DE Decompiler

User manual

English

Delphi 3-13 / C++Builder · VCL

DotFix Software

Contents

1 Getting started 4

1.1 Introduction 4

1.2 Quick start 6

1.3 Compiler compatibility and RTTI 7

1.4 The main window 8

1.5 Units, classes and functions 10

2 Opening and saving 12

2.1 Opening a file 12

2.2 Analysis databases 13

2.3 Saving recovered material 14

2.4 Saving the procedure list 15

3 Examining an application 16

3.1 Viewing and comparing code 16

3.2 Navigating by address and tree path 19

3.3 Finding procedures and text 20

3.4 String references 21

3.5 Decompilation from an address 22

3.6 Inspecting and patching bytes 23

3.7 VCL forms and visual inspection 24

4 Options and tools 26

4.1 Program options 26

4.2 Decompilation processing 27

4.3 Disassembly display 29

4.4 Themes and fonts 30

4.5 Processing modes and caching 32

4.6 Interface language 33

4.7 AI Helper 34

4.8 Plugins and the SDK 36

4.9 Obfuscation 39

5 Reference 40

5.1 Interpreting recovered code 40

5.2 Technical support 41

5.3 Licensing 42

5.4 About the developer 43

1 Getting started

1.1 Introduction

Professional decompiler for VCL applications written in Delphi or built with C++Builder. Analyze Win32/Win64 code, forms, units and classes for reverse engineering, forensics and code recovery.

Decompile **VCL applications written in Delphi 3–13 or built with C++Builder, for Win32 and Win64**, into Delphi-like Pascal. Recover control flow, calls, forms, units and classes from native instructions and available metadata, then follow the implementation from a VCL event to its handler.

Use the recovered structure and code to investigate program behavior, malware and potentially unwanted or concealed functionality. Follow VCL events to their handlers, examine control flow and calls, and use disassembly and the HEX Editor for detailed inspection.

Native-code decompilation

The high-level listing reconstructs program logic, including `if` conditions, `case` blocks, loops and calls to recognized VCL methods. Recognized units, classes and public functions provide a structural view; heuristic discovery also looks for private functions within each unit.

Forms and tools

Inspect recovered VCL controls and properties in the read-only designer and navigate from events to their handlers. Export units and forms as a Pascal/DFM project or a combined `.pas` listing. The integrated obfuscator supports Win32 and Win64 VCL applications written in Delphi or built with C++Builder.

Information retained in the binary

The result is Delphi-like code, rather than the original source text. Native compilation does not preserve all source-level information. Local variable names are commonly generated, data structure layouts may be inferred from access sizes and addresses, and an unidentified call may be displayed without its original VCL method name. Compiler-specific RTTI knowledge helps recover the names and types that remain available.

The high-level listing reconstructs native program logic for analysis with the detail supported by that evidence. Check inferred types, unexpected expressions and important conclusions against the corresponding instructions. See Interpreting recovered code.

Source recovery is an additional use. The output does not reproduce a compilable source project. Before opening recovered code in an IDE, manually analyze and correct decompilation inaccuracies and incompatibilities and restore missing dependencies.

Applications built with C++Builder

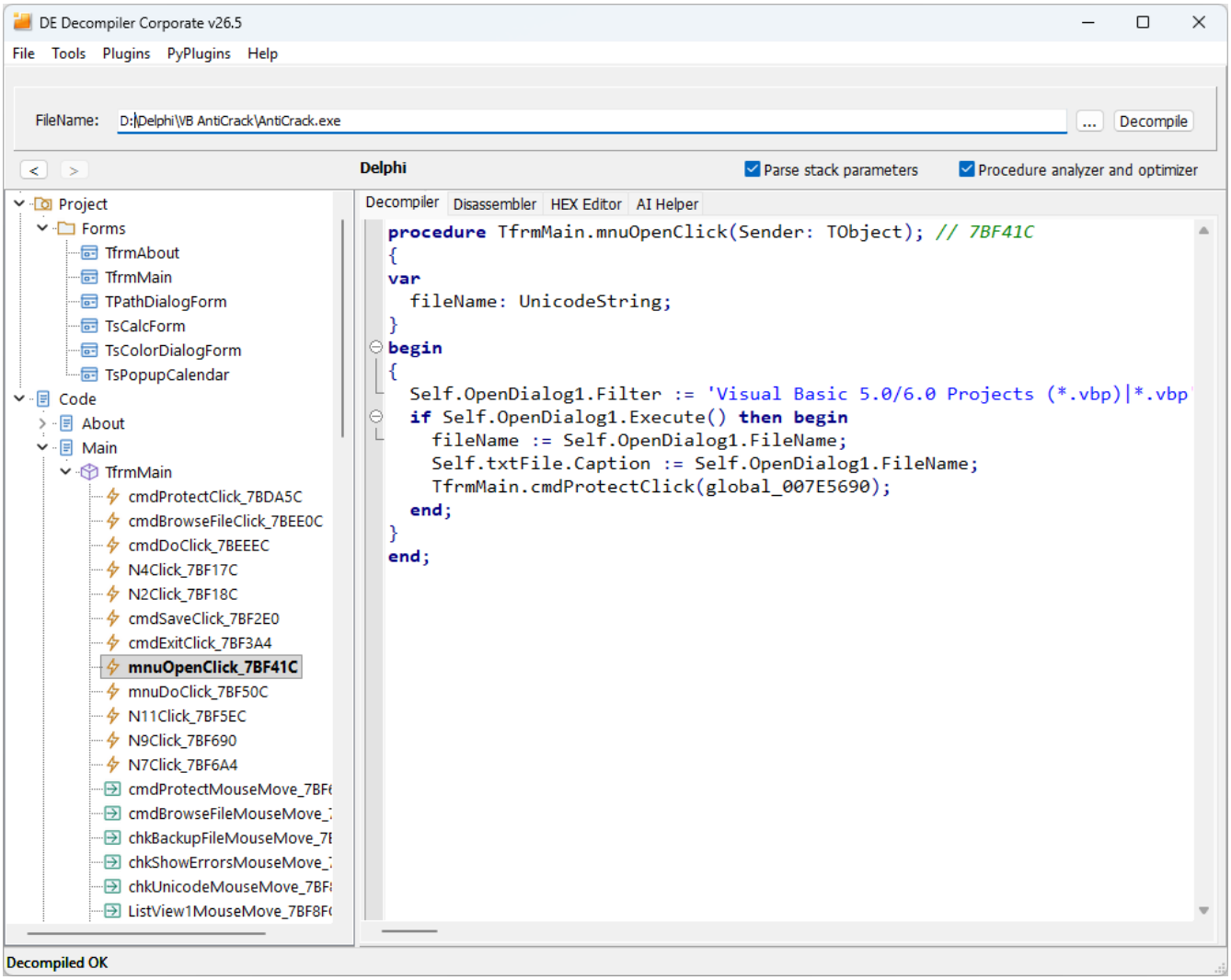
DE Decompiler supports VCL applications built with C++Builder for Win32 and Win64. Recover and visualize forms, examine units, classes and functions, follow event handlers and use the other analysis and export tools in the same workspace.

C++Builder-specific library RTTI knowledge bases are not yet available. Functions outside the shared VCL coverage, and object properties or methods that cannot be identified, may be shown without their library names. Code decompilation quality for C++Builder is therefore noticeably lower than for applications written in Delphi. This limitation concerns code-reconstruction quality and library-name recognition. The other supported features remain available for C++Builder under the same edition rules as for applications written in Delphi.

C++Builder code is reconstructed as Delphi-like Pascal, with PAS/DFM output. It is not converted back into the original C++ source. As with other native-code reconstruction, review and correct inferred types, generated names and decompilation inaccuracies manually before using the recovered material in an IDE.

Editions

High-level code decompilation, AI Helper, the HEX Editor and obfuscation require a Single Developer or Enterprise license. Free Lite recovers VCL form information and provides native disassembly and String references. See Licensing.



Decompiled Delphi-like code with the recovered project structure.

1.2 Quick start

Open a supported Win32 or Win64 VCL application written in Delphi or built with C++Builder. Inspect the recovered project structure and compare code with native instructions. High-level code decompilation, AI Helper and the HEX Editor require a commercial license; Free Lite provides form inspection, native disassembly and string references.

1. Select **File > Open program** and choose the binary to examine. Let DE Decompiler finish its initial processing.
2. Expand **Code** in the project tree. Select a unit to examine its recovered declarations, then select a class or procedure.
3. Compare **Decompiler** and **Disassembler**. Use **HEX Editor** when an address or instruction needs to be checked against the file bytes.
4. Expand **Forms**, select a recovered form, and open **Designer (read only)**. Inspect its controls and properties, and use recovered event entries to open the corresponding handler code.
5. Use **Ctrl+T** to find a procedure, **Ctrl+F** to search displayed text, and **Tools > String references** to investigate recognized string literals.
6. Save an analysis database through **File > Data Base** when you need to return to the analysis later.

Saving recovered material

Use **File > Save decompiled project** to export a recovered Delphi project, or **File > Save all in one module** for a combined `.pas` listing. Before opening recovered code in an IDE, manually correct decompilation inaccuracies and incompatibilities and provide missing dependencies. Keep an unmodified copy of the analyzed binary with your investigation material.

Choosing a processing mode

Fast decompilation processes individual procedures when you open them. Enable **Analyze prototypes** for preliminary analysis of procedure signatures. Before searching all high-level text, disable **Fast decompilation** and process the file again. Project and combined-listing exports process procedures as needed; review their completion messages. See Processing modes and caching.

A useful first check

Start with a known form and one event-related method or public function. Check the recovered declarations, follow the code, and compare a few statements with disassembly. This establishes how much useful structure DE Decompiler has recognized in that particular build.

1.3 Compiler compatibility and RTTI

DE Decompiler natively recognizes Delphi compiler versions 3 through 13 and supports VCL applications built with C++Builder for Win32 and Win64.

Compiler versions and recovered types

Delphi releases differ in executable structures and RTTI formats. Version-specific type knowledge lets DE Decompiler interpret these differences. Recognizing the compiler generation does not guarantee identical reconstruction quality for every type, library or procedure.

Check a recognized class against its declarations, form properties, and method code. If a name, field type, signature, or layout appears inconsistent, use native instructions and file bytes to investigate before relying on the high-level representation.

Compiler identification and signature databases

C++Builder and Free Pascal inputs are now identified separately from Delphi. Compiler identification remains independent of the compatible signature database, including after loading a saved project. Long compiler names and version ranges fit the main-window label; full details remain available in its tooltip.

The compiler label identifies the input independently of the signature database selected for code analysis. If the available evidence cannot distinguish compiler versions, the label retains that ambiguity.

In the commercial editions, DE Decompiler selects a matching database from the `de` directory: an exact compiler version first, then the nearest newer version, then the nearest older version of the same bitness. The files follow the `vcl<version>.bin` and `vcl<version> x64.bin` naming patterns.

An asterisk (*) marks a substituted database or a database match that could not be established with confidence. Hover over the compiler label to see the selected BIN filename. The marker describes the database match; it does not by itself mean that the input cannot be analyzed. Signature databases are a commercial-edition feature.

Applications built with C++Builder

DE Decompiler supports VCL applications built with C++Builder for Win32 and Win64. Recover and visualize forms, examine units, classes and functions, follow event handlers and use the other analysis and export tools in the same workspace.

C++Builder-specific library RTTI knowledge bases are not yet available. Functions outside the shared VCL coverage, and object properties or methods that cannot be identified, may be shown without their library names. Code decompilation quality for C++Builder is therefore noticeably lower than for applications written in Delphi. This limitation concerns code-reconstruction quality and library-name recognition. The other supported features remain available for C++Builder under the same edition rules as for applications written in Delphi.

C++Builder code is reconstructed as Delphi-like Pascal, with PAS/DFM output. It is not converted back into the original C++ source. As with other native-code reconstruction, review and correct inferred types, generated names and decompilation inaccuracies manually before using the recovered material in an IDE.

Private functions

The decompiler attempts to identify private functions within each unit heuristically. Such discovery helps expose code beyond recognized public entry points, but it does not establish that every function boundary or unit association has been recovered.

VCL scope

The visual form workflow applies to VCL forms and controls recovered from Win32 and Win64 input files.

