VCL TO FMX CONVERTER

v6.0 NAVIGATOR

ENGINEERING GUIDE

Detailed Engineering and Maintenance Edition

Revision date: August 3, 2026

This manual is intended for developers, maintainers, and technical reviewers who need to understand how the converter works, how its units collaborate, and how future improvements should be made safely and globally.

Table of Contents

1. Introduction and Engineering Intent

The VCL2FMXConverter is a migration tool for Delphi applications. Its purpose is to read an existing VCL-oriented codebase and emit a first-pass FireMonkey version that is as structurally valid, compilable, and reviewable as possible.

The converter is not intended to behave like a blind search-and-replace utility. It operates more like a staged migration compiler. It discovers files and companion files, parses forms and source units, builds intermediate models, applies mapping and rewrite logic, injects compatibility behavior where needed, and emits a new FMX-oriented project structure.

The engineering intent behind the current implementation is strongly global. Improvements should be always be expressed as reusable converter rules so that each fix benefits future projects as well as the one currently under review.

1.1 Primary Engineering Goals

1. Preserve user intent so the converted application still represents the same business behavior, model and recognizable user experience as the source project.
2. Prefer valid FMX output over risky mimicry so unsupported VCL constructs do not become invalid FMX artifacts or fail to function as originally intended.
3. Always improve the converter globally so fixes that are introduced are captured in the converter itself rather than being repeated manually in each converted project.

1.2 Practical Meaning of a Successful Conversion

Layer	Description	Typical Evidence
Structural	Generated forms load and units compile.	Delphi opens the project and the code builds.

Runtime	Startup, timers, queries, dialogs, and events behave acceptably.	The converted application launches and performs real workflows.
Visual	Layout, captions, colors, and images remain usable and recognizable.	Side-by-side comparison with the VCL original is reasonable.
Maintainability	Generated code and converter logic remain understandable enough to extend.	A future maintainer can diagnose and improve the system.

2. Product Scope and Design Principles

2.1 Scope

- Delphi Pascal source files (.pas)
- VCL form definitions (.dfm), including binary forms normalized to text
- Delphi startup and project metadata (.dpr, .dproj, and related artifacts)
- Reports and operational status output

The converter also includes special handling for color translation, images, data-aware controls, grid behavior, WinAPI compatibility, media controls, and unsupported third-party patterns. Version 6.0 Navigator adds optional mapping packs so third-party conversion assumptions can live outside the core and remain transparent to the operator.

2.2 Design Principles

- Use Embarcadero FMX behavior as the reference model wherever the answer is documented or visible in Delphi source code.
- Preserve semantic meaning before preserving literal syntax.
- Prefer readable output over opaque generated code.
- Keep risky fixes centralized and global rather than improvising project-specific patches.
- Treat data-aware conversion, custom drawing, and WinAPI migration as high-risk engineering areas.

3. System Architecture Overview

The converter is organized as a layered system. Each layer has a specific responsibility, and most files belong naturally to one of these roles.

3.1 Layer Diagram

The diagram below illustrates the layering.

User Interface MainForm	➜	Engine and Orchestration Core.Engine Core.FileManager Core.Integration	➜	Parsers and Mappers Parser.DFM Parser.Pascal Mapper.Component	➜	Output Generation FMX source Project files Reports

The Advanced modules sit beside the orchestration layer and intervene when specialized rewriting is needed. This includes WinAPI compatibility, data-aware conversion, custom drawing, and unsupported component scenarios.

3.2 Responsibility Matrix

Unit Group	Primary Responsibility	Main Files
User interface	Collect options, launch conversion, and report progress without containing core conversion logic.	MainForm.pas, MainForm.fmx
Engine core	Coordinate file scanning, conversion sequencing, issue aggregation, and lifecycle control.	Converter.Core.Engine.pas, Converter.Core.FileManager.pas, Converter.Core.Types.pas
Integration	Apply global Pascal rewrites, rebuild uses clauses, inject LiveBindings, and normalize FMX behavior.	Converter.Core.Integration.pas

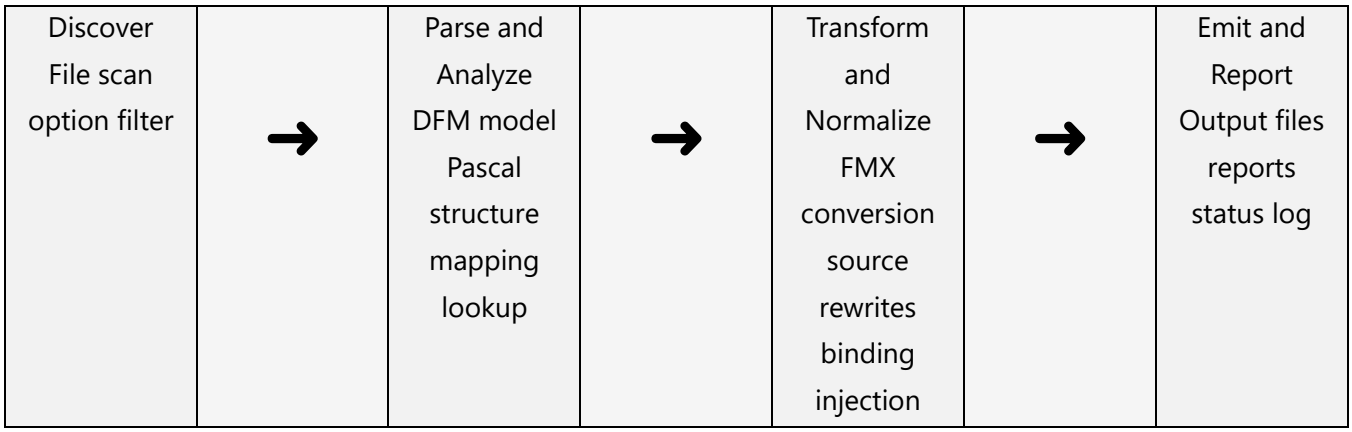
DFM processing	Parse VCL forms, preserve collections and field objects, and regenerate FMX text.	Converter.Parser.DFM.pas
Pascal processing	Parse unit structure and support source-safe rewrites.	Converter.Parser.Pascal.pas
Mapping layer	Decide FMX class and property equivalents, including fallbacks and confidence rules.	Converter.Mapper.Component.pas
Specialized conversion	Rewrite high-risk patterns such as WinAPI, DB controls, custom drawing, and unsupported components.	Converter.Advanced.DataAware.pas, Converter.Advanced.WinAPI.pas, Converter.Advanced.CriticalAreas.pas, Converter.Advanced.ThirdParty.pas
Project generation	Produce FMX-aware startup and project metadata.	Converter.Project.Generator.pas

4. End-to-End Conversion Workflow

A single conversion request proceeds through a predictable sequence. Understanding this pipeline is essential when diagnosing why an output artifact is wrong.

