

ZX SPECTRUM NEXT · NATIVE Z80 ASSEMBLER

```
NextForge 1.2 by MB Maniax

; Hello, NextForge
    org $8000
Start:
    ld hl,0
    ld de,16384
    ld bc,6144
    ret

HELLO.ASM  Ln 1 Col 1  Lines 7
```

Illustrative editor diagram

NEXTFORGE

User manual

Editor · assembler · monitor and debugger · unit tests

Contents

PART I · EDITOR AND ASSEMBLER

1. Introduction and first launch
2. Your first binary file
3. Editor and controls
4. Files on the SD card
5. Blocks, search, and labels
6. Assembly and the result report
7. Syntax, symbols, and expressions
8. Directives and macros
9. Include projects and output
10. Supported instructions and limits
11. Troubleshooting
12. Shortcut reference and printing

PART II · MONITOR AND DEBUGGER

13. Meet the debugger
14. A first debugging session

15. The debugger screen
16. Menus, context menus and help
17. Stepping and running
18. Breakpoints, conditions and watches
19. Monitor: memory, registers and editing
20. Source code and labels
21. How the debugger runs a program
22. Limits and troubleshooting
23. Debugger key reference

PART III · UNIT TESTS

24. Meet unit tests
25. Your first tests step by step
26. Test directives
27. How tests run
28. Tips and limits

MANUAL VERSION

Describes current NextForge 1.2 features. Planned features are explicitly identified.

Editor and assembler

The first part covers the editor, files on the SD card, the assembler and its directives — from the first launch to a finished binary or NEX file.

Chapters 1–12. Debugging an assembled program is covered in Part II · Monitor and debugger, automatic tests in Part III · Unit tests.

1. Introduction and first launch

NextForge is an editor and native assembler for the ZX Spectrum Next. Write source on the Next itself, save files to SD, and assemble them into a binary file. This manual describes development version **1.2** as of 22 October 2026.

NEW IN VERSION 1.2

- A T-states column in the editor: **EXT** → **K** or **Assemble** → **T-states** shows at the end of instruction lines how many T-states they take, from the last assembly. It never covers text or comments (chapter 3).

NEW IN VERSION 1.1

- The debugger has a menu bar, right-click context menus and its own searchable help.
- Programs may start at `$2000` (so `ORG $8000` works); banked `MMU` / `SAVENEX` output can be debugged with the `SAVENEX` entry and stack.
- Stepping simulates `DI/EI/HALT`, ports, `NEXTREG` and paging; `BREAK` stops a fast run.
- Conditions on any register, pair or memory with `= <> < > <= >=`, also without breakpoints.
- Longer code, stack and memory views, a memory preview at PC, red-dot breakpoints and the new J and V keys.
- The program's screen in the debugger: F shows it at any time, L during slow tracing and fast runs always — `ULA`, `LoRes`, `Timex`, `Layer 2`, `tilemap`, `sprites` and `palettes`. Fast runs get their pages in place, so the program's own paging works too.
- A calls the routine at the cursor; the context menu offers Run from here.
- Shortcuts for frequent actions: **EXT** → **A** assembles, **EXT** → **R** assembles and opens the debugger, **EXT** → **D** opens the monitor and debugger.
- Unit tests: `TESTCASE` / `ENDTEST` blocks with `EXPECT` checks, taken in with `TESTCLUDE`; **EXT** → **T** assembles and runs them, the report window lists them with their T-states and the editor colours the results. Every test starts with the same memory, `STUB` replaces ROM and `esxDOS` calls and `MOCK` fakes a routine of your own (Part III).
- Breakpoints in the editor: **EXT** → **B** or the context menu turns the line red and the debugger takes it over; B in the debugger marks the source line.
- Expressions know the `SjASMPlus` operators with their precedence: shifts, comparisons, bitwise and logical operators and the remainder (chapter 7). `ASSERT` can combine conditions.
- A larger symbol table: names up to 64 characters, fast lookups, and more pages from the document pool when needed.
- Save fix: source and binary files are emptied before they are written. On some NextZXOS installations version 1.0 left old text at the end of a saved file and assembly reported `Save failed: Set binary length esx=19`.
- NextForge assembles its own source and produces the same NEX as `SjASMPlus`. This brought `DISP/ENT`, `SAVENEX BANK`, `REPT` counters, `Global.local` references, longer macros with several parameters and include search in parent directories (chapter 7).
- Conditional assembly with `IF/IFN/ELSEIF/ELSE/ENDIF` and `IFDEF/IFNDEF` with `DEFINE` (chapter 8).
- `STRUCT`, `DZ`, `DC`, `DD` and `D24`; local labels in macros (chapter 8).
- The debugger counts T-states and tests check them with `EXPECT CYCLES`; the editor can find and replace.
- The manual is split into two parts and illustrated with screenshots.

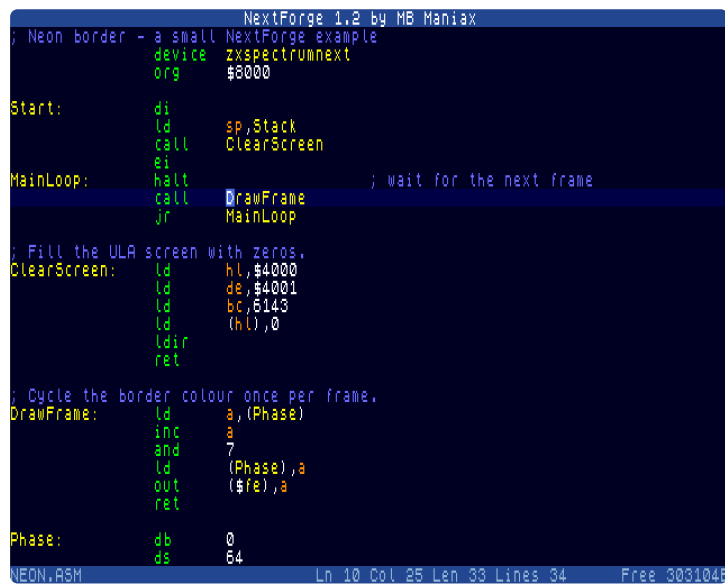
The debugger is described in detail in Part II.

What you need

A ZX Spectrum Next running NextZXOS with an SD card, or CSpect with a suitable system and SD image. The application uses an 80-column text display. A mouse helps with menus and dialogs; the main workflow is also available from the keyboard.

NextForge runs on a Next with 1 MB or 2 MB of memory. `NEXTFORGE.NEX` is 176.5 KB (11 banks of 16 KB). While running, NextForge permanently occupies 288 KB and reserves another 320 KB (1 MB Next) or 960 KB (2 MB) for documents, assembler output and the debugger. Chapter 21 lists the pages and what a debugged program should avoid.

1. Copy `NEXTFORGE.NEX` to the SD card.
2. Select and launch it in the NextZXOS browser.
3. The application opens with an empty editor. Click the top bar or use **EXT** → **0** to open the menu.
4. Open source using **File** → **Load...**, or start typing in the empty editor.



```
NextForge 1.2 by MB Maniax
; Neon border - a small NextForge example
device zxspectrumnext
org $8000

Start:
    di
    ld sp,Stack
    call ClearScreen
    ei
MainLoop:
    halt ; wait for the next frame
    call DrawFrame
    jr MainLoop

; Fill the ULA screen with zeros.
ClearScreen:
    ld hl,$4000
    ld de,$4001
    ld bc,6143
    ld (hl),0
    ldir
    ret

; Cycle the border colour once per frame.
DrawFrame:
    ld a,(Phase)
    inc a
    and 7
    ld (Phase),a
    out ($fe),a
    ret

Phase:
    db 0
    ds 64

NEON.ASM Ln 10 Col 25 Len 33 Lines 34 Free 303104B
```

The editor with an open source: title and menus at the top, syntax-highlighted text, the status bar at the bottom.

MONITOR AND DEBUGGER

The **Debug** menu or **EXT** → **D** opens the monitor and debugger: the program paused before its first instruction, stepping, breakpoints with conditions, memory and registers. See Part II of this manual.

Reading this manual

Menu names match the English text in the application. **CS** means CAPS SHIFT, and **SS** means SYMBOL SHIFT. **EXT** → **key** means press CS + SS, release both, then press the indicated key. The CZE/ENG switch changes the language of this page; your browser remembers the choice.

KEYS / MENU	ACTION
EXT → 0	Open the menu bar.
Help → Help... / EXT → H	Open searchable help.

2. Your first binary file

Type this small example into the editor and assemble it. It sets the origin, emits a few instructions, and selects an output name. It does not draw anything on screen.

```
org $8000

Start:
    ld hl,0
    ld de,16384
    ld bc,6144
    ret

End:
    savebin "hello.bin",$8000,End-$8000
```

1. Type the source. Put `Start:` and `End:` on separate lines; instructions may be indented.
2. Choose **File** → **Save As...** and enter `hello.asm`. Select a directory and confirm it with SPACE.
3. Choose **Assemble** → **Assemble** or press **EXT** → **A**. Select `hello.asm` in the main-source picker and press ENTER.
4. A successful report shows **10 bytes** and the path to `hello.bin`. The output is saved beside the source.
5. Close the report with ENTER or BREAK. Load the resulting binary at `$8000`.



Choosing the main source before assembly (here with three open files).

LABELS ARE SUPPORTED

You do not need to remove labels to assemble a program. Keep the same letter case in definitions and references: `Start` and `START` are different symbols.

FRESHLY TYPED SOURCE

You can assemble an editor buffer without loading it from SD first. Saving beforehand gives you a backup and a clear project directory. An unnamed source defaults to `untitled.bin` unless `savebin` selects another name.

KEYS / MENU	ACTION
SPACE	Confirm the save directory.
ENTER	Confirm the main source / close the report.
CS + SPACE (BREAK)	Cancel a dialog / close the report.

3. Editor and controls

The top bar contains the application title and menus. The main area shows syntax-highlighted source. The bottom status bar shows the filename, line **Ln**, column **Col**, line length **Len**, line count **Lines**, and free memory **Free**.

Editor lines hold at most **80 characters**. When a file opens, a longer comment wraps onto further lines under its semicolon (from column 0 when the semicolon is right of column 50), so nothing is lost and the file still assembles:

```
PERIPHERAL_3_NR_08 equ $08 ; ABC/ACB Stereo, Internal Speaker, SpecDrum,  
                           ; Timex Video Modes, Turbo Sound Next, RAM  
                           ; contention and [un]lock 128k paging.
```

A longer line of code, such as a long DB string, is cut; do not save such a file from the editor. Text memory is shared between open files. Up to eight files can be open; available text space depends on the machine's memory.

```
NextForge 1.2 by MB Maniax  
; Neon border - a small NextForge example  
device zxspectrumnext  
org $8000  
  
Start: di  
      ld sp,Stack  
      call ClearScreen  
      ei  
MainLoop: halt ; wait for the next frame  
          call DrawFrame  
          jr MainLoop  
  
; Fill the ULA screen with zeros.  
ClearScreen: ld hl,$4000  
            ld de,$4001  
            ld bc,6143  
            ld (hl),0  
            ldir  
            ret  
  
; Cycle the border colour once per frame.  
DrawFrame: ld a,(Phase)  
          inc a  
          and 7  
          ld (Phase),a  
          out ($fe),a  
          ret  
  
Phase: db 0  
       ds 64  
  
NEON.ASM Ln 10 Col 25 Len 33 Lines 34 Free 303104B
```

The editor screen with the sample program NEON.ASM.

- | | |
|--|--------------------------------|
| ① Application title; the menu bar on mouse hover or EXT → 0. | ② A comment. |
| ③ A label. | ④ An instruction or directive. |
| ⑤ Operands: registers, numbers and symbols have their own colours. | ⑥ The current line and cursor. |

7 The active file name.

9 Free text memory.

8 Line, column, line length and line count.

Navigation and editing

KEYS	ACTION
CS + 5 / 6 / 7 / 8	Left / down / up / right.
CS + SS + 7 / 6	Page up / down; hold all keys together.
SS + Q / E	Start / end of line.
ENTER	Insert a new line.
CS + 0	Delete the character before the cursor.
CS + 9	Delete the character under the cursor.
CS + 3	Tab / indent an assembly line.
SS + W	Format the current assembly line.
CS + SPACE	BREAK: close or cancel a dialog.
Mouse wheel	Scroll editor text.

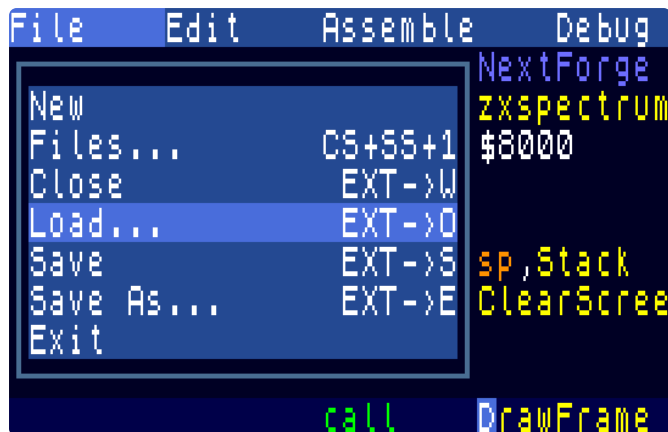
EMULATOR KEYBOARD

Shortcuts use Spectrum keyboard names. On a PC, CAPS SHIFT and SYMBOL SHIFT depend on your emulator's keyboard mapping; use the keys that send those Spectrum signals.

Menus

Open the menu with **EXT** → **0** or a mouse click on the top bar. Arrows select menus and items, ENTER confirms, and BREAK closes. The mouse can select menu items and files in dialogs.

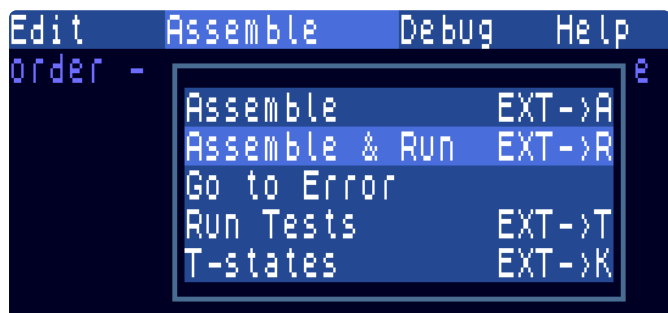
KEYS / MENU	ACTION
EXT → 0	Open the menu bar.
CS + 5 / 8	Previous / next menu.
CS + 7 / 6	Previous / next item.
ENTER	Run the selected item.
CS + SPACE (BREAK)	Close the menu.



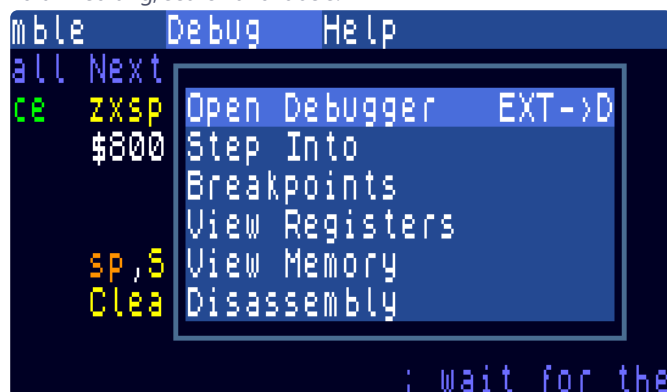
File — files and exit.



Edit — editing, search and labels.



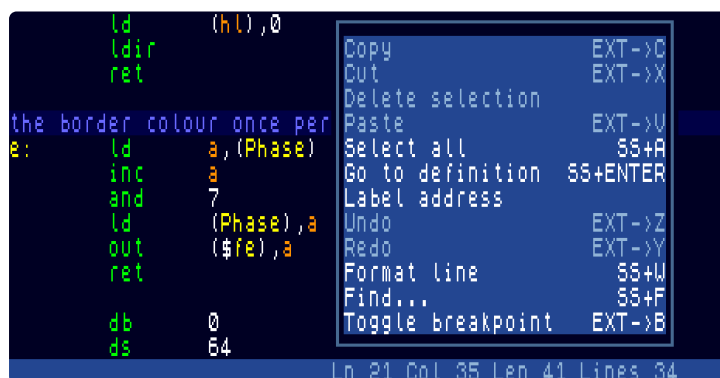
Assemble — assemble (EXT → A), assemble and debug (EXT → R), go to error, unit tests (EXT → T, Part III).



