VCL2FMXConverter v7.0 Atlas

Detailed Runtime Flow

A source-based walkthrough from launch to application close

Delphi VCL to FireMonkey conversion

Last changed October 18, 2026

V7 Atlas documentation

Contents

Lifecycle at a glance....3
1 Launch and ready state....3
1.1 Executable and form initialization....3
1.2 Reference content at startup....4
1.3 The visible ready state....4
2 Preparing a conversion....5
2.1 Source and destination....5
2.2 Scope and rule settings....5
2.3 Help and reset before a run....6
3 Starting the worker....6
3.1 Input validation and early exits....6
3.2 Transition from form state to running state....6
3.3 Context, translation, and engine handoff....7
4 Engine preflight and discovery....7
4.1 A fresh conversion session....7
4.2 Path and project validation....8
4.3 Semantic index and output preparation....8
5 Converting Pascal source....9
5.1 Reading and source analysis....9
5.2 Conversion and compatibility passes....9
5.3 Final normalization and save....10
6 Converting form designs....11
6.1 Input guards and parsing....11
6.2 FMX generation and review signals....11
7 Project output and reports....12
7.1 Project companions and dependencies....12
7.2 Output audit and mapping suggestions....13
7.3 Report files and result classification....13
8 Completion and next action....14
8.1 Returning results to the form....14
8.2 Reviewing, resetting, and closing....14
Interpreting the result....15

This guide traces the implemented v7 control flow from the Windows executable entry point to application close. It explains when the form prepares reference content, how a conversion worker is started, how Pascal and DFM files are processed, and when project files and reports are written. Each section names the source code that supports the description.

The sequence is based on the current local Pascal source, FMX form, and project file. It describes code paths and their conditions; it does not claim that every generated project compiles or behaves correctly without developer review.

Lifecycle at a glance

Stage	Result
Launch	The FMX form and reference views reach a ready state.
Preparation	The operator selects source, destination, file scope, and rule options.
Worker startup	The form validates inputs and passes a snapshot of options to a background thread.
Preflight	The engine validates paths, discovers files, and builds conversion knowledge.
Source conversion	Pascal and DFM branches process each selected file in the engine's order.
Project and reports	The engine generates companions, audits output, and writes reports when reached.
Completion	The UI receives final counts and paths for developer review.

1 Launch and ready state

The executable initializes an FMX form, prepares its reference views, and waits on Project Scan. No conversion engine is running while the operator reviews the initial window.

1.1 Executable and form initialization

The project block and streamed form establish the initial application state.

Windows starts the compiled program. The DPR calls Application.Initialize, creates TfrmMain as frmMain, then enters Application.Run. No conversion context or engine is created by this program block. Application.CreateForm loads MainForm.fmx. Its designer-owned settings make the window maximized and bind OnCreate, OnShow, and OnCloseQuery to the form methods. The designer gives the file-type selector “Both PAS and DFM”, recursive scanning and the four rule-family boxes checked, and dry-run preview unchecked. Source and target edits begin empty.

Source: VCL2FMXConverter.dpr, final begin block; MainForm.fmx:1–19, 1194–1238, 1981–2119

1.2 Reference content at startup

Form creation first locates Help and establishes the synchronization needed by the later worker. It then prepares the three reference views.

The form first tries a docs/Help path two levels above the executable. If index.html is not there, it falls back to docs/Help beside the executable. This only chooses a location; it does not start the local Help server. The form creates FActiveEngineLock. Later the worker uses it to publish or clear the active engine safely, and destruction uses it when attempting cancellation.

Source: MainForm.pas:392–399; MainForm.pas:400, 2289–2308, 424–460

PopulateReferencePages writes “Building ... reference” placeholders into the component, property, and event memos before their actual content is prepared. LoadComponentMapReference calls BuildComponentMapContent when its cached HTML or plain text is absent. That routine creates a temporary conversion context and component mapper, enumerates discoverable VCL classes and mappings, and builds both HTML and fallback text. The browser receives HTML when available; otherwise the memo remains visible. Errors become a visible fallback message.

