5 .
<code>mbxNext.serverPath</code>	empty	Path to the <code>mbx-next-server</code> debug server; empty = the server bundled with the extension. Change only for development or when told to by the author.

13.3 Command overview

Command	Where
MB Maniax Next: Show Emulator Screen	palette
MB Maniax: Next Panel (Sprites, Palettes, ...)	debug toolbar, palette
MB Maniax: Save State...	debug toolbar, palette
MB Maniax: Restore State...	debug toolbar, palette
MB Maniax: Export State to File...	palette (while debugging)
MB Maniax: Import State from File...	palette (while debugging)
MB Maniax: Start Profiler	debug toolbar, palette
MB Maniax: Stop Profiler	debug toolbar, palette
MB Maniax: Show Profiler Results	palette, Status Bar
MB Maniax: Toggle Live Profiler Updates	palette (while debugging)
MB Maniax: Clear Profiler Data	palette (while debugging)

14 Troubleshooting

Cannot set a breakpoint – no red dot appears when hovering over the margin

VS Code allows breakpoints only in files whose **language** some debugger supports. This extension reports them for the languages `z80-asm`, `asm-collection`, `z80-macroasm`, `asm`, `asm-x86-nasm`, `nas`, `masm`, `c6510` and `plaintext` (Plain Text). If you have an extension for another assembler installed that claims `.asm` under a different language, breakpoints will not work.

1. Look at the bottom right of the Status Bar – it shows the language of the open file (e.g. **Plain Text**, **Z80 Assembly**, **ARM...**).
2. Click it. A language picker opens at the top; choose **Configure File Association for '.asm'...** and select a Z80 language (depending on the installed extension e.g. *Z80 Assembly*, *Z80 Macro-Assembler*) or **Plain Text**.

VS Code writes this into the `files.associations` setting. You can also write it by hand (**Preferences: Open User Settings (JSON)** from the Command Palette):

```
"files.associations": {
  "*.asm": "z80-macroasm",
  "*.i.asm": "z80-macroasm"
}
```

The value must be a language identifier from the list above that is available to you (`plaintext` always is).

The breakpoint is grey and hollow

The breakpoint is not verified. Hover over it – the tooltip says why: *SLD from sjasmpplus is missing ("sld" attribute)* → add `"sld"` to `launch.json`; *No instruction on this line or below it (according to SLD)* → the line is not in the SLD (it is a comment or data, or the file was not built with this SLD); a condition error → fix the condition (6).

The program does not stop at a breakpoint

- The SLD is missing or out of date: build with `--sld=...` `--fullpath` and check that `launch.json` points to the same file. When starting without an SLD, the extension warns *"sld" is missing in launch.json...* or *Cannot load SLD....*
- The paths in the SLD do not match the files in the editor: use `--fullpath`, or set `sourceRoot`.
- You are building a different copy of the file than the one open in the editor (another folder).
- Paged code: the breakpoint stops only when the page is mapped in (5.1). Check which page is in the slot under Variables › Machine (MMU, time).
- The breakpoint is disabled (unticked in the Breakpoints section), or all breakpoints are deactivated (the **Toggle Active Breakpoints** button in the section header).
- The program was not rebuilt after a change – `preLaunchTask` is missing from `launch.json`, or the build failed (the **Problems** and **Terminal** tabs).

Debugging does not start

- *Program not found: ...* – the `"program"` path is wrong, or the program has not been built yet.
- *Debug server not found (...)* – the extension is damaged or `mbxNext.serverPath` is set wrong; clear the setting, or reinstall the extension.
- *The debug server is not responding / The debug server exited during startup* – try starting debugging again; details (server output) are usually in the Debug Console. An antivirus can also block the server

from starting.

- The `sjsmplus` task cannot be run – `sjsmplus` is not on the `PATH` (4.1).

Choppy sound, slow emulation

- Check *Running · N frames/s* in the screen bar. Below 50 the computer cannot keep up.
- Turn off Step Back (`"rewind": false`) – the instruction trace slows things down by about a quarter.
- Turn off **Live** in the Next panel and live profiler updates; close the Next panel when you do not need it.
- Logpoints printing thousands of times per second slow down the Debug Console – use a hit count (`%50`) or a condition.
- The screen panel redraws only while it is visible; hidden behind another tab it costs nothing.

The screen panel or the Next panel disappeared

The panels are editor tabs: look for them in another tab group, or open them again – **MB Maniax Next: Show Emulator Screen** and **MB Maniax: Next Panel (Sprites, Palettes, ...)**.

The keyboard does not work in the emulator / shortcuts do not work in VS Code

Keys go to the emulator only while the screen has the focus (click into it). Conversely, VS Code shortcuts (`F5` , `F10`) work only when the screen does not have the focus – click into the editor.

No sound

Click the screen or the  button – sound may start only after a click. Check that the program is running (it is silent while paused) and that the icon shows .

The extension speaks a different language than I want

Set `mbxNext.language` (2.5) and start debugging again. Command names and the descriptions in Settings follow the VS Code display language (the **Configure Display Language** command), not this setting.

The old version still runs after an update

Stop debugging and reload the window: `Ctrl` + `Shift` + `P` → **Developer: Reload Window** (2.3).

A saved state file cannot be imported

A `.mbxstate` file can be loaded only by the same version of the extension and for the same machine type (`machine`).

15 Keyboard shortcuts

Default VS Code keyboard shortcuts on Windows that are useful while debugging. Customise them in **File > Preferences > Keyboard Shortcuts** (**Ctrl** + **K** **Ctrl** + **S**).

Shortcut	Action
F5	Start debugging / Continue
F6	Pause
Shift + F5	Stop debugging
Ctrl + Shift + F5	Restart
F10	Step Over
F11	Step Into
Shift + F11	Step Out
F9	Toggle breakpoint on the cursor line
Ctrl + Shift + D	Run and Debug side bar
Ctrl + Shift + Y	Debug Console
Ctrl + Shift + P , F1	Command Palette
Ctrl + Shift + B	Run the default build task (sjasmplus)
Ctrl + Shift + M	Problems (build errors)
Ctrl + `	Terminal
Ctrl + J	Show / hide the bottom panel
Ctrl + Shift + E	Explorer (files)
Ctrl + Shift + X	Extensions
Ctrl + ,	Settings
Ctrl + P	Quick open a file by name
Ctrl + G	Go to line
F12	Go to definition (if your assembler extension provides it)

In the emulator screen (when it has the focus): **Shift** = CAPS SHIFT, **Ctrl** = SYMBOL SHIFT, **Esc** = BREAK.