Debug — monitor and debugger (EXT → D, Part II).

Mouse and context menu

The left button places the cursor; dragging selects text — even part of a line. The wheel scrolls the source. The right button opens a context menu for the place under the pointer; items that are not available right now are dimmed (for example Copy without a selection or Undo without history). **Label address** shows the address of the label under the cursor from the last successful assembly. **Toggle breakpoint** marks the line as a breakpoint (see below).



The context menu over source text.

Breakpoints in the editor

EXT → B or **Toggle breakpoint** in the context menu marks the line under the cursor as a breakpoint; repeat it to remove the mark. A marked line has a dark red background (lighter on the cursor line) and keeps its syntax colours. The mark does not change the text: the file does not become modified and no undo step is used.

```

        call    ClearScreen
        ei
MainLoop: halt                ; wait for the next frame
        call    DrawFrame
        jr      MainLoop

; Fill the ULA screen with zeros.
ClearScreen: ld    hl,$4000
             ld    de,$4001
             ld    bc,6143
             ld    (hl),0

```

Breakpoint marks: a dark red line and the lighter cursor line.

A mark travels with its line when lines are inserted or deleted; ENTER at the start of a line moves it down together with the text, and joining lines with Backspace keeps the marks of both. Pasted text and clipboard blocks arrive without marks. When the debugger opens, the marks become breakpoints — see chapter 18. Marks are not saved to the file; they last while the file is open.

Instruction T-states

EXT → **K** or **Assemble** → **T-states** turns on a column that shows in grey, at the end of each line with an instruction, how many T-states it takes; repeat it to hide the column. The top bar shows the heading **T-states** above it. The values come from the last assembly, match the debugger and the unit tests, and ignore memory contention and the Next's wait states at 28 MHz.

NextForge 1.2 by MB Maniax		T-states
; Neon border - a small NextForge example		
	org \$8000	
Start:	di	4
	ld sp,Stack	10
	call ClearScreen	17
	ei	4
MainLoop:	halt ; wait for the next frame	4
	call DrawFrame	17
	jr MainLoop	12
; Fill the ULA screen with zeros.		
ClearScreen:	ld hl,\$4000	10
	ld de,\$4001	10
	ld bc,6143	10
	ld (hl),0	10
	ldir	16/21
	ret	10
; Cycle the border colour once per frame.		
DrawFrame:	ld a,(Phase)	13
	inc a	4
	and 7	7
	ld (Phase),a	13
	out (\$fe),a	11

The T-states column after an assembly: 16/21 for LDIR, and the comment on the HALT line stays whole.

- Conditional instructions show two numbers, for example 7/12: the smaller one when the branch is not taken, the larger one when it is (JR cc, DJNZ, CALL cc, RET cc). Block instructions such as LDIR show 16/21: the last round and each round before it. Loops are not added up.

- A macro line shows the total of the whole expansion.
- Data lines (DB, DW, DS, INCBIN...), labels and comments have no value.
- The column never covers text: a line whose text or comment reaches the value's place stays whole and shows no number. The number also hides while the cursor is in its place.
- After any change to the source the numbers disappear and the heading becomes T:EXT->A until you assemble again (EXT → A). An undo that brings the text back to the assembled state brings them back.
- Values exist only for a flat image (BIN, SAVEBIN). An assembly with DEVICE into a NEX has no source map, so the column stays empty.

Integrated searchable help

Open the function and shortcut reference using Help → Help... or EXT → H. Start typing, for example label, MMU, or SS+ENTER: the list immediately narrows to matches in descriptions or shortcuts. A match count is shown; an empty filter shows the full reference. The filter accepts up to 48 characters. Searching and scrolling update the window without flicker, and closing preserves the source and cursor position. Integrated help uses English text.

KEYS / MENU	ACTION
Help → Help... / EXT → H	Open help; EXT → H also closes it.
Typing text	Filter descriptions and shortcuts, ignoring letter case.
CS + 0	Remove the last filter character.
CS + 7 / 6	Scroll the list up / down.
CS + SS + 7 / 6	Previous / next page of matches.
SS + Q / E	First / last page of matches.
Mouse wheel	Scroll the list.
CS + SPACE (BREAK) / [Close]	Close help and return to the editor.

```

NEXTFORGE 1.2 - SEARCHABLE HELP [ Close ]

FUNCTION / CONTEXT                                SHORTCUT / MENU
Blocks: mark start / confirm end (whole lines)    CS+4
Blocks: select all lines                          SS+A
Blocks: copy selected text or lines                EXT->C
Blocks: cut selection or current line              EXT->X
Blocks: paste clipboard, including another file    EXT->U
Blocks: cancel selection                          BREAK / CS+SPACE

Filter: block
Matches: 6
Type to filter  UP/DOWN/PGUP/PGDN/WHEEL scroll  BREAK / EXT->H close

NEON.ASM                               Ln 10 Col 25 Len 33 Lines 34       Free 303104B
```

Help filtered by "block".

4. Files on the SD card

ITEM / SHORTCUT	PURPOSE
File → Load... / EXT → O	Open source through the NextZXOS browser.
File → Save / EXT → S	Save the active file using its name and directory.
File → Save As... / EXT → E	Enter a name, choose a directory, and save.
File → Files... / CS + SS + 1	Switch between open files.
File → Close / EXT → W	Close the active file; modified files offer a save choice.
File → New	Clear the current editor buffer.
File → Exit	Reset the entire editor in this version.

Dialog controls

KEYS / MENU	ACTION
CS + 7 / 6	Select an open file.
ENTER / click	Confirm a file or a Save As filename.
SPACE	Save in the selected browser directory.
CS + SPACE (BREAK)	Cancel selection or saving.
ENTER / S	Save changes when closing a file.
D	Discard unsaved changes when closing.

Save As

Enter the filename first and confirm with ENTER. A directory browser follows. Navigate to your destination and press **SPACE** to choose "save here". BREAK cancels the operation. Ordinary Save uses the directory recorded for that file.

```
out      ($fe),a
ret

Phase:   db      0
         ds      64

Save as: NEON.ASM
```

Save As: the name is typed in the status bar; ENTER then opens the directory browser.

Several open files

The **Files...** dialog lists files already open in the editor. Select with arrows or the mouse, then press ENTER or click. Close a file when all eight slots are occupied. Includes do not need to be open in the editor: the assembler reads them from SD.

```

ld      sp,Stack
call    ClearScreen
ei
halt    ; wait for the next frame
call    Open Files
jr      1. NEON.ASM
        2. COLOURS.ASM
screen wi 3. HELPERS.ASM
ld      ESC=Cancel ENTER=Open
ld      de,$4001
ld      bc,6143
ld      (hl),0
ldir
ret

```

The Files... dialog with three open files.

NEW AND EXIT

In this version, **New** clears the current buffer and **Exit** clears all open buffers. Exit does not yet return to NextZXOS. Save your work before using these items; use Close when you need the modified-file save prompt.

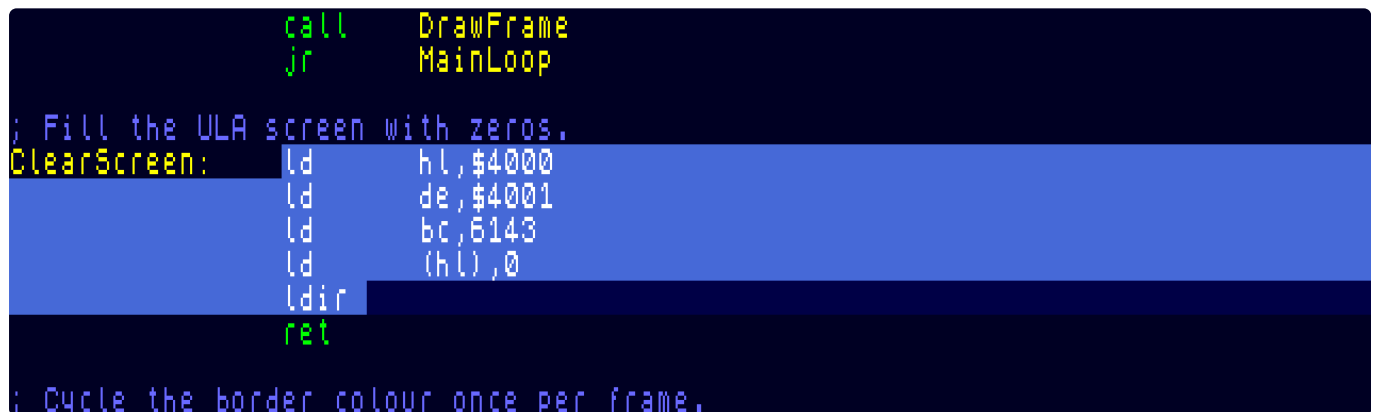
5. Blocks, search, and labels

Working with blocks

Block operations work on complete lines. Press **CS + 4** to set a block anchor, move the cursor to extend the selection, and press **CS + 4** again to fix its end. Copy or cut the selected block; this clears the selection. You can paste the clipboard into another open file. Cutting without a selection cuts the current line.

SHORTCUT	ACTION
CS + 4	Set the block start / fix its end.
SS + A	Select all lines.
EXT → C	Copy the block.
EXT → X	Cut the block or current line.
EXT → V	Paste the clipboard block.

Dragging with the mouse can also select part of a line (a character selection); copy and cut then work on the selected text only.



```
call DrawFrame
jr MainLoop

; Fill the ULA screen with zeros.
ClearScreen: ld hl,$4000
             ld de,$4001
             ld bc,6143
             ld (hl),0
             ldir
ret

; Cycle the border colour once per frame.
```

A mouse selection from an operand on the first row to an instruction on the last.

Searching

Choose **Edit → Find...** or press **SS + F**. Enter the search text and press ENTER. Search covers open files; another ENTER in the dialog finds the next occurrence. BREAK closes the dialog.

Replace: press **SS + F** again in the search dialog, type the new text and press ENTER. The matches in the current file from the cursor down are selected one by one; answer **Y** (replace), **N** (skip) or **A** (replace this and every later one), and BREAK stops. The status bar then shows how many were replaced. Each Y, and A as a whole, is one Undo step. A replacement that would make a line longer than 80 characters stops there.

KEYS / MENU	ACTION
Edit → Find... / SS + F	Open search across open files.
ENTER	Find the first / next occurrence.
SS + F in the dialog	Replace: type the new text, then Y / N / A per match.
CS + 0	Remove the last search character.

KEYS / MENU	ACTION
CS + SPACE (BREAK)	Close the search dialog.

```

ld bc,6143
ld (hl),0
ldir
ret

; Cycle the border colour once per frame.
DrawFrame: ld a,(Phase)
inc a
and 7
ld (Phase),a
out ($fe),a
ret

Phase: db 0
ds 64
Find: Phase

```

Find: the text is typed in the status bar; the cursor marks the match.

Label completion

Type at least **four characters** of a label name, for example `lores`, and leave the cursor immediately after them. Press **CAPS SHIFT + ENTER**. A **Label completion** popup lists matching labels from open files. It offers up to 32 matches, with eight visible at a time.

SHORTCUT	ACTION
Edit → Complete label / CS + ENTER	Open matching labels after typing at least four characters.
CS + 7 / CS + 6	Previous / next match (up / down).
CS + SS + 7 / 6	Page up / down through matches.
ENTER	Complete the selected label in the source and close the popup.
CS + SPACE (BREAK)	Close without inserting.

```

Start:      di
           ld      sp,Stack
           call    ClearScreen
           ei      Label completion
MainLoop:  ha      ClearScreen
           ca
           jr
           ca

; Fill the ULA screen
ClearScreen: ld
            ld
            ld      UP/DOWN, PGUP/PGDN, ENTER, ESC
            ld      (hl),0
            ldir
            ret

; Cycle the border colour once per frame.
DrawFrame: ld      a,(Phase)

```

Completion after typing `Clea` and CS + ENTER.

A short prefix produces `Type 4+ label chars`; no match produces `No label matches`. The popup is invoked by this shortcut rather than opening automatically after every typed character.

Go to definition

KEYS / MENU	ACTION
Edit → Go to definition / SS + ENTER	Jump to the definition of the name under the cursor in open files.

For a line such as `call lores_init`, place the cursor on `lores_init` and press **SYMBOL SHIFT + ENTER**. The editor locates its definition, switches to another open file when necessary, and moves the cursor to the label line. If it cannot find the definition, it displays `Definition not found`.

TWO DISTINCT SHORTCUTS

CS + ENTER completes a typed prefix from a popup of matches. **SS + ENTER** jumps to the implementation of an existing name. Open an include first if it is stored only on SD; these editor features inspect open files.

EDIT MENU ITEMS

Undo and Redo work with EXT → Z / Y and the Edit menu. Each open file keeps eight steps; a new edit discards its redo branch. History is reclaimed when text allocation needs memory. Cut, Copy, and Paste use the same operations as their keyboard shortcuts.

Left click positions the cursor; dragging selects characters across lines. Typing replaces the selection; Backspace and Delete remove it. Right click opens a context menu for copy, cut, delete, paste, select all, undo/redo, formatting and find. On a label it offers go to definition and a global address from the last successful assembly; edit the source and assemble again to refresh addresses. Right click outside a selection positions the cursor at the click. BREAK closes the menu or cancels the selection.

The cursor line has a subtle background highlight, preserving syntax colours; selected text takes precedence.

If go to definition cannot find the label in open files, it offers **Search includes**. SD search starts only after confirmation. A separate window shows the file being opened or searched, the line and file count. Nested includes

are followed; open files use their current unsaved text. BREAK or Cancel stops the search. A found file opens at the definition; if all slots are full, close another file first. Names are matched exactly; repeated local names are not resolved by scope, and assembler conditions are not evaluated.

6. Assembly and the result report

1. Choose **Assemble** → **Assemble** or press **EXT** → **A**.
2. Select the project's main source from open files. For an include-based project, choose the entry file containing **ORG** and **INCLUDE**, rather than an individual module.
3. The assembler runs two passes: the first builds symbols and the address layout; the second resolves references and emits bytes.
4. After successful assembly it saves the binary. A source error stops before saving begins.

KEYS / MENU	ACTION
CS + 7 / 6	Scroll the report up / down.
CS + SS + 7 / 6	Page up / down.
Mouse wheel	Scroll the report.
ENTER / CS + SPACE (BREAK) / click	Close the report.

Reading the progress display

While running, the report shows the pass number, current main or include filename, source line, and scanned include lines. A spinner and highlighted counters show activity. Updates change only the necessary character cells, avoiding full-window flicker.



During saving, the display shows written bytes and the current operation: directory preparation, open, write, set length, sync, and close. These are live activity figures rather than a percentage of the whole assembly: the total number of included source lines is not known in advance.

Success or failure

A successful result shows binary size and path. Source errors include line and column, plus the include filename when applicable. Save failures show the target path, failed operation, and esxDOS code; they are not errors on the source's first line.

```

NextForge 1.2 by MB Maniax
; Neon border - a small NextForge example
device zxspectrumnext
org $8000

Start:
MainLo
; Fill
ClearS
; Cycl
DrawFr

Phase:
db 0
ds 64

NEON.ASM Ln 10 Col 25 Len 33 Lines 34 Free 319488B

```

Assembly report done

- ✓ 05 Scan source lines
- ✓ 06 Skip comments and blank lines
- ✓ 07 Validate labels
- ✓ 08 Recognize directives
- ✓ 09 Recognize mnemonics
- ✓ 10 Check required operands
- ✓ 11 Reject unexpected operands
- ✓ 12 Reject empty operand slots
- ✓ 13 Check operand separators
- ✓ 14 Save binary / store error
- ✓ 15 Restore screen banks
- ✓ 16 Present result

Result: Binary OK Bytes 16896
 /project/neon.hex
 UP/DOWN/PGUP/PGDN/WHEEL scroll ENTER/ESC/CLICK close

Success: every step green, output size and path.

```

NextForge 1.2 by MB Maniax
; Neon border - a small NextForge example
device zxspectrumnext
org $8000

Start:
MainLo
; Fill
ClearS
; Cycl
DrawFr

Phase:
db 0
ds 64

NEON.ASM Ln 10 Col 25 Len 33 Lines 34 Free 319488B

```

Assembly report done

- ✓ 05 Scan source lines
- ✓ 06 Skip comments and blank lines
- ✓ 07 Validate labels
- ✓ 08 Recognize directives
- ✗ 09 Recognize mnemonics
- ? 10 Check required operands
- ? 11 Reject unexpected operands
- ? 12 Reject empty operand slots
- ? 13 Check operand separators
- ? 14 Save binary / store error
- ? 15 Restore screen banks
- ? 16 Present result

Result: error line 10 col 1
 UP/DOWN/PGUP/PGDN/WHEEL scroll ENTER/ESC/CLICK close

Failure: a red cross on the step, line and column; the status bar names the cause.

