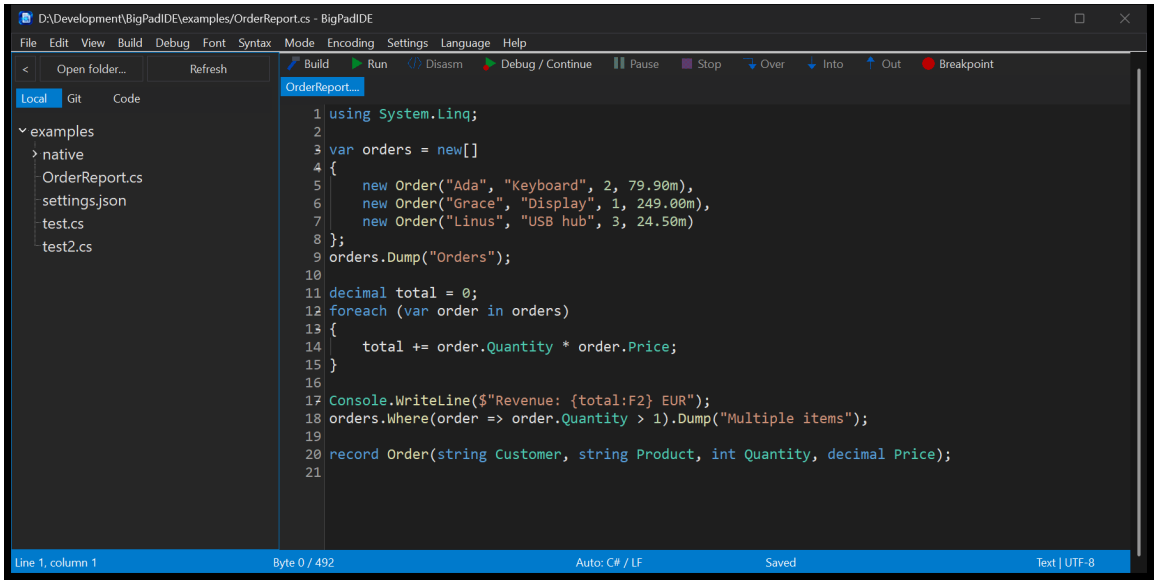
BigPadIDE

The complete user manual

From your first document to debugging, Git and IL/C# inspection



Native Windows workspace with explorer and C# editor.

English | All four interface styles

Developer: **Rudolf Stepan**
rstepan123@gmail.com

Edition 14 November 2026 | Version 1.2.4

With practical workflows, screenshots, shortcuts and troubleshooting.

Preface

BigPadIDE is a practical addition to a developer's toolbox. Many tasks start with a small question: what is in this file, which bytes are at this offset, what does this method look like as IL, or which change is actually in a Git diff?

The app brings editing, small experiments, debugging and inspection into one workspace. You choose which tools fit your task and which dedicated programs you want to use alongside them. Comfortable fonts, colors, layout and storage choices help the workspace fit your way of working.

This manual explains the tools, walks through real workflows and states the practical limits that matter. Use the searchable local help for a quick answer, or follow a chapter here from setup to a complete result.

Contents

Click a chapter, section or page number to navigate. You can also use your PDF reader's bookmarks.

Preface 2

1. Welcome and your first result 9

1.1 Your first C# program . . . 9

2. Installation and working environment 10

2.1 Check the installation . . . 10

3. Workspace, themes and views 11

3.1 Themes and hex view . . . 12

3.2 Explorer colors and refresh . . . 15

3.3 Folder explorer font size . . . 15

3.4 Split-screen editing . . . 15

3.5 Open views as tabs or windows . . . 15

4. Edit files and save safely 18

4.1 File tabs and unsaved changes . . . 18

4.2 External file changes . . . 18

4.3 Create files and folders in the explorer . . . 19

4.4 Rename and move files and folders . . . 19

4.5 Large files and long lines . . . 20

5. Navigation, finding and replacing 21

5.1 Go to line or offset . . . 21

5.2 Quick open . . . 21

5.3 Project search and regular expressions . . . 21

6. Write, build and run C# 24

6.1 Files, folders and code . . . 24

6.2 Compiler diagnostics . . . 24

7. Solutions, active projects and C# tools 26

7.1 Keep the project selection synchronized . . . 26

7.2 C# project tools . . . 26

8. Output, results and launch profiles 30

8.1 Tables and program output . . . 30

8.2 Build/run profiles and input . . . 30

9. Debug C# programs 33

9.1 Breakpoints and watches . . . 33

9.2 Runtime hover values, object browser and changing variables . . . 35

10. Native projects: C, C++ and assembly 38

10.1 Building and running ASM, C and C++ . . . 38

11. Native debugging 39

11.1 Native debugging . . . 39

12. Hex editor and byte tools 41

12.1 Hex tools: byte search, values and export . . . 41

13. Native disassembly and PE/ELF 42

13.1 Disassembler . . . 42

13.2 PE/ELF disassembly and branch targets . . . 42

14. .NET assemblies: IL and reconstructed C# 44

14.1 .NET assemblies: IL and reconstructed C# . . . 44

15. Images, frames and pixel values 46

15.1 Viewing images . . . 46

15.2 Image controls . . . 46

16. Git, diffs and collaboration **48**

16.1 Git changes and diffs . . . 48

16.2 Fetch, Pull and Push . . . 48

16.3 Manage branches . . . 49

16.4 Work with individual diff hunks . . . 51

16.5 Resolve Git conflicts . . . 52

16.6 Automatic Git refresh . . . 53

17. Sessions and recovery **54**

17.1 Session restoration . . . 54

17.2 Crash recovery . . . 54

17.3 Session capacity when opening files . . . 55

17.4 Remember window and folder width . . . 55

18. Practical workshops **56**

18.1 Inspect and change a file safely . . . 56

18.2 A complete build/debug cycle . . . 56

18.3 Review a Git change . . . 56

18.4 Make a portable personal profile . . . 56

19. Troubleshooting **57**

19.1 Report a reproducible problem . . . 57

20. Keyboard and menu reference **58**

21. Limits and stored data **60**

22. Glossary and further help **61**

23. Trial, purchase and product keys **62**

23.1 Trial and purchase entitlement . . . 62

23.2 Activate a product key . . . 62

23.3 Move settings or ask for help . . . 62

24. Make the workspace your own 63

24.1 Interface style and horizontal menus . . . 63

24.2 Independent fonts and application colors . . . 63

24.3 Ctrl+mouse wheel and panel size . . . 64

24.4 Import and export the complete settings as JSON . . . 64

24.5 Piece table or complete RAM buffers . . . 64

24.6 Build a comfortable personal profile . . . 65

25. Toolsets, compilers and build configuration 66

25.1 Build configuration and compiler options . . . 66

25.2 Installer . . . 67

26. Semantic code assistance and local AI 68

26.1 C# completion and gray suggestions . . . 68

26.2 Set up local AI . . . 68

26.3 Use inline suggestions and chat . . . 69

27. Architecture and visual diagrams 70

27.1 Analyze saved source code . . . 70

27.2 Draw UML, flows and system landscapes . . . 70

27.3 Save and exchange diagrams . . . 71

27.4 Explorer and assembly preview . . . 71

28. Licenses and release status 72

Screenshot reference: more views 73

List of figures

Figure 1. Modern dark workspace 11

Figure 2. Modern light workspace 11

Figure 3. Sepia with warm editor and explorer colors 12

Figure 4. Midnight with dark blue working surfaces 13

Figure 5. JSON in the dark theme with indentation guides 14

Figure 6. The same file in the editable hex view 14

Figure 7. Assembly viewer in the app 16

Figure 8. Separate diff window 16

Figure 9. Original snapshot progress 19

Figure 10. Renamed tab retaining its unsaved edits 20

Figure 11. Project regex search with filters and exact result positions 23

Figure 12. The search window follows the Midnight theme 23

Figure 13. BigPadIDE with a folder tree and C# syntax highlighting 24

Figure 14. Red source underline and the corresponding compiler diagnostic 25

Figure 15. Native project workspace 26

Figure 16. Reference finder with filter and source preview. 27

Figure 17. References stay visible in a dedicated output panel tab. 28

Figure 18. Code map with resolved method calls. 28

Figure 19. Semantic code action with before/after context and its exact text replacement. 29

Figure 20. Order objects displayed as rows and columns in Results 30

Figure 21. Run profile with working directory, arguments and environment 32

Figure 22. Running program with a visible prompt and input field 32

Figure 23. Enlarged debugger panel with actual session data 33

Figure 24. A real breakpoint, call stack, locals and evaluated expression 34

Figure 25. Conditional breakpoint dialog 35

Figure 26. Expanded collection and nested object members 36

Figure 27. Actual C# runtime value above a source variable 36

Figure 28. Dialog for changing a real paused variable 37

Figure 29. Native breakpoint, stack and evaluated watch 40

Figure 30. Byte interpretation, exact search and range export 41

Figure 31. PE/ELF 43

Figure 32. Symbols 43

Figure 33. Reconstructed C# and browsable types 45

Figure 34. IL view of the same type 45

Figure 35. Zoom, pixel values and the second TIFF page 47

Figure 36. Git synchronization dialog 49

Figure 37. Local and remote branch management 50

Figure 38. Hunk navigation and Git actions 52

Figure 39. Base, ours and incoming conflict versions 53

Figure 40. Independent fonts and colors dialog 63

Figure 41. Workspace after choosing complete RAM buffers 65

Figure 42. Toolsets and shared tool paths 67

Figure 43. Semantic completion and a Tab suggestion 68

Figure 44. Native chat window 69

Figure 45. Architecture analyzed from saved source 70

Figure 46. Visual diagram designer 71

Figure 47. Read third-party notices offline 72

Figure 48. C# breakpoint with stack and variables 73

Figure 49. Debugger using the Midnight theme 74

1. Welcome and your first result

BigPadIDE combines a native Windows editor, C# tools, native build/debug workflows, Git, binary inspection and image viewing in one workspace. This manual describes version 1.2.4, feature edition 14 November 2026. Older installation packages may contain an earlier feature set. Screenshots were automatically captured from the actual app on the same day. All screenshots in this edition show the English program interface.

Press **F1** to open help in the language selected under **Language**. Both local help pages work offline and link to the corresponding PDF manual. Choose a chapter in the sidebar or search for a task, shortcut or error message.

1.1 Your first C# program

1. Choose **Language** → **English**, then **File** → **Open folder** and select your workspace.
2. Press **Ctrl+N** and enter the code below.
3. Save it as `Hello.cs` using **Ctrl+Shift+S**.
4. Press **Ctrl+J** to show the output panel and **F6** to build and run.
5. Read **Program output**. If compilation fails, check **Build output** first.

```
using System;
Console.WriteLine("Hello from BigPadIDE!");
```

Single-file execution uses a temporary console project. For package references, multiple source files and libraries, use a normal SDK project.

2. Installation and working environment

The EXE installer supports the current Windows user or all users. The shared `BigPadIDE.exe` offers Classic, Workbench (Retro), Modern - Light and Modern - Dark. The full distribution includes local help, PDF manuals, the C# helper, a private .NET 10 SDK, NetCoreDbg and CodeLLDB. Keep the supporting folders with the executable. The private SDK does not modify the system PATH or replace other .NET installations.

Native compilers are separate tools. Optional GCC/ARM downloads are available in the EXE installer; installed MSVC and Intel tools can be imported into named toolsets. See chapter 25. Extra workloads and NuGet packages depend on your projects.

Store packages install into a read-only location. Copy bundled examples into your own writable project folder before building. Configure external compilers under **Settings → Toolsets and tool paths**. Availability and price depend on the published distribution. Generating this manual does not update an older installer.

2.1 Check the installation

Open and save a small text file, run the example in chapter 1, then build a copy of a native example. For Git, open an existing repository and switch to the Git pane. After changing the system PATH, restart the app so it inherits the new environment. Explicit tool paths can be changed directly in Settings.

3. Workspace, themes and views

The workspace contains menus, a toolbar, the optional explorer, document tabs, editor areas, the output panel and a status bar. Focus matters: F5 refreshes the explorer when it has focus, and starts or continues debugging in the editor. Choose **Settings** → **Interface / style** to change the complete interface while keeping open documents. Chapter 24 explains independent fonts, colors and zoom.

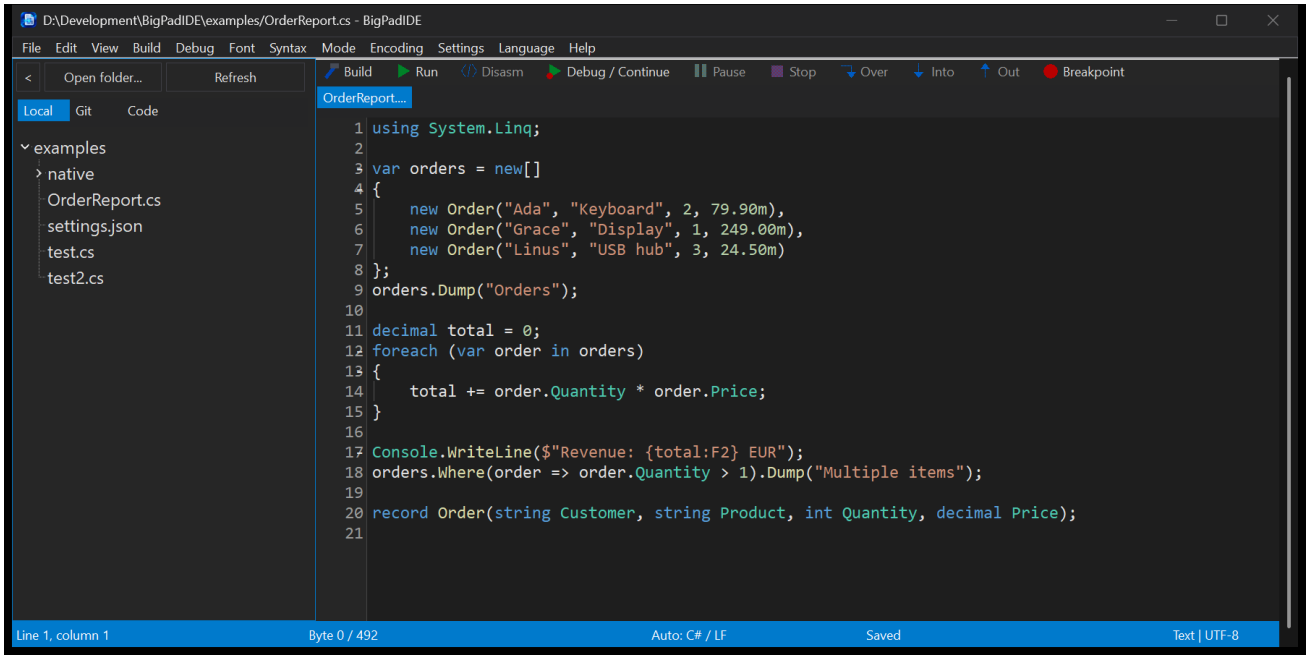


Figure 1. Modern dark workspace

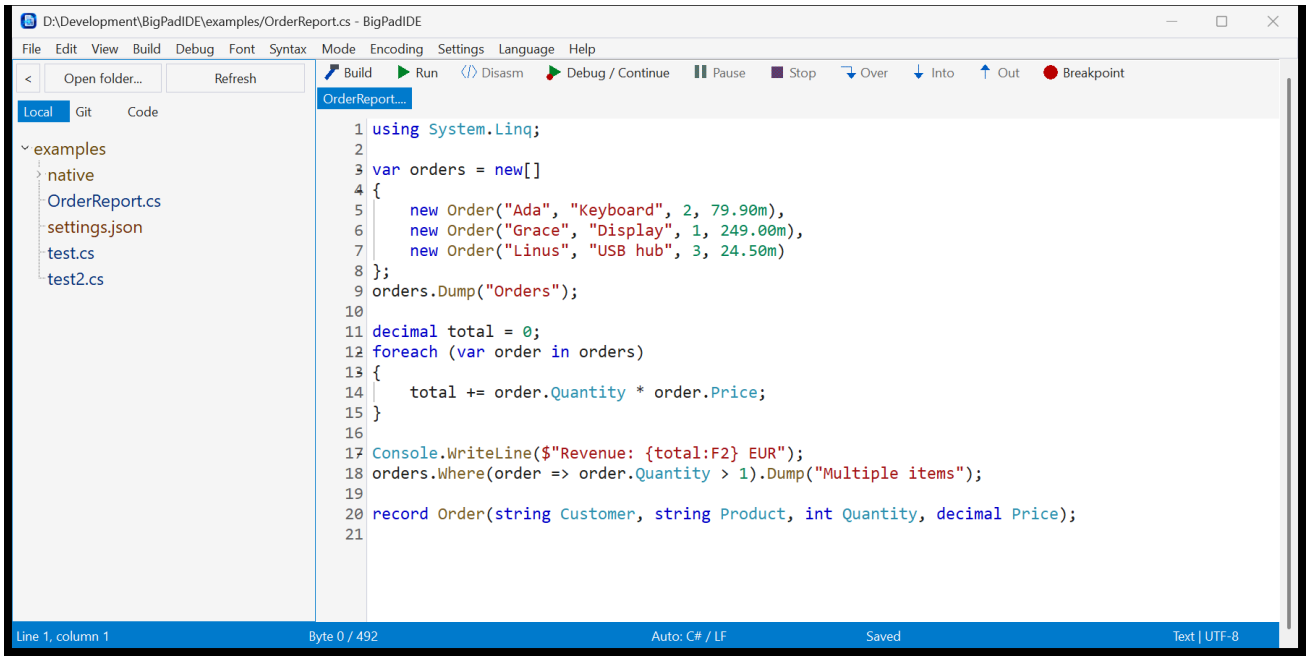


Figure 2. Modern light workspace

3.1 Themes and hex view

Use **View** → **Theme** to choose **Light**, **Dark**, **Sepia** or **Midnight**. Colors change immediately for the text/hex editor, line numbers, explorer and output/debugger contents. Open documents and undo history are preserved. The choice is saved with settings and restored at the next start. If automatic saving is disabled, use **Settings** → **Save settings now**. **Restore defaults** selects Light.

Interface style controls frames and controls. Themes select the working-surface palette; the custom colors described in chapter 24 can override its entries.

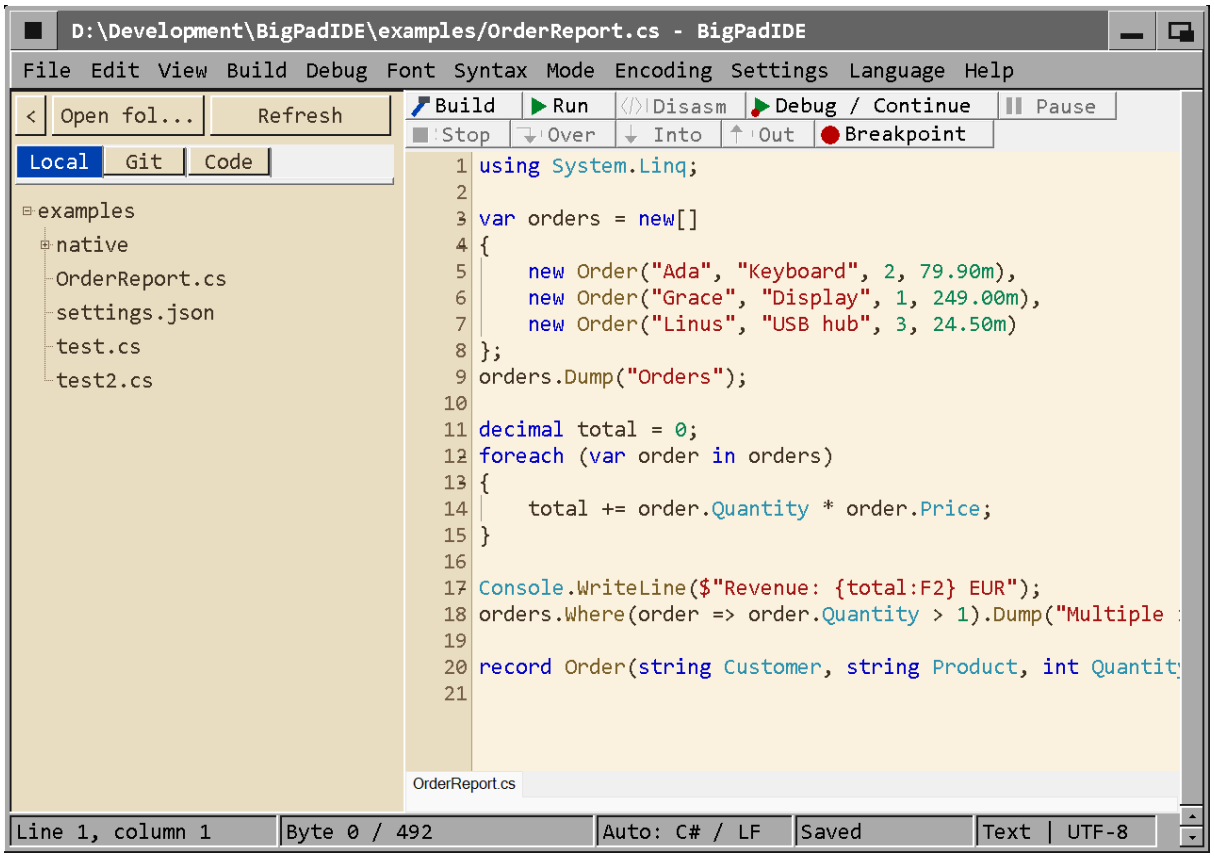


Figure 3. Sepia with warm editor and explorer colors

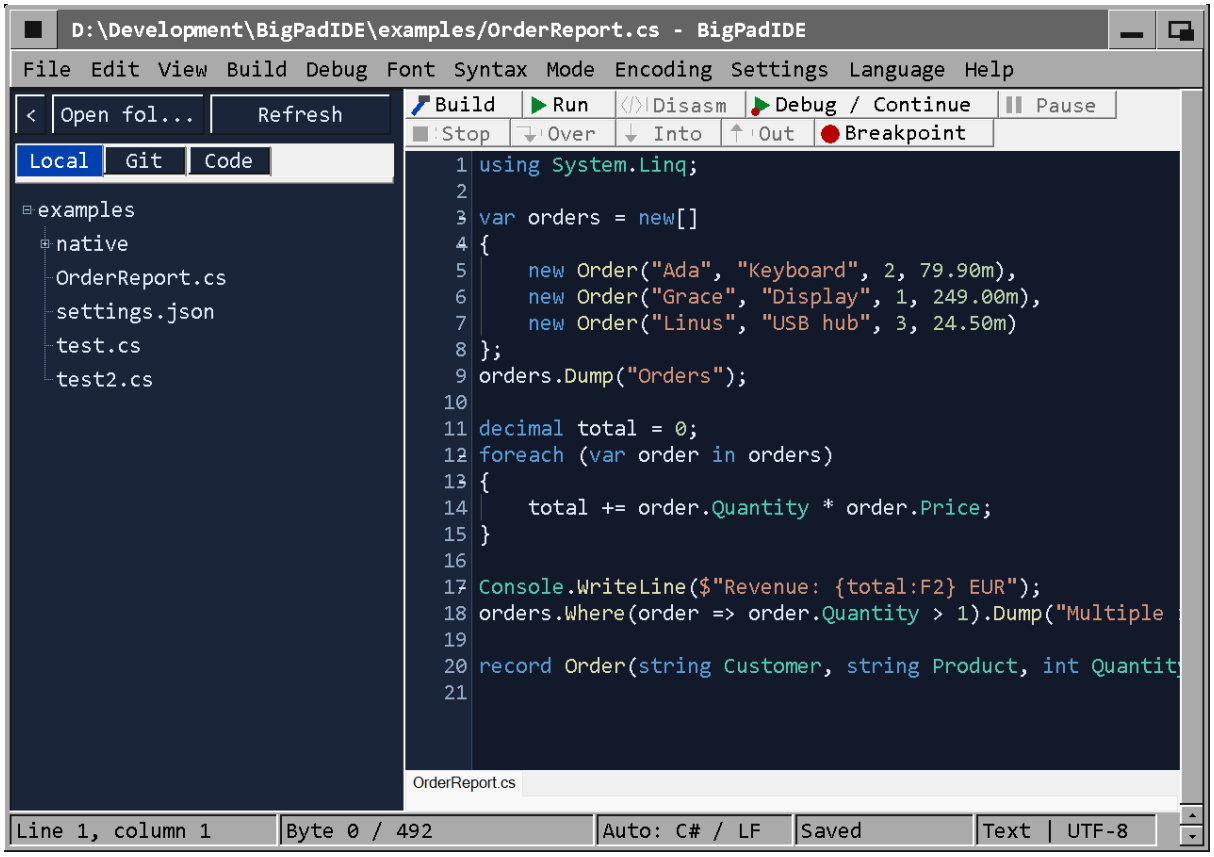


Figure 4. Midnight with dark blue working surfaces

Choose a theme and word wrap from **View**. **Mode** → **Hex** exposes byte offsets, hexadecimal values and their text representation.

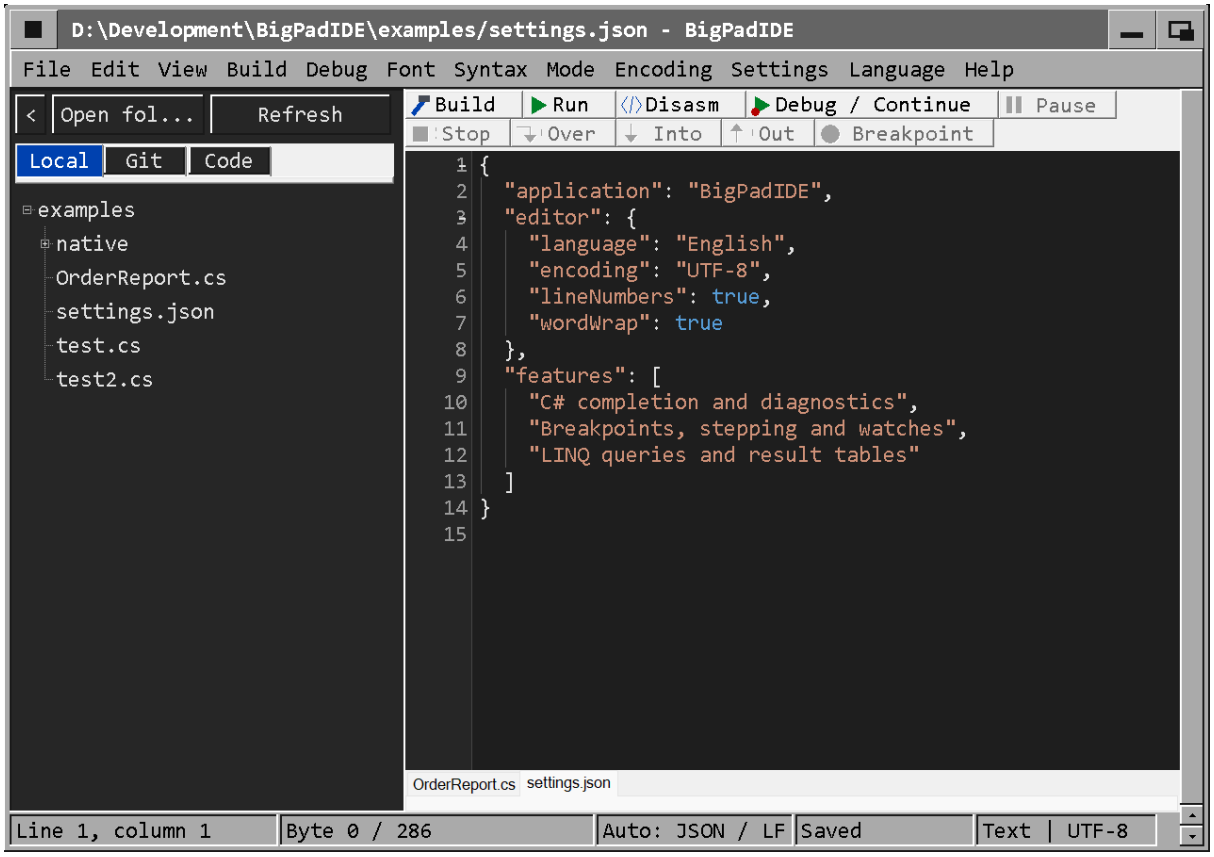


Figure 5. JSON in the dark theme with indentation guides

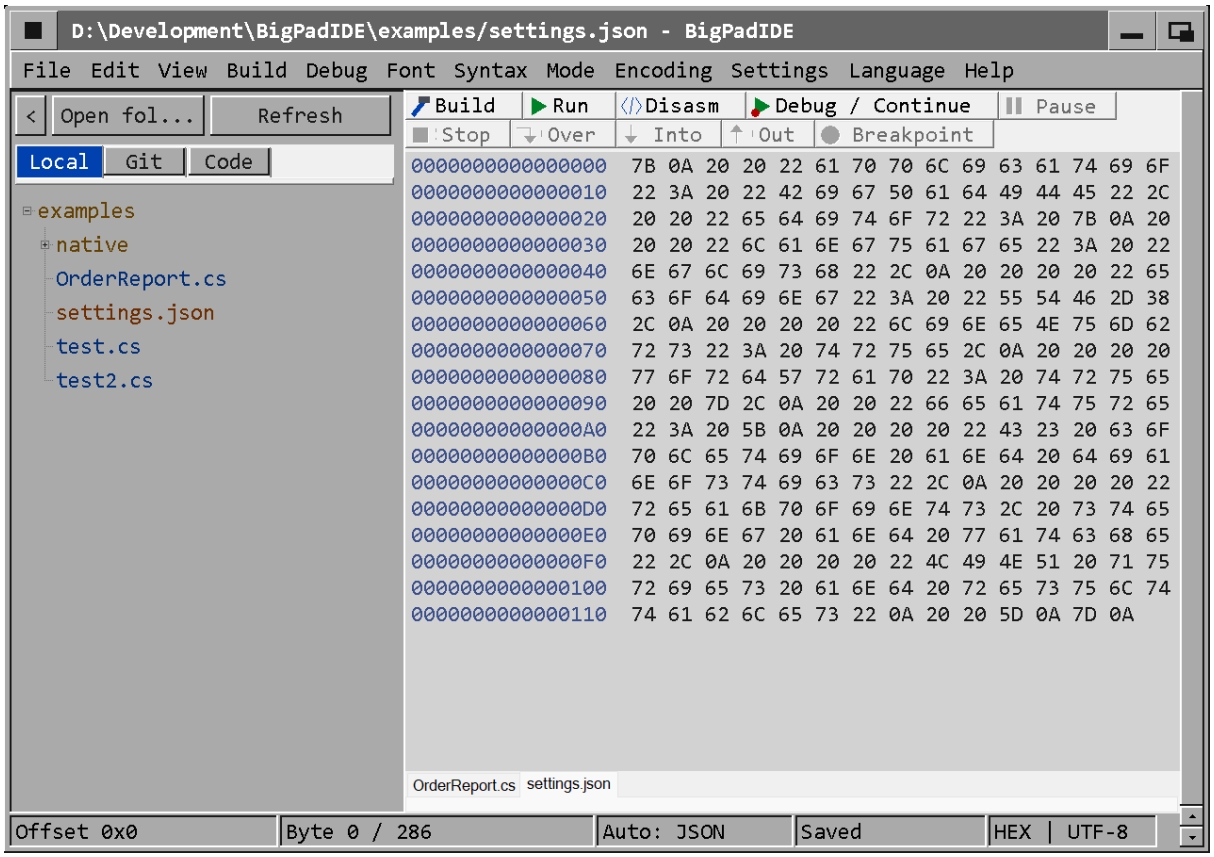


Figure 6. The same file in the editable hex view

3.2 Explorer colors and refresh

The explorer colors filenames by extension: source code in blue, scripts in purple, configuration files in brown, documents in teal, images in green, archives in burgundy, binaries in red and project files in indigo. Folders use a dark gold color; unknown file types remain neutral. Selected entries retain white text on blue.