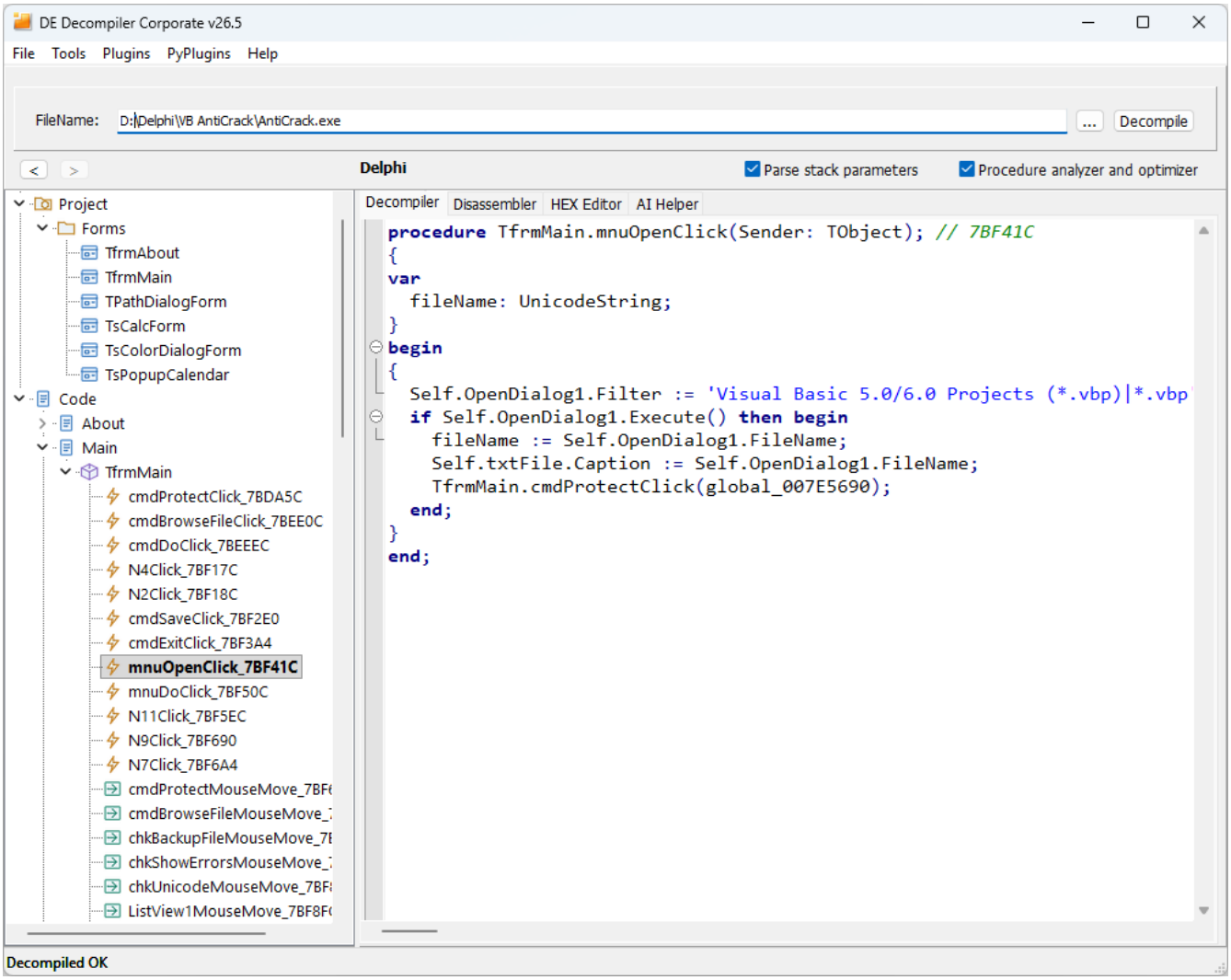
Reporting a compatibility issue

DE Decompiler is continually developed, and decompilation quality improves as compiler support, type recognition and code reconstruction are refined.

If you would like to report a specific issue, send the binary file that reproduces the problem and identify the form or function where it occurs. Include its name and, for a function, its address when available, together with the DE Decompiler build and a description of the error. See Technical support for contact details.

1.4 The main window

The project tree connects recovered forms, units, classes and procedures with the code, declaration and binary views.



The DE Decompiler workspace in the light interface theme.

The project tree

Forms contains recovered form objects. **Code** is grouped as **Unit > Class > Procedure**. Units and classes remain collapsed until opened; expanding a unit with one visible class also reveals that class's procedures. Selecting a unit displays its reconstructed declarations; selecting a procedure opens the corresponding code views.

Views and processing controls

- **Decompiler** displays reconstructed high-level statements.

- **Disassembler** displays native instructions and addresses.
- **HEX Editor** displays the underlying file bytes.
- **Designer (read only)** displays a recovered VCL form and its available properties.
- **DFM** displays the recovered DFM source, with syntax highlighting, folding, search and copy.
- **AI Helper** processes an existing decompiled listing with a selected model or API service.

Parse stack parameters and **Procedure analyzer and optimizer** affect reconstruction of the current listing. When output is unclear, compare it with these processing options disabled and inspect the corresponding disassembly. With cached or fully processed output, process the file again after changing a setting.

Navigation

The arrow buttons above the tree move through viewing history. **Esc** and the mouse Back and Forward buttons navigate the same history across code, declarations, form previews and source views. Returning to a form restores its selected control, page and scrolling; returning to code restores the caret and view position. **Alt+1** switches focus between the tree and the active view, and **Enter** opens the selected tree item. In the HEX Editor, **Tab** moves between the hexadecimal and text fields and then to the next control.

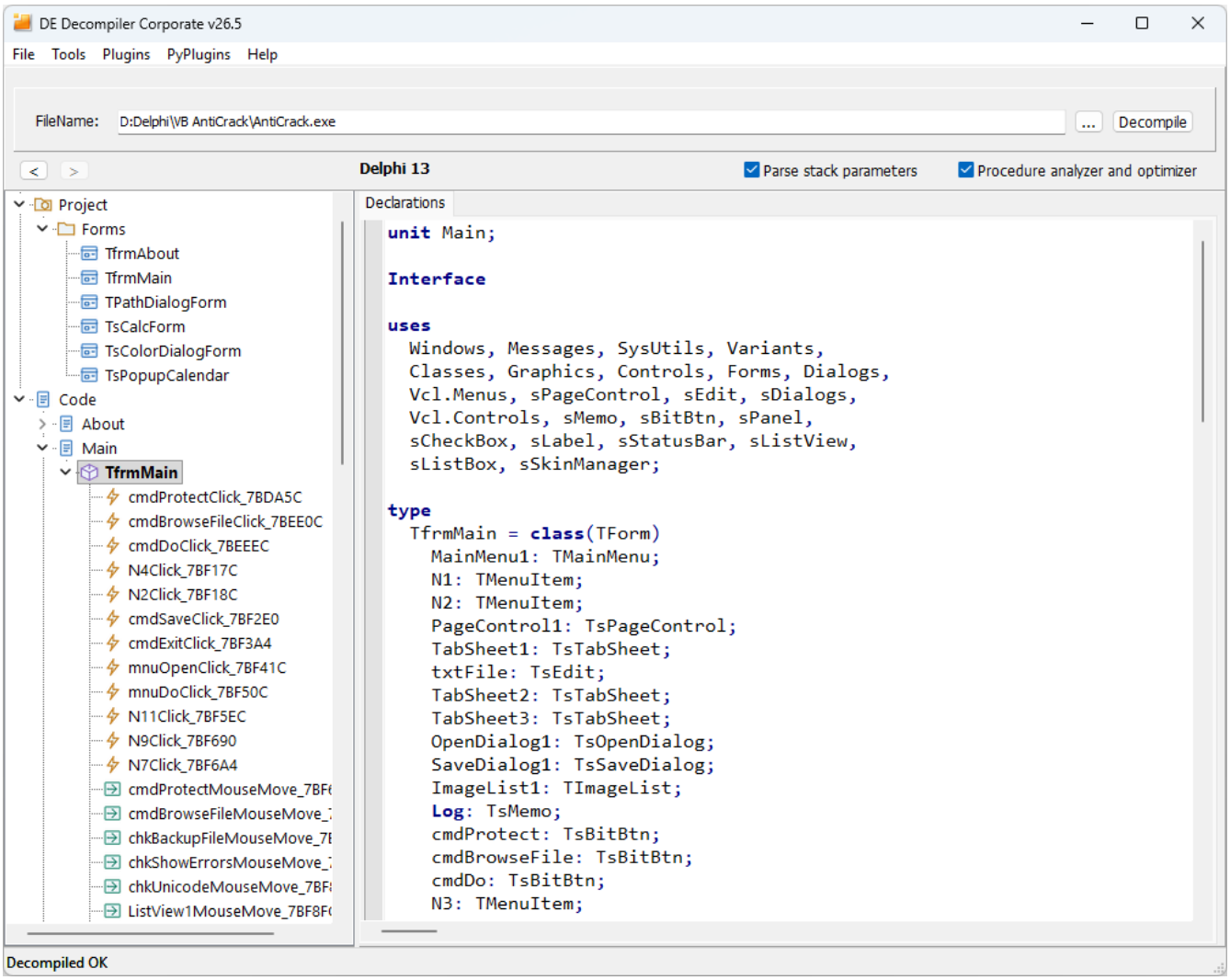
Window style as in Visual Studio places the code on the left and the project tree on the right. The available tabs depend on the selected object. See Viewing and comparing code for context-sensitive shortcuts.

1.5 Units, classes and functions

Recovered declarations provide a starting point for examining the structure of supported VCL applications written in Delphi or built with C++Builder.

Unit declarations

Select a unit under **Code** to view its reconstructed declaration listing. Use the unit name, available references, declared classes and visible members to establish which part of the application you are examining.



Recovered unit and class declarations for a Delphi application.

An excerpt from the recovered class declaration:

```
type
  TfrmMain = class(TForm)
    MainMenu1: TMainMenu;
    N1: TMenuItem;
    N2: TMenuItem;
    PageControl1: TsPageControl;
```

Classes and methods

Expand a recognized class and select a method or function. Read the signature and body together: a plausible body does not by itself confirm an inferred parameter type or class-field interpretation.

Public and private functions

Recognized public functions and methods appear in the recovered structure. The decompiler also attempts to locate private functions in each unit. Private discovery is heuristic; investigate unexpected boundaries, missing arguments, or inconsistent unit associations using disassembly.

Compiler-generated anonymous-method frames are hidden from the normal tree. Search results, address jumps and history can reveal a required frame. **Show VCL classes** provides the complete class view. Opening a unit or class declaration does not require decoding all its methods.

Names and analyst annotations

Distinguish names preserved in the binary from generated identifiers and names assigned during analysis. A reconstructed or analyst-assigned name may help explain a procedure, but it does not prove that this was the original source identifier.

2 Opening and saving

2.1 Opening a file

Select **File > Open program** to open the binary of a supported Win32 or Win64 VCL application written in Delphi or built with C++Builder.

VCL applications built with C++Builder use the same file-opening workflow. The compiler label distinguishes C++Builder and Free Pascal from Delphi; its tooltip shows full compiler and signature-database details.

1 Choose **File > Open program** and select the input file.

2 Wait for the project tree to populate and initial processing to finish.

3 Check the recovered code structure and forms before proceeding to individual procedures.

Drag and drop and command-line opening

You can drag a binary from Explorer onto DE Decompiler or pass its path on the command line. Recognition depends on the file contents, so an unfamiliar filename extension or a missing extension does not by itself prevent analysis. Supported Windows shortcuts can be resolved to their referenced files.

Saved analysis databases can also be opened by drag and drop or by passing their path on the command line. Keep paths and file versions unambiguous when working with several builds.

Cancelling initial analysis

Cancellation is more responsive while opening large applications, discovering classes, reading forms and decompiling all procedures. A pending cancellation stops further procedure processing; cancelling full decompilation still discards its incomplete result.

Cancelling optional prototype analysis keeps the parsed file open and retains the prototypes, APIs and strings already found. Incomplete analysis reports are omitted, and decoding options are restored after interrupted analysis.

Inputs with limited recognition

The available instructions and metadata determine the detail of recovered names, types and calls. Compiler variation, unusual type information, protection and file damage can affect recognition; inspect an unexpected result in the native instruction view. See Compiler compatibility and RTTI and Interpreting recovered code.