Scroll the report with arrows, Page Up / Page Down, or the mouse wheel. ENTER, BREAK, or a click closes it.

Assemble → **Go to Error** navigates to the relevant position in the main source. For an include error, use the reported path and line to open and inspect that file.

WHICH SOURCE IS ASSEMBLED

The main source is read from the selected editor buffer. Includes and incbin files are read from SD. Save edits to an open include before assembly, or the assembler will use its older saved contents.

DEBUGGING THE PROGRAM

Assemble → **Assemble & Run** (EXT → R: assemble, then debug) or, after a successful assembly, the **Debug** menu (EXT → D) opens the monitor and debugger. They are described in Part II · Monitor and debugger.

Assemble → **Run Tests** (EXT → T) assembles the project with its tests and runs them, saving nothing (Part III).

7. Syntax, symbols, and expressions

A semicolon `;` starts a comment. Blank lines and comments are skipped. Instruction mnemonics and register names are case-insensitive; symbols are case-sensitive. Global labels are shared across the entire project, including all includes; dot labels use the scope of the preceding global definition.

```
Start: ld a,1      ; instruction on a label line
Loop:
    djnz Loop
Count equ 16
    dw Start,Count
```

Labels may use a colon, appear alone on a line, or use the form `name instruction`. A colon makes the source clearer. Names can contain up to 64 characters. The symbol table first fills two pages of its own (about 800 typical names) and then takes pages from the shared document pool as needed; the next assembly gives them back. A dot label such as `.loop` belongs to the preceding global definition, so different routines can reuse a local name. Local names allow up to 59 characters including the dot. A reference to `.loop` uses the current scope. `Global.loop` refers to a local label of another scope and `Global.loop:` defines one, as in SjASMPlus; such a definition also opens the scope of the dot labels after it. Module namespaces are not implemented.

Numbers and operators

NOTATION	MEANING
123	Decimal number.
\$8000, #8000, 0x8000	Hexadecimal number.
%10101010	Binary number.
%1'0000'000, \$FF'FF	An apostrophe between digits only groups them, as in SjASMPlus.
'A'	Value of a single character.
\$	Address at the start of the current statement.

Operators have the same precedence as in SjASMPlus (and C). The table lists them from the highest; operators of equal precedence work from left to right.

OPERATORS	MEANING
+ - ~ ! HIGH LOW	Unary plus and minus, bitwise NOT, logical NOT, high and low byte.
* / %	Multiplication, division, remainder.
+ -	Addition and subtraction.
<< >> >>>	Shift left, arithmetic and logical shift right.
< <= > >=	Signed comparisons.
= == !=	Equal and not equal.
&	Bitwise AND.
^	Bitwise XOR.

OPERATORS	MEANING
	Bitwise OR.
&&	Logical AND.
	Logical OR.
()	Parentheses, up to 16 deep.

Comparisons and logical operators give `-1` (true) or `0` (false). Signed division truncates toward zero, and the remainder takes the sign of the dividend. Values and intermediate results must stay within `-65535` to `65535`; numeric literals cannot exceed `65535`. Bitwise operators work on the two's complement, so `~0` is `-1` and `value & $FF` is a byte. `HIGH value` and `LOW value` select the high and low bytes; `high(label)` also works. In DB a quoted character can start an expression, as in `'A' | $80`. SjASMPlus keyword operators (`and`, `shl`, ...) and string escapes are not supported yet.

```
FLAG_A equ 1 << 0
FLAG_B equ 1 << 3
MASK   equ ~(FLAG_A | FLAG_B) & $FF
Start: ld a, MASK
      and FLAG_A | FLAG_B
      ld h1, Table + 2 * 3
      assert Table >= $8000 && Table < $C000
Table: dw 1, 2, 3, high Table
```

Parentheses around the whole operand mean memory access, as in `ld h1, (table)`. An operand that only starts with a parenthesis is a value: `ld bc, (ROWS-1)*80` loads a number, as in SjASMPlus.

Forward references

Instructions, DB, and DW may reference labels defined later. EQU can also contain forward references; up to 32 such expressions are resolved between passes. ORG, MMU, the byte count of DS and similar values that decide the layout must be known on their first encounter; a forward EQU passes there once every symbol it depends on is defined before that line. Cyclic EQU definitions and missing symbols cause errors.

8. Directives and macros

DIRECTIVE	USAGE
ORG address	Set the initial address or change the write address; after MMU, any CPU address is allowed without filling gaps.
MMU first [last],page[,address]	Map Next 8KB pages to slots at assembly time; an optional address performs ORG. See chapter 9 for examples and NEX export.
name EQU expression	Define a constant without emitting bytes.
DB / DEFB / DEFM	Comma-separated bytes and strings; byte expressions range from –128 to 255.
HEX 00 01 FF	Bytes from pairs of hex digits; spaces, commas and quotes between them do not count.
DZ / DC	As DB; DZ adds a zero at the end, DC sets bit 7 of each string's last character.
DW / DEFW	16-bit values, low byte first (little-endian).
DD / DWORD / D24	32-bit and 24-bit values; the expression has 16 bits and its sign is extended.
DS / DEFS count[,fill]	Reserve a known number of bytes; default fill is 0.
REPT count[,name] / ENDR	Repeat a block during assembly; count 0–65535 must be known in pass one. The optional <code>name</code> numbers the repetitions from 0.
STRUCT name[,offset] / ENDS	A structure with fields; see below.
DISPLAY / OPT	Only for SjASMPPlus on the PC; NextForge skips them, even a DISPLAY line longer than 160 characters in an include.
IF / IFN / ELSEIF / ELSE / ENDIF	Conditional assembly on an expression known where the condition stands; see below.
DEFINE name [value], IFDEF / IFNDEF	A flag for IFDEF and IFNDEF; with a value also a constant.
DISP address / ENT	Code after DISP is assembled for the given address but stored at the current one; ENT returns. ORG inside DISP moves only the assembled address.
INCLUDE "file.asm"	Insert a source file. It is looked for beside the including file, then in its parent directories, so <code>include "src/x.asm"</code> works from any file of a project.
INCBIN "file.bin"	Insert an entire binary file, found as for INCLUDE.
SAVEBIN "file"[,address,length]	Select the output filename and optional address range.
DEVICE zxspectrum48 / zxspectrum128 / zxspectrumnext	Accepted device names. DEVICE ZXSPCTRUMNEXT enables MMU paging; other devices remain flat.

DIRECTIVE	USAGE
ASSERT expression	Require a nonzero expression in pass two; comparisons, <code>&&</code> and <code> </code> combine conditions.
TESTCASE "name" / ENDTTEST, EXPECT, TESTCLUDE "file", STUB RST n,label, MOCK routine,fake	Unit tests, their checks, taking tests in only for Run Tests and replacing ROM calls and routines in the debugger; see chapter 26.

```

Message: db "NextForge",0
Table:   dw Message,Start
Buffer:  ds 32,0
        assert Buffer >= $8000

```

ASSERT supports `<` `<=` `>` `>=` `=` `==` `!=` comparisons. DW emits the low 16 bits of a value. DB accepts both single-quoted and double-quoted strings.

Repeating a block: REPT

REPT count and ENDR repeat a source block during assembly. They create no runtime loop. The count must be 0? 65535 and known in pass one; zero skips the body, including label definitions. `$` is evaluated again for each repetition.

```

REPT 4
db low $
ENDR

```

Four nesting levels are supported, with 128 buffered lines shared across active blocks; nested bodies consume additional space. Put REPT and ENDR on separate lines without labels. A block must close in the same file. Its body may use INCLUDE and invoke previously defined macros; REPT inside an invoked macro is unsupported. Repeated definitions of the same label cause an error. BIN and banked NEX output both work; see `tests/fixtures/rept` for an example.

Conditional assembly

Lines between IF expression and ENDIF assemble only when the expression is nonzero; IFN inverts the condition. ELSEIF expression and ELSE add further branches, and the first true one wins. The expression must resolve where the condition stands, so it may use only symbols defined earlier and both passes take the same branches.

```

DEBUG    equ 1
        DEFINE NEXT_ONLY

        IF DEBUG
        call DumpRegisters
        ELSE
        nop
        ENDIF

        IFDEF NEXT_ONLY
        nextreg $07, 3          ; 28 MHz
        ENDIF

```

`IFDEF name` and `IFNDEF name` test names given by `DEFINE name`. As in SjASMPlus, they do not see labels or EQU constants and are case-sensitive. `DEFINE` with a value also defines a constant with that value (an expression), which covers the usual numeric uses; NextForge has no general SjASMPlus text substitution. Skipped lines define no labels, invoke no macros, and may contain directives NextForge does not know. Conditions nest up to 16 levels and work in REPT blocks and in macros, where they can test a parameter; an IF in a macro must close within its body.

Macros

Define a macro before its first use with `macro name [param1, param2, ...]` and finish it with `endm`. Separate several parameters with commas; an invocation must pass the same number of arguments. Arguments split at commas outside quotes, so `' '` or `"a,b"` stays one argument. Macro and parameter names are case-insensitive; SjASMPlus treats parameters as case-sensitive, so spell them the same in the body for compatibility. Substitution matches complete identifiers and preserves quoted text.

```

        macro ESXCALL code
        rst $08
        db code
        ret
        endm

ReadFile:
        ESXCALL $9D

        macro SETMMU slot, page
        nextreg $50+slot, page
        endm

        SETMMU 6, 40

```

There are eight macro slots. A body stores the parameter names and then its lines one after another in 960 bytes, so dozens of short lines fit, each up to 160 characters. Labels with a dot (`.loop`) are local to each invocation, so a macro can hold a loop; a label without one is defined by every invocation, so a second invocation reports a duplicate. A macro may call another, up to 6 levels deep. `INCLUDE` in macro bodies is unsupported.

Structures: STRUCT

Between `STRUCT name[,offset]` and `ENDS`, describe the fields: `[label] type [default]`. The types are BYTE/DB, WORD/DW, D24, DWORD/DD, BLOCK/DS count[,fill], ALIGN [boundary[,fill]] and an earlier structure. As in

SjASMPlus, `name` is the structure's size and `name.field` a field's offset. An instance `[label] name [initializers]` emits the structure's bytes and, with a label, defines `label.field` addresses, nested fields included.

```
        STRUCT sprite
x:      BYTE 0
y:      BYTE 0
pattern: BYTE 1
flags:  WORD $8000
        ENDS

player: sprite 120, 96      ; x, y; the rest keeps its defaults
        ld ix, player
        ld a, (ix + sprite.y)
        ld (player.x), a
```

Initializers fill the BYTE, WORD, D24 and DWORD fields in definition order; BLOCK and ALIGN are skipped. An empty item (`{, 5}`) keeps the default and braces group a nested structure's values (`{{1, 2}, 3}`). Up to 16 structures with 128 fields in total fit.

Data alignment

`ALIGN boundary[,fill]` aligns the address to a known power of two (1 to 32768). Padding bytes are zero or the specified fill value. For example, `align 256` places a table at an address whose low byte is zero.

9. Include projects and output

Split the project into modules and keep one entry file to define their order. The example below assumes that `src/main.asm` defines `Start:` and that all referenced files exist:

```
device zxspectrum128
org $2000
jp Start

include "src/constants.asm"
include "src/main.asm"
incbin "assets/font.bin"

End:

assert End-$2000 <= $2000
savebin "build/demo.bin", $2000, End-$2000
```

How paths are resolved

Relative INCLUDE and INCBIN paths are resolved against the directory of the file containing them. If `src/main.asm` uses `include "util.asm"`, it loads `src/util.asm`. A relative SAVEBIN path always uses the main source directory instead.

Forward and backward slashes, absolute paths, and drive prefixes are accepted. Quote paths containing spaces. Includes allow four active nesting levels; a circular include eventually exceeds that limit. CRLF, LF, CR, tabs, and a final line without a terminator are supported. Include lines may contain up to 160 characters; a longer line passes when a comment starts before character 160, as only its end is lost. When the editor opens a file it wraps long comments but cuts longer lines of code; do not save a file with those from the editor, or their ends are lost.

The output file

Without SAVEBIN, `main.asm` produces `main.bin`; other extensions receive an added `.bin` suffix. Each assembly produces one output. If several SAVEBIN statements occur, the last one selects it. Missing output parent directories are created automatically.

The binary is a contiguous block of at most 65535 bytes. It has no header or stored load address. Its default range begins at the first emitted byte; SAVEBIN with an address and length can select a smaller range. Without MMU, new gaps created by ORG are zero-filled. In flat mode, later ORG statements may revisit used addresses, but cannot go below the first emitted address; existing bytes remain until overwritten.

A LARGER VERIFIED PROJECT

The regression suite includes the complete PAPER project with includes, the ESXCALL macro, and separate code and variable layout. Its 7155-byte output is compared byte for byte with SjASMPlus. See `tests/fixtures/paper` for the example.

A VALID RESULT

A source error leaves the previous binary untouched. If the SD card fails during saving, a partial output file may remain. Treat an output as newly valid only after a successful completion report.

Preparing a banked NEX reports the scanned bank count. Saving reports written/total bytes; exports over 64 KB use KiB.

Assembly-time paging: MMU

After `DEVICE ZXSPECTRUMNEXT`, use `MMU first_slot [last_slot], first_page [,address]`. Slots are 0–7, pages 0–223, and each page is 8 KB. A slot range maps consecutive pages. An aligned starting address can replace a slot number, such as `$8000` for slot 4. The optional final address also performs ORG; otherwise the PC is unchanged.

```
device zxspectrumnext
MUSIC_CODE_PAGE_LO equ 24
MMU 6 7,0
org $C000
Start: ret
MMU 4,MUSIC_CODE_PAGE_LO
org $8000
include "music.asm"
savenex open "music.nex",Start,$BFF0
savenex auto
savenex close
```

MMU controls data placement during assembly and emits no instruction. Use NEXTREG for runtime paging. Distinct pages at the same CPU address remain separate, and NEX saves their actual contents. Output borrows free editor memory; exhaustion reports **assembly output memory full**. Open source documents are preserved.

After MMU, ORG may move to a lower address without filling gaps or erasing existing data. For BIN, specify `SAVEBIN "name.bin",address,length`, which reads the final slot mapping. Prepare all data before SAVENEX AUTO. MMU e/w/n options, PAGE/SLOT, and 16KB ZX128 paging are not implemented yet.

An executable NEX for NextZXOS

`SAVENEX` creates a file containing a NEX V1.2 header and bank data, ready to launch from the NextZXOS browser:

```
device zxspectrumnext
org $8000
Start:
nextreg 7,3
mul d,e
ret
savenex open "demo.nex",Start,$BFF0
savenex core 3,0,0
savenex cfg 7,0,0,1
savenex auto
savenex close
```

OPEN selects the name, entry address, and stack. **CORE** sets the minimum core version. **CFG 7,0,0,1** means white border, close the file after loading, reset NextRegs, and require extended RAM. **AUTO** selects nonzero banks and **CLOSE** completes the configuration; the file is written only after successful assembly.

`SAVENEX BANK n[,n...]` stores the listed banks, empty ones too, in NEX order (5, 2, 0, 1, 3, 4, 6, ...) and rejects a bank that order has already passed. AUTO then adds the nonzero banks after the last one. Without BANK and AUTO the file holds only the header.

The default map and banked MMU layouts are supported. AUTO saves nonzero banks in order 5,2,0,1,3,4,6,7,...,111. Each bank occupies 16 KB; unwritten regions are zero-filled. Entry bank is 0–111 and file version is 2. Banks 48–111 require the 2MB RAM flag, for example `SAVENEX CFG 7,0,0,1`. SCREEN, BAR, CFG3, and appended files are unsupported.