Source: MainForm.pas:402, 479–486; MainForm.pas:403, 488–520, 590–767

LoadPropertyMapReference follows the same cache-and-fallback pattern. It creates another temporary context and mapper, builds property-rule content from the mapping database and available inventories, and loads HTML into the property browser when possible. LoadEventMapReference again creates a temporary context and mapper if its cache is empty. It prepares event names, mapped target events, and signature review information, then displays HTML or memo text. Each TComponentMapper.Create in the three reference builders loads built-in mappings and the FMX catalog, optionally loads an existing user mapping JSON file, loads enabled mapping packs, builds RTTI inventories, enriches and normalizes the FMX catalog, and rebuilds its lookup index. This work is part of opening the app, before any conversion is started.

Source: MainForm.pas:404, 522–554, 768–1049; MainForm.pas:405, 556–588, 1050–1320; MainForm.pas:618, 807, 1090; Converter.Mapper.Component.pas:297–335

1.3 The visible ready state

The form clears previous run artifacts, enables the controls for a new conversion, and selects the initial view.

ResetArtifacts clears remembered output and report paths, zeroes run counters, sets status to “Ready to convert”, displays “No run started yet” on the dashboard, and disables report/output actions until real paths exist. UpdateRuleSummary shows the current option states on the dashboard and Rules tab. UpdateUIForConversion(False) enables Convert, Reset, Browse buttons, the file-type selector, and the rule-option boxes. It restores “Convert Project” on the button and calls UpdateReportActions, which leaves report and output actions disabled without files. BuildTabChrome records the seven designer-owned navigation controls and scroll boxes. FormCreate scrolls the Rules summary to the top, chooses Project Scan explicitly, and applies the active-tab visual style. This is why Project Scan normally appears first, even though the fallback in FormShow names Dashboard.

Source: MainForm.pas:406–407, 1565–1583, 1638–1650, 1721–1745; MainForm.pas:408, 1757–1784, 1626–1636; MainForm.pas:409–411, 1485–1511, 1519–1551

When the form becomes visible, FormShow chooses Dashboard only if no tab is active, applies the tab theme, and calls the three reference loaders again. Their cached content normally prevents a rebuild; the loaders still refresh the browser or memo display. Maximization comes from MainForm.fmx, not FormShow. Application.Run continues the FMX event loop. The converter has no selected source, target, active worker, or output report. Opening a navigation tab only changes the visible tab and its button styling; it does not start the conversion engine.

Source: MainForm.pas:413–423; MainForm.fmx:13; VCL2FMXConverter.dpr; MainForm.pas:1553–1563

2 Preparing a conversion

The operator supplies folders and options on the form. These choices remain editable until Convert Project is pressed; choosing a folder or opening Help does not itself start a conversion.

2.1 Source and destination

Folder selection populates the edits used for the later run. A suggested destination is only a convenience and does not create output yet.

btnBrowseSourceClick opens SelectDirectory with the current source edit as its starting value. On confirmation it writes the chosen folder to edtSource and calls SuggestTargetFromSource. A canceled dialog changes nothing. SuggestTargetFromSource returns immediately when edtTarget already has text. Otherwise it takes the selected folder's parent and name, then fills edtTarget with "<source name> - FMX Output" beside the source. It does not create that directory. btnBrowseTargetClick opens SelectDirectory with the current target edit. Confirmation replaces edtTarget; cancellation leaves it unchanged. Directory creation is deferred to engine preflight.

Source: MainForm.pas:2056–2066; MainForm.pas:1893–1913; MainForm.pas:2068–2075; Converter.Core.FileManager.pas:202–265

2.2 Scope and rule settings

The selected file types, recursive search, rule families, dry-run mode, and language determine the work captured at conversion time.

The combo offers PAS only, DFM only, or both. The selected ItemIndex is captured when Convert is pressed and later mapped to ftPas, ftDfm, or ftBoth. Project-file validation is a separate path from the .pas/.dfm file loop. Include subfolders controls recursive enumeration. Critical Areas, Data Aware, 3rd Party, WinAPI, and Dry-run preview are checked or cleared on the form. RuleOptionChanged refreshes the summaries when a rule box changes. Graphics conversion is not exposed as one of those switches and is applied by the Pascal pipeline. The language combo is a designer-owned TDATFMXLanguageComboBox. On Convert, DATActiveLanguageCode is captured for the report runtime; language-pack loading occurs in the worker, not in the form's startup sequence described above.