Refresh the folder structure with **Refresh** at the top of the explorer, **F5** while the explorer has focus, or **Ctrl+R**. Newly created and deleted files are reflected while expanded folders and selection are preserved, including solution views. Outside the explorer, F5 continues to start the C# debugger.

3.3 Folder explorer font size

Use **Font** → **Folder explorer font size** to choose 10, 12, 14, 16, 18, 20, 24 or 28 pt, independently of the editor. The default is 12 pt. Row height and the header adapt; large fonts use two header rows when space is limited. Selection and expanded folders are retained.

The size follows the existing automatic-settings option. With automatic saving disabled, use **Settings** → **Save settings now**. **Restore defaults** returns the explorer to 12 pt. Editor zoom does not change this setting.

3.4 Split-screen editing

Use **View** → **Split vertically (left/right)** or **Split horizontally (top/bottom)** to split the active editor pane. Both directions can be combined and nested. Drag the dividers to resize panes.

Click a pane or use **Next pane** (F7) to activate it. A blue heading identifies the active pane. Selecting a hidden tab displays it in the active pane; selecting a tab already visible focuses its existing pane. A new split uses a previously hidden tab or creates an empty document. Each document retains its edits and scroll position. The scrollbar controls the active pane; the mouse wheel activates and scrolls the pane beneath the pointer.

Text, images, hex and disassembly views can be displayed together. **Close current pane** removes only the pane and keeps its tabs open. **Restore single view** expands the active pane to the whole workspace. Closing a file also removes its pane when multiple panes are visible. Split layouts last for the current session.

3.5 Open views as tabs or windows

Settings → **Open IL / C#** and **Settings** → **Open diff** independently select **As a tab in the app** or **Separate window**. Preferences are saved with the usual application settings, for the shared application. Defaults are a separate assembly viewer and in-app diff tabs. Choices apply to newly opened views; close an existing view and reopen it to change its location.

Docked assembly views use the normal tab/split layout and editor font. Closing the view removes its tab. Diff placement also applies to tracked hunk and conflict views. Existing views of the same Git request are reused. Separate windows remain owned by the app and close with it. A running Git write prevents closing its window or the app; normal dirty-document checks remain in force.

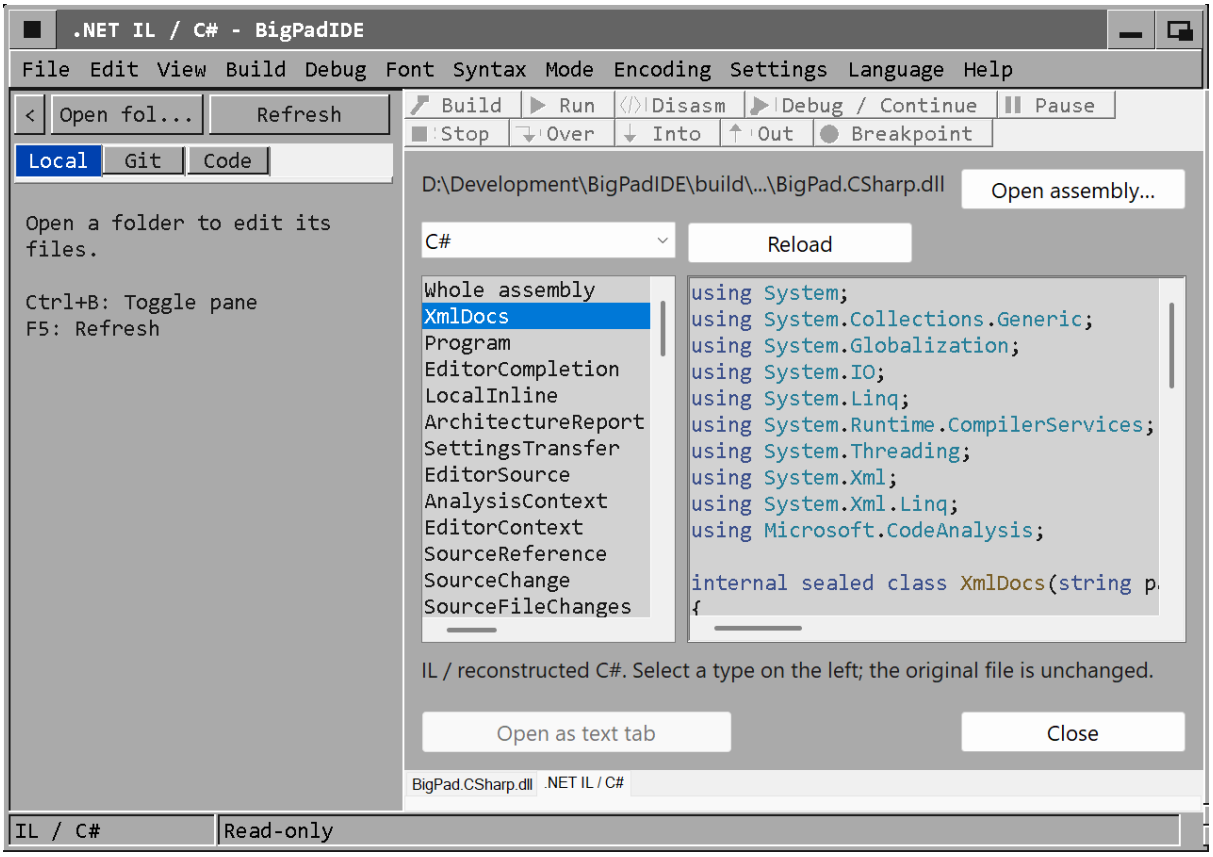


Figure 7. Assembly viewer in the app

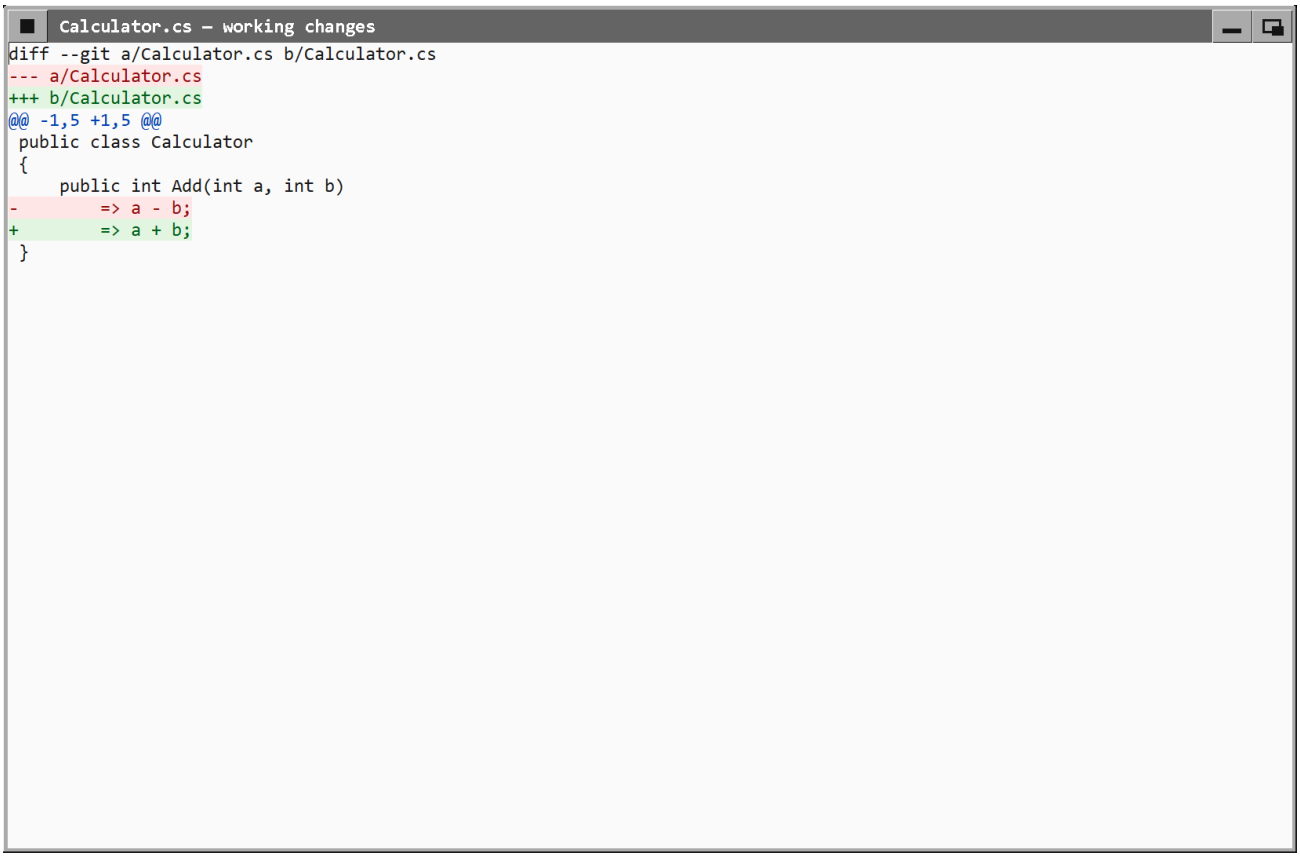


Figure 8. Separate diff window

Active C# solution project

Click a C# project node in the solution tree to make it active; bold text marks it. Alternatively choose “Set as active C# project” from its context menu. Build, Run, Debug and launch profiles use this project, including after opening or switching document tabs. Selecting a target in the output panel updates the marker too. Opening another solution or folder resets the selection. Language services and refactoring continue to use the document being edited.

4. Edit files and save safely

Use Ctrl+N for a new document, Ctrl+O to open, Ctrl+S to save and Ctrl+Shift+S for Save As. Select text by dragging or holding Shift while navigating. Ctrl+A/X/C/V selects all, cuts, copies and pastes. Ctrl+Z and Ctrl+Y undo and redo. A focused input field receives editing commands instead of the source editor.

The **Encoding** menu offers UTF-8, Windows ANSI and an optional UTF-8 BOM. **Edit → Convert line endings** changes actual CRLF/LF/CR bytes; word wrap only changes their display. Saving writes a temporary file and then replaces the target. The clipboard is limited to 32 MiB of selected bytes. Undo retains up to 128 actions; adjacent typing within one second can be grouped. Reloading or closing clears history.

4.1 File tabs and unsaved changes

Multiple files stay open in the tab bar. An asterisk (*) next to the filename marks unsaved changes. Use Ctrl+Tab or Ctrl+Shift+Tab to switch tabs; Ctrl+W or right-clicking a tab closes it. Closing prompts you to save unsaved changes.

4.2 External file changes

BigPadIDE checks open files approximately once per second and when the application is activated again. It compares file identity, size, write time and change time where supported by the filesystem. An external version is marked first; after another unchanged observation, clean documents reload automatically. This includes inactive tabs, split panes, images, hex and disassembly views. Focus and view mode are retained; byte positions are clamped to the new file size when necessary.

Unsaved local edits are never automatically reloaded. Their tab keeps its asterisk and adds **changed externally**, **file missing** or **file unavailable**. Saving to the original path checks again and lets you explicitly overwrite, save under another name or cancel. The external version is not replaced without consent. Deleted files are not silently recreated. A restored file reloads after stable observations if its document is clean. Unreadable files or invalid replacement images retain their previous view and warning until they can be reloaded successfully.

File → Reload from disk (**Ctrl+Shift+R**) reloads the current document explicitly. Unsaved edits require an explicit discard confirmation with a safe default. Failed reloads preserve the previous editor contents.

In file-backed piece-table mode, before the first local edit BigPadIDE preserves the original bytes in a temporary disk file. Original piece-table ranges therefore remain consistent even if another program overwrites the source in place. The copy uses bounded RAM but preflights free space in the Windows temporary directory for the source size plus a 16 MiB reserve. At 32 MiB or above, a dialog shows progress and **Cancel** while the copy runs in a worker. Editing is temporarily disabled. Cancellation discards partial backing and leaves the original edit unapplied, so it can be retried. The snapshot remains after saving for undo, and is released on reload or close. A failed snapshot or an already overwritten backing file stops the edit with an error instead of constructing an inconsistent version. New, unsaved documents require no source-file snapshot. Full-RAM mode already retains an independent buffer and does not need this disk snapshot; see chapter 24.5.



Figure 9. Original snapshot progress

4.3 Create files and folders in the explorer

In the **Local** tab, the context menu offers **New file…** and **New folder…**. The selected folder is the destination; for a selected file, its parent folder is used. With no selection, the open root folder is used. Open the menu with a right click or the context-menu key.

Enter a single name and press Enter. Invalid Windows names remain in the dialog for correction; Escape cancels. Existing files and folders are never overwritten. After success, the new entry is selected and revealed in the refreshed tree. New files open in the editor; folders need no extra tab. Session capacity is checked before creating a file. Folder pickers inside file dialogs stay read-only.

4.4 Rename and move files and folders

Select an entry in the local explorer and use **F2** or **Rename…** from its context menu. **Move to…** selects an existing destination folder. Escape cancels. Files and folders can be moved within the same volume; existing targets are never replaced.

Open tabs retain unsaved contents, undo, caret and view mode. Directory operations also update descendant documents, the explorer root, history, session, breakpoints and selected build target. Recovery is regenerated with the new path, keeping the previous checkpoint until its replacement succeeds. The new item is revealed and selected when it remains within the explorer root.

Stop builds, live mode and debugging first. Conflicting tabs, externally modified open files, file locks and session limits are checked before filesystem changes. Open files in moved directories first receive independent disk backings; this can require extra disk space and time. External changes during the following rename retain conflict protection and are reported. If session/history saving fails after the filesystem operation, the message explicitly states that the move already happened. These local commands are unavailable in Git view and folder pickers.

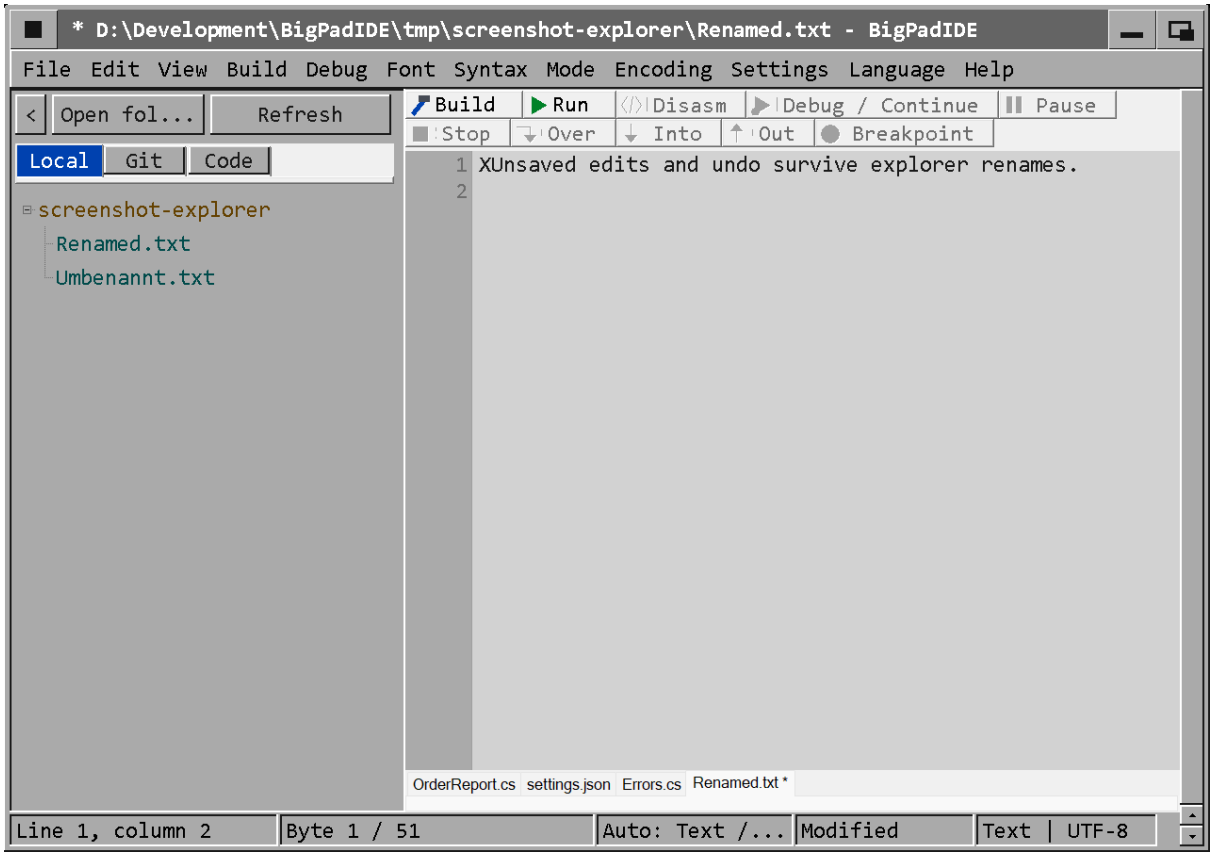


Figure 10. Renamed tab retaining its unsaved edits

4.5 Large files and long lines

Paint uses bounded folding and indentation previews so long lines are not completely rescanned on every redraw. Very long lines may therefore omit a folding marker; an explicit click in the folding margin still checks the complete structure. Saving combines small edits into bounded output buffers, retaining encoding, BOM and undo history.

The performance report includes reproducible 64 MiB to 1 GiB measurements, hardware, limitations and raw data. Results apply to the stated local system and selected text fixtures.

5. Navigation, finding and replacing

Ctrl+F searches the current text document, Ctrl+H replaces, and F3 finds the next match. Text search is exact and case-sensitive, includes unsaved edits and wraps once at the end. Find and replacement strings are limited to 64 KiB each in the document encoding. Replace All forms one undo action and does not search inserted replacement text again. Cancelling keeps completed replacements undoable. Project-wide search reads saved files; save the document first when needed.

5.1 Go to line or offset

Ctrl+G opens the dialog from the Edit menu. Lines start at 1; offsets are zero-based byte positions, written in decimal or hexadecimal with `0x`. A UTF-8 BOM is excluded from offsets. End of file is a valid target. Line targets switch to text view; byte targets switch to hex view. Disassembly keeps its view and requires an address aligned for the selected architecture. Image and diff views do not support this navigation.

Line lookup reads small blocks, shows progress and can be cancelled. It handles CRLF, LF and CR. Missing targets, read failures and intervening document changes leave the caret in place.

5.2 Quick open

Ctrl+P or File → Quick open searches file names recursively in the open folder. With no open folder, it uses the current file's directory. Spaces separate terms that must all occur in the relative path. Matching ignores case; `/` and `\` are equivalent. Exact file names precede name prefixes, name contents and directory matches.

Up/down selects a result, Enter opens it and Escape closes the dialog. The list shows at most 200 results and reports the total count. Already open files retain their tab and unsaved changes. An open failure keeps the dialog available for another attempt.

Indexing and filtering show progress. `.git` and linked directories are omitted. The index is limited to 50,000 files, 200,000 visited entries, 64 directory levels and 16 MiB of stored path text. Reached limits and unreadable entries are reported. Reopening the dialog refreshes the index; File → Open remains available for files outside the list.

5.3 Project search and regular expressions

Use **Edit** → **Find in project...** or **Ctrl+Shift+F** to open a movable, resizable search window. The explorer folder is the search root; without one, the current file's directory is used. Search reads **saved files** and never saves pending edits.

Enter text, optionally select **Regex** and **Match case**, then press **Search**. Separate file filters and exclusions with `;`: `*` matches any characters and `?` one character. Patterns match file names or relative paths using `/` or `\`. Excluded directories are pruned. Example: filters `*.cs;*.cpp`, exclusions `bin;obj;tests`. Git metadata and reparse points (links/junctions) are omitted. The search root must be a directly accessible directory.

Results show **file:line:column** and a preview. Columns count UTF-16 code units, as the C# tools do. Double-click, Enter in the list, or **Open match** opens the tab and selects the exact byte range. The window stays open for further matches. Files changed since the scan, and tabs with pending edits or a different file/encoding state, reject navigation; save/reload or reopen with the appropriate encoding, then search

again. UTF-8 with/without BOM and the Windows ANSI codepage are supported. NUL-containing files and UTF-16/UTF-32 BOMs are treated as binary and reflected in the summary.

Regex patterns run **line by line** across CRLF, LF and CR. Groups, alternations, character classes and quantifiers are supported. Lookarounds, backreferences and other constructs unavailable in nonbacktracking mode report an error. Empty matches make forward progress. **Cancel/Escape** stops work and discards partial matches; **Close/Escape** then closes the window. Editing search options cancels running work and invalidates previous results.

Limit	Value
Query / filter field	4096 / 1024 characters
Matches	1000
Files / directory entries / depth	10,000 / 100,000 / 32
Per-file size / sum of considered file sizes	64 MiB / 512 MiB
Line length	65,536 UTF-16 code units
Search time / process deadline	80 / 90 seconds
Per-match regex timeout / helper memory / output	100 ms / 512 MiB / 8 MiB

Skipped files/lines, read errors and reached limits remain visible in the summary. In the default piece-table mode, large files and long lines are not loaded entirely into RAM. Full-RAM mode deliberately reads complete document buffers; see chapter 24.5. ANSI fallback may read a file twice; the time and process bounds still apply.

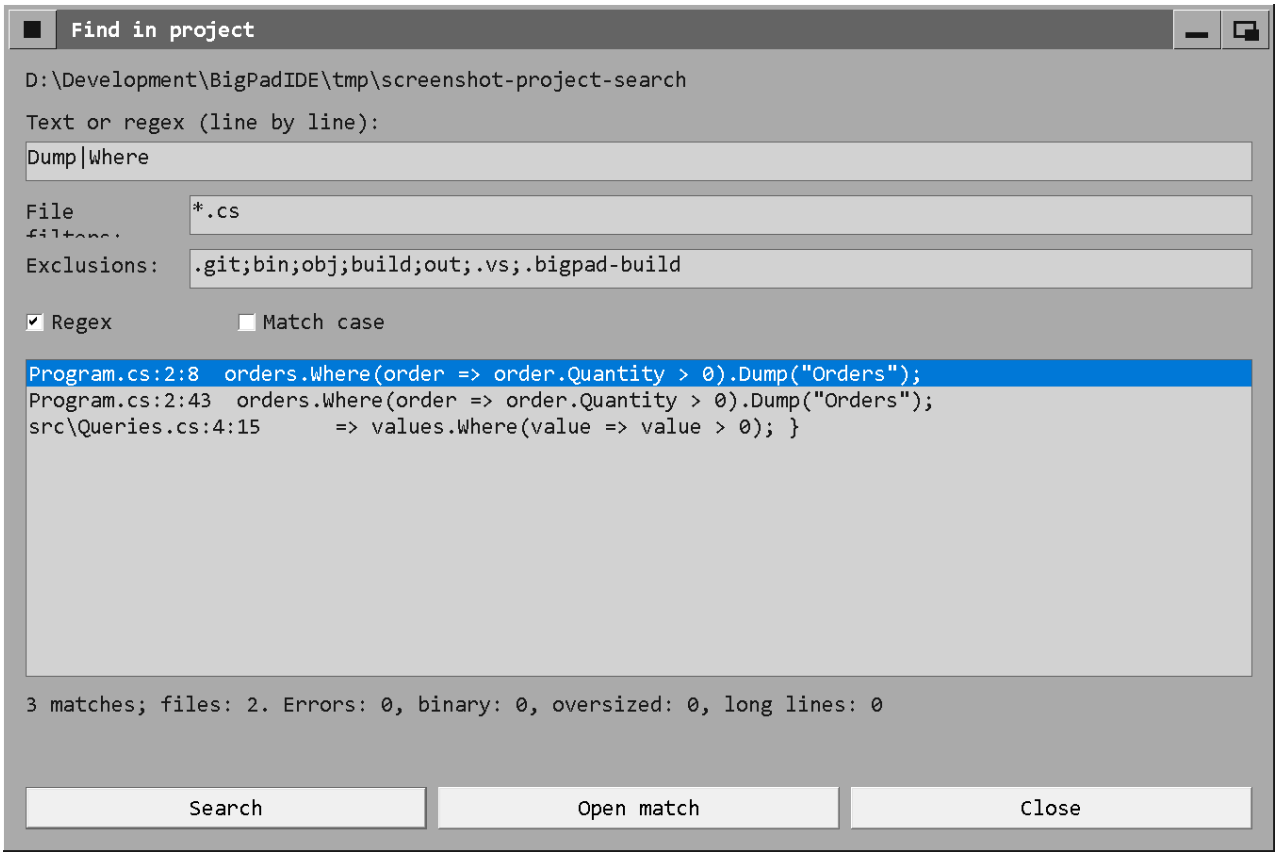


Figure 11. Project regex search with filters and exact result positions

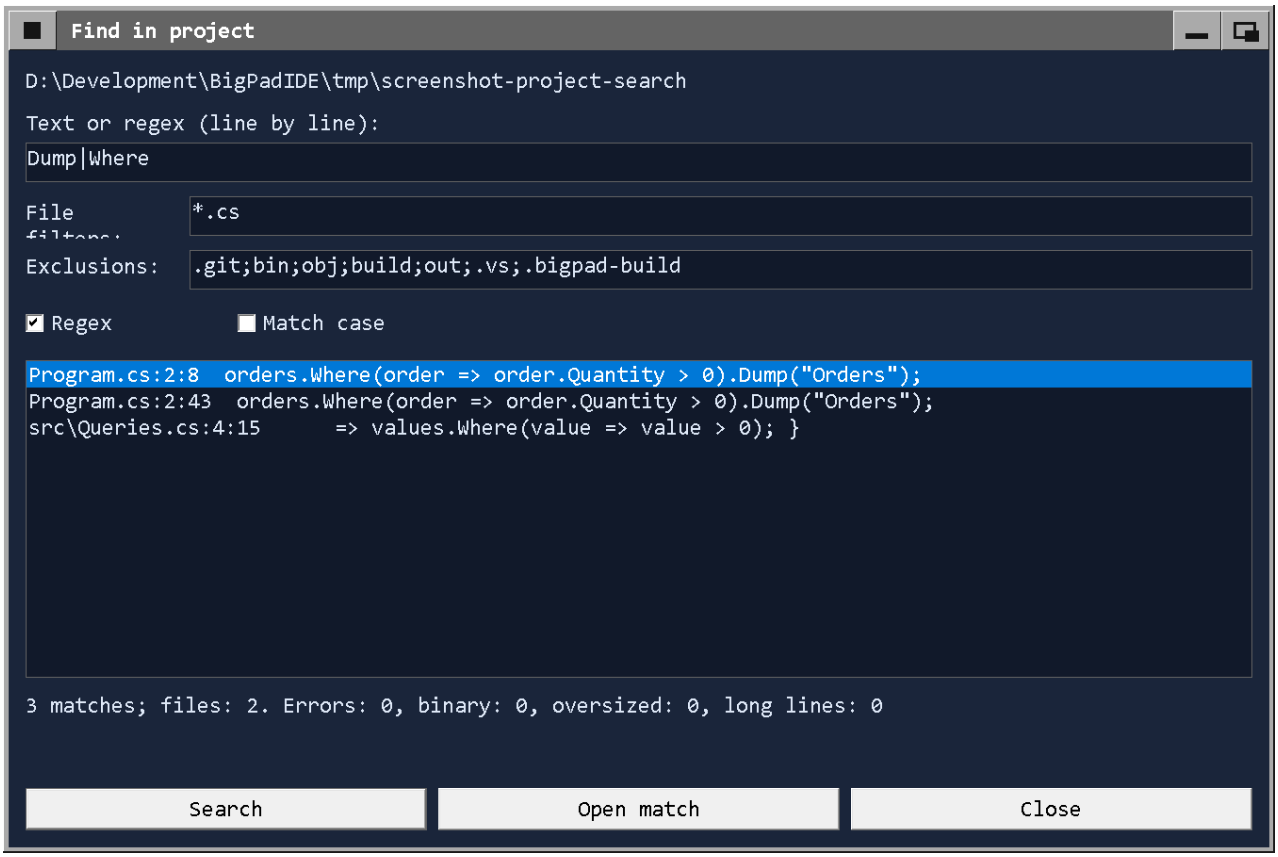


Figure 12. The search window follows the Midnight theme

6. Write, build and run C#

Open a `.cs` file or create a new one and save it before normal Build/Run. Ctrl+Shift+B builds; F6 builds and runs. The output panel separates build messages, console output, structured results, diagnostics and debugging information. Live C# is intended for small standalone experiments; project dependencies need a proper project context. Review the original source before running it.