The LoRes tutorial with precomputed tables

Open `lowres_nextforge.asm` and keep `wave_table.bin` and `ribbon_texture.bin` beside it. Assembly produces a 16896-byte `lowres_tutorial.nex`, identical to the original tutorial assembled by SjASMPlus. The NextForge edition replaces two Lua tables with INCBIN and removes host OPT/DISPLAY statements. NextForge does not yet include a general Lua interpreter.

10. Supported instructions and limits

The assembler supports the following baseline Z80 groups in the forms listed. This is not complete compatibility with every SjaSMPlus extension.

GROUP	SUPPORTED FORMS
LD	8-bit registers, (HL), immediates; BC/DE/HL/SP; A with (BC)/(DE) or absolute memory; word registers with absolute memory; SP,HL; $A \leftrightarrow I/R$.
INC, DEC, PUSH, POP	INC/DEC for byte registers, (HL), and BC/DE/HL/SP; PUSH/POP for BC/DE/HL/AF.
Arithmetic	ADD, ADC, SUB, SBC, AND, XOR, OR, CP with a register, (HL), or immediate; ADD/ADC/SBC HL,rr.
Branches and returns	JP/CALL/RET with all eight conditions; JP (HL); unconditional JR or NZ/Z/NC/C; DJNZ; RST.
Bits and shifts	RLC/RRC/RL/RR/SLA/SRA/SLL/SRL and BIT/RES/SET for registers and (HL).
Other	EX DE,HL; EX (SP),HL; EX AF,AF'; IM 0/1/2; IN/OUT with (C), immediate ports with A; standard fixed instructions and ED block transfer/search/I/O.
IX and IY	Immediate or absolute-memory word loads/stores; LD SP,index; INC/DEC; PUSH/POP; ADD index,BC/DE/same index/SP; EX (SP),index; JP (index).
Indexed memory	(IX/IY), (IX/IY±expression): byte loads/stores, INC/DEC, arithmetic, shifts, and BIT/RES/SET. Ordinary H/L are supported too.

Relative branches and index displacements must fit -128 to 127 . `jp (ix)` works, but `jp (ix+0)` is rejected because that instruction has no displacement.

Z80N instructions

Compatibility spellings `MUL DE`, `MUL`, `MIRROR`, `PIXELAD HL`, and `PIXELDN HL` also work. A complete encoding audit is available in `docs/assembler-z80n.md`.

All 29 official Z80N opcode forms are supported, including `NEXTREG n,n`, `NEXTREG n,A`, `MUL D,E`, `SWAPNIB`, `MIRROR A`, `TEST n`, `BSLA/BSRA/BSRL/BSRF/BRLC DE,B`, `ADD HL/DE/BC,A` and immediate-word `ADD`, `PUSH nn`, `OUTINB`, `PIXELDN`, `PIXELAD`, `SETAE`, `JP (C)`, `LDIX`, `LDWS`, `LDDX`, `LDIRX`, `LDPIRX`, and `LDDR`. `PUSH nn` encodes its immediate high byte first. Next instructions are available without a host `OPT --zxnext` statement.

Further technical details are in the repository's `docs/assembler-binary.md` and `docs/assembler-includes.md`.

11. Troubleshooting

PROBLEM	WHAT TO CHECK
Unknown operand / undefined symbol	Exact label case, spelling, definition, and whether its include is present. Start is not START.
Unsupported directive / mnemonic	The supported syntax in chapters 8 and 10. Not every SjASMPlus command is implemented yet.
Error inside an include	The filename and line in the report. Open that file; the location belongs to the include, not necessarily the main source.
Cannot open an include	The file must exist on SD; a relative path starts in the containing source file's directory.
Line too long	At most 80 characters in the editor; at most 160 in an include read directly by the assembler.
Range error	DB -128..255; index displacement and relative branches -128..127; addresses cannot wrap past \$FFFF.
ASSERT failed	The condition and expected code size or address. ASSERT is checked in pass two.
Binary save failed	The target path, free space, SD status, and reported operation / esxDOS code. Editing a label will not fix a save error.
Assembly output memory full	MMU output needs one free 8KB editor-pool page per written virtual page. Close unnecessary source documents to free memory, or reduce the assembled data.
Assembly takes a long time	Watch the pass, file, line, and activity indicator. Both passes read includes again; a larger project can take time on a native Z80.
All file slots full	Close an open file using File → Close.

REPORTING AN ISSUE

Include the NextForge version, main source, required includes and binary inputs, complete error message, and whether you use a Next or CSpect. For save errors, add the output path and esxDOS code.

12. Shortcut reference and printing

CS = CAPS SHIFT, SS = SYMBOL SHIFT. EXT → means press and release CS + SS, then press a key. In contrast, hold CS + SS + number combinations together.

ACTION	SHORTCUT
Menu	EXT → 0
Integrated help	EXT → H
Open / save / save as / close	EXT → O / S / E / W
Open files	CS + SS + 1
Left / down / up / right	CS + 5 / 6 / 7 / 8
Page up / down	CS + SS + 7 / 6
Start / end of line	SS + Q / E
Delete before / under cursor	CS + 0 / 9
Indent / format line	CS + 3 / SS + W
Mark block / select all	CS + 4 / SS + A
Copy / cut / paste	EXT → C / X / V
Find	SS + F
Breakpoint on the line	EXT → B
Assemble / assemble and debug	EXT → A / R
Open the monitor and debugger	EXT → D
Assemble with tests and run them	EXT → T
T-states column after the lines	EXT → K
Go to definition / label completion popup	SS + ENTER / CS + ENTER
Confirm popup selection / cancel dialog	ENTER / CS + SPACE
Debugger: menu / help	EXT → 0 / H
Debugger: step / over / out / continue	S / O / Q / C
Debugger: run to line / set PC	T / J
Debugger: breakpoint / condition / watch	B / K / W
Debugger: tests again, with breakpoints	EXT → T

Every debugger key is listed in chapter 23.

HTML and PDF

The manual also works locally without an internet connection. The CZE/ENG switch preserves your current chapter. To print, select a language and use your browser's print command. The layout is prepared for A4 and the language

switch is hidden in print.

Ready-made files: Czech PDF and English PDF. After editing, regenerate both with `py tools/build_manual.py` from the repository root; Chrome or Edge is required.

NextForge 1.2 · MB Maniax · Manual updated 22 October 2026.

Monitor and debugger

The second part shows how to inspect and debug an assembled program on the Next itself: pause it before the first instruction, step through it, set breakpoints and conditions, view and change memory and registers, and follow what the program does with paging and hardware.

```

Debugger Run Breakpoints View Help PAUSED before instruction PC $801A
[S Step] [O Over] [C Continue] [T Run to] [Q Out] [U Reset] [H Help] [Z Slow]
Focus: CODE Origin $8000 Entry $8000 Size $0054
Disassembly > cursor • breakpoint
801A 3200C0 LD ($C000),A
801D 013B24 LD BC,$243B
8020 3E14 LD A,$14
> 8022 ED79 OUT (C),A
8024 C9 RET
8025 Frame:
8025 3A3080 LD A,(Phase)
8028 C602 ADD A,$02
802A 323080 LD (Phase),A
802D D3FE OUT ($FE),A
802F C9 RET
8030 Phase:
8030 00 NOP
8031 Palette:
8031 E3 EX (SP),HL
8032 1C INC E
8033 03 INC BC
Memory hex / ASCII
C000 00 00 00 00 00 00 00 .....
C008 00 00 00 00 00 00 00 .....
C010 00 00 00 00 00 00 00 .....
C018 00 00 00 00 00 00 00 .....
C020 00 00 00 00 00 00 00 .....
C028 00 00 00 00 00 00 00 .....
C030 00 00 00 00 00 00 00 .....
Source: line $001C main.asm
Breakpoints: 8020 802A Condition HL<>8030
Paused: S step, O over, C continue, T run to, Q step out, U set PC, H help.
Stack
8052 800A AF E344 AF' 0044
8054 0000 BC 0000 BC' 0000
8055 0000 DE 0000 DE' 0000
8058 0000 HL 8031 HL' 0000
805A 0000 IX 0000 SP 8052
805C 0000 IY 0000 PC 801A
805E 0000 I/R 00/6D IM1 DI
8060 0000
8062 0000 S Z H P/V N C
8064 0000 0 1 0 1 0 0
8066 0000
8068 0000 MMU slots 0-7 (pages)
806A 0000 -- 0F 0A 0B 04 05 20 01
806C 0000 T +88 -68
806E 0000 Mode SLOW trace
8070 0000 NEXTREG 56=20
8072 0000 Watch 8030=00
Memory at registers
HL 8031 E3 1C 03 00 00 00 00
DE 0000 C3 AB 02 00 00 00 00
BC 0000 C3 AB 02 00 00 00 00
IX 0000 C3 AB 02 00 00 00 00
IY 0000 C3 AB 02 00 00 00 00
SP 8052 0A 80 00 00 00 00 00
PC 801A 32 00 C0 01 3B 24 3E 14

```

The debugger with a paused program: disassembly, stack, registers, memory and memory previews are visible together.

Chapters 13–23. Start with chapter 14 — a short example walks through everything essential.

13. Meet the debugger

NextForge combines a **monitor** and a **debugger** in one screen. The monitor shows memory as a hexadecimal dump and as disassembly with labels, displays registers, flags and the stack, and lets you change values. The debugger adds controlled execution: the program runs one instruction at a time or up to a breakpoint, and you can inspect what it did in between.

The debugger works with the **last successful assembly**. The program runs in its own private memory; it cannot overwrite the editor, open files, edit history or NextForge itself.

Opening the debugger

1. Assemble the project's main source (**Assemble** → **Assemble**, **EXT** → **A**).
2. Choose **Assemble** → **Assemble & Run** (**EXT** → **R**, assemble, then debug) or **Debug** → **Open Debugger** (**EXT** → **D**). Unit tests run from **Assemble** → **Run Tests** (**EXT** → **T**) in the report window; **D** in that window opens the debugger and runs them there (Part III).
3. The debugger opens **paused before the entry instruction**: nothing has run yet, registers are zero and PC points at the program's entry.

EDITOR DEBUG MENU	WHAT IT DOES
Open Debugger	Opens the debugger with the code panel focused.
Step Into	Opens the debugger and executes one instruction.
Breakpoints	Opens the debugger with breakpoint hints in the status row.
View Registers / View Memory / Disassembly	Opens the debugger with the register / memory / code panel focused.

Back to the editor

BREAK (CS + SPACE) on the idle screen or **Debugger** → **Return to editor** closes the debugger and restores the editor. The paused session is kept: reopen the debugger (**EXT** → **D**) to continue where you left off. Only a new assembly discards it.

WHAT THE DEBUGGER NEEDS

- A successful assembly of the main source — flat binary output or banked output with **MMU** / **SAVENEX**.
- An entry address in **\$2000-\$FFFF**; **\$0000-\$1FFF** belongs to the debugger kernel.
- Enough free memory for a copy of the program; if it runs out, close an unneeded file and assemble again.

CONTROLS

Everything works from the keyboard and the mouse: the menu bar at the top (**EXT** → **O**), context menus on the right button and searchable help (**H**). Every key is listed in chapter 23.

14. A first debugging session

Try the debugger on a short routine. It is in the repository as `tests/fixtures/monitor/monitor.asm`; you can also type it into an empty editor.

```
org $a000
Start:
    ld hl,$1234
    ld de,7
    call AddValue
    ld bc,$0203
    ld ix,$4567
    ld iy,$89ab
    ld ($b100),hl
    ld a,$42
    ld ($b102),a
    ret
AddValue:
    add hl,de
    ret
```

1. **Start.** Choose **Assemble** → **Assemble & Run (EXT → R)** and confirm the main file. The debugger stops at `Start`: the green row is PC `$A000` and the top row reads *PAUSED before instruction*.
2. **Step.** Press **S**. `LD HL,$1234` executes, HL turns yellow (it changed) and PC moves on. A second **S** sets DE to 0007.
3. **Step over a subroutine.** PC is on `CALL AddValue`. **O** (Step Over) runs the whole subroutine: HL = 123B and PC is after the CALL.
4. **Restart.** **U** and ENTER return to the entry and clear the registers. Breakpoints and edited bytes stay.
5. **Breakpoint.** Press **G**, type `AddValue` and ENTER — the code view jumps to the subroutine. **B** on the selected row adds a breakpoint (red dot). **C** runs the program; it stops on the dot, before `ADD HL,DE`.
6. **Back to the caller.** **Q** (Step Out) finishes the subroutine and stops after the CALL.
7. **Memory.** **M** focuses the memory panel and **G B100** ENTER moves it. **C** finishes the program: memory shows `3B 12 42` and the top row reads *RETURNED*.
8. **Condition.** **U** ENTER, **X** clears the breakpoints. **K** opens the list of targets; press **1** (pair HL) and type `=123B` ENTER. **C** stops right after `ADD HL,DE`, where HL first becomes 123B.
9. **Source.** **N** closes the debugger and opens the source line of the code view in the editor. **EXT → D** or the **Debug** menu brings you back — the session continues.

```

Debugger  Run  Breakpoints  View  Help  PAUSED before instruction  PC $A01D
[S Step] [O Over] [C Continue] [T Run to] [Q Out] [U Reset] [H Help] [Z Slow]
Focus: CODE  Origin $A000  Entry $A000  Size $001F
Disassembly  > cursor  ● breakpoint
> A01D AddValue:
● A01D 19      ADD HL,DE
A01E C9      RET
A01F 00      NOP
A020 00      NOP
A021 00      NOP
A022 00      NOP
A023 00      NOP
A024 00      NOP
A025 00      NOP
A026 00      NOP
A027 00      NOP
A028 00      NOP
A029 00      NOP
A02A 00      NOP
A02B 00      NOP
A02C 00      NOP
Memory  hex / ASCII
B100 00 00 00 00 00 00 00 00 .....
B108 00 00 00 00 00 00 00 00 .....
B110 00 00 00 00 00 00 00 00 .....
B118 00 00 00 00 00 00 00 00 .....
B120 00 00 00 00 00 00 00 00 .....
B128 00 00 00 00 00 00 00 00 .....
B130 00 00 00 00 00 00 00 00 .....
Source:  line $0012 main.asm
Breakpoints: A01D
Stopped at a breakpoint. S steps, C continues, B on this line removes it.

```

Stack		Registers	
1FFA	A009	AF	0044
1FFC	028A	BC	0000
1FFE	0000	DE	0007
2000	0000	HL	1234
2002	0000	IX	0000
2004	0000	IY	0000
2006	0000	I/R	00/3B
2008	0000		IM1 EI
200A	0000	S	Z H P/V N C
200C	0000		0 1 0 1 0 0
200E	0000		
2010	0000		MMU slots 0-7 (pages)
2012	0000		-- 0F 0A 0B 04 05 00 01
2014	0000		T +37 =37
2016	0000		Mode SLOW trace
2018	0000		I/O none
201A	0000		Watch off (W)

Memory at registers	
HL 1234	00 00 00 00 00 00 00 00
DE 0007	00 C3 AB 02 00 00 00 00
BC 0000	C3 AB 02 00 00 00 00 00
IX 0000	C3 AB 02 00 00 00 00 00
IY 0000	C3 AB 02 00 00 00 00 00
SP 1FFA	09 A0 8A 02 00 00 00 00
PC A01D	19 C9 00 00 00 00 00 00

Step 5: the program stops at the breakpoint in `AddValue`. The label row precedes the instruction; the dot is on the instruction, the green row is PC and changed registers (SP, PC) are yellow.

15. The debugger screen

All panels are visible at once. The **focus** (active panel) only decides what the arrows, the wheel, **G** and **E** act on.

