VCL TO FMX CONVERTER

v6.0 NAVIGATOR

USER GUIDE

Detailed Operator, Validation, and Workflow Manual

Revision date: August 3, 2026

Audience: developers, technical operators, project owners, and maintainers who will run the converter and validate converted Delphi projects.

Table of Contents

1. Purpose and Audience

The VCL2FMXConverter exists to automate the majority of the engineering work involved in moving a Delphi application from standard VCL to FMX. It is intended to reduce repetitive and tedious migration labor, surface the most important conversion risks early, and leave the user with an FMX project that can be compiled, tested, and polished in Delphi with as little of work as possible.

VCL to FMX is not an easy task. FMX was designed as a completely different architecture from VCL, hence while there are many similarities, there are many missing properties and events in FMX that are present in VCL. This makes for significant re-engineering in some instances so a developer should evaluate the reasons for moving an application from VCL to FMX, especially the benefits gained.

While small projects may convert 80-100%, larger projects with many forms and files to convert may experience only from 50-75%. This should be expected at the outset, simply due to the differences between VCL and FMX, and the missing properties and events.

This manual is written for people who will actually operate the converter. That includes developers converting their own applications, technical staff helping with migration efforts, project owners supervising conversion work, and maintainers who need to understand how to rerun the tool safely and interpret its output.

- Use this guide when you need to run the converter, interpret its output, and validate the converted FMX project.
- Use the engineering guide when you need to understand the converter internals or change its code.
- Use both guides together when a conversion is being actively refined and stabilized. The converter itself, although written logically, is extremely complicated. Modification of the converter, although possible, is discouraged, and only the most experienced programmers should attempt it.

2. What the Converter Is, and What It Is Not

Converter Behavior	What It Means in Practice
Automates most of the migration work, most times from 40% to 75% or more	The converter handles the majority of syntax, form, mapping, and startup translation work automatically.
Works as a migration system, not a blind text replacer	It uses parsing, mapping, and rewrite logic rather than relying only on global text substitution.
Aims for buildable and reviewable output	The goal is not merely changed text; the goal is a project that can be opened, built, and exercised in Delphi.
Leaves a small review tail	A converted application can still need font, layout, or minor behavioral review after the automated pass.
Does not replace software testing	A successful conversion still needs compile, runtime, and user validation to confirm the result is correct.

A useful way to think about the converter is this: it is a serious productivity accelerator, not a promise that every converted application is instantly production-ready without verification. In practice, it can bring the vast majority of a migration under control and make the remaining issues understandable.

The current generic baseline is broader than the earlier releases. It now includes better FMX form emission, broader standard-control coverage for many items, along with more honest blocking-issue reporting when the converter knows the output is not yet trustworthy. Reporting is more reliable and gives more insight and more suggestions for conversion problems.

The live Component Map, Property Map, and Event Map pages are now backed by the same current mapper knowledge used during conversion. That makes them the best quick-reference view of what the current v6.0 build knows about standard component, property, and event coverage. Version 6.0 Navigator also provides **consolidated JSON mapping packs** that make third-party component handling visible and auditable before a conversion run. Many new properties have been added to the vendor components.

2.1 All About Mapping Packs

Mapping packs are optional JSON files that extend VCL2FMX Converter's component knowledge without hard-coding every third-party library into the converter itself.

A mapping pack tells the converter how to recognize selected third-party VCL components and what to do with them during conversion. This is especially useful for third-party Delphi libraries such as DevExpress, TMS, JVCL, Raize/Konopka, TeeChart, FastReport, EhLib, InfoPower, Virtual TreeView, Devart, and 12 others. Currently there are 22 exported mapping packs as of this release, with possibly more coming later.

Mapping packs are designed to be transparent and conservative. They do not magically guarantee a perfect conversion of every third-party component. Instead, they help the converter make better decisions, produce better reports, and safely convert only the components that can be mapped with reasonable confidence. However one should remember that third-party components many times cannot be adequately converted to FMX. This is not a fault of the converter but rather the engineering of the components themselves.

The `mapping_packs` folder should be placed beside the converter executable. This is very important because the converter expects it and if they are not placed correctly, no packs will load for a conversion , with the converter assuming there are none available.

For example:

`C:\Converter\VCL2FMXConverter.exe`
`C:\Converter\mapping_packs\`

Each mapping pack is a `.json` file. The converter reads these files before a conversion run and uses the rules inside them when it encounters matching components.

A mapping-pack rule may tell the converter to do one of several things:

- `convert` means the converter can create an FMX replacement with reasonable confidence.
- `partial` means the converter can create an approximate FMX replacement, but the result needs review because some behavior, styling, data binding, or advanced features may not transfer exactly.
- `manual_review` means the component was recognized, but the converter should not automatically replace it. The report will explain what needs attention.
- `detect_only` means the converter identifies and reports the component but does not generate an FMX replacement. This is used for complex controls such as grids, ribbons, schedulers, docking systems, report designers, dashboards, and other components that usually require manual redesign.
- `preserve` means the converter may keep the original class/reference when appropriate, often for nonvisual or cross-platform components, but the result should still be reviewed.