Ctrl+Space requests completion, Ctrl+Shift+Space signature help, Ctrl+I symbol information and F12 a source definition. These features require the bundled C# helper and a compatible SDK. Syntax highlighting alone does not establish a project.

6.1 Files, folders and code

Open a directory with **File** → **Open folder**. Expand its tree and select a file. The status bar shows the active encoding; the debug toolbar remains above it. The example used in these screenshots is OrderReport.cs.

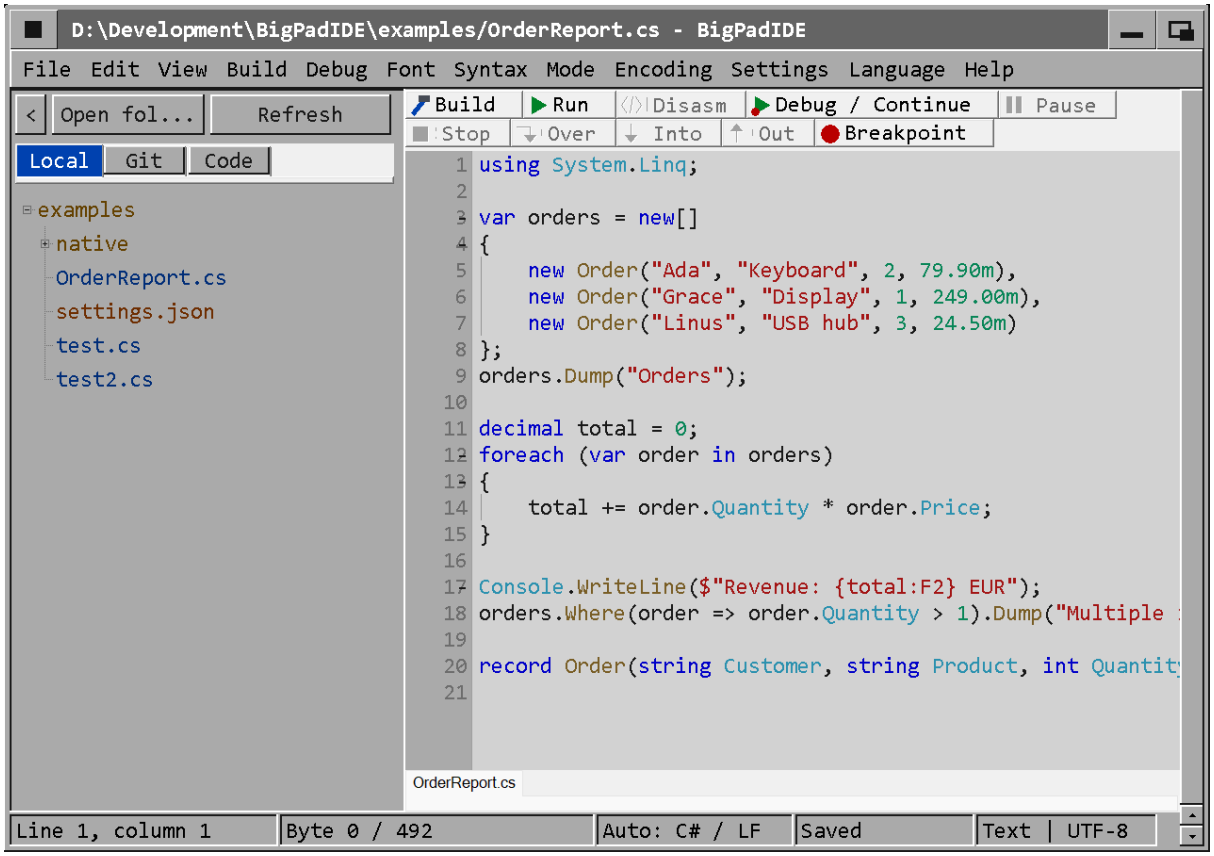


Figure 13. BigPadIDE with a folder tree and C# syntax highlighting

6.2 Compiler diagnostics

C# errors are underlined in the source and listed in **Problems**. Double-click a diagnostic to navigate to its location. This example assigns a string to an integer, producing CS0029.

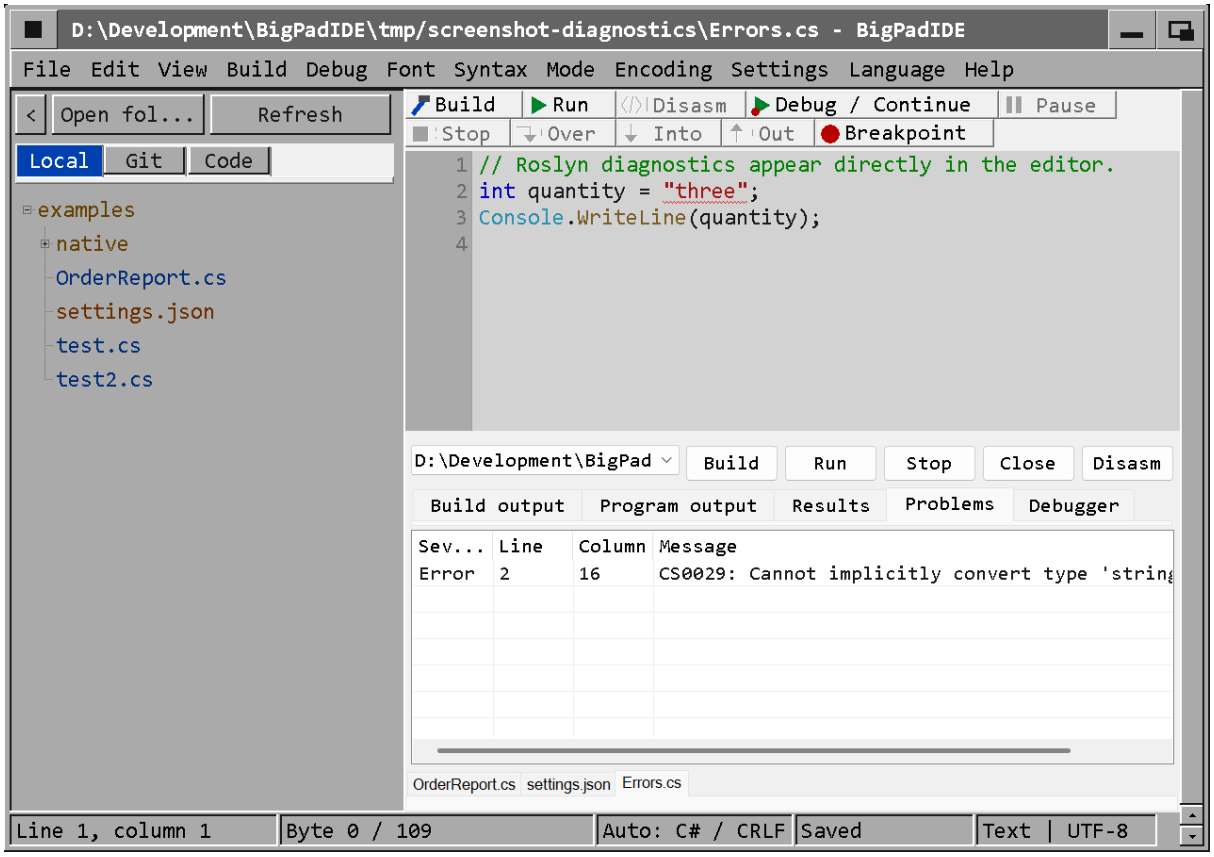


Figure 14. Red source underline and the corresponding compiler diagnostic

7. Solutions, active projects and C# tools

The active project controls Build, Run, Debug and launch profiles. The C# analysis context controls references, symbol search and refactoring. These are separate choices.

7.1 Keep the project selection synchronized

Select a project folder, project file or associated source in the explorer using the mouse or keyboard. Native folders use the nearest `bigpad.ini`; C# uses an explicit solution project or an unambiguous project file. Ambiguous folders do not choose an arbitrary target. The bottom dropdown follows automatically.

Selecting a target below updates the explorer selection and bold marker in return, including nested projects while the sidebar is collapsed. The sidebar stays closed and reveals the selected project when reopened. Switching a document tab alone does not reset an explicitly chosen target. Finish running work before changing it.

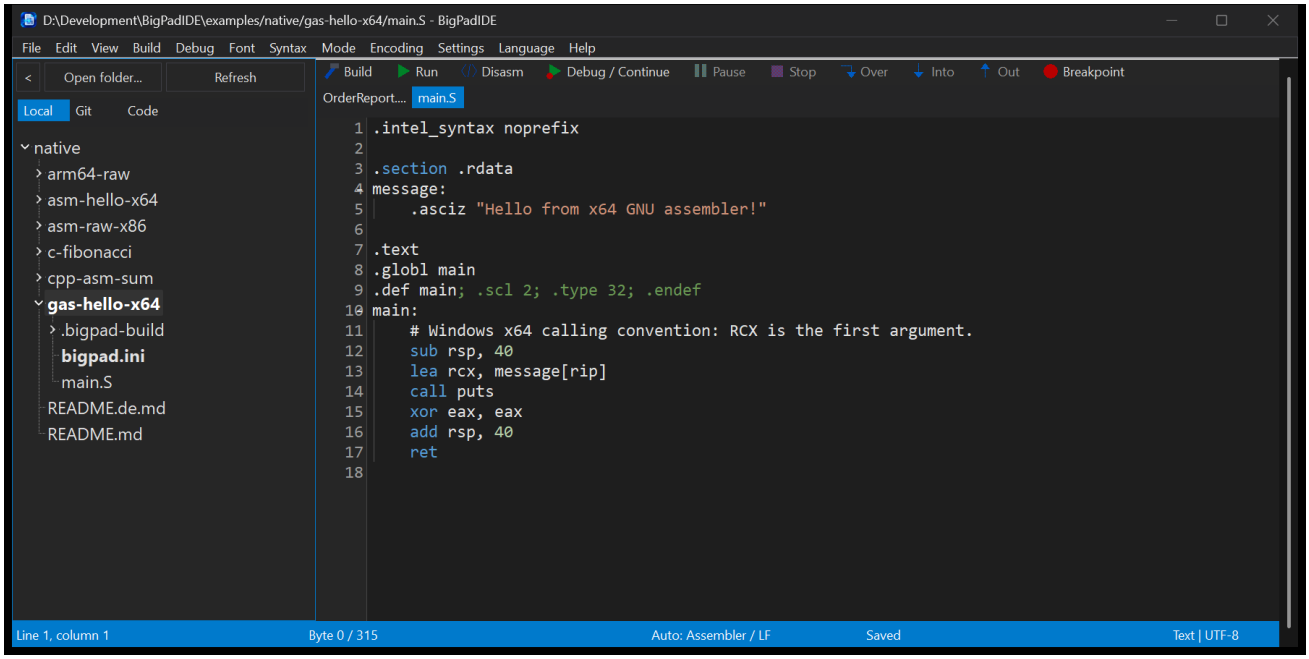


Figure 15. Native project workspace

7.2 C# project tools

Use **Build → C# project context ...** for automatic discovery or an explicit `.csproj`, `.sln` or `.slnx`. The selection lasts for this editor instance. Discovery prefers the nearest unambiguous solution, then the nearest unambiguous project, up to twelve parent directories and stopping at the Git root. Ambiguous contexts report an error. Select a `.csproj` explicitly for sources linked into multiple projects. Ambiguous target-framework contexts are rejected. Load the whole solution when renaming shared project symbols so consumers are in scope.

MSBuild loads the Debug configuration with real Compile entries, preprocessor symbols and restored NuGet/project references. Restore/build the project first; missing workloads and incomplete contexts report errors. Analysis does not run its own restore. Sources outside projects and untitled buffers retain standalone

standard-.NET references and existing helper support. Standalone/live execution still builds an independent program.

Open **C# find references ...** at a symbol to list source definitions and usages, including file, line and column. Enter, double-click or **Open** navigates. Use the source context menu or **Shift+F12** directly in code. The filter narrows paths and source text; the preview shows the selected location. Overloaded methods are distinguished semantically. **Ctrl+click** or **F12** goes to the C# definition.

Show in output panel transfers the results to a dedicated **References** tab. Choose this as the default under **Settings → Open references**, or keep using the separate search window. This preference is included in JSON settings exports. Double-click or press Enter in the output tab to open a location. Tooltips show the full path and source preview. Each new search replaces the previous result list. Search again after changing source so symbols and locations match the buffer.

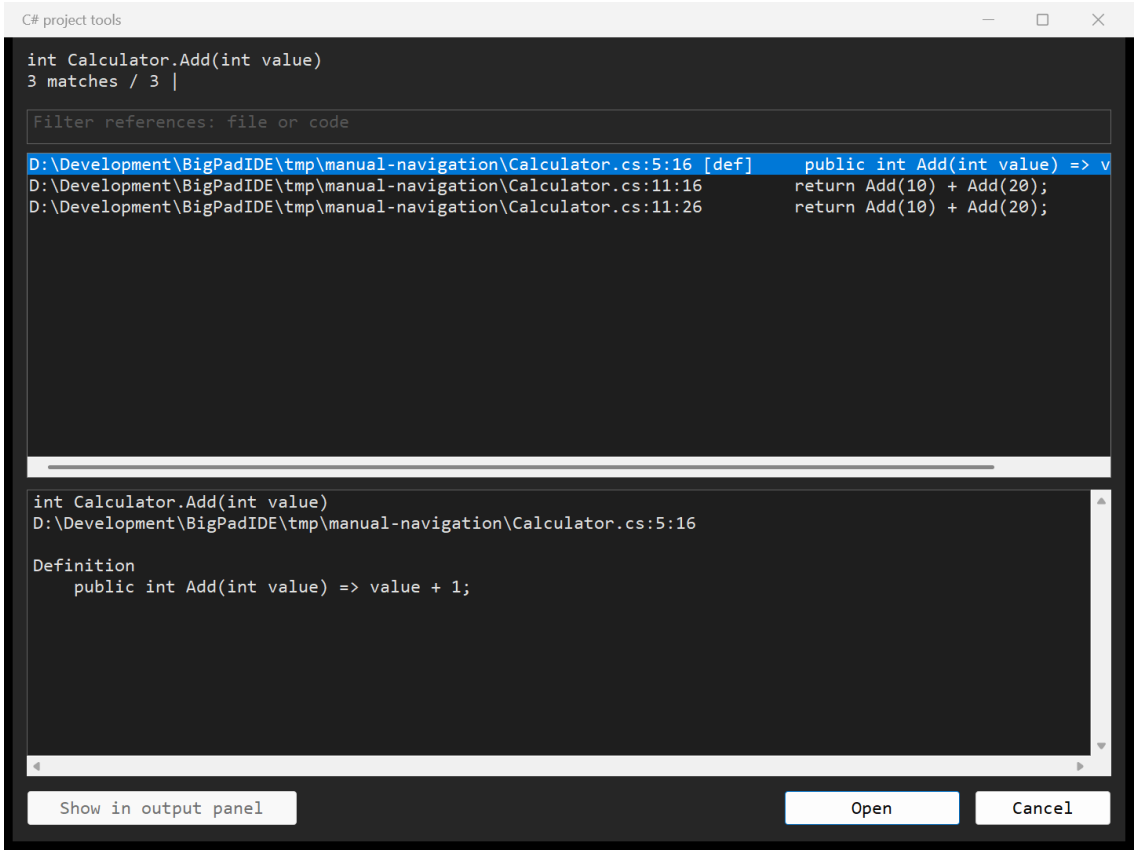


Figure 16. Reference finder with filter and source preview.

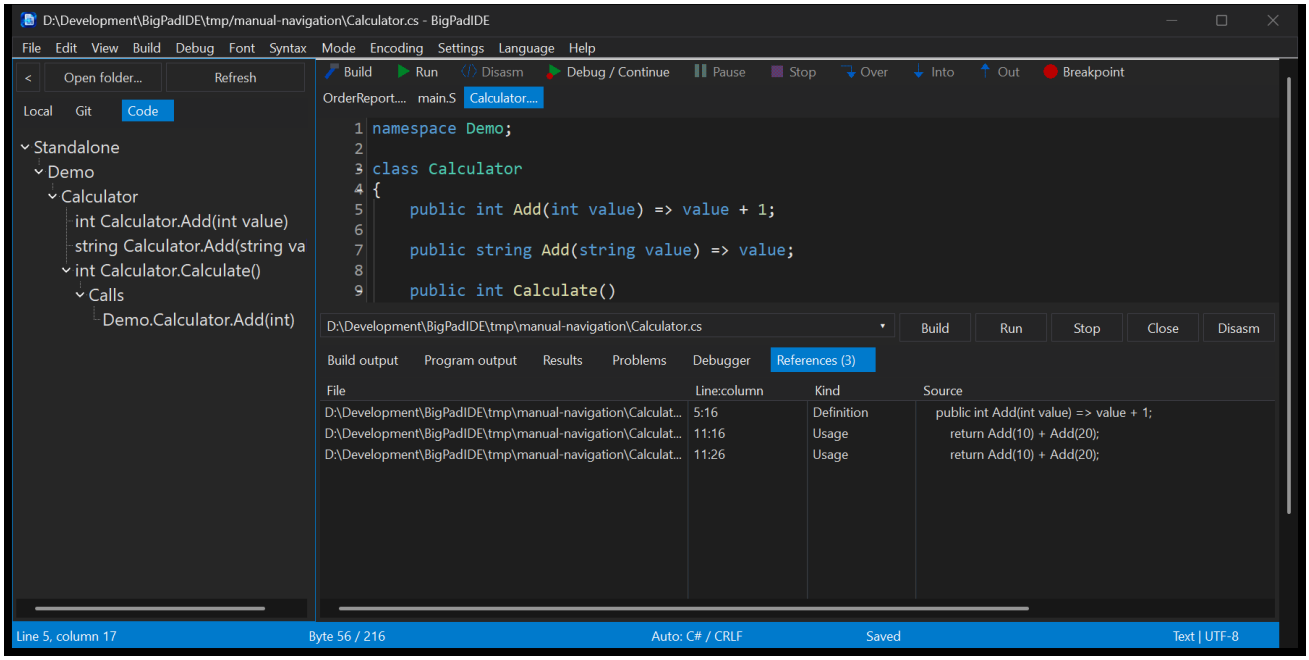


Figure 17. References stay visible in a dedicated output panel tab.

The explorer **Code** tab, also available through **Build → Code map**, analyzes the current C# project context. Projects contain namespaces, types and members. **Calls** and **Dependencies** groups show resolved method calls and type relations. Click a symbol or call to navigate; its context menu finds references. **Refresh** rebuilds the tree using unsaved C# buffers too. Call leaves do not recursively expand cycles. Limits are 5,000 nodes and 10,000 dependency edges. These semantic tools cover C#. Use project text search for other languages.

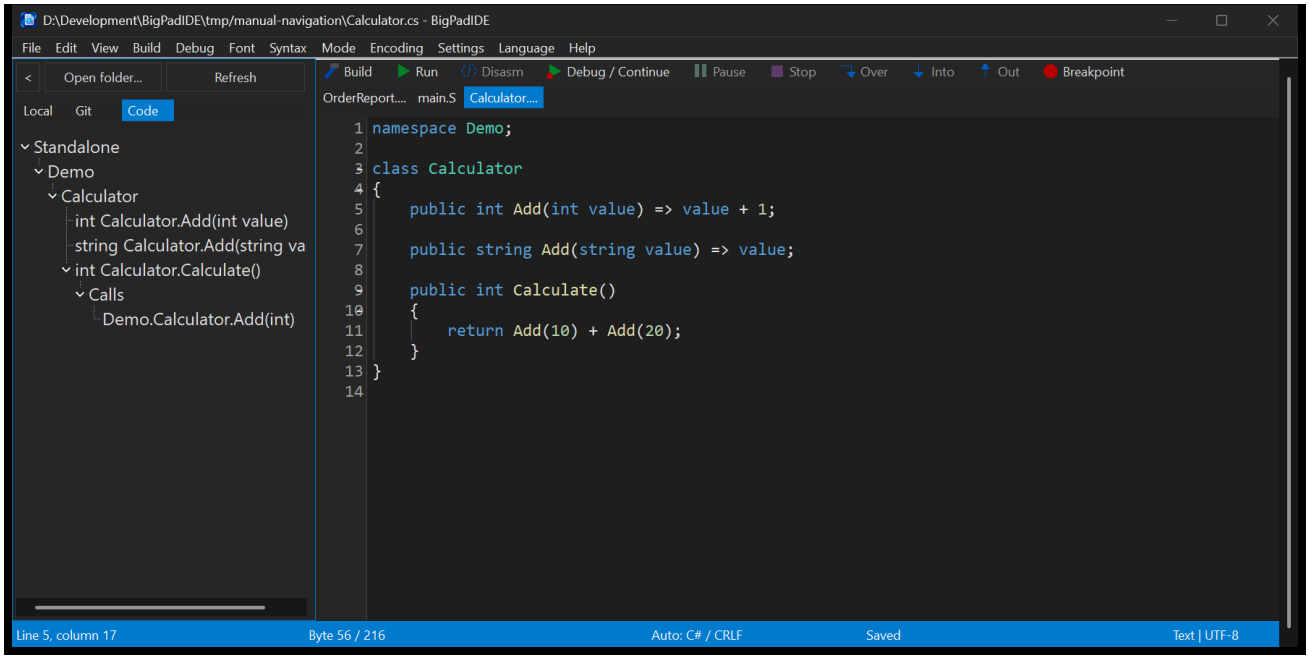


Figure 18. Code map with resolved method calls.

In the text editor, one click places the caret and a **double-click** selects a whole word. Keep the button pressed and drag to extend the selection by whole words, including backwards. Ordinary dragging selects individual characters. Buffered painting displays text and background together while typing and selecting.

C# rename symbol ... requests a new identifier and previews every text edit in the loaded context. Metadata symbols, edits to generated code and new compiler errors are rejected. Files are not renamed. **C# code actions ...** offers an explicit type or `var` at a local declaration when its semantic type stays unchanged. Unused `using` directives can also be removed.

Choose **Apply** in the preview. The operation runs again; changed proposals need another review. Applying validates source hashes, old spans, encoding and unchanged document generations. All buffers are prepared before any content changes. Closed affected files open as normal modified tabs. Existing unsaved changes are retained. Files are not saved automatically. **Ctrl+Z** undoes the action per file. BOMs and line endings are retained; ANSI edits that cannot be represented without loss are rejected. Affected tabs move their caret to the start. Escape/**Cancel** stops a running helper and its BuildHost processes. These tools are disabled during builds, live mode and recovery prompts. Read-only reference search remains available during debugging; source changes require stopping the session.

Limits: 1 MiB per current/unsaved source, 64 editor snapshots and 8 MiB combined; 2000 project documents, 16 MiB per saved source and 128 MiB combined; refactoring 64 files, 4 MiB per affected file, 32 MiB combined, 10,000 text edits, 1 MiB per insertion and 8 MiB inserted text. At most 10,000 reference locations, 16 MiB protocol output, 60 seconds per project request, 1 GiB combined helper/BuildHost memory and eight processes. Normal code assistance has a 30-second deadline. Larger requests are rejected without partial application; choose a smaller scope. Normal session capacity limits also apply.

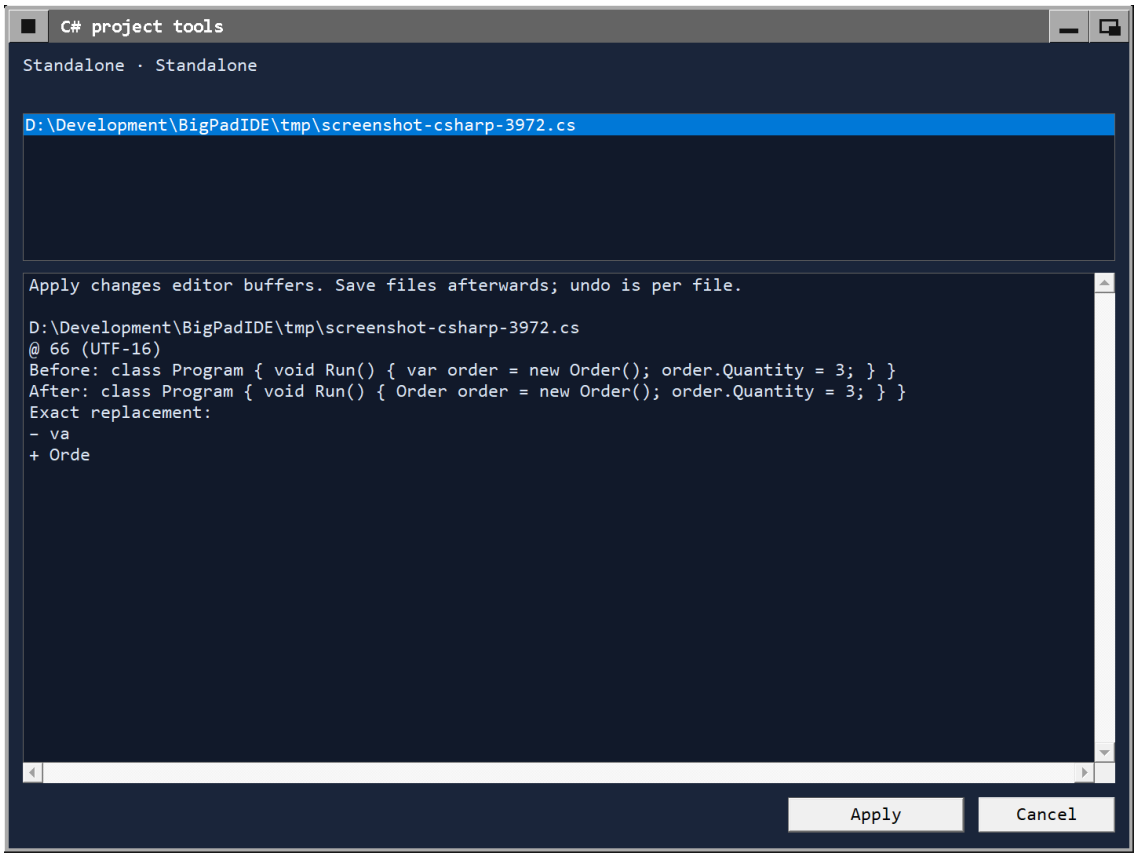


Figure 19. Semantic code action with before/after context and its exact text replacement.

8. Output, results and launch profiles

Ctrl+J shows or hides the output panel. Drag its upper edge upward or downward to change the panel height; the vertical resize pointer marks the draggable strip. This changes the window area without changing fonts. The chosen height survives closing/reopening and is saved with settings. A small main window temporarily clamps it; enlarging the window restores it. Some editor space remains available. Ctrl+mouse wheel changes the font under the pointer instead.