4.1 Workflow Diagram

This flow shows the major transformation stages from discovery through output emission.



4.2 Step-by-Step Narrative

1. The file manager discovers candidate files according to the selected source path, recursion setting, and file-type options.
2. Each artifact is routed to the correct processing branch. DFM files become component trees; Pascal files become structured source units.
3. The integration layer applies global normalization, including unit inference, VCL-to-FMX substitutions, resource rewrites, and compatibility injection.
4. Specialized modules rewrite higher-risk patterns such as WinAPI, custom drawing, media behavior, and data-aware controls.
5. Output files and reports are written, and the converted project is then validated in Delphi.

5. User Interface and Operator Flow

The operator-facing layer is intentionally simple. Its job is to collect options, start conversion work, and show progress. It should not become the home for core rewrite logic.

5.1 Main Operator Tasks

- Select the source project or root folder.
- Select the output directory.
- Choose recursion and file-scope behavior.
- Launch the conversion.
- Review the log, open the converted project, and validate the output in Delphi.

5.2 UI Responsibilities

MainForm is responsible for operator workflow, progress output, and status visibility. Conversion logic should remain in the engine, parser, integration, and mapping units unless the user experience itself needs to change. The current v6.0 interface is a tabbed operator workspace with Dashboard, Project Scan, Component Map, Property Map, Event Map, Conversion Output, and Rules. Open Report, Print Report, and Open Output Folder remain UI shell actions, while the four rule toggles are passed into explicit engine options rather than treated as decorative markers.

6. Core Engine, File Management, and Orchestration

6.1 Core Types

Converter.Core.Types defines shared conversion types, issue tracking, options, and context state. It is the vocabulary layer the rest of the converter relies on.

6.2 Engine

Converter.Core.Engine coordinates top-level conversion sequencing, lifecycle control, and status accounting. It should remain focused on orchestration rather than file-format specifics. The current report path in this layer is also where the clearer v4 report wording is decided: Blocking issues present, Manual review required, Clean conversion, Distinct files needing attention, and the HTML-first report behavior exposed through the UI.

6.3 File Management

Converter.Core.FileManager handles source discovery and output writing. It is responsible for honoring recursion options, preserving relative structure where practical, and preventing stale output from being mistaken for current output. In the current v6.0 path it also owns the practical output-reset behavior: the destination folder is created when conversion starts, and stale build subtrees such as Win32, Win64, Debug, Release, deploy, and __history are removed before regenerated files are written.

6.4 Integration

Converter.Core.Integration is the highest-leverage engineering unit in the system. It applies cross-file FMX rewrites, uses-clause rebuilding, binding injection, runtime compatibility changes, and late-stage normalization. Recent v6.0 work in this layer is also where many stability fixes now live: combo popup suppression and preservation, data-aware cleanup restoration, DisplayText refresh helpers, runtime text-metric helpers, and the narrow float-division rewrite used for FMX size and position math.

7. DFM Parsing and FMX Form Generation

7.1 Internal Model

Converter.Parser.DFM builds an intermediate component tree that can retain names, classes, properties, events, child objects, collection items, and dataset field objects. This intermediate structure is what lets the converter preserve more than literal text.

7.2 Parsing Responsibilities

The parser must correctly handle text DFMs, binary form normalization, nested objects, multiline properties, image payloads, collection items such as DBGrid columns, and dataset field definitions.

7.3 Generation Responsibilities

The generator must emit FMX-safe form text while preserving class intent, parent-child structure, collection syntax, geometry, and field object definitions.

Recent generic hardening in this layer focused on form-reader safety and FMX layout fidelity. The DFM generator now strips or remaps several common VCL-only streamed properties before emission, including KeyPreview on root forms, TMemo.ScrollBars, track and progress Position properties, root-form OnDblClick, and MaxLength on numeric FMX controls such as TNumberBox and TSpinBox. It also preserves more of the original label intent by emitting safer AutoSize, anchor, StyledSettings, and text-setting combinations so centered labels and runtime-sized labels survive FMX more faithfully. Root forms and datamodules are also kept out of unsupported-component matching so user-defined form classes do not become false blockers.
Risk Area

Area	Failure Mode	Mitigation Approach
Collections	Grid columns, menu items, or other nested items disappear.	Preserve collection objects in the intermediate model and regenerate them explicitly.
Images	Binary payloads become invalid or disappear.	Normalize image data and emit FMX-safe image payloads.
Property leaks	FMX form reader rejects VCL-only properties.	Filter or remap unsupported properties before emission.
Layout	Controls appear stacked, hidden, or mis-parented.	Preserve parent-child relationships and emit FMX-native geometry.

8. Pascal Parsing and Source Rewriting

8.1 Parser Responsibilities

Converter.Parser.Pascal identifies interface and implementation sections, class declarations, method declarations, resource directives, and other structural boundaries so that rewrites do not corrupt valid Delphi syntax.

8.2 Rewrite Objectives

- Remove or replace VCL-only unit references.
- Add the FMX units required by the rewritten code.
- Normalize resource directives from *.dfm to *.fmx.
- Preserve valid declarations and method boundaries.
- Emit FMX-safe replacements or manual-review comments for unsupported constructs.

Current generic rewrites also normalize several high-friction code patterns discovered in real projects: TMemo.Clear is rewritten to Lines.Clear, numeric Position reads and writes are adapted to FMX Value semantics where appropriate, Windows multimedia code keeps Winapi.MMSystem when wave APIs are present, thread marshaling is normalized toward public TThread.Queue and TThread.Synchronize call shapes that the installed Delphi compiler accepts, folder-only browse logic is redirected toward SelectDirectory, and generated FMX MessageDlg calls now qualify button tokens through TMsgDlgBtn so they match the FMX and System.UITypes API surface.

9. Component Mapping and Property Transformation

Converter.Mapper.Component acts as the mapping knowledge base. It determines class targets, property equivalents, and some heuristics for how a VCL control should be represented in FMX. In v6.0 it also loads deterministic JSON mapping packs, validates mapping actions, and records pack metadata such as vendor, pack name, version, confidence, and review guidance.

9.1 Mapping-Pack Architecture and JSON Files

V4.1 mapping packs are external JSON files loaded from the mapping_packs folder before conversion. They extend the mapper without placing commercial-library assumptions directly in the converter core. This boundary is important: core code should provide the loader, schema validation, action semantics, property/event integration, and reporting, while individual vendor assumptions live in separate pack files.

The schema file is docs\mapping-pack-schema-v1.json. The operator-facing guide is docs\guides\Mapping_Packs_v4.md. The V4.1 distribution includes starter JSON packs for DevExpress, TMS, Raize/Konopka, JVCL, EhLib, InfoPower, FastReport, TeeChart, VirtualTreeView, Devart, and IntraWeb/UniGUI.

