VB Decompiler

User manual

English

Visual Basic 5.0/6.0 · .NET

DotFix Software

Contents

1 Description 4

2 Quick start 5

3 The main window 6

4 Viewing of code 8

5 AI Code Enhancement 11

6 Open and save 15

6.1 Opening file 15

6.2 Working with database 16

6.3 Saving the list of procedures 17

7 Program functions 18

7.1 Jump to address 18

7.2 String Search 19

7.3 String References 20

7.4 Decompilation from address 22

7.5 HEX Editor 23

7.6 Obfuscation 25

8 Program options 26

8.1 Program options 26

8.2 Forms 27

8.3 Decompilation 28

8.4 Disassembler 30

8.5 Color Scheme 31

8.6 Advanced features 33

8.7 Language selection 37

9 Plugins 38

10 Visual form reconstruction 42

11 Tracing 46

12 Analytic features 48

13 Obfuscation restrictions 50

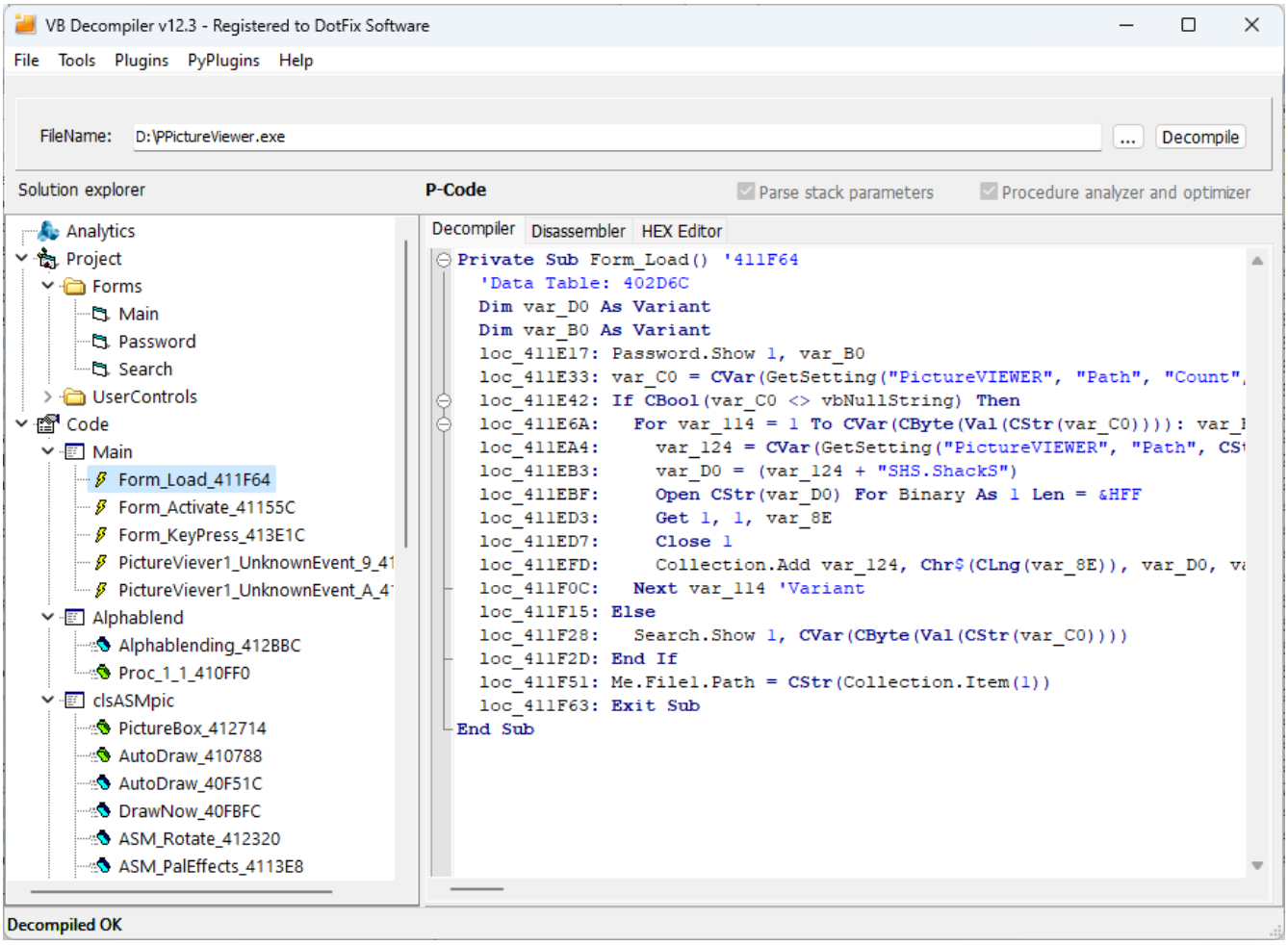
14 Support 52

15 Registration 52

16 About author 53

1 Description

VB Decompiler is a reverse-engineering and source-recovery tool for Visual Basic 5.0/6.0 programs and .NET assemblies. It combines high-level decompilation, low-level disassembly, metadata inspection, a linked HEX Editor, and visual interface reconstruction. Typical uses include legacy software maintenance, incident response, malware analysis, and investigation of undocumented application behavior.



Supported code representations

- **VB5/6 P-Code:** reconstructs Visual Basic statements from the compiler's P-Code instructions and runtime operations.
- **VB5/6 Native Code:** disassembles x86 instructions and uses an internal emulator, runtime signatures, and heuristics to reconstruct Visual Basic-like code.
- **.NET:** disassembles Common Intermediate Language (CIL, also called MS IL) and decompiles supported IL patterns into C#-like code, irrespective of the original .NET source language. Native x86 and x64 code in .NET assemblies can also be disassembled.

The .NET analysis engine does not require the .NET runtime or Reflection for code analysis or visual form reconstruction.

Analysis tools

Use procedure and text search, cross-references and variable renaming, and String References to investigate behavior. The visual viewer reconstructs classic VB forms, .NET Windows Forms, and WPF windows, pages, and user controls stored in compiled BAML resources.

Licenses with Tracing features provide step-by-step emulation of VB5/6 Native Code and P-Code. Licenses with Analytic features provide categorized reports for VB5/6 and .NET applications. AI Helper can process recovered VB5/6 or C# code using Ollama or the OpenAI API integration.

Interpreting recovered code

Compilation removes information such as comments, many local variable names, and some type and procedure metadata. Optimization and obfuscation can further change the code structure. The reconstructed listing supports code analysis and recovery. Verify important conclusions against disassembly, metadata, string references, and the surrounding procedures.

2 Quick start

1. Select **File > Open program** and open the EXE, DLL, or OCX file to analyze. Processing starts automatically. Wait for the project tree to populate and for processing to finish.
2. Open a procedure in **Solution explorer**. Compare the **Decompiler** and **Disassembler** views; use the **HEX Editor** to inspect the corresponding bytes.
3. Press **Ctrl+T** to find a procedure by its name or part of its name. Use **Ctrl+F** to search displayed text, or **Tools > String references** to investigate strings and the procedures that use them.
4. Open a recovered form and select **Designer (read only)** for a visual preview. For classic VB, Windows Forms, and WPF, use recovered events to navigate to the corresponding handler code.
5. Save the analysis through **File > Data Base** when you need to resume it later or share the results with another analyst.

Choosing a decompilation mode

Fast Decompilation processes procedures when you open them. Enable **Analyze prototypes** for an additional preliminary analysis of private procedure signatures. For searching all decompiled text or exporting a complete project, disable Fast Decompilation and process the file again.

Saving recovered source

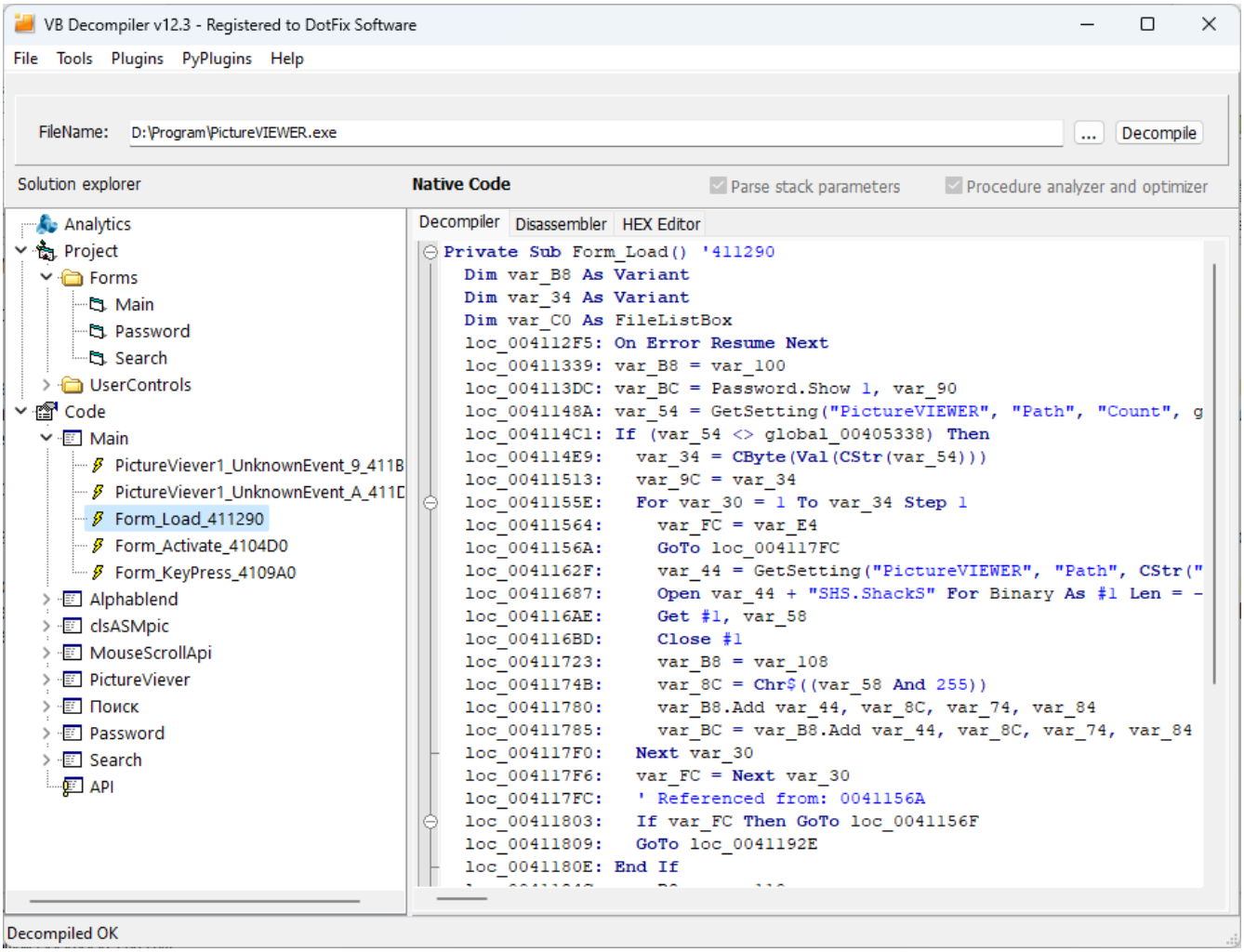
To export VB5/6 decompilation results, select **File > Save decompiled project** and choose an output folder. The export includes recovered forms and modules; FRX files hold associated images and other binary form data.

Before opening the exported code in an IDE, manually correct all decompilation inaccuracies and incompatibilities in the recovered source.

See Visual form reconstruction, Viewing of code, and AI Code Enhancement for the detailed workflows.

3 The main window

Solution explorer organizes the recovered project into forms, modules, classes, and procedures. The adjacent view displays the selected code, form, or metadata. The **Window style as in Visual Studio** option places the tree on the right and the code on the left.



Code and data views

- **Decompiler:** high-level reconstructed code.
- **Disassembler:** the corresponding x86/x64 instructions, VB P-Code, or .NET IL, with addresses.
- **HEX Editor:** the underlying file bytes, with navigation by virtual address or file offset.
- **Designer (read only):** a reconstructed form and its recovered properties, where supported.
- **AI Helper:** optional AI processing of the currently decompiled procedure.

The available tabs depend on the selected object. The code type indicator distinguishes **P-Code**, **Native Code**, and **.NET Assembly**.

Parse stack parameters

For P-Code, this control combines stack operations and calls into expressions with arguments. Turning it off exposes intermediate operations and can help when an unknown prototype consumes unexpected stack values. For Native Code, enabling it uses the emulator to reconstruct higher-level expressions; disabling it exposes assembly-level output. For .NET, stack processing contributes to reconstruction of IL as C#-like expressions.

Procedure analyzer and optimizer

This control simplifies the reconstructed listing by combining expressions and recognized control-flow patterns and removing supported forms of compiler scaffolding. It applies to VB5/6 and .NET code. The .NET optimizer can reconstruct supported `while`, `do...while`, and `foreach` patterns, nested conditions, operator expressions, and interpolated-string handler patterns. These transformations are performed by the decompiler itself and do not require AI Helper.

If an expression or branch is unclear, compare the output with optimization disabled and examine the disassembly. In **Fast Decompilation** mode without caching, these controls can be changed as you inspect a procedure. With full decompilation or cached output, process the file again after changing the settings.

Navigation and keyboard access

The arrow buttons above the tree navigate through previously viewed procedures; **Esc** returns to the previous procedure. Mouse Back and Forward buttons use the same history, including the remembered position in a procedure.

Press **Alt+1** to switch focus between Solution explorer and the active view. Press **Enter** in the tree to open the selected object. In the HEX Editor, **Tab** moves between the hexadecimal and text fields and then to the next control. Interface labels are associated with their input fields for screen-reader navigation.

See Viewing of code for context-sensitive shortcuts, cross-references, and renaming, and Analytic features for the Analytics view.

4 Viewing of code

The code views combine syntax highlighting, collapsible blocks, address-based navigation, and cross-references. Use the high-level listing to understand a procedure and the disassembly and file bytes to check how that result was obtained.

Highlighting and navigating identifiers

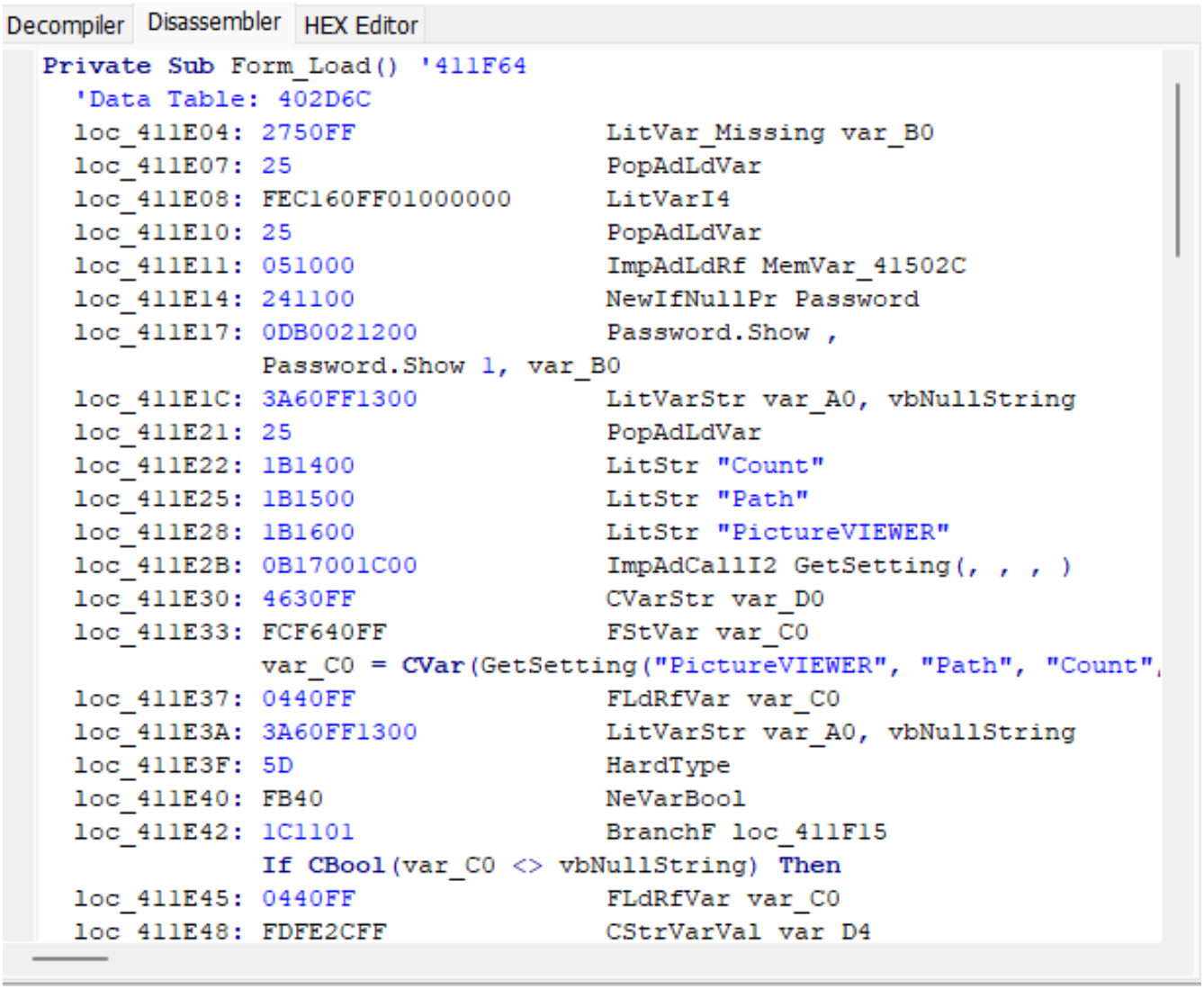
Place the caret on a register, variable, or function name to highlight matching occurrences in the current listing. Local variables such as `var_XX` and `str_XX` are also highlighted when used as the base of a member expression, for example `str_1.Value`. Selecting a `global_XX` identifier highlights its occurrences within the current procedure; this highlighting is separate from a project-wide reference search.