The screenshot shows the NextForge debugger interface. At the top is a menu bar (1) with options: Debugger, Run, Breakpoints, View, Help. To the right of the menu bar is the session state (2) showing 'PAUSED before instruction' and the current PC 'PC \$801A'. Below the menu bar is a toolbar (2) with buttons: [S Step], [O Over], [C Continue], [T Run to], [Q Out], [U Reset], [H Help], [Z Slow]. The main area is divided into several panels. The 'Focus' panel (3) shows 'CODE' with 'Origin \$8000', 'Entry \$8000', and 'Size \$0054'. Below it is the 'Disassembly' panel (4) showing a list of instructions with addresses, hex values, and assembly code. The 'Stack' panel (5) shows a list of stack entries with addresses, hex values, and stored words. The 'Registers' panel (6) shows a list of registers with their current values and flags. The 'Memory' panel (8) shows a table of memory addresses, hex values, and ASCII characters. The 'Memory at registers' panel (9) shows a table of memory addresses, hex values, and stored words. The 'Source' panel (10) shows the source file 'line \$001C main.asm'. The 'Breakpoints' panel (11) shows a list of breakpoints with addresses and conditions. The 'Status' panel (12) shows the current status 'Paused: 5 step, 0 over, C continue, T run to, Q step out, J set PC, H help.'.

```
Debugger Run Breakpoints View Help PAUSED before instruction PC $801A
[S Step] [O Over] [C Continue] [T Run to] [Q Out] [U Reset] [H Help] [Z Slow]
Focus: CODE Origin $8000 Entry $8000 Size $0054
Disassembly > cursor breakpoint
801A 3200C0 LD ($C000),A
801D 013B24 LD BC,$243B
8020 3E14 LD A,$14
> 8022 ED79 OUT (C),A
8024 C9 RET
8025 Frame:
8025 3A3080 LD A,(Phase)
8028 C602 ADD A,$02
802A 323080 LD (Phase),A
802D D3FE OUT ($FE),A
802F C9 RET
8030 Phase:
8030 00 NOP
8031 Palette:
8031 E3 EX (SP),HL
8032 1C INC E
8033 03 INC BC
Memory hex / ASCII
C000 00 00 00 00 00 00 00 00 .....
C008 00 00 00 00 00 00 00 00 .....
C010 00 00 00 00 00 00 00 00 .....
C018 00 00 00 00 00 00 00 00 .....
C020 00 00 00 00 00 00 00 00 .....
C028 00 00 00 00 00 00 00 00 .....
C030 00 00 00 00 00 00 00 00 .....
Source: line $001C main.asm
Breakpoints: 8020 802A Condition HL<>8030
Paused: 5 step, 0 over, C continue, T run to, Q step out, J set PC, H help.
```

A program paused inside Init: two breakpoints, a condition on HL and a watch on the Phase byte.

- 1 Menu bar and session state (PAUSED, TRACING, RETURNED...) with PC.
- 2 Toolbar — the buttons are clickable; [Z Slow/Fast] shows the run mode.
- 3 Focused panel, origin, entry and size; MMU for banked output.
- 4 Disassembly: address, bytes, instruction, labels.
- 5 Stack from SP: address and stored word.
- 6 Registers, I/R, interrupt mode and flags.
- 7 The program's MMU map, run mode, last I/O and the watched byte.
- 8 Memory: eight bytes per row, hex and ASCII.
- 9 Memory previews at HL, DE, BC, IX, IY, SP and PC.
- 10 Source file and line for the viewed address.
- 11 Breakpoints and the condition.
- 12 Status message; when entering values, rows 11–12 become the input.

Colours and marks

MARK	MEANING
Green row	The instruction at PC — it runs next.
Blue row and >	The view cursor (selected code row); the first memory row when memory has focus.
Red dot	A breakpoint on that instruction row.
Yellow register value	Changed by the last step or run.
Instruction colours	As in the editor: labels, instructions, registers and numbers.

Session state

MENU BAR TEXT	MEANING
PAUSED before instruction	The program is stopped; the instruction at PC has not run yet.
TRACING	Slow tracing is running; BREAK pauses it.
RETURNED	The program finished with its final RET. U runs it again.
IMAGE ONLY	The image can be viewed but not run (entry below <code>\$2000</code> or outside the output).
NO PROGRAM	No successful assembly.

Register panel

The top six rows hold AF, BC, DE, HL, IX, IY, SP, PC and the alternate AF', BC', DE', HL'. Below them are **I/R** (R includes debugger overhead) and **IMn EI/DI** — the simulated interrupt mode and enable state. Then come the flags **S Z H P/V N C**, the **MMU slots 0–7** row with the program's virtual pages (slot 0 is the debugger kernel), **Mode SLOW/FAST** (*SLOW* +screen with L on), the **I/O** row with the last simulated access (for example `NEXTREG 56=20` or `I/O OUT 02FE=02`) and **Watch** with the watched byte.

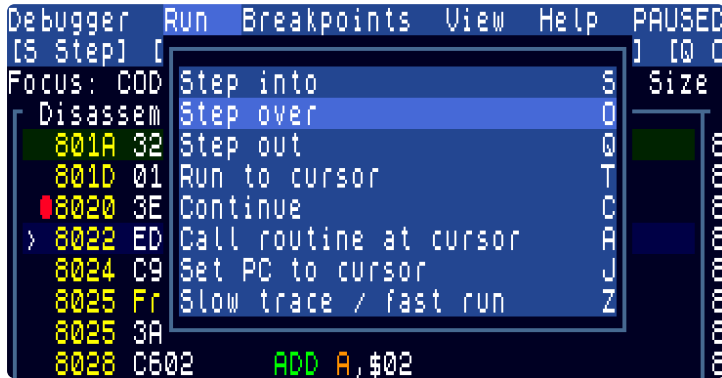
The **T +n =m** row counts T-states while stepping and in slow traces: +*n* since the last command began (S, O, C, run to cursor...), =*m* since the session started or U. The documented Z80 and Z80N timings apply: a taken branch adds its T-states and a repeating block instruction such as LDIR counts every repetition. Fast runs count nothing and show ? instead.

Stack and memory previews

The stack panel lists 17 words upward from SP — return addresses and saved registers. The previews show eight bytes from the address in HL, DE, BC, IX, IY, SP and PC; keys **1–7** move the main memory panel to the same address.

16. Menus, context menus and help

The top row is the menu bar **Debugger · Run · Breakpoints · View · Help**. Open it with **EXT** → **0** or by clicking a menu name. LEFT/RIGHT switch menus, UP/DOWN select an item, ENTER or a click runs it. The letter or digit at the right of an item runs it directly. BREAK or a click outside closes the menu.

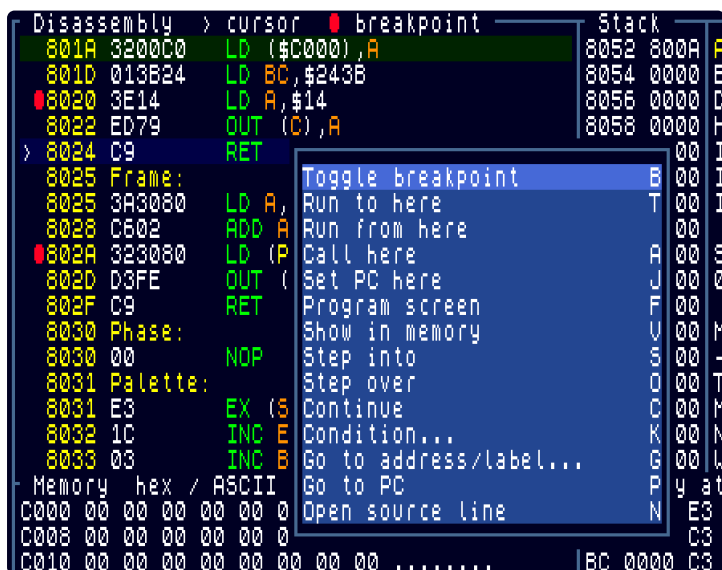


The Run menu with Step over selected.

MENU	ITEMS	KEYS
Debugger	Return to editor · Open source line · Reset at entry · Set restart entry · Debugger help	BREAK · N · U · I · H
Run	Step into · Step over · Step out · Run to cursor · Continue · Call routine at cursor · Set PC to cursor · Slow trace / fast run	S · O · Q · T · C · A · J · Z
Breakpoints	Toggle breakpoint · Condition · Watch byte change · Clear breakpoints	B · K · W · X
View	Focus code · memory · registers · Next panel · Go to address/label · Go to PC · Program screen · Screen in slow traces · Memory at code cursor · Edit value · Memory at HL, DE, BC, IX, IY, SP, PC	D · M · R · TAB · G · P · F · L · V · E · 1–7
Help	Debugger help · Return to editor	H · BREAK

Context menus

A right click on a code row, memory byte or register first selects what is under the pointer, then opens a menu of commands that fit it. Pick an item with the left button, the arrows and ENTER, or its letter; the right button or BREAK closes the menu.



The context menu of a code row.

PANEL	ITEMS
Code	Toggle breakpoint, Run to here, Run from here, Call here, Set PC here, Program screen, Show in memory, Step into, Step over, Continue, Condition, Go to address/label, Go to PC, Open source line.
Memory	Edit byte, Watch byte change, Condition, Go to address/label, Memory at PC, Memory at SP.
Registers	Edit register, Condition, Memory at HL, Memory at SP, Go to PC.

Searchable help

H (or EXT → H) opens the debugger's help window. It works like the editor's help: typing filters descriptions and keys (case-insensitive), Backspace removes filter characters, the arrows, Page Up / Down and the wheel scroll, and BREAK, EXT → H or a click on **[Close]** closes it. The help lists every command and the rules for running programs.

```

Debugger  Run  Breakpoints  View  Help  PAUSED before instruction  PC $801A
[
F  NEXTFORGE DEBUGGER - SEARCHABLE HELP                                [ Close ]
  FUNCTION / CONTEXT                                                    SHORTCUT / MENU
  Continue to breakpoint, watch, condition or RET                      C
  Condition: pick register/pair/memory, then a value                   K, = <> < > <= >=
  Condition stops at breakpoints; with none, anywhere                  K; X clears
  Clear all breakpoints, editor marks, condition                       X

  Filter: cond
  Matches: 4
  Type to filter  UP/DOWN/PGUP/PGDN/WHEEL scroll  BREAK / EXT->H close
E
S/O step  C continue  T run to  Q step out  EXT->Q menu  H help  BREAK editor

```

Help filtered by "cond".

17. Stepping and running

KEY	ACTION
S — Step into	Executes one instruction. CALL and RST enter the subroutine.
O — Step over	Runs a whole CALL, including nested calls, and stops after it. Acts like S for other instructions.
Q — Step out	Runs to the caller using the return address at the top of the stack, so undo your own PUSHes first. At the top level it runs to the final RET.
T — Run to cursor	Runs to the selected code row; stops earlier at any breakpoint on the way.
C — Continue	Runs to a breakpoint, a met condition, a watched byte change or the final RET.
A — Call routine	Calls the routine at the selected row: the current PC is pushed as its return address and the program runs until the routine returns. Registers stay as the routine left them.
J — Set PC	Moves PC to the selected row without executing code — skips or repeats instructions. Registers and stack are unchanged.
Run from here	Code context menu: PC to the selected row, then continue (J and C).
F / L	The program's screen / the screen during slow tracing (see below).
P — Go to PC	Moves the code view to PC.
U — Reset	After ENTER, resets registers, stack, paging and simulated hardware at the entry. Program memory (including edits) and breakpoints stay; only a new assembly reloads the image.
I — Entry	Sets the restart entry (hex address or label).

When the program executes its final RET, the session ends and the menu bar shows *RETURNED*. **U** or **S** then starts a new session at the entry.

Slow tracing and fast runs

Continue (C), Step Over and Run to cursor have two modes. **Z** or the **[Z Slow]** / **[Z Fast]** button switches between them.

	SLOW TRACING (DEFAULT)	FAST NATIVE RUN
Execution	Instruction by instruction, hardware simulated.	The program runs at full speed on the Z80.
DI/EI/HALT, I/O, NEXTREG, MMU	Safely simulated (chapter 21).	Reach the real hardware.
BREAK	Pauses at any time, checked once per frame.	Stops only while the program has interrupts enabled.
Conditions and watches	Yes.	Always switch to tracing.

	SLOW TRACING (DEFAULT)	FAST NATIVE RUN
Speed	Thousands of instructions per second; the screen refreshes about 3 times a second.	Full speed.
Screen	The dashboard; with L the program's screen.	The program's screen.

FAST MODE

A fast run is real: the program owns the ports, Next registers, paging and the screen. Before the run the debugger moves every program page to the physical page its number names (also the display banks 5 and 7 and both Layer 2 buffers), so paging the program does itself finds its data and the display shows its picture. When the run stops the pages go back, the debugger adopts the MMU map, reads the display state and restores its own display. Moving takes about 15 ms per 8KB page, so even a short O or T in fast mode briefly shows the program's screen.

The program's screen

The debugger keeps a shadow of the program's display state: Layer 2 (banks, resolution, scroll, port \$123B), LoRes, Timex modes, the shadow screen in bank 7, the tilemap, ULA scroll, sprites and layer order, transparency, palettes, clip windows and the copper control. Steps write to the shadow, fast runs straight to the hardware, and the debugger reads the state back after a run. The border colour and sprite patterns go to the hardware at once.

KEY	WHAT YOU SEE
F — Program screen	The program's screen as it stands — also after the program finished. Any key or mouse button returns to the debugger.
L — Screen in slow traces	Slow tracing (C, T, Q, O) shows the program's screen instead of the dashboard; every instruction shows at once. BREAK or a stop brings the dashboard back. The mode row reads <i>Mode SLOW +screen</i> .
Z = Fast	A fast run shows the program's screen for its whole duration.

EXAMPLE: LORES

Step through the LoRes set-up with **S** and press **F** to see what the program has drawn so far. **L** and **C** show the drawing live, instruction by instruction. After **Z** and **C** the program runs at full speed on its own screen until you press BREAK or it reaches a breakpoint.

Stop messages

MESSAGE	MEANING
Paused: S step...	A step or fast run ended at a breakpoint or target address.
Stopped at a breakpoint	Tracing reached a breakpoint.
Condition met at breakpoint	A breakpoint with the condition met.
Condition became true	A condition without breakpoints has just become true.
Watch changed after instruction at \$xxxx	The instruction at that address changed the watched byte.
Trace paused by BREAK	Tracing was interrupted by BREAK.

MESSAGE	MEANING
Fast run stopped by BREAK	A fast run stopped by BREAK; PC is the interrupted instruction.
Stopped at RST	The program called ROM or esxDOS (chapter 22). <code>STUB RST n,label</code> replaces the call with a routine of yours (chapter 26).
Routine returned	The program finished with its final RET.

18. Breakpoints, conditions and watches

Breakpoints

B (or EXT → B) toggles a breakpoint on the selected code row, or at an address chosen with **G**. The code context menu offers the same. A red dot marks the instruction, and all breakpoint addresses are listed on the second-to-last row. Up to eight can be set, in `$2000-$FFFF`. **X** clears all breakpoints (including the editor marks) and the condition.

A breakpoint belongs to a CPU address, not to a particular memory page: it applies to whatever code is mapped there.

Breakpoints from the editor

Lines marked in the editor (EXT → B, chapter 3) become breakpoints every time the debugger opens — from the **Debug** menu (EXT → D) and through **Assemble & Run** (EXT → R). Breakpoints therefore survive a new assembly: assembling clears them, and the editor marks bring them back the next time the debugger opens.

- **A line without code** (a label, comment or blank line) stops at the first instruction that follows it in the same file.
- **Include files:** marks work in every open file that the build reached through `INCLUDE`; the file is recognised by its path.
- **From the debugger to the editor:** B on the instruction that starts a source line marks that line in the editor. Removing the breakpoint removes the mark, including a mark on a label or comment above that led to the instruction. **X** clears the marks in all open files.
- **The debugger's own breakpoints:** a breakpoint at an address that no line starts (inside data, an address chosen with G), or in a file that is not open in the editor, belongs to the debugger alone and stays when it opens.
- **After editing the source** the assembly's line map no longer applies: the debugger keeps its list and takes new marks over after the next assembly.
- The limit of eight breakpoints applies; further marks are not used.

Conditions

A condition compares a register or memory with a value. **K** (also **Breakpoints** → **Condition...** and the context menus) opens the list of targets:

WHAT WATCHES AND CONDITIONS MISS

Values are always compared after a whole instruction: writing the same value, reads and the intermediate steps of a block instruction (LDIR) are not caught. Conditions and watches always use slow tracing.