8.1 Tables and program output

Call `.Dump("Orders")` on a sequence to display its public properties as table columns. Use the **Results** tab and selector to switch between tables. Console output appears separately in **Program output**.

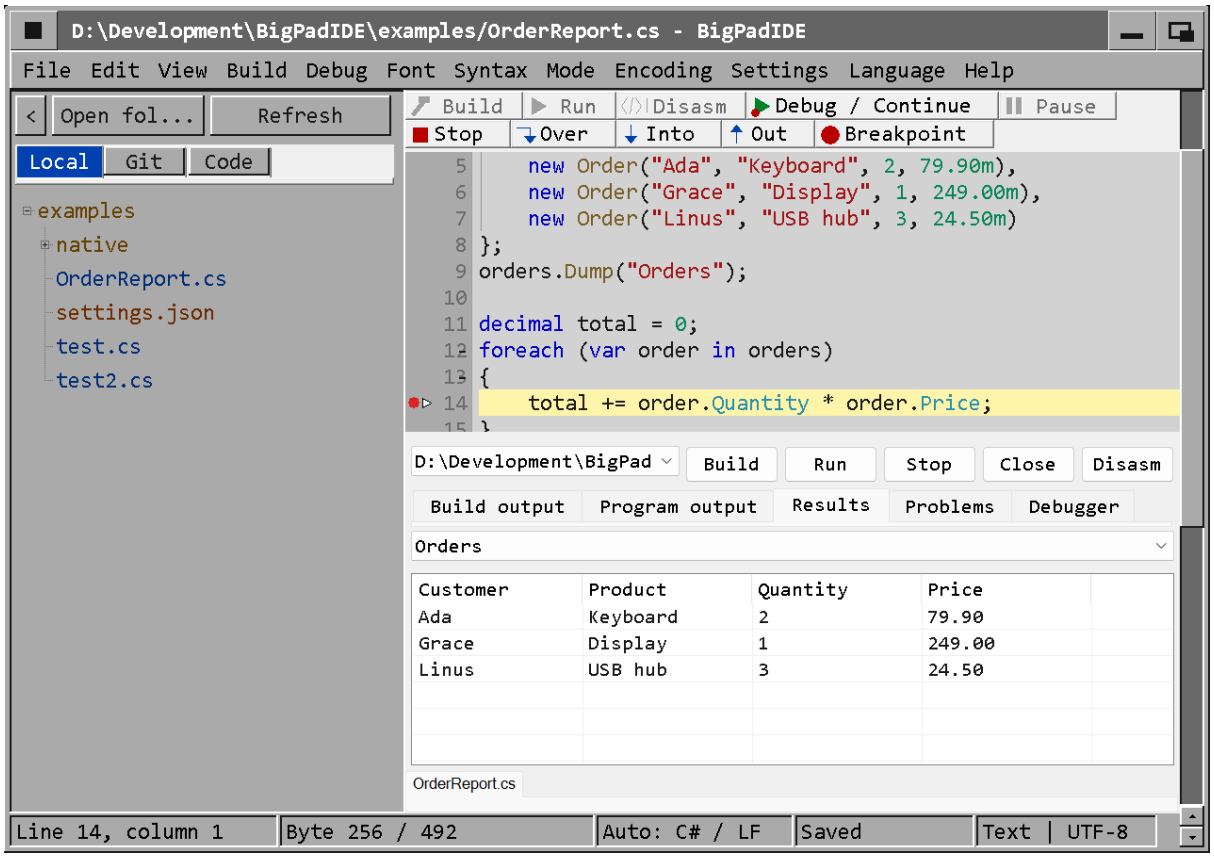


Figure 20. Order objects displayed as rows and columns in Results

8.2 Build/run profiles and input

Select a target in the output panel, then open **Build → Build configuration and compiler options ...**. **Build** and **Run** keep independent working-directory, argument and environment drafts. **Save** applies them to this target; **Cancel / Escape** discards the dialog draft. Stop builds, debugging, live mode or recovery prompts first. Save or close an open configuration tab that has unsaved edits before using the dialog.

Relative working directories resolve from the selected target's parent. Blank keeps the default: the project directory for build tools, a temporary build directory for standalone C# sources, and the original source/project directory for execution. .NET's Run working directory changes the application context without

accidentally changing SDK selection through another directory's global.json. An explicit Build working directory also controls the SDK host's working directory.

Arguments use Windows quoting: `"two words"` is one argument and `""` is an empty argument. Processes receive them directly without shell evaluation. **Build** contains additional compiler/linker or dotnet arguments; **Run** contains application arguments. .NET Build & Run forwards Build arguments to `dotnet run` before `--`; use supported options such as `--configuration Release` or `-p:DefineConstants=DEMO`. Native pipelines apply Build arguments to compilers and linkers, excluding objcopy extraction.

Enter environment overrides one per line: `NAME=value` sets/replaces an inherited value, `NAME=` sets an empty value, and `-NAME` removes it. Names are case-insensitive; duplicates are rejected. The parent environment is unchanged. .NET Build & Run uses Build values with Run overrides in its shared SDK/application process; Run wins. Native build/run processes use their respective profiles. Named live snapshots use their original file's profile. Untitled live sources use the selected explorer folder, otherwise their temporary context. A replacement live run stops the old process and input.

Profiles are stored locally as UTF-8 `<target>.launch.ini`, for example `Program.cs.launch.ini` or `bigpad.ini.launch.ini`. Saving checks the loaded file generation and atomically replaces it only after a complete flushed write. Failed or stale saves preserve the previous settings. Linked sidecars are not replaced; edit those directly. The format has a `[BigPadLaunch1]` header, `[build]` and `[run]` sections, and `directory=`, repeated `argument=`, `input=auto|utf8|ansi|oem`, `env.NAME=value` and `unset.NAME=` entries. File-format argument values are literal, without additional quoting.

Under **Program output**, enter a line and choose **Send / Enter**. Normal prompts appear before a newline. **EOF** drains the queue and then closes stdin, disabling further input. **Stop** terminates the job and child processes. Edit-menu clipboard and undo actions target the focused input field rather than the source document. Input is a line-based pipe with CRLF, rather than a console terminal.

Choose the program's expected input encoding in the Run profile. **Automatic** uses system OEM for Windows C# and UTF-8 for native programs. UTF-8, Windows ANSI and Windows OEM are also selectable. A C# UTF-8 reader can set `Console.InputEncoding = new System.Text.UTF8Encoding(false);`. Unrepresentable characters are rejected and remain in the input field for correction. Errors and full queues are reported explicitly.

Limits: 256 KiB settings file, 128 arguments/profile, 4096 UTF-16 characters/argument, 24,000 characters in the formatted argument list, and a complete Windows command line under 32,767 characters. At most 64 environment overrides per profile or merged .NET Run profile, names up to 256 and values up to 4096 characters, and a generated environment block up to 1,048,576 UTF-16 characters. Input field: 16,384 UTF-16 characters, 64 KiB/message and 1 MiB queued bytes. A child that does not read input does not block the UI.

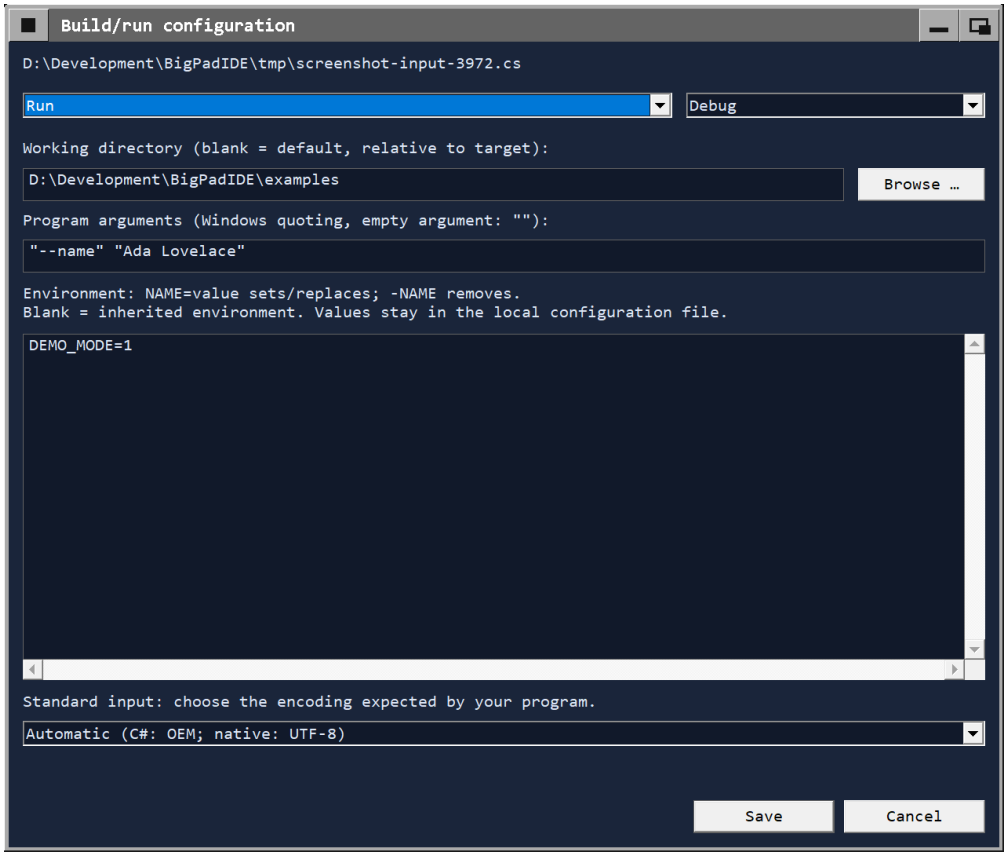


Figure 21. Run profile with working directory, arguments and environment

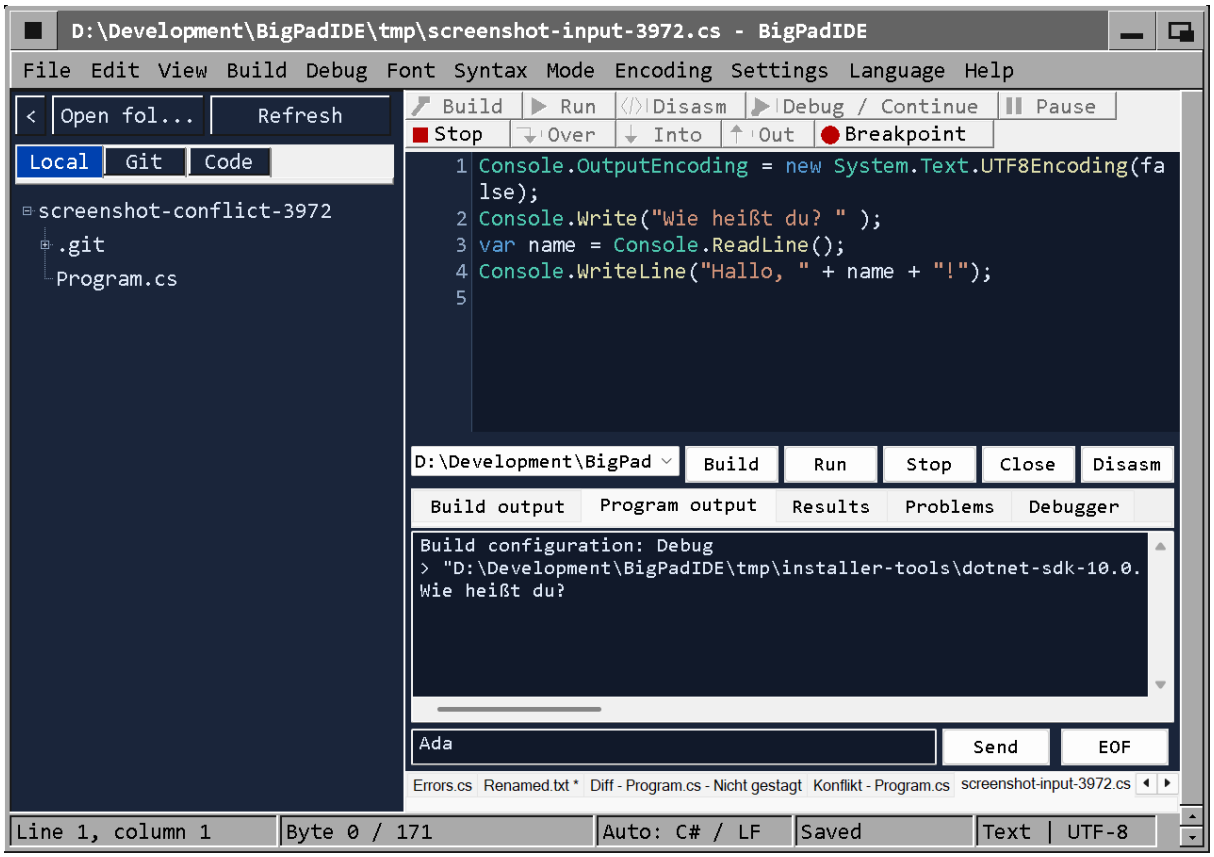


Figure 22. Running program with a visible prompt and input field

9. Debug C# programs

Build the intended target with compatible debug symbols before starting. The yellow line and arrow mark the current execution position. Red markers are breakpoints. A source jump clears an old blue text selection.

Call-stack and variable columns adapt to their contents and available width, prioritizing file paths and values. Hover truncated cells for tooltips. Manually resized widths remain until the next debugging session.

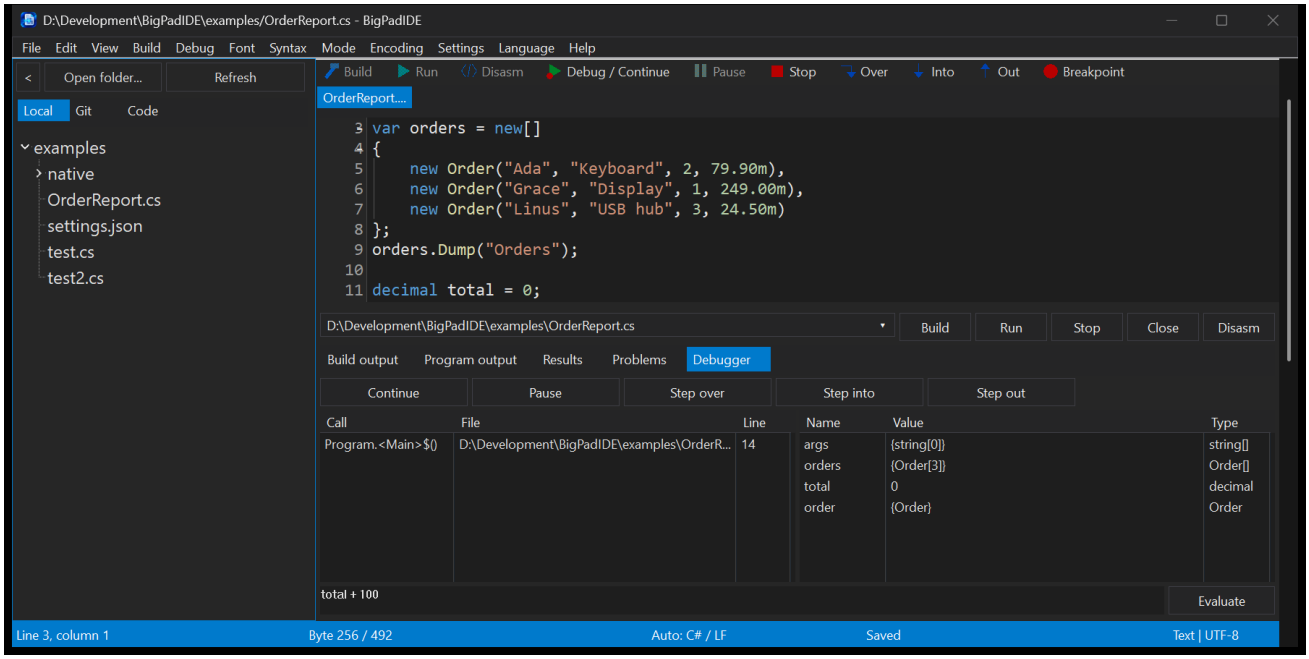


Figure 23. Enlarged debugger panel with actual session data

9.1 Breakpoints and watches

Click beside a line number or press **F9** to toggle a breakpoint. **F5** starts debugging or continues execution. The red marker identifies the breakpoint; the yellow row and arrow identify the paused instruction. The Debugger tab shows the call stack and local variables. Enter an expression such as `total + 100` and choose **Evaluate** to inspect its value.

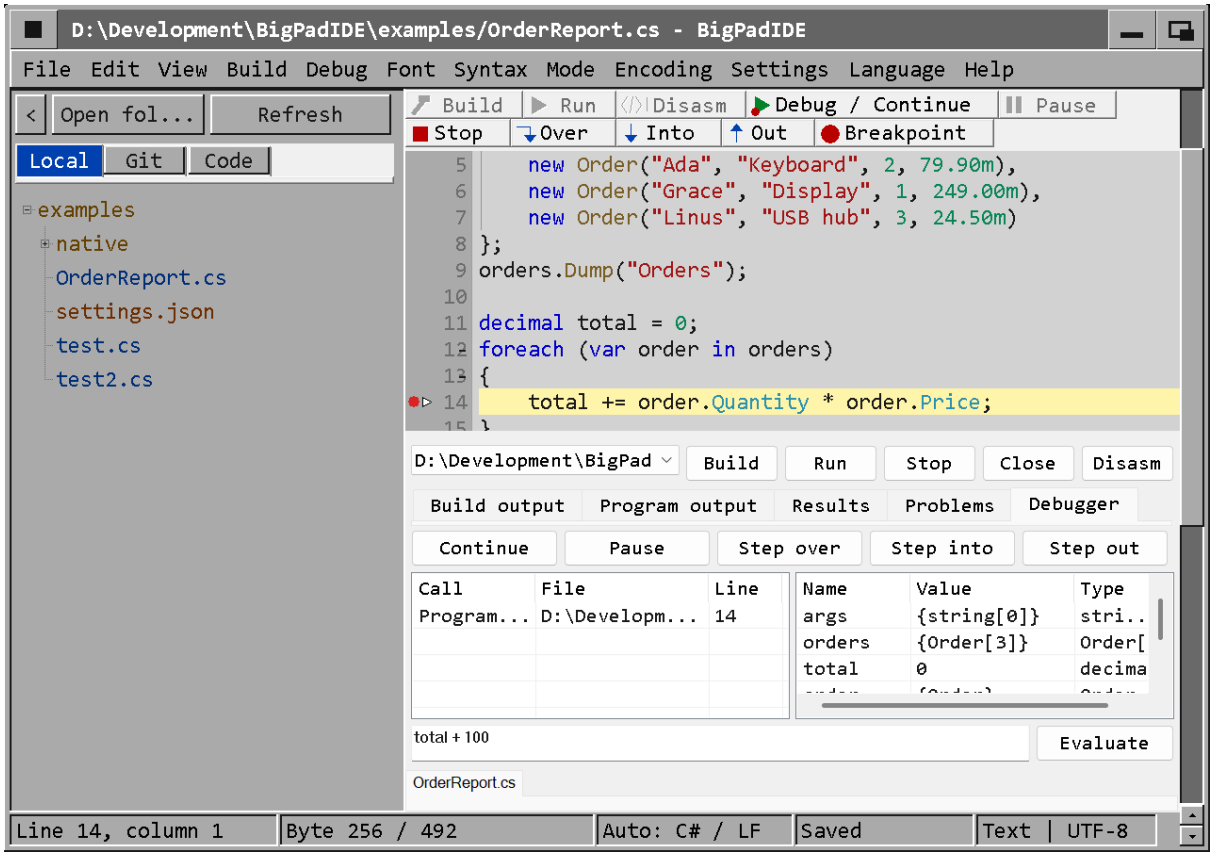


Figure 24. A real breakpoint, call stack, locals and evaluated expression

Use **F10** to step over, **F11** to step into, **Shift+F11** to step out, **F8** to pause and **Shift+F5** to stop. Source editing is disabled during a session.

Shift+F9 / Debug → Breakpoint condition edits the current line before a session starts. An empty condition always stops; a C# expression such as `count == 3` stops only when true. The Enabled checkbox keeps a breakpoint without binding it. Conditional markers are amber diamonds; disabled markers have muted outlines. Conditions and enabled states follow line edits and Undo/Redo.

Breakpoints are saved automatically between launches, independently of automatic UI settings, for the shared application. A title warning reports unreadable or unsaved state; corrupt saved data is retained until explicitly replaced with **Debug → Clear all breakpoints**.

Debug → Attach C# process lists accessible processes by PID and full image path. Select an already running .NET process; matching source and portable PDBs must be available for source breakpoints. Refresh stale selections. Attach checks the process creation time and does not build or restart the target. **Stop** detaches and leaves the attached process running; stopping a launched session terminates its target. Unsupported targets and adapter failures appear in Program output. Attach needs the same .NET SDK and NetCoreDbg setup as launch.

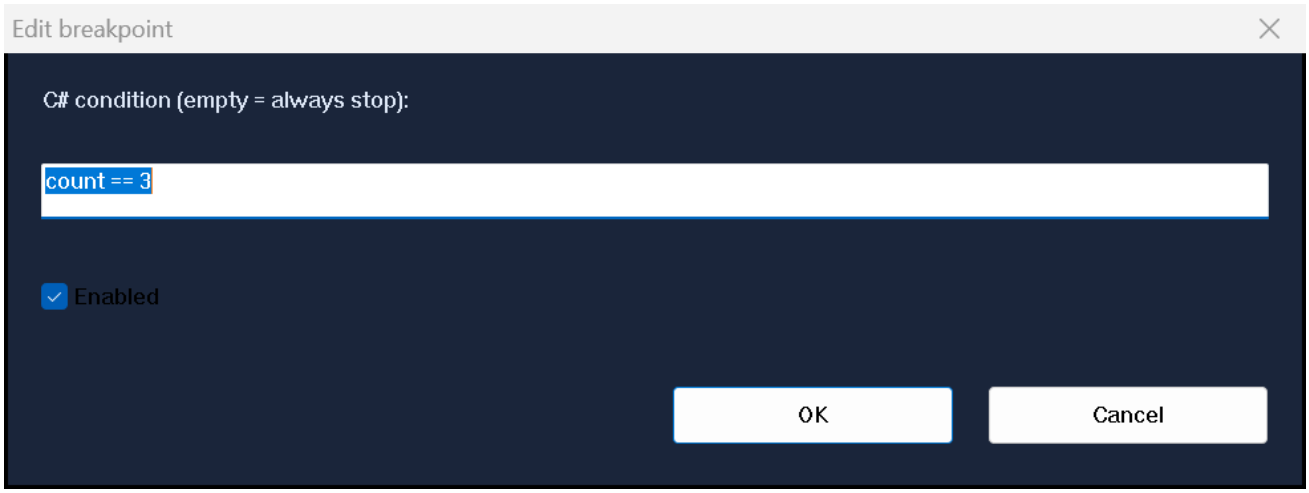


Figure 25. Conditional breakpoint dialog

9.2 Runtime hover values, object browser and changing variables

While paused, hold the pointer over a source variable briefly. Its tooltip shows the expression, current runtime value and type in the selected stack frame. Simple identifiers and dotted member access such as `order.Price` are recognized; arbitrary selected code is not automatically run. The adapter determines whether a member can be evaluated. Unavailable identifiers do not display stale values.

This works with C# through NetCoreDbg and supported native targets through CodeLLDB. Continuing, changing frame, scrolling or leaving the view discards the old tooltip. Hovering the variable table also shows name, value and type, including long values.

The value window uses native Windows controls and a readable DPI-aware font, at least 12 pt by default. Its font can be customized through the Panels area in **Settings → Fonts and colors**. Routine status updates do not close it while the pointer remains still.

For collections and objects, click the small expansion arrow beside the value. The object browser requests children only when expanded. Array entries and their object members can be expanded recursively. Move the pointer from the code into the value window to explore it; it opens close to the pointer and closes about two seconds after leaving both areas. Escape also closes it. A request displays up to 200 children, with up to 2000 nodes in the value window. Children are those exposed by the debug adapter; the IDE does not invoke arbitrary enumerator methods itself.

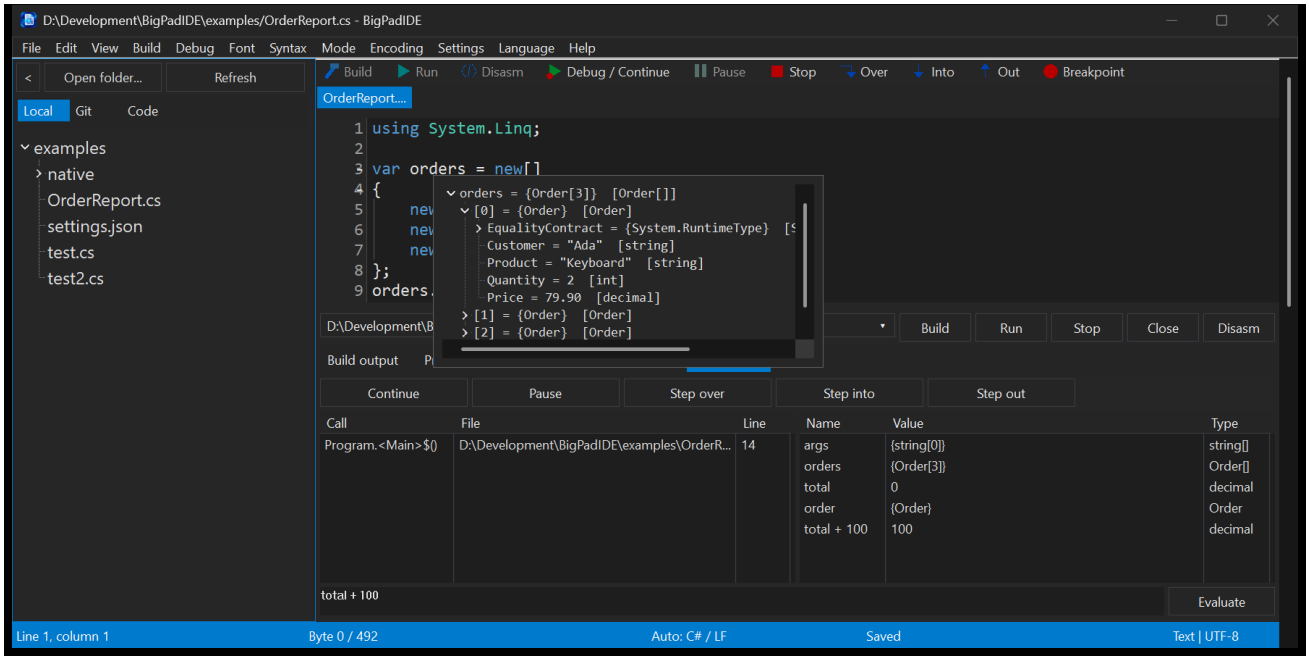


Figure 26. Expanded collection and nested object members

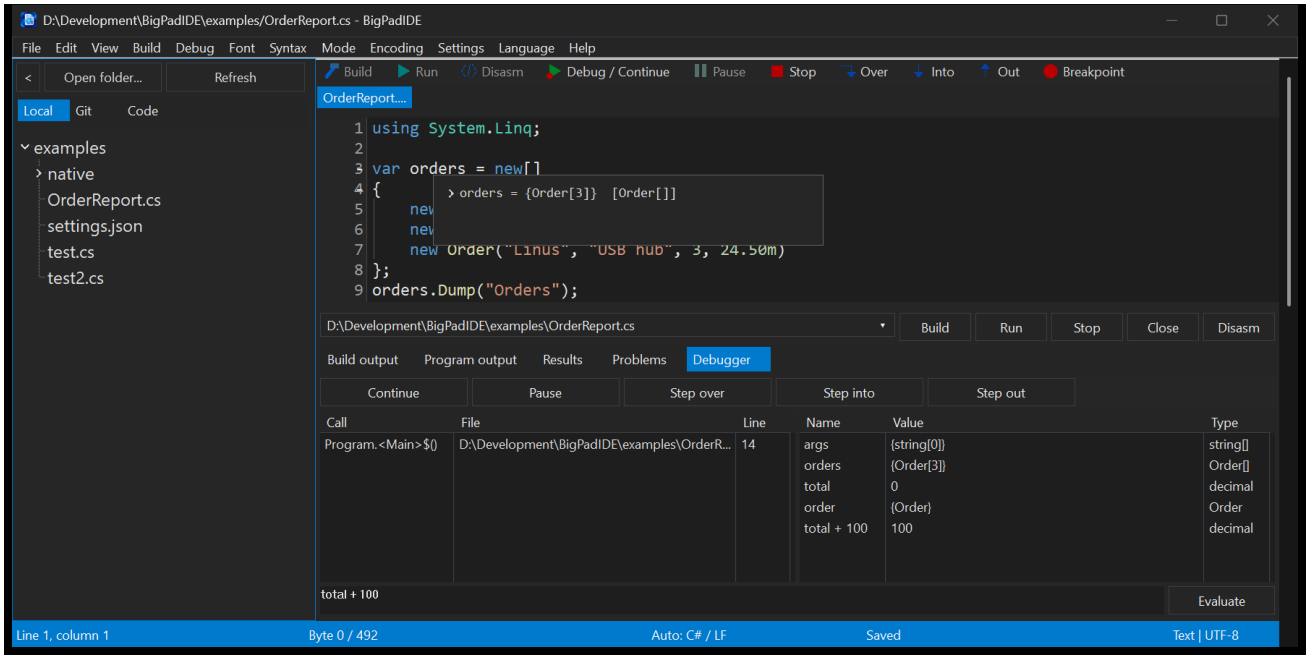


Figure 27. Actual C# runtime value above a source variable

To change a paused variable, **double-click its Value cell** or select its row and press **F2**. Enter the replacement using the debug language's syntax, for example `42` for an integer or `"Hello"` for a string. **Apply** sends the edit to the paused process; **Cancel** discards it. Source code and its undo history are unchanged.