Double-click a recognized procedure call to open the called procedure. Address-like constants can instead lead to the corresponding bytes in the HEX Editor. Use **Esc**, the navigation arrows, or the mouse Back/Forward buttons to move through viewing history.

Combining decompiled and disassembled code

Right-click the decompiled listing and select **Copy to disassembler**, or press **D** in the code view. This merges the reconstructed statements with the low-level listing for the current procedure. To preserve your place, position the caret on the relevant `loc_` address before invoking the command.

Keep **Add Addresses** enabled in the decompilation settings when using this workflow. The Disassembler always displays addresses; Add Addresses controls the Decompiler view.



Renaming variables

For VB5/6 P-Code and Native Code, right-click a variable and select **Rename variable**, or place the caret on the variable and press **N**. Enter a meaningful analysis name. The decompiler checks for name collisions and reserved keywords. To revert a rename, use **Undo renaming** or **Ctrl+Shift+Z**. This command is independent of ordinary text-editing undo.

Renaming makes the recovered listing easier to analyze; it does not recover a name that was removed by compilation. Treat analyst-assigned names as annotations when communicating findings.

Finding callers and global variable references

To find callers, right-click a procedure in Solution explorer and select **Find references**, or press **X** with that procedure selected in the tree. The results list the procedures containing recognized calls to it. Double-click a result to inspect the caller. This workflow supports VB5/6 and .NET; .NET support was added in version 12.4.

To investigate a global variable, place the caret on its identifier in the code view and press **X**. The same key therefore searches procedure references in the tree and global variable references in code. A list of recognized references does not establish that there are no additional indirect, dynamically resolved, or obfuscated uses.

Context-sensitive shortcuts

Focus	Shortcut	Action
Code	N	Rename the variable under the caret (VB5/6).
Code	X	Find references to the global variable under the caret.
Code	D	Copy the current decompiled code to the Disassembler.
Code	F4	Trace to the address on the current line; requires Tracing features.
Code	Ctrl+Shift+Z	Undo renaming of the variable under the caret.
Tree	X	Find references to the selected procedure.
Tree	G	Open Jump to the section.
Tree	Ctrl+C	Copy the selected object's full tree path.
Tree	Space or Insert	Mark or unmark the selected item.

Single-letter shortcuts depend on the focused view and do not replace normal typing in input fields. Use **Alt+1** to change focus. See String Search for Ctrl+T and Ctrl+F, Jump to address for tree paths and addresses, and Tracing for emulator limitations.

5 AI Code Enhancement

AI Helper processes code that VB Decompiler has already reconstructed. It supports decompiled C# from .NET assemblies and Visual Basic 5.0/6.0 code from P-Code or Native Code. You can use a local model through Ollama or connect to a service through the OpenAI API integration.

AI can suggest clearer control flow, variable names, declarations, and expressions. It cannot reliably restore information absent from the binary or repair a fragment that the decompiler has interpreted incorrectly. Compare suggestions with the original decompiled listing, disassembly, and surrounding procedures before using them in recovered source or an investigation report.

Processing a procedure

1. Open the required procedure in Solution explorer and review its **Decompiler** listing.
2. Switch to **AI Helper** and configure the provider, host, and model as described below.
3. Start processing with the action button beside the model selector.
4. Review the returned code in AI Helper and compare it with the decompiler's output.

AI processing runs in a separate thread. Use **Cancel** if the model takes too long to respond. Secondary connection settings can be collapsed to leave more space for code.

Using a local model through Ollama

1. Install Ollama from its official download page and start it.
2. Download the recommended model by running `ollama pull qwen3:8b` in a terminal.
3. Select the local LLM/Ollama provider in AI Helper. Set **Ollama host** to the server address; the default local address is `http://localhost:11434`.
4. Select the installed model in **Model** and start processing.

We recommend `qwen3:8b`. AI Helper's built-in prompts are tuned for this lightweight model. Newer models can take longer to process the code without improving the result.

Model selection is unrestricted: you can choose any model installed in the selected Ollama instance. With an unfamiliar model, the response parser may leave model-specific tags or other auxiliary text in the displayed result.

Hardware for local Ollama inference

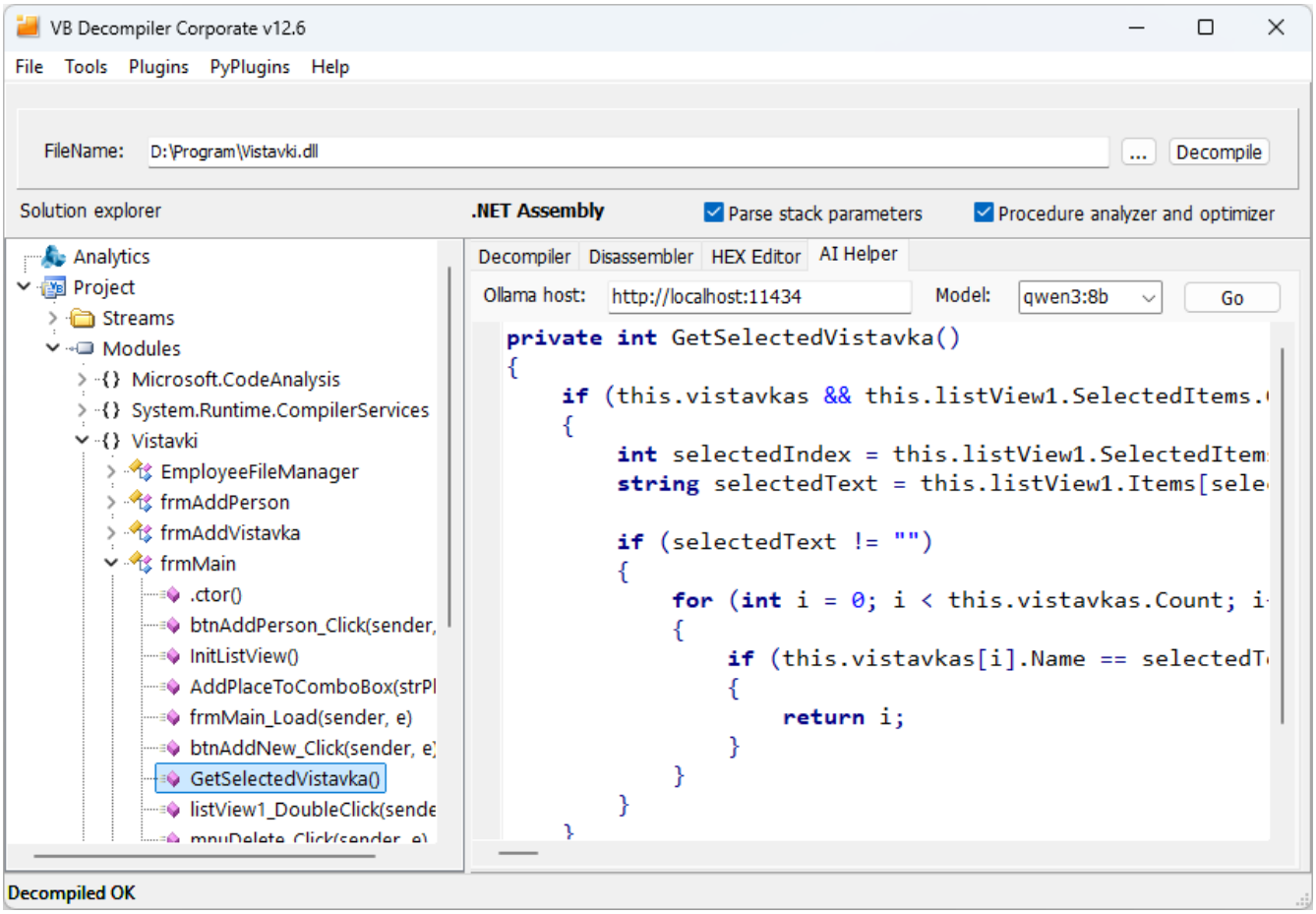
Local AI processing runs on the computer hosting Ollama and requires substantially more resources than decompilation. The following practical configurations assume `qwen3:8b` with `Q4_K_M` weights, the standard `f16` context cache, and one request at a time. Context is measured in tokens: 4K means 4096 tokens and 128K means 131072 tokens. The budget includes the instructions, source code, and generated response.

Component	Practical minimum: 4K context	Recommended: 128K context
Operating system	64-bit Windows 10 (22H2) or Windows 11	64-bit Windows 10 (22H2) or Windows 11
Processor	Modern x64 CPU with AVX2 and at least 4 physical cores	Modern x64 CPU with AVX2 and at least 8 physical cores
System memory (RAM)	16 GB	64 GB
Graphics card	Ollama-compatible GPU with at least 8 GB of dedicated VRAM	Ollama-compatible GPU with at least 32 GB of dedicated VRAM
Storage	SSD with at least 20 GB free for Ollama and this model	SSD with at least 20 GB free for Ollama and this model

Set the context size with the slider in Ollama settings. An 8 GB graphics card suitable for 4K is not sufficient for the 128K configuration above. The model download is approximately 5.2 GB; this is not its total memory requirement. At 128K, the `f16` context cache alone is approximately 18 GiB, before model weights and working buffers. Leave memory available for Windows and other applications.

Ollama can use the CPU when sufficient RAM is available, but code processing can be very slow. Pentium M systems and similarly old computers are unsuitable for local `qwen3:8b` inference. Shared memory used by legacy Intel HD Graphics does not replace the dedicated VRAM specified above. These requirements apply to the Ollama host; a remote server or external API service uses its own hardware.

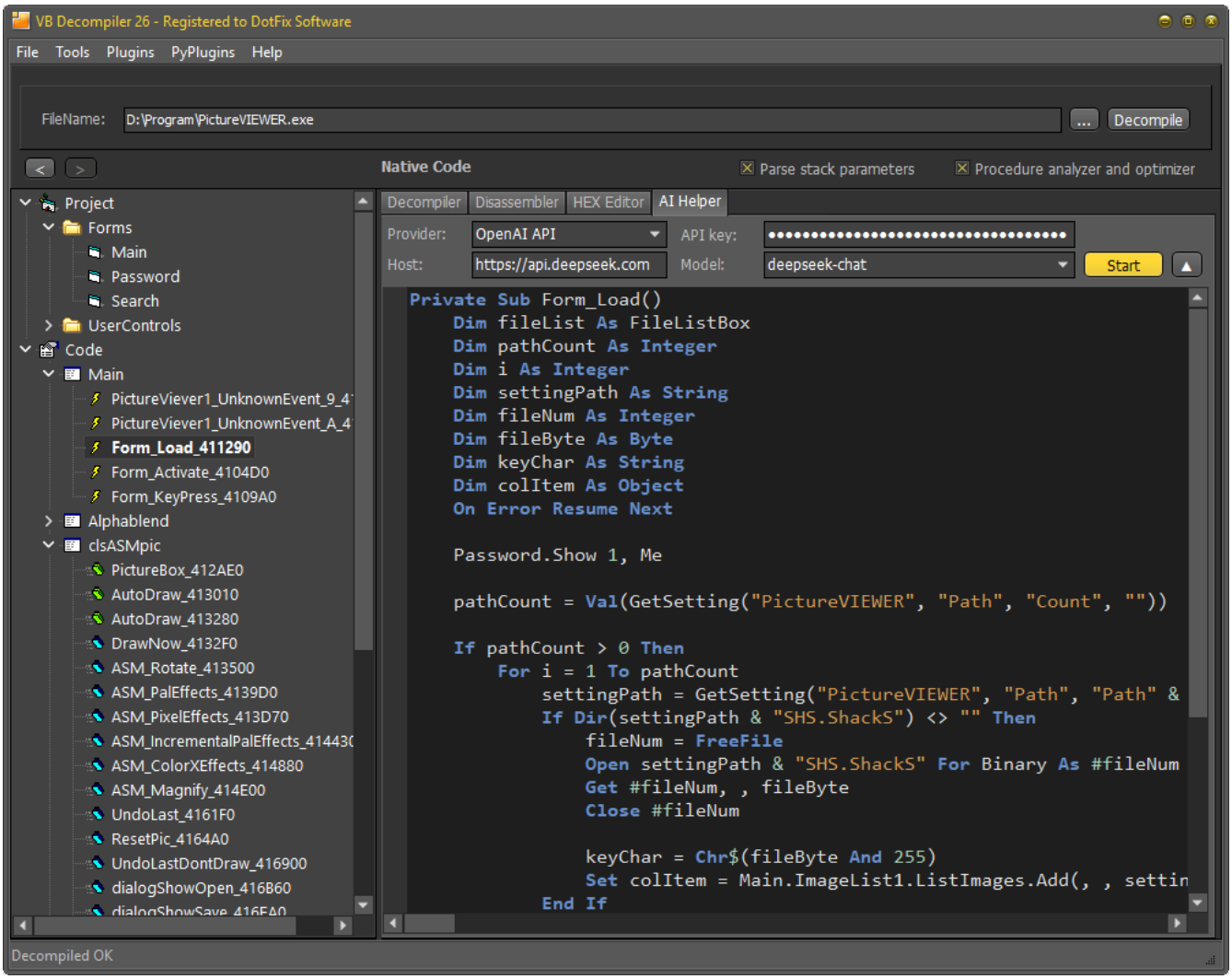
For short procedures, start with 4K. A larger context increases memory use and the time needed to process long input. If processing is too slow, cancel it, reduce the context, and check GPU use with `ollama ps`. Actual memory use depends on the Ollama version, backend, and other running workloads.



Using the OpenAI API integration

Set **Provider** to **OpenAI API**. Configure **Host**, **API key**, and **Model** for the intended service. Enter a valid API key and a model identifier available to that key. Keep API keys out of screenshots and shared analysis material.

AI Helper also supports DeepSeek through the **OpenAI API** provider. Enter the DeepSeek host, API key, and model identifier in the corresponding fields. The screenshot below shows this configuration.



Where the code is processed

AI Helper sends the current decompiled code to the configured service. For a local-only workflow, use a locally running Ollama server and a locally executed model. An Ollama host on another machine, an Ollama cloud model, or an external API service can transfer code outside the analysis workstation.

For confidential source recovery or security investigations, choose the endpoint according to your organization's data-handling requirements. Do not infer offline operation solely from the presence of Ollama or a localhost URL.

Troubleshooting

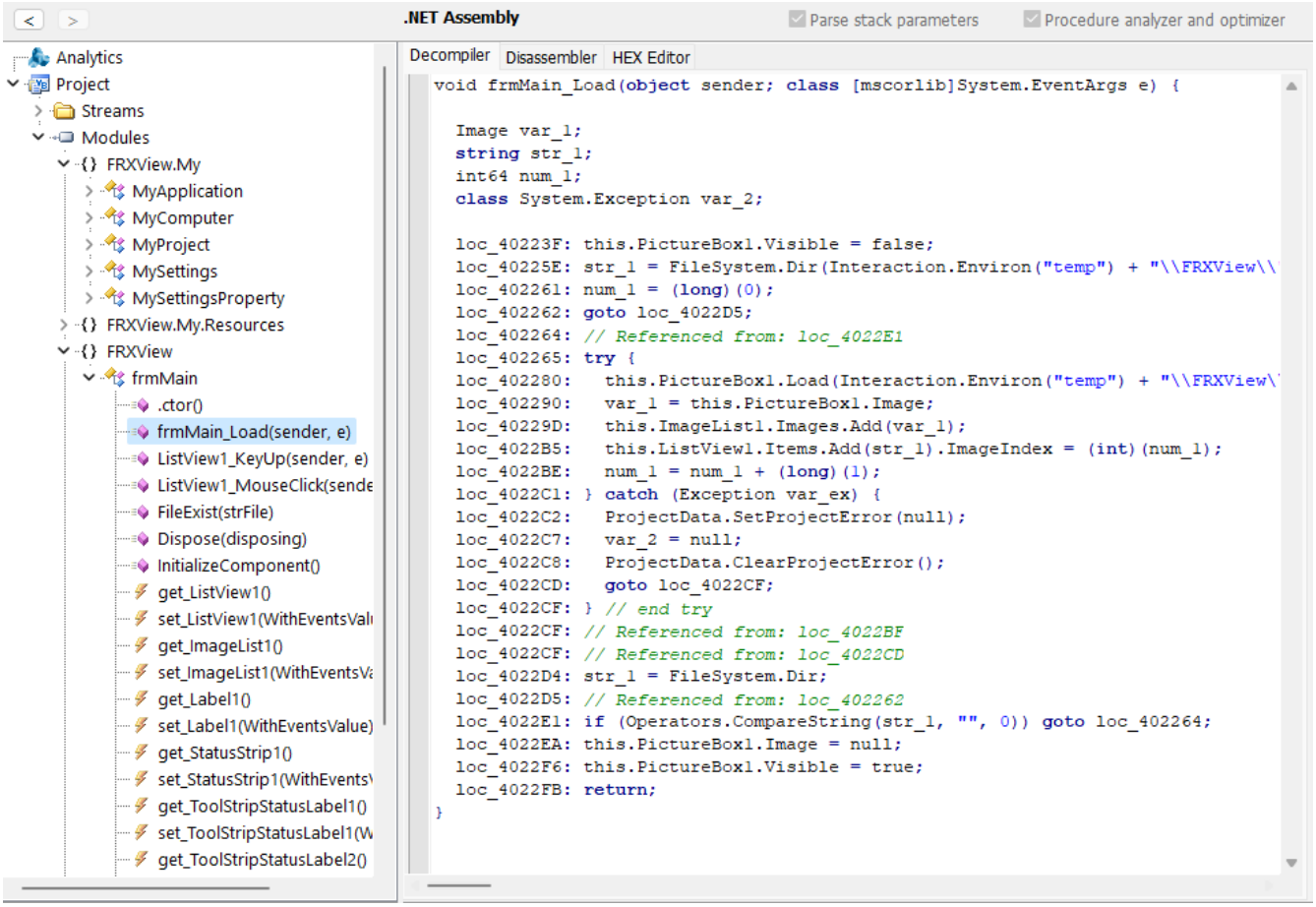
- If a local model is missing, confirm that it is installed in the Ollama instance specified by the host field.
- If there is no response, check that the selected service is running and reachable, and that the endpoint, credential, and model identifier correspond to the same provider.
- If processing is too slow, cancel it. When using another local model, try the recommended `qwen3:8b` model.
- If an unfamiliar model leaves tags or other auxiliary text in its response, try `qwen3:8b` or remove that text manually while reviewing the result.
- If the result changes a condition, call, constant, or type unexpectedly, verify it against the low-level listing. Readability alone does not establish equivalent behavior.