Mapping packs can include confidence percentages. A high-confidence rule is safer for automatic conversion. A low-confidence rule should normally produce a report item or manual-review note instead of silently changing the component. If you make your own packs, keep this in mind.

The converter report shows which mapping packs were loaded and what mapping-pack rules were used. This makes the process auditable. Users can see what was converted, what was partially converted, what was detected only, and what still needs manual review.

Mapping packs are especially helpful when you own and understand a third-party component library. You can inspect the JSON rules, adjust them, add new rules, or create your own pack for components used in your applications.

On the web site as of this release, v 5.0, there is now a companion program to allow you to make your own .json packs. The program name is **JSON Mapping Pack Studio** and it is also free to use under the same licensing. It comes with its own source code and documentation. It resides in the Downloads section, with the other documentation and source code.

A mapping pack does not replace testing and never will. Always open the converted FMX project in Delphi, build it, run it, and test the actual application workflow. Third-party components often include styling systems, editors, repository items, data controllers, design-time behavior, custom painting, or platform-specific code that cannot be fully converted by a basic rule or possibly not even converted at all.

In short, mapping packs let VCL2FMX Converter recognize and handle more real-world Delphi projects while keeping the converter open, transparent, and safe. They are a practical bridge between automatic conversion and careful manual review.

2.2 Mapping-pack actions

Action	What it means for the user
convert	The converter can generate the target FMX component with reasonable confidence.
partial	The converter can generate an approximate FMX component, but the report must describe the limitations.
manual_review	The converter should report the component and avoid unsafe automatic conversion.
detect_only	The converter identifies the component or vendor but does not generate a replacement.
preserve	The converter may keep the original class/reference when preservation is safer or explicitly intended.

Repeat: As stated earlier, for normal operation, keep the mapping_packs folder beside VCL2FMXConverter.exe, in whatever folder you install it. This is very important.

For example if the converter structure is similar to: **C:\converter\VCL2FMXConverter.exe**

then the mapping packs would be in: **C:\converter\mapping_packs**

The packs act as companion files in unison with the converter executable. After a run, review the generated report and their associated report sections for loaded packs, converted third-party components, partial conversions, detect-only findings, preserved components, unknown third-party-looking components, and manual-review reasons.

2.3 Loading the Consolidated Mapping packs

Certain mapping packs were previously offered as two separate files. This inadvertently caused some duplication, despite the converter running normally. As of this version, all packs for the vendors have been consolidated so there is no overlap or duplication. Follow these directions to install the new packs:

1. Remove/delete all old mapping packs from the mapping_packs directory
2. Copy the zipped consolidated packs into the directory
3. Unzip the file there. There should be 22 packs displayed.
4. Remove the zip file from the directory, as it is no longer needed.

2.4 Mapping-pack results

The generated HTML or text reports are a detailed audit trail. If a pack converts a component, the report should say which pack was used and what confidence level was declared. If a pack only detects a component, the report should explain why no FMX replacement was generated. It is recommended to use the HTML report since it is the easiest to read. The loaded mapping packs will be prominently displayed in the report.

3. System Requirements and Workspace Preparation

- A Windows development environment with Delphi 12.1 or later installed and working. This was developed under Delphi 13, patch 1.
- Access to the original VCL source code and form files.
- A separate target directory for generated FMX output. **This is very important**.
- Sufficient disk space for source, output, reports, and milestone backups.
- A backup strategy before major converter runs or large rule changes. Backup files cost you very little. In the end you may not need them. However it would cost you more if you needed a backup file to recover from a major error, and you did not have one. Caveat lector!

A clean folder layout is strongly recommended. Keep the converter source, the original VCL project, and the converted FMX output in separate locations. This prevents stale output files from being mistaken for current results and makes it much easier to compare source and target artifacts during troubleshooting. If you must make multiple conversions (which can be the case at times), it is recommended to clear the output directory each time. The converter WILL overwrite the files but it is wise to remove all files with each run.

Workspace Area	Recommended Use
Converter folder	Contains the converter source and its own documentation/scripts.
Source project folder	Contains the original VCL application being migrated.
Output folder	Contains only the newly generated FMX project for the current run.
Backup storage	Contains milestone snapshots, cloud copies, or archived converter states.