For repeatable analysis, preserve the exact binary and record the settings used. A later build with the same filename can have different addresses, layout and behavior.

2.2 Analysis databases

An analysis database saves recovered project information so that you can return to an investigation without starting from the binary each time.

Use **File > Data Base** to save or open analysis data. The compressed VDD database stores recovered structure, forms and resources, analyst-assigned names, method metadata and cached listings for later review. Saving fully processed output is useful when initial analysis of a large application takes a long time.

You can also open a saved database through the file-opening workflow, drag it onto the window or pass its path on the command line. Keep the corresponding original binary alongside the database when transferring investigation material between workstations.

Reviewing a database without the executable

Previously cached native code remains available when the original executable is absent. Saved form resources and recovered structure can also be inspected. A procedure that was not decoded before saving still needs the matching original image; DE Decompiler does not decode it against a missing or changed executable.

Review or export the available cached material, and check any messages identifying undecoded procedures. For a portable investigation snapshot, process the required procedures before saving and keep the exact input binary with the database whenever possible.

Saving and compatibility

Database saving supports cancellation and replaces an existing VDD only after a complete write. If an individual procedure cannot be decoded, the database can still retain the other analysis data; review the warnings reported after saving.

Newer databases can contain extended model and resource data. An older build may display only a compatibility subset. Use the current DE Decompiler build to inspect all saved data.

What to preserve

- The analysis database and the exact binary it describes.
- The DE Decompiler build and relevant processing settings.
- Any separate notes, analyst-assigned names and exported material used in the investigation.

A database is an analysis artifact. It is distinct from the recovered source files and does not replace the original executable as the source of addresses and bytes.

Checking a saved analysis

After opening a database on another workstation, inspect a known unit, method and form. When comparing results from different decompiler builds, retain the earlier database and record which build produced each result.

2.3 Saving recovered material

Save the output needed for the investigation while retaining the original binary and the analysis context.

Project and module output

File > Save decompiled project exports the recovered material as a Delphi project, including recovered units and forms. Choose an output folder and review the saved files:

- `.pas`: one UTF-8 source file for each unit, combining declarations and methods from its recovered classes.
- `.dfm`: separate form definitions, including distinct files when several forms belong to one unit. Non-ASCII string literals use Delphi # character codes while binary property data is preserved.
- `Project.dpr`: project references and recovered startup information based on units and form resources successfully written during the current export.

The primary form is linked through `{$R *.dfm}`; additional forms use explicit resource directives. Form-only analyses export available DFM files for inspection without inventing missing Pascal classes or startup code. Review export diagnostics for missing or conflicting resources.

Use **File > Save all in one module** when a single combined `.pas` listing is more convenient for review or comparison. A combined listing and a multi-file Delphi project serve different purposes; choose the output required by the investigation.

Commented procedure bodies

By default, recovered local declarations and statements are enclosed in a Pascal comment wrapper while the method signature is retained. The body remains available for inspection, search and analysis. This presentation does not mean that the exported unit is ready to compile.

Progress, cancellation and failures

Project and combined-listing exports process procedures as required. Review warnings for methods that could not be decoded. Saved cached code can be exported without the original executable; undecoded procedures require the matching binary.

Use **Cancel** to stop between procedures. Existing PAS, DFM, DPR and combined-listing files are replaced only after complete writes. A cancelled multi-file project export retains files completed before cancellation, so its output folder may contain a mixture of completed files and older files. Use a separate output folder when preserving an earlier export.

Preparing material for an IDE

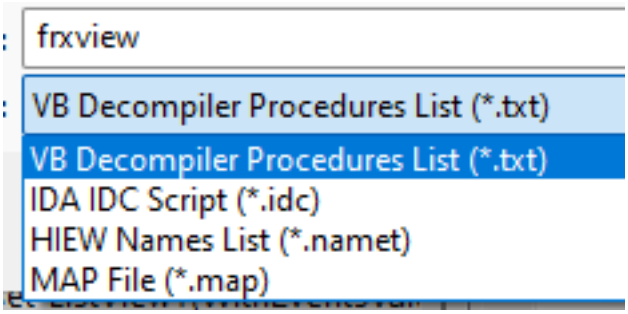
Before opening recovered code in an IDE, manually resolve decompilation inaccuracies and incompatibilities. Reconcile generated identifiers, inferred types and signatures, unidentified calls and missing dependencies. The listing reconstructs program logic for analysis; it does not reproduce a compilable source project. A build requires manual analysis and correction of the exported code.

Investigation records

For a security review, retain the procedure address and binary identity with exported fragments. Keep decompiled statements, native instructions, model-generated suggestions and analyst conclusions distinguishable in the resulting report.

2.4 Saving the procedure list

File > Save procedures list exports recognized procedure names and addresses for use in other analysis workflows.



Procedure-list export formats.

Select the required file type in the save dialog:

Format	Use
`TXT`	A plain-text procedure list for review, comparison and address/name records.
`IDC`	An IDA script for transferring recognized function names and addresses.
`NAMET`	A name list for use with HIEW.
`MAP`	A map file for compatible debuggers, disassemblers and symbol-import workflows.

All formats retain full Win64 virtual addresses. IDC export uses rebased relative addresses and checks bitness; it preserves existing IDA instructions and data. Saving can be cancelled and uses complete-file replacement, preserving an existing list if writing fails.

An exported list describes recognized procedures. Private-function discovery is heuristic, so absence from the list does not establish that an address cannot contain executable code. Preserve the original input and its address context with the exported list.

3 Examining an application

3.1 Viewing and comparing code

Use reconstructed Delphi-like Pascal statements to understand a procedure, and native disassembly to check how a result was obtained. This workflow applies to supported input from both Delphi and C++Builder.

Following the listing

Select a procedure in the tree. Inspect the signature, local declarations and control flow before following individual expressions. Generated names identify recovered values where an original source name is unavailable. Syntax highlighting and collapsible blocks help navigate longer procedures.

Recovered bodies may be enclosed in Pascal comment wrappers for export. DE Decompiler still treats those bodies as code for identifier highlighting, reference search and renaming. String literals and actual line comments remain excluded from code-reference matches.

Identifiers and code navigation

Place the caret on a register, variable or function name to highlight matching occurrences in the current listing. Highlighting a local or global identifier in one procedure is separate from a project-wide reference search.

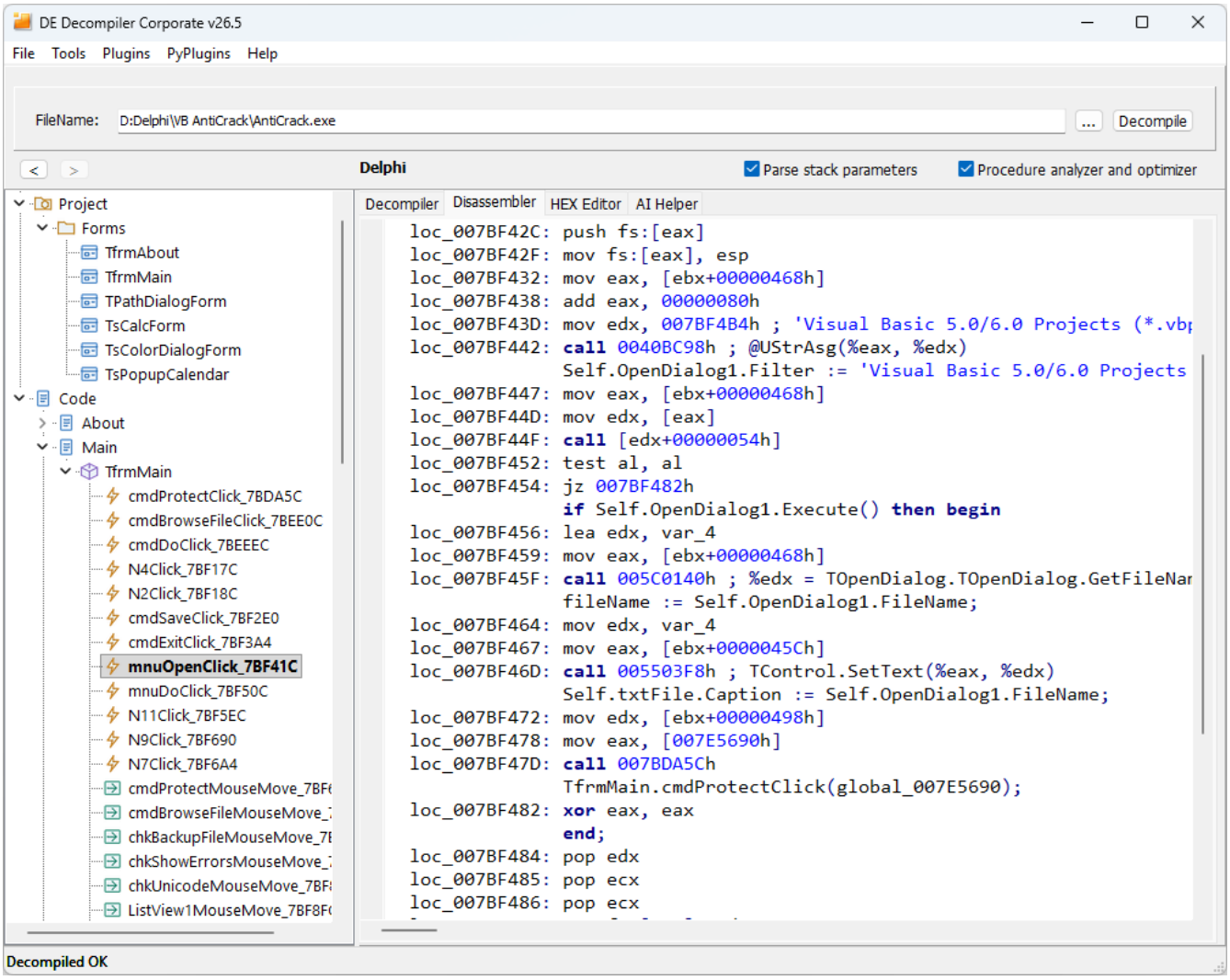
Double-click a recognized procedure call to open the called procedure. Address-like constants can lead to the corresponding file bytes in the HEX Editor. Use **Esc**, the history arrows or mouse Back and Forward buttons to return.

Synchronizing addresses

Switching between Decompiler and Disassembler follows the current caret address, including full Win64 addresses. If there is no identical instruction label, the nearest available address is selected. Only folds containing the destination are opened.

Mixed high-level and native views

Right-click the code and select **Copy to disassembler**, or press **D** in the code view. The resulting view places address-labelled reconstructed statements beside the matching native instructions. Folded source is included without expanding the original editor, and unaddressed declarations are omitted from the annotations. Keep **Add Addresses** enabled when using address-based comparison.



Reconstructed Delphi statements interleaved with native instructions.

References and names

Use **Find references** on a procedure selected in the tree, or press **X**, to examine recognized callers. Double-click a result to inspect the caller. Reference searches require full processing: disable **Fast decompilation** and reopen the input if prompted. A reference list based on undecoded cached bodies is incomplete. In the code view, place the caret on a global variable and press **X** to search for its references. Indirect or dynamically resolved uses may not appear in the recognized-reference list.

Use **Rename variable** or **N** to assign a meaningful analysis name. Names follow Pascal identifier rules. Rename matching is case-insensitive and checks collisions with parameters, local declarations and reserved keywords. **Undo renaming** or **Ctrl+Shift+Z** reverses a variable rename independently of ordinary text-editing undo. Keep analyst-assigned names distinct from original names preserved in the binary.