Ollama and the external model providers are third-party products. Their availability, model behavior, and service policies are independent of VB Decompiler.

6 Open and save

6.1 Opening file

Before starting the decompilation process, you need to select a file for analysis. For this purpose, there is an **Open program** section in the **File** menu. **VB Decompiler** supports opening EXE/OCX/DLL files created in **Visual Basic** versions 5.0 or 6.0 as well as applications compiled in **Visual Studio .NET**. In the latter case, the source language does not matter since Visual Studio .NET compiles applications regardless of the original language into a single internal format - **MS IL**.



After opening the file, the decompiler checks the program for belonging to the specified development environments. If the opened program was developed in an unsupported environment (e.g., **Delphi** or **Visual C++**), a message appears stating that such an environment is not supported and it is impossible to decompile the program. In case this message appears, you can contact us, and we may be able to help with decompiling your program. We have internal developments in the field of decompilation of **Delphi** and **FreePascal** applications and can offer decompilation services for such files as a service.

If the opened program is packed with an **EXE packer**, obfuscator or protected - the decompiler will offer to try unpacking this file. This feature is only available in 32-bit versions of Windows and provided without any warranties. If unpacking is possible - the file will be launched, unpacked in memory, saved on disk with a different name, and opened for decompilation. **Note!** Never attempt to unpack using the built-in unpacker malicious programs or any other code that could harm your system because when unpacking, the program being opened will be executed, and the consequences of its execution on your PC can be extremely sad.

Important note: under no circumstances should you try to decompile or unpack software without having the copyrights for it, and if the copyright holder prohibits decompression and analysis of this file by license. This may violate the laws of your country.

*Visual Studio .NET, Visual C++ and Visual Basic are registered trademarks of Microsoft Corporation. Delphi is a registered trademark of Embarcadero.

Additional opening workflows

You can drag a file from Explorer onto VB Decompiler. Files with an unfamiliar extension or no extension can also be submitted this way; recognition still depends on the file's actual contents. Supported Windows **.lnk** shortcuts can be resolved to their referenced files. Saved VDD databases can also be opened by drag and drop or from the command line.

6.2 Working with database

This feature is available through the **Data Base** section in the **File** menu and serves to enable saving decompilation results into a separate compressed file. This can be useful for faster loading of decompilation results in the future without needing to re-decompile code.

In case you need to return to analyzing the program or share these results with a colleague, there's no need to perform decompilation again. In the case of full decompilation (with the **Fast Decompilation** option disabled), this process can take longer and consume additional time. It is much more convenient to pre-save the decompiled code in the **Database**.

VB Decompiler uses an internal compressed database format, which allows for significant optimization of size and density of information contained within such a database. Additionally, there is compatibility with databases saved by most previous versions of VB Decompiler, allowing you to store information without risking the possibility of not being able to open such a database in the new version of the product.

Opening and transferring VDD databases

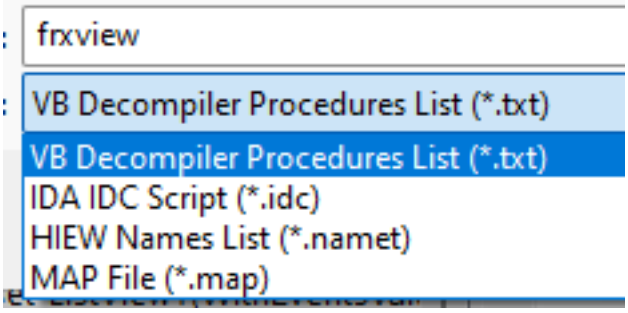
Saved analysis databases use the **.vdd** extension. You can open a VDD database through the file-opening workflow, drag it onto the decompiler window, or pass its path on the command line.

When locating the original analyzed binary, VB Decompiler checks both its saved path and the directory containing the database. For collaboration, keep the corresponding binary beside the VDD file when its original path is unavailable. Preserve the exact analyzed build so that addresses and file bytes remain meaningful.

6.3 Saving the list of procedures

This operation is performed by calling the **Save procedures list** option in the **File** menu and is a powerful tool for analyzing programs as it allows external utilities to assist the **VB Decompiler**.

Currently, addresses, function names, and procedures used in the processed program can be saved in several formats. You can choose any of these options by specifying the file type in the standard save dialog.



Saving in TXT format allows you to save the list of procedures in a simple text file. This can be useful for comparing procedures between different versions of a program, changing addressing and so on.

Saving in IDC script format allows you to load function and procedure addresses into the **IDA Pro** disassembler. In cases where the analyzed program is compiled into **Native Code**, this feature can be very useful.

Saving in NAMET format allows you to simplify analysis using **HIEW** utility. Copy the obtained file into the folder with the analyzed program. Then open the program in HIEW and press the F12 button. A window with an empty list of procedures will appear. Press the F12 button again if the address of the NAMET file matches the created one - just press Enter. The addresses of procedures will be loaded into HIEW, allowing you to navigate through methods using the list by pressing the F12 key.

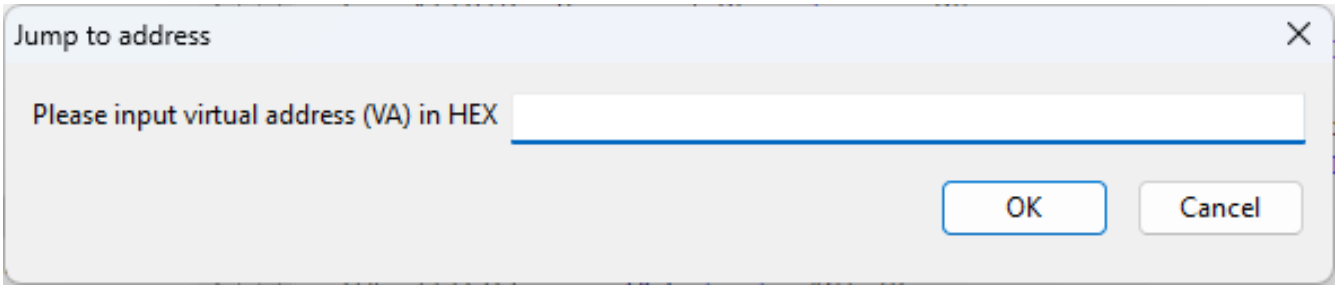
Saving in MAP format generates a standard map file that is usually created by compilers. This file can be used with many debuggers and disassemblers to mark the start of program functions. Since programs written in Visual Basic are often disassembled incorrectly by third-party software, the MAP file generated in **VB Decompiler** will greatly simplify the process. This format is supported through plugins in various products such as **OllyDbg**, **IDA Pro** and others.

7 Program functions

7.1 Jump to address

Jumping to a virtual address

Select **Tools > Jump to virtual address** or press **Ctrl+G**. Enter the virtual address of the code to inspect. You can also select an address in the listing and invoke Ctrl+G. The decompiler opens the code associated with that address.



A virtual address (VA) identifies a location in the executable image's address space and includes its image base. A file offset identifies a byte position in the file on disk. Do not substitute one for the other: the HEX Editor supports both forms of navigation.

Returning through navigation history

Use the arrow buttons above the tree to move backward and forward through previously viewed procedures. **Esc** returns to the previous procedure. Mouse Back and Forward buttons also use this history, with the position within the procedure retained.

Sharing a location by its tree path

In Solution explorer, right-click the required object and select **Copy the name of section**, or press **Ctrl+C** while the tree has focus. This copies the full path to the selected node.

To revisit that location, select **Jump to the section** in the tree's context menu, or press **G** in the tree, and paste the copied path into the dialog. This is useful when handing an investigation to another analyst or returning to a large project. Use the path with the corresponding analyzed file; a path from another build may no longer identify the same code.

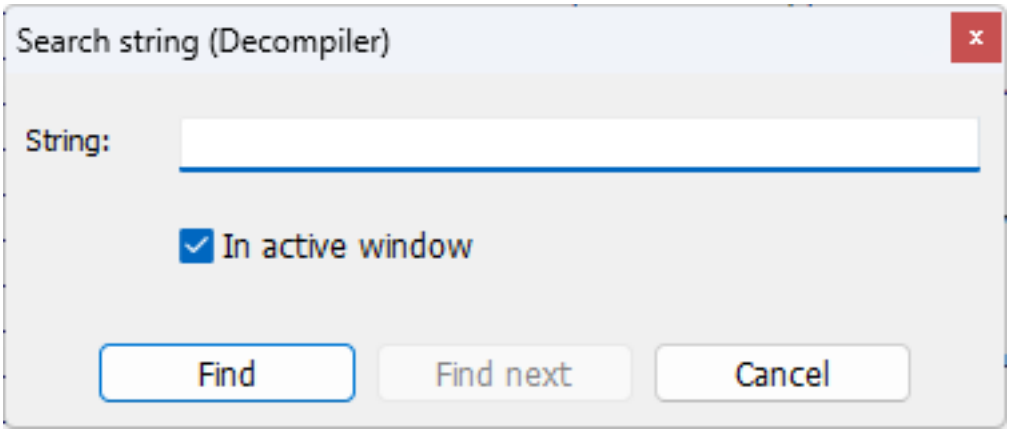
7.2 String Search

Finding a procedure by name

Select **Tools > Find procedure** or press **Ctrl+T**. Enter a full procedure name or a distinctive part of it. The search covers the project tree, so you do not need to know which module or class contains the procedure. If several names match, the result list contains all matching entries; double-click the required entry to open it.

Searching displayed text or the whole project

Select **Tools > Find string** or press **Ctrl+F** to open the text-search dialog. With **In active window** selected, the search applies to the currently displayed code or form data. Clear this checkbox to search across the decompiled project.



Enter the text in **String** and select **Find**. Use **Find next** to continue searching for the same text. **Cancel** closes the dialog.

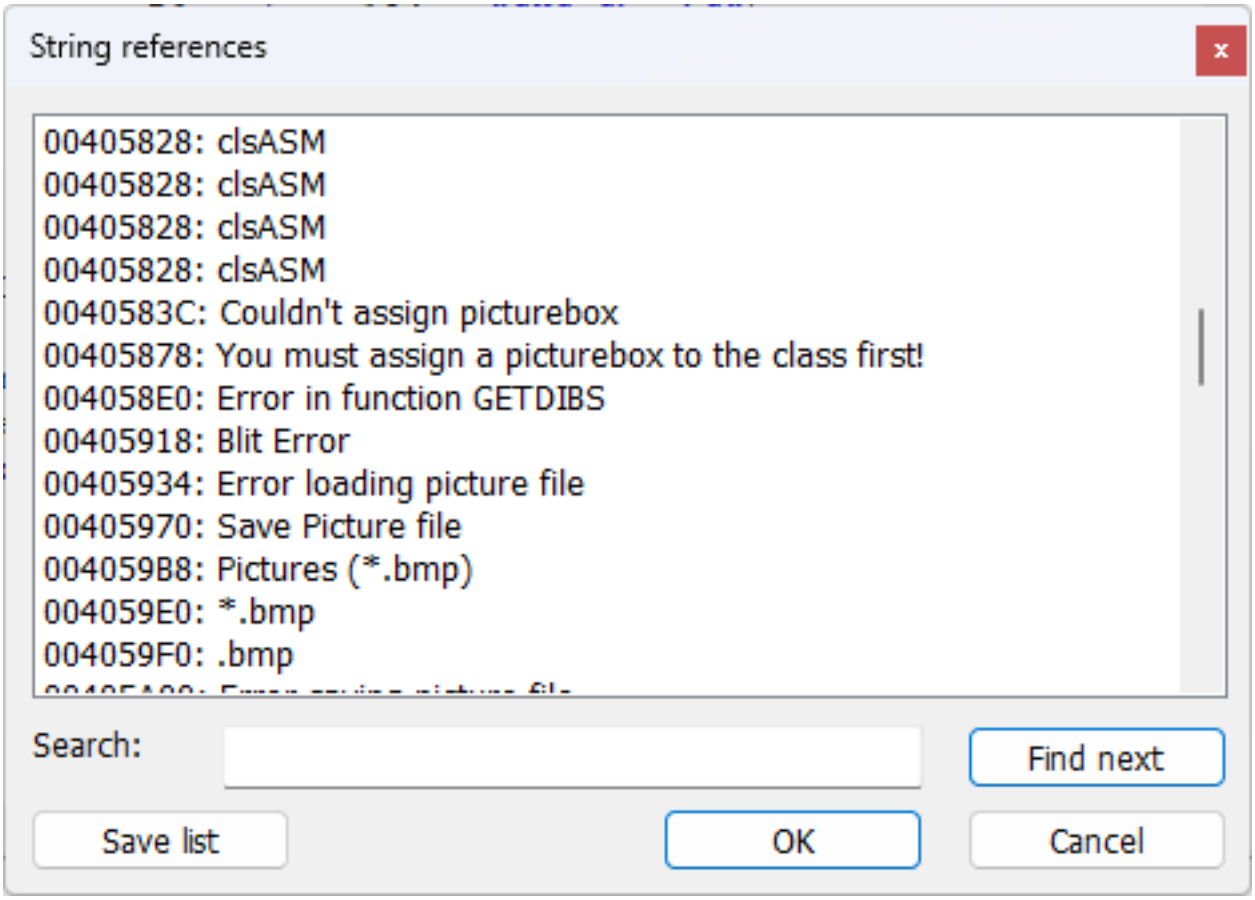
Searching all decompiled text requires full decompilation. Disable **Fast Decompilation** in Advanced features and process the file again first. Finding procedures by name is a separate operation on the project structure.

Searching string references

If you need literal strings and the code that uses them, open String References. Its **Search** field and **Find next** button search the string list. String References can be collected by preliminary disassembly in Fast Decompilation mode; they do not require a search of every fully decompiled procedure.

7.3 String References

In any program's code, you often encounter string data (registry keys, file names, error messages, etc.). It can be quite useful to see the list of strings used by procedures and functions in decompiled applications. That's exactly what our **String references** function is designed for, accessible through the **Tools** menu. To use it, simply navigate to **Tools** -> **String references**, and if your program contains any string data, they will be displayed as shown below:

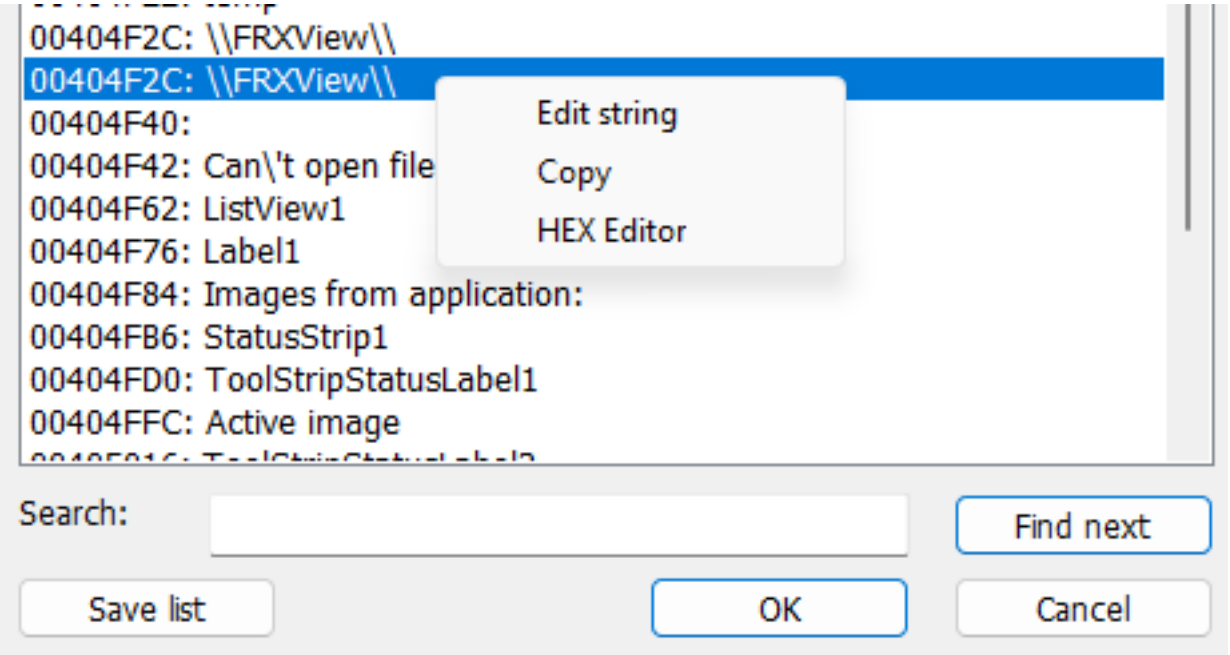


Find the desired line in the list and double-click on it. The code segment containing that line will be located, and the corresponding procedure will be opened in the code window. Double-clicking again on the line continues the search for this line within the program's code. Clicking 'OK' will find the highlighted line in the code and close the row selection dialog. Cancel simply closes the dialogue box.

The function to display **String references** is only available in the licensed version of the software and not accessible for **Lite** version.

Important note: strings are revealed not only for the active module as in versions 1.x program, but for the entire program. This change was made to facilitate analysis of programs with a large number of forms and modules.

VB Decompiler also supports editing strings. To call the edit string dialog, right-click on the string you want to change and select **Edit string** from the context menu that appears. A dialogue box for entering a new string will appear. After changing the string, click **Ok** and the string will be directly modified in the edited file. We recommend creating a backup of the file being examined beforehand, bacause no backup is made during this action and canceling this operation is not possible.



Important note: when editing strings, remember that the length of the new string should not exceed the length of the old one. This is related to the format of binary application files where no space is reserved for inserting a longer string.

You can also view and partially edit the string in the HEX editor. To do this, right-click on the string and select HEX Editor from the menu that appears. In the HEX editor, there are functions for undoing and redoing changes, which simplifies experimentation when modifying string data of applications.

The window with a list of strings also has a text field called **Search** and a **Find next** button. This functionality will be useful for finding strings within the list, especially when dealing with large projects where manual visual search through the list is extremely difficult.