The adapter must confirm the write before the UI reports success. Locals and existing watches are then reread. F5 continues execution using the new value. Double-click the name of an expandable value to list its members; their Value cells can be edited in the same way.

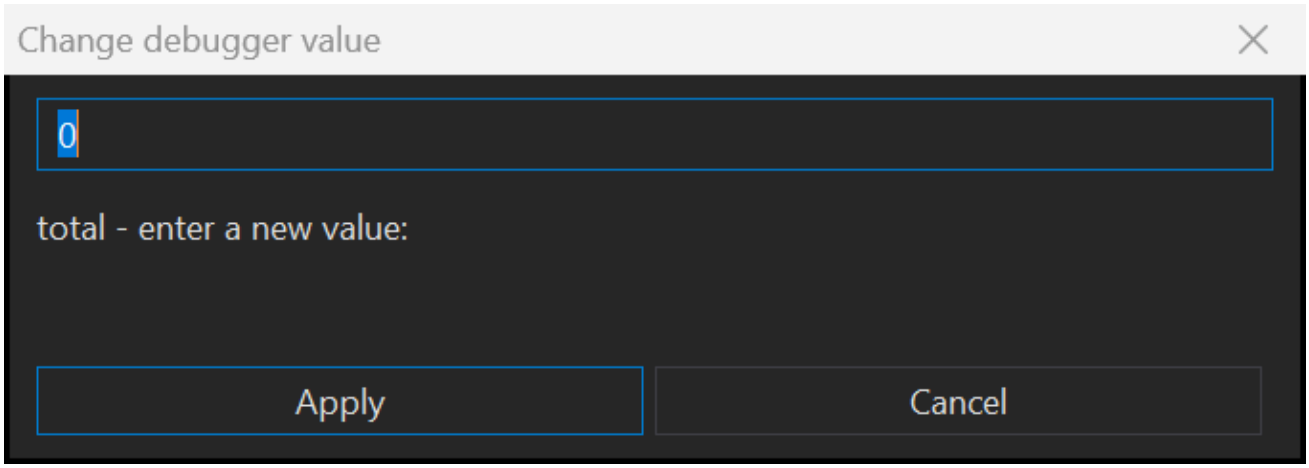


Figure 28. Dialog for changing a real paused variable

Changes affect this session, not the saved source. Read-only values, unsupported assignments and invalid inputs are rejected. Assigning a watch requires adapter support for setting expressions; otherwise edit its underlying local variable. If execution or the selected frame changes while the dialog is open, the old reference is rejected. Select the value again in the current context. Adapter errors are shown in Program output.

10. Native projects: C, C++ and assembly

Copy a native example into your own writable folder before experimenting. Select its project folder in the explorer to choose its `bigpad.ini` as the target. Configure the required compiler family and architecture as described in chapter 25.

10.1 Building and running ASM, C and C++

Open a `.c`, `.cpp`, `.cc`, `.cxx`, `.asm` or `.s`/`.S` source or a `bigpad.ini` project. The **Build** menu and output panel provide Build (Ctrl+Shift+B), Build & Run (F6), Build & Disassemble (Ctrl+F6) and Stop. Open, modified source files belonging to the selected project are saved before building. Compiler errors stop the pipeline and prevent execution of stale binaries.

Native support uses GCC/G++, NASM and GNU objcopy, located automatically at build start from PATH, `toolchains` beside the EXE and common install folders; `BIGPAD_CC`, `BIGPAD_CXX`, `BIGPAD_NASM` and `BIGPAD_OBJCOPY` can select executable paths. Compilers are not bundled, but the installer offers to download missing toolchains. C# builds, live mode and debugging remain available. Native programs have captured output and line-based stdin; native debugging is described below.

Six documented projects are in examples/native, also copied alongside built programs under `examples/native`. Five use the locally installed tools; the ARM64 raw example requires an additional cross toolchain. Project files support multiple sources, an architecture and Windows console or raw output. Artifacts are written to `.bigpad-build`. Build & Disassemble automatically opens the extracted `.text` section or raw binary with the matching decoder architecture.

For an all-users installation, toolchain packages remain available even when the installing account has its own compiler. Other accounts may not have access to it. Uninstall removes logged installed files; additional manually placed toolchain files are retained.

11. Native debugging

Native debugging requires an executable compatible with the current machine and a complete CodeLLDB installation. Cross-compiled raw ARM images can be inspected but are not automatically runnable or remotely debugged by the Windows host.

11.1 Native debugging

Run `tools/Setup-NativeDebugger.ps1` once to install the pinned, SHA-256-verified CodeLLDB 1.12.3 package under `build/debugger`. Alternatively set `BIGPAD_CODELLDB` to `extension/adapter/codelldb.exe` in a complete CodeLLDB installation; its adjacent LLDB/Python files are required. Native debugging uses the same .NET helper runtime as the C# tools. The existing installer is not repackaged.

For a native source or `bigpad.ini` target, **F9** sets source breakpoints and **F5** saves modified project sources, builds and starts the result under CodeLLDB. Compiler failures prevent launch. Debug builds append `-g -O0` after build-profile arguments; NASM requests CodeView information. **Debug → Launch native program** selects an existing executable without rebuilding it. Use matching sources and debug information; the tested toolchain is MinGW/GCC with DWARF on Windows x64. NASM source locations depend on the available symbol information. The Run profile supplies arguments, environment and working directory; its default working directory is the source/project directory. Prebuilt launches read the executable sidecar profile. Raw binaries and cross-compiled ARM targets remain disassembly targets.

The same **F10 / F11 / Shift+F11 / F8** controls step, pause and inspect native programs. The Debugger tab shows stack frames, locals, registers and Watch values; select frames or expand values by double-clicking. **Shift+F9** edits native conditions before starting a session. Source editing is disabled during debugging. An initial Windows runtime breakpoint can have no application source; Continue reaches a source breakpoint. Debug output is captured separately from debugger messages. Interactive stdin is available in Run mode; the debugger uses its input channel for debugger commands.

Debug → Attach native process selects an existing process by PID and image path, checks its creation time and attaches without rebuilding. Stop detaches and resumes attached targets; it terminates targets launched by the debugger. Refresh stale selections and use a target accessible to the current Windows account.

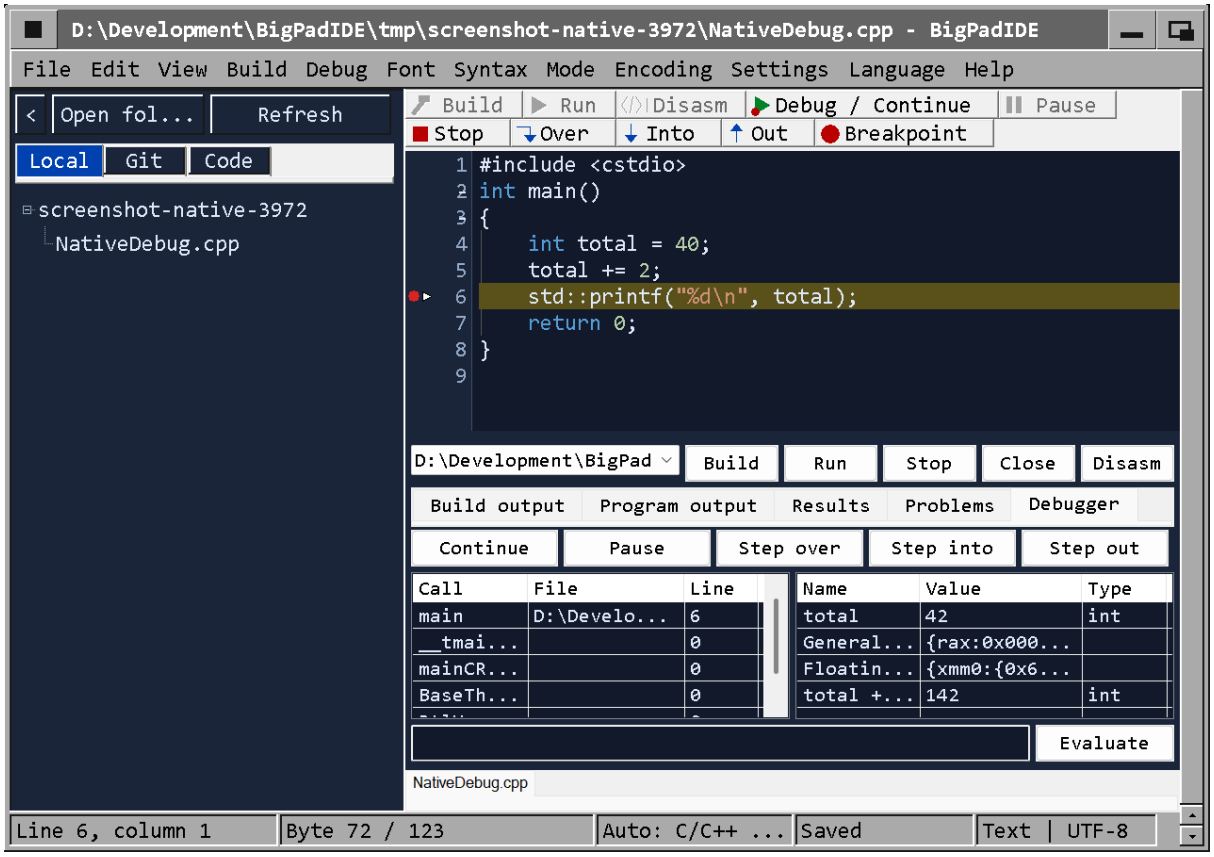


Figure 29. Native breakpoint, stack and evaluated watch

Adapter: CodeLLDB release, Windows debug formats.

12. Hex editor and byte tools

Mode → **Hex** shows offsets, byte values and their text representation. Use the same document's undo history for byte edits. Copy/paste byte pairs such as `00 FF 41`. Encoding conversion and raw byte editing are different operations; save a separate copy when experimenting with a binary format.

12.1 Hex tools: byte search, values and export

In **Mode** → **Hex editor**, use **Mode** → **Hex tools** or **Ctrl+F**. Enter complete hexadecimal byte pairs such as `DE AD 00 FF`; spaces and compact input are accepted. Search is exact, includes NUL/high bytes and unsaved edits, and is limited to 64 KiB of query bytes. **Find next / F3** begins after the selection and wraps once at EOF. Cancel stops the bounded background search; text replacement remains a text-mode tool.

The numeric list follows the selection start, or the cursor when nothing is selected. Choose **Little endian** or **Big endian** for unsigned/signed 8-, 16-, 32- and 64-bit integers and IEEE float32/float64. At most eight bytes are read; a shorter selection or EOF leaves unavailable values marked with a dash. NaN, infinities and negative zero are retained. Changing byte order only changes interpretation.

Export range uses a decimal or `0x` start offset and byte count, initially taken from the selection. Choose a new filename: existing destinations, including the source file, are preserved. Export writes the displayed logical bytes, including unsaved edits, without inserting BOM metadata or converting encoding/byte order. The complete result is published atomically after all writes; cancellation, read errors or a changed source leave no partial destination. Clean source files are held against in-place writers during export. Closing the tool cancels its work. Search/export fields keep normal text Undo and Ctrl+A independently of source Undo.

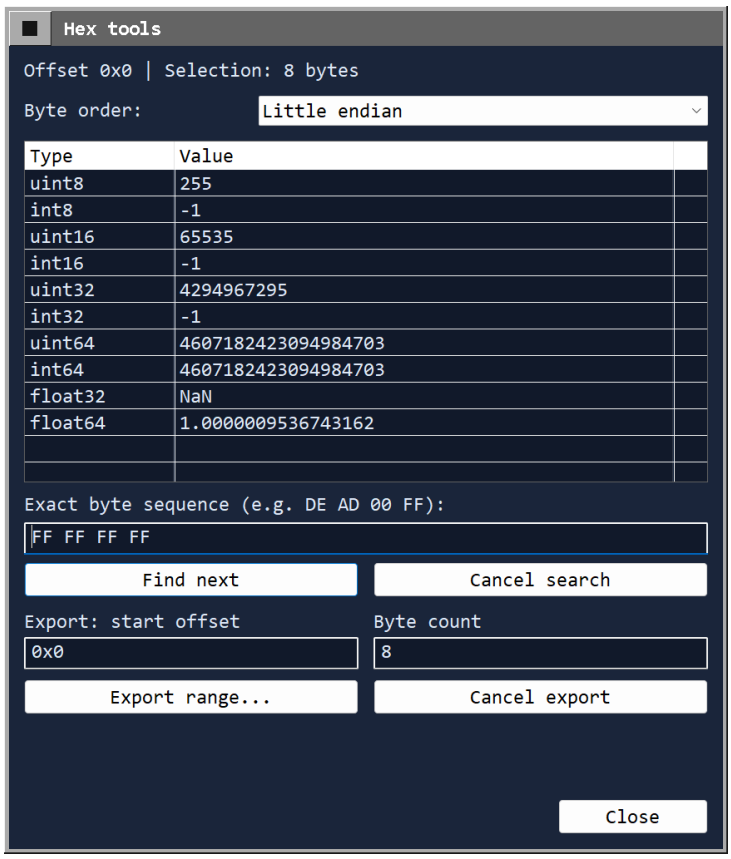


Figure 30. Byte interpretation, exact search and range export

13. Native disassembly and PE/ELF

Choose raw architecture mode for byte streams, or analyze PE/ELF for file offsets, virtual addresses, sections and symbols. These are read-only views of the logical document. An offset is not necessarily a valid instruction boundary.

13.1 Disassembler

Open a binary file and select **Mode** → **Disassembler**. Supported modes are x86 (16/32 bit), x64, ARM, ARM Thumb and ARM64. ARM modes use little endian. Each row shows the file offset, machine bytes and assembly instruction. Invalid or truncated instructions appear as `.byte`. Alternating pale and green bands span the full width like continuous paper, with matching dark-theme colors.

Use arrow keys and the mouse wheel to move through instructions, or Page Up/Down to page. Home jumps to the beginning; End shows the final file window. The scrollbar jumps to a byte offset. For x86, decoding must start at an instruction boundary; after an arbitrary jump, synchronization is ambiguous. You can also select a start offset in the hex editor before activating the disassembler. ARM and ARM64 starts are aligned to four bytes; Thumb starts are aligned to two bytes.

The view is read-only and retains its mode per tab. Switch back to **Mode** → **Hex editor** to edit bytes. Only bounded windows are decoded. This is a raw-byte view: PE/ELF sections, virtual load addresses, symbols and executable code are not automatically detected.

13.2 PE/ELF disassembly and branch targets

Mode → **Disassembler** → **Analyze PE/ELF** reads sections, preferred virtual addresses and available COFF/ELF symbols. Architecture comes from the header. The first column remains the file offset, followed by the virtual address. Addresses describe the file, without a running process's ASLR load relocation. Relative operands use virtual addresses.

Sections / symbols / VA opens a case-insensitive filter. Enter a name or an exact virtual address starting with `0x`. At most 500 matches are displayed; filtering reaches symbols beyond that display limit. Ranges without file bytes (BSS) and ambiguous mappings cannot be navigated. **Enter / Follow branch target** follows a direct jump/call at the cursor. **Backspace / Back** restores up to 256 earlier positions. Indirect register or memory targets require runtime state and are not guessed.

The six manual architecture entries restore raw file-offset mode. Mapping is per tab; restored sessions initially use raw mode and can be analyzed again. Metadata reads current logical document bytes; no external symbol download or PDB parsing is performed. PE32/PE32+ and ELF32/ELF64 are supported. ELF metadata also supports big endian; decoders retain their existing little-endian architectures. Relocatable ELF objects require a linker layout and remain raw. ELF without section headers/with extended counts and oversized metadata are explicitly rejected: 4096 sections, 50000 symbol records, 1 MiB per string table, 4 MiB of symbol names and 16 MiB total metadata reads. Malformed headers publish no partial model; raw mode remains available.

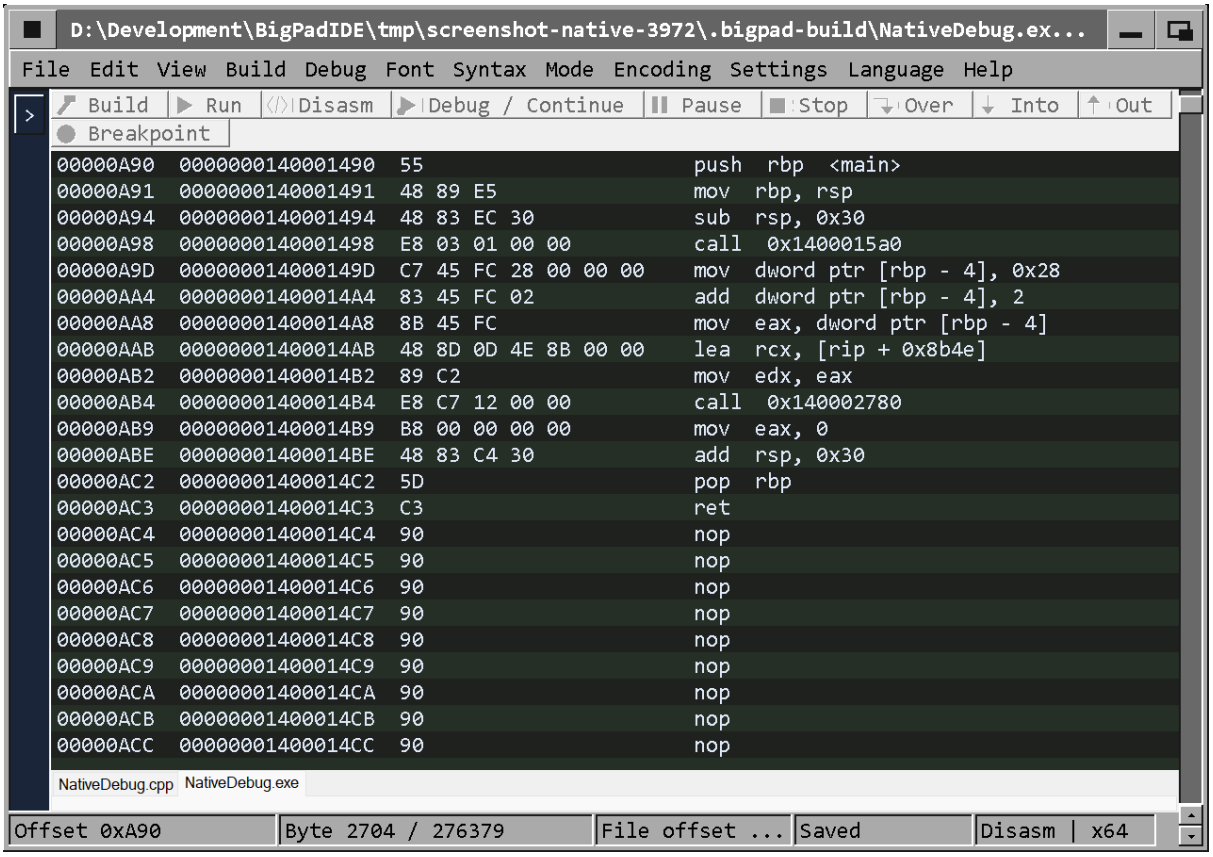


Figure 31. PE/ELF

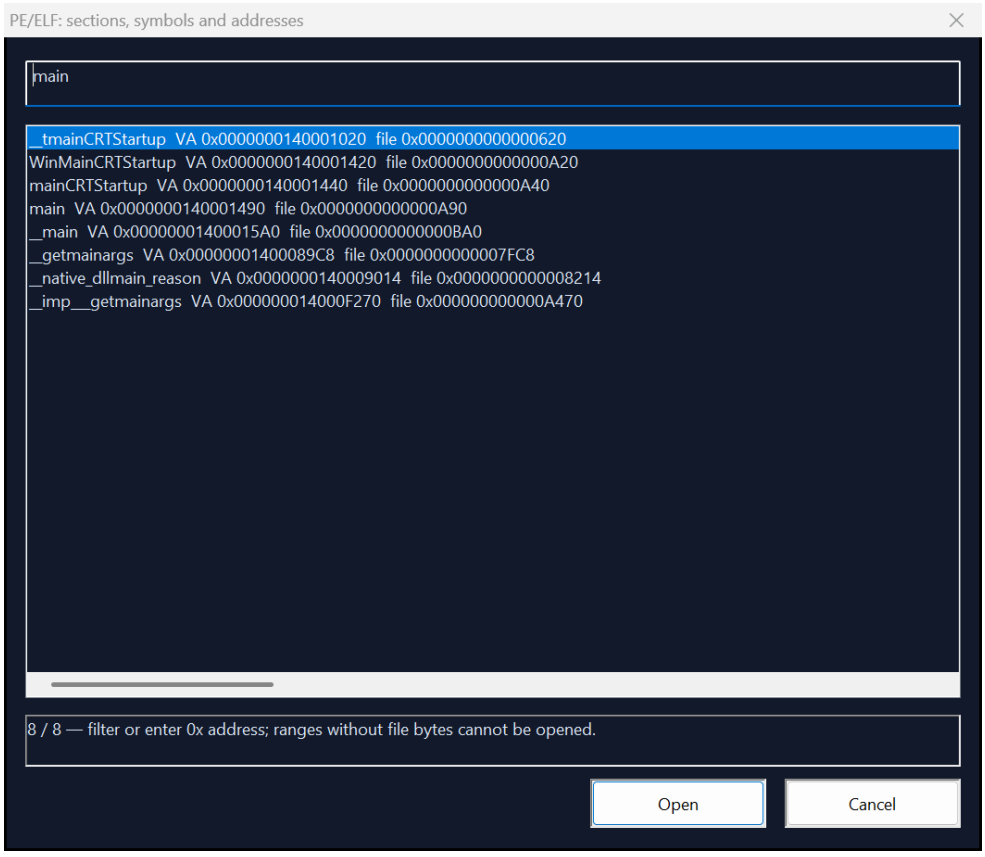


Figure 32. Symbols

14. .NET assemblies: IL and reconstructed C#

Use assembly inspection to understand compiled code without running its module initializers. Decompiled C# is a reconstruction and cannot restore comments or information removed by optimization. Keep original sources when available.

14.1 .NET assemblies: IL and reconstructed C#

Choose **Mode** → **Disassembler** → **.NET assembly: IL / C#...**. An open DLL/EXE is used as the initial file; otherwise use **Open assembly...**. Modern SDK applications usually have a native apphost `.exe`: choose their managed `.dll`. Native PE files are rejected with an error.

The code preview and type list use the configured Assembly font, inheriting the editor font when automatic. Changes in the app's Font/Size menus update an already open IL/C# viewer immediately.

The window first reads a type catalog, then previews the first user type. Select another type (including nested types) on the left; the first entry selects the whole assembly. The language dropdown switches between IL instructions with offsets and reconstructed C# classes/method bodies. **Reload** rereads the saved file. **Cancel** stops the worker and discards unfinished code. Copy from the read-only preview, or use **Open as text tab** to edit/save a separate document. Automatic C# live execution is disabled when opening generated text.

C# is reconstructed from IL; original comments, formatting and optimized-away information cannot be recovered. Missing references or unsupported/obfuscated methods can limit reconstruction; IL remains available independently. Input code and module initializers are never executed by the viewer.

Limits: 128 MiB input, 10,000 browsable types, 64 nested type levels, 4 MiB generated UTF-8 / 2,097,152 UTF-16 units, 30 seconds and 1 GiB per helper process. Choose a single type for large assemblies. A complete type catalog remains usable after a code-generation error. Assemblies and references are read from disk, not dirty tabs. The bundled ILSpy engine requires no separate ILSpy installation.

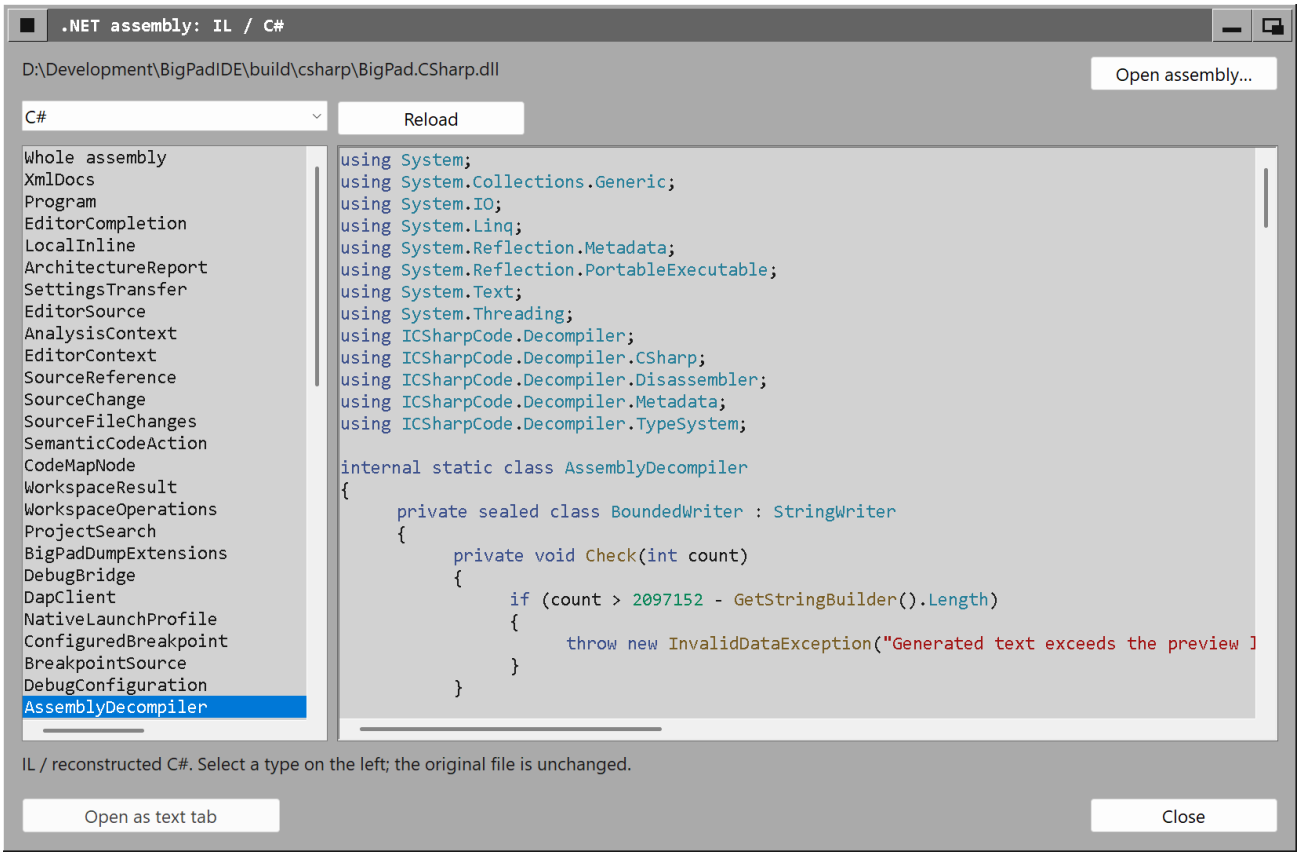


Figure 33. Reconstructed C# and browsable types

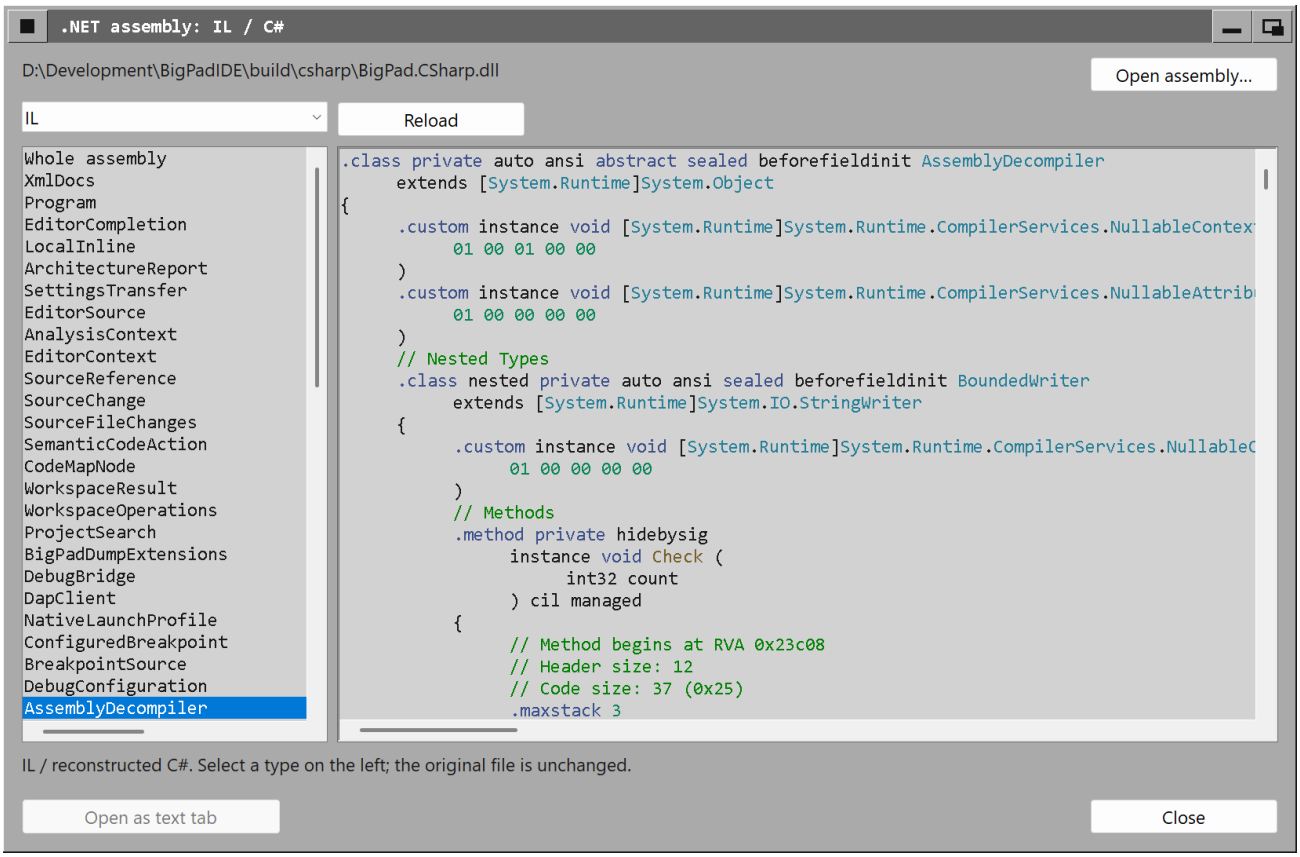


Figure 34. IL view of the same type

15. Images, frames and pixel values

Image viewing shares the tab workspace but does not modify decoded image data. Keep the original image when inspecting formats or comparing individual frames.