Keyboard access

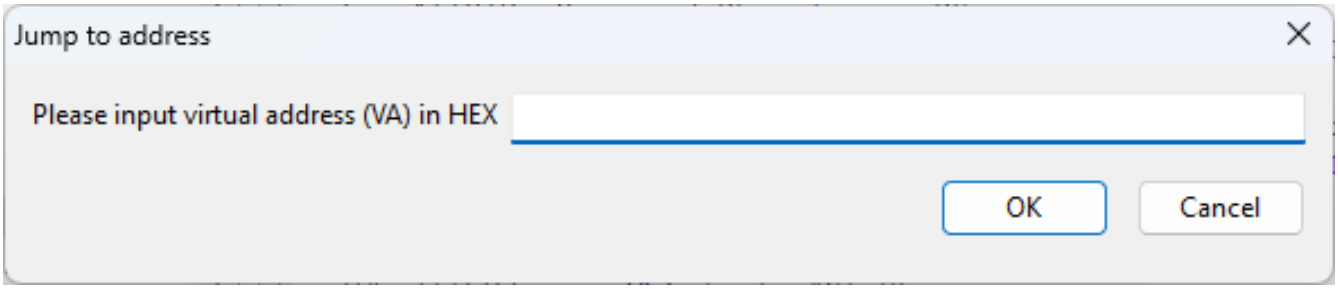
Focus	Shortcut	Action
Code	D	Combine the reconstructed listing with disassembly.
Code	N	Rename the selected variable.
Code	Ctrl+Shift+Z	Undo variable renaming.
Code	X	Find references to the global variable under the caret.
Tree	X	Find references to the selected procedure.
Tree	G	Navigate using a section path.
Tree	Ctrl+C	Copy the selected object's full tree path.
Tree	Space or Insert	Mark or unmark the selected item.
Views	Alt+1	Switch focus between the tree and active view.
History	Esc	Return to the previous procedure.

Single-letter shortcuts depend on keyboard focus. They do not replace ordinary typing in input fields.

3.2 Navigating by address and tree path

Tools > Jump to virtual address opens a location by its virtual address.

Use **Ctrl+G** or select **Tools > Jump to virtual address**, enter the required address and confirm. Addresses refer to the currently analyzed build. File offsets and virtual addresses are different coordinate systems; keep that distinction when comparing a listing with bytes in the file.



Entering a virtual address for navigation.

Sharing a tree location

In the project tree, use **Copy the name of section** or **Ctrl+C** to copy the selected object's path. Use **Jump to the section** or **G** to navigate with a copied path in the corresponding analysis.

You can also select an address in the listing before invoking **Ctrl+G**. Use viewing history to return after following a reference; mouse Back and Forward buttons retain the position within a procedure. A path or address shared with another analyst is meaningful only in the context of the same binary and compatible recovered structure.

3.3 Finding procedures and text

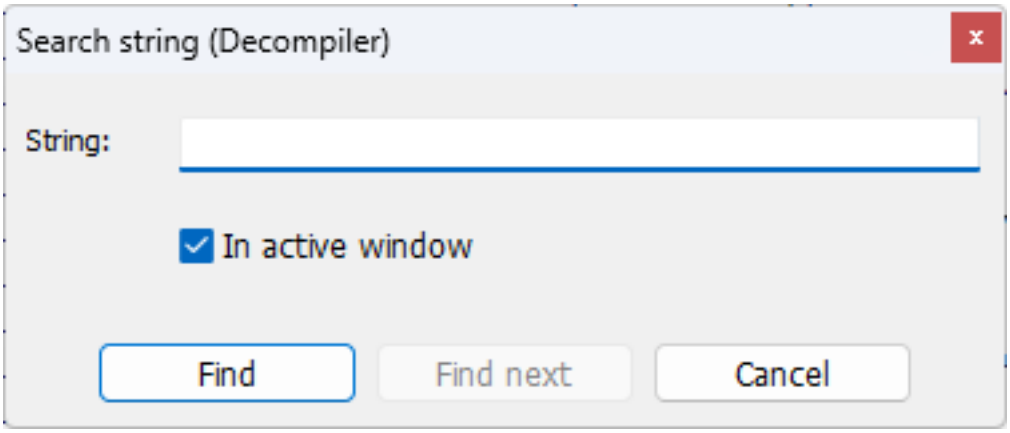
Procedure-name search and listing-text search serve different purposes.

Find a procedure

Use **Tools > Find procedure** or **Ctrl+T** and enter a complete name or a distinguishing part. Matching is case-insensitive and accepts qualified procedure names. Select the required match from the results.

Find displayed text

Use **Tools > Find string** or **Ctrl+F** to open text search. With **In active window** selected, the search applies to the current code or form-data view. Clear this checkbox to search the decompiled project. Enter the text in **String**, then use **Find** and **Find next** to move through matches.



Text search in the active code view.

Whole-project high-level text search requires the corresponding procedures to be decompiled. Disable **Fast decompilation** and process the file again when you need complete coverage for high-level text search.

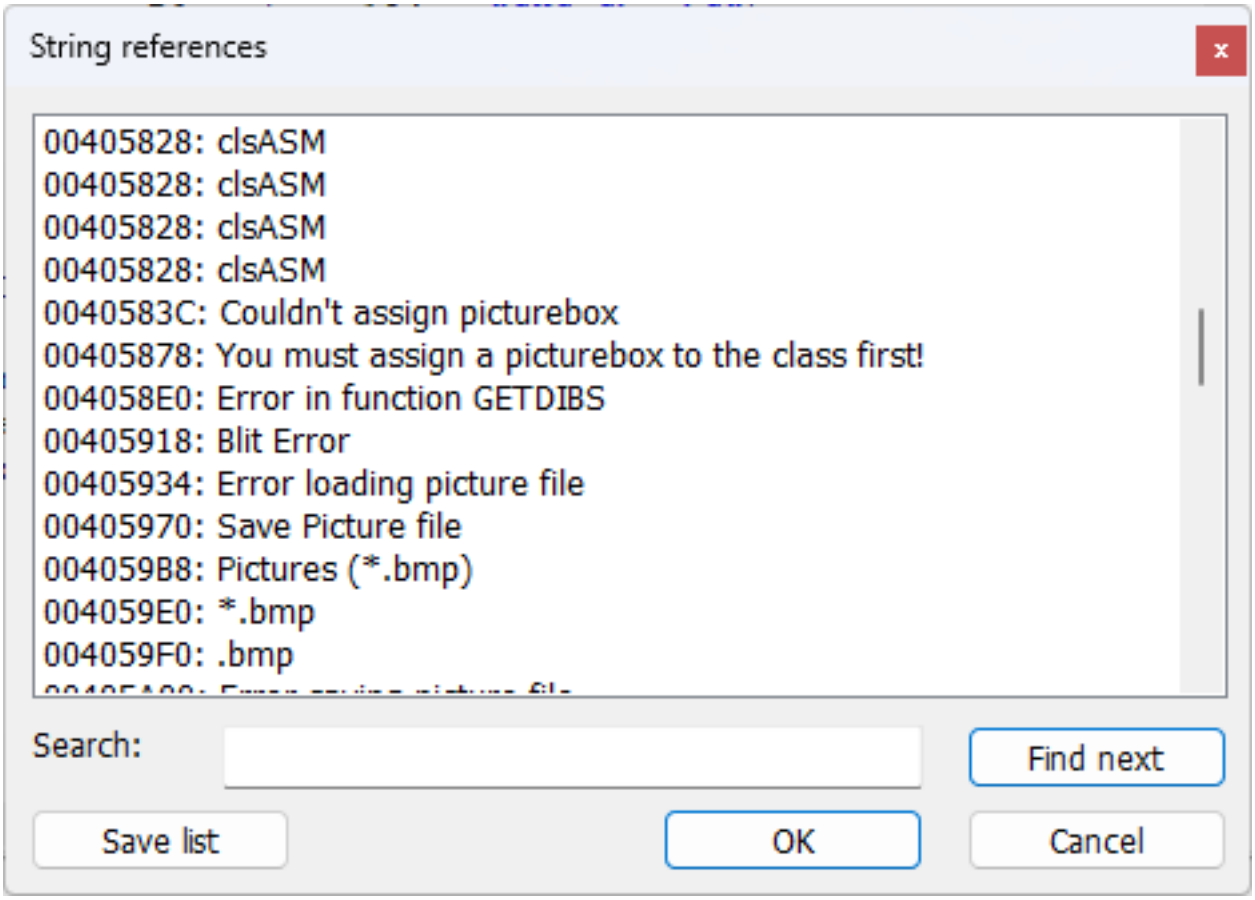
Text search includes folded content in the Decompiler, Disassembler, DFM source and AI views. A found match is revealed while unrelated folds remain closed.

String literals

Use String references to investigate recognized string literals and their locations. These references can be collected by preliminary disassembly in Fast decompilation mode; they do not require every procedure to have a complete high-level listing.

3.4 String references

Tools > String references helps locate recognized literals and the procedures associated with them. String-reference browsing is available in Free Lite and commercial editions.



The string-reference browser.

Use **Search** and **Find next** to locate an interesting literal. Double-click a string to open a procedure that references it; double-click again to continue finding its uses. **OK** navigates to the selected string and closes the dialog; **Cancel** closes the dialog.

The list contains strings collected during disassembly. In Fast decompilation mode, preliminary analysis can supply these references when **Analyze prototypes** is enabled. If DE Decompiler reports that a full pass is required, disable **Fast decompilation** and reopen the input. In Lite, selecting a reference opens the native disassembly. Inspect referring procedures and the corresponding bytes to determine how a value is used. Paths, registry-related strings, network addresses and command fragments can help prioritize code review.

A reference is an analysis lead. The presence of a string in a procedure does not establish that it is passed to a particular API or that the relevant branch executes. Encrypted, dynamically constructed or otherwise unrecognized strings can be absent.

Editing a recognized string

Right-click a supported string and select **Edit string**. The replacement must not be longer than the original string because the file does not reserve space for a larger value. Confirming the edit modifies the file directly; this operation does not create a backup and cannot be undone through the string-edit dialog. Work on a copy and preserve the unmodified sample.

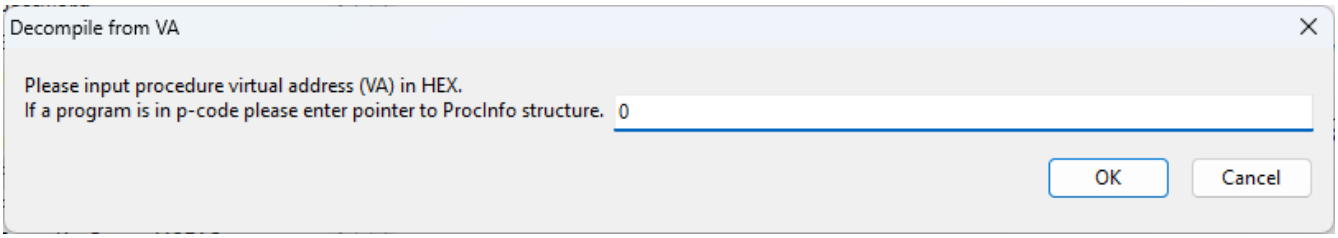
In commercial editions, the context menu also provides **HEX Editor** for inspecting the string's bytes. The HEX Editor offers undo and redo for its own edits. Check the resulting bytes and affected code after a change.

3.5 Decompilation from an address

Tools > Decompile from VA requests decompilation from a specified virtual address.

This workflow is useful when investigating code not represented as the expected procedure in the recovered tree. If a file is already open, enter the virtual address to process. If no file is open, DE Decompiler prompts you to select one and opens it in simplified mode.

Inspect the resulting native instructions and reconstructed listing together. Without normal initial processing, some imported-call and type information may be unavailable.