Each pack declares pack metadata plus a rules array. Pack metadata identifies pack_name, pack_id, vendor, version, description, and mode. Rules can specify vcl_class, fmx_class, action, confidence or probability, notes, property mappings, event mappings, manual_review_reason, vendor, and inherited pack metadata.

The supported rule actions are convert, partial, manual_review, detect_only, and preserve. The generator must treat detect_only and manual_review as report-only decisions, partial as generated output with limitations reported, and preserve as an explicit decision that remains visible in the report.

Action behavior contract

Action	Generator Behavior	Report requirement
convert	Generate the declared FMX class.	Record pack name, source class, target class, and confidence.
partial	Generate the declared FMX class when useful.	Record limitations and require review.
manual_review	Do not generate an unsafe replacement.	Record the reason and affected component.
detect_only	Do not generate a replacement.	Record the detected class/vendor and location.
preserve	Keep the original reference only when preservation is intentional.	Record what was preserved and why it needs review.

Loader guardrails matter. Pack files should load in deterministic filename order, unsupported actions must be rejected, malformed packs should not partially activate, and duplicate mappings should resolve predictably. Low-confidence or report-only rules must never silently generate replacement FMX components.

At runtime, the mapper supplies pack metadata to the DFM conversion path so component, property, and event decisions can be reported. External pack property and event rules should flow through the same mapping paths used by built-in mappings so FindMappedProperty, FindMappedEvent, ResolvePropertyMapping, and ResolveEventMapping remain the integration points.

Reporting and test expectations

The HTML report should remain the audit trail for loaded packs, converted third-party components, partial conversions, detect-only findings, preserved components, unknown third-party-looking classes, confidence values, and manual-review reasons.

Engineering tests can use mock DFM classes to verify pack loading, rule matching, action behavior, property/event mapping, and report output without installing the vendor libraries. Those tests validate converter behavior; they do not certify vendor runtime equivalence.

VCL Class	FMX Target	Engineering Note
TForm	TForm	Root forms now participate in the explicit class matrix while still flowing through dedicated parser and generator rules.
TFrame	TFrame	Frames are treated as first-class FMX targets rather than falling into a generic form-only path.
TPanel	TPanel	Preserves visible container behavior better than TLayout.
TGroupBox	TGroupBox	Keeps captioned frame semantics visible to the user.
TMaskEdit	TEdit	Uses an FMX edit target and leaves mask-specific behavior as explicit manual review.
TStaticText	TLabel	Converts cleanly to an FMX label with caption-to-text normalization.
TFlowPanel	TFlowLayout	Uses the closest stock FMX flowing container.

TGridPanel	TGridPanelLayout	Uses the closest stock FMX grid-layout container.
TCheckListBox	TListBox	Maps to an FMX list box and flags checked-item behavior for review.
TDBGrid	TStringGrid	Requires FMX DB-aware binding support, not just a renamed class.
TDBCtrlGrid	TStringGrid	Requires structural redesign and is kept as a low-confidence substitute with manual review expectations.
TDBEdit	TEdit	Usually paired with TLinkControlToField.
TDBListBox	TListBox	Uses LiveBindings plus list/lookup review rather than pretending FMX has a direct DB-aware twin.
TDBComboBox	TComboEdit	Generated combo/data bridge is safer than a blind class rename.
TNumberBox	TNumberBox	Supported directly with numeric property remaps and FMX.NumberBox unit injection.
TSpinEdit	TSpinBox	Converted through numeric-property and value-semantics normalization.
TColorBox	TColorComboBox	Mapped to the FMX color-picker style control.
TPaintBox	TPaintBox	Supported directly when the FMX.Objects unit is present.
TRadioGroup	TRadioGroup	Handled through a reusable compatibility path that rebuilds internal FMX radio buttons safely.
TRichEdit	TMemo	Uses TMemo as the closest stock FMX text surface and leaves rich-text behavior for review.

TDrawGrid	TStringGrid	Uses an FMX string grid and marks custom drawing semantics as a follow-up concern.
TFileSaveDialog	TSaveDialog	Normalizes the Vista-style dialog to the closest stock FMX save dialog.
TLinkLabel	TLabel	Preserves visible text while leaving hyperlink behavior for review.
TFontDialog	TFontDialog	Uses the generated FMX compatibility dialog with Execute and Font semantics.
TTaskDialog	Manual review	No stock FMX task dialog exists, so this remains an explicit manual-review row.

A class rename alone is rarely enough. Mapping also depends on property filtering, event translation, runtime compatibility rewrites, and in many cases additional FMX infrastructure.

The current mapper also exports explicit RTTI-backed reference artifacts: vcl_class_inventory.json, fmx_class_inventory.json, class_mapping_matrix.json, property_mapping_matrix.json, and event_mapping_matrix.json. Those artifacts make the current mapping surface auditable and give the reporting/documentation layer a firmer source of truth than older ad hoc summaries.

These broader mappings and matrices were added only after they were verified as generic patterns across multiple independent projects. They are a reminder that release-quality coverage depends on repeated corpus testing, not on one successful application.

10. Data-Aware Conversion and LiveBindings

Data-aware conversion is one of the most sensitive parts of the converter because VCL DB controls and FMX controls differ both visually and architecturally.

10.1 Current Binding Relationship

The current preferred binding model uses an FMX data source bridge and then control-specific binding components.

Dataset TFDQuery or other dataset	→	Data Source TDataSource	→	FMX Bridge TBindSourceDB	→	Binding Link Control or DB grid binding component	→	FMX Control Edit, checkbox, or string grid

10.2 Current Preferred Binding Model

- TBindSourceDB as the bridge from a dataset-backed data source into FMX LiveBindings.
- TLinkControlToField for most edit-like controls.
- TLinkPropertyToField for property-based cases such as IsChecked.
- TBindDBGridLink for converted DB grids.

10.3 Why the Grid Needed Special Handling

A converted VCL DBGrid should not rely on the generic FMX grid-link class. The FMX platform exposes a DB-specific binding path for database grids. Without that DB-aware path, the grid may remain empty or generate the wrong columns even while the dataset itself is active.

10.4 Column Preservation

Preserving the original VCL DBGrid.Columns collection is essential. Without it, FMX tends to generate default columns for every dataset field, which can expose irrelevant fields and render memo-backed values as generic blob placeholders.

Recent stabilization in the data-aware path also tightened the hand-off between generated FMX controls and live dataset state. Generated combo setup no longer fires popup logic during form creation, external

datasource OnDataChange handlers are restored on cleanup instead of being nilled out, external navigator BeforeAction handlers are restored after teardown, and explicit TDBEdit.Field.DisplayText refresh code now survives conversion through a generated helper instead of collapsing to self-assignment.

11. Special-Case Conversion Modules

Converter.Advanced.CriticalAreas handles high-risk patterns such as custom painting, owner-draw behavior, synchronization, and message-oriented code. Converter.Advanced.WinAPI covers selected Windows compatibility rewrites. Converter.Advanced.ThirdParty provides a home for unsupported and third-party control handling.