Source: MainForm.fmx:1194–1204; MainForm.pas:2184, 2262–2268; MainForm.fmx:1210–1238, 1981–2119; MainForm.pas:1747–1750; Converter.Core.Integration.pas:1930–1977; MainForm.fmx:432; MainForm.pas:2191, 2274–2287

2.3 Help and reset before a run

The operator can inspect local Help or return the form to its startup values without entering the engine.

btnHelpClick checks for index.html, starts the designer-owned HTTP server on localhost, and opens the help URL with a language query parameter. If local hosting fails it opens the installed HTML file directly. This is an optional UI action and does not run conversion. btnResetClick calls ResetToStartupState when no worker is assigned. That method blanks source and target, selects both file types, turns the default rule boxes on, turns dry run off, resets the log and summaries, scrolls tabs to the top, returns to Project Scan, and focuses the source edit. Reset is blocked while a conversion thread exists.

Source: MainForm.pas:1942–1966; MainForm.fmx:2263–2273; MainForm.pas:1930–1940, 1585–1624

3 Starting the worker

The Convert Project handler validates the visible inputs on the UI thread. Once they pass, it captures the settings, disables conflicting controls, and starts a background worker that owns the conversion context and engine.

3.1 Input validation and early exits

The handler prevents a duplicate run and rejects incomplete or immediately invalid folder choices before a worker exists.

btnConvertClick first checks FConversionThread. If it is assigned, it shows “A conversion is already running” and exits. No second context or engine is created. The handler copies edtSource and edtTarget into local Src and Dst after Trim. It does not yet canonicalize them; the file manager performs canonical-path checks later.

Source: MainForm.pas:2128–2144; MainForm.pas:2146–2147; Converter.Core.FileManager.pas:63–93

If Src is empty, a message requests a source folder, Project Scan is selected, and the handler exits before clearing the old log or creating a thread. If Dst is empty, a message requests a target folder, Project Scan is selected, and the handler exits.

Source: MainForm.pas:2149–2155; MainForm.pas:2157–2163

DirectoryExists is checked for Src. A missing folder produces a message and a return to Project Scan. More detailed tree safety checks still happen later inside TFileManager.Reset. The form compares expanded source and target paths after removing a trailing delimiter, ignoring case. If equal, it asks for a different output folder and exits. The engine later adds stronger same-or-nested checks.

Source: MainForm.pas:2165–2171; MainForm.pas:2173–2181; Converter.Core.FileManager.pas:202–265

3.2 Transition from form state to running state

The visible log and status are prepared for this run, and the operator cannot change conversion controls while the worker runs.

The prior memo log is cleared and receives “Preparing conversion”; the active tab switches to Issues. The handler captures file type, recursion, four rule families, dry-run state, and language into local values so later UI edits cannot change this run’s options. The destination is stored as the provisional last output path. Old report paths are cleared; a default completion snapshot zeroes counters. Dashboard and Issues labels show the target and “Report will appear here after this run”; the running summary appears. UpdateUIForConversion(True) disables Convert, Reset, Browse, file-type, and option controls. It changes the button caption to “Converting...”, sets status to “Running conversion”, and disables Open Report, Print Report, and Open Output.

Source: MainForm.pas:2183–2192; MainForm.pas:2194–2202, 1683–1720; MainForm.pas:2203, 1757–1784

3.3 Context, translation, and engine handoff

A new thread receives a snapshot of the chosen options. It constructs the services for this run and enters the engine.

The handler creates a TThread, sets FreeOnTerminate to False, and starts it. Its body initializes a completion snapshot with the selected target and a failed default status, calls ConversionThreadProc, then queues a UI-thread callback in a finally block. ConversionThreadProc constructs TConversionContext and copies the captured source, output, recursion, report-on, four rule-family toggles, dry-run state, and file-type enum into its TConversionOptions. Defaults also enable mapping packs and point to executable-side mapping files.

Source: MainForm.pas:2205–2232; MainForm.pas:2235–2268; Converter.Core.Types.pas:845–862

The worker constructs TConverterEngine, then a TTranslationRuntime for Localization/Languages and a local report-language preference path. It loads the selected report language, assigns text direction and translation callbacks, publishes the active engine under the lock, and installs the log callback. If the pack cannot load, the worker raises an error before Engine.Convert. Engine.Convert(Context) is the handoff to discovery and conversion. Log messages from the worker call the form's Log method, which synchronizes additions to the FMX memo on the main thread.