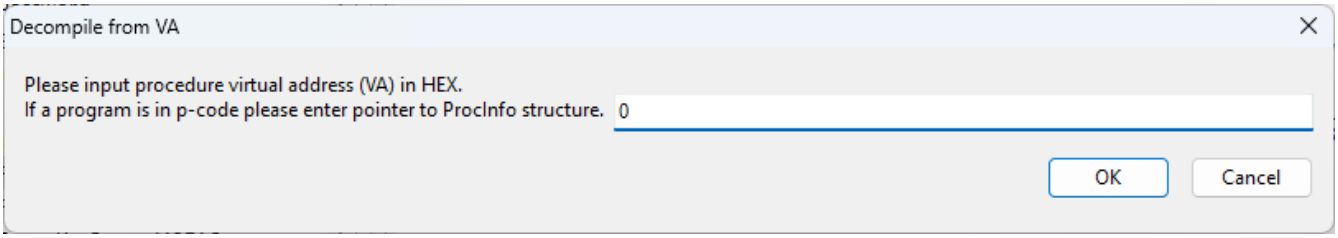
Fast Decompilation and large string lists

String references can be collected through preliminary disassembly, including in **Fast Decompilation** mode. This differs from searching all fully decompiled text. Version 26.2 expands the dialog beyond 32,000 references and improves memory handling for large lists.

For .NET assemblies, user strings from the `#US` heap can be inspected with their file locations. The list represents strings available to static analysis; encrypted or dynamically generated strings may require separate investigation.

7.4 Decompilation from address

In the **Tools** menu, there is a very useful function - **Decompile from VA** (decompile any code from the provided virtual address). The main feature of this function is that it does not require the file being decompiled to have a correct structure and be processed using standard way. Of course, in this case, the code will be decompiled with a number of limitations. For example, if imported functions or OLE objects were not pre-processed, the code will be uninformative at those points.



To decompile a specific piece of code within a file, you simply need to specify the virtual address of that code (if we're talking about **Native Code** or **.NET assembly**), or provide a **ProcInfo** structure reference (if we're talking about classic **Visual Basic P-Code**). The uniqueness of working with such **P-Code** is that the **ProcInfo** structure contains an array of data used by the code. Without this array, it is impossible to determine which functions the code works with, what strings it refers to, and so on. In the case of working with **.NET assemblies**, after entering the address, the decompiler will offer you a choice between decompiling the code as **MS IL** pseudo-code or disassembling this code. This option is due to the fact that .NET assemblies can contain both MS IL pseudo-code (**managed code**) and machine code (so-called **unmanaged code**).

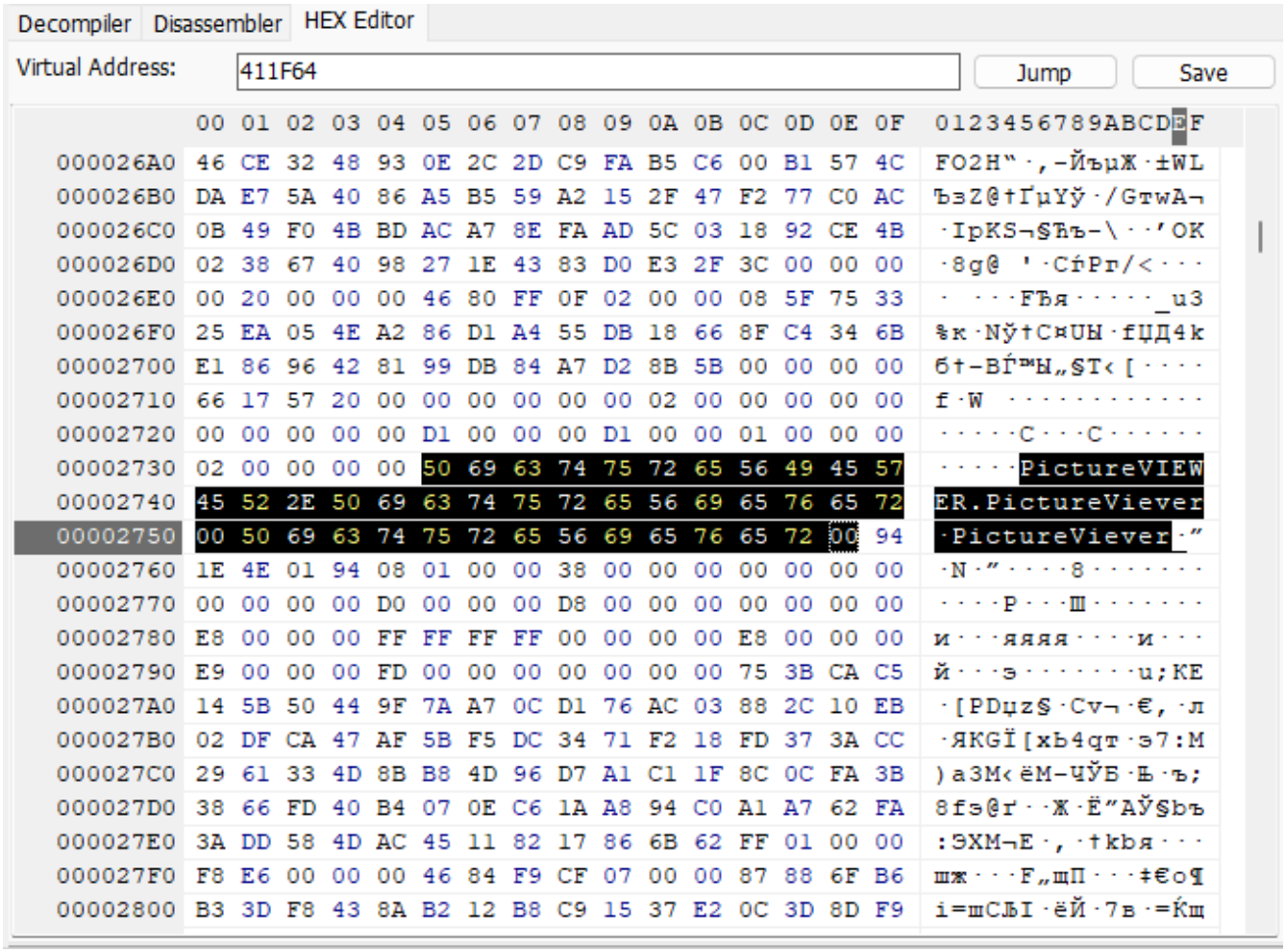
If the file is already open in the decompiler, you just need to call the **Decompile from VA** function and specify the address for processing. If the file has not yet been opened, this will be suggested automatically, and the file will be opened in simplified mode.

The **Decompile from VA** function is particularly useful when the desired code does not decompile automatically or if the file is partially damaged, preventing it from being decompressed using standard tools. Of course, you should understand that the result of the decompilation in this case will be worse than with a standard decompilation.

7.5 HEX Editor

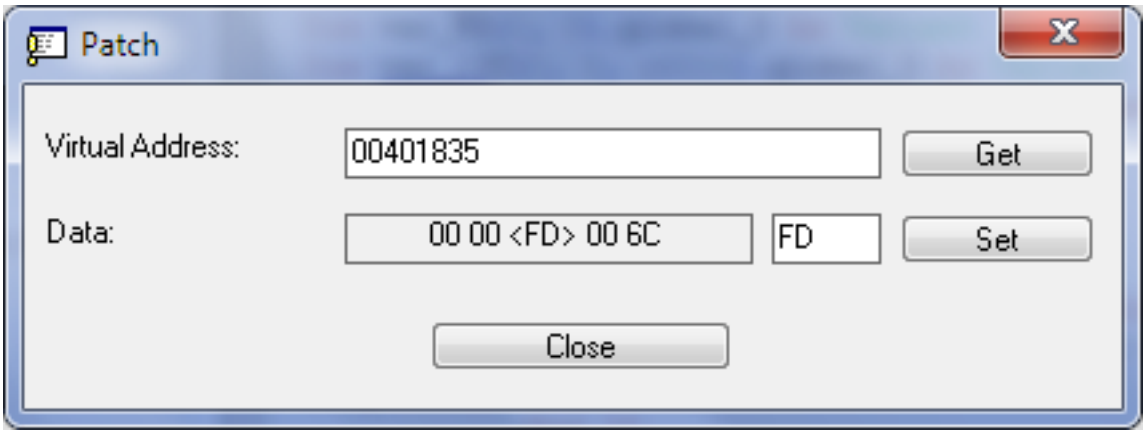
Starting from version 9.4, **VB Decompiler** comes with a modern and convenient tool for editing binary data within files - a full-fledged **HEX Editor**. It is accessible from the main program window alongside decompiled code in the **HEX Editor** tab for each procedure or form.

The **HEX Editor** allows you to edit the file byte by byte as well as in blocks. To navigate to the desired address within the examined file, you can use either virtual addresses (including **Image Base**) or offsets (a regular address indicating the offset of the needed data relative to the beginning of the file).



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
000026A0	46	CE	32	48	93	0E	2C	2D	C9	FA	B5	C6	00	B1	57	4C	FO2H"·,-ЙъµЖ·±WL
000026B0	DA	E7	5A	40	86	A5	B5	59	A2	15	2F	47	F2	77	C0	AC	ЪзZ@†Ґµ Yў·/GтwA¬
000026C0	0B	49	F0	4B	BD	AC	A7	8E	FA	AD	5C	03	18	92	CE	4B	·IpKS¬§Ћъ-\··'OK
000026D0	02	38	67	40	98	27	1E	43	83	D0	E3	2F	3C	00	00	00	·8g@ '·Cѓ Рг/<···
000026E0	00	20	00	00	00	46	80	FF	0F	02	00	00	08	5F	75	33	· ···FЂя·····_u3
000026F0	25	EA	05	4E	A2	86	D1	A4	55	DB	18	66	8F	C4	34	6B	%к·NўtCЊUЊ·fЏД4k
00002700	E1	86	96	42	81	99	DB	84	A7	D2	8B	5B	00	00	00	00	б†–BЃ™Њ„§Т‹[····
00002710	66	17	57	20	00	00	00	00	00	00	02	00	00	00	00	00	f·W ·····------
00002720	00	00	00	00	00	D1	00	00	00	D1	00	00	01	00	00	00	·····С···С·····
00002730	02	00	00	00	00	50	69	63	74	75	72	65	56	49	45	57	·····PictureVIEW
00002740	45	52	2E	50	69	63	74	75	72	65	56	69	65	76	65	72	ER.PictureViever
00002750	00	50	69	63	74	75	72	65	56	69	65	76	65	72	00	94	·PictureViever·"
00002760	1E	4E	01	94	08	01	00	00	38	00	00	00	00	00	00	00	·N·"····8····---
00002770	00	00	00	00	D0	00	00	00	D8	00	00	00	00	00	00	00	····Р···Ш····---
00002780	E8	00	00	00	FF	FF	FF	FF	00	00	00	00	E8	00	00	00	и···яяяя····и···
00002790	E9	00	00	00	FD	00	00	00	00	00	00	00	75	3B	CA	C5	й···э·------u;КЕ
000027A0	14	5B	50	44	9F	7A	A7	0C	D1	76	AC	03	88	2C	10	EB	·[PDџz§·Cv¬·€,·л
000027B0	02	DF	CA	47	AF	5B	F5	DC	34	71	F2	18	FD	37	3A	CC	·ЯКGЇ[xb4qт·э7:M
000027C0	29	61	33	4D	8B	B8	4D	96	D7	A1	C1	1F	8C	0C	FA	3B	)a3M‹ёM–ЧЎБ·Ь·ъ;
000027D0	38	66	FD	40	B4	07	0E	C6	1A	A8	94	C0	A1	A7	62	FA	8fэ@ґ··Ж·Ё"АЎ§bъ
000027E0	3A	DD	58	4D	AC	45	11	82	17	86	6B	62	FF	01	00	00	:ЭXM¬E·,·†kbя···
000027F0	F8	E6	00	00	00	46	84	F9	CF	07	00	00	87	88	6F	B6	шж···F„щП···‡€o¶
00002800	B3	3D	F8	43	8A	B2	12	B8	C9	15	37	E2	0C	3D	8D	F9	і=шСЉІ·ёЙ·7в·=Ќщ

In earlier versions of the decompiler (before version 9.4), the function for editing files was applied using a more primitive tool: Data Patcher. It worked as follows. In the **Tools** menu -> **Patch data**, there was an instrument called patcher that could be used in cases where after analyzing the code of the program it was necessary to change some part. To avoid applying external tools and changing the needed bytes, **VB Decompiler** had built-in functionality. The window of the patcher looked like this:



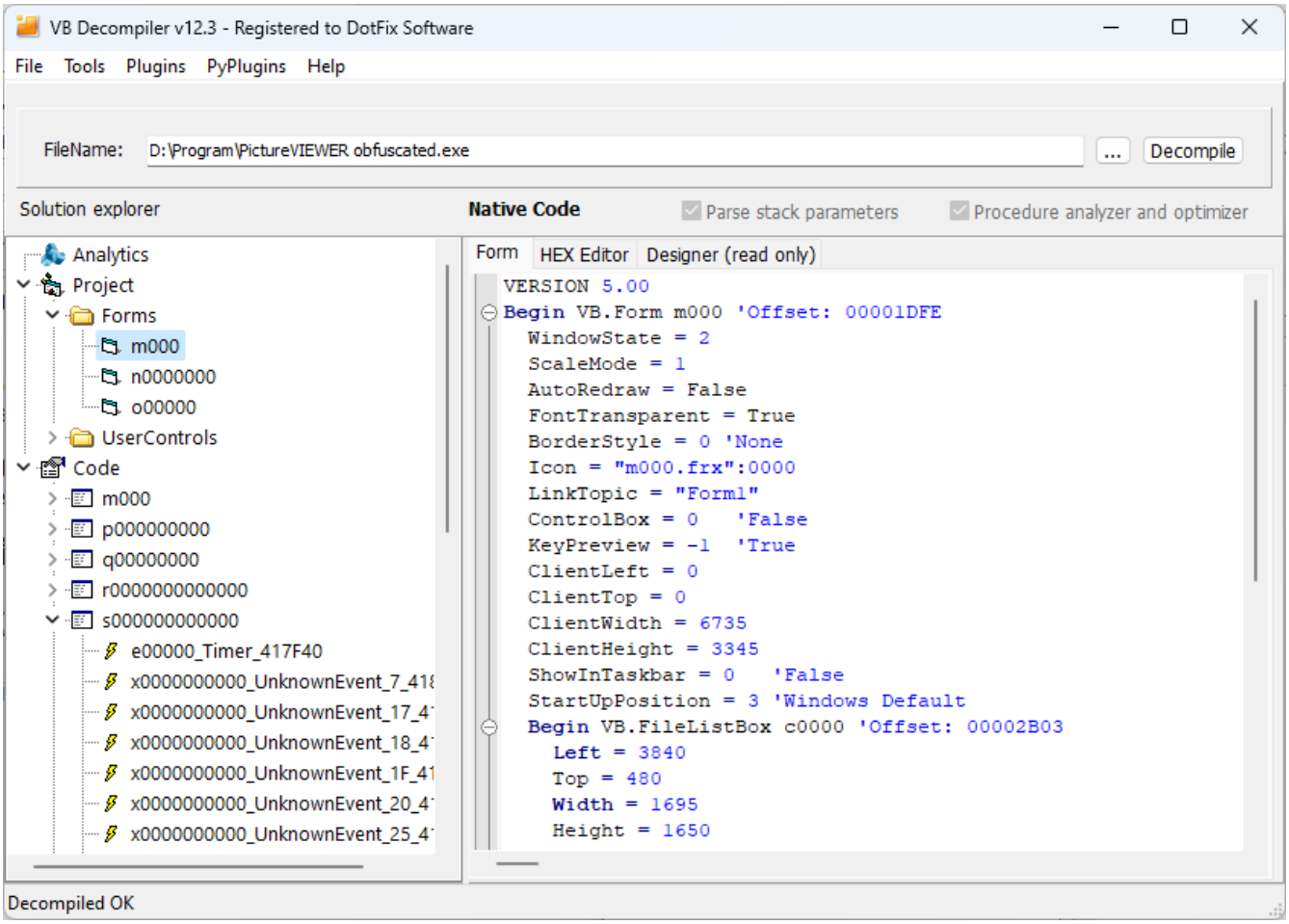
Working with the patcher was extremely simple. In the **Virtual Address** field, you entered the address for editing a byte, then either pressed **Enter** or clicked **Get**. The data bytes located in the vicinity of the selected address were displayed in the Data field, and the selected byte could be changed, after which you had to click **Set**, and the entered byte was changed.

If in Program options there was an option for **Fast Decompilation** enabled and the option **Cache decompiled code** disabled, then all changes were reflected in that function, where the bytes were modified. To do this, it was enough to reopen that same function by double-clicking on its name in the tree.

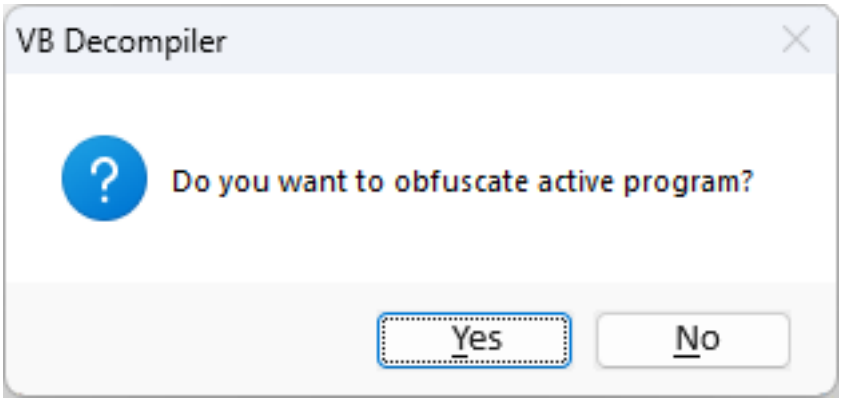
Important note: this functionality is designed for editing only your program files or those for which you have the appropriate rights, as well as researching malicious code in anti-virus laboratories. Before changing someone else's code - make sure to review the license agreement regarding the presence of such a right.