The WinAPI module is intentionally more selective now than some earlier blanket downgrade passes. Valid Windows-targeted FMX patterns such as ShellExecute help and document launch flows, serial-port and named-pipe CreateFile and WriteFile communication paths, and surrounding process-control calls are preserved when the converter can identify them safely, while more ambiguous file and message rewrites stay on the warning or manual-review path.

These modules exist to keep the rest of the system cleaner. If a rule is especially risky or specialized, it generally belongs here rather than in a broad parser pass.

12. Project Generation and Output Artifacts

Converter.Project.Generator is responsible for making the converted project openable and buildable as an FMX application rather than leaving the user with only transformed units and form files.

The current generator also copies common companion assets more broadly than the earlier baseline, including items such as .res, .ico, and .manifest files plus selected project-referenced local assets when they can be resolved safely. It also reports blocking issues honestly, so unsupported controls or hard conversion gaps stop being presented as a clean success.

- Converted Pascal units
- Converted FMX form files
- FMX-aware startup and project metadata
- Reports and issue summaries
  [v6.0 report accounting rule] Report accounting treats Total issues as the actionable count: warnings plus grouped manual-review items plus errors. Informational notices and dry-run preview notices are reported separately so preview output does not inflate the issue total.
  The HTML manual-review section includes an explicit expansion prompt. Users click the > arrow on each grouped item to view affected locations and recommended fixes.

Timestamped snapshots are recommended so each converter iteration is preserved independently. This makes regression tracking easier and prevents accidental overwriting of known-good backup states. Before a public release, run tools\run_release_readiness_audit.ps1 and tools\run_regression_guards.ps1. Treat any blocker as a release stop rather than as a documentation-only reminder. The release audit covers shipped-tree cleanliness such as local paths, debug leftovers, packaging assets, and license presence, while the regression guards watch converter invariants that should remain true as internals evolve.

13. Validation, Maintenance, and Enhancement Guidance

A disciplined validation sequence is one of the strongest defenses against converter regressions.

1. Rebuild the converter itself.
2. Run a fresh conversion into a clean output folder.
3. Open the converted project in Delphi.
4. Resolve structural and syntax issues first.
5. Run the application and validate real workflows.
6. Review layout, colors, captions, images, and data presentation after runtime stability is in place.

Stage	What to Check	Common Defects
IDE load	Forms open and units parse.	Invalid FMX properties, malformed collections, wrong form class names.
Compile	Project and units build cleanly.	Missing uses, bad rewrites, wrong event names, visibility issues.
Runtime	Application launches and executes real logic.	Access violations, binding exceptions, canvas misuse, startup ordering bugs.
UX review	Layout and presentation fidelity.	Stacked controls, unreadable captions, wrong columns, missing images.

13.1 Conversion Contract System for v6.0

The v6.0 engineering baseline is contract driven. A conversion contract consists of a small sample Delphi project, form, or source file together with an expectation file that defines the converter output, report items, uses-clause behavior, and required validation behavior. Contracts are stored under the project contracts folder and are executed by the contract runner, which is the test utility that loads each contract, runs the converter against its sample input, and verifies that the actual result matches the expectation file. They are not runtime data for normal conversions.

[v6.0 contract baseline] The current v6.0 suite contains 227 executable expectations. The latest release-candidate contracts also cover image API cleanup and unsupported-routine try/except depth guards. New structural behavior should continue to be added behind fixtures, expected output/report rules, and regression guards.

The contract runner discovers expectation files from the contracts tree, loads the related fixture input, runs the converter against a temporary output location, and validates the generated Pascal, FMX/DFM, conversion report, status, expected issue categories, required units, absent units, required output patterns, and forbidden output patterns. This keeps the test search bounded to the contract inventory instead of comparing every production source file with every contract.

The first v6.0 contract families cover Pascal include-file analysis, Windows messaging detection and reporting, false-positive boundaries, protected and conditional uses cleanup, TMemo/TStrings collection output, TStringGrid event ordering, UTF-8 FMX output, GDI-to-FMX canvas substitutions, and whole-project integration behavior. Include handling is analysis first: include files are resolved safely from the source tree, recursive and outside-tree cases are reported, include contents are analyzed with the same encoding rules as Pascal files, and original include directives remain in generated source.

Windows messaging rules remain category based. The converter detects message APIs, WM/CM/CN/common-control families, TWM/TCM records, WndProc overrides, message declarations, WM_USER offsets, system-command handlers, and false positives. Unsafe or ambiguous behavior is reported with source location, original symbol or API, code excerpt, category, suggested FMX path, and blocking status. Standalone safe cases may be converted only when the contract proves the result is reliable for the general user base.

Future structural changes should start with a fixture and expectation before converter code is changed. This keeps v6.0 fixes reusable, prevents project-only patches, and protects the mapping packs, rewrite rules, include analysis, Windows messaging reports, and generated output from silent regression.

14. Unit-by-Unit Technical Reference

This section is a technical reference for the twelve primary converter units. It is intentionally more detailed than the earlier architectural overview because future maintainers need to know where to place a change, what each unit owns, and what kinds of edits are likely to cause regressions.

Each subsection below explains the unit's role, major structures or routines, how it interacts with the rest of the converter, and what kinds of changes belong there.

It should be noted that not all .pas files appear here. Only the main players are described. The unmentioned ones are helper files that do various background actions.

14.1 Converter.Core.Types.pas

Converter.Core.Types.pas is the converter's shared vocabulary layer. It defines the issue model, the mapping model, the options container, and the conversion context used to carry state across a run. The live v6.0 options container now exposes the real per-run flags for Critical Areas, Data Aware, ThirdParty, and WinAPI passes, so the UI, manuals, and code should stay aligned with those actual engine flags.

Aspect	Detail
Primary purpose	Provide shared types, issue objects, mapping records, and the run context so the rest of the converter can exchange structured information.
Key structures	TConversionIssue, TComponentMapping, TConversionOptions, TConversionContext, and the severity and file-type enums.
Important interactions	Used by the engine, file manager, parser, mapper, integration layer, and project generator as the common state and diagnostics model.
Change guidance	Add new context or issue fields here when a new cross-cutting capability is needed. Avoid storing transient parsing logic here; this unit should remain a stable shared contract.

14.2 Converter.Core.Engine.pas

Converter.Core.Engine.pas coordinates an entire conversion session. It owns the file manager, orchestrator, and project generator, dispatches per-file work, tracks counts, writes the report, and manages logging and

encoding-safe reads. It is also where the report wording is chosen, including Blocking issues present, Manual review required, Clean conversion, and Distinct files needing attention. Current report generation also summarizes loaded mapping packs and the mapping-pack decisions used during a run.