15.1 Viewing images

Image files (PNG, JPEG, BMP, GIF and TIFF) open in their own tabs. Images fit proportionally within the available space, with pixel dimensions shown in the status bar. GIF and TIFF display the first frame or page. The image view is read-only.

15.2 Image controls

PNG, JPEG, BMP, GIF and TIFF open as read-only image views. **View** → **Image** provides Fit (**0**), Actual size (**1**), Zoom in/out (**+ / −**) and frame/page navigation. The wheel zooms around the image point beneath the pointer. Drag with the left mouse button to pan an enlarged image; arrow keys pan by 32 device pixels. Image zoom is independent of editor font size. At 100%, one decoded pixel occupies one device pixel. Fit preserves aspect ratio and does not enlarge small images. Manual scale is bounded to 1/65536 through 64; fitting very large images may use a smaller scale. Panning stops at the image edges.

The status bar shows dimensions, zoom, current frame/page and total count. Hovering the image displays **X/Y and RGBA (0–255)** for the decoded pixel beneath the pointer. No values are invented outside the image. Values describe the current decoded frame, including transparency, rather than source palette indices or compressed bytes.

Page Up / Page Down select GIF frames or TIFF pages. **Home / End** select the first/last entry. Navigation is manual, without autoplay or wraparound. Different page dimensions are supported. Failed changes preserve the previous displayed pixels and boundary menu commands are disabled. Tabs retain independent views; newly opened or restored images start fitted at frame 1. View controls never modify or save image data.

The viewer owns a compressed snapshot and one decoded frame. Its decoder releases the source file. Limits: 256 MiB compressed input, 67,108,864 pixels per frame and 10,000 frames/pages. Decoder/memory failures are reported. PNG/JPEG/BMP are usually single images.

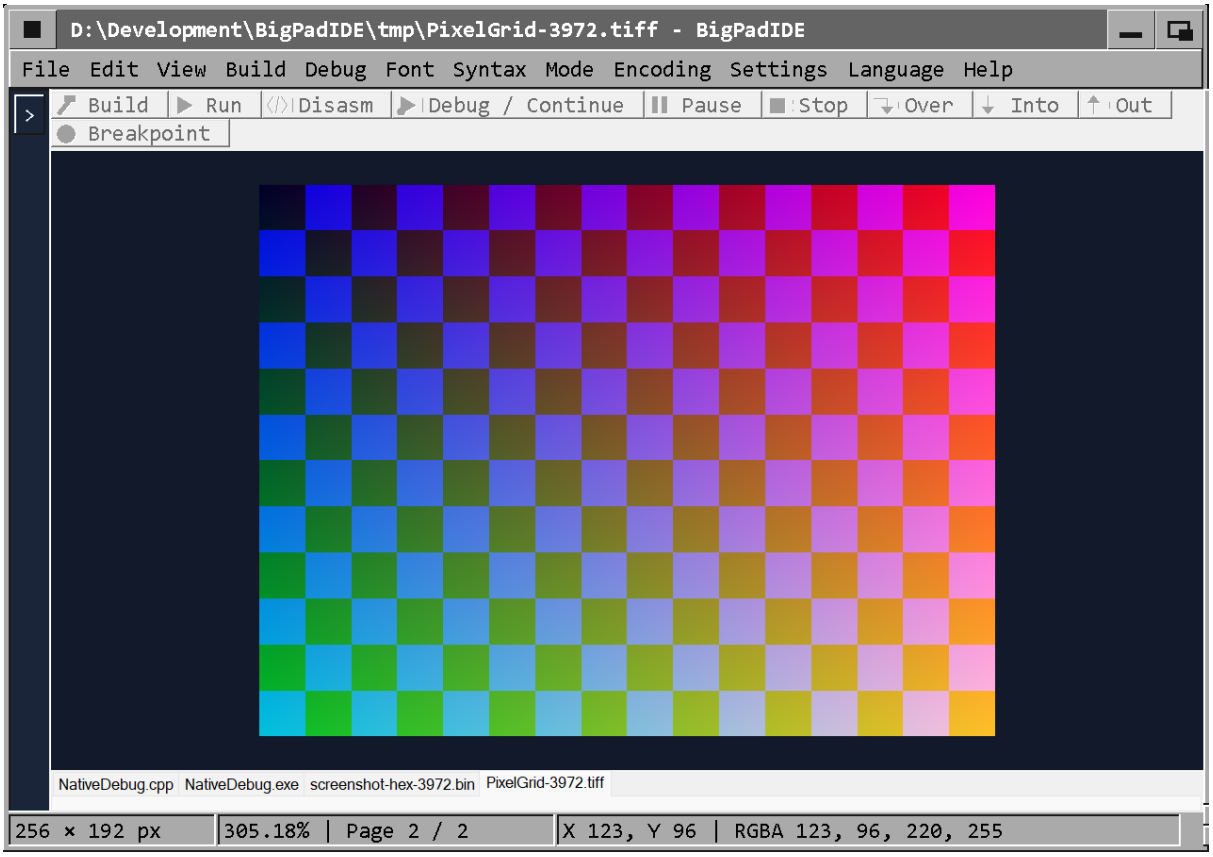


Figure 35. Zoom, pixel values and the second TIFF page

16. Git, diffs and collaboration

The Git pane operates on saved worktree files, the index and repository references. Unsaved editor changes are separate. Read the proposed operation before confirming a discard, commit, branch switch or remote write. Git authentication is configured outside the app; this pane is not a terminal for interactive password prompts.

16.1 Git changes and diffs

Install Git and make `git.exe` available on PATH, then open a local repository folder or one of its subfolders. The **Local** and **Git** tabs switch the left panel; the active tab has a blue background and white text.

The Git tree covers the entire repository and groups pending changes as **Staged**, **Unstaged** and **Untracked**. Clean committed files are omitted. Status letters include M (modified), A (added), D (deleted), R (renamed), U (conflict) and ? (untracked). A file can appear in both Staged and Unstaged when it has additional working-copy edits. A clean repository displays a no-changes message.

Click or press Enter on a staged or unstaged file to open a read-only unified diff as a tab inside the application. Staged compares HEAD with the index; Unstaged compares the index with the working copy. Added lines are green, removed lines red, and hunk headers blue. Deleted files show their removed content. Untracked files open normally in the editor. Binary changes display Git's binary-file notice.

Diff tabs can be selected, closed and displayed beside source files in split-screen views. Opening a diff retains unsaved edits in existing source tabs. The comparison uses Git's index and files saved on disk; save editor changes first to include them. F5 in the explorer refreshes the change list; clicking the same file/diff kind again updates its existing tab. Refresh in the diff, or Ctrl+Shift+R, reloads its Git state. External changes still require an explicit refresh. Missing Git, non-repository folders and read failures are reported in the panel or as an error.

16.2 Fetch, Pull and Push

In the Git panel, **Fetch / Pull / Push…** opens a dialog showing the branch, commit, upstream and destination branch. **Ahead/Behind** counts local and incoming commits using the stored tracking ref; **Fetch** updates it. **Refresh** reads only local refs. Network actions require a configured upstream and a committed branch; detached HEAD is not supported. Fetch can create a missing tracking ref.

- **Fetch** downloads from the configured upstream remote without submodules.
- **Pull (FF)** permits fast-forward updates only. Save or close every modified editor tab first; the index and working tree must also be clean, including untracked files. Stop builds, live mode, debugging and recovery dialogs first. Divergence produces an error; no automatic merge, rebase or stash is performed.
- **Push** sends the displayed commit to the displayed upstream branch. Other branches, tags, mirror and force pushes are excluded. The named remote uses its configured push URLs; multiple push URLs mean the operation affects all those destinations. Configure a missing upstream in a terminal first.

Git runs in a worker while the dialog shows elapsed time and current Git output. Editor and dialog actions are disabled during an operation; closing waits for completion or timeout. Each network command has a 120-second timeout, and each local query has five seconds. HTTP transfers stalled without data for 30

seconds are aborted. Captured output is bounded at 32 MiB; the dialog shows at most the last 64 KiB of final output.

Terminal/askpass prompts and interactive Git Credential Manager login are disabled. Set up credentials or an SSH agent in a terminal first; custom credential/SSH helpers remain subject to the process timeout. Authentication, connection and push rejection errors remain visible in the dialog. After a timeout, verify the remote: the server may already have accepted changes. Refs are reloaded after every action; closing refreshes the Git panel. Clean source tabs retain normal external-change monitoring; open diff tabs remain snapshots.

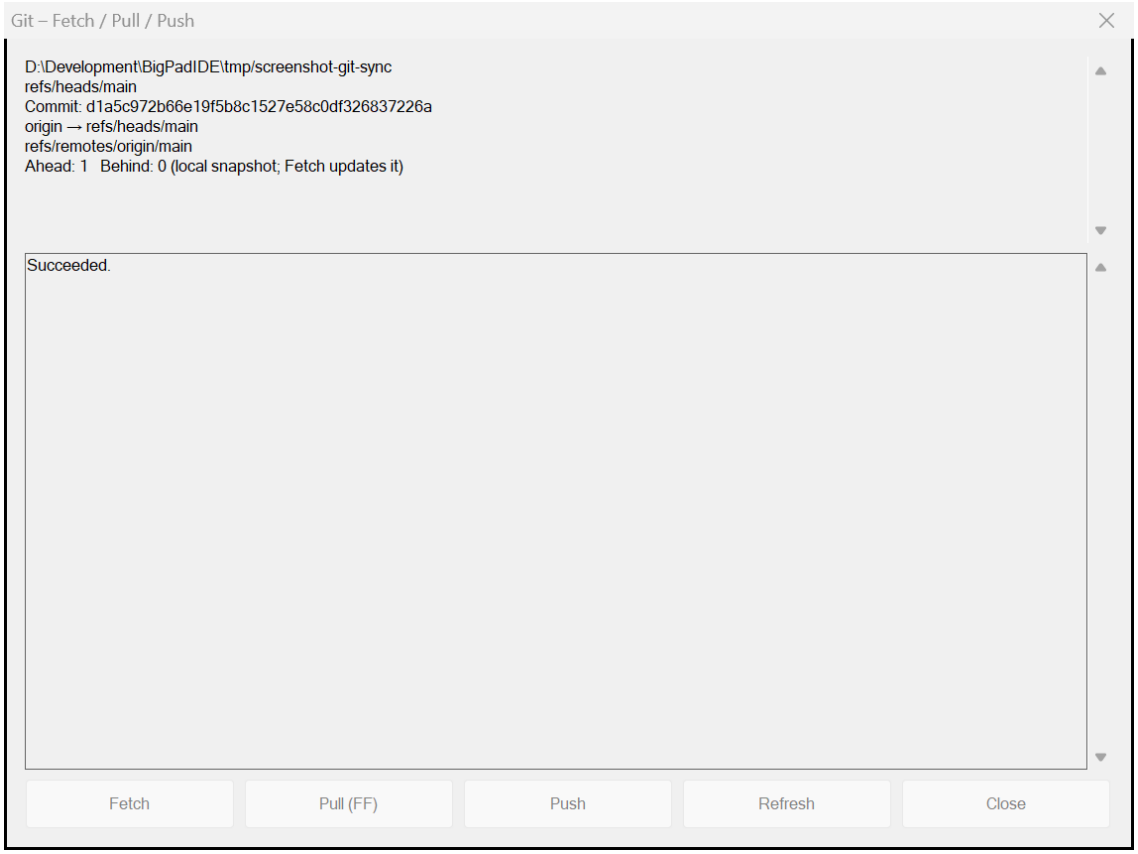


Figure 36. Git synchronization dialog

16.3 Manage branches

Branches... in the Git panel lists local branches and already fetched remote-tracking branches. Markers: `[*]` current, `[L]` local, `[R]` remote, `[W]` checked out in another worktree. Each row includes a commit abbreviation and any upstream. **Refresh** reads local refs only; use Fetch first to discover new remote branches. Symbolic remote HEAD aliases are omitted. The list is bounded at 10,000 branches.

- **Create from selection / HEAD** creates a local branch at the displayed commit without switching. With no selection, HEAD is used, including detached HEAD. No upstream is configured implicitly.
- **Rename** renames the selected local branch. Git updates the current branch and upstream configuration when applicable. Existing names are not overwritten.
- **Delete...** requires confirmation naming the exact branch and commit. The branch must be merged into HEAD and also satisfy normal `git branch -d` checks, including upstream safety. Current branches and branches occupied in a worktree cannot be deleted.

- **Switch** checks out the selected local branch. For `[R]`, the action becomes **Check out remote locally**: enter a new local name. Git creates and switches to a tracking branch with that exact remote upstream. Existing local branches are not replaced.

New names are limited to 1024 characters and must follow Git ref-name rules. `HEAD`, `@` and leading dashes are reserved. Invalid or occupied names show an error and retain the draft for correction. Existing reserved special names are not interpreted as Git shortcuts; handle them through their full ref in a terminal. Branch writes require a first commit.

The existing branch selector at the bottom uses the same protected worker flow. Before checkout, save or close all modified tabs and stop builds, debugging, live mode and recovery dialogs. The index and working tree must be clean, including untracked files. Conflicting ignored files are not overwritten. Force/merge checkouts and submodule recursion are excluded. Branches occupied in other worktrees also cannot be renamed or checked out here. Rejected actions preserve unsaved editor tabs.

Refs, current branch and worktree occupancy are revalidated before each action. Writes and hooks run in a worker with a 120-second timeout; the disabled editor prevents new edits during checkout. Closing waits for completion. Refs are reloaded even after failure: a post-checkout hook may fail after Git has already switched. Clean source tabs use normal external-change monitoring; diff tabs remain snapshots. Branch management does not publish or delete remote branches.

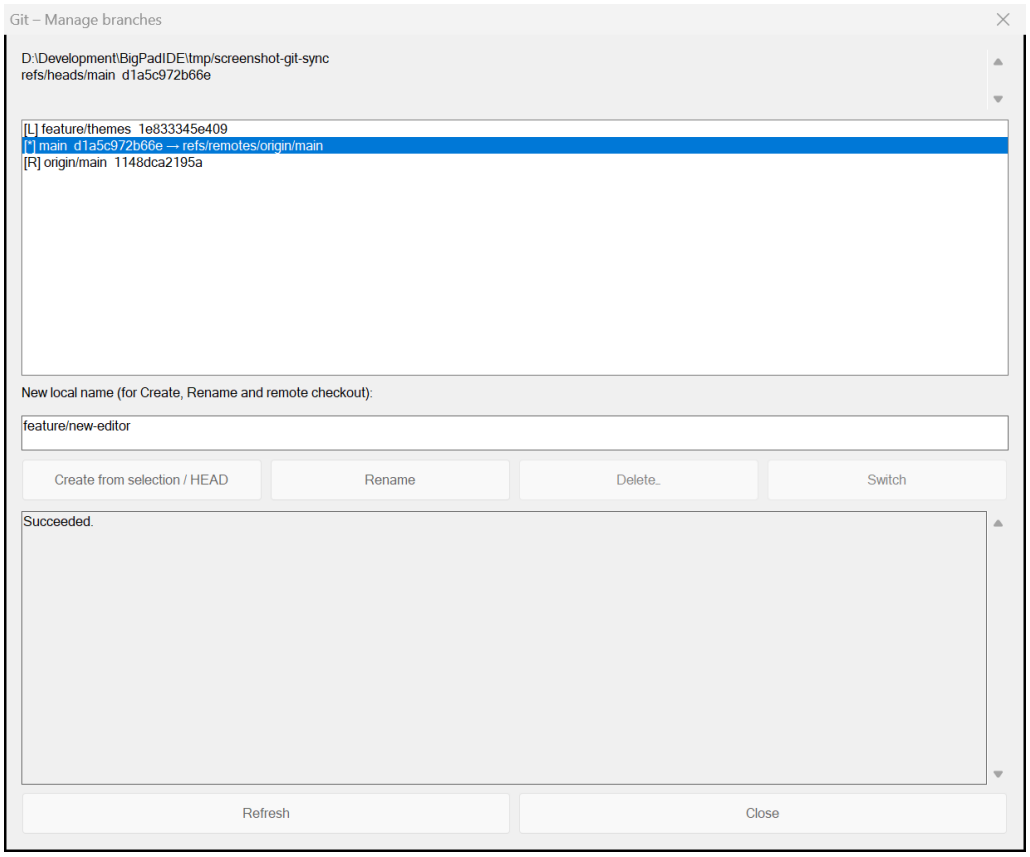


Figure 37. Local and remote branch management

16.4 Work with individual diff hunks

A Git diff offers **Refresh**, **Previous change**, **Next change**, **Stage hunk** and **Discard hunk…**. In a staged diff the index action becomes **Unstage hunk**; worktree discard is disabled there. Clicking inside a hunk selects it, and the status shows its number. Previous/Next jumps to hunk headers and wraps at either end. With focus in the diff, **Alt+Up/Down** navigates and **Ctrl+Shift+R** refreshes. Staged and unstaged diffs remain separate tabs; reopening updates the matching existing tab, including in split-screen views.

Stage copies only the selected saved hunk into the index. Unstage also changes only the index and leaves the working file intact. Discard restores only the selected working-file hunk from the index and requires confirmation naming the file and hunk. Other hunks remain intact. Before discarding, save or close every modified tab and stop builds, debugging, live mode and recovery. Staging does not save unsaved source tabs; their content and undo history remain available.

Before each write, HEAD/branch, index, working file and the complete raw diff are compared with the displayed snapshot. Stale actions are rejected and current state is reloaded. The patch is checked before applying, retaining normal Git path/context checks. The editor is disabled during writes and closing waits for completion. Each Git command has a five-second timeout; unstable reads are retried at most three times. Writes are never retried automatically. The current diff and Git list refresh after an action. Other open diff tabs can be refreshed explicitly when needed.

Original patch bytes are retained, including ANSI, CRLF and missing final newlines; normal Git attributes such as `text eol=crlf` still apply. New and deleted text files form one complete hunk each. Binary data, conflict diffs, mode changes, symlinks, submodules and special metadata do not permit hunk writes. Linked or unavailable paths are also protected. Existing whole-file Git actions remain available.

The interactive view is bounded at 1 MiB of raw diff, 20,000 lines and 2,000 hunks. Limits are reported without enabling partial writes; Git output also remains bounded at 32 MiB. A failed refresh is distinguished from a hunk already applied successfully.

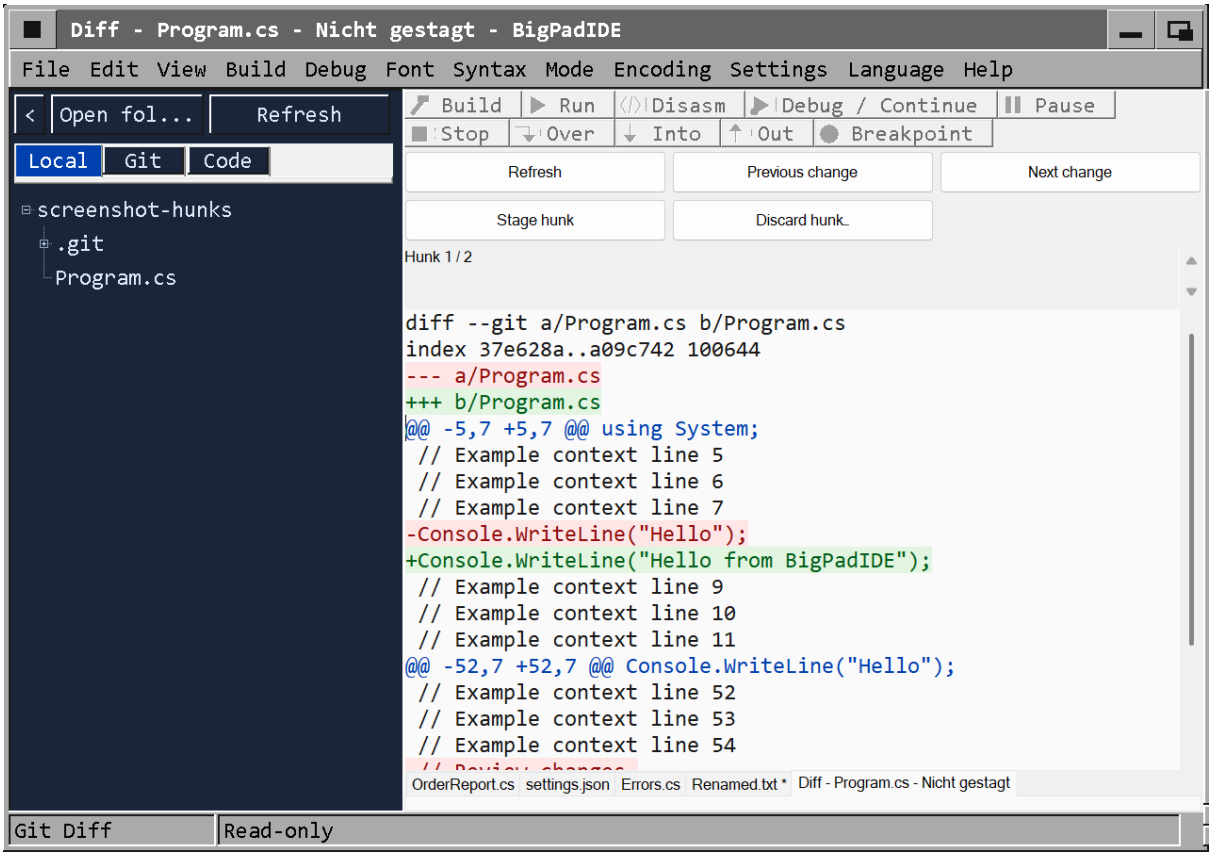


Figure 38. Hunk navigation and Git actions

16.5 Resolve Git conflicts

Click an unmerged file in the Git tree or choose **Resolve conflict...** from its context menu. **Base (1)**, **Ours (2)** and **Incoming (3)** show the Git index versions with mode, object ID and size. These labels follow the index stages of the current Git operation; review the actual content before taking a side. Missing versions are explicitly marked absent, including an add/add base or a deleted side.

1. **Edit result** opens the working file in the regular editor. Combine changes there and save; encoding, undo and normal file-safety checks remain available. If the working file is missing, first take an available side or retain deletion.
2. Alternatively, **Take ours** or **Take incoming** replaces the saved working contents after confirmation. An absent side means an explicitly confirmed deletion. Taking a side changes only the working file, leaving the conflict unmerged until reviewed. Git working-tree filters, including EOL conversion, apply.
3. Return to the conflict tab and **Refresh**, or press Ctrl+Shift+R with conflict focus. **Mark resolved** validates and stages only the reviewed saved result or its deletion after confirmation. Other files are not staged. Merge/rebase is not completed automatically; use the appropriate Git commit/continuation command.

Before writes, save or close modified tabs and stop builds, debugging, live mode and recovery dialogs. HEAD/branch, index stages and working file are compared with the displayed version. Changed previews are rejected and reloaded; failed writes are never retried automatically. The editor is disabled during writes. Git write/ filter commands have a 120-second timeout; local queries have five seconds each. Actions refresh the conflict view and Git list.

Marker checks detect line prefixes <<<<<<<, |||||||, ======= and >>>>>>> in UTF-8/ANSI data, including a UTF-8 BOM, with limits of 512 MiB and five seconds. **Explicitly retain markers / skip marker checks** permits a deliberate exception, repeated in the confirmation, for literal marker text or externally reviewed large binary results. Reopening a preview resets this option to off.

Each version preview is bounded at 1 MiB and 20,000 text lines. Larger versions retain mode/object ID/size; edit their working result in the regular editor, as direct side replacement is disabled. Binary previews show an explicit notice; use external or hex editing for the result. Symlink/submodule conflicts and linked paths remain read-only. Ordinary batch stage/reset operations do not resolve conflicts implicitly.

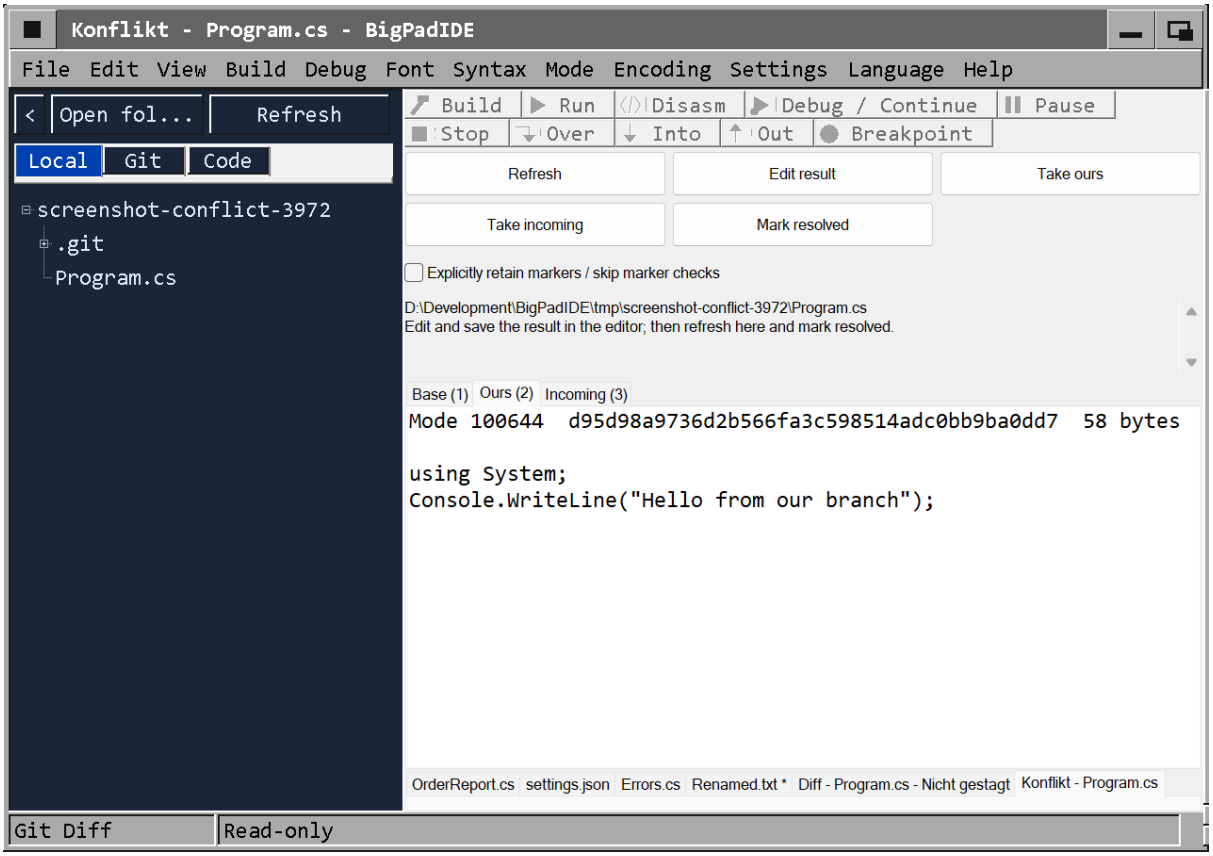


Figure 39. Base, ours and incoming conflict versions

16.6 Automatic Git refresh

In Git view, BigPadIDE watches the repository and Git metadata, including shared metadata of linked Git worktrees. Saving, external edits and renames refresh the list in batches. Window activation processes pending events; with healthy native watching, unchanged activation starts no Git query.