7.6 Obfuscation

If you are developing programs in **Visual Basic 5.0/6.0**, you can increase their protection against decompilation. To achieve this, the **Obfuscator** is embedded within the decompiler, capable of removing all function, form, module, and object names from the binary file of your program, as well as various signatures. This will make analyzing the program more difficult whether using the provided decompiler or other solutions. Moreover, most third-party products will refuse to decompile this file altogether.



To apply this protection, open your project in the decompiler and select **Obfuscate project** from the **Tools** menu. This process does not take much time, so we recommend obfuscating your projects before publishing them.



This feature is also integrated into our comprehensive solution for protecting software - **DotFix NiceProtect**, which can be downloaded from www.niceprotect.com.

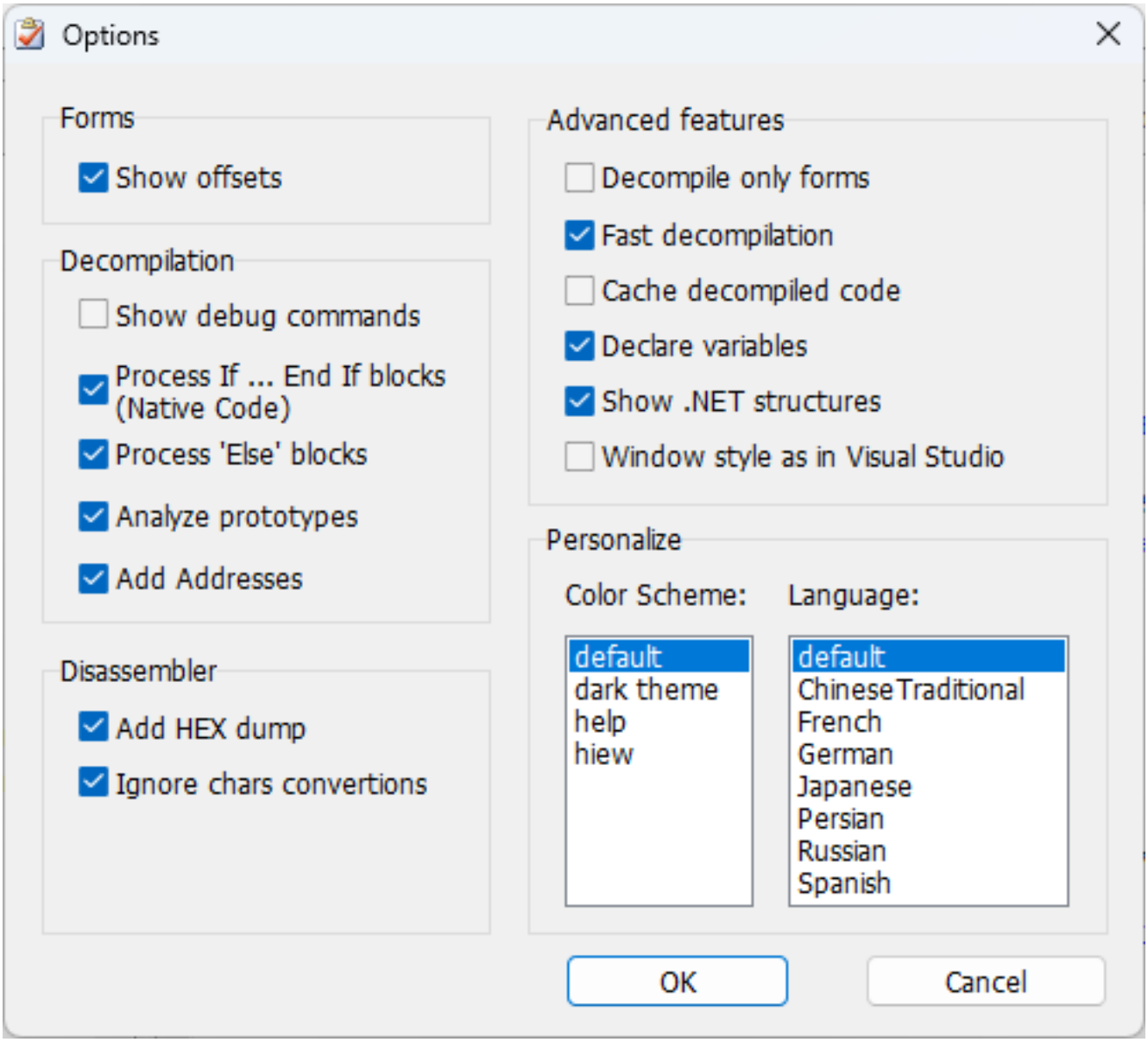
Obfuscation is available only to users with a Business license of the **VB Decompiler**.

Obfuscation restrictions

8 Program options

8.1 Program options

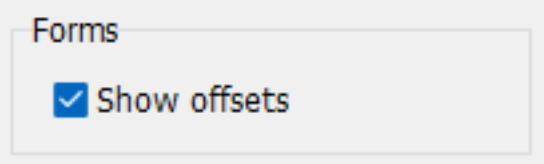
To change the program settings, go to **Tools** menu > **Options**. In the resulting dialog, you will see the following options:



Forms
Decompilation
Disassembler
Color Scheme Advanced features Language selection

8.2 Forms

You can change the following decompilation settings for forms:



In this section of settings, you can enable displaying offsets next to form objects. This may be necessary, for example, when localizing a program in another language without access to source codes. **Show offsets** will allow you to find the location within the program where these objects are located, their names and properties.

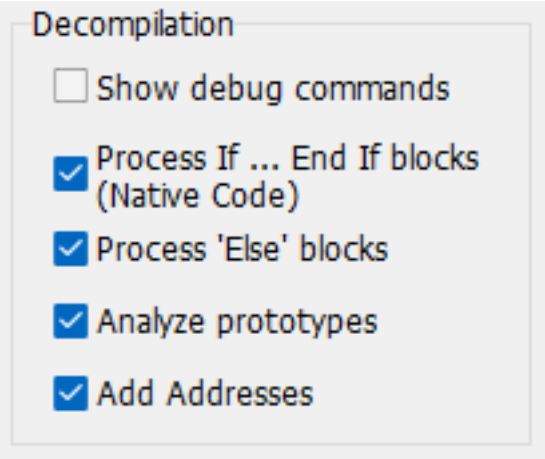
After which, using a HEX Editor, you can modify these values. Note that if your knowledge is limited to basic user level and you're unfamiliar with the inner workings of binary files - it is recommended not to enable this option. As any incorrect modifications made to the file will make it non-functional, and unnecessary addresses will simply hinder you.

Visual preview and event navigation

Form options are separate from the **Designer (read only)** viewer. See Visual form reconstruction for classic VB forms, Windows Forms, WPF/BAML support, event-handler navigation, and the limitations of each interface technology.

8.3 Decompilation

Open **Tools > Options** to configure decompilation. When comparing settings interactively, use Fast Decompilation without caching; otherwise process the file again after changing an option.



Show debug commands

For classic VB P-Code, includes compiler and debugger operations that would otherwise be omitted from the high-level listing. Enable it when inspecting the underlying instruction sequence rather than only the reconstructed program logic.

Process If ... End If blocks (Native Code)

Reconstructs recognized Native Code conditional jumps as `If...End If` blocks. Disable it when you need to inspect the original branching pattern or investigate a procedure whose structure appears incorrect.

Process 'Else' blocks

Uses additional heuristics to distinguish Else branches from nested conditions and GoTo transitions. For Native Code, this requires processing of If blocks. Keep it enabled for normal reading; compare with it disabled when a reconstructed branch is ambiguous.

Analyze prototypes

Performs preliminary heuristic analysis of procedure arguments and return values. This is particularly useful with Fast Decompilation: private functions and procedures in standard modules may lack explicit signature metadata. The preliminary analysis takes additional time, but avoids treating a call solely according to the currently known stack contents. A heuristic signature is still an inference, especially when metadata, external types, or code are incomplete.

Add Addresses

Displays virtual addresses next to reconstructed statements. Keep it enabled to correlate code with disassembly, file bytes, and findings from other tools. It is also needed for **Copy to disassembler** to merge high-level statements with their low-level counterparts. The Disassembler always displays addresses independently of this setting.

Understanding Native Code reconstruction

The Native Code engine recognizes supported `Select Case` patterns, multidimensional array access, COM calls, and VB runtime operations. Error handling can be reconstructed as `On Error Resume Next`, `On Error GoTo`, and `On Error GoTo 0`. Compiler optimization and missing type information can still limit the result; inspect the disassembly when the high-level output is incomplete.

Version 26.2 expanded Variant typing, multidimensional SafeArray reads and writes, and `VarPtr` reconstruction for array elements. The output may identify an array reference as `SafeArray`, reflecting the runtime structure recognized

by the analysis engine. Branch-state isolation and temporary-variable analysis also improve reconstruction of complex If/Else code. Version 26.5 further improves Select Case and VB5/VB6 runtime-function handling.

COM and ActiveX type information

VB Decompiler first uses its built-in knowledge base for known interfaces. For an unknown external COM object, it can resolve library, class, and interface identifiers through the system registry and parse the corresponding DLL/OCX type library. This supplies method and property signatures, including returned interface types, for early-bound and supported late-bound calls.

If a required type library is absent or its registration cannot be resolved, some calls may remain unidentified, for example as `UnkVCall`. Missing dependencies are therefore relevant when comparing output between analysis machines. The expanded IID knowledge base includes Office automation, ADO/DAO, XML and HTTP, scripting, shell, messaging, and directory-service interfaces; it does not imply that every possible COM call can be resolved.

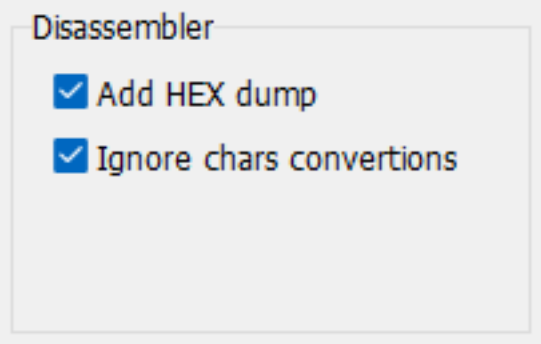
Only limited UDT metadata survives in some VB binaries. The decompiler can recover information such as a structure's size and string-field count without recovering the complete original field layout. Do not interpret a partial declaration as a complete source-level definition.

.NET reconstruction and optimization

Supported IL is reconstructed as C#-like expressions and control flow. The **Procedure analyzer and optimizer** control in the main window can simplify supported loop, conditional, operator, and temporary-variable patterns. Inspect IL and metadata where an unsupported or obfuscated fragment remains. This processing is independent of AI Helper.

8.4 Disassembler

This block of options is responsible for disassembling machine code:



Add HEX Dump displays machine codes before each disassembled command, which may be necessary only when modifing the binary file after analysis in a HEX Editor.

Ignore chars convertions is used to disable converting non-English string data into numerical Unicode. Enabling this option makes the displayed strings easier to read. Disable it when you need the numeric Unicode representation.

Show user classes includes parsing user classes inside **Delphi** / **Free Pascal** applications. This option is not used for **Visual Basic** and **C#** applications.

Show VCL classes includes decompiling all internal classes of the **VCL** library. It is recommended to disable this option so that the disassembler does not analyze thousands of service classes, which are usually unnecessary when analyzing programs. This option is only used for decompiling **Delphi** applications for internal purposes and was created for unification with other products, but it is not implemented in **VB Decompiler**.

Partial decompilation of **Delphi** and **Free Pascal** applications is currently offered exclusively as a separate service (not as a software).

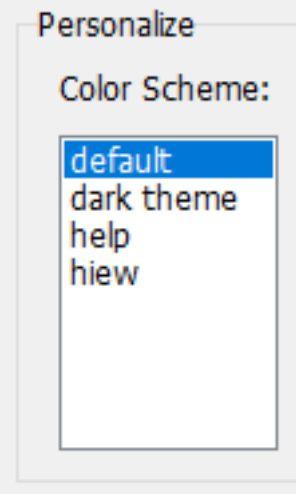
VB P-Code and .NET IL

Add HEX Dump also supports the low-level P-Code listing. Use **Copy to disassembler** from the Decompiler view when you need to compare recovered statements with P-Code or native instructions in one listing.

For .NET assemblies, the Disassembler can show IL or native x86/x64 methods as applicable. Recovered local-variable signatures can include arrays, generics, value types, and TypeSpec references. Enable **Show .NET structures** when you need to inspect metadata tables and heaps alongside the code. Later releases expand support for obfuscated metadata, including the FieldPtr table; this does not guarantee successful recovery of every damaged or obfuscated assembly.

8.5 Color Scheme

This section defines the color scheme and syntax highlighting when viewing code in a decompiler:



Presets are stored in the "colors" folder within .ini files located in the directory where VB Decompiler was installed. The filenames determine the preset names in the program's settings section. These files have a simple format, and you can easily create them yourself or edit existing ones.

The configuration format is as follows:

```
[Colors]
Background = decimal color code (for example: 1973790)
Foreground = decimal color code (for example: 14671840)
KeyWords = decimal color code (for example: 13471849)
LocationsAndArguments = decimal color code (for example: 11849128)
Procedures = decimal color code (for example: 5676246)
; next properties is only available in version 12 or later
UnderCursorBackground = decimal color code (for example: 6986069)
UnderCursorForeground = decimal color code (for example: 1973790)
UseDarkMode = 0 or 1 (0 for Light Theme, 1 for Dark Theme)
Colors2 = 0 or 1 (using extended colors at Colors2 section)
```

```
[Font]
Name = font name (for example: Consolas)
Size = font size (for example: 11)
KeyWordsBold = 0 or 1 (font bold)
LocationsAndArgumentsBold = 0 or 1 (font bold)
ProceduresBold = 0 or 1 (font bold)
```

```
; this section is only available in version 12 or later
[Colors2]
TVFontName = Solution Explorer treeview font name or list of font names, separated by comma (for example:
    Segoe UI, Tahoma, Arial)
TVFontSize = size of the font (for example: 10)
FontName = Decompiler and Disassembler font name or list of font names, separated by comma (for example:
    Courier New, Consolas, Liberation Mono, monospace)
FontSize = size of the font (for example: 11)
Background = BackColor in hexadecimal for Decompiler and Disassembler tab (for example: 1E1E1E)
Foreground = ForeColor in hexadecimal for Decompiler and Disassembler tab (for example: DFDFE0)
; code highllighting
Key = color in hexadecimal (for example: CD9069)
Comment = color in hexadecimal (for example: CD9069)
Char = color in hexadecimal (for example: B4CDA8)
Float = color in hexadecimal (for example: B4CDA8)
Hex = color in hexadecimal (for example: B4CDA8)
Number = color in hexadecimal (for example: B4CDA8)
String = color in hexadecimal (for example: 569CD6)
Symbol = color in hexadecimal (for example: DFDFE0)
; HEX Editor font and colors
HexFontName = font name or list of font names, separated by comma (for example: Courier New, Consolas,
    Liberation Mono, monospace)
HexFontSize = size of the font (for example: 11)
HexBackground = color in hexadecimal (for example: 1E1E1E)
HexForeground = color in hexadecimal (for example: DFDFE0)
HexGrid = color in hexadecimal (for example: 1E1E1E)
HexActiveFieldBackground = color in hexadecimal (for example: 1E1E1E)
HexOffsetBackground = color in hexadecimal (for example: 1E1E1E)
HexCurrentOffset = color in hexadecimal (for example: B4CDA8)
HexOddColumn = color in hexadecimal (for example: B4CDA8)
HexEvenColumn = color in hexadecimal (for example: B4CDA8)
HexOffset = color in hexadecimal (for example: 569CD6)
```

The color code is specified in the 32-bit true-color format. Decimal in a Colors section (for backward compatibility with old versions of the VB Decompiler) and Hexadecimal in a Colors2.

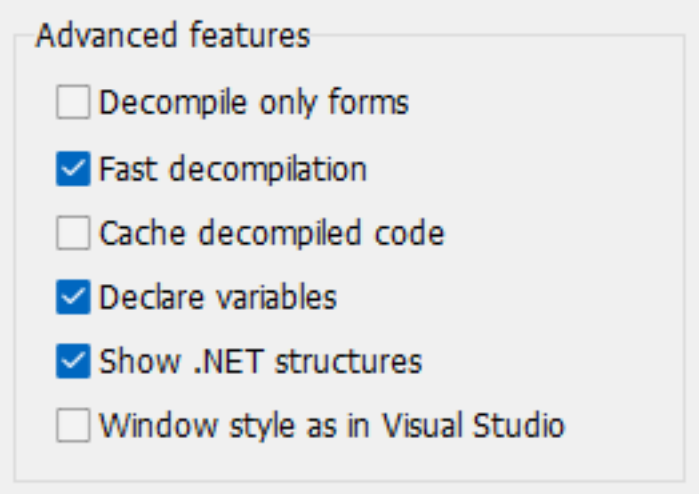
Changing themes while analyzing code

The theme system supports dark mode and can apply a selected theme without restarting VB Decompiler, including after editing its INI file. Font settings in the extended configuration can contain comma-separated fallback names, which helps when equivalent fonts differ between Windows and compatibility environments.

The under-cursor colors control identifier highlighting. See Viewing of code for local, member, and global variable highlighting.

8.6 Advanced features

These settings control how the project is processed and how recovered information is presented.



Decompile only forms

Disables code decompilation when you need only recovered form and UserControl definitions. For the visual viewer and its technology-specific capabilities, see Visual form reconstruction.

Fast Decompilation

When enabled, VB Decompiler processes the project structure first and decompiles individual procedures as you open them. This reduces the initial wait when investigating selected procedures. With **Cache decompiled code** disabled, you can change analysis settings and reopen a procedure to obtain a new listing.

Enable **Analyze prototypes** in the Decompilation settings to perform preliminary heuristic analysis of private procedure signatures. This adds initial processing time but can improve argument reconstruction without requiring full decompilation.

Fast Decompilation does not provide a complete searchable high-level listing of the project. Disable it and process the file again before using whole-project text search or exporting a complete decompiled project. String References are different: they can be collected through preliminary disassembly and are available in Fast Decompilation mode.