Aspect	Detail
Primary purpose	Orchestrate one conversion run from startup through report generation and shutdown.
Key routines	Convert, ProcessPasFile, ProcessDfmFile, GenerateReport, UILog, LogToFile, TryReadWithEncoding, DetectFileEncoding, and StripInvalidCharacters.
Important interactions	Consumes TConversionContext, delegates file discovery and save work to TFileManager, delegates transformation to TConversionOrchestrator, and delegates startup/project output to TProjectGenerator.
Change guidance	Put top-level sequencing, counters, and report decisions here. Do not bury file-format-specific rewrites here; push those down into parser, mapper, or integration code.

14.3 Converter.Core.FileManager.pas

Converter.Core.FileManager.pas is the filesystem-facing part of the converter. It scans source trees, respects file-type and recursion options, builds relative output paths, sanitizes output text, and preserves source encodings when it saves converted files. It also prepares the output tree at run time, including cleanup of stale build artifacts before regenerated files and reports are written.

Aspect	Detail
Primary purpose	Discover source files and write converted output safely and predictably.
Key routines	Reset, PrepareOutput, MatchesSelectedFileTypes, BuildOutputFileName, SanitizeOutputText, SaveTextPreservingSourceEncoding, and SaveConvertedFile.

Important interactions	Uses TConversionContext for source/output settings and issue reporting. Feeds discovered files into the engine and writes transformed artifacts after integration completes.
Change guidance	File path policy, output filtering, and encoding-preservation rules belong here. Avoid mixing semantic code rewrites into this unit.

14.4 Converter.Parser.Pascal.pas

Converter.Parser.Pascal.pas gives the converter structural awareness over Delphi source units. It tracks interface, implementation, initialization, and finalization regions; class declarations; method declarations and bodies; and uses clauses so later rewrites can be inserted safely.

Aspect	Detail
Primary purpose	Parse Pascal units into a structure detailed enough for safe transformations.
Key structures	TPascalMethod, TPascalClass, TPascalUnit, and TPascalParser.
Important routines	ParseUnit, ParseUsesClause, ParseMethod, ParseTypeSection, ParseImplementationUses, FindMethod, and FindMethodsByPattern.
Change guidance	Any change that affects method detection, class boundaries, or safe insertion points belongs here. This is the right place to improve syntax awareness, not to make FMX-specific business decisions.

14.5 Converter.Parser.DFM.pas

Converter.Parser.DFM.pas parses VCL form definitions into an internal component tree and regenerates them as FMX form text. It handles nested components, collections, multiline properties, dataset field objects, property filtering, image payloads, and many layout-oriented translation rules.

Aspect	Detail

Primary purpose	Convert VCL DFM structure into FMX form structure while preserving hierarchy and semantics.
Key responsibilities	Child object parsing, collection preservation, dataset field emission, string quoting, property filtering, root-form safety, numeric-property suppression, status bar panel generation, root-form mouse-event adaptation, label AutoSize and anchor handling, StyledSettings emission, and size and font translation.
Important interactions	Feeds on mapping information, collaborates with integration-generated Pascal code, and must remain consistent with rewritten field declarations and event names.
Change guidance	FMX form-reader errors are often rooted here. Put DFM-to-FMX property and structure fixes here instead of compensating later in manual output edits.

14.6 Converter.Mapper.Component.pas

Converter.Mapper.Component.pas is the mapping knowledge base. It stores built-in mappings, loads FMX component knowledge, researches likely class matches, and produces property and event mappings for later passes.

Recent mapping additions in this unit broaden the standard class surface materially. The mapper now includes explicit rows for TForm, TFrame, TMaskEdit, TStaticText, TFlowPanel, TGridPanel, TCheckListBox, TDBListBox, TRichEdit, TDrawGrid, TDBCtrlGrid, TFileSaveDialog, and TLinkLabel, along with manual-review rows for TTaskDialog, TClientDataSet, TQuery, TADODataSet, and TSQLDataSet. Property and event knowledge now lives in explicit matrix rows instead of name matching alone.

This unit also exports RTTI inventories and explicit class, property, and event matrices so the live mapper surface can be audited directly from the compiled converter. If those matrix artifacts are missing from the expected build folders, the documentation tooling can reconstruct the current surface from Converter.Mapper.Component.pas. Keep property-level rules such as ScrollBars-to-ShowScrollBars, numeric Min and Max translation, masked-edit review paths, and picker-event renames centralized here.

Aspect	Detail
Primary purpose	Translate VCL control identity into the most appropriate FMX target and supporting mapping metadata.
Key structures	TComponentResearch, TFMXComponentInfo, and TComponentMapper.
Important routines	LoadBuiltInMappings, LoadFMXComponentCatalog, ResearchVCLComponent, FindFMXMatch, CalculateMatchScore, and ResearchPropertyMapping.
Change guidance	Class-name mapping, confidence, and property/event relationship knowledge belong here. Runtime behavior patches usually belong in integration, not in the mapper.

14.7 Converter.Core.Integration.pas

Converter.Core.Integration.pas is the global rewrite engine and the highest-leverage unit in the project. It repairs uses clauses, injects helper routines, adapts runtime behavior, preserves startup semantics, stabilizes event timing, and applies category-wide FMX normalization rules discovered during real-world testing. Recent stabilization work in this unit also preserves and chains existing handlers when generated FMX support needs to hook lifecycle or data-aware behavior, instead of overwriting those handlers blindly.

Recent generic behaviors in this unit include TMemo.Clear to Lines.Clear rewrites, Position and Value normalization for slider-style controls, Winapi.MMSystem preservation for waveOut code, public TThread.Queue and TThread.Synchronize normalization, plain ShowMessage preservation, qualified FMX MessageDlg buttons, and folder-picker helper insertion. The same layer also handles root-form double-click adaptation, runtime font-size helpers that release StyledSettings.Size, VCL-style text-layout helpers for label stacking, and DisplayText refresh helpers for manually bound edits. Data-aware stability work here suppresses combo popup logic during startup, preserves combo OnClosePopup behavior, and chains existing OnDestroy, OnCalcFields, dataset AfterOpen, datasource OnDataChange, and navigator BeforeAction handlers when FMX support needs to hook them.

Aspect	Detail
Primary purpose	Apply global FMX compatibility and behavior rewrites across generated Pascal output.

Key responsibilities	Uses-clause reconstruction, helper injection, startup adaptation, typed Pascal rewrites, grid and combo support, data-aware edit and display synchronization, media cleanup, graphics scene safety, thread marshaling normalization, and many post-processing rules.
Important interactions	Consumes parser and mapper output, emits code that must stay in sync with FMX forms, and frequently coordinates with DataAware and Project.Generator behavior.
Change guidance	Most global fixes land here. Because this unit affects many projects at once, every change should be tied to a repeatable pattern and regression-tested against previously fixed paths.