Source: MainForm.pas:2269–2303; MainForm.pas:2304–2308, 1786–1808; Converter.Core.Engine.pas:988

4 Engine preflight and discovery

Before converting a selected source file, the engine rebuilds session services, validates input and output locations, discovers files, and prepares cross-unit knowledge. Blocking preflight failures can return before report generation.

4.1 A fresh conversion session

The engine clears previous counters and issues, creates its orchestration services, and loads available mapping knowledge.

Convert clears report paths and existing issues, resets the current-file name, loaded-pack and usage records, counters, cancellation flag, timestamps, and in-memory conversion log. It rebuilds the file manager, orchestrator, and project generator for this context, then logs the session heading, source, and output. TConversionOrchestrator.Create constructs DFM and Pascal parsers and a component mapper, followed by critical, data-aware, third-party, WinAPI, graphics, normalization, auto-fix, LiveBindings, uses, and compatibility services. The mapper loads built-in mappings, an existing user mapping file if configured, enabled JSON mapping packs, RTTI inventories, and lookup indexes. When enabled, LoadMappingPacks looks beside the executable and then development-folder fallbacks. It sorts top-level JSON filenames and loads them in that order. LoadMappingPackFile validates schema, pack metadata, version, mode, IDs, and rule content. A missing folder records an informational issue; invalid packs record diagnostic issues.

Source: Converter.Core.Engine.pas:988–1035; RebuildServicesForContext:167–180; Converter.Core.Integration.pas:81–109; Converter.Mapper.Component.pas:297–335; Converter.Mapper.Component.pas:369–end of LoadMappingPackFile

4.2 Path and project validation

Discovery is restricted to eligible source paths, and project inputs are checked for ambiguity and consistency.

TFileManager.Reset clears its list and index, canonicalizes source and output, and rejects a nonexistent source. With recursion on, it checks the source tree for ordinary directories before traversal. Exceptions become scan issues; a blocking issue causes Convert to return early. CollectMatchingFiles walks the selected tree when recursion is on, or only its root when off. It rejects links/reparse points and previous conversion reports inside the source tree, ignores build-artifact folders and any path inside the chosen output, and adds only eligible .pas/.dfm files. Reset sorts the final list. ValidateProjectSelection searches for one relevant DPR and DPROJ, preferring source-root candidates and rejecting ambiguous multiple matches. If both exist it reads DPROJ MainSource and verifies that it points to the selected DPR. A DPR or DPK counts as project input even if the .pas/.dfm list is empty.

Source: Converter.Core.FileManager.pas:63–93; Converter.Core.Engine.pas:1036–1038; Converter.Core.FileManager.pas:95–156, 178–201; Converter.Project.Generator.pas:89–149; Converter.Core.Engine.pas:1039–1044

If no eligible source files and no project input were found, Convert records “No eligible source files” and exits. Because this happens before the report stage, the normal report files are not generated on this path.

Source: Converter.Core.Engine.pas:1040–1045, 1110–1114

4.3 Semantic index and output preparation

With a valid selection, the engine builds cross-unit information and secures the output location before taking the first file.

BuildCrossUnitSemanticIndex reads .pas files from the selected source scope, excludes build artifacts, strips comments, indexes class-like declarations and qualified method implementations, and records counts. Enumeration failure is a warning; cancellation can stop the indexing pass. PrepareOutput rejects missing output, identical source/output, either directory nested inside the other, and unsafe directory trees. It creates the output directory even for dry run. On normal runs only, it removes named stale output subfolders. A failure records an issue and returns before the file loop.

Source: Converter.Core.Engine.pas:877–987; Converter.Core.FileManager.pas:202–265

After successful output preflight, Convert captures FileCount and logs “Found N files to process”. This count covers the selected .pas/.dfm list, not the project files handled later by TProjectGenerator. While files remain and the cancellation flag is clear, GetNextFile advances its index. Convert increments processed count, logs “[i/total] Processing”, and dispatches .pas to ProcessPasFile or .dfm to ProcessDfmFile. An unexpected per-file exception records an error and the loop continues.

Source: Converter.Core.Engine.pas:1047–1053; Converter.Core.Engine.pas:1054–1089; Converter.Core.FileManager.pas:267–296

5 Converting Pascal source

Each selected Pascal file follows the same processing branch. It is read, analyzed, rewritten through ordered conversion passes, checked for structural integrity, and written only when the save path is allowed.