Events settle for 400 ms, with a maximum two-second burst delay and at least one second between queries. Active reads, Git dialogs, context menus and writes are not interrupted; pending changes are processed afterwards. Selection, expanded folders and check marks are restored where possible. Local view runs no automatic Git queries. If Windows watching is unavailable, the Git view reports this; F5 and window activation remain available without periodic idle Git polling.

17. Sessions and recovery

Session restoration remembers the workspace; crash recovery protects completed checkpoints of unsaved work. Neither replaces saving documents or keeping backups. When a recovery prompt appears, compare paths and timestamps before choosing.

17.1 Session restoration

On normal close, BigPadIDE saves tab order, active tab, each document's caret and viewport, split axes and divider ratios. The next startup restores this session before opening additional command-line files. Text, hex and disassembly views retain their mode, disassembly architecture and selected text encoding. All four styles of the shared application use the same session settings.

Missing or unreadable files are reported together. Other tabs reopen and their remaining split panes expand to fill the workspace. Positions are clamped to current file sizes and character/instruction boundaries. Empty tabs reopen; discarded unsaved contents and temporary Git diffs are not reconstructed from session metadata. Existing save prompts remain in effect. Crash recovery also retains unsaved contents as described below.

Sessions support up to 256 tabs, 32 split levels and 4 MiB of metadata. Corrupt or unknown formats are rejected with an error message.

17.2 Crash recovery

BigPadIDE automatically checkpoints modified and unnamed documents. Checkpoint files are written in the background, with at least two seconds between job starts for each document. Recovery uses the last fully completed checkpoint. Large files need proportionally more time and space on the user-profile drive.

Startup verifies abandoned checkpoints in the background and protects running instances. The prompt offers **Restore**, **Discard all** and **Keep**. Closing, Escape and the initial Enter action keep the data. Discard requires another confirmation. Corrupt or unverifiable files are identified; Restore is disabled when no valid data is available.

Recovered contents open as modified tabs labelled **[Recovered]**. Encoding, BOM, raw bytes, caret and viewport are retained. Already edited tabs are preserved. Unverifiable files remain even when valid files from the same instance are restored. If the 256-tab limit is reached, remaining checkpoints are retained for a later startup. Existing save-conflict checks protect externally changed source files.

Saving or explicitly discarding a document removes its checkpoint. Old recovered generations are removed only after a new durable checkpoint, a save or an explicit discard. Empty abandoned instance directories are cleaned at the next startup. Failures appear in the title and status bar; click the status bar for details. Failed document jobs can be retried immediately there, or retry after 30 seconds.

The shared application uses `%LOCALAPPDATA%\BigPadIDEWorkbench\Recovery`; older Classic installations may use `%LOCALAPPDATA%\BigPadIDE\Recovery`. Data stays local and is cleaned on normal close after save/discard decisions.

17.3 Session capacity when opening files

Up to 256 tabs are supported, including temporary diff tabs. At capacity, existing files can still be activated and an empty tab can be replaced by a file. Splits reuse a hidden tab first. New tabs, overly deep splits and paths exceeding the session budget are rejected before changing the workspace. New Save As paths are subject to the same check.

17.4 Remember window and folder width

On normal exit, BigPadIDE saves the window position, normal window size, maximized state and the folder panel's last dragged width. The next launch restores them, for the shared application. A window closed while minimized reopens normally or maximized. Obsolete off-screen positions are adjusted to available monitors. A failure to save prevents closing so it can be retried.

18. Practical workshops

18.1 Inspect and change a file safely

1. Open a copy of a small JSON or text file. Observe encoding and line endings.
2. Find a term with Ctrl+F; change it and undo the change.
3. Save As a second name, reopen it and compare the result.
4. Switch to Hex to inspect the actual bytes, then back to Text.

Expected result: the original stays available and you can distinguish bytes, encoding and display-only word wrap.

18.2 A complete build/debug cycle

1. Run the C# example from chapter 1 and check the exit code.
2. Add a variable and a breakpoint before printing it.
3. Start with F5, inspect locals, evaluate an expression, and step with F10.
4. Stop with Shift+F5. A debugger session makes source code read-only.
5. Drag the output panel higher if values need more room.

Expected result: you can identify the active target, execution line, values and program output without confusing a breakpoint marker with the current instruction.

18.3 Review a Git change

1. Work in a disposable repository or branch and save a small source edit.
2. Open the Git pane, review the diff and stage the intended file or hunk.
3. Inspect the staged diff and commit locally with an explanatory message.
4. Fetch and inspect upstream state before choosing a remote operation.

Expected result: only reviewed changes enter the commit. A local commit does not automatically push it to a remote repository.

18.4 Make a portable personal profile

Choose a style, set comfortable fonts, resize the panel and configure a toolset. Export Settings as JSON, make a reversible appearance change, then import the file. Check paths before using that profile on another machine. Project files, credentials and license activation are separate from the settings snapshot.

19. Troubleshooting

Symptom	What to check
Build starts the wrong project	Select its folder/project in the explorer and inspect the bottom target dropdown
Compiler or SDK not found	Open Settings → Toolsets and tool paths; verify explicit paths and architecture
Breakpoint does not bind	Rebuild with symbols, check source paths and target, and read debugger output
Register value or path is shortened	Enlarge the output panel, widen the column, or hover the cell
Characters look wrong	Verify UTF-8 versus Windows ANSI; changing the font does not change bytes
A source edit is missing from Git or project search	Save first; those operations read the saved file
File changed outside the app	Compare the external version; choose reload, Save As or explicit overwrite
RAM mode cannot be selected	Check available memory and whether a source changed during conversion
A generated C# preview differs from original source	IL reconstruction cannot restore comments or optimized-away data
Interactive program cannot use terminal controls	Program input is a line-based pipe, not a full console terminal
A shared JSON profile cannot build	Tool paths are machine-specific; project sidecars and source files are separate
F1 opens the wrong language	Choose Language in the program; an already open browser tab can be switched in its header
Activation or trial is rejected	Check system date, distribution and the displayed license status; contact the publisher

19.1 Report a reproducible problem

Include the program version, Windows version, chosen toolset and architecture, expected result, actual result and exact error text. Add a minimal example and a screenshot if helpful. Remove credentials, keys and private source paths before sharing. Contact: rstepan123@gmail.com.

20. Keyboard and menu reference

Shortcuts apply to the focused view. Click the intended area first when a key has a different effect in the explorer, an input field, the editor or an image.

Shortcut	Action
Ctrl+N / Ctrl+O / Ctrl+Shift+O	New document / open file / open folder
Ctrl+S / Ctrl+Shift+S	Save / Save As
Ctrl+Shift+R	Reload from disk; refresh a focused diff/conflict view
Ctrl+W	Close current tab
Ctrl+Tab / Ctrl+Shift+Tab	Next / previous tab
Ctrl+P / Ctrl+G	Quick open / go to line or offset
Ctrl+Home / Ctrl+End	Start / end of document
Ctrl+A/X/C/V	Select all / cut / copy / paste
Ctrl+Z / Ctrl+Y / Ctrl+Shift+Z	Undo / redo
Ctrl+F / Ctrl+H / F3	Find / replace / find next
Ctrl+Shift+F	Search project files
Ctrl+Space / Ctrl+Shift+Space	C# completion / signature help
Ctrl+I / F12	C# symbol information / source definition
Shift+F12	Find all C# references to the symbol
Ctrl+click	Open the C# definition under the pointer
Ctrl+B / Ctrl+Shift+E	Collapse explorer / focus explorer
Ctrl+R or F5 in explorer	Refresh local or Git tree
F2 in local explorer	Rename selected entry
F2 in debugger variables	Change the selected runtime value
Space in Git tree	Select a file for a multi-file action
Enter in explorer	Open file or diff
F7	Focus next split area
Ctrl+J	Show or hide output panel

Shortcut	Action
Ctrl+mouse wheel	Change the font under the pointer
Drag the output panel's upper edge	Resize the panel without changing fonts
Ctrl+Shift+B / F6 / Ctrl+F6	Build / Build & Run / native Build & Disassemble
F5 / Shift+F5	Start or continue debugging / stop or detach
F9 / Shift+F9	Toggle breakpoint / edit its condition
F8 / F10 / F11 / Shift+F11	Pause / step over / step into / step out
Ctrl+Enter in commit dialog	Create a local commit
0 / 1 / + / - in images	Fit / 100% / zoom in / zoom out
Page Up / Page Down in images	Previous / next frame or page
Enter / Backspace in analyzed disassembly	Follow direct branch / return
F1	Local help in the selected program language

File manages documents; Edit manages editing/search; View manages layout and images; Build manages execution and C# tools; Debug manages sessions and breakpoints. Font, Syntax, Mode and Encoding control the current document. Settings includes style, fonts/colors, JSON, storage, toolsets, view placement and saving preferences.

21. Limits and stored data

Feature-specific limits are documented in the relevant chapters. They are working budgets, not guarantees about arbitrary formats or available system resources. MiB means 1,048,576 bytes. Operations report skipped inputs or exhausted limits.

Data	Location or scope
Documents	Paths chosen by the user
Native build files	`.bigpad-build` under the project folder
Build/run profiles	`<target>.launch.ini` beside the target
Preferences, history, sessions	Local Windows user settings
Recovery	Local application-data recovery directory; older variants may use separate roots
JSON configuration	Chosen export path; excludes license and trial state
C# analysis context	Current editor instance
Active project	Current workspace; not a new solution-file property

Editor/hex export uses logical bytes including edits. Project search and Git read saved files. C# tools combine a loaded context with supported editor snapshots. Assembly inspection reads assemblies from disk. Recovery uses its last completed checkpoint. These sources can legitimately show different states until saved or refreshed.

22. Glossary and further help

Term	Meaning
Active project	Target for build, run, debug and launch profiles
Analysis context	C# project/solution references used by language tools
BOM	Optional UTF-8 byte-order marker at the start of a text file
CodeLLDB / NetCoreDbg	Native / managed debug adapters
Hunk	One section of a Git diff
Index	Git's staged content for the next commit
Piece table	Document representation based on original and inserted byte ranges
RAM backend	Contiguous in-memory document buffers and retained history
Sidecar	A related settings file beside a project or source
Toolset	Named compiler family, architecture and tool paths
Virtual address	Address in an analyzed executable image, distinct from a file offset

F1 opens local help in the selected program language. The header changes language or opens the matching PDF. Both outputs are generated from the same language's manual source. The images are direct application captures, not illustrated mockups.

Developer: **Rudolf Stepan**. Contact: rstepan123@gmail.com. Project: BigPadIDE. Third-party notices belong to the installed distribution.

23. Trial, purchase and product keys

23.1 Trial and purchase entitlement

Without a valid full license, the first start begins a 14-day trial. Its beginning and latest observed time are protected per Windows user. Restarting, updating or reinstalling does not start another trial. Damaged state or a significantly rolled back clock is not silently treated as a new installation.

After the trial, continuing work requires a valid entitlement. If access expires during a session, open documents remain available for saving, Save As and closing. Save your work before leaving. The app shows status and a licensing option.

A recognized full Microsoft Store purchase in the correct package needs no extra product key. A copied executable does not inherit the package's Store entitlement. Availability and purchase terms are those of the actual published offer.

23.2 Activate a product key

Signed 30/90/365-day and perpetual keys are supported for separate EXE distribution. The variants actually offered are specified in your purchase.

1. Open **Help** → **Product key / license**.
2. Confirm activation or replacement and paste the complete supplied `BP1.…` key.
3. Check the customer name and expiry time in UTC.

Validity begins at issuance, not installation or first entry. An existing activation resumes automatically. Already used license IDs cannot be manually activated again; renewal requires a newly issued key. Do not include a complete key in a public report.

23.3 Move settings or ask for help

JSON settings transfer does not carry purchase entitlement, activation or trial state to another user. Use the activation method for your distribution and contact the publisher for account changes or rejected activation. Include version, distribution, Windows version and the exact message, without sending private keys.

24. Make the workspace your own

24.1 Interface style and horizontal menus

Settings → **Interface / style** switches between Classic (Windows), Workbench (Retro), Modern - Light and Modern - Dark in the same app. Open documents and unsaved edits remain intact. Open a top-level menu once, then move horizontally over another heading, such as Encoding to Settings. It opens without another click. Escape closes the menu.

24.2 Independent fonts and application colors

1. Open **Settings** → **Fonts and colors**.
2. Choose one of 13 areas, then **Choose font** for face, size and style.
3. Choose a color area and original RGB value, then **Choose color**.
4. **Apply** previews the changes; **OK** applies and closes.
5. Reset the selected area, selected color or all appearance overrides as needed.

Areas include Editor, Explorer, Menus, Toolbar, Status, Tabs, Panels, Output, Dialogs, Assembly, Hex, Diff and Search. Colors with the same original RGB value inside one area share the override. Windows-owned title bars and common file dialogs follow Windows settings. Some open dialogs use the new font after reopening.

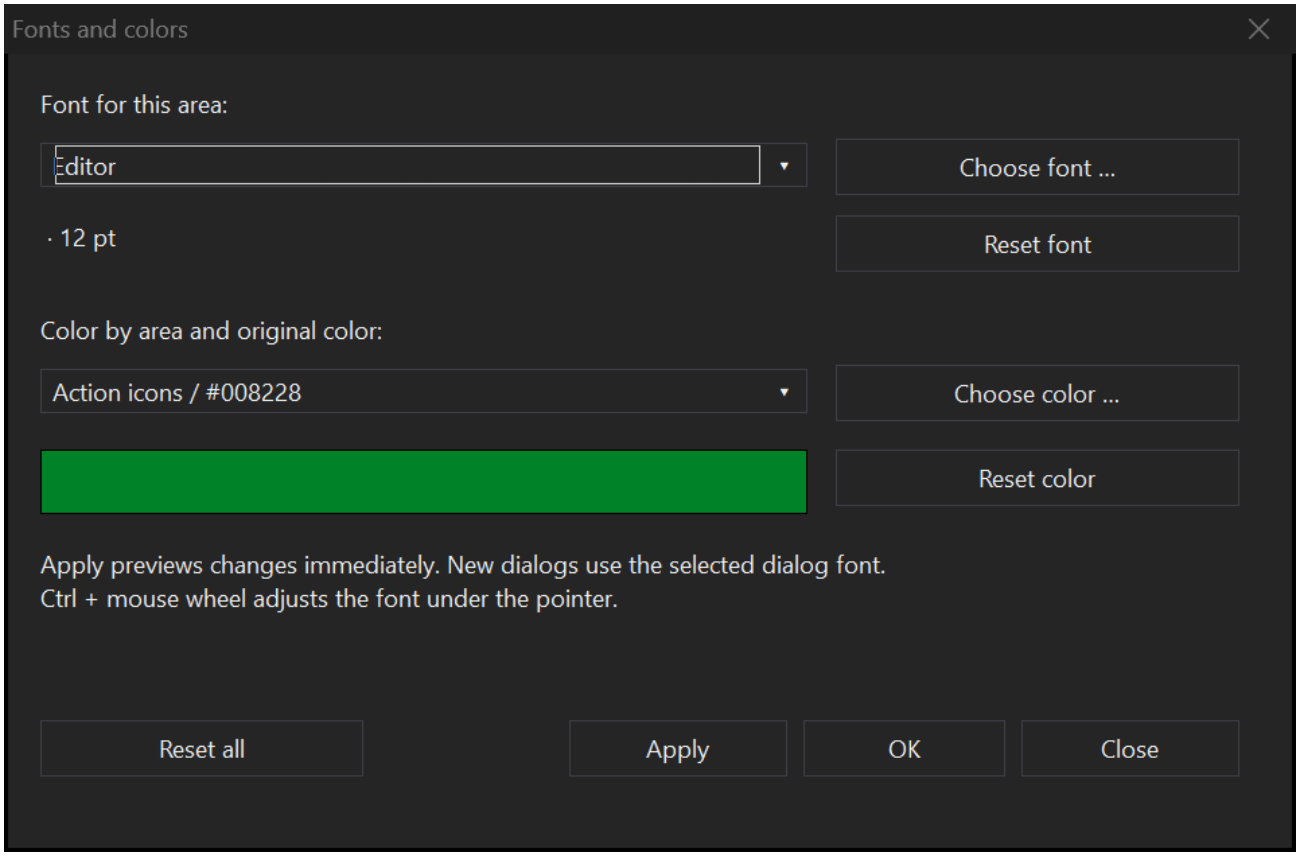


Figure 40. Independent fonts and colors dialog

24.3 Ctrl+mouse wheel and panel size

Move the pointer over an area, hold Ctrl and rotate the wheel. Full detents change one point; high-resolution partial movements accumulate. Sizes range from 6 to 72 pt. Explorer, editor and output text are independent. Changes are saved. Without Ctrl, the wheel continues to scroll normally. Dragging the output panel's upper edge changes its physical height without changing any font.

Debugger tables adapt to content and width. Hover shortened cells for tooltips; manually adjusted columns remain until the next session. A source jump clears a stale selection while preserving the yellow execution-line marker.

24.4 Import and export the complete settings as JSON

Settings → Export settings as JSON writes a versioned UTF-8 snapshot of saved preferences, fonts, colors, toolsets, tool paths and workspace/window settings. **Import settings from JSON** validates before replacing that settings snapshot. Export your own profile before importing another. Invalid data is rejected; write failures trigger restoration of the previous snapshot. The file limit is 4 MiB.

Unsaved document bytes are not in this export; use Save and recovery. Project-local `.launch.ini` sidecars travel with the project separately. License, trial and activation data are excluded. Paths may not exist on another machine; inspect Toolsets and tool paths after importing. Check private path names before sharing.

24.5 Piece table or complete RAM buffers

Choose **Settings → Document storage mode → Piece table (file backed)** or **Fully in RAM (files and undo history)**.

Mode	Suitable when	Resource cost
Piece table	Large documents should not require a complete RAM copy	Original backing and temporary original snapshots use disk space
Fully in RAM	Every open file should be fully resident	Full contents and retained undo versions require memory

Switching prepares all open documents and undo/redo states, then commits them together. Content, dirty state, breakpoints, cursor, selection and scroll positions remain intact, including inactive tabs. Allocation failure or a source changing during conversion rejects the operation; there is no silent backend fallback.

Text, wrapped text, hex and disassembly scrolling use the same logical positions in both modes. The switch does not jump to the start. RAM buffers share unchanged snapshots and copy shared data before editing. Switching back preserves undo history, which may still occupy RAM. Save and reopen for a fresh file-backed document.

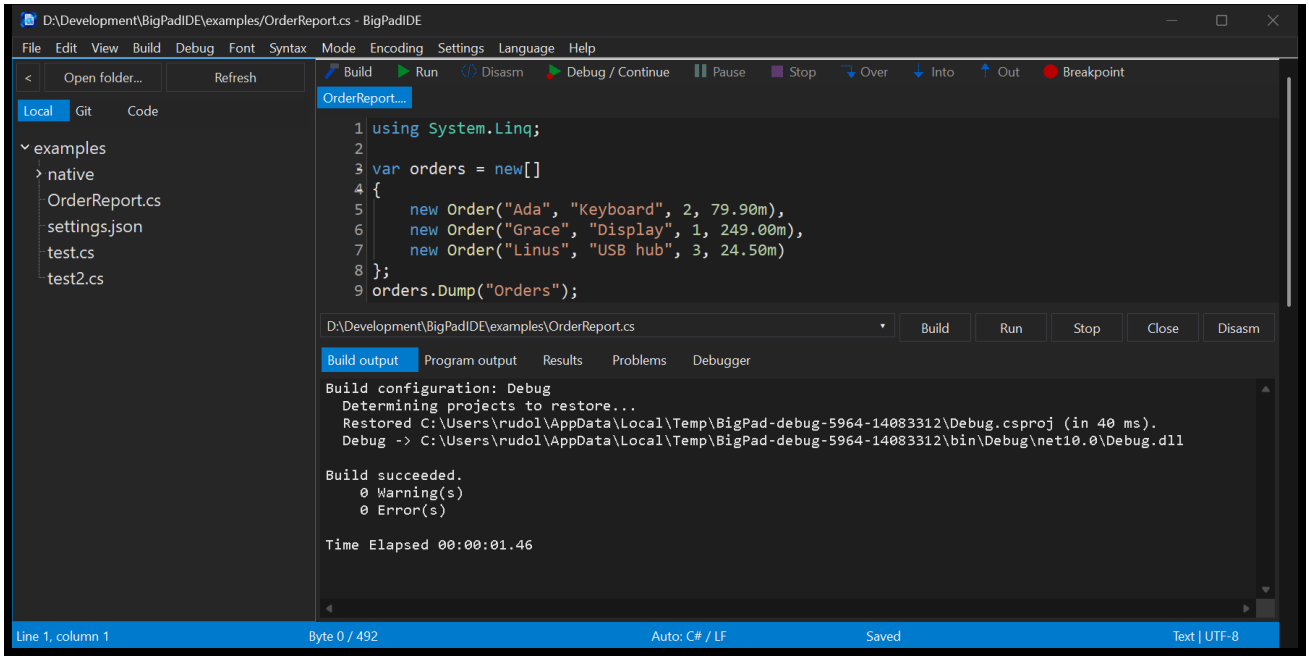


Figure 41. Workspace after choosing complete RAM buffers

24.6 Build a comfortable personal profile

Start with a readable style and fonts. Adjust output height, then colors. Build a small project, inspect the result and export a JSON backup. On another machine, import it and update machine-specific paths. This keeps a familiar workflow while leaving your source projects and license separate.

The workspace also remembers whether the output panel was open or closed at normal shutdown. The next start restores that state and its chosen height. This session information is saved even when automatic preference saving is disabled. Starting a build or debug task can still open the panel when needed.

25. Toolsets, compilers and build configuration

Open **Settings → Toolsets and tool paths** to manage named compiler profiles per target architecture. Settings are stored for the current Windows user independently of the open source file.

Create a profile with **New**, choose **GCC / MinGW**, **Microsoft MSVC** or **Intel oneAPI (icx-cl)**, and choose the architecture. **Detect installation** imports MSVC or Intel compiler/assembler/linker paths together with the required VC, Windows SDK and Intel headers/libraries. For GCC, select the compiler's `bin` folder.

Select a property in the table to edit its value below. **Browse** selects a file or folder; **Apply value** updates the draft. Include, library and additional PATH directories accept one absolute directory per line, in search order. Mark **Active for target architecture**, then **Check** and **Save**. Multiple profiles can be stored for each architecture; a star marks its active profile. **Duplicate** makes an independent copy. Finish running builds/debug sessions before switching toolsets.

Blank executable fields use the default name in the configured `bin` directory. Explicit paths override automatic discovery and never silently fall back when missing. With no active profile, existing environment/PATH/installed-toolchain discovery remains available. Additional PATH directories affect build children, not the Windows system environment.

Common tools configures the .NET SDK host, NetCoreDbg, CodeLLDB, Git and the C# helper directory. SDK selection applies to build/run, analysis and debugging, including nested .NET processes. The selected SDK installation must provide the runtime needed by the shipped helper; the installer ships .NET 10. Blank fields restore defaults.

GCC profiles use GNU-style arguments and NASM for `.asm`. MSVC/Intel profiles use MSVC-style arguments, `link.exe` and MASM (`ml64.exe`/`ml.exe`). Changing compilers does not translate NASM/MASM syntax. GNU `.s`/`.S` files require a GCC profile. Configure GNU/LLVM `objcopy.exe` explicitly for MSVC/Intel **Build & Disassemble**. The GCC linker field expects a compiler driver, usually `g++.exe`. MSVC/Intel builds support Windows x86/x64 (MSVC) or x64 (Intel) console projects; GNU ARM profiles retain raw-binary support.

25.1 Build configuration and compiler options

Open **Build > Build configuration and compiler options** to choose **Debug** or **Release** for the current project or file. Build, Run and debugging use this selection, including C#. Release enables optimization; debugging Release retains optimization and adds debug symbols.

Build accepts compiler or MSBuild options, such as `-p:DefineConstants=MY_FEATURE`. Native projects have separate **Linker options** and **Assembler options** profiles. Arguments use Windows quoting. **Run** holds program arguments. Settings are saved in the associated `.launch.ini` file; older configurations remain supported.

Mixed GCC/NASM debug builds repair GNU ld PDB file information so CodeLLDB resolves both C++ and ASM breakpoints regardless of source path length.

25.2 Installer

The installer offers optional GCC/ARM downloads plus unchecked MSVC and Intel tasks. MSVC downloads a fixed, SHA-256-verified Microsoft bootstrapper and opens its C++ setup. Intel opens the official compiler download page for platform/package selection. Vendor setup retains its own license terms and elevation prompts. These two tasks require an interactive installation. Afterwards, detect/import and activate a profile in BigPadIDE. The BigPadIDE uninstaller does not remove separately installed Microsoft/Intel products. CodeLLDB and the .NET SDK are included with the BigPadIDE package.

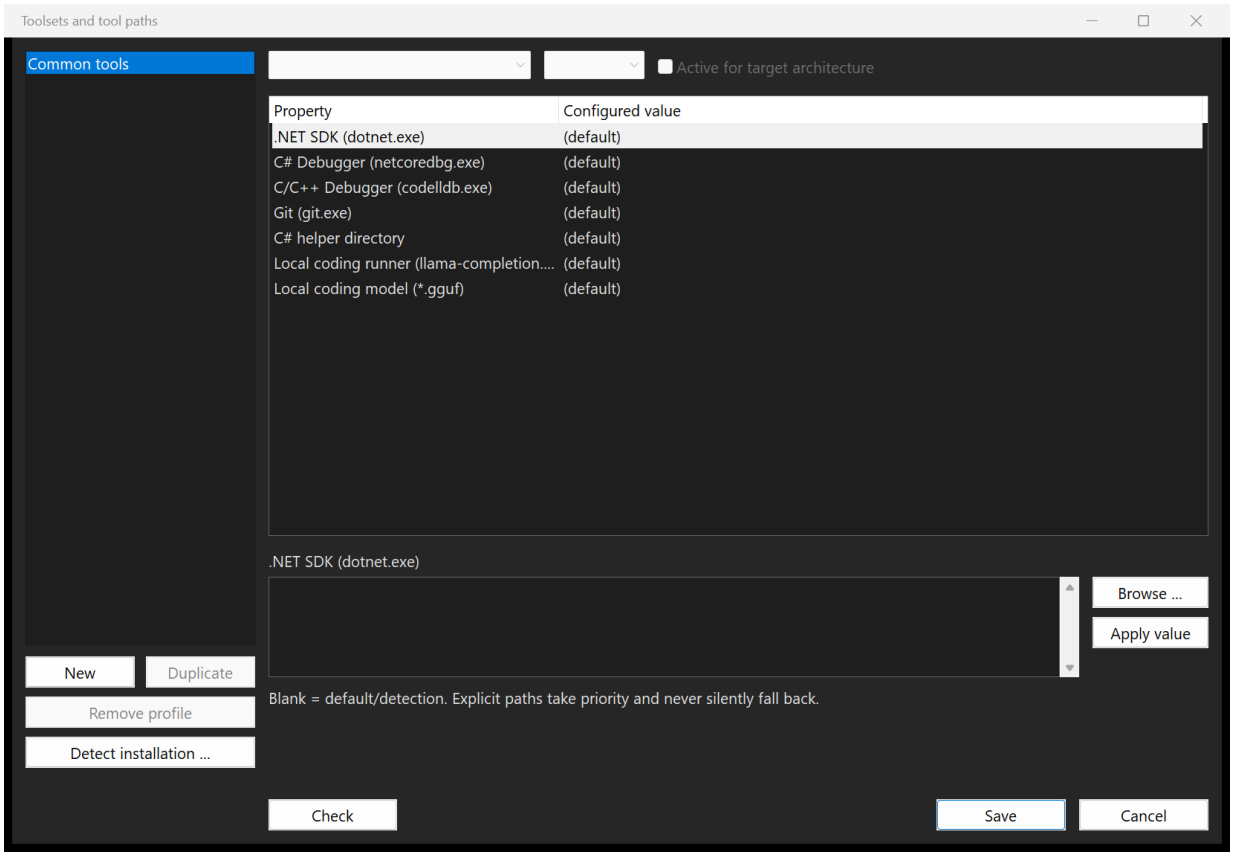


Figure 42. Toolsets and shared tool paths

26. Semantic code assistance and local AI

26.1 C# completion and gray suggestions

C# assistance uses Roslyn completion providers for project/solution context, framework members, inferred types and extension methods. Select the correct C# project context and inspect build errors for reliable project results. A quick local list is enriched asynchronously with semantic results. This does not imply complete feature parity with Visual Studio.