4. Building and Launching the Converter

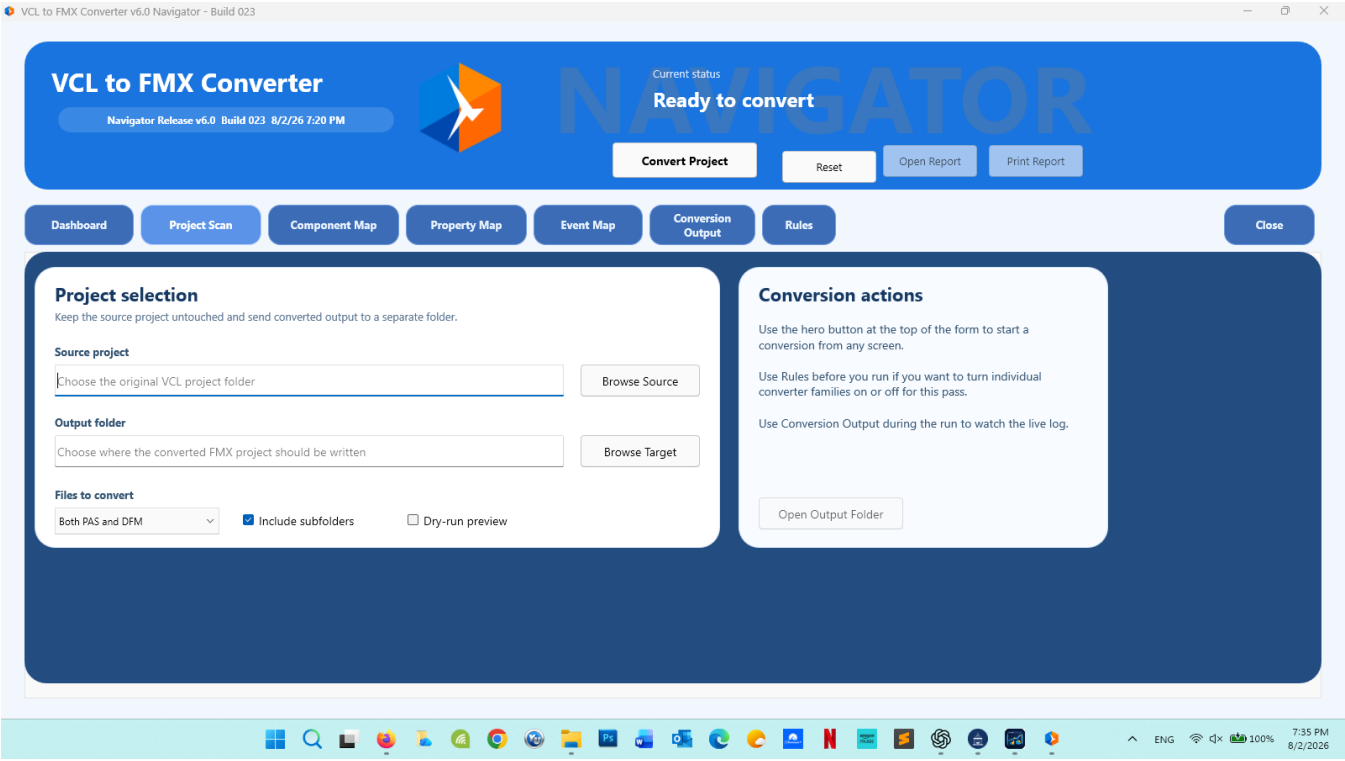
1. Open the VCL2FMXConverter project in Delphi.
2. Build the converter and confirm the build completes cleanly.
3. Run the converter application.

After compiling and building, the converter app should open with a slight delay while it loads files it needs. Verify that the tabbed workspace opens and that Dashboard, Project Scan, Component Map, Property Map, Event Map, Conversion Output, and Rules are available.

If the converter itself does not build, stop there and resolve that issue before attempting any project migration. The converter must be treated like any other software tool: it needs a healthy build before its output can be trusted. If you use mapping packs, be sure they are in the correct location, beside the converter .exe file.

5. Main Window and Controls Walkthrough

The converter main window is simple and organized logically by tabs and button order. Currently there is no ‘Help’ button offered due to the amount of information available in this guide.



Main screen

Controls Explanation:

Control	Purpose	How a User Should Think About It
Source project	Points at the original VCL project or source root.	This is the source of truth the converter will scan.
Output folder	Points at the output directory for generated FMX artifacts.	Treat this as disposable and regenerable output, not as your only copy of the application.
Files to convert	Chooses PAS only, DFM only, or both.	Use narrower scopes for focused retests, and use both for full conversions.
Include subfolders	Controls whether subdirectories are scanned.	Leave it enabled for complete projects unless you intentionally want to limit the run.