14.8 Converter.Advanced.DataAware.pas

Converter.Advanced.DataAware.pas exists because VCL data-aware controls do not map cleanly to FMX. It classifies DB-aware controls, chooses the right binding strategy, and supports the integration layer when a control must become a direct FMX control, a LiveBindings-driven control, a grid bridge, or a special combo or navigator path. The current path uses normalized class matching with mapper-backed ancestry instead of loose substring checks, which makes DB-aware detection more trustworthy across derived controls.

Aspect	Detail
Primary purpose	Provide specialized knowledge about data-aware control conversion.
Key focus areas	DB edits, DB combo boxes, DB grids, navigator controls, and the binding metadata needed for runtime hookup.
Important interactions	Works closely with the integration layer, mapper, and generated FMX bindings to preserve database-driven workflows.
Change guidance	If a problem is fundamentally about DB-aware control semantics, binding type, or dataset interaction, start here before changing more general parser logic.

14.9 Converter.Advanced.WinAPI.pas

Converter.Advanced.WinAPI.pas isolates Windows-specific rewrite logic. It exists so WinAPI-dependent source patterns can be preserved, adapted, or constrained without polluting the general parser and integration code with platform-specific branches.

Aspect	Detail
Primary purpose	Handle Windows API compatibility and rewrite patterns during conversion.
Typical responsibilities	ShellExecute preservation, Windows-unit handling, serial, named-pipe, and device CreateFile and WriteFile preservation, process-control preservation, message or API call normalization, and selected compatibility substitutions.
Important interactions	Influences uses-clause requirements and any generated code that still needs explicit Windows units in FMX projects.
Change guidance	Keep platform-specific logic here when it is truly Windows-focused. General FMX startup or event logic should still live in integration.

14.10 Converter.Advanced.CriticalAreas.pas

Converter.Advanced.CriticalAreas.pas is a reserved module for high-risk conversion categories that do not belong in broad, always-on parser passes. It provides a safer home for patterns such as custom drawing, owner-draw style behavior, synchronization concerns, or other specialized logic that can destabilize output if handled too casually.

Aspect	Detail
Primary purpose	Contain risky or specialized conversion behavior in a bounded unit.
Typical use	Future expansion for difficult painting, synchronization, or message-driven scenarios that require isolated logic.

Important interactions	Supports the integration layer by keeping dangerous rules out of general-purpose code paths until they are stable.
Change guidance	When a fix is powerful but risky and should not immediately become a blanket global parser rule, this is an appropriate staging area.

14.11 Converter.Advanced.ThirdParty.pas

Converter.Advanced.ThirdParty.pas provides a controlled home for unsupported or third-party component handling. Real-world Delphi projects often depend on controls that do not have first-party FMX equivalents, so the converter needs a place to record approximations, fallbacks, or explicit manual-review guidance. This unit remains primarily a detection and reporting layer; pack-driven replacement rules belong in the mapper and DFM generation path.

Aspect	Detail
Primary purpose	Support third-party and unsupported component conversion strategies.
Typical responsibilities	Fallback mapping, warning generation, or controlled substitutions for controls outside the built-in VCL/FMXX mapping set.
Important interactions	Works with the mapper and integration layer to avoid silent failure when a control class is recognized as nonstandard.
Change guidance	Put vendor-specific or unsupported-control logic here instead of hard-coding it into broad parser decisions.

14.12 Converter.Project.Generator.pas

Converter.Project.Generator.pas makes the final output openable and runnable in Delphi. It locates original project files, transforms DPR and DPROJ content, adapts FMX startup semantics, and copies or regenerates the project artifacts needed to load the converted application. The current path breaks DPR startup handling

into smaller helper routines for startup normalization, splash-sequence detection, Application.Run placement, and immediate CreateForm/Show cleanup, which makes startup fixes easier to reason about.

This unit now carries more responsibility for practical project portability. It should preserve the source Application.CreateForm list where appropriate, normalize namespace settings for FMX, inventory runtime-support files from the source root and existing build-output folders, and stage selected companions into the FMX output directory. It should also report which support files were staged versus merely found, turn serious unsupported conditions into blocking results, and keep public source distributions clean enough to ship with a root LICENSE.txt and without local-machine release noise.

Aspect	Detail
Primary purpose	Generate FMX-aware project startup and project metadata.
Key routines	FindOriginalDPR, FindOriginalDPROJ, TransformDPRContent, TransformDPROJContent, TransformDeployProjContent, companion-asset copy logic, and GenerateProject.
Important interactions	Depends on the overall conversion context and must stay aligned with startup assumptions introduced by integration logic.
Change guidance	Fix splash handling, Application.CreateForm ordering, project-search rules, and other project-level semantics here rather than inside individual generated target units.

15. Windows Messaging: Architecture and Implementation

This section documents the design decisions, data flow, and implementation details for the Windows Messaging conversion subsystem added in v6.0. It covers the five files modified, the three-tier detection model, the FMXMessageBridge infrastructure, and all regression-protected invariants the engineering team must maintain.

15.1 Background: Why VCL Messaging Cannot Be Ported Directly

VCL is architecturally a thin object-oriented wrapper over Win32. Windows messages — integers dispatched through a per-window HWND queue — are a first-class extensibility mechanism in VCL. Components communicate with WM_USER offsets, forms intercept system events by overriding WndProc, and some patterns depend on the blocking semantics of SendMessage versus the queued semantics of PostMessage.

FireMonkey was designed from the ground up for cross-platform deployment. It has no HWND, no per-window message queue, and no message pump exposed to user code. Embarcadero's replacement architecture has three layers:

- First-class events (OnResize, OnActivate, etc.) for the system messages VCL most commonly handled.
- TMessageManager in System.Messaging — a publish/subscribe broker using strongly-typed message objects — for application-level inter-component communication.
- Platform services (IFMXApplicationEventService and others) for the small set of OS-level hooks that WndProc overrides typically targeted.

The converter maps VCL patterns to these three layers automatically where possible, and flags items for manual review where architectural redesign is required.

15.2 Detection Model: Three Tiers

The message detection and conversion logic lives in Converter.Advanced.CriticalAreas.pas, ConvertMessageHandlers procedure. Detection is applied line-by-line to the Pascal source after initial rewriting passes have completed.

15.2.1 Tier 1 — Standard WM_ Handler Declarations

Detection regex (interface section scan):

```
^\s*procedure\s+\w+\s*\(\s*var\s+\w+\s*:\s*TMessage\s*\)\s*;\s*message\s+WM_\w+
```

The scanner identifies procedure declarations with the message WM_XXX keyword in the class interface. On detection:

- The message keyword and constant are stripped from the declaration.
- A lookup against FMessageMap (populated in the constructor from a hardcoded VCL-to-FMX event name table) resolves the WM_ constant to its FMX event equivalent where known.
- The implementation body is replaced with a comment block: { FMX: WM_XXX -> use YYY event handler }.
- Severity: csInfo. No NeedsFMXMessageBridge flag set.