19. Monitor: memory, registers and editing

Navigation and focus

KEY	ACTION
TAB / D / M / R	Next panel / code / memory / registers. A click also focuses a panel.
Arrows / wheel	Code: select a row and scroll. Memory: move by 8 bytes. Registers: select a register.
CS + SS + 7 / 6	Move the view by 256 bytes.
G	Move the focused view to a hex address (\$ optional) or global label. PC is unchanged.
P	Code view to PC.
1–7	Memory at the address in HL, DE, BC, IX, IY, SP, PC.
V	Memory at the address of the selected code row.

Labels in **G** (and in **K** and **I**) can be typed in any letter case: an exact match wins, otherwise the first label that differs only in case is used (`start` finds `Start`). Labels are valid only until the source is edited; then the debugger reports *Source changed since assembly* and you need to assemble again (**EXT** → **A**). Dotted local labels are not looked up, and a name that looks like a hex number (for example `BEEF`) is read as a number.

The memory panel

Seven rows of eight bytes in hex and ASCII (non-printable characters show as dots). Clicking a byte selects it for **E**, **W** and **K**. Memory the program has not used yet reads as 00. During a session `$0000-$1FFF` shows the debugger kernel and the default stack.

Editing

E edits the value in the focused panel:

- **A memory byte** — hex 00–FF at `$0700-$FFFF` . The page is allocated on the first write. Changing the watched byte resets the watch baseline.
- **A register** — the selected register's word, only while paused. PC must be in `$2000-$FFFF` ; SP may be 0 (stack from `$FFFF`) or at least `$0700` . I and R are not editable.

Select a register with **R** and the arrows, or by clicking; the selected register is highlighted.

Disassembly

The disassembler knows the full Z80 set including IX/IY, CB and ED, and the Z80N instructions (for example `NEXTREG` , `MUL D,E` , `PUSH nn`). Unknown ED codes show as `DB` . Addresses matching a label are replaced by its name in operands; a label itself gets its own row above the instruction. Byte values stay numeric.

20. Source code and labels

The debugger uses the symbol table and source line map of the last successful assembly.

- **Labels** appear in the disassembly and can be used in **G**, **I**, **E** and conditions. When you change the source, *Symbols stale* appears at the top, labels are hidden and names cannot be entered — assemble again.
- The **Source row** in the bottom frame shows the file and line (hexadecimal, one-based) for the address in the code view, or the memory cursor.
- **N** closes the debugger and opens that line in the editor. An include that is not open is loaded from SD with its correct directory; an open one is reused.

THE SOURCE MAP

Assembly records up to 1024 address ranges and 31 include paths, including nested includes and REPT replay. An expanded macro maps to its invocation line. Data, INCBIN and ORG padding map to the line that emitted them. When capacity runs out the map is partial and * follows the filename.

TIP

Includes are assembled from the SD card. Save them before assembling, or the debugger shows lines of the old version.

21. How the debugger runs a program

Program memory

AREA	CONTENTS DURING A SESSION
\$0000-\$1FFF (slot 0)	The debugger kernel: RST vectors, the interrupt table, the state capture code and the default stack (\$1FFC). Data may be written from \$0700 .
\$2000-\$FFFF (slots 1–7)	The program's private pages: code, data, stack and breakpoints.

Every 8KB page the program uses has its own copy in the debugger, so the program cannot overwrite the editor, documents or NextForge code.

Start state

- The **entry** is the SAVENEX OPEN start address, otherwise the first emitted byte. Change it with I.
- The **stack** is the SAVENEX stack, otherwise \$1FFC in the kernel. A return address sits on top, so the final RET ends the session.
- **Registers** are zero, flags as after XOR A, interrupts enabled in IM 1 — as after starting from BASIC.
- **Paging**: flat output uses the map 14, 15, 10, 11, 4, 5, 0, 1; NEX output the loader's map (banks 5, 2 and the entry bank, ROM in slots 0/1); banked output without SAVENEX the final map of its MMU directives.

Hardware simulation

Steps and slow tracing never send hardware instructions to the Next:

INSTRUCTION	STEP AND SLOW TRACING	FAST RUN
DI, EI	Change the simulated interrupt state.	Native.
IM 0/1/2, LD I,A, LD R,A, LD A,I	Simulated.	Native (IM is not tracked).
HALT	Completes; in IM 2 it enters the program's handler through the vector at I×256+\$FF . After DI it stops with an explanation.	Waits for an interrupt.
RETI, RETN	Return like RET.	Native.
NEXTREG \$50–\$57, ports \$243B/\$253B	Switch a virtual page; reading \$253B returns the simulated MMU value.	The real MMU.
Ports \$7FFD, \$DFFD	Switch the bank at \$C000 .	Real paging.
Display NEXTREG and ports (Layer 2, LoRes, Timex, tilemap, palettes, sprites, copper, clip, scroll)	Go to the program's screen shadow (chapter 17); with L to the hardware too. The border and sprite patterns go to the hardware at once.	Real hardware.
Other OUT, NEXTREG, OUTI/OTIR, OUTINB	Recorded in the I/O row, not sent to the hardware.	Real hardware.
IN, INI/INIR	Reading \$253B or \$123B returns the shadow; otherwise \$FF (no key pressed), Kempston 0.	Real hardware.
RST \$00–\$38	Stop with an explanation — ROM and esxDOS are not available. An RST with a STUB calls its replacement.	Stop with an explanation. An RST with a STUB jumps to its replacement.

INSTRUCTION	STEP AND SLOW TRACING	FAST RUN
JP (C)	Refused.	Native.

Banked MMU and NEX output

When the source uses the `MMU` directive, the assembler's output pages are handed to the debugger. The program then switches the same pages the assembler produced, and the **MMU slots** row shows which virtual page is in which slot. Reset (U) restores the start map.

Fast runs

In fast mode the kernel first exchanges pages: every program page (except page 30, the kernel), the display banks 5 and 7 and the Layer 2 banks move to the physical page their numbers name, while NextForge's pages wait in the program's copies. The kernel then applies the display registers and CPU speed, restores the program's registers and hands it the CPU. While the program has interrupts enabled, every frame interrupt passes through the kernel, which checks BREAK; a program with its own IM 2 uses its own handler. When the run stops the kernel reads the MMU map (the debugger adopts the program's paging) and exchanges the pages back; the debugger reads the display registers, clip windows and palette and restores the text display, sprites, Layer 2, the copper and the CPU speed.

Memory pages and NextForge

The page map matters only for programs debugged in the debugger. A program started outside NextForge — after a reset, from the NextZXOS browser or BASIC — no longer has NextForge in memory and can use any pages under the usual NextZXOS rules.

IN SHORT

In the debugger a program can use almost any page number 0–223, including the ones NextForge uses: the debugger gives it private copies and exchanges them into place for fast runs. Only avoid page 30 and slot 0, and let the debugger learn the pages a program maps itself before the program uses them in a fast run.

The debugger knows a page when:

- the assembler wrote output into it (`ORG` , `MMU` , `SAVENEX BANK`),
- it is mapped in a slot, or the program mapped it while stepping or slow tracing,
- it is a display page: bank 5, the first half of bank 7 and the Layer 2 buffers once the program uses them.

The program uses such a page safely under its own number. Watch out for:

- **Page 30** (the lower half of 16K bank 15) — the debugger kernel. Fast runs do not exchange it into place, so a program that maps it itself overwrites the kernel.
- **Slot 0** (`$0000–$1FFF`) — it belongs to the kernel during a session.
- **Pages the program first maps during a fast run** — a page number computed at run time, a buffer outside the assembler output, or Layer 2 enabled only in a fast run. These are physical pages: if NextForge uses one, the program overwrites its code, data or open documents, and the debugger does not know their contents after the stop. The NextZXOS default Layer 2 (banks 9–11, pages 18–23), for example, includes the debugger's page 22. Map such pages once while stepping or slow tracing, or put them in the output.
- **The amount of memory** — copies of program pages come from the pool shared with documents; when it runs out the debugger reports *Not enough free memory*. A fast run exchanges at most 96 pages.

Physical page map

PAGES	BANK 16K	CONTENTS
0–1	0	NextForge main code.
10–11	5	NextZXOS: system variables and the ULA screen. NextForge backs them up in 24–25.
14	7 (lower)	Tilemap and font of the 80-column text display.
15	7 (upper)	NextZXOS: the file browser.
22	11 (lower)	Debugger: the saved editor screen and the program's palette.
24–25	12	Backup of pages 10–11.
26	13	Variables, stack and the IM 2 interrupt table.
27, 31	13, 15	Symbol table records (more come from the pool).
28–30	14–15	Debugger code (28–29) and its kernel (30).
32–39	16–19	Assembler output (64 KB).
40–41	20	Macros and forward EQUs.
42–43	21	Include file search, the editor breakpoint sync, the unit test runner and the editor's T-states column.
44–45	22	Symbol table and expression code.
46–47	23	Help and the program screen code; the blank text page.
48–49	24	More editor and assembler code.
50–51	25	Include stack; debugger menus and help.
52–55	26–27	Assembler, editor extensions and debugger variables.
56–95 (1 MB) 56–175 (2 MB)	28–47 28–87	Shared pool: document text, edit history, banked MMU output, a larger symbol table and copies of the debugged program's pages.

NextForge does not use pages 2–9, 12–13, 16–21 and 23 (and 176–223 on a 2 MB Next). That does not make them automatically safe for pages a program first maps in a fast run: NextZXOS reserves pages 0–23 for itself (the system and the default Layer 2), and installed drivers get the highest pages.

How much memory NextForge takes

`NEXTFORGE.NEX` is 176.5 KB: a header and 11 banks of 16 KB of code and data. While running, NextForge permanently uses 36 pages (288 KB: pages 0–1, 14, 22 and 24–55). It also reserves the shared pool — 40 pages (320 KB) on a 1 MB Next, 120 pages (960 KB) on 2 MB — and allocates pages from it as needed. That is 608 KB of the 768 KB a 1 MB Next has for programs, or 1,248 KB of 1,792 KB on a 2 MB Next. The debugger takes the copies of the program's pages from the pool.

22. Limits and troubleshooting

MESSAGE OR SITUATION	CAUSE	WHAT TO DO
NO PROGRAM / No program image	No successful assembly.	Assemble the main source.
IMAGE ONLY	Entry below <code>\$2000</code> or on a page without output.	Change ORG or the entry (I).
Not enough free memory	The shared page pool is full.	Close an unneeded file and assemble again.
Cannot step here	PC below <code>\$2000</code> , JP (C) or too many prefixes.	J to a valid row, or U.
RST ... calls ROM/esxDOS / Stopped at RST	The program calls ROM or esxDOS.	J skips the call; the debugger offers no ROM services.
EXPECT failed / FAIL n/N ...	A unit test check did not hold.	C continues after the check; see chapter 27.
HALT with interrupts disabled	HALT after DI would never end.	Add EI, or use J.
Breakpoint rejected	Address below <code>\$2000</code> or eight already set.	Remove another breakpoint.
Watch/condition needs...	The program is not paused or the address is below <code>\$0700</code> .	U ENTER, another address.
Not accepted	Invalid number or label, or above FF for a byte.	Correct the input.
Step out needs...	No return address on top of the stack.	Step to RET, or use T.
Symbols stale / Source changed since assembly	The source changed after assembly; labels and source lines are unavailable.	Assemble again (EXT → A).
No source range	The address does not come from source (empty memory).	—
A fast run cannot be stopped	A loop after DI without a breakpoint.	Reset the machine; debug such parts with tracing.
BREAK closed the debugger	BREAK on the idle screen leaves the debugger.	Open Debug again — the session continues.
The program's screen is black	The program has not drawn anything yet, or draws to a page that is not displayed.	Step on and use F; for Layer 2 check <code>NEXTREG \$12</code> .
Fast runs flash the screen	Every fast run (also O and T) shows the program's screen.	Step in slow mode (Z).

Limits

- At most eight breakpoints, one condition and one watched byte.
- Breakpoint marks in the editor are not saved to the file.
- Breakpoints belong to CPU addresses, not memory pages.
- Watches and conditions compare state after whole instructions; they do not monitor the bus.
- Sprite attributes cannot be read back: when the debugger returns, the program's sprites are hidden (patterns stay, except pattern 63 used by the mouse pointer). Programs usually set them again on the next frame.
- The copper stops while the debugger is shown; it restarts with the last `NEXTREG $62` written while stepping.
- Layer 2 paging through port `$123B` (writes to `$0000-$3FFF`) works in fast runs only; map Layer 2 with MMU while stepping.
- The live screen during tracing (L) shows pages 2–25 and 32–39 (banks 5 and 7, Layer 2 in banks 1–12 and 16–19); F or a fast run shows the rest. Page 30 (the kernel) cannot be shown.

- ROM and esxDOS are unavailable (`STUB` replaces calls through RST, chapter 26); IN returns `$FF` , so the program does not read the keyboard.
- Slot 0 (`$0000-$1FFF`) belongs to the kernel; program writes there can damage it.
- A program cannot use page 30; pages it first maps during a fast run are physical (see Memory pages and NextForge).
- The R register includes debugger overhead. IM changes in fast mode are not tracked; IM 2 is detected from a change of I.
- Take care with self-modifying code: it must not overwrite an armed breakpoint.

23. Debugger key reference

KEY / MOUSE	ACTION
EXT → 0 / click row 0	Menu bar.
Right button	Context menu for code, memory or a register.
Left button / wheel	Select a row, register or byte / scroll the focused panel.
S / O / Q	Step into / over CALL / out to the caller.
C / T	Continue / run to the selected row.
A	Call the routine at the selected row.
F / L	The program's screen / the screen during slow tracing.
J / P	Set PC to the selected row / view PC.
U, ENTER / I	Reset at the entry / choose the entry.
Z	Slow tracing / fast run.
B or EXT → B	Toggle a breakpoint (and the line mark in the editor).
K	Condition: target, operator, value.
W	Watch / stop watching a byte.
X	Clear breakpoints, editor marks and the condition.
TAB / D / M / R	Next panel / code / memory / registers.
Arrows / CS + SS + 7, 6	Select and scroll / by 256 bytes.
G	Go to an address or label.
1–7 / V	Memory at HL, DE, BC, IX, IY, SP, PC / at the code cursor.
E	Edit a byte or register.
N	Open the source line in the editor.
H or EXT → H	Searchable help.
EXT → T	Run the unit tests again, with breakpoints (Part III).
BREAK	Pause tracing, close a menu, leave the debugger.

Unit tests

The third part shows how to check assembly routines with automatic tests right on the Next: you write tests with the expected registers, flags and memory next to your routines, and one key assembles the project, runs the tests and lists in the report window which ones passed and which did not.

```

NextForge 1.2 by MB Maniax
; Arithmetic and memory routines with their unit tests in arithmetic.tst.
; Assemble (EXT->A) builds the routines alone; Run Tests (EXT->T) also
; takes in the TESTCLUDE file at the end and runs its tests.
; Each routine states what it takes, what it returns and what it keeps:
; exact
;
; HL =
add16: ✓ 01 add16: 1000 + 234                      68 T
        ✓ 02 add16: overflow sets carry              68 T
        ✓ 03 add8sat: below the limit                 56 T
        ✓ 04 add8sat: stops at 255                    67 T
; A =
        ✓ 05 sub16: 1234 - 234                        72 T
add8sa: ✓ 06 sub16: equal numbers give zero            72 T
        ✓ 07 sub16: borrow when DE is bigger          72 T
        ✗ 08 sub16: ignores the carry flag: HL=#0006, expected #0007
        ✓ 09 memmove: copies 8 bytes                  596 T
; HL =
        ✓ 10 memmove: zero length copies nothing      96 T
; The
        ✓ 11 memmove: overlap, move up by 2           404 T
sub16:  ✓ 12 memmove: overlap, move down by 2         357 T

Result: 1 of 12 tests failed.

UP/DOWN/WHEEL scroll ... D debugger ... ENTER/ESC/CLICK close ...
; Copi
; BC = 0 copies nothing (a plain LDIR would copy 65536 bytes).
memmove: ld     a,b
         or     c
         ret    z
         push  hl
arithmetic.asm Ln 47 Col 17 Len 41 Lines 48 Free 294912B

```

The Run Tests window: eleven tests passed (green tick), test 8 failed (red cross) and the result is red.

Chapters 24–28. Start with chapter 25 — the example with addition, subtraction and moving memory covers everything that matters.