Full decompilation

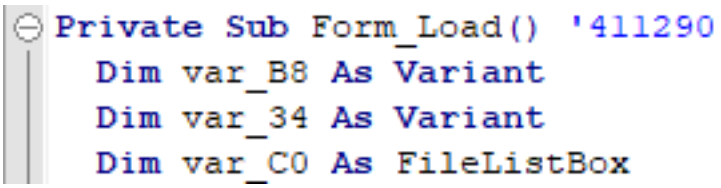
With Fast Decompilation disabled, the decompiler processes the project's procedures in advance. This takes longer, but makes the complete generated listing available for text search and export. Changes to decompilation settings require processing the file again.

Cache decompiled code

Retains already generated listings in Fast Decompilation mode. This avoids repeatedly decompiling an opened procedure, but cached results do not reflect later analysis-setting changes. When comparing **Parse stack parameters** or **Procedure analyzer and optimizer** settings interactively, disable caching.

Declare variables

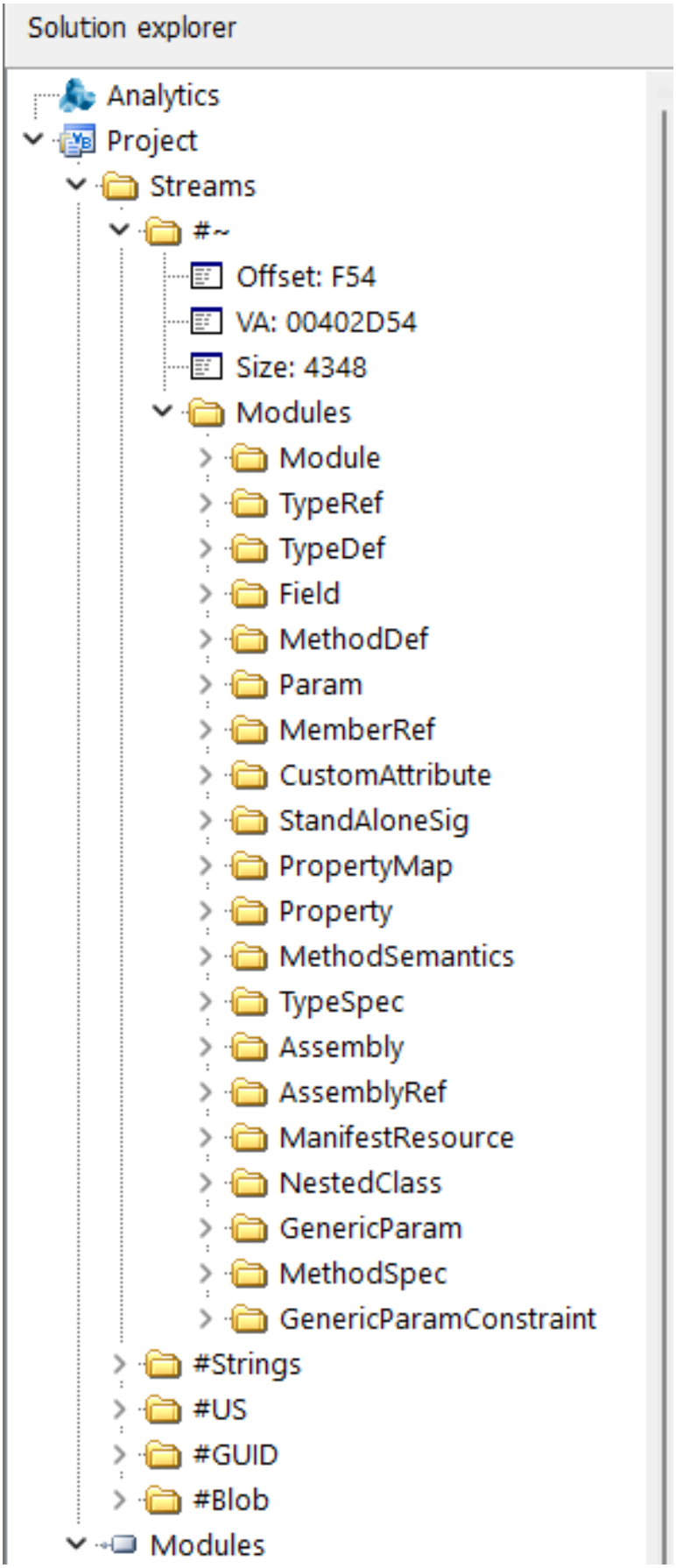
Adds a declaration block for recovered variables at the start of each decompiled procedure. This is useful for source recovery and type inspection. For a concise behavioral review, you may prefer to hide the declarations. Recovered types are subject to the metadata and analysis available to the decompiler.

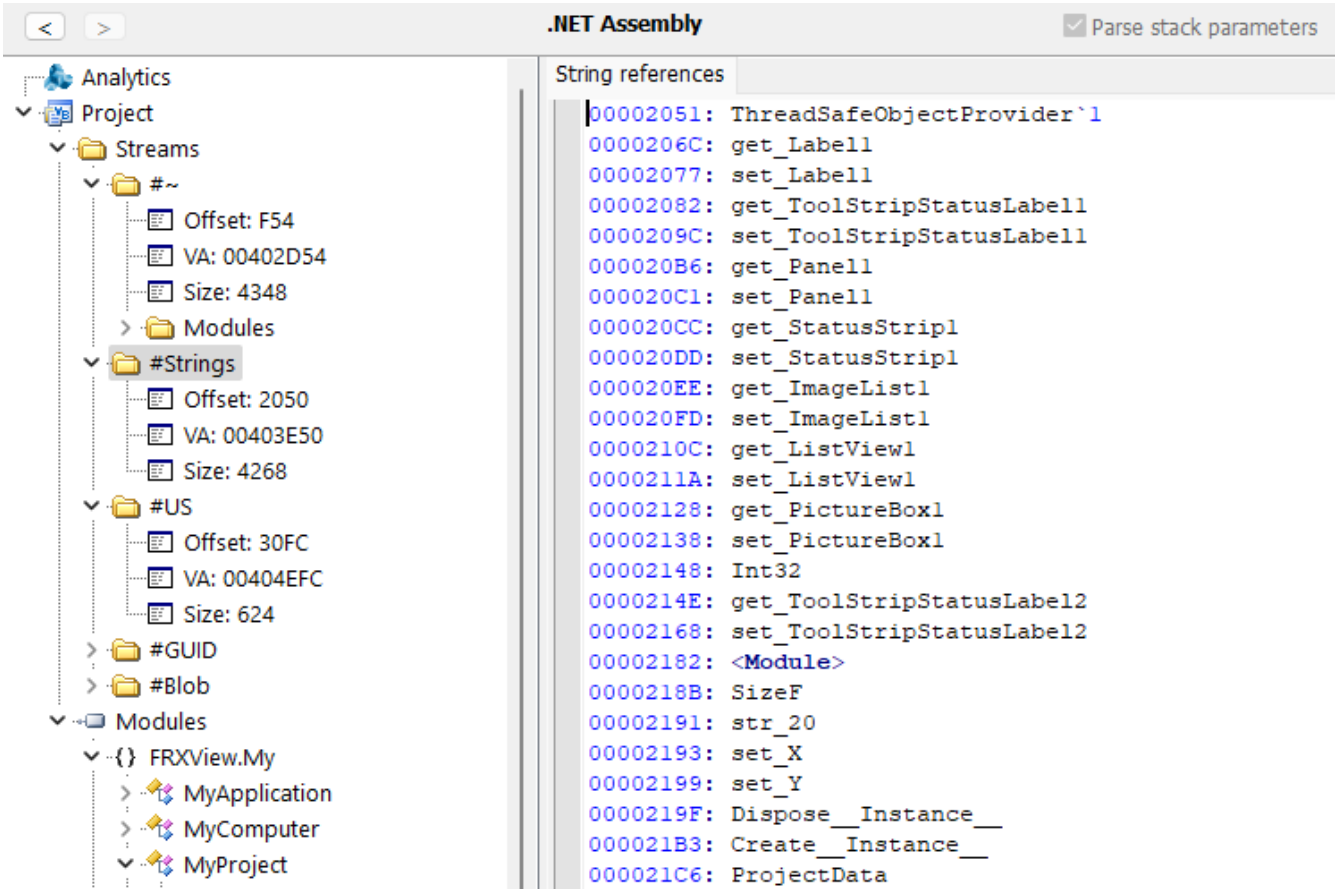


```
Private Sub Form_Load() '411290
  Dim var_B8 As Variant
  Dim var_34 As Variant
  Dim var_C0 As FileListBox
```

Show .NET structures

Exposes .NET metadata tables and heaps in Solution explorer. Use it to inspect structures and signatures alongside the code. Selecting a table displays its contents; the #US heap contains user strings. Disable the option when you do not need metadata navigation.


Solution explorer
Analytics
Project
Streams
#~
Offset: F54
VA: 00402D54
Size: 4348
Modules
Module
TypeRef
TypeDef
Field
MethodDef
Param
MemberRef
CustomAttribute
StandAloneSig
PropertyMap
Property
MethodSemantics
TypeSpec
Assembly
AssemblyRef
ManifestResource
NestedClass
GenericParam
MethodSpec
GenericParamConstraint
#Strings
#US
#GUID
#Blob
Modules



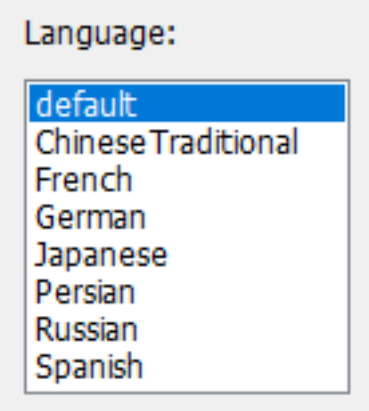
Window style as in Visual Studio

Places the code on the left and Solution explorer on the right. This changes the workspace layout without changing the recovered code.

8.7 Language selection

VB Decompiler provides interface localizations for more than 20 languages used worldwide. Documentation is also available in multiple languages.

You can choose the preferred language interface for the program in the **Tools** -> **Options** menu, under **Personalize** settings. After choosing the language, you need to restart **VB Decompiler** for proper application of language settings throughout the entire program. If restarting the program is undesirable, you may continue using **VB Decompiler**; however, some text may still be displayed in the previous language.



The help language is selected automatically based on your computer's system language. To read the manual in another language, open the corresponding help file from the folder where **VB Decompiler** is installed. PDF is the primary format for localized manuals; CHM help files remain available for some languages.

The English manual is the source for all translations. If you notice an error or inconsistency in a localized manual, please report it to technical support.

Accessing the manual

If a local help file or the required PDF viewer is unavailable, VB Decompiler can open the online documentation. For an offline investigation environment, keep the required local help files available in advance.

9 Plugins

Plugins extend VB Decompiler through native DLLs. Installed plugins are available from the **Plugins** menu. The callback API provides access to recovered project information and code, allowing a plugin to inspect or modify supported internal structures.

VB/DE Decompiler MCP Server 2.1.0

Connect an MCP client to a project already open in VB Decompiler or DE Decompiler. The plugin exposes modules, forms and procedures through the host's plugin SDK. It supports reading recovered listings without running the analyzed program.

Download MCP Server 2.1.0 for x86 hosts · Delphi source code

`vb_get_form` returns `form_index`, `form_name`, and `form_text`. When the decompiler supplies no name, `form_name` is an empty string. Use `vb_list_modules` with `query` to find units or classes, and `vb_search_module_code` to search function bodies. MCP tool reference and examples

Requirements

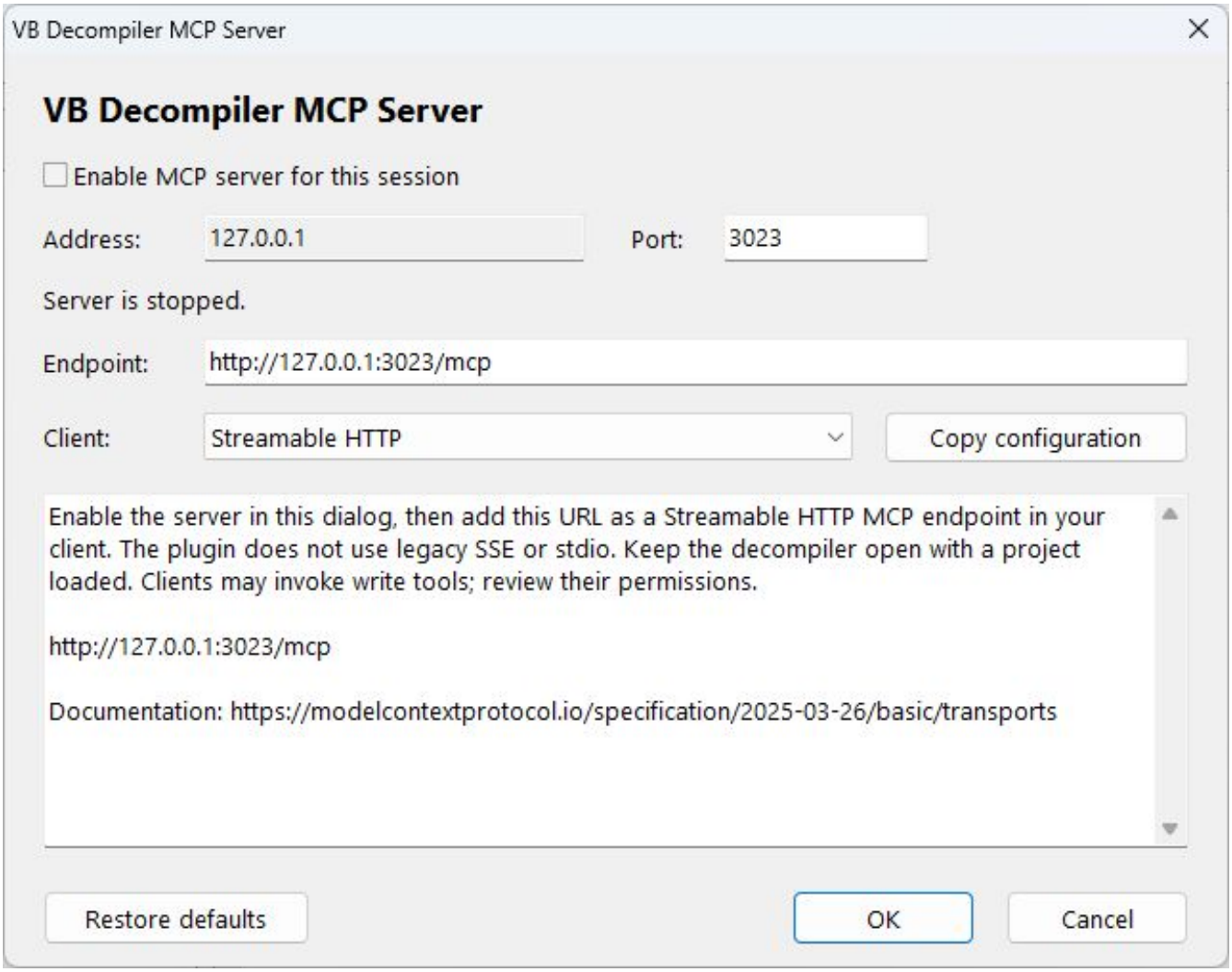
Use VB Decompiler or DE Decompiler for Windows with native DLL plugin support.

The archive contains a Windows x86 plugin. Match the DLL to the architecture of the decompiler that loads it. A 32-bit host can still analyze 64-bit files where its own feature set supports them.

The available tools depend on the host version, edition and open project. On older SDKs, some code reads use the existing decompilation cache and some workspace operations are unavailable. This plugin uses tool names beginning with `vb_`. If you replaced another implementation, reload the tool list in your client.

Install and start

1. Close the decompiler. Extract the archive and copy `vbmcp.dll` into the `plugins` subfolder of the decompiler installation. If the C version is already installed, replace its DLL instead of keeping both.
2. Start the decompiler and open a sample you are authorized to analyze. Choose **Plugins → VB/DE Decompiler MCP Server**.
3. Select **Enable MCP server for this session** and click **OK**. Copy the endpoint shown in the dialog.



VB Decompiler MCP Server settings. The plugin detects which decompiler loaded it and shows that product name in the window title.

Connect an MCP client

Use Streamable HTTP. The default endpoint is `http://127.0.0.1:3023/mcp`. If you run both decompilers with the server enabled, assign a separate port to each and use the endpoint shown in its dialog.

Codex

```
codex mcp add vbmcp --url http://127.0.0.1:3023/mcp
```

Claude Code

```
claude mcp add --transport http vbmcp http://127.0.0.1:3023/mcp
```

VS Code

In VS Code, merge this entry into `.vscode/mcp.json` or use **MCP: Add Server** and choose HTTP.

```
{
  "servers": {
    "vbmcp": {
      "type": "http",
      "url": "http://127.0.0.1:3023/mcp"
    }
  }
}
```

Client configuration references · Codex · Claude Code · VS Code

Check the connection

Open a sample that contains forms. Ask the client to list up to five forms using the read-only tool `vb_list_forms` with `limit` set to `5`. If the client cannot connect, check that the server is enabled in the current session and that the endpoint matches the dialog.

Local access and project changes

The listener binds to `127.0.0.1` and has no user authentication. A local application that reaches it can read project data or request changes to the current decompiler workspace, including edits to listings and function marks. Keep write approvals enabled in the client. To stop the server, clear **Enable MCP server for this session** and click **OK**. The server starts disabled each time the decompiler launches.

The plugin communicates locally. Your MCP client may send returned code or metadata to its model provider, depending on how the client is configured.

Developed in Delphi by Sergey Chubchenko (DotFix Software). The original MCP plugin provided the starting point. Distributed under the MIT License.

FRX Image Viewer

FRX Image Viewer 1.2.0 previews and extracts images registered by VB Decompiler or DE Decompiler. Use VB Decompiler for VB5/6 and .NET form graphics, and DE Decompiler for binary Delphi DFM graphics. Supported picture formats are BMP, PNG, JPEG, GIF, ICO and CUR.