5.1 Reading and source analysis

The engine records the current file and prepares the code and paired form information used by later passes.

ProcessPasFile logs its filename, resets its local ConvertedOK flag, and stores the path in Context.CurrentFile so subsequent issues can be tied to the source being handled. TryReadWithEncoding reads text with supported encodings and logs the detected encoding. A failure increments skipped and error counts, records a blocking Pascal-read issue, and returns without converting or saving this file. The original text is retained, then ScanPascalIncludeDirectives locates include references and copies reachable include files only through guarded source-root paths. This occurs before the main Pascal rewrite.

Source: Converter.Core.Engine.pas:1695–1752; Converter.Core.Engine.pas:215–225, 1753–1771; Converter.Core.Engine.pas:1772–1774, 1161–1694

ConvertPascal first clears prior binding state. When Data Aware is enabled and a same-named DFM exists, PrepareFormDataBindings loads and parses it, analyzes data-aware controls, and enriches metadata. Failure becomes a warning and clears the partial binding state; it does not by itself stop this Pascal pass. The orchestrator aligns shape field types, rewrites folder-picker dialog patterns, and rewrites simple grid draw handlers after preparing bindings. These passes work on the source text before deep Pascal parsing. FPascalParser.Parse extracts a current unit model from the working source. The next passes use this model rather than relying only on global text substitutions. ProcessMethodBodies identifies paint/draw methods with GDI calls and records manual-review issues for unsupported drawing paths. ProcessStructuredPascalDeclarations examines declarations, including Windows-message patterns. ProcessSemanticResolution checks handlers and implementation/class relationships against the project-wide index.

Source: Converter.Core.Integration.pas:261–269, 1366–1405; Converter.Core.Integration.pas:264–270; Converter.Core.Integration.pas:272–273; Converter.Parser.Pascal.pas:503 onward; Converter.Core.Integration.pas:275–277, 1748–1929

5.2 Conversion and compatibility passes

Automatic fixes and rule-specific conversion operate in a defined order; disabled rule families do not perform their optional passes.

FAutoFixes.Apply rewrites known source constructs before the advanced converters. The same pass will run again after advanced code insertion, so its transformations must remain safe when repeated. RewriteStructuralWrapperPanels loads the paired DFM when present, identifies visual containers that are structural wrappers, and changes matching Pascal field types to TLayout. This keeps declarations aligned with the DFM-to-FMX component mapping. ApplyAdvancedConverters invokes ConvertAllCriticalAreas for patterns such as custom drawing, message handlers, and owner-draw behavior only when Critical Areas is enabled. It records an informational issue when code changes.

Source: Converter.Core.Integration.pas:279–281, 286–289; Converter.Core.Integration.pas:283–284, 127–260; Converter.Core.Integration.pas:291–292, 1930–1953

The advanced pass converts recognized Windows API calls and adds platform conditionals when WinAPI is enabled. Turning that option off skips these two calls, not the rest of the Pascal pipeline. The advanced pass invokes FGraphics.ConvertGraphics regardless of the four rule-family checkboxes. It handles graphics-related positioning and colors and records an informational issue. After advanced conversion, FAutoFixes.Apply runs again on the changed text to normalize constructs and compatibility code emitted by those converters.

Source: Converter.Core.Integration.pas:1954–1976; Converter.Core.Integration.pas:1959–1966; Converter.Core.Integration.pas:294–297

WrapBitmapCanvasScenes examines generated bitmap canvas drawing and adds FMX BeginScene and EndScene handling where its patterns apply. FRuntimeNormalizer.NormalizeColors converts recognized stored VCL color values and can add runtime support required by the rewritten source. It is called again late in the pipeline because later passes may emit new references. PromoteFullTextMemoFieldsInPascal changes recognized data-field handling so FMX grids do not present full-text memo data as opaque blobs.

Source: Converter.Core.Integration.pas:299–300, 2066 onward; Converter.Core.Integration.pas:302–303, 326–329; Converter.Core.Integration.pas:305–306, 758–766

The injector first warns about unusual multiple-form declarations, then adds generated LiveBindings-related source based on prepared binding metadata. Data-aware choices affect whether the earlier DFM analysis provided candidates. ReduceGeneratedManualCleanup removes routine generated manual-cleanup comments after binding injection so remaining review notes retain meaning. FCompatibilityInjector.InjectClasses adds supported VCL-to-FMX compatibility classes for recognized high-value controls, then InjectLifecycleFixes adds schedule/media cleanup behavior where recognized.