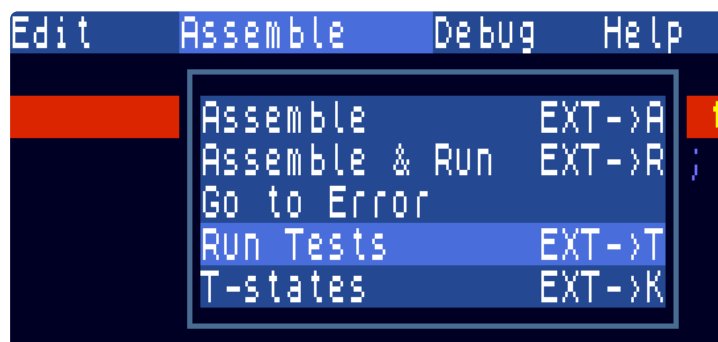
24. Meet unit tests

A unit test is a short piece of code that calls one routine with prepared inputs and checks its result. When you change the routine later, run the tests again and you see at once whether it still does its job — including the cases you would not try by hand: overflow, a zero length, overlapping memory.

NextForge runs the tests **on the Next itself**, in the same isolated environment as the debugger from Part II: the code under test runs in its own memory, so it cannot overwrite the editor or open documents, and the checks look at the real registers, flags and memory of the Z80N.

How it works

1. You write tests between `TESTCASE "name"` and `ENDTEST`, usually in a separate file with the `.tst` extension.
2. Inside a test you set up the inputs with ordinary instructions, call the routine and state with `EXPECT` what must hold: a register value, a flag or memory contents.
3. The project's main file takes the tests in with `TESTCLUDE "file.tst"`. A normal assembly skips it, so the finished program contains no tests.
4. **Assemble → Run Tests (EXT → T)** assembles the project with its tests and runs them in the background. Instead of the assembly steps, the report window lists every test: a green tick passed, a red cross failed (with the reason). The *Result* row is green or red.
5. After the window closes, the `TESTCASE` lines in the editor are coloured by their result and the status row shows the summary. The **D** key in the window opens the debugger, which runs the tests again and stops at the first failure.



Run Tests in the Assemble menu (EXT → T).

WHAT A TEST IS

Each `TESTCASE ... ENDTEST` block is a subroutine: `ENDTEST` emits `RET`. Tests are called with registers cleared, their own stack and the memory as it was before the first test, as if the program started afresh. An `EXPECT` check is an `RST $30` instruction followed by a few data bytes; the debugger recognises it and evaluates it as one step.

WHY ON THE NEXT

The tests run on the real CPU. You can inspect, step through and fix a failed test straight away in the debugger you already know. No PC and no other tools are needed.

25. Your first tests step by step

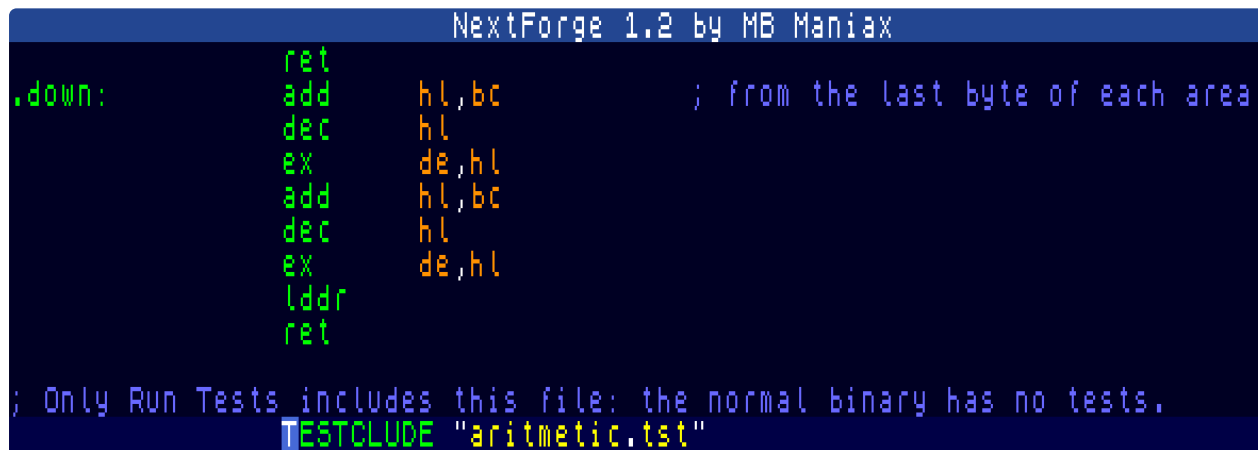
The example lives in the repository in `tests/fixtures/unittest: arithmetic.asm` with the routines and `arithmetic.tst` with their tests. Each routine's comment says what it takes, what it returns and what it keeps — exactly what the tests check:

```
org      $8000

; HL = HL + DE. Carry set when the sum overflows 16 bits. Keeps DE.
add16:   add    hl,de
        ret

; HL = HL - DE. Carry set when DE was bigger, Z when the result is 0.
; The carry flag on entry does not matter. Keeps DE.
sub16:   or     a                ; SBC also subtracts the carry
        sbc    hl,de
        ret

...
; Only Run Tests includes this file: the normal binary has no tests.
TESTCLUDE "arithmetic.tst"
```



```
NextForge 1.2 by MB Maniax

        ret
.down:   add    hl,bc            ; from the last byte of each area
        dec    hl
        ex     de,hl
        add    hl,bc
        dec    hl
        ex     de,hl
        lddr
        ret

; Only Run Tests includes this file: the normal binary has no tests.
TESTCLUDE "arithmetic.tst"
```

The end of `arithmetic.asm`: `TESTCLUDE` takes the tests in only for Run Tests.

The file also holds `add8sat` (addition capped at 255) and `memmove` (moving memory, overlapping areas included). `arithmetic.tst` starts with `include "arithmetic.asm"` and has 12 tests; three of them for subtraction:

```

TESTCASE "sub16: 1234 - 234"
ld      hl,1234
ld      de,234
call    sub16
EXPECT  HL,1000
EXPECT  NC
ENDTEST

TESTCASE "sub16: borrow when DE is bigger"
ld      hl,1
ld      de,2
call    sub16
EXPECT  HL,-1          ; $FFFF
EXPECT  C
ENDTEST

TESTCASE "sub16: ignores the carry flag"
scf                                ; a carry left over by the caller
ld      hl,10
ld      de,3
call    sub16
EXPECT  HL,7
ENDTEST

```

And two for moving memory: a copy must not write past the end of its target, and a move into an overlapping area must not smear the data. Each test prepares its own input — a loop writes the values 1–8, LDIR fills the buffer — so the test shows what the routine works with. Whatever an earlier test wrote is undone before the next test anyway (chapter 27).

```

source:      ds      8          ; the tests fill it themselves
target:      ds      8
guard:       db      $aa       ; a copy must never reach it
...
                TESTCASE "memmove: copies 8 bytes"
                ld      hl,source ; prepare: source = 1,2,...,8
                ld      a,1
                ld      b,8
.fill:        ld      (hl),a     ; .fill is local to this test
                inc     hl
                inc     a
                djnz    .fill
                ld      hl,source
                ld      de,target
                ld      bc,8
                call    memmove
                EXPECT  (target),1,2,3,4,5,6,7,8
                EXPECT  (guard),$aa ; nothing written past the end
                ENDTST

                TESTCASE "memmove: overlap, move up by 2"
                ld      hl,letters ; prepare: buffer = ABCDE
                ld      de,buffer
                ld      bc,5
                ldir
                ld      hl,buffer
                ld      de,buffer+2
                ld      bc,5
                call    memmove
                EXPECT  (buffer),"ABABCDE" ; LDIR alone gives ABABABA
                ENDTST

```

1. **Normal assembly.** Open `arithmetic.asm` (**EXT** → **O**) and assemble it (**EXT** → **A**). The binary holds only the routines; `arithmetic.tst` is not even read.
2. **Running the tests.** Press **EXT** → **T** and confirm the main file (pressing **EXT** → **T** in `arithmetic.tst` works just as well, see chapter 26). After the assembly the tests run right in the report window: instead of the assembly steps it shows 12 rows with green ticks, each with its T-states on the right, and a green *All 12 tests passed*. **ENTER** closes the window.
3. **A bug in a routine.** In `sub16`, delete `or a`. A manual try easily misses this bug: as long as the carry happens to be clear, the result is right.
4. **Running again.** **EXT** → **T**. The window shows a red cross at test 08 and the reason *HL=\$0006, expected \$0007* — SBC also subtracted the carry left set by the calling code. The other tests ran too and passed; the result *1 of 12 tests failed*. is red.
5. **The result in the editor.** **ENTER** closes the window. The status row shows *1 of 12 tests failed*. in red, and in `arithmetic.tst` the **TESTCASE** line of test 8 is red, the others green. Put the cursor or the mouse on the red line: the status row shows the reason *FAIL sub16: ignores the carry flag: HL=\$0006, expected \$0007*. On a green line it shows the test's T-states in green, such as *PASS sub16: borrow when DE is bigger: 72 T*.
6. **A look in the debugger.** Run the tests again and press **D** in the window. The debugger runs the tests and stops at the check of test 8: the code cursor on `RST $30`, PC just after it, HL = 0006, the *Source* row names `EXPECT HL,7` and the bottom row reads *FAIL 8/12 ...* in red. **N** opens the line in the editor; **U** **ENTER** restarts the test so you can step through it (**S**).

7. **Fix.** Put `or a` back and press **EXT** → **T** again: all 12 tests pass and the lines turn green.

```

NextForge 1.2 by MB Maniax
Arithmetic and memory routines with their unit tests in arithmetic.tst.
Assemble (EXT-)>A) builds the routines alone; Run Tests (EXT-)>T) also
takes in the TESTCLUDE file at the end and runs its tests.
Each routine states what it takes, what it returns and what it keeps:
exact

Assembly report done
; HL =
add16:  ✓ 01 add16: 1000 + 234          68 T
        ✓ 02 add16: overflow sets carry  68 T
        ✓ 03 add16: below the limit      56 T
        ✓ 04 add16: stops at 255         67 T
        ✓ 05 sub16: 1234 - 234          72 T
        ✓ 06 sub16: equal numbers give zero 72 T
        ✓ 07 sub16: borrow when DE is bigger 72 T
        ✗ 08 sub16: ignores the carry flag: HL=$0006, expected $0007 596 T
        ✓ 09 memmove: copies 8 bytes      96 T
        ✓ 10 memmove: zero length copies nothing 404 T
        ✓ 11 memmove: overlap, move up by 2 357 T
        ✓ 12 memmove: overlap, move down by 2

Result: 1 of 12 tests failed.

UP/DOWN/WHEEL scroll  D debugger  ENTER/ESC/CLICK close
; Copi
; BC = 0 copies nothing (a plain LDIR would copy 65536 bytes).
memmove:  ld    a,b
          or    c
          ret   z
          push  hl

```

Step 4: test 8 failed with its reason, the others passed.

```

Debugger Run Breakpoints View Help PAUSED before instruction PC $8193
[5 Step] [Q Over] [C Continue] [T Run to] [O Out] [U Reset] [H Help] [Z Slow]
Focus: CODE Origin $0000 Entry $8184 Size $02A6
Disassembly > cursor breakpoint Stack Registers
> 818E F7 RST $30 1FFC 028A AF 0002 AF' 0044
818F 54 LD D,H 1FFE 0000 BC 0000 BC' 0000
8190 86 ADD A,(HL) 2000 0000 DE 0003 DE' 0000
8191 07 RLC A 2002 0000 HL 0006 HL' 0000
8192 00 NOP 2004 0000 IX 0000 SP 1FFC
8193 C9 RET 2006 0000 IX 0000 PC 8193
8194 60 LD L,L 2008 0000 I/R 00/22 IM1 EI
8195 65 LD H,L 200A 0000
8196 60 LD L,L 200C 0000 S Z H P/V N C
8197 60 LD L,L 200E 0000 0 0 0 0 1 0
8198 6F LD L,A 2010 0000
8199 76 HALT 2012 0000 MMU slots 0-7 (pages)
819A 65 LD H,L 2014 0000 -- 0F 0A 0B 04 05 00 01
819B 3A2053 LD A,($6320) 2016 0000 T +66 =66
819C 6F LD L,A 2018 0000 Mode SLOW trace
819D 70 LD (HL),B 201A 0000 I/O none
819E 69 LD L,C 201C 0000 Watch off (W)
81A0 69 LD L,C 201E 0000
Memory hex / ASCII Memory at registers
8020 00 00 00 00 00 00 00 00 ..... HL 0005 00 00 C3 AB 02 00 00 00
8028 00 00 00 00 00 00 00 00 ..... DE 0003 00 00 00 00 00 C3 AB 02
8030 AA 41 42 43 44 45 00 00 ,ABCDE.. BC 0000 C3 AB 02 00 00 00 00 00
8038 00 00 00 00 00 00 00 00 ..... IX 0000 C3 AB 02 00 00 00 00 00
8040 64 31 36 3A 20 31 30 30 d16: 100 IY 0000 C3 AB 02 00 00 00 00 00
8048 30 20 2B 20 32 33 34 00 0 + 234. SP 1FFC 8A 02 00 00 00 00 00
8050 21 E8 03 11 EA 00 CD 00 !..... PC 8193 C9 6D 65 6D 6D 6F 76 65
Source: line $0055 /project/arithmetic.tst
Breakpoints: none
✗ 08 sub16: ignores the carry flag: HL=$0006, expected $0007

```

Step 6: the debugger at the check of test 8; the cursor on the check, PC after it, HL = 0006.

```

; Copies BC bytes from HL to DE; the two areas may overlap.
; BC = 0 copies nothing (a plain LDIR would copy 65536 bytes).
memmove:  ld    a,b
          or    c
          ret   z
          push  hl
1 of 12 tests failed.

```

Step 5: the summary in the editor's status row after the window closes.

TRY ANOTHER BUG

Delete the line `jr c, .down` in `memmove`, so the routine always copies forwards with LDIR. Test 11 `memmove: overlap, move up by 2` then reports `($8039)=$41, expected $43`: instead of `ABABCDE` the buffer holds `ABABABA`.

26. Test directives

DIRECTIVE	MEANING
TESTCASE "name"	Starts a test. The name is optional; it is stored in the output just before the test's code and appears in the results. Tests cannot be nested.
ENDTEST	Ends a test and emits <code>RET</code> . Every TESTCASE needs its ENDTTEST.
EXPECT reg,value	The register or pair must hold the value. The value is an expression; -128 to 255 for byte registers.
EXPECT cc	The flag condition must hold: NZ, Z, NC, C, PO, PE, P or M — as in <code>JP cc</code> .
EXPECT (address),bytes	Memory from the address must contain the bytes, written as in DB: numbers, characters and strings separated by commas, at most 255 bytes.
EXPECT CYCLES op n	The T-states since the test began or since the previous CYCLES check must compare <code>= <> < <= > >=</code> with n (up to 65535), for example <code>EXPECT CYCLES <= 300</code> . A test with this check is always traced slowly, even when the debugger is set to fast runs (chapter 27); the checks themselves take no T-states.
STUB RST n,label	In the debugger, RST n calls the routine <code>label</code> instead of ROM or esxDOS; n is <code>\$08</code> , <code>\$10</code> , <code>\$18</code> , <code>\$20</code> , <code>\$28</code> or <code>\$38</code> . It emits nothing (see Replacing ROM and esxDOS below).
MOCK routine,fake	Until the test ends, every call of <code>routine</code> jumps to the fake <code>fake</code> ; then the routine is back (see Faking a routine of your own below).
TESTCLUDE "file"	The same as INCLUDE for Run Tests; skipped by every other assembly. It belongs in the project's main file.

Registers and conditions

GROUP	NAMES
Bytes	A, F, B, C, D, E, H, L, IXH, IXL, IYH, IYL, I
Pairs	AF, BC, DE, HL, IX, IY, SP
Alternate	AF'; check the other alternate pairs after <code>EXX</code>
Conditions	NZ, Z, NC, C, PO, PE, P, M

```
EXPECT A,'x'           ; a character
EXPECT HL,buffer+4     ; an expression with a label
EXPECT DE,-1           ; $FFFF
EXPECT NC              ; no carry
EXPECT (result),$34,$12 ; the word $1234 in memory
EXPECT (text),"OK",0    ; a string with its zero
```

Connecting tests to a project

WAY	HOW	WHEN IT FITS
TESTCLUE	The project's main file contains <code>TESTCLUE</code> <code>"module.tst"</code> ; the <code>.tst</code> file starts with <code>INCLUDE</code> <code>"module.asm"</code> and the tests follow. <code>EXT → T</code> works in either file.	The tests belong to the project: a normal assembly leaves them out, Run Tests runs all of them. Example: <code>arithmetic</code> .
Standalone file	The test file has its own <code>ORG</code> , takes the routines in with <code>INCLUDE</code> and the tests follow. Press <code>EXT → T</code> in the test file.	A routine library without a program of its own. Example: <code>strings_test.asm</code> .