Installation and opening

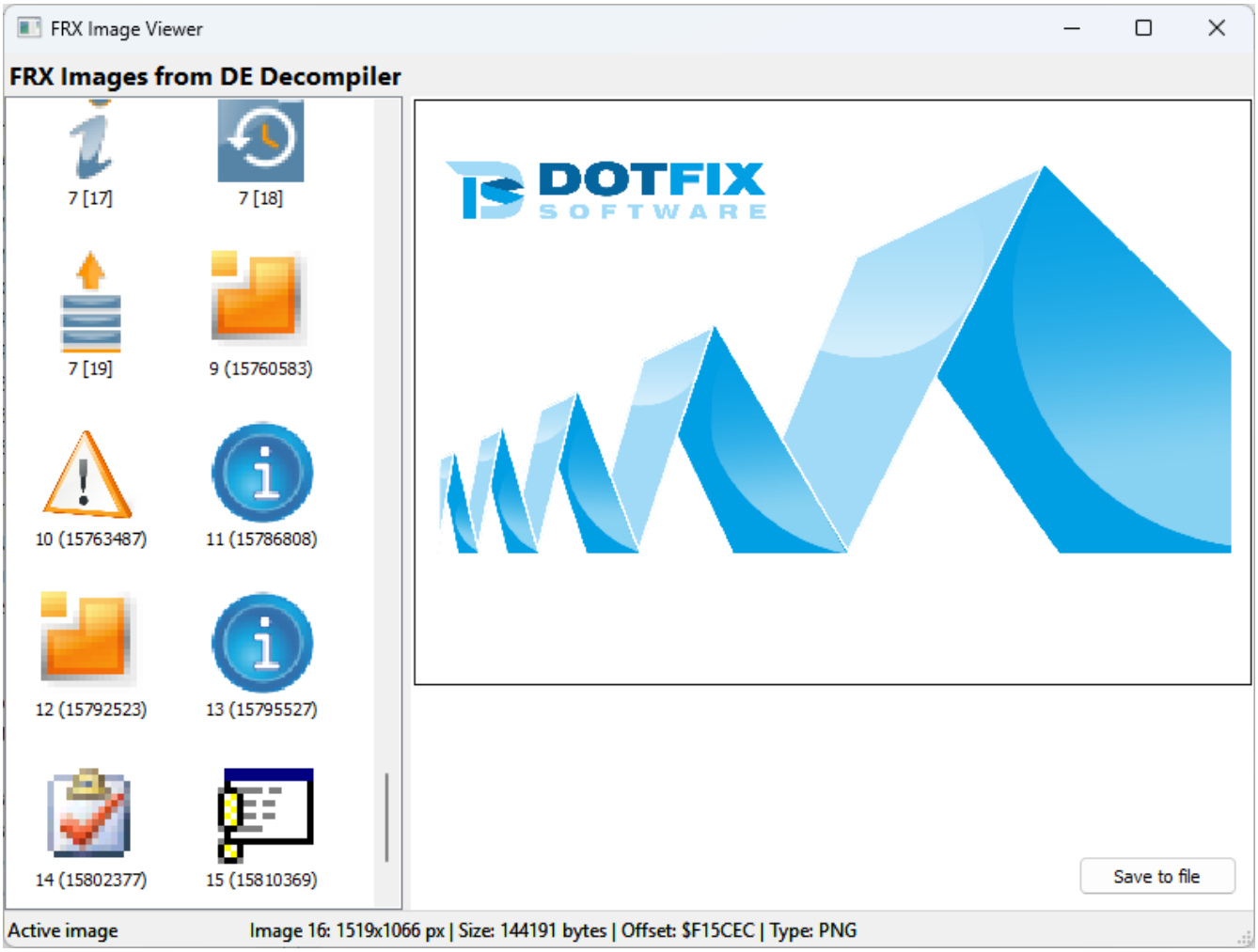
1. Copy `FRXViewer.dll` to the host installation's `plugins` subdirectory and restart the decompiler. The DLL architecture must match the host application, regardless of the analyzed file's architecture.
2. Open the executable or DLL and decode its forms so that the host registers the available images.
3. Select **Plugins** > **FRX Image Viewer**, then choose a thumbnail to inspect the image.

The 1.2.0 download contains a DLL for an x86 host. Use a matching build for an x64 host. The plugin requires Windows 10/11; its source repository includes build instructions.

Preview and export

The gallery and preview show available images, including compound ICO/CUR files. The status area identifies the selected item, dimensions, size or source-bitmap offset, and format. The viewer follows the host theme and supports dark mode.

Choose **Save to file** to export the selected item. Standalone pictures retain their original bytes and format. Individual ImageList cells are exported as ICO files with their masks and alpha transparency; these ICO files are derived from the source bitmap sheets.



FRX Image Viewer in DE Decompiler, displaying Delphi DFM graphics and individual ImageList icons.

Native Delphi ImageLists and compatibility

With a compatible host, recognized native Delphi ImageLists appear as individual icons. Independently encoded pictures in collection items remain separate ordinary images. Unsupported streams retain whole-picture handling wherever original image entries are available.

Shared SDK revision 26.6, supported by VB Decompiler 26.6 and DE Decompiler 26.5, adds optional `GetFrxImageListInfo` (63) and `GetProductName` (64). Older hosts retain the original picture API and whole-image behavior. See the SDK reference for capability detection and offset rules.

Viewing and extraction do not execute the inspected file or load its components. Animated GIFs display their first frame. The available assets depend on images successfully decoded and registered by the host.

Developing a plugin

The SDK includes examples for Visual C++, Rust, Delphi, and Free Pascal. See the VB Decompiler Plugin API reference for the exported entry points, callback signature, command identifiers, data representations, and links to the example projects.

A DLL plugin supplies its name and author information and receives the host callback through `VBDecompilerPluginLoad`. The callback is used to request or update supported project data. Follow the SDK's calling convention, pointer, string-encoding, and buffer requirements when implementing this interface; do not substitute a managed-language declaration without a compatible native boundary.

VB Decompiler initializes available DLL plugins at startup. Build plugins for the intended host architecture and Windows environment, and verify the resulting DLL with that environment. The SDK exposes both active-view code and

project-level code access; operations requiring all decompiled procedures need full decompilation rather than only an on-demand listing.

10 Visual form reconstruction

The **Designer (read only)** tab displays a visual reconstruction of supported interfaces from the analyzed binary. It helps connect interface elements with application behavior without manually interpreting every recovered property or initialization statement.

Opening a recovered form

1. Open the application and let VB Decompiler recover its project structure.
2. Select a recovered form in Solution explorer and open **Designer (read only)**.
3. Select a control to inspect its available properties and recovered event information. Use the events to navigate to the corresponding handler code.

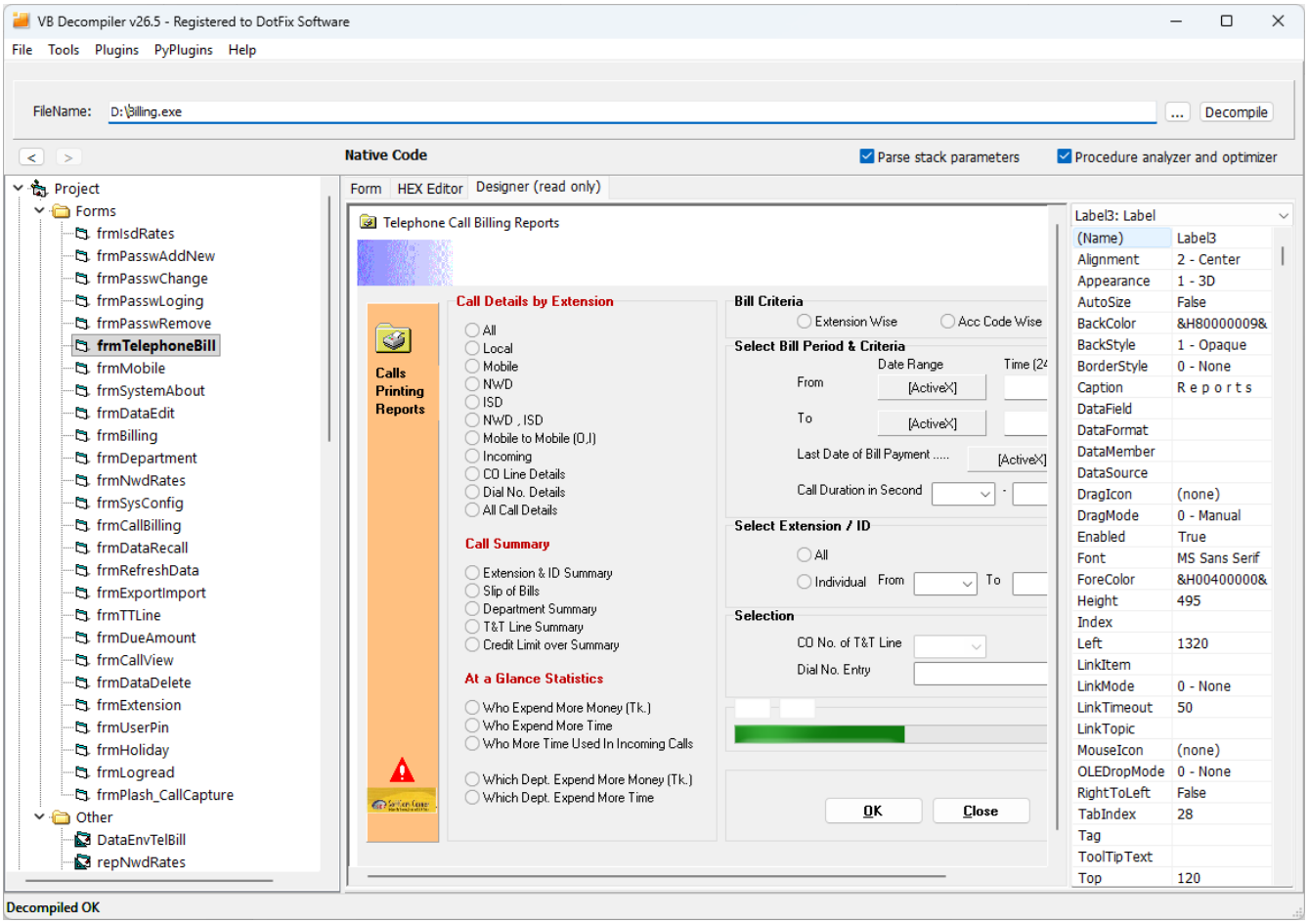
The preview is reconstructed from stored information. It is not a running instance of the original interface. Unsupported controls, runtime-created content, missing resources, and custom layout logic can produce incomplete or approximate results.

Visual Basic 5.0/6.0

The viewer reconstructs forms and UserControls, including supported control properties, fonts, colors, and images. It reads binary form data used by lists, combo boxes, and other controls. Images can include icons, cursors, bitmaps, JPEG, PNG, and GIF data.

Visual menu reconstruction supports both P-Code and Native Code applications. Supported controls include `SSTab`, `MSComctlLib.StatusBar`, `MSComctlLib.ListView`, `MSComctlLib.ProgressBar`, and `MSComctlLib.TreeView`.

To inspect a classic VB control's handler, select the control and double-click its **_Event** field in the property list. Choose the required event from the resulting list to open the handler.

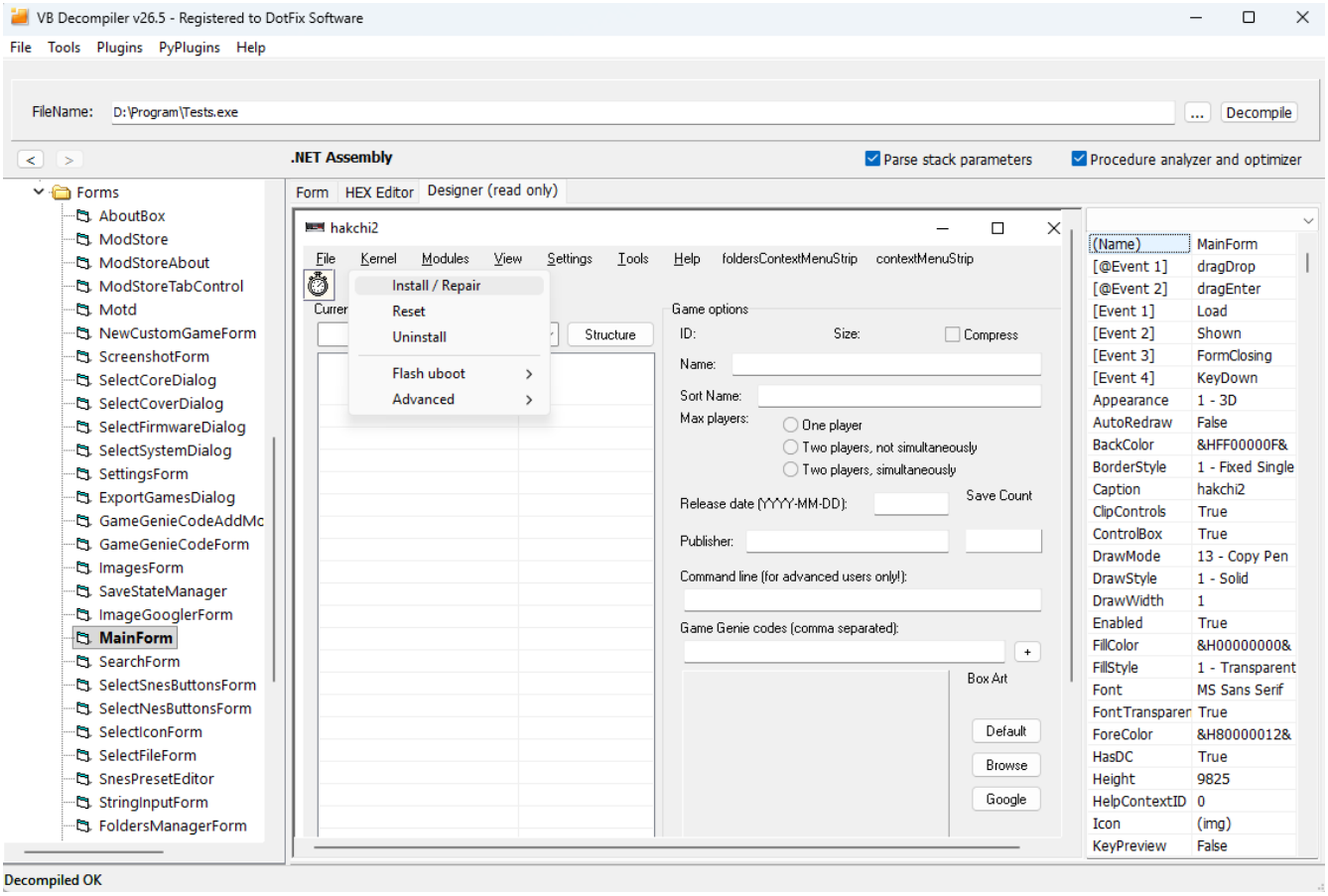


.NET Windows Forms

VB Decompiler statically analyzes `InitializeComponent` and reconstructs supported Windows Forms controls and their layout. The parser and renderer do not require the .NET runtime or execution of the target application. Supported recovered images can also be displayed on controls and form backgrounds.

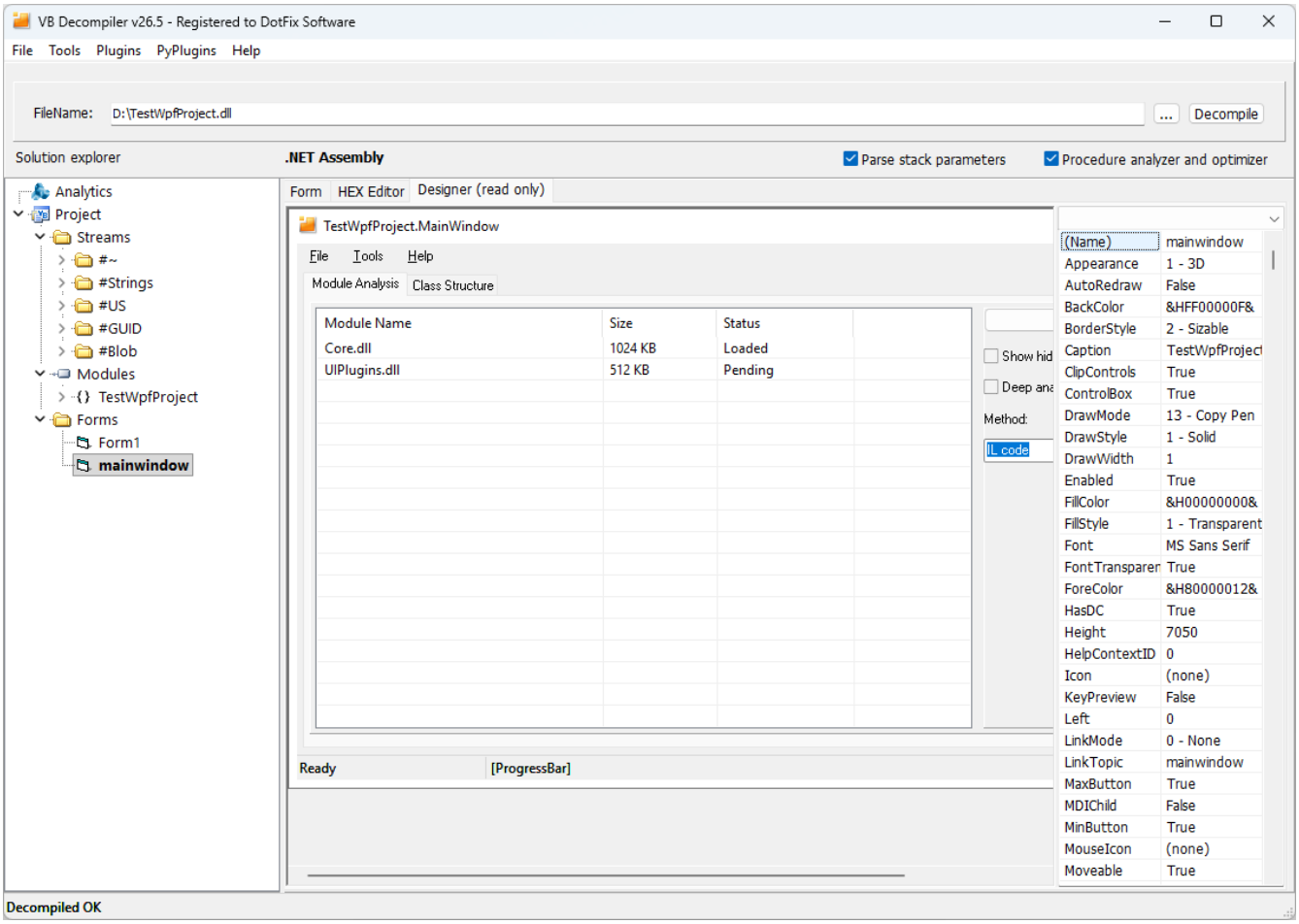
Select a control to inspect its recovered events, then double-click an event to open the corresponding handler. The viewer reconstructs menu and context-menu hierarchies; clicking a recovered menu item opens its corresponding Click event. Custom Windows Forms events are shown with an @ prefix.