Source: Converter.Core.Integration.pas:308–310; Converter.Core.Integration.pas:312–313, 1978 onward; Converter.Core.Integration.pas:315–320

5.3 Final normalization and save

The last passes repair resource and implementation structure, preserve the Delphi terminator, and save the resulting source or record a failure.

NormalizeFMXResourceDirectiveSpacing changes recognized DFM resource references toward FMX and isolates directives before implementation-structure checks. FixImplementationSection checks and repairs recognized implementation-section patterns in the rewritten unit. FUsesRewriter.RemoveDuplicateImplementationUses then removes repeated implementation uses. RewriteRootFormDoubleClick adds root-form double-click adapters after structural rewriting. RewriteTextLayoutMath adjusts recognized label stacking calculations to FMX text metrics. NormalizeColors is then rerun for late helper references.

Source: Converter.Core.Integration.pas:321–323, 418–430; Converter.Core.Integration.pas:325–330, 459 onward; Converter.Core.Integration.pas:332–339

FUsesRewriter.Fix performs the final uses-clause cleanup, EnsureFMXResourceDirective establishes the FMX form-resource reference, and spacing is normalized again. If this block fails, an error issue with manual guidance is recorded but ConvertPascal reaches its success issue. Back in ProcessPasFile, if the original unit ended with end. and the converted text no longer does, RestoreFinalEndDotIfLost appends the marker and raises a warning for IDE review. The engine counts the conversion as converted before attempting the save. SaveConvertedFile derives a source-relative destination, refuses paths outside the output tree, sanitizes control characters, and writes Delphi source as UTF-8 with BOM. In dry run it records a “would save” issue and writes no converted .pas. A conversion or save failure increments errors and leaves a specific issue.

Source: Converter.Core.Integration.pas:341–349, 431–458; Converter.Core.Engine.pas:1695–1748, 1776–1782; Converter.Core.Engine.pas:1783–1810; Converter.Core.FileManager.pas:298–322, 422–468

6 Converting form designs

The DFM branch is separate from any paired-form analysis already performed for Pascal source. It parses the component tree, generates FMX form text, checks minimum structure and source fidelity, and then saves or withholds the result.

6.1 Input guards and parsing

The engine applies size and read guards before the parser produces a component tree.

ProcessDfmFile logs the filename, clears ConvertedOK, and stores CurrentFile. It rejects a DFM larger than VCL2FMX_MAX_TEXT_DFM_BYTES with skipped/error counts and a blocking issue. Before calling the parser, ProcessDfmFile requires VCL2FMXTryReadTextFile to read the source and retain its text for a later fidelity audit. Failure records a blocking read issue and returns. The parser contains a binary-DFM conversion path, but this earlier guard may prevent some binary inputs from reaching it; the guide does not claim all binary DFMs convert. ConvertDFM calls FDfmParser.LoadDFM, then Parse. LoadDFM applies its own size/read checks and can attempt binary-to-text conversion for a binary stream. ConvertDFM rejects empty input, an empty component list, or a root without a name and class.

Source: Converter.Core.Engine.pas:1982–2014; Converter.Core.Engine.pas:2016–2035; Converter.Parser.DFM.pas:429–508; Converter.Core.Integration.pas:351–395; Converter.Parser.DFM.pas:477–508, 654 onward

6.2 FMX generation and review signals

The component tree informs enabled analyses and FMX emission. Structural and fidelity checks surface problems before the output file is accepted.

If enabled, third-party analysis walks non-root components and their children/collection items. Data-aware analysis examines the parsed form when enabled; otherwise prior data-binding state is cleared. These analyses produce mapping and review evidence for the form being generated. FDfmParser.Convert constructs TFMXGenerator with the context and mapper and generates FMX text from parsed components. The orchestrator promotes full-text memo fields in the FMX output. The orchestrator requires nonempty output that starts with object or inherited and ends with end. This catches an empty or obviously malformed form stream; it is not a full IDE load or runtime layout test.