Gray text after the caret is a suggestion and has not changed the document. **Tab** accepts it as one undo action; **Esc** dismisses it. Up/down arrows instead select an individual completion item, and **Enter** accepts that name. Moving the caret or changing the document discards stale suggestions. Debugger value tips and code assistance use smaller typography derived from the editor scale.

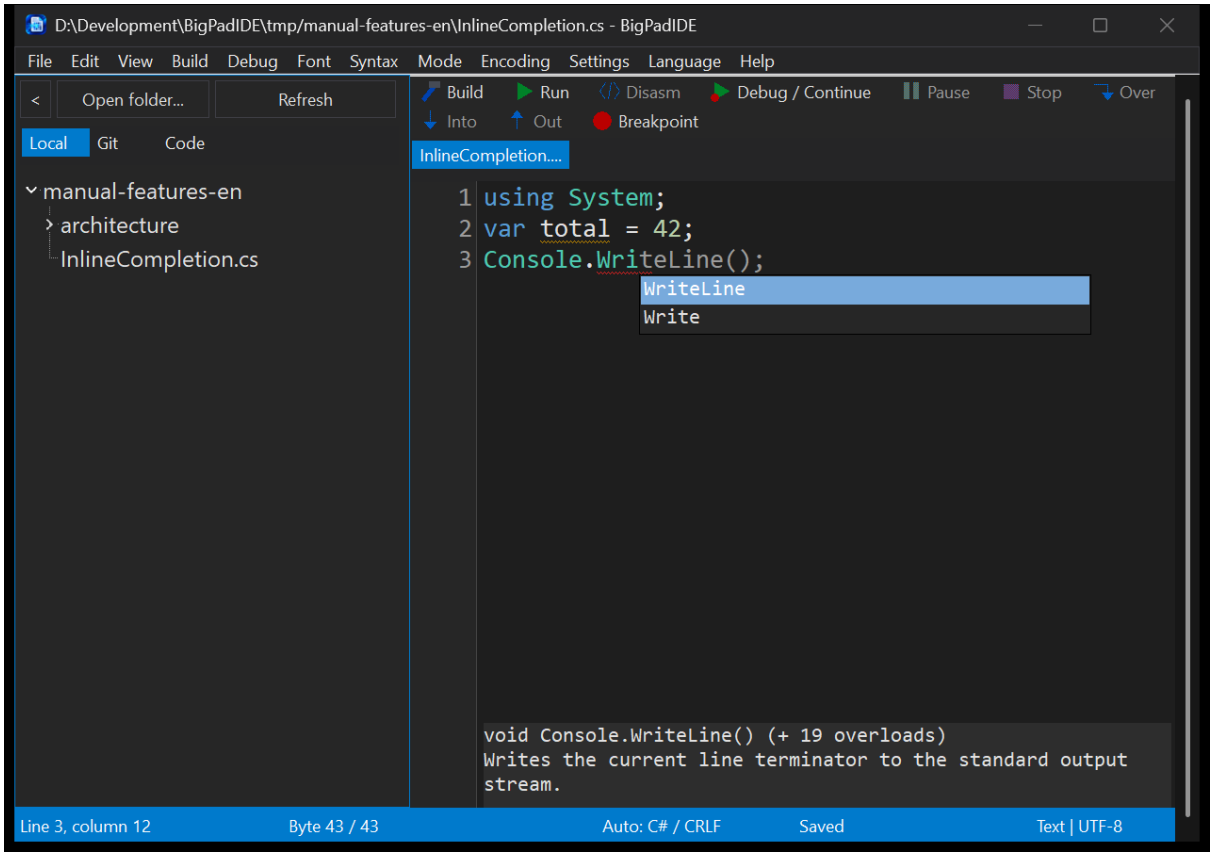


Figure 43. Semantic completion and a Tab suggestion

26.2 Set up local AI

EXE Setup offers separate, initially unchecked options for the **Local AI runtime** (llama.cpp CPU x64, about 25 MB download) and the **Coding and chat model** (Qwen2.5-Coder 1.5B Instruct Q4_K_M, about 1.12 GB). Installing both enables automatic path discovery. Selected downloads use fixed SHA-256 checksums; subsequent inference stays local. The CPU package needs no GPU drivers.

For custom packages, use **Settings → Toolsets and tool paths → Common tools** to select the full paths to `llama-completion.exe` and the GGUF model. Keep the complete llama.cpp package, including its DLLs.

Explicit paths override bundled defaults. For development without Setup, `tools/Setup-LocalCoding.ps1 -ModelPath "E:\Models\model.gguf"` can use an existing b12377 Vulkan ZIP. Restart the IDE afterward.

Templates support Qwen/StarCoder FIM, Qwen Instruct and Gemma 4 text conversations with thinking disabled. Other model families need matching templates. An `mmproj` file is a multimodal projector, not a language model. Models are read from their selected location without copying them.

26.3 Use inline suggestions and chat

At the end of a C# line, press **Ctrl+Alt+Space** to request a local AI suggestion. **Tab** accepts the gray suffix; **Esc** dismisses or cancels it. Loading and running a model can take noticeable time and memory. Requests have time and output limits, and failures are displayed.

Build → Local AI chat … opens the native chat window. The input is multiline; **Ctrl+Enter** sends, **Cancel** stops generation, and **New chat** clears the history. **Add selection** explicitly includes selected source text. Up to four recent exchanges fit within the model context limit. History remains in memory during the IDE session. Replies are not automatically executed or applied to files; review suggested code before accepting it.

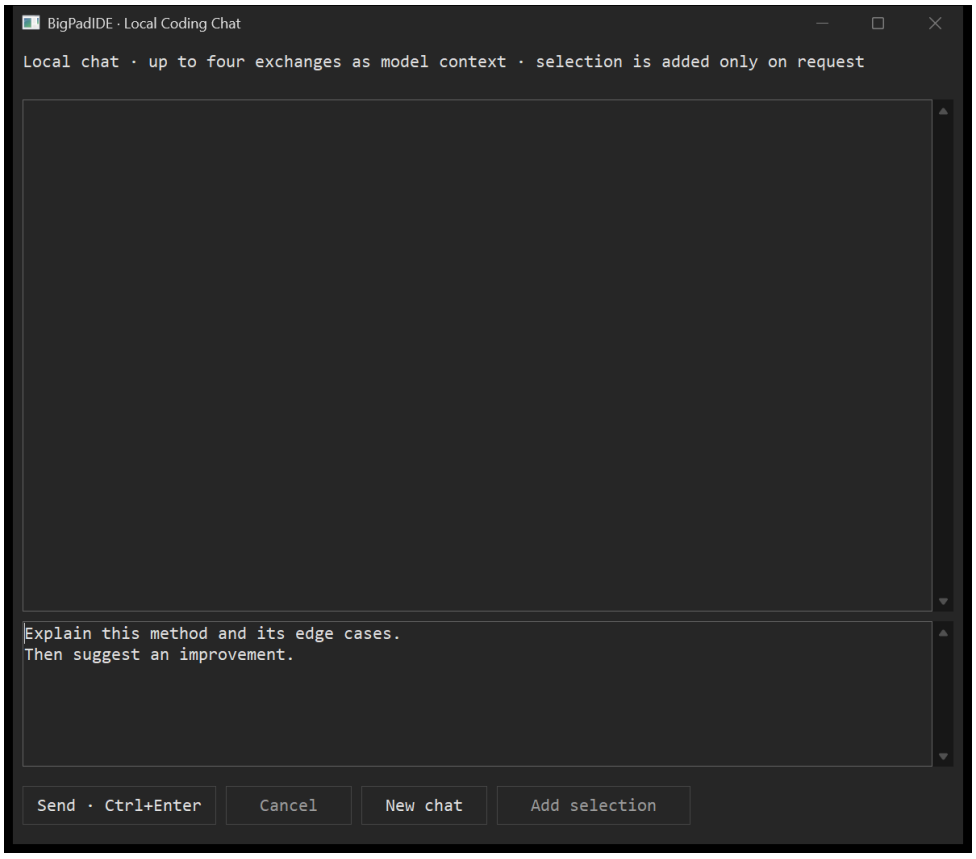


Figure 44. Native chat window

27. Architecture and visual diagrams

27.1 Analyze saved source code

Open a folder in Explorer and choose **Build → 2D architecture analysis ...**. Parsers extracted from CodeCity3D analyze saved files in several languages. C# projects additionally use MSBuild/Roslyn to resolve recognized relationships. Unresolved and non-C# relationships can be heuristic; analysis errors are reported in the result. Unsaved editor changes are excluded. Analyze a subproject when a large folder exceeds the supported limits.

Results open as an editable native diagram. Zoom, fit, element details and **Open source** help navigate to the recognized source file and line. The model captures code relationships; it cannot automatically reconstruct a complete infrastructure or runtime architecture.

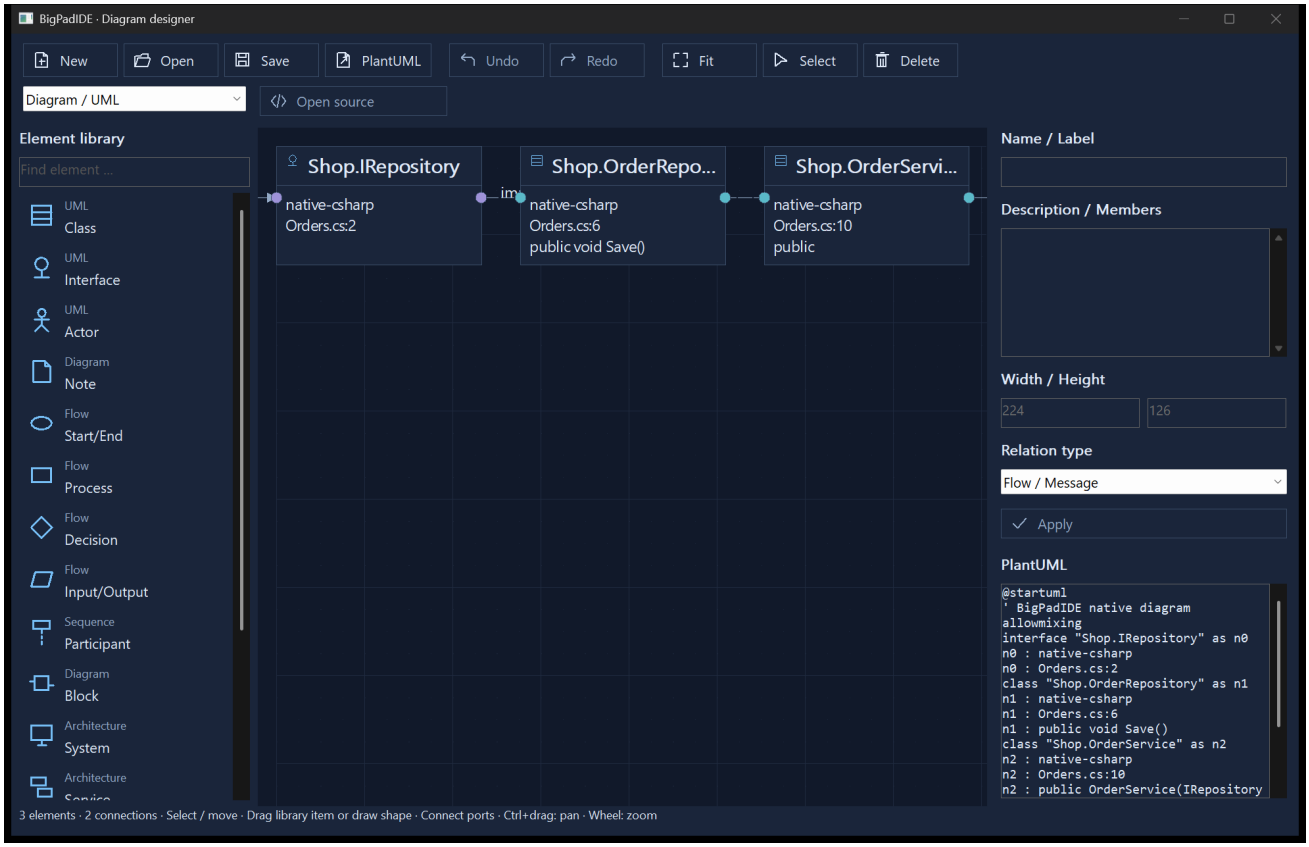


Figure 45. Architecture analyzed from saved source

27.2 Draw UML, flows and system landscapes

Build → Visual diagram designer ... opens the native surface without a browser. The palette is on the left, the canvas in the center, and properties/PlantUML preview on the right. Drag elements from the palette or draw them on the canvas. Move them, resize using corner handles and connect ports. Edit labels, members and relationship categories in the properties area and apply the changes.

Element families cover UML classes/interfaces, flows with start/process/decision nodes, sequence participants, systems, services, databases, external systems and infrastructure. Infrastructure relationships are documented manually. Icons, typography and colors follow the IDE theme.

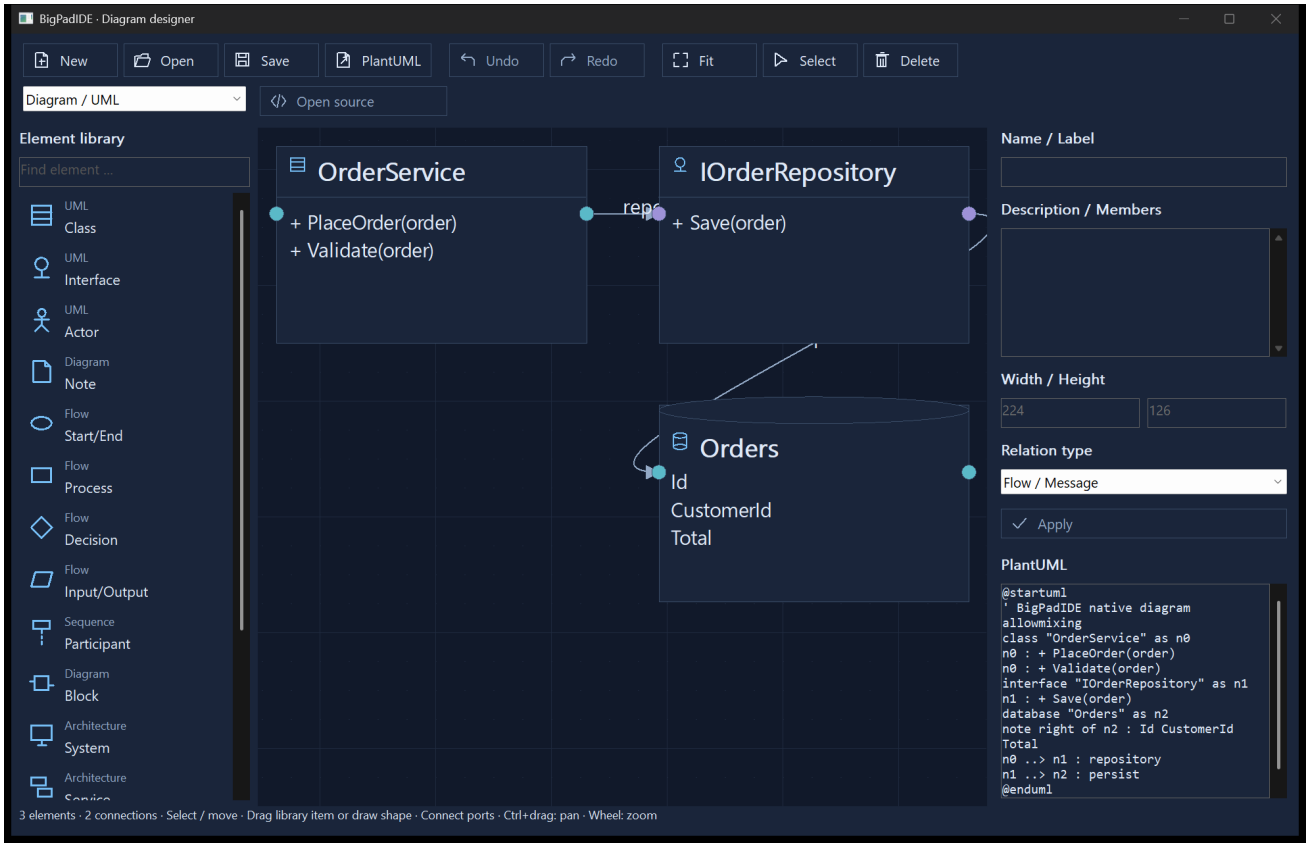


Figure 46. Visual diagram designer

27.3 Save and exchange diagrams

Save `.bpdiagram` files to retain positions, sizes, text, relationships and source links for later editing. Undo/redo supports local changes. PlantUML export creates readable diagram text suitable for Git. It is not a complete semantic UML/XMI exchange model. XMI and hierarchical C4 drill-down are not implemented yet. Models are limited to 1000 nodes, 3000 edges and 4 MiB; use smaller diagrams at these limits.

27.4 Explorer and assembly preview

The Explorer context menu switches between **tree**, **list** and **details**. Details show file sizes and modification dates; folders can be opened directly. **Reveal in Explorer** opens a directory or selects a file in Windows Explorer. The view preference survives restarts.

Detected .NET assemblies offer **Analyze .NET: IL / C# ...**, and supported native PE/ELF files offer machine-code disassembly. Files also offer opening in the hex editor. The read-only IL/C# preview uses theme syntax colors and **Ctrl+mouse wheel** font zoom. Reopening the same DLL reloads the existing panel. For a blank panel, check its status message, DLL accessibility and helper/SDK paths first.

28. Licenses and release status

Help → Licenses and third-party components ... opens the native offline catalog. Select a component to read its version, license, copyright and included third-party notices. **Project website** opens the publisher reference; **License folder** opens the installed files. Optional components are identified. User-selected models retain their publisher license terms.

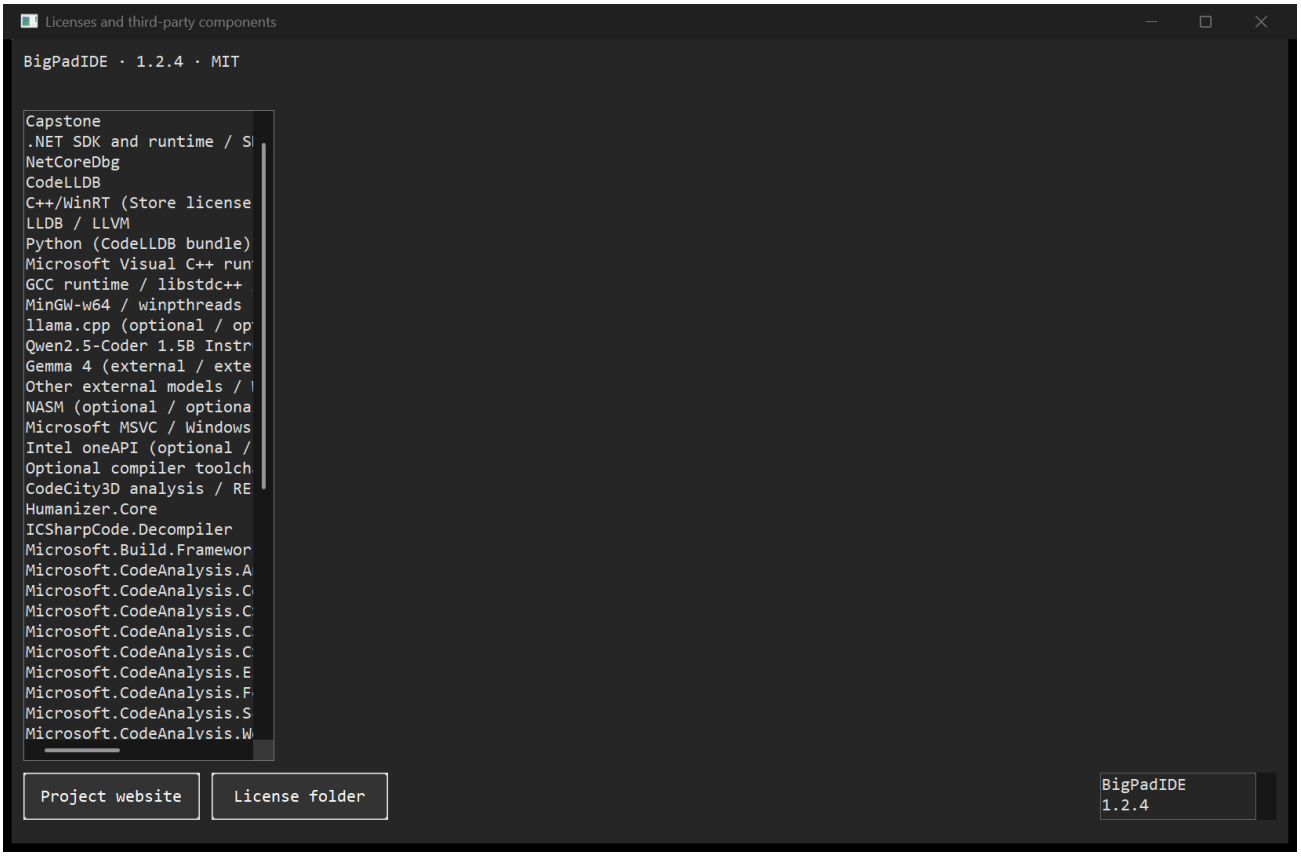


Figure 47. Read third-party notices offline

The build derives this catalog from resolved NuGet packages, SDK notices and native license snapshots. Entries include Capstone, Roslyn/ILSpy, debug adapters, LLVM/Python, GNU/MinGW runtimes, C++/WinRT and optional AI packages. Notices are included in both EXE and Store payloads.

This documented snapshot is an initial milestone for Microsoft Store preparation. A successful local build or generated MSIX does not establish Store certification or publication. Submission, signing, packaged entitlement testing and approval remain separate release steps.

Screenshot reference: more views

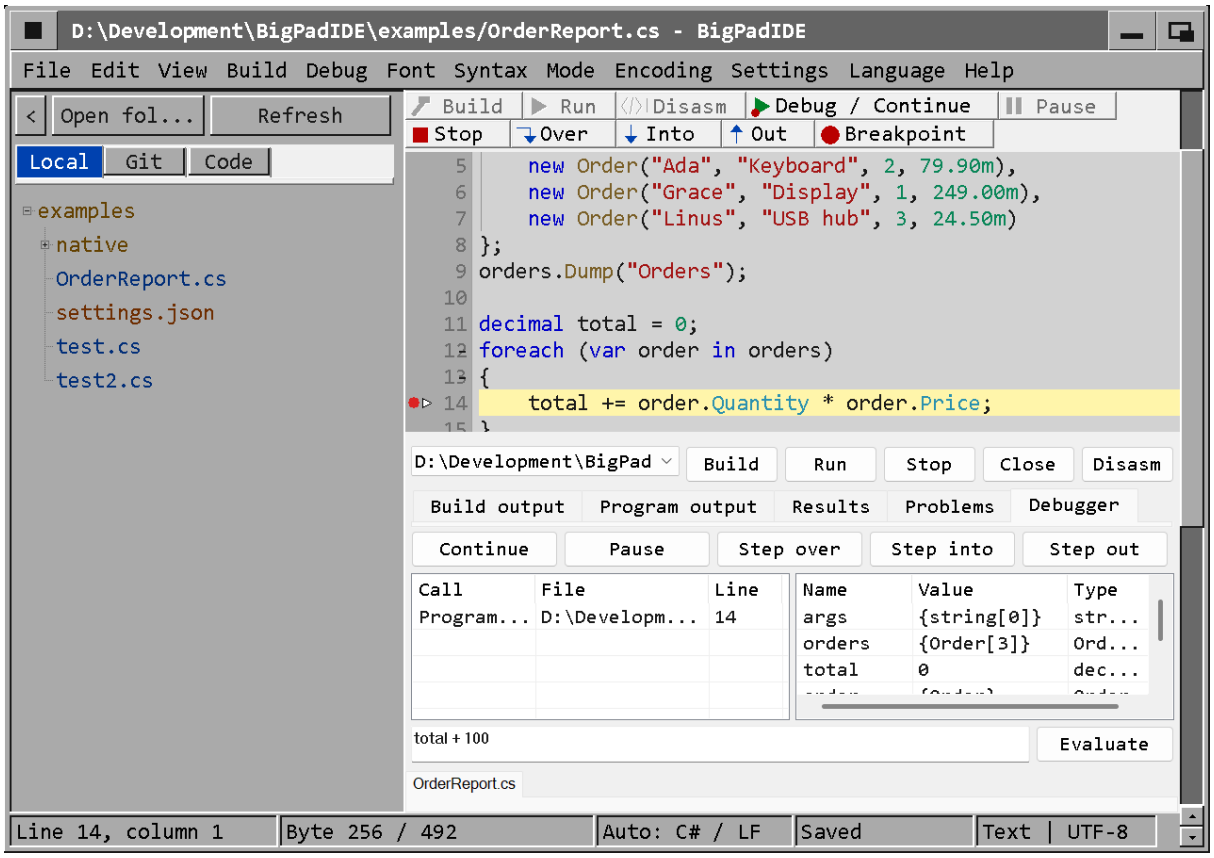


Figure 48. C# breakpoint with stack and variables

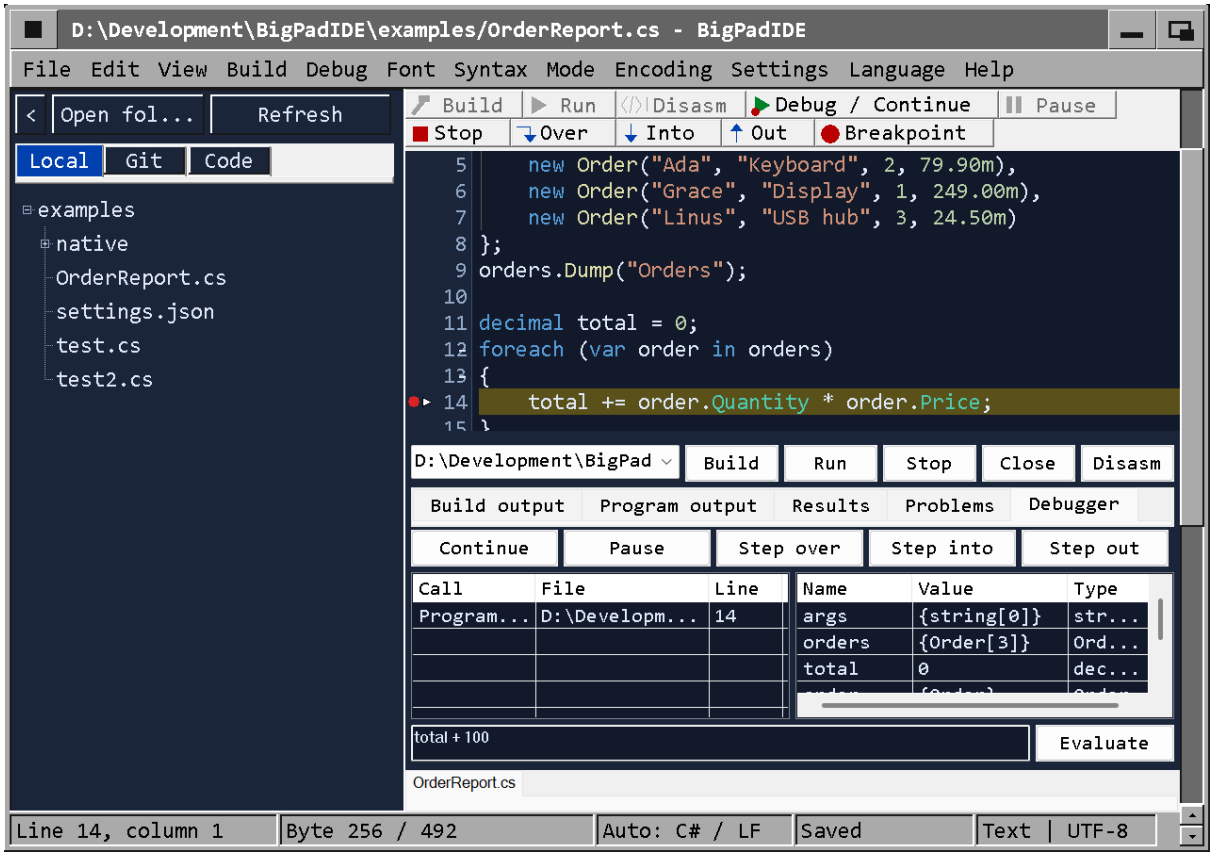


Figure 49. Debugger using the Midnight theme