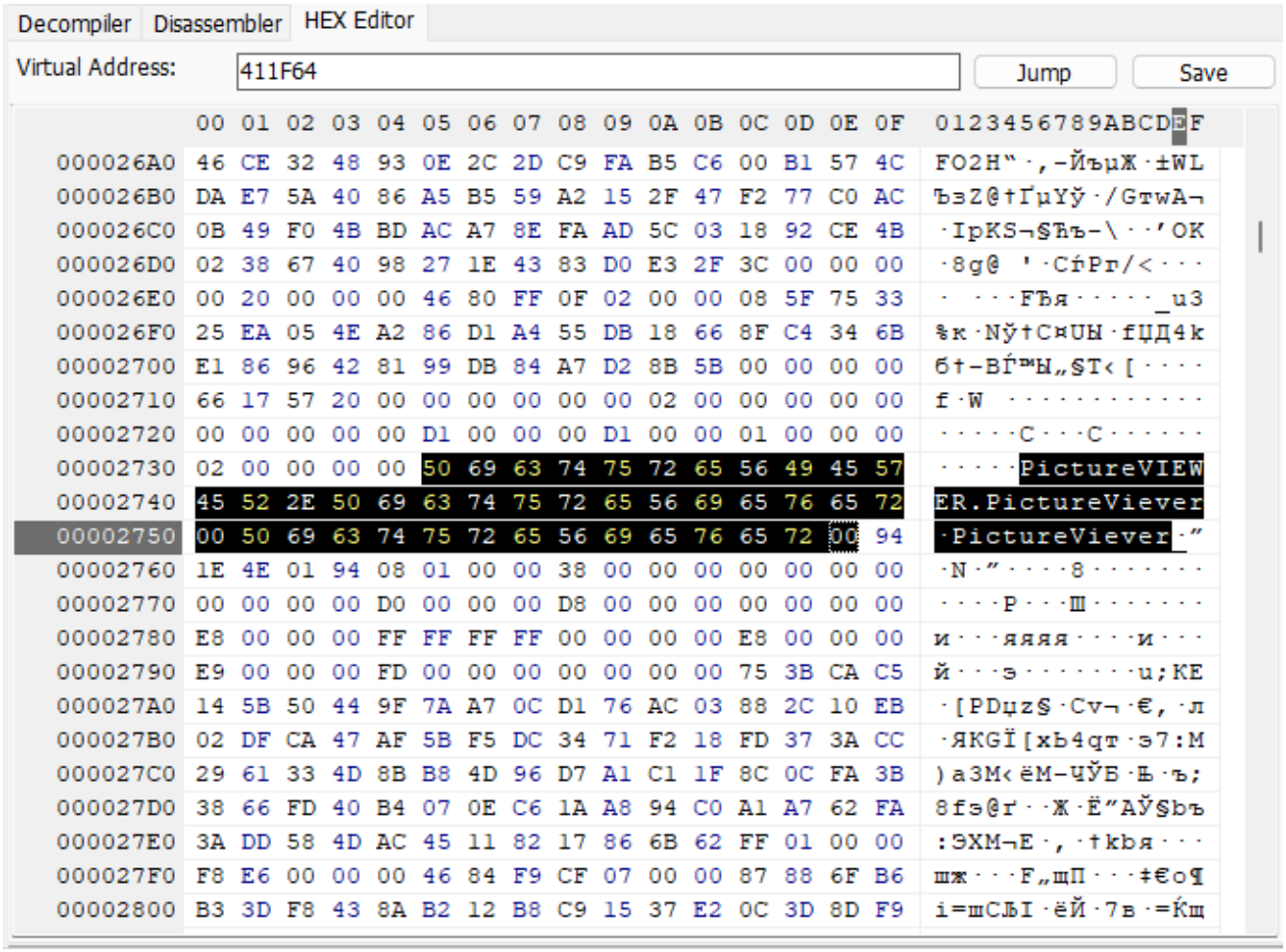


The decompilation-from-address dialog.

Choosing an address does not establish a valid native function boundary or signature. An incorrect entry point can produce misleading statements, argument interpretations or incomplete output. Check surrounding instructions and recognized callers before relying on the result.

3.6 Inspecting and patching bytes

Use the HEX Editor tab to inspect and edit the file bytes underlying the recovered code and form data.



Binary inspection in the HEX Editor.

Inspecting file data

Open **HEX Editor** alongside the code or form view. Navigate using a virtual address, including **Image Base**, or a file offset measured from the start of the file. Compare the displayed bytes with the corresponding native instructions or recovered form data.

Editing bytes

The HEX Editor supports editing individual bytes and blocks of data within the file. Check whether the location is expressed as a virtual address or a file offset before making a change.

Work from a separate analysis copy. Reopen or reprocess affected code as needed after a change, especially when cached listings are enabled. Preserve both the original bytes and the change record when comparing behavior or documenting an investigation.

3.7 VCL forms and visual inspection

DE Decompiler decompiles VCL form information and visualizes recovered controls and properties.

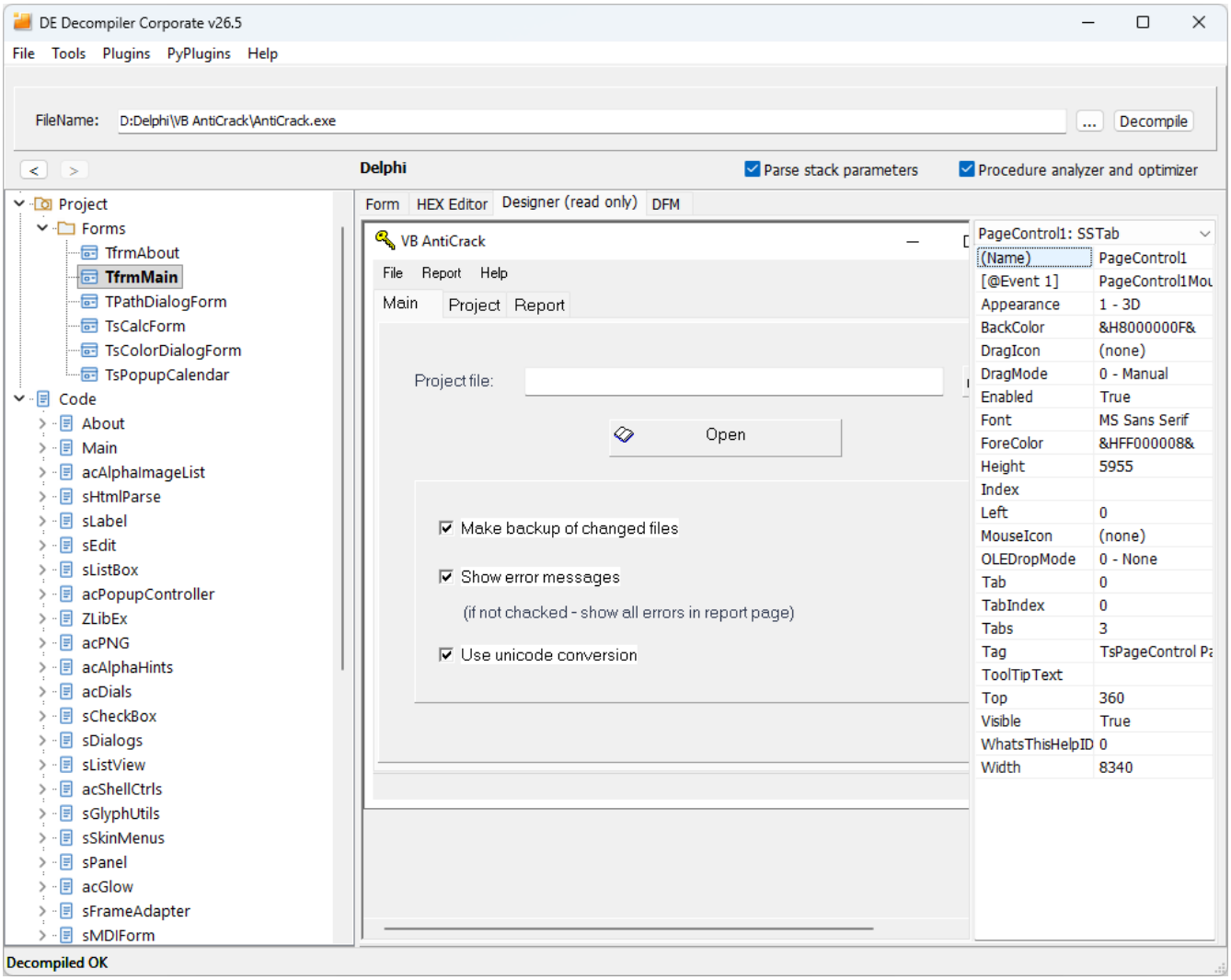
The same VCL form inspection and event-navigation workflow is available for applications built with C++Builder.

1 Expand **Forms** in the project tree and select the form to examine.

2 Open **Designer (read only)** to inspect the recovered visual layout.

3 Select a control and examine its available properties and recovered event entries in the property area.

4 Compare the form representation with the available **DFM**, form-data and HEX views.



A reconstructed VCL form with its recovered component properties.

Use FRX Image Viewer to preview and export registered form pictures and recognized native ImageList icons.

DFM source and the visual preview

The **DFM** source view retains recovered component class names, properties and event-handler identifiers. It supports syntax highlighting, matching-identifier highlighting, search and copy. Fold `object`, `inherited` and `inline` blocks, collection items, binary properties and string lists to navigate large forms.

The **Form** tab contains the converted representation used for the visual preview. Inspect the recovered DFM source for the stored property values and original component class names.

The preview reconstructs saved pages, menus, lists, grid geometry and toolbar buttons, including supported local ImageList graphics, masks and alpha transparency. Supported AlphaSkins controls are mapped to their VCL

equivalents for display while their DFM class names remain available in the source. Drawing and event code from the analyzed application is not executed.

Following event handlers

Select a recovered control or menu event in the form view to open its handler. Returning through viewing history restores the selected control, active page and scroll position. Inspect the procedure in **Decompiler** and compare it with **Disassembler** when investigating the event behavior.

Form data and offsets

The **Show offsets** option adds location information to the form-data presentation. Use it when relating a recovered property or definition to its binary representation.

Interpreting the preview

The designer is an inspection view of recovered form data. It is not the original application executing its initialization or event code. Runtime-created controls and values assigned only by code require investigation in the corresponding procedures.

Custom controls and version-specific type recognition can affect how a form is represented. Inspect the recovered data and code when a component, property or layout appears incomplete.

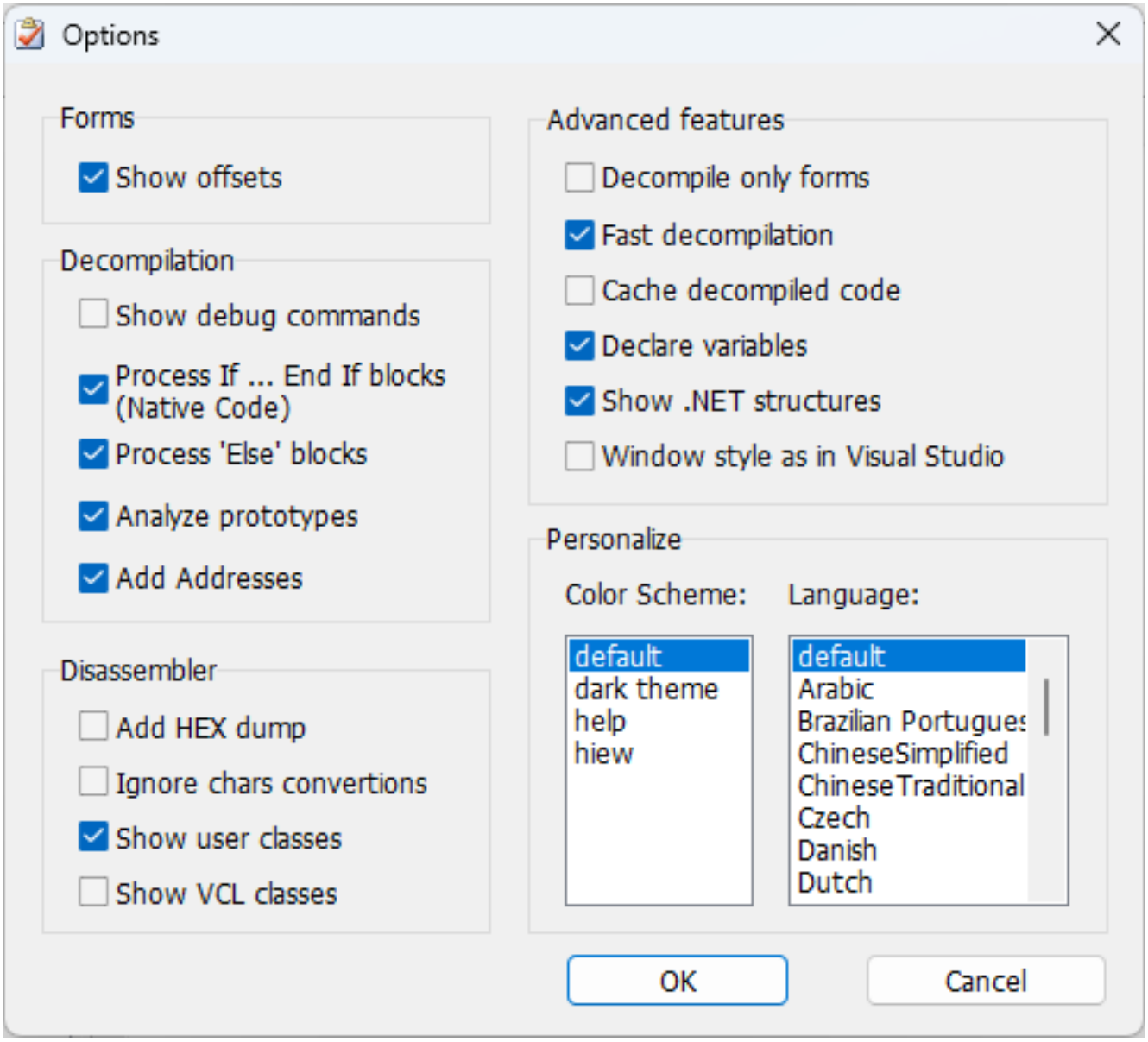
TMS controls and static previews

The visual preview supports recognized TMS labels, buttons, lists, group boxes, sliders, scrollbars and tab strips, using their stored text and layout. Activity indicators have static previews; embedded web areas are marked without loading their pages.