Source: Converter.Core.Integration.pas:360–405; Converter.Core.Integration.pas:406–408; Converter.Parser.DFM.pas:947–959; Converter.Core.Integration.pas:409–416

AuditDfmFmxFidelity scans source object names, Caption/Text values, and On... handler assignments and warns when they cannot be found in generated FMX text. The audit flags possible losses for inspection; textual presence is not proof of equivalent visual behavior. After successful conversion, ProcessDfmFile calls SaveConvertedFile, which maps the source-relative .dfm path to .fmx and writes UTF-8 with BOM on a normal run. Dry run logs a would-save issue. A failed conversion is not written as .fmx; skipped and error counts plus issues explain why.

Source: Converter.Core.Engine.pas:1811–1981, 2040–2046; Converter.Core.Engine.pas:2047–2074; Converter.Core.FileManager.pas:298–322, 422–468

7 Project output and reports

After the selected file loop, a normal run generates project companions and audits current-run output. A dry run takes an earlier branch. Reports are written only when execution reaches the report stage.

7.1 Project companions and dependencies

The project generator validates the selected project again and processes its startup, project XML, packages, deployment assets, and companions in order.

If DryRunPreview is true, Convert logs that converted source and project generation were skipped and records an informational preview issue. If false, it logs “Generating project files”, calls GenerateProject, then audits current-run output. Dry run still has an output folder and can later write reports and mapping suggestions. GenerateProject calls ValidateProjectSelection again before writing companions. It then calls CopyDPR, CopyDPROJ, CopyDPKFiles, CopyDeployProj, and CopyProjectCompanionFiles in that exact order. CopyDPR reads the selected DPR, applies TransformDPRContent, and writes a UTF-8 Delphi project source file. The transform normalizes recognized startup/splash sequencing, removes VCL themes/styles references, and can add FMXMessageBridge to uses when needed. A read/write failure becomes an issue.

Source: Converter.Core.Engine.pas:1091–1110; Converter.Core.FileManager.pas:202–265; Converter.Project.Generator.pas:1246–1260; Converter.Project.Generator.pas:517–672, 806–828

CopyDPROJ reads the selected project XML, changes DFM references and framework/form type to FMX, normalizes namespace entries, and stages referenced icon, manifest, and deployment assets where supported. It writes the target DPROJ or records an error. CopyDPKFiles enumerates package sources under the selected recursion scope, transforms recognized VCL/DFM references, and writes their source-relative paths. CopyDeployProj transforms a selected deployproj and referenced deployment paths. Per-file failures become issues. CopyProjectCompanionFiles examines project resources and selected companion extensions. It stages supported referenced assets and inventories runtime dependencies while filtering build artifacts. A copied dependency or unresolved reference is recorded for later review rather than assumed correct for every target platform.

Source: Converter.Project.Generator.pas:673–765, 829–851; Converter.Project.Generator.pas:766–906; Converter.Project.Generator.pas:907–1245

When the context marks that Windows-message patterns need a bridge, GenerateProject calls GenerateFMXMessageBridge. The unit is written to the output root only if it does not already exist, preserving user-customized output. Its message behavior still requires source review.

Source: Converter.Project.Generator.pas:1254–1259, 1261 onward

7.2 Output audit and mapping suggestions

The engine inspects generated files and may offer a mapping-pack authoring aid when the recorded issues support it.

AuditGeneratedOutput walks output .pas, .dpr, and .fmx files, skipping files older than this run's start threshold. AuditGeneratedFile checks for remaining VCL or other risky output patterns and records issues. This is a static output audit, not a Delphi compile. ExportMappingPackAuthoringSuggestions examines mapping-related issues. If it finds suitable missing/unsupported component evidence, it writes MappingPack_Authoring_Suggestions.json as a starter pack in the output folder; otherwise it writes nothing. Suggested rules require human completion and contract testing. After generation and suggestions, Convert sets FEndTime. If CreateReport is true, GenerateReport begins. In the normal GUI path ConversionThreadProc sets CreateReport to true, so the stage is enabled unless an earlier exit or exception bypasses it.

Source: Converter.Core.Engine.pas:226–736; Converter.Core.Engine.pas:737–876, 1112; Converter.Core.Engine.pas:1114–1116; MainForm.pas:2255

7.3 Report files and result classification

The report model collects results and issues, then writes its configured representations. The engine classifies completion according to errors and review needs.