> ℹ **Design note:** FMessageMap is populated once in the TWinAPIConverter constructor. If a WM_ constant is not in the map, the comment guidance falls back to the generic 'use TMessageManager' wording. Extending the map does not require changes elsewhere.

15.2.2 Tier 2 — WM_USER Custom Messages

Detection regex (implementation section scan):

```
// Matches: const WM_MYEVENT = WM_USER + N;
^\s*\w+\s*=\s*WM_USER\s*\+\s*\d+\s*;

// Also matches handler declarations:
^\s*procedure\s+(\w+)\s*\(\s*var\s+\w+\s*:.*\)\s*;\s*message\s+(WM_USER.*)
```

On detection, the converter:

- Derives a Pascal class name from the message name (e.g. WM_MYEVENT → TMyEventMsg).
- Appends a TMessageManager.SubscribeToMessage call to the FormCreate method. If FormCreate does not exist, a stub is injected.
- Appends a TMessageManager.DefaultManager.Unsubscribe call to FormDestroy. Same injection rule applies.
- Sets FContext.NeedsFMXMessageBridge := True on the conversion context.
- Raises severity csManualReview — the generated handler body is a stub requiring developer completion.

The SubscribeToMessage injection uses a two-pass approach: the first pass collects all custom message names, the second builds and injects the subscription block after all other rewrites have completed. This ordering ensures FormCreate is already present before injection attempts to locate it.

15.2.3 Tier 3 — WndProc Overrides and Message Pump Calls

Detection patterns:

```
// WndProc override:
^\s*procedure\s+WndProc\s*\(\s*var\s+Message\s*:\s*TMessage\s*\)

// Message pump:
\b(PeekMessage|GetMessage|TranslateMessage|DispatchMessage)\s*\(
```

WndProc overrides are the most architecturally significant pattern. The converter:

- Wraps the entire method body (from begin to matching end) in { } comment delimiters.
- Prepends a structured comment block explaining the FMX alternatives.
- Does not attempt to rewrite — the semantics are too varied for automated transformation.
- Raises csManualReview.

Message pump calls (PeekMessage, GetMessage, TranslateMessage, DispatchMessage) are simpler: the call line is replaced with a comment explaining that FMX owns its event loop. Raises csManualReview.

15.3 ConvertMessageAPI — WinAPI Layer (Converter.Advanced.WinAPI)

A parallel detection pass runs in TWinAPIConverter.ConvertMessageAPI. This handles SendMessage, PostMessage, and related Win32 API calls that appear in the implementation body (as opposed to the message handler declarations handled by CriticalAreas).

The function receives a single line and the extracted API name. As of v6.0 the branch structure is:

```
if ContainsText(APIName, 'SendMessage') then
  // Sync dispatch — generate TMessageManager.SendMessage stub
  // Sub-branch: WM_USER constant present vs absent
else if ContainsText(APIName, 'PostMessage') then
  // Async dispatch — generate TThread.Queue + TMessageManager stub
  // Sub-branch: WM_ constant present vs absent
else if ContainsText(APIName, 'PeekMessage') or ... then
  // Message pump removal comment
else
  // Generic TMessageManager guidance
```

⚠ **v6.0 fix:** Prior to the v6.0 dead-code fix, SendMessage and PostMessage were combined in a single if branch (ContainsText(APIName, 'SendMessage') or ContainsText(APIName, 'PostMessage')). This meant the else if PostMessage branch beneath it was unreachable — PostMessage was silently handled by the SendMessage path and received sync-dispatch guidance instead of async. The fix separates the two APIs into distinct arms. SendMessage generates a synchronous TMessageManager stub. PostMessage generates a TThread.Queue wrapper stub to preserve the async semantics of the original call.

15.4 System.Messaging Uses Clause Injection (Converter.Rewrite.UsesClause)

The converter must add System.Messaging to the output unit's uses clauses whenever TMessageManager code has been generated. This is handled by NeedsSystemMessaging detection in the uses clause rewriter.

Detection (fires after all rewrite passes):

```
NeedsSystemMessaging :=
  (Pos('TMessageManager', AnalysisCode) > 0) or
  (Pos('SubscribeToMessage', AnalysisCode) > 0) or
  (Pos('System.Messaging', AnalysisCode) > 0);
```

Injection targets both the interface uses clause (lines 741–742) and the implementation uses clause (lines 805–806). The injection is idempotent — if System.Messaging is already present in either clause, UnitList.IndexOf returns a non-negative index and the Add is skipped.

ℹ **Design note:** AnalysisCode is the full post-rewrite Pascal text of the unit. Scanning the final text (rather than tracking a flag through each pass) makes the detection robust to any rewrite path that might generate TMessageManager references — including future passes added by other contributors.

15.5 FMXMessageBridge Infrastructure (Converter.Project.Generator)

15.5.1 NeedsFMXMessageBridge Flag

FContext.NeedsFMXMessageBridge is a Boolean property on TConversionContext (Converter.Core.Types). It is set to True by ConvertMessageHandlers when a Tier 2 (WM_USER) pattern is found. The project generator reads this flag after all unit conversions are complete.

Declaration in TConversionContext:

```
private
  FNeedsFMXMessageBridge: Boolean;
public
  property NeedsFMXMessageBridge: Boolean
    read FNeedsFMXMessageBridge write FNeedsFMXMessageBridge;
```

15.5.2 GenerateFMXMessageBridge Procedure

Called from the project generator's main output loop when NeedsFMXMessageBridge is True:

```
if FContext.NeedsFMXMessageBridge then
  GenerateFMXMessageBridge;
```

The procedure:

- Constructs the output path: TPath.Combine(FContext.Options.OutputPath, 'FMXMessageBridge.pas').
- Checks if the file already exists. If so, skips generation entirely — never overwrites.
- Generates the file content as a TStringList and saves to disk.

Generated file contents:

- TFMXUserMessage — abstract base class inheriting from TMessage (System.Messaging).
- FMXSendMessage — synchronous dispatch helper calling TMessageManager.DefaultManager.SendMessage.
- FMXPostMessage — asynchronous dispatch helper wrapping the SendMessage call in TThread.Queue(nil, procedure begin ... end).
- Full usage guide in block comments at the top of the file.

15.5.3 DPR Uses Clause Injection

The project .dpr file must reference FMXMessageBridge.pas for the unit to be compiled into the project. Injection is handled in TransformDPRContent:

```
if FContext.NeedsFMXMessageBridge and
   not ContainsText(Result, 'FMXMessageBridge') then
begin
  Result := TRegEx.Replace(Result,
    '(uses\s+[^;]+?)(;)',
    '$1,' + sLineBreak + '  FMXMessageBridge in ''FMXMessageBridge.pas''$2',
    [roSingleLine]);
end;
```

The ContainsText guard makes the injection idempotent — running the converter a second time on a project that already has FMXMessageBridge in its .dpr will not duplicate the entry.