4 Options and tools

4.1 Program options

Tools > Options controls reconstruction, presentation, caching and the interface language.



The DE Decompiler Options dialog, including user-class and VCL-class settings.

The following sections describe analysis options for supported VCL applications written in Delphi or built with C++Builder:

- Forms and visual inspection
- Decompilation processing
- Disassembly display
- Themes and fonts
- Processing modes and caching
- Interface language

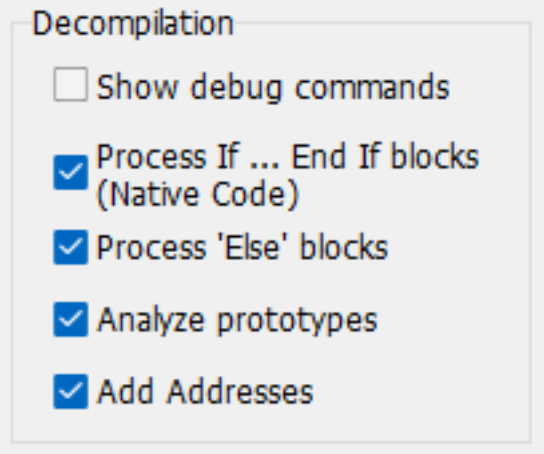
When comparing processing settings, retain the input binary and record which settings produced each listing. Cached output may require reprocessing before it reflects a changed option.

4.2 Decompilation processing

Native-code decompilation reconstructs Delphi-like Pascal from machine instructions and available metadata. The same processing options apply to supported applications written in Delphi or built with C++Builder.

Reconstructed program logic

The decompiler recovers conditions, `case` blocks, loops and calls, including recognized VCL methods. Source-level names and types depend on the information retained in the binary. Generated variable names, inferred structure layouts and calls without recognized VCL names are normal consequences of native-code reconstruction. See Interpreting recovered code for details.



Controls for decompilation processing.

Delphi expressions and runtime operations

Recognized control flow includes `if/else`, `case`, `for`, `while` and `repeat/until`, with `Break` and `Continue` where the native branches establish their meaning.

Available RTTI, class metadata and signatures support object fields, properties, virtual methods, VCL calls and imported Windows API calls. Common RTL operations can be represented as Delphi string expressions, `Length`, `Copy`, `Delete`, `Insert`, array resizing and indexing, Variant-array access, and `TextFile` input operations.

These transformations depend on recognized types, calling conventions and data flow. Where the evidence is insufficient, the listing retains explicit calls, pointers or branches. Generated semantic variable names describe inferred roles and are not recovered original identifiers.

Stack parameters and optimization

Parse stack parameters contributes to reconstruction of calls and higher-level expressions. **Procedure analyzer and optimizer** simplifies recognized expression and control-flow patterns. Compare the resulting listing with these controls disabled when an expression or branch is unclear.

Control-flow processing

Process If ... End If blocks (Native Code) reconstructs recognized conditional jumps as high-level conditional blocks. **Process 'Else' blocks** adds heuristics for Else branches and requires If-block processing. Compare these controls enabled and disabled when investigating an ambiguous branch.

Prototypes and addresses

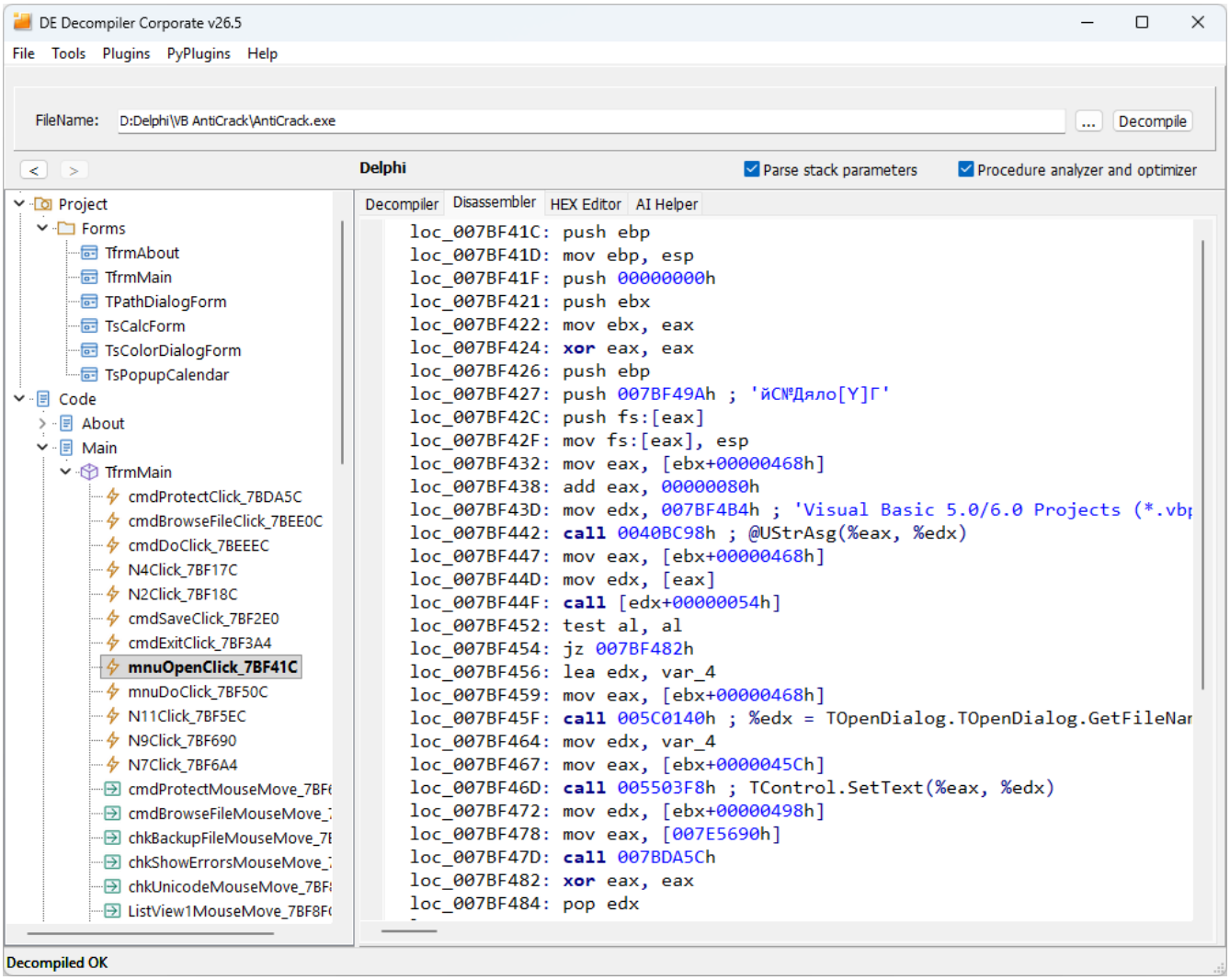
Analyze prototypes performs preliminary heuristic analysis of procedure arguments and return values. It is useful in Fast decompilation mode when private functions lack explicit signature metadata. This analysis takes additional initial processing time. Check inferred signatures against callers and native instructions when they affect an analysis conclusion.

Add Addresses retains virtual addresses in the high-level listing for comparison and navigation. Keep it enabled for **Copy to disassembler**. The Disassembler displays addresses independently of this setting.

When output is cached or the complete project has already been processed, reopen or reprocess the input after changing the relevant settings.

4.3 Disassembly display

The Disassembler shows native instructions for checking code recovered from supported applications written in Delphi or built with C++Builder.



Native instructions for a recovered Delphi procedure.

Display options

Add HEX dump displays machine-code bytes before each instruction. **Ignore chars convertions** keeps non-English strings readable instead of converting their characters to numeric Unicode values. Disable it when you need the numeric representation. Use code addresses to move between the listing and the HEX Editor.

User classes and VCL classes

In **Tools > Options > Disassembler**, **Show user classes** includes discovered application classes in class processing, filtering recognized library units and inherited methods. Application classes with names that also occur in libraries are retained when their unit identity distinguishes them. **Show VCL classes** also includes internal VCL library classes and the complete class view, including compiler-generated frames. The latter can add many library classes to the analysis; leave it disabled when you want to concentrate on the application's own implementation.

Enable VCL-class processing when the investigation requires the library implementation as well. Set the required class options before opening or reprocessing the file. Both controls are shown in Program options.

Comparing interpretations

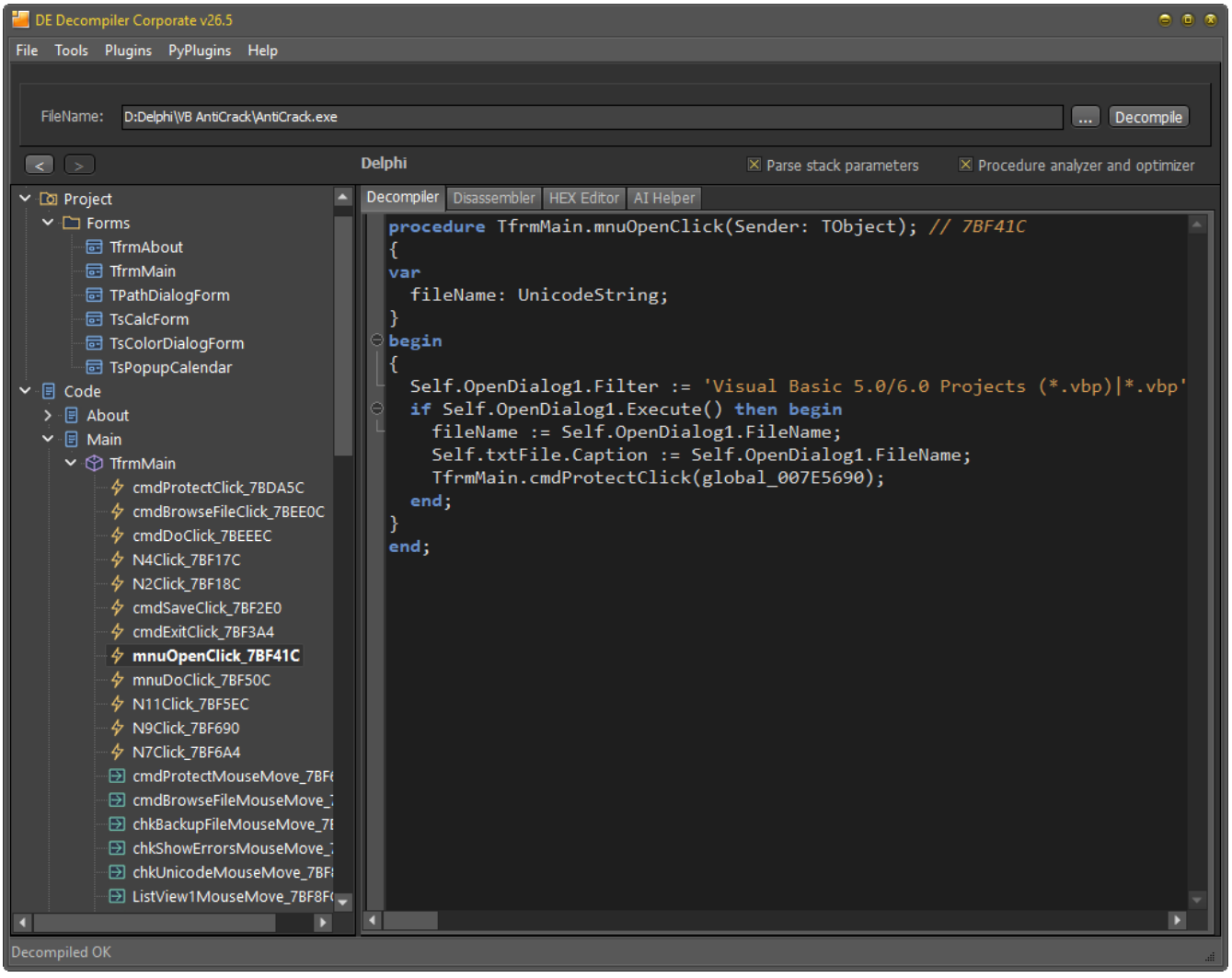
Check argument preparation, field accesses, conditional branches, calls and returned values against native instructions. A readable high-level statement can omit a detail that matters to a security investigation.