The viewer handles control Z-order, inherited foreground and background colors, `Color.FromArgb` values, numeric `FontStyle` flags, Boolean resource properties, and `MinimumSize` alongside `ClientSize`.



WPF windows, pages, and user controls

The viewer reconstructs supported WPF windows, pages, and user controls from compiled XAML resources stored as **BAML**.



Supported layout containers include **Grid** with fixed, Auto, and proportional (*) row and column sizes, **StackPanel**, **DockPanel**, and **Canvas**, including nested layouts. Reconstruction takes supported sizes, margins, padding, alignment, orientation, docking, and Grid spans into account.

Beyond common input controls, the viewer can reconstruct tabs, group boxes, expanders, menu hierarchies and separators, status bars, ListView columns and supported static data, static TreeView nodes, and static combo box items. Ribbon interfaces, navigation panes, and docked panels can have simplified representations.

Where resource data is available, the viewer recovers names, visible text, tooltips, fonts, background colors, enabled and visibility states, and embedded images, including supported PNG transparency and background opacity. Unsupported custom controls can appear as placeholders identifying the original WPF type.

The WPF viewer links recovered control events to their handlers. Use the event information in the visual viewer to open the corresponding decompiled code.

Reconstruction limits: the visual preview is approximate. Templates, styles, bindings, custom controls, and dynamically generated content may be represented partially.

Inspecting and extracting form data

See Forms for the Show offsets setting and linked binary inspection. Use the FRX Image Viewer plugin to inspect and export supported recovered image assets.

11 Tracing

Tracing features provide step-by-step emulation for Visual Basic 5.0/6.0 Native Code and P-Code. This functionality requires a license with Tracing features. The tracer operates on the internal emulator's representation of a procedure; it does not execute the target procedure on the workstation's CPU.

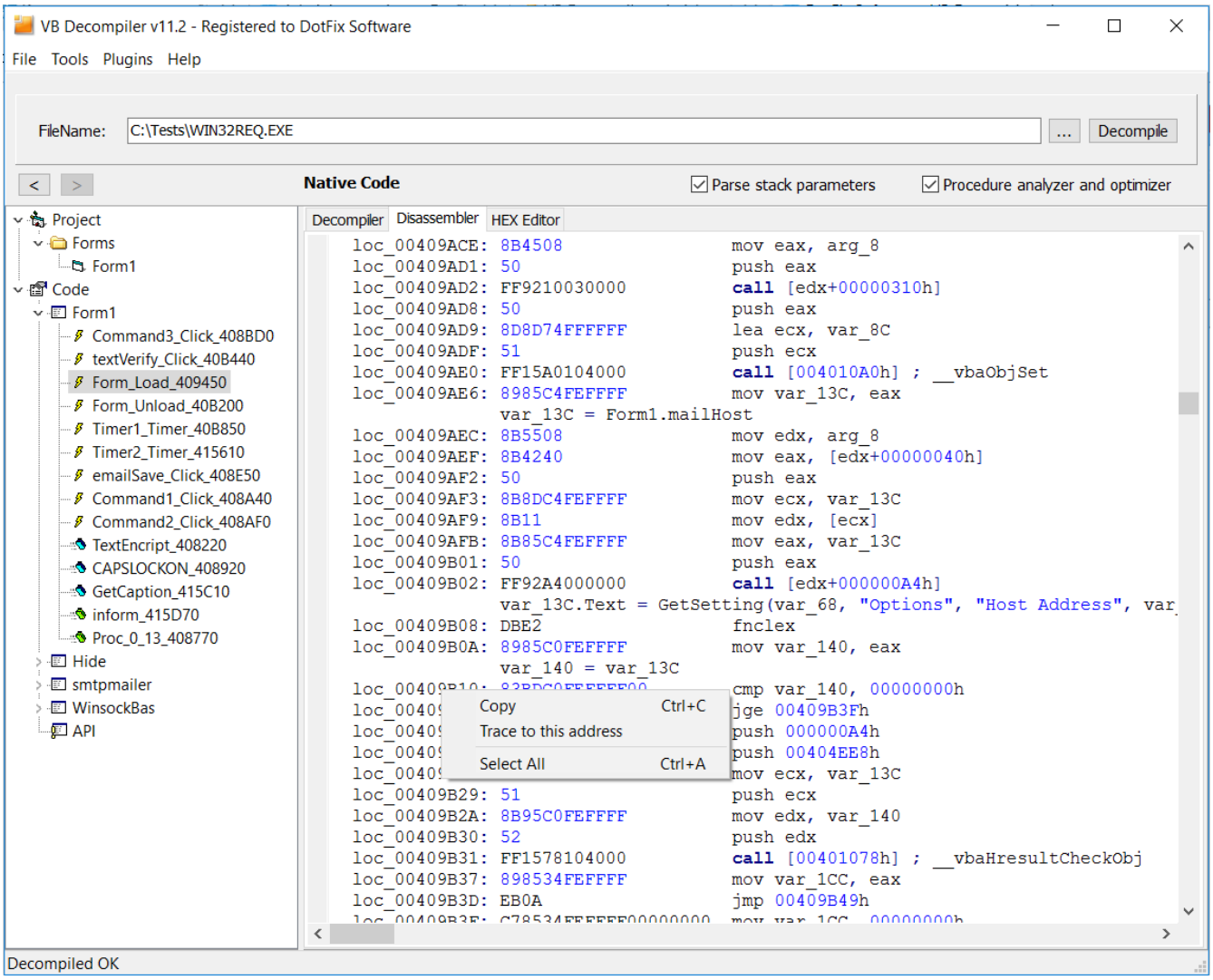
Starting a trace

1 Open a VB5/6 procedure and keep addresses visible in the listing.

2 On the line to investigate, invoke **Trace to this address** from the code context menu, or press **F4**.

3 Inspect the tracer's code and state displays and advance using **Next**. Cancel the trace when finished.

From version 26.4, the command can resolve the leading `loc_` address after trimming whitespace at the start of the line. You can place the caret elsewhere on that line rather than precisely on its address label.



Native Code state

The tracer exposes CPU and FPU registers, the stack, variables, and the reconstructed code as emulation advances. Use these views to examine how a value or expression was derived. Variable-list search and display of the current object's recovered type can help with larger procedures.

P-Code tracing

P-Code tracing, introduced in version 11, shows how variable information and decompiled statements develop during processing. It can help explain differences between the intermediate P-Code operations and the optimized high-level listing.

Scope and limitations

- Tracing applies to an individual procedure. It starts with an empty emulated register, stack, and variable state rather than the state of a running process.
- Unsupported Native Code instructions may be skipped. External function calls are not executed; supported calling-convention handling can adjust the modeled stack without reproducing the external function's behavior.
- The tracer does not provide a complete processor-flags display. Do not interpret its state as a complete hardware execution trace.
- The documented model is forward step-by-step emulation, without reverse tracing or repeated execution of loop iterations.
- This workflow describes VB5/6 tracing. The presence of .NET disassembly does not imply a .NET runtime debugger.

Use tracing to investigate the decompiler's reconstruction and intermediate values. An emulated value alone does not establish the value produced by an actual process with initialized globals, external dependencies, and input data.

12 Analytic features

Licenses with **Analytic features** generate categorized reports of recognized operations in the analyzed program. These reports help security analysts identify procedures worth investigating before reading the entire codebase. They support Visual Basic 5.0/6.0 applications and, from version 11.7, .NET assemblies.

Generating a report

1 Enable **Analyze prototypes** in **Tools > Options**.

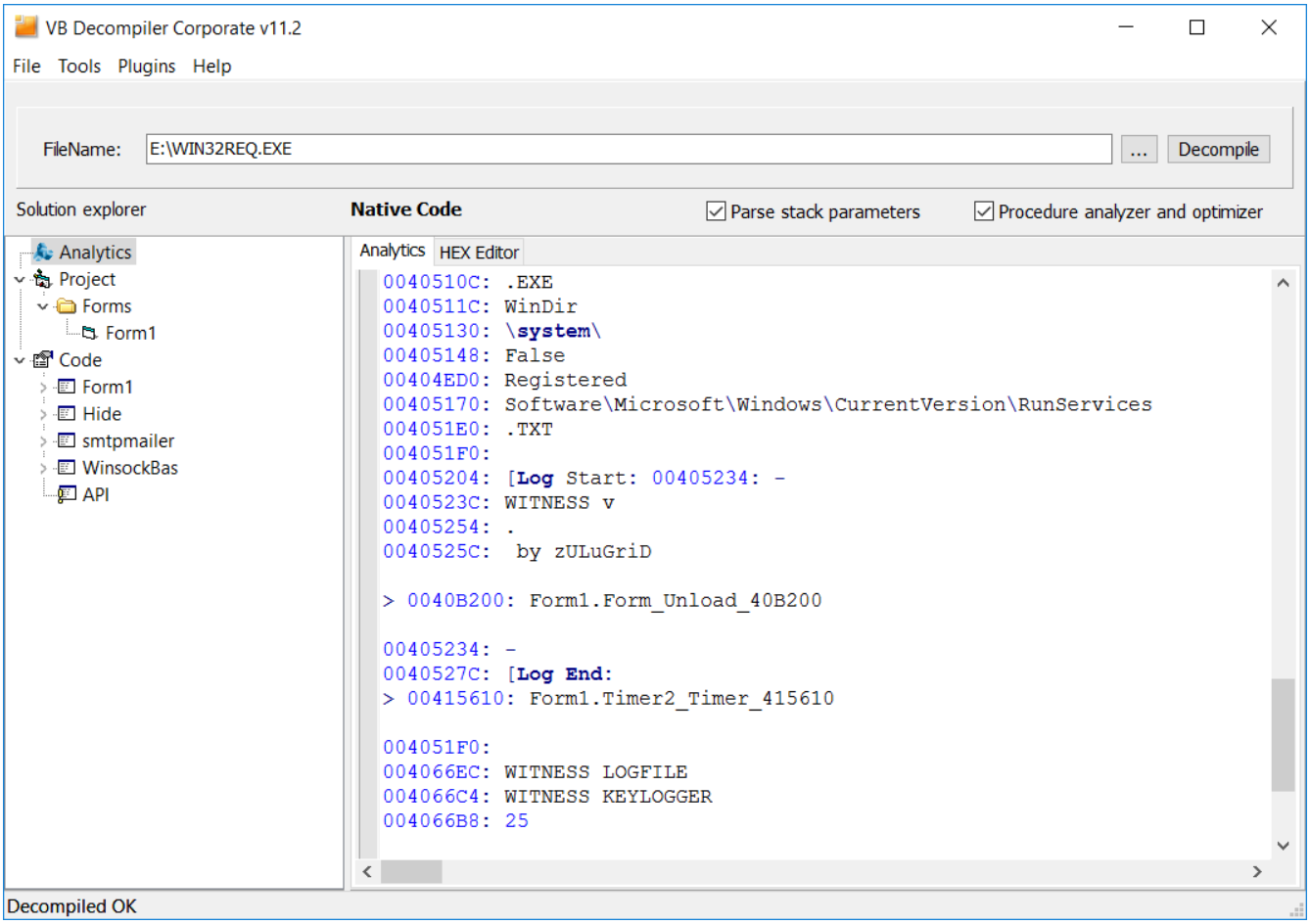
2 Open the file and wait for analysis to complete. With the required license and settings, the report is generated and displayed automatically.

3 Review the categories, then open relevant procedures and inspect their code and string references.

Categories cover recognized operations such as file and volume access, registry access, network activity, process creation, window operations, and selected VB internal functions such as `CallByName` and `VarPtr`.

Quick report and string context

The quick report lists addresses and names of procedures containing recognized operations. The extended report adds the available string references from those procedures. This context can reveal candidate paths, registry keys, host names, or other values that deserve examination.



Strings that are encrypted, obfuscated, or generated at runtime may be absent. A string associated with a procedure is useful context, but does not by itself prove that it is passed to a particular API call or that a branch is executed.

Navigating and saving results

Double-click a procedure name in the report to open its code. Double-click an address preceding a procedure or string to inspect its location in the HEX Editor. Use the **A** button above the object tree to return to the report at the previous position, and the navigation arrows or **Esc** to revisit inspected procedures.

Select and copy report text as needed, or choose **File > Save analytics report** to save it for further analysis.

Interpreting findings

The report identifies statically recognized code patterns and references. It is a triage aid, not a record of observed execution or an automatic verdict that a file is malicious. Indirect calls, unresolved external interfaces, damaged metadata, obfuscation, and runtime-generated values can limit coverage. Confirm significant findings in the corresponding procedures, disassembly, and file data.

For deeper inspection of VB5/6 reconstruction, see Tracing. For literal strings and their code locations, see String References.

13 Obfuscation restrictions

VB Decompiler, armed with a unique obfuscation technology, effectively conceals data left behind by the compiler in our research on decompiling **Visual Basic 5.0 and 6.0 applications**. Modern programming languages often store significant amounts of metadata and debugging information within compiled files, making it easier for crackers to locate desired functions within your program. To counter this threat, we've developed an obfuscator specifically designed for **Visual Basic** applications.

The Obfuscation Process: A Closer Look

The obfuscator works by breaking down the entire program into its constituent parts, including forms, modules, classes, and controls like buttons and text fields. Each form is then dissected into individual controls with associated events. Next, information about functions and procedures, as well as events tied to each object in the program, are gathered. During this process, the obfuscator checks for dependencies between objects and synchronously alters their names to make it challenging to determine which function belongs to a particular module or class.

As a result, finding specific forms like **frmTrial** or **frmRegistration** within the program becomes significantly more difficult.

Limits of Visual Basic 5.0 / 6.0 Application Obfuscation: Key Considerations

The algorithm employed by the obfuscator is complex and designed to analyze object dependencies, but it still doesn't support everything. Consequently, in some cases, you'll need to prepare your program beforehand for the obfuscation process. The points discussed below apply exclusively to the obfuscation of Visual Basic 5.0 / 6.0 applications.

1. The obfuscator does not support design-time references to other objects, which can lead to issues during runtime.

For instance, consider a scenario where you have a **TreeView** control that relies on an associated **ImageList**. If the name of this **ImageList** object is hardcoded into the properties of your **TreeView**, any attempts to obfuscate will result in errors, as the original reference remains intact. This limitation also applies to database-related components, such as **DataSets**

```
Begin ComctlLib.TreeView TreeView1
  Height          =   1932
  Left            =   3600
  TabIndex        =   3
  Top             =   1320
  Width           =   2412
  _ExtentX        =   4255
  _ExtentY        =   3408
  _Version        =   327682
  Style           =   7
  ImageList       =   "ImageList1"
  Appearance      =   1
End
```

This issue can easily be resolved by moving the assignments to Runtime from the Design-time, just by moving these assignments to the **Form_Load** event:

```
Private Sub Form_Load()
  TreeView1.ImageList = ImageList1
End Sub
```

The same correction needs to be applied for each same instance. Fortunately, there are usually only a few such cases in most projects.

2. Early object declarations are not supported (LateCall calls)

This limitation typically doesn't affect the majority of VB6 projects, but it's essential to keep it in mind. If you're working with forms as classes and creating copies of form objects, you might be familiar with this scenario:

```
Private Sub Form_Load()
  Dim oForm as New frmMain
  oForm.Show
End Sub
```

In such cases, the VB6 compiler inserts the name **frmMain** directly into the code of the **Form_Load** function. Since our obfuscator only modifies data and not code, this insertion remains invisible after obfuscation. Unfortunately, this leads to a program termination with error.

Unfortunately, it's impossible for us to support such calls in obfuscator of **VB Decompiler**, as we need to disassemble and decompile the entire code of all functions. Despite having this capability, decompilation is an extremely time-consuming process. For large projects, it can take over half an hour just to decompose the entire codebase. Moreover, we also need to analyze references to objects and rename them, which adds even more processing time.

Given these technical constraints, we decided not to implement full decompilation and code analysis in obfuscator of **VB Decompiler**, especially since this would only address the issue within too rare cases in Visual Basic 5.0 / 6.0 apps and wouldn't provide any benefits for most customers.

14 Support

If you have questions regarding the program and cannot find answers in this help guide, please contact our technical support team. This option is only available for registered users. For non-registered users, we are happy to assist with inquiries regarding purchasing license for **VB Decompiler**.

Additionally, if you encounter any bugs within the program or have suggestions on how to improve it, feel free to reach out to our technical support service with your feedback.

15 Registration

To use the full capabilities of the **VB Decompiler**, you need to purchase a **Business License**. By purchasing a Business license for Single Developer or Enterprise license, you will receive the ability to:

1. Recover close to 85% of code in typical supported **Visual Basic 5.0/6.0 P-Code** applications;
2. Recover up to 85% of code in typical supported **Visual Basic 5.0/6.0 Native Code** applications;
3. Decompile typical supported, non-obfuscated **.NET** assemblies into **C#**-like code, with recovery of up to 95%;
4. Always have the latest version of the program: updates are absolutely free for one year after purchasing the license;
5. Receive technical support via email for one year after purchasing the license;
6. Support the development of this decompiler.

Additionally, we recommend considering purchasing special licenses for the Malware Analysts and Forensics:

1. license with tracing features
2. license with analytic features

To purchase a **License** of the decompiler right now, select the **Order Pro version** option in the **Help** menu of the program. At this moment, you must be connected to the internet. A browser window will open to the program's website where you can buy the license.

The **Business License** of the decompiler is designed for use on only one computer and only by a **Single Developer** (license owner). If you need to use the program on multiple computers, you will need to purchase a separate license for each or consider purchasing the **Business Enterprise license**.(can be used on up to 20 computers).

16 About author

The decompiler is developed by the **DotFix Software**.

Product website: www.vb-decompiler.org
Support email: admin@vb-decompiler.org