15.6 Files Modified in v6.0

File	Changes
Converter.Advanced.CriticalAreas	ConvertMessageHandlers completely rewritten with three-tier detection. GenerateFMXMessageHandlers updated.
Converter.Advanced.WinAPI	ConvertMessageAPI rewritten with separate SendMessage / PostMessage arms. Dead-code PostMessage branch eliminated.
Converter.Rewrite.UsesClause	NeedsSystemMessaging detection and System.Messaging injection added to both uses clause rewrite paths.
Converter.Core.Types	FNeedsFMXMessageBridge field and NeedsFMXMessageBridge property added to TConversionContext.
Converter.Project.Generator	GenerateFMXMessageBridge procedure added (private, declared and implemented). FMXMessageBridge DPR injection added to TransformDPRContent.
Converter.Core.Integration	FixImplementationSection whitelist extended: type, const, var, resourcestring, class, end, end. added as valid first tokens after implementation.

15.7 Regression-Protected Areas

The following areas must not be modified without a documented reason reviewed by the engineering team. They represent either previously debugged fixes or architectural invariants that other subsystems depend on.

⛔ **REGRESSION PROTECTED:** SplitTrailingBlockComment — private TAutoFixRewriter method with {$ guard. Handles Apply loop trailing comment split/re-attach. Do not modify signature, guard, or TrailingComment variable usage.

⛔ **REGRESSION PROTECTED:** FixImplementationSection — structural check in Converter.Core.Integration. Whitelist now includes: uses, procedure, function, constructor, destructor, initialization, finalization, type, const, var, resourcestring, class, end, end. — do not narrow this list without verifying against the full test suite of real-world VCL projects.

⛔ **REGRESSION PROTECTED:** NeedTrackBarCompat / NeedProgressBarCompat / NeedSpinBoxCompat — use real detection logic, not hardcoded False. AudioManager duplicate wrapper suppression present. Do not replace with stubs.

⛔ **REGRESSION PROTECTED:** AudioManager pre-exit rewrite pass — rewrites bare TTrackBar references in files that import AudioManager without receiving a wrapper injection. Located in Converter.Rewrite.Compatibility. Do not remove.

⛔ **REGRESSION PROTECTED:** RegisterFmxClasses removal — the injection of RegisterFmxClasses was removed because it caused runtime error 217. It must not be re-added.

⛔ **REGRESSION PROTECTED:** FMXMessageBridge generation and DPR injection — idempotent by design (ContainsText guard + file existence check). Do not remove these guards.

⛔ **REGRESSION PROTECTED:** UILog and AddIssue overloads in Converter.Core.Engine and Converter.Core.Types — these are legitimate overloads, not duplicates. Do not remove either implementation.

15.8 Known Limitations and Future Work

15.8.1 Not Converted — Manual FMX Decisions Required

- ScaleBy calls — no FMX equivalent. Flagged for manual review.
- WMSysCommand handlers — Tier 3 (WndProc), preserved as comment blocks.
- DoubleBuffered and RecreateWnd — VCL-only properties, flagged for manual review.
- Status bar panel access — flagged for manual review.
- Data-aware property wiring — LiveBindings conversion is handled by a separate subsystem.
- OnNotify / MediaPlayer notification — no automated mapping.

15.8.2 Potential Future Enhancements

- Platform service mapping: IFMXApplicationEventService could be suggested automatically for the most common WndProc override patterns (WM_ACTIVATEAPP, WM_QUERYENDSESSION, etc.).
- FMessageMap extension: the VCL-to-FMX event name table could be expanded from the current set of most-common mappings to cover edge cases.
- Typed message property scaffolding: when lParam or wParam usage is detected in a WM_USER handler body, the generator could scaffold typed properties on the generated message class.

Appendix A. File Inventory

File	Engineering Role
MainForm.pas / MainForm.fmx	Operator-facing front end and progress display.
Converter.Core.Types.pas	Shared types, options, issue model, and context.
Converter.Core.Engine.pas	Top-level conversion sequencing and status control.
Converter.Core.FileManager.pas	File discovery and output writing.
Converter.Core.Integration.pas	Global rewrite and FMX normalization layer.
Converter.Parser.DFM.pas	DFM parsing and FMX regeneration.
Converter.Parser.Pascal.pas	Pascal structure parsing and source-safe rewriting.
Converter.Mapper.Component.pas	Class and property mapping knowledge base.
Converter.Advanced.DataAware.pas	Detection and support metadata for DB-aware control conversion.
Converter.Project.Generator.pas	FMX-aware project startup and project generation.

Appendix B. Operational Checklists

Before Editing Converter Logic

- Confirm which stage is making the wrong decision.
- Confirm whether the defect is structural, compile-time, runtime, or visual.
- Gather the original VCL artifact and the generated FMX output.
- Prefer a documented global rule over a speculative one-off patch.

Before Declaring a Fix Complete

- Rebuild the converter.
- Run a fresh conversion into an empty output directory.
- Open the generated project in Delphi.
- Compile and run the converted application.
- Check for regressions in previously stabilized forms or units.

Appendix C. Current Limitations and Forward Work

- Owner-draw and custom paint logic often require manual review or targeted FMX redesign.
- Some data-aware controls still need conservative handling because there is no safe generic FMX substitute.
- Third-party components require either explicit mapping knowledge or a disciplined fallback strategy.
- Visual fidelity often converges only after structural and runtime validity are already achieved.

Appendix D. Making Code Changes Using Codex

Overview

Codex is an AI coding assistant, offered with ChatGPT subscription by OpenAI, that operates as an interactive engineering collaborator. It can inspect source code, search a codebase, explain architecture, analyze compiler and runtime failures, revise documentation, apply targeted edits, and run selected local tools, including the compiler (dcc32) and MSBuild when permissions allow. Making changes to the converter is a highly complicated task, which branches to many areas and should only be attempted by knowledgeable and experienced programmers. If you wish to make changes to the converter or regular code, it is suggested to use Codex for assistance. Codex far out performs its competitors with results and capabilitites and understanding of a project.

In this workflow, Codex is most useful as a developer-side acceleration tool for:

- debugging
- regression analysis
- code review
- refactoring
- documentation maintenance
- release-readiness auditing
- repetitive engineering tasks that benefit from large-context inspection

Operational Model

Codex does not work as a standalone compiler or as a general restricted/unrestricted remote-control system. It works through a session-based shared workspace model.

That model typically includes (with appropriate permissions for the development system):

- access to the active project tree
- access to selected local tools
- access to a shell or terminal
- access to attached images or documents if relevant
- conversation context that preserves task continuity during the session

Codex builds its understanding from the files it can read in the active session. It does not inherently "know" the project beyond what is available in the current workspace and what the developer provides during the interaction.

Connection To The Developer Environment

- Codex