The **Copy to disassembler** workflow combines reconstructed statements with native instructions. See Viewing and comparing code.

4.4 Themes and fonts

The Color Scheme list changes the presentation of code and binary views, including the available dark theme.

Select a scheme under **Tools > Options > Personalize**. Choose a readable combination of foreground, background and syntax colors for long review sessions.



The same Delphi code workflow in the dark interface theme.

Custom theme files

Presets are `.ini` files in the `colors` folder under the installation directory. The filename supplies the preset name. Copy an existing preset to create your own and preserve the keys you do not intend to change. A selected theme can be applied without restarting the application, including after editing its INI file.

The [Colors] section uses decimal color values. Its keys include Background, Foreground, KeyWords, LocationsAndArguments, Procedures, UnderCursorBackground and UnderCursorForeground. UseDarkMode selects a light (0) or dark (1) interface. Colors2=1 enables the extended color section.

An excerpt using extended code-view settings:

```
[Colors]
UseDarkMode=1
Colors2=1

[Colors2]
TVFontName=Segoe UI, Tahoma, Arial
TVFontSize=10
FontName=Consolas, Courier New, monospace
FontSize=11
Background=1E1E1E
Foreground=DFDFE0
Key=CD9069
String=569CD6
HexFontName=Consolas, Courier New, monospace
HexFontSize=11
HexBackground=1E1E1E
HexForeground=DFDFE0
```

Font and syntax settings

The basic [Font] section provides Name, Size, KeyWordsBold, LocationsAndArgumentsBold and ProceduresBold. In [Colors2], font names can contain comma-separated fallbacks. Tree fonts use TVFontName and TVFontSize; code views use FontName and FontSize; the HEX Editor uses HexFontName and HexFontSize.

Extended colors use hexadecimal values. Syntax keys include Key, Comment, Char, Float, Hex, Number, String and Symbol. HEX Editor keys include HexBackground, HexForeground, HexGrid, HexActiveFieldBackground, HexOffsetBackground, HexCurrentOffset, HexOddColumn, HexEvenColumn and HexOffset. Under-cursor colors control matching-identifier highlighting in code.

4.5 Processing modes and caching

Fast decompilation, full processing and cached output support different analysis workflows.

Decompile only forms

Enable **Decompile only forms** when you need recovered VCL form information without processing procedure bodies.

Fast decompilation

In **Fast decompilation**, the initial processing establishes project structure and procedure bodies are processed as you open them. This is useful when investigating selected units or methods. With caching disabled, reopen a procedure after changing its processing options to obtain a new listing.

Analyze prototypes adds preliminary heuristic signature analysis. This preliminary disassembly can also collect String references. If a complete string-reference pass has not run, the dialog prompts you to disable Fast decompilation and reprocess the input.

Full processing

Disable **Fast decompilation** and process the file again to generate procedure listings in advance. This takes longer, but makes the generated text available for whole-project search and reference analysis. Project and combined-listing exports can also process procedures on demand. A change to decompilation settings then requires processing the file again.

Cache decompiled code

Cache decompiled code retains generated procedure listings. Disable caching when comparing reconstruction options interactively. Cached results do not automatically demonstrate the effect of a changed setting.

Declare variables

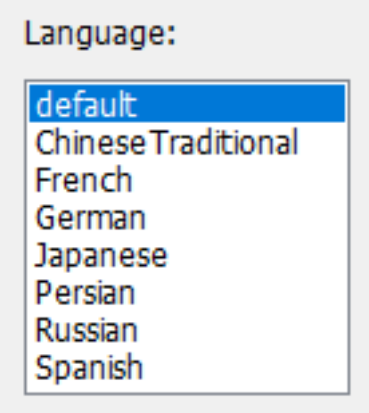
Declare variables adds recovered variable declarations to the listing. Read inferred types in the context of the available RTTI knowledge, callers and native instructions.

Window layout

Window style as in Visual Studio places code on the left and the project tree on the right. It changes workspace layout rather than the underlying analysis.

4.6 Interface language

Select the interface language under Tools > Options in the Personalize section.



The interface-language selector.

Choose an available localization and restart DE Decompiler so that the selected language is applied consistently across its windows. This manual uses the English interface captions.

Local and online documentation

The English manual is available online and as a PDF. Keep a local copy when working in an isolated analysis environment. The local manual includes the product website and support email in About the developer.

If the selected help translation is unavailable, DE Decompiler falls back to the English PDF or website. When a local PDF or CHM cannot be opened, the corresponding online documentation is used.

4.7 AI Helper

AI Helper processes the Delphi-like Pascal listing already reconstructed by DE Decompiler from supported applications written in Delphi or built with C++Builder. It supports local models through Ollama and online models from OpenAI (ChatGPT) and DeepSeek through the OpenAI API integration.

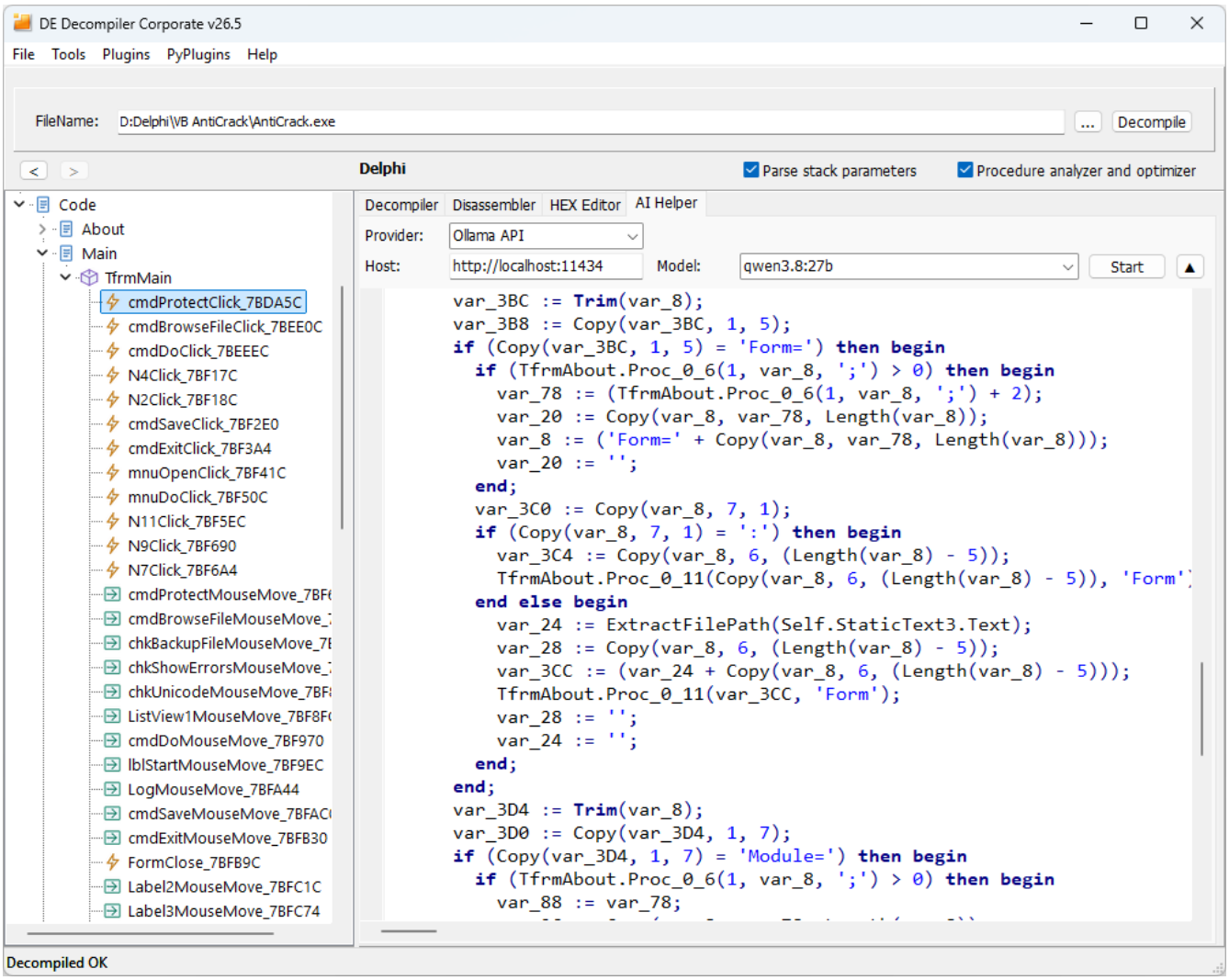
1 Open a procedure and review its **Decompiler** output.

2 Select **AI Helper**.

3 Choose the **Provider**, set the server address and select a model.

4 Start processing and compare the suggestion with the original decompiled listing and native instructions.

Processing runs in a separate thread and uses the complete current procedure, including folded code. The prompt requests Object Pascal output and retains unknown call targets and recovered code inside export wrappers. Use **Cancel** to cancel a long request. Secondary connection settings can be collapsed to leave more space for code.



AI Helper processing a recovered Delphi procedure through Ollama.

Ollama

Run Ollama on a compatible workstation or server, install the required model there, and connect AI Helper to that instance. A local Ollama server normally uses `http://localhost:11434`. The model must be installed in the instance selected by the **Host** field.

Model choice and GPU memory

Lightweight models such as `qwen3:8b` provide only limited improvement when processing recovered Delphi code. The larger `qwen3.8:27b` model is shown in this manual.

For this local `qwen3.8:27b` setup, use the following GPU-memory guidance:

Configuration	GPU memory	Expected use
Minimum	16 GB VRAM	Processing can be very slow. This is an entry-level configuration, not a comfortable workstation for regular model use.
Recommended	32 GB VRAM	The preferred configuration for regular local use of the illustrated model.

These figures refer to dedicated video memory on a GPU supported by Ollama. System RAM and the shared-memory figure reported for an old integrated graphics adapter are not equivalent to dedicated VRAM. A Pentium M-era computer with old integrated graphics is unsuitable for this setup. Check Ollama hardware compatibility before selecting a GPU.

The requirements apply to the computer running Ollama. If AI Helper connects to a separate Ollama server, that server must provide the model resources.

Context and other models

Choose the context size using the slider in Ollama settings; no additional Ollama extensions are required for this workflow. Context is measured in tokens and includes model instructions, input code and the generated answer. Increasing it requires more memory, so the figures above do not guarantee that every model format or context length will fit.

You may select other models supported by the configured provider. Their output quality and formatting can differ, and model-specific service text may remain in the result. Review the generated listing before using it.

Online models: OpenAI (ChatGPT) and DeepSeek

Select **OpenAI API** in **Provider** to use OpenAI models (ChatGPT) or DeepSeek online. Set **Host**, **API key** and **Model** for the selected service. The host, API key and model identifier must belong to the same service; use a model available to that API key.

DeepSeek uses the same **OpenAI API** provider. Enter the DeepSeek host, API key and model identifier in the corresponding fields. Online inference runs on the provider's infrastructure, so the local Ollama GPU-memory recommendations do not apply to the analysis workstation when using these services.

Data handling

Keep API keys out of screenshots and shared investigation material.

AI Helper sends the current decompiled code to the selected service. A server on another workstation or an external API can place that code outside the analyst's computer. Select the endpoint in accordance with your organization's handling requirements for proprietary source, samples and investigation evidence.

Reviewing suggestions

AI can suggest clearer names, declarations or control flow, but a suggestion is not proof of the binary's behavior. Review changed calls, conditions and types against disassembly and related procedures. Preserve the original decompiled result when recording a model-assisted interpretation.

Connection and model troubleshooting

- If a model is missing, confirm that it is installed in the Ollama instance selected by **Host**.
- If no response arrives, check that the service is reachable and that the host, API key and model identifier refer to the same provider.
- If processing is too slow, cancel the request and review the Ollama host's resources, model size and context setting.
- If an unfamiliar model leaves auxiliary tags or service text, remove that text while reviewing the result or select another suitable model.