The link works both ways because **the main source is never included again**: an `INCLUDE` or `TESTCLUE` of the file being assembled is skipped (name and directory, letters in any case). Press `EXT → T` in `arithmetic.tst` and its `INCLUDE` brings in the routines while their `TESTCLUE` is skipped; in `arithmetic.asm` the `TESTCLUE` brings in the tests and their `INCLUDE` is skipped.

The NextZXOS browser offers `.tst` files just like `.asm`; the extension is only a convention, the assembler takes any source.

How a check is assembled

`EXPECT` emits an `RST $30` instruction (`$F7`), the character `T`, the kind of check and the expected data. A byte register check takes 4 bytes, a pair 5, a condition 3 and memory 6 plus the byte count; `MOCK` is the same kind of record, 7 bytes long. The debugger's disassembly therefore shows `RST $30` followed by meaningless instructions, while the *Source* row names the `EXPECT` line.

Replacing ROM and esxDOS: STUB

The debugger has no ROM and no esxDOS, so a routine that calls `RST $10` (print a character) or `RST $08` (an esxDOS service) stops its test. `STUB RST n,label` makes `RST n` call your routine `label` in the debugger instead: the return address goes on the stack as with a real `RST`, and the replacement's `RET` returns after the `RST`. The replacement can record the call, and the test then checks what the routine printed. The example `print.asm` and `print.tst` in `tests/fixtures/unittest`:

```
                STUB    RST $10,fake_print

; A = character: appended to output. Keeps every register but F.
fake_print:     push    hl
                ld      hl,(out_ptr)
                ld      (hl),a
                inc     hl
                ld      (out_ptr),hl
                pop     hl
                ret
out_ptr:        dw      output
output:         ds      16

                TESTCASE "print: HELLO"
                ld      hl,hello
                call    print                ; RST $10 for each character
                EXPECT  (output),"HELLO"
                ENDTEST
```

- STUB emits nothing and does nothing outside the debugger. It applies to the whole assembly, also outside TESTCASE, while stepping and in fast runs alike: the RST vector in the debugger's kernel jumps straight to the replacement.
- `RST $30` belongs to EXPECT checks and breakpoints, and `RST $00` would map divMMC in a fast run; neither can be replaced.
- The replacement must lie in `$2000-$FFFF`. The last STUB of an RST applies.
- `RST $08` is followed by the esxDOS service number: the replacement reads it through the return address and returns after it (`ex (sp),h1, ld a,(h1), inc h1, ex (sp),h1`).
- A replaced RST counts 11 T-states, as on the Z80; the replacement's T-states are added.

Faking a routine of your own: MOCK

The routine under test often calls another one that a test should not run: reading the keyboard, disk access, a random number. `MOCK routine,fake` writes `JP fake` over the start of `routine`, so every call of it lands in `fake` until the test ends. When the test ends, the debugger puts the original three bytes back. The example `player.asm` and `player.tst` in `tests/fixtures/unittest` fakes the keyboard this way; the test itself decides which keys are held:

```
; The fake keyboard: returns fake_held and counts the calls.
fake_keys:    ld      h1,fake_calls
              inc     (h1)
              ld      a,(fake_held)
              ret
fake_held:    db      0
fake_calls:   db      0

TESTCASE "P moves right"
MOCK      read_keys,fake_keys
ld        a,%01          ; P
ld        (fake_held),a
call      step            ; step calls read_keys
EXPECT    (player_x),17
EXPECT    (fake_calls),1 ; the keyboard was read once
ENDTEST
```

- `MOCK` is a record like `EXPECT`: the debugger executes it as one step, changes no register and counts no T-states.
- The jump applies from where `MOCK` stands in the test to the end of the test. Later tests see the original routine, even when memory is not reset for lack of RAM.
- The fake can record how often and with what it was called, and the test checks that with `EXPECT`. Such a counter starts from zero in each test, as memory is reset.
- Routine and fake lie in `$2000-$FFFF`. The jump overwrites the routine's first three bytes; a shorter routine also loses the start of the next one until the test ends. A ROM call with `CALL` cannot be faked this way; `STUB` covers `RST`.
- At most 16 fakes per test; repeating the same `MOCK` does not count again.

RULES

- At most 32 tests per assembly.
- EXPECT also works outside TESTCASE; the debugger evaluates it while stepping and tracing.
- Each TESTCASE starts a new scope for local labels: every test can have its own `.loop` or `.fill`.
- The directives work in macros, in REPT and in included files.
- `TEST` stays the Z80N instruction (`test n`), which is why a test block is called TESTCASE.
- Only NextForge knows the test directives. Keep routines that SjaSMPlus must also assemble in a file without tests and without TESTCLUE.

27. How tests run

The Run Tests window

Run Tests assembles the main source together with the TESTCLUDE files. When the assembly succeeds, it runs the tests in the background while the report window stays open; meanwhile the *Result* row shows which test is running. Every test starts like a fresh run of the program: registers zero, flags as after `XOR A`, interrupts enabled in IM 1, the default stack `$1FFC` with a return address, the initial paging and the program's memory as the first test found it. A test passed when it runs to the `RET` of its `ENDTEST` and all of its `EXPECT` checks held.

- **Every test runs.** One failed test does not stop the others.
- **Breakpoints are skipped** and stay set; a forgotten breakpoint in a routine does not stop the run.
- **Slow tracing is used** even if the debugger is set to fast runs: hardware is only simulated and the screen stays on the report.
- **BREAK** ends the run; the tests it did not reach stay unrun.

IN THE LIST	MEANING
Green tick · 01 add16: 1000 + 234 ... 68 T	The test passed; on the right are its T-states from its start to the <code>RET</code> of its <code>ENDTEST</code> , routines it calls included.
Red cross · 08 sub16: ...: HL=\$0006, expected \$0007	A check did not hold: the register, its actual and its expected value; for a condition <code>NZ</code> expected, <code>F=\$44</code> ; for memory the first differing byte (<code>\$8039</code>)= <code>\$41</code> , expected <code>\$43</code> .
Red cross · ... stopped: ...	The test stopped for the reason that follows: a ROM call (<code>RST</code> without a <code>STUB</code>), <code>HALT</code> after <code>DI</code> .
Red cross · ... cannot start: ...	The test's code lies below <code>\$2000</code> (owned by the debugger's kernel), or memory is short.
Question mark	The test did not run because <code>BREAK</code> ended the run.

Scroll the list with the arrows, Page Up / Page Down or the mouse wheel. The *Result* row is green (*All 12 tests passed.*) or red (*1 of 12 tests failed., Stopped by BREAK after 9 of 12 tests, 1 failed., No TESTCASE blocks in the assembled source.*). `ENTER`, `BREAK` or a click closes the window; the editor's status row then shows the same summary in the same colour. `D` closes the window and opens the debugger.

```
kes in the TESTCLUDE file at the end and runs its tests,
ch routine states what it takes, what it returns and what it keeps:
ac
Assembly report done

= ✓ 01 add16: 1000 + 234 68 T
6: ✓ 02 add16: overflow sets carry 68 T
✓ 03 add8sat: below the limit 56 T
✓ 04 add8sat: stops at 255 67 T
= ✓ 05 sub16: 1234 - 234 76 T
6a ✓ 06 sub16: equal numbers give zero 76 T
✓ 07 sub16: borrow when DE is bigger 76 T
✓ 08 sub16: ignores the carry flag 80 T
✓ 09 memmove: copies 8 bytes 596 T
= ✓ 10 memmove: zero length copies nothing 96 T
8 ✓ 11 memmove: overlap, move up by 2 404 T
6: ✓ 12 memmove: overlap, move down by 2 357 T

Result: All 12 tests passed.

UP/DOWN/WHEEL scroll D debugger ENTER/ESC/CLICK close
= 0 copies nothing (a plain LDIR would copy 65536 bytes).
```

All tests passed.

```
kes in the TESTCLUDE file at the end and runs its tests,
ch routine states what it takes, what it returns and what it keeps:
ac
Assembly report done

= ✓ 01 add16: 1000 + 234 68 T
6: ✓ 02 add16: overflow sets carry 68 T
✓ 03 add8sat: below the limit 56 T
✓ 04 add8sat: stops at 255 67 T
= ✓ 05 sub16: 1234 - 234 76 T
6a ✓ 06 sub16: equal numbers give zero 72 T
✓ 07 sub16: borrow when DE is bigger 72 T
X 08 sub16: ignores the carry flag: HL=$0006, expected $0007 596 T
= ✓ 09 memmove: copies 8 bytes 596 T
8 ? 10 memmove: zero length copies nothing
6: ? 11 memmove: overlap, move up by 2
? 12 memmove: overlap, move down by 2

Result: Stopped by BREAK after 9 of 12 tests, 1 failed.

UP/DOWN/WHEEL scroll D debugger ENTER/ESC/CLICK close
= 0 copies nothing (a plain LDIR would copy 65536 bytes).
```

BREAK during test 10: tests 10–12 did not run.

EVERY TEST STARTS WITH THE SAME MEMORY

When the first test starts, the debugger copies every page of the program and puts the copies back before each further test; pages a test mapped first are released. Whatever one test writes, the next one does not see, and after the run memory is as it was before. The copies need free pages in the memory shared with the documents. When it is short, the tests share memory and the summary ends with (*memory not reset: low RAM*); closing a file you do not need helps. Pages the program first maps in a fast run in the debugger are unknown to the debugger and are not restored.

Colours in the editor

After each run the editor colours the `TESTCASE` lines in every open file that took part in the assembly — the included `.tst` file too:

LINE PAPER	MEANING
Green	The test passed.
Red	The test failed.
Unchanged	The test did not run (BREAK), or did not finish in the debugger.

```
NextForge 1.2 by MB Maniax
TESTCASE "sub16: ignores the carry flag"
scf                                ; a carry left over by the caller
ld    hl,10
ld    de,3
call  sub16
EXPECT HL,7
ENDTEST

; --- moving memory -----
TESTCASE "memmove: copies 8 bytes"
ld    hl,source                    ; prepare: source = 1,2,...,8
ld    a,1
ld    b,8
.fill: ld    (hl),a                ; .fill is local to this test
inc    hl
inc    a
djnz  .fill
ld    hl,source
ld    de,target
ld    bc,8
call  memmove
EXPECT (target),1,2,3,4,5,6,7,8
EXPECT (guard),$aa                ; nothing written past the end
ENDTEST

TESTCASE "memmove: zero length copies nothing"
ld    a,$ee
ld    (target),a                    ; a marker that must survive
FAIL sub16: ignores the carry flag: HL=$0006, expected $0007
```

After a run ended by BREAK: test 8 failed, test 9 passed, test 10 did not run. The cursor is on the line of test 8, so the status row shows why it failed.

On a red line the status row shows **the reason for the failure** in red, on a green line **the test's T-states** in green (*PASS sub16: borrow when DE is bigger: 72 T*; only the name after a fast run in the debugger) — while the cursor is on the line or the mouse points at it (the pointer comes first). Once neither is on a coloured line, the normal status row returns. The result is found by the test's quoted name, so it survives edits that move lines; for a renamed test the status row says that the result belongs to an earlier run.

The colour belongs to the last run and is not saved in the file. The next run renews it; if you edit the source after assembling, the lines cannot be found and the colours of the previous run remain. A breakpoint on the line takes precedence (dark red).

A test build saves nothing

Run Tests writes no output file (neither binary nor NEX): the program with its tests stays in memory, and the output of an earlier normal assembly is not overwritten. A failed test can never leave anything on the card that looks like a finished program; only **Assemble** (EXT → A) produces the output file.

Tests in the debugger

D in the Run Tests window or **EXT → T** inside the debugger runs the same tests in the debugger. Such a run differs from the one in the window:

- it stops at the **first** failure or other stop and the debugger stays there;
- **breakpoints count** — a test stops at them with *Test ... stopped: Stopped at a breakpoint* and you can step through it;
- it runs in the debugger's current mode: slow tracing, or a fast native run (**Z**) that stops at each check and shows the program's screen. A test with **EXPECT CYCLES** is always traced slowly, because a fast run counts no T-states; the chosen mode returns after the run.

The debugger's bottom row shows the result in green (*All 12 tests passed.*) or red (*FAIL 8/12 name: HL=\$0006, expected \$0007*). After a failed test the code cursor is on the check, PC on the instruction after it, and registers, flags and memory are as they were at the check:

- **C** finishes this test; its further checks are evaluated too.
- **U ENTER** runs this test again from its start, step by step if you like (**S**). Memory and fakes (MOCK) stay as the test left them.
- **N** opens the check's source line in the editor.
- **EXT → T** runs all tests again without a new assembly. The stopped test's fakes are undone first; the rest of memory stays as the test left it, and a new assembly gives a clean program image.

Checks while stepping

Even when you step by hand (S, O, C), each **EXPECT** and **MOCK** is one step. A check that holds is simply skipped; one that fails shows *EXPECT failed: ...* in red and PC moves past it, so you can carry on.

28. Tips and limits

Tips

- Test small routines with clear inputs and outputs; describe in the routine's comment what it takes, returns and keeps.
- Name each test after what it checks — you see the name when it fails.
- Try the edge cases: zero, 255 and 256, `$FFFF`, overflow, an empty string, a zero length, the end of a buffer.
- Check what must not change too: kept registers (`EXPECT DE,234`) and a guard byte after a buffer.
- Check the flags a caller relies on (`EXPECT C` on overflow) and test both branches of a condition.
- After fixing a bug, add a test that would have caught it.
- Test a routine that calls ROM or esxDOS through RST with a `STUB`: the replacement records what the routine printed or called, and the test checks it.
- Test a routine that depends on the keyboard, timing or another routine with `MOCK`: the fake returns exactly the values the test needs.

Limits

- At most 32 tests per assembly and 255 bytes per memory check.
- Test code must lie in `$2000-$FFFF` on a page the program maps at its start; in projects with `SAVENEX`, put `TESTCLUDE` where the main code is assembled.
- The debugger provides no ROM or esxDOS calls through RST; a routine that uses them stops its test unless a `STUB` replaces the RST. A ROM call with `CALL` cannot be replaced.
- In the Run Tests window tests always run with slow tracing; run long tests in the debugger with a fast run (Z, D). Tests with `EXPECT CYCLES` are traced slowly there too.
- Resetting memory between tests needs free pages for a copy of the program (chapter 27).
- `EXPECT` compares the state after a whole instruction and does not watch the bus; input and output can only be checked indirectly (the debugger's I/O row).
- Outside the debugger, `EXPECT` is a plain `RST $30` instruction: add tests to a program only through `TESTCLUDE`.

ASSEMBLY ERROR	CAUSE
bad operand at TESTCASE	A nested TESTCASE, or a name without quotes.
bad operand at ENDTEST or at the end of the source	ENDTEST without TESTCASE, or TESTCASE without ENDTEST.
bad operand at EXPECT	Unknown register or condition, a missing value or no comma after the address.
too many operands at EXPECT	More than one value for a register; byte lists belong to memory checks.
value/address out of range	A byte register value outside -128..255, or more than 32 tests.
missing operand at TESTCLUDE	The file name is missing.
bad operand at STUB	The word RST or the replacement's label is missing.
bad operand at MOCK	The routine or the fake is missing.
value/address out of range at MOCK	The routine or the fake lies below <code>\$2000</code> .
value/address out of range at STUB	<code>RST \$00</code> or <code>\$30</code> , a number that is not an RST vector, or a replacement below <code>\$2000</code> .
undefined symbol in a test	A test file assembled as the main source does not take in the routines under test: start it with <code>INCLUDE "module.asm"</code> .

ASSEMBLY ERROR	CAUSE
file not found opening ...	The TESTCLUDE file (looked up only by Run Tests) or an INCLUDE file does not exist.