Report and output actions	Opens the generated report, prints it when supported, or opens the output folder.	Use these after every run so report review is part of the normal workflow.
Critical Areas / Data Aware / 3rd Party / WinAPI	Per-run rule toggles for the specialized conversion families.	Leave these enabled for a normal full pass, and only turn one off when intentionally isolating behavior.
Convert Project	Starts the conversion process.	Only press this after source, output, scope, and rule choices are correct.
Conversion Output log	Displays progress and key messages while also supporting report review.	Read it after every run; it is the first indicator of what happened.

The main window is a tabbed workspace. **Dashboard** shows the latest run summary and quick actions. **Project Scan** holds the path and scope controls, **Component Map / Property Map / Event Map** expose the live mapper reference surface, Conversion Output shows the live log and report status, and Rules controls the four specialized conversion families. The converter still does its real work behind the scenes, but the operator needs to use the front end carefully because path mistakes, stale output folders, and poorly scoped runs can create misleading results.

6. Recommended Conversion Workflow

6.1 Practical Operator Workflow

Stage	Operator action	Expected result
Prepare	check source	set clean target
Run	select options	start conversion
Open	load FMX project	build in Delphi
Validate	run app	exercise real workflows
Refine	improve converter	rerun cleanly

1. Prepare a clean output directory. Always make sure it is empty.
2. Run the converter against the selected VCL project.
3. Open the generated FMX project in Delphi.
4. Resolve structural and compile issues first. If necessary, make multiple conversions after corrections.
5. Run the converted app and validate real user workflows.
6. Only after runtime stability, review and tune layout or styling details.

This sequence matters. Trying to perfect colors or spacing before the application builds and runs cleanly is usually wasted effort. Stabilize structure first, runtime second, and presentation third. Fix any problems that are reported during the compilation of the app, then fix the minor things. Don't expect a 100%, fix-free conversion.

7. Preparing a Source Project for Best Results

Some preparation work in the source project can make the converter easier to use and the output easier to validate. This is not always required, but it is often helpful.

- Know which folder is the real source root and which files are truly active in the project.
- Be aware of duplicated unit names or old backup copies or unneeded files inside the source tree. The should be removed, since they serve no purpose.
- Understand any database, media, or file-path dependencies that the converted application will need at runtime.
- Keep note of major workflows that must be validated after conversion, such as startup, scheduling, playback, database edits, dialogs, and reporting.
- If possible, have the original VCL application available for side-by-side behavior comparison.

The converter can do a great deal, but it is still easier to use when the operator knows which source files matter and which business workflows are critical.

8. Running a Conversion

1. Enter or browse to the source folder.
2. Enter or browse to the target folder. A convenience folder name is provided if you wish to use it.
3. Choose PAS only, DFM only, or both. Both is highly recommended for ALL conversions.
4. Set the include-subfolders option and the four rule toggles as needed. This is your choice.
5. Review the specialized rule choices if your workflow needs a focused retest.
6. Click Convert and allow the process to complete.

Read the log before opening the output in Delphi. When the HTML report exists, use Open Report or Print Report directly from the converter, and use Open Output Folder when you need the generated files immediately. The HTML report is highly organized and easy to read.

For full project migrations, the normal choice is to convert both PAS and DFM files recursively into a clean output folder. **It is recommended to always do this**. Targeted runs are useful only when you already know that a specific phase is the thing under review.

During the same run, the generator can inventory real support files in the source root and in existing build-output folders. That makes the report more trustworthy because helper executables, DLLs, library files, sound-font files, and similar runtime companions are reported from real locations instead of being guessed.

9. Reading the Log and the Conversion Report

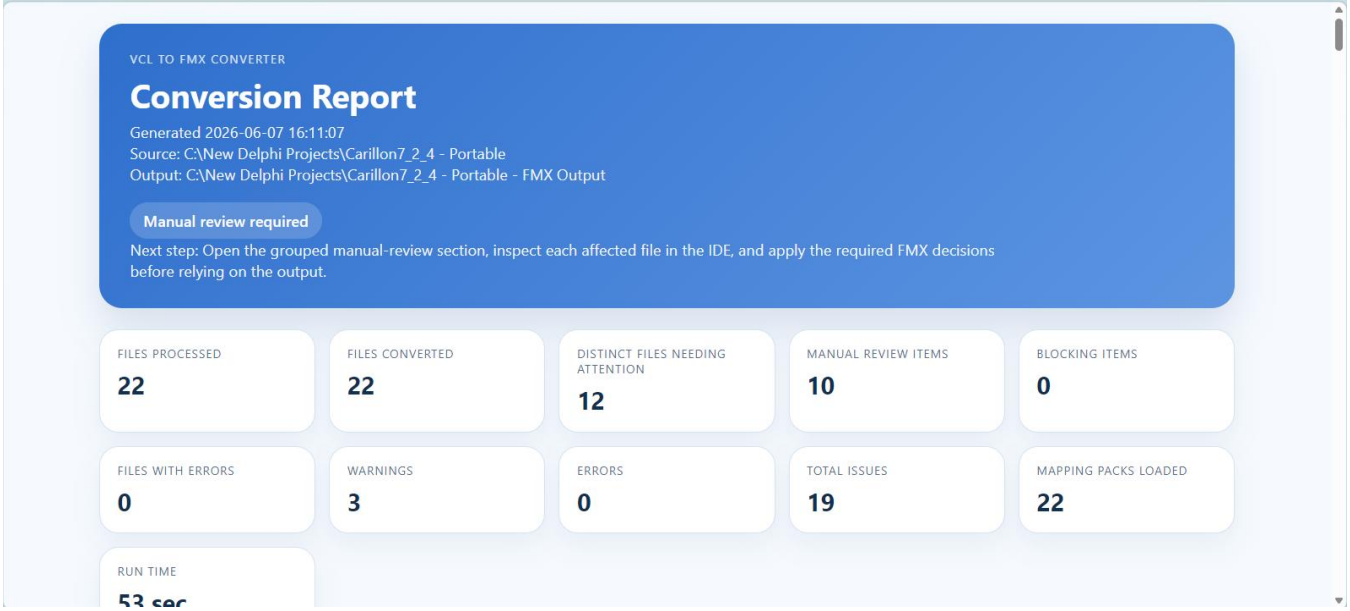
Every conversion run should be reviewed through two channels: the live log in the converter UI and the generated report files in the output folder. When the HTML report exists, the Open Report and Print Report buttons use it first. These are not optional extras; they are part of the operating workflow. The HTML report for v 5.0 has been totally revamped and has more information, better suggestions and is easier to read.

[v6.0 report-counting note] Report totals in v6.0 separate actionable issues from informational notices and dry-run notices. Total issues means warnings, manual-review items, and errors. Dry-run preview entries, such as files that would be saved, appear in their own Dry-run notices section and are not counted as conversion issues.

Manual review items in the HTML report are grouped. Click the > arrow on each item to view affected locations and recommended fixes.

Information Source	What It Tells You	How to Use It
Conversion Output log	Live progress and immediate messages during the run.	Use it to see which files were processed and whether the run completed normally.
Conversion reports	Persistent text and HTML records of what the converter saw and reported.	Use them for later review, troubleshooting, printing, and communication with other engineers.
Delphi compile output	The authoritative correctness check after generation.	Use it to identify real syntax, unit, and form-reader issues.
Runtime behavior	The final reality check for workflow correctness.	Use it to confirm that the converted application actually behaves acceptably.

HTML Conversion Report Example:



Example HTML Report

Pay special attention to the final run status. The current v6.0 report flow distinguishes Blocking issues present, Manual review required, and Clean conversion. The generated report now suggests a recommended next step, distinct files needing attention, and files with conversion errors. Treat blocking issues as stop conditions even if files were generated. Informational messages about external project assets are different: they warn that companion files may still need manual copying or relocation before runtime testing. Mapping-pack runs additionally report loaded packs, third-party conversions, partial conversions, detect-only findings, preserved components, and unknown third-party-looking classes.

The current report also distinguishes between support files that were actually staged into the FMX output directory and support files that were only found in the source root or beside an existing built executable. Read those companion-file notes carefully, because staging a helper file into the output tree is not the same thing as confirming it will deploy beside the final built executable.

If the report and the compiler disagree, the compiler wins. If **Code Insight** and the real compiler disagree, the compiler wins again. The memo log and report are aids, not replacements for build and run validation.

10. Understanding the Output Folder and Files

Output Artifact	Purpose	User Action
Converted .pas files	FMX-oriented Pascal source units derived from the original VCL source.	Open in Delphi and build as part of the generated project.
Converted .fmx files	FireMonkey form definitions derived from DFM files.	Allow Delphi to load them and watch for reader errors.
Project files	The FMX project shell and startup metadata.	Open the project file in Delphi and build it.
Conversion report	Text summary of the run.	Keep it for diagnostics and engineering review.
Milestone backups	Snapshots of converter state or other preserved artifacts.	Store separately so they do not pollute source or output folders.

Project files now include a more complete FMX shell, and the generator attempts to carry along common companion assets such as .res, .ico, .manifest, and safe local project references. If the source project points at an absolute external asset outside the source tree, you should expect to verify or copy that file manually before runtime testing.

The output folder may now also contain staged runtime companions such as helper executables, DLLs, library files, or sound-font style support files. The generator adds them when it finds real candidates in the source root or in existing build-output folders such as Win32, Win64, Debug, Release, or deploy. Those files are copied to make review easier, but you still need to verify the final deployment location expected by the generated app.

If you get messages about these missing file types, most likely the converter cannot see them for some reason. You should investigate and make them available and reconvert or you may just want to move them manually.

As stated previously, a clean output folder is one of the most important habits in using the converter. Stale generated units and forms can create false problems, hide real changes, and lead Delphi to load the wrong copy of a file.

11. Opening the Converted Project in Delphi

1. Open the generated project in Delphi.
2. Let Delphi load the forms and watch for form-reader errors first.
3. Build the project and capture the real compiler results.
4. Fix structural problems before pursuing visual polish.
5. Rebuild and rerun after each significant converter improvement.

The converter strips or remaps several common VCL-only form properties before Delphi sees the FMX output. Even so, form-reader errors still deserve first priority because they block every later stage of validation.

The first successful load of the project is a major milestone, but it is not the end of validation. A project can open and still fail at compile time, or compile and still fail at runtime, generating runtime errors or throwing exceptions. Treat these as separate checkpoints.

12. Validating the Converted Application

Validation Layer	Goal	Typical Questions
Structural	Can Delphi read the generated forms and units?	Are there reader errors, malformed collections, or invalid properties?
Compile-time	Does the project build on the first pass?	Are units missing, events wrong, or symbols unresolved?
Runtime	Does the application start and execute real workflows?	Do timers, media, data, dialogs, and navigation work?
Visual	Does the application look acceptable and remain usable?	Are fonts, labels, colors, and layouts readable and sensible?

Validation should now explicitly cover the newer generic support surface. That includes numeric entry controls, radio-group style selections, font-picking workflows, scrollable long-message dialogs, and any wave-audio paths that depended on Winapi.MMSystem or waveOut APIs some of which have no FMX counterparts.

Real validation means exercising the actual workflows that matter to the application: data entry, startup, scheduling, media playback, dialogs, and reporting. A converted app is only as good as the workflows it can successfully perform.

- Test startup and shutdown behavior.
- Test any timers or schedules if the original app depended on them.
- Test data-aware screens and save/cancel workflows.
- Test media or file operations where applicable.
- Test secondary forms, help, menus, and common user actions.
- Test numeric entry controls such as TNumberBox and converted TSpinBox screens.
- Test TRadioGroup and TFontDialog workflows if the original application used them.
- Test wave-audio or former WinMM paths where the source project uses MMSystem or waveOut APIs. These have no counterparts in FMX and may be blockers.

If a form uses shared datasets, shared data sources, shared navigators, or shared combo controls, close and reopen the form and verify the shared object still behaves correctly afterward. In addition, if the application launches help files, external documents, URLs, or utility executables, test those paths explicitly in the converted FMX build.

12.1 Windows Message Handlers and How They Are Processed

The VCL2FMX Converter automatically detects and transforms all three common forms of Windows message handling found in VCL projects. You do not need to identify these patterns yourself — the converter finds them, rewrites what it can, and leaves clear guidance comments in your source code for anything that requires a manual decision.

12.2 The Three Patterns and What Happens to Them

12.2.1 Standard Windows Messages (WM_SIZE, WM_ACTIVATE, etc.)

These are the built-in Windows system messages that VCL components handled directly. In FMX, each of these has been replaced by a proper event that you can wire up in the Object Inspector.

What the converter does:

- Strips the message handler declaration from the class interface section.
- Replaces the handler body with a comment block naming the correct FMX event to use.
- Issues an informational note in the conversion report.

What you need to do:

- Open the converted form in the IDE.
- In the Object Inspector, locate the event named in the comment (for example OnResize, OnActivate, or OnClose).
- Double-click the event to create the handler and move your logic there.

> **Note:** You do not need to reference System.Messaging for standard Windows messages. These map directly to FMX events with no extra units required.

12.2.2 Custom User Messages (WM_USER + N)

Many VCL applications define their own private messages using WM_USER offsets to communicate between forms and components. FMX has no equivalent to this mechanism, so the converter generates working replacement code based on TMessageManager from the System.Messaging unit.

What the converter does:

- Generates a typed message class (for example TMyAppMsg) for each custom message it detects.
- Injects a TMessageManager.SubscribeToMessage call into your FormCreate method.
- Injects the corresponding Unsubscribe call into your FormDestroy method.
- Adds System.Messaging to your uses clauses automatically.
- Generates a FMXMessageBridge.pas support file in your output folder if any custom messages were found.

What you need to do:

- Review the generated message class names — rename them to match your application's naming conventions.
- Fill in the message handler body with your original logic.
- If you were passing data in lParam or wParam, map those values to properties on your typed message class.

Note: FMXMessageBridge.pas is generated once and never overwritten. It contains helper classes and a usage guide. Add it to your project and review the comments inside.

12.2.3 WndProc Overrides and Raw Message Pumps

Some VCL applications override WndProc to intercept messages at a low level, or call the Windows message pump directly (PeekMessage, GetMessage, DispatchMessage). FMX has its own cross-platform event loop and does not expose a message pump hook.

What the converter does:

- Wraps the entire WndProc override body in a comment block so your original code is preserved for reference.
- Adds an explanatory comment describing the FMX alternative approach.
- Marks the item in the conversion report as requiring manual review.

What you need to do:

- Read the preserved comment block to understand what the original WndProc was doing.
- Identify the underlying intent — most WndProc overrides are intercepting a specific message for one of these reasons: responding to a system event, preventing default behaviour, or passing data between windows.

- Implement the equivalent using FMX events, TMessageManager subscriptions, or a platform service (IFMXApplicationEventService) as appropriate.

> **Note:** Raw message pump calls (PeekMessage, GetMessage, DispatchMessage) are removed with explanatory comments. FMX manages its own event loop on every platform and these calls are not needed.

12.3 The Conversion Report

After conversion, check the report for any items flagged as Manual Review Required in the Windows Messaging category. Each entry includes:

- The original source line that triggered the flag.
- The file name and line number.
- A plain-English description of what needs to be done.

Items flagged as Info only (standard WM_ messages) require no code changes — just the Object Inspector step described in section 2.2.1.

12.4 FMXMessageBridge.pas

If your project contained any WM_USER custom messages, the converter generates a file named FMXMessageBridge.pas in your output folder. This file provides:

- TFMXUserMessage — a base class for all converted custom messages.
- FMXSendMessage — a helper for synchronous message dispatch.
- FMXPostMessage — a helper for asynchronous dispatch via TThread.Queue.
- A usage guide in the comments at the top of the file.

You may add FMXMessageBridge.pas to your project manually or it will appear in the .dpr uses clause automatically if the converter detected custom messages. The file will not be regenerated or overwritten on subsequent conversions.

12.5 Quick Reference

VCL Pattern	Converter Action	Your Action
Standard WM_ handler	Replaced with comment + event name	Wire event in Object Inspector
WM_USER custom message	TMessageManager code generated + FMXMessageBridge.pas	Rename classes, fill handler logic
WndProc override	Preserved as comments with guidance	Redesign using FMX events or platform services
Message pump calls	Removed with explanatory comment	No action needed — FMX manages its own loop

13. Troubleshooting and Recovery Procedures

Symptom	Likely Cause	Recommended Response
Form reader error	An unsupported VCL property or malformed FMX output was emitted.	Fix the converter rule, regenerate into a clean output folder, and retest.
Compiler errors after regeneration	A converter rule is missing or incomplete.	Treat the compiler message as the next engineering target and improve the converter globally.
Runtime access violation	A startup, event, media, drawing, or binding mismatch remains.	Trace the workflow, isolate the phase, and improve the converter behavior globally.
Visual mismatch only	The app is basically working but presentation differs.	Defer deep visual polishing until structural and runtime behavior are stable.
Code Insight disagreement	The IDE parser is stale or confused.	Trust the real compiler and reopen files or restart the IDE if needed.
Conversion completed with blocking issues	A real unsupported component or blocking rule gap was detected.	Stop there, read the report, and improve the converter globally before trusting the output.
External project asset reference not copied	The project shell references a file outside the source tree or in an unresolved path.	Copy or relocate the asset manually, then decide whether the generator needs a new generic asset-copy rule.

If a run goes badly do not try to repair everything at once. The fastest route is usually to identify the highest-severity problem, make changes and reconvert.

14. Making Changes to the Converter Software

Refer to the Engineering Manual for guidance for making changes to the converter. It is a highly complex application which uses much integration and branches to many areas. Only qualified programmers who have a deep understanding of Delphi should attempt any modifications that may be required.

14.1 Conversion Contracts in v6.0

Version 5.0 adds an executable conversion-contract system. A contract is a small Delphi fixture, paired with an expectation file, that proves a known conversion rule still behaves correctly. Contracts are used by the engineering test runner before release. They are not loaded during a normal conversion and they do not compare every user source file against a long checklist.

[v6.0 contract baseline] The current v6.0 suite contains 227 executable expectations, including report accounting, dry-run preview behavior, image API cleanup, unsupported-routine depth guards, include analysis, Windows messaging, DFM/FMXL fidelity, graphics/GDI checks, semantic resolution, and protected uses cleanup.

For an operator, the contract system matters because it explains why v6.0 is more disciplined than earlier releases. The converter rules are tested against specific examples for include files ($INCLUDE), Windows messaging, protected uses clauses, DFM/FMXL output, graphics and GDI substitutions, and project integration. When a real project reveals a reusable conversion issue, the preferred v6.0 workflow is to add a contract first, then update the converter until the contract passes.

The contracts live in the project contracts folder. Each expectation records the input file, expected status, expected conversions, expected manual-review items, units that must be present or absent, required output patterns, and forbidden output patterns. The release-candidate contract run is part of the validation evidence for the converter, while normal user conversions continue through the parser, mapper, rewrite, integration, and project-generation pipeline.

If a conversion report mentions a Windows message, GDI drawing path, include-file issue, protected uses cleanup, or generated FMX behavior that is covered by the contracts, that report item is the converter applying the tested rule to the user project. Some items still remain manual review by design, especially where an automatic FMX replacement would be unsafe for the general user base.

15. Operational Best Practices

- Always keep the source project, output project, and converter workspace separate.
- Use clean output folders for serious test cycles.
- Back up milestone converter states before risky changes. Backups are extremely important to save accomplished work.
- Keep zipped milestone backups out of the live converter workspace so later workspace backups do not recurse and grow uncontrollably. Make sure they are off the development computer in another location, like a cloud service like DropBox, Google Drive, etc.
- Treat real compiler output as the authority over IDE-only hints or stale parse errors.
- Record what was fixed globally so later projects benefit immediately.
- Do not burn time on tiny visual tweaks until the application builds and runs correctly.

16. Frequently Asked Questions

Question	Answer
Will every project convert perfectly on the first run?	No. The converter is powerful, but real-world projects still expose new patterns that may need another global rule. Refer to earlier sections of this manual for proper expectations.
Should I hand-edit the generated FMX project permanently?	Only for temporary testing or minor final polish. The real fix should go into the converter whenever the pattern is reusable.
What do I do in Delphi after conversion?	The generated project still has to be built, run, and validated in the real Delphi environment.
Why are backups important?	Because converter improvements are cumulative, and known-good milestone states are valuable when comparing behavior or recovering from regressions.
When should I review visuals?	After the project loads, compiles, and runs acceptably. Visual polish is usually the last phase.
What does blocking issues mean?	It means the converter found a serious unsupported pattern or rule gap. Generated files may exist, but you should resolve the blocking issue before treating the output as a valid converted project.

Appendix A. Quick Start Checklist

1. Build the converter.
2. Choose source and target folders.
3. Use a clean output directory.
4. Run conversion.
5. Open the FMX project in Delphi.
6. Build, run, and validate the converted app.
7. Improve the converter globally and regenerate as needed.

Appendix B. Pre-Conversion Checklist

- Confirm the real source root folder.
- Confirm the target output folder is correct and clean.
- Know which workflows are critical in the original application.
- Ensure supporting files such as databases or media are available for runtime testing.
- Plan where milestone backups will be stored.

Appendix C. Post-Conversion Review Checklist

- Did the forms open without reader errors?
- Did the project compile on the first pass?
- Did the application start and close normally?
- Did core workflows run correctly?
- Are the remaining issues structural, runtime, or merely visual?
- Was the converter fixed globally if a reusable defect was found?

Appendix D. Glossary

Term	Meaning
VCL	Visual Component Library, Delphi's traditional Windows UI framework.
FMX	FireMonkey, Delphi's multi-platform UI framework.
DFM	A Delphi form definition file used by VCL projects.
FMX form	The converted FireMonkey form file, typically emitted as .fmx.
LiveBindings	The FMX binding model used for data-aware behavior.
Global fix	A converter change that benefits multiple current or future projects, not just one output file.
Mapping pack	Optional JSON file that adds transparent third-party component detection, conversion, preservation, or manual-review rules.
Conversion expectations	The amount of conversion of a VCL project you can expect. Generally from 50% to 75%. Each user experience may vary.

Maintenance Update - August 3, 2026

This maintenance update incorporates field-test feedback from real Delphi projects. The converter now recognizes additional standard VCL controls, including TBitBtn and TTrayIcon, so they are not incorrectly reported as unknown third-party components.

TBitBtn receives a conservative FMX TButton substitution note. TTrayIcon remains an explicit platform-specific manual-review item because FMX has no direct standard tray-icon component.

The current v6.0 contract suite contains 227 executable expectations. The latest verification run passed 227 of 227 contracts, and the Win32 and Win64 Release builds were rebuilt from the current source.

Navigator V6.0

Last changed: August 3, 2026

What is new in Navigator

Navigator strengthens confidence before and after conversion. It uses designer-owned release identification, model-backed text, HTML, and JSON reports, strict mapping-pack validation, safer temporary validation workspaces, and broader compiled contract coverage.

Build identification

Every Navigator iteration shows its release name, v6.0 version, release-tracking details inside the centered badge. V6.0 is the current testing release.

Conversion and review expectations

The converter creates a strong FMX starting point. Compile and test generated projects, review every blocking and manual-review report item, and verify visual behavior, Windows-message replacements, custom drawing, data-aware controls, and third-party components. Dry-run preview remains the safest first pass for an unfamiliar project.