4.8 Plugins and the SDK

Use FRX Image Viewer to inspect and extract recovered form graphics, or develop a custom plugin through the shared SDK.

FRX Image Viewer

FRX Image Viewer 1.2.0 previews and extracts images registered by VB Decompiler or DE Decompiler. VB Decompiler supplies VB5/6 and .NET form graphics; DE Decompiler supplies binary DFM graphics from supported VCL applications written in Delphi or built with C++Builder. Supported picture formats are BMP, PNG, JPEG, GIF, ICO and CUR.

Installation and opening

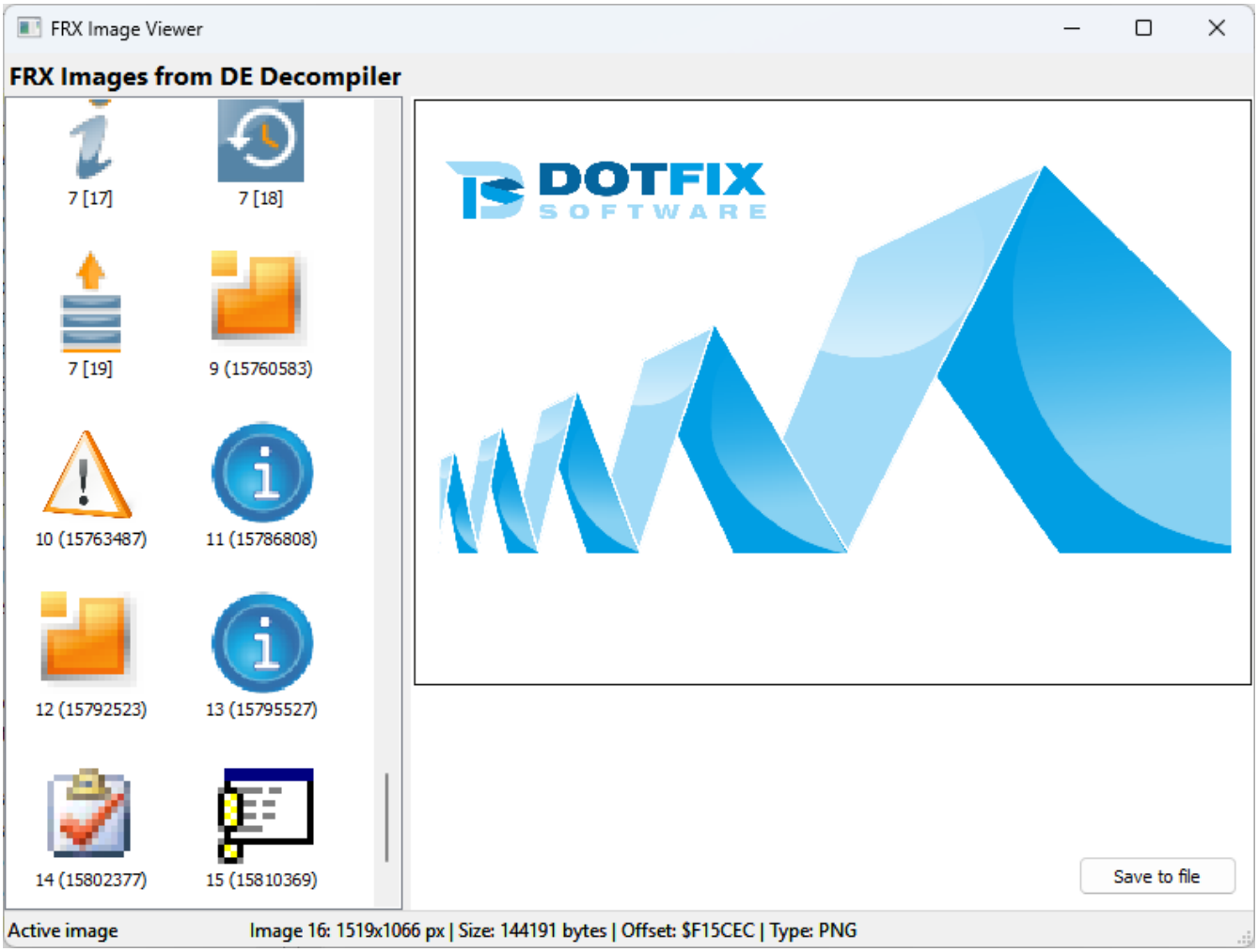
1. Copy `FRXViewer.dll` to the host installation's `plugins` subdirectory and restart the decompiler. The DLL architecture must match the host application, regardless of the analyzed file's architecture.
2. Open the executable or DLL and decode its forms so that the host registers the available images.
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 VCL ImageLists and compatibility

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

Shared SDK revision 26.6, supported by VB Decompiler 26.6 and DE Decompiler 26.6, 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.

The **Plugins** menu is the entry point for installed native DLL plugins. The shared callback interface exposes supported project information and code to custom extensions. Available DLL plugins are initialized at startup; build for the intended host architecture and Windows environment.

Developing a plugin

The SDK includes examples for Visual C++, Rust, Delphi and Free Pascal. Use its exported entry points and host callback to implement the operations required by your workflow.

You can also ask an AI coding agent to implement a plugin. Provide the SDK and API reference link and a description of the required behavior, then review, build and test the resulting DLL against DE Decompiler and representative input files.

Compatibility of the interface

Use the documented SDK calling convention, exported entry points and data representations. Shared API identifiers keep their existing names, including the `VBDecompilerPlugin` prefix; changing the product name in an exported symbol would change the ABI.

Sharing the interface does not make every VB Decompiler plugin suitable for DE Decompiler data. Check the expected object types, recovered code language and operations available for the current selection.

Developer reference

See the Plugin API reference on the product website. It describes exported entry points, command identifiers, callback data and buffer requirements. The SDK provides active-view and project-level code access; operations that search every decompiled procedure need the corresponding listings to be available. Code-query callbacks can process individual procedures when the matching input is available.

Displayed items, function marks and addresses

Plugins can query the displayed form, unit, class or procedure with GetDisplayedTreeItem (65), read or change a function mark with GetModuleFunctionMark (66) and SetModuleFunctionMark (67), and locate functions by virtual address with FindModuleFunctionByVA (68). GetModuleFunctionVA (69) returns the full virtual address, including 64-bit addresses. Existing callback signatures and command numbers remain compatible.

4.9 Obfuscation

The integrated obfuscator supports Win32 and Win64 VCL applications written in Delphi or built with C++Builder.

This workflow also applies to supported Win32 and Win64 VCL applications built with C++Builder.

Use obfuscation when preparing your own supported VCL application for distribution. This operation changes the compiled binary; exporting recovered Pascal code is a separate operation.

Obfuscating an application

1. Preserve the original compiled application and work with a separate copy.
2. Open the application through **File > Open program**.
3. Select **Tools > Obfuscate project** and follow the output dialog.
4. The operation backs up the native input before writing and rebuilds the recovered model after saving. On a conversion, write or reopen failure, it attempts to restore the original file and model; if restoration cannot complete, it reports the recovery-copy path. Test the resulting application with the components, data and workflows used in its intended deployment.

Checking the result

Keep the original and obfuscated builds distinguishable in release records. Check form creation, component interaction and event-driven behavior in the resulting application. Obfuscation makes analysis more difficult, but does not guarantee that a program cannot be examined or decompiled.

5 Reference

5.1 Interpreting recovered code

DE Decompiler reconstructs Delphi-like program logic from native instructions and available metadata. The information retained in the binary determines the detail of the result.

What the high-level listing represents

Native-code decompilation recovers expressions, `if` conditions, `case` blocks, loops and calls, including recognized VCL methods. This makes program logic accessible for reverse engineering and security analysis. The listing reconstructs the implementation; it does not reproduce the original source text.

Names, data structures and calls

- **Variable names:** original local names are commonly absent from the binary. Generated identifiers represent the recovered values; analyst-assigned names can make their role clearer.
- **Types and layouts:** RTTI supplies type information where it is available. Other structures may be inferred from access sizes and addresses, so field types, boundaries and signatures can need further interpretation.
- **Calls:** recognized VCL methods can be shown by name. A call that cannot be identified may remain a method or address-based call without its original VCL name.
- **Compiler variation:** compiler generations, libraries and optimization affect the emitted instructions and metadata. Version-specific RTTI knowledge helps interpret them, but an inferred type or expression can still need correction.

Check significant conclusions

1. Identify the exact binary, unit, class and procedure under review.
2. Read the decompiled logic and declarations, using the native instructions to resolve uncertain expressions or types.
3. Examine callers, argument preparation, conditional branches and associated strings.
4. Use the HEX Editor when an instruction or recovered value needs to be checked against file data.
5. Record what was observed directly and what remains an inferred interpretation.

Security investigation

The recovered structure and code help forensic investigators, antivirus analysts and corporate security teams locate suspicious logic, potential backdoors and unwanted functionality. Establish how a value or call is used before drawing a conclusion. A string or API name alone does not establish malicious behavior or prove that a branch executed.

Source recovery

Recovered forms, declarations and procedure code can reduce manual recovery work. Native-code decompilation does not reproduce a compilable source project. Before opening exported code in an IDE, manually analyze and correct decompilation inaccuracies and incompatibilities, reconcile inferred types and unidentified calls, and provide missing units, components and dependencies. Building the recovered material requires that manual work.

Applications built with C++Builder

C++Builder-specific library RTTI knowledge bases are not yet available. Functions outside the shared VCL coverage, and object properties or methods that cannot be identified, may be shown without their library names. Code decompilation quality for C++Builder is therefore noticeably lower than for applications written in Delphi. This limitation concerns code-reconstruction quality and library-name recognition. The other supported features remain available for C++Builder under the same edition rules as for applications written in Delphi.

C++Builder code is reconstructed as Delphi-like Pascal, with PAS/DFM output. It is not converted back into the original C++ source. As with other native-code reconstruction, review and correct inferred types, generated names and decompilation inaccuracies manually before using the recovered material in an IDE.

5.2 Technical support

A focused report helps reproduce a recognition, form or decompilation issue.

Include the DE Decompiler build, operating system, input filename and exact build identity, compiler version when known, and the relevant unit, class, form or procedure address. State the processing options and the steps needed to reproduce the issue.

Attach a minimal example or original binary only when you are authorized to share it. For confidential investigations, describe the issue first and agree on an appropriate way to provide material. Never include API keys or unrelated customer information.

See the support page for current contact information.

Support email: support@de-decompiler.com

5.3 Licensing

DE Decompiler is offered as Free Lite and with Single Developer and Enterprise commercial licenses.

Editions

Edition	Scope
Free Lite	VCL form information, native Win32 and Win64 disassembly, syntax highlighting, procedure search and String references. High-level code reconstruction is not included.
Single Developer License	The commercial product features for one named developer on one physical computer or locally hosted virtual PC.
Enterprise License	The commercial product features for up to 20 users on up to 20 physical computers or locally hosted virtual PCs within the licensed company.

AI Helper, the HEX Editor and obfuscation require a commercial license.

Updates, support and renewal

New commercial licenses include one year of updates and technical support. One-year renewals are available for both Single Developer and Enterprise.

After the updates and support period ends, the installed version continues to work on the computer where it was activated. Access to newer versions, technical support and moving an activation to another computer require an active updates and support period.

Assignment and activation

A Single Developer license is personal to its named developer. It cannot be shared between developers, rotated among team members, transferred or reassigned to another developer.

Licenses cannot be transferred or re-registered to another person or company. The licenses are not floating or concurrent licenses. Enterprise activations are allocated within the licensed company's user and computer limits.

See the licensing page for current prices and purchase links, and the download page for Free Lite.

5.4 About the developer

DE Decompiler is developed by DotFix Software.

DotFix Software has been developing software tools since 2001. See the company website and VB Decompiler.

DE Decompiler was first offered publicly in 2007–2008. Development continued for internal use before its return to public licensing. Its primary focus is native-code analysis of Win32 and Win64 VCL applications written in Delphi 3–13.

VCL applications built with C++Builder are also supported. Forms, units, classes, functions and the other tools are available in the same workspace; compiler-specific RTTI limitations affect the quality of reconstructed code, rather than access to these features.

Product website: www.de-decompiler.com
Support email: support@de-decompiler.com