GenerateReport defines TXT, HTML, and JSON filenames in the output folder, captures paths, totals, elapsed time, mapping packs and usages, detailed issues, risk categories, grouped manual-review entries, next steps, and conversion-log lines. Source identifiers and paths are protected from presentation translation. The text renderer writes VCL_to_FMX_Conversion_Report.txt, the HTML renderer writes its .html counterpart, and the JSON renderer saves .json. Selected-language translation is applied to text and HTML presentation; JSON remains structured source evidence. Text and HTML paths are returned to the form.

Source: Converter.Core.Engine.pas:2425–3008; Converter.Core.Engine.pas:3009–3035

Convert sets false for file errors or blocking issues. It returns true with a manual-review log message when review issues remain, and true with a clean message otherwise. In the GUI, ConversionThreadProc independently derives the display status from the engine counts and context issues. The outer engine handler records a fatal conversion issue and end time. The finalizer also sets end time for early returns and logs blocking preflight issues. Because some early exits happen before GenerateReport, a preflight failure may leave no report file.

Source: Converter.Core.Engine.pas:1117–1138; MainForm.pas:2312–2331; Converter.Core.Engine.pas:1139–1159

8 Completion and next action

The worker returns a completion snapshot to the UI thread. The form restores controls and presents report paths and counts; the developer must then validate the generated FMX project in Delphi.

8.1 Returning results to the form

The worker captures the engine outcome even when an exception interrupts normal processing, and the queued UI callback refreshes the application state.

After Engine.Convert, ConversionThreadProc copies report paths, processed/converted/error/skipped counts, manual and blocking totals, and formatted duration into the completion snapshot. In finally blocks it clears the published active engine under the lock, frees engine and translator, then frees the context. An exception outside Engine.Convert's own handling calls ShowError and leaves the completion status as "Conversion failed". The anonymous thread's finally block still queues a UI callback with its snapshot. If the form is not closing, the callback copies the completion snapshot into FPendingCompletion and calls CompleteConversionOnUIThread. That method frees FConversionThread, reenables controls, and calls UpdateArtifactsAfterConversion.

Source: MainForm.pas:2304–2345; MainForm.pas:2339–2349, 2205–2232; MainForm.pas:2216–2230, 1915–1923

UpdateArtifactsAfterConversion stores final counts, status, duration, target, and report paths. It updates dashboard and Issues labels. GetPreferredReportPath chooses an existing HTML report first, then TXT, and otherwise none; UpdateReportActions enables only actions backed by a real file or directory.

Source: MainForm.pas:1830–1892, 1626–1636

8.2 Reviewing, resetting, and closing

The operator can inspect outputs and reports, perform IDE and runtime validation, begin another attempt, or close once active conversion has stopped.

Open Output checks that the directory exists before calling the shell. Open Report uses the preferred existing HTML/TXT path. Print Report asks the shell to print; when direct printing is unavailable it opens the report and informs the operator. The converter's own last step is a static result and report. A developer must inspect reported blockers or manual-review items, load the generated .dproj in Delphi, compile it, and compare visual, data, and runtime behavior with the VCL source. These verification actions are external to VCL2FMXConverter, not hidden engine steps.

Source: MainForm.pas:2077–2126; Converter.Core.Engine.pas:1117–1138; MainForm.pas:1841–1892

When no worker is assigned, Reset clears source and target, restores designer-equivalent selections and option defaults, clears run artifacts, refreshes reference pages, returns to Project Scan, and focuses source entry. Reset does not delete the previous output folder. FormCloseQuery frees an already-finished thread if necessary. It refuses closure while FConversionThread remains assigned and displays a wait message. Once close is allowed, destruction stops the Help server, safely cancels/waits for any remaining worker state, removes queued worker events, frees the lock, and completes inherited destruction.

Source: MainForm.pas:1930–1940, 1585–1624; MainForm.pas:424–477, 1925–1928

Interpreting the result

A report is available only if the engine reaches GenerateReport. An early preflight return can leave no report even though a run was attempted. The converted-file count can also increase before a save succeeds, so compare it with the error and blocking-issue totals and inspect the generated files.

After a normal conversion, open the generated project in Delphi, compile it, and compare screen layout, data behavior, and runtime interactions with the VCL source application. Warnings and manual-review issues are engineering work items, not proof that the resulting FMX application is ready to ship.

Source baseline: VCL2FMXConverterV7 local checkout reviewed 18 October 2026. The archived v4 runtime flow informed the scope; the v7 sequence and citations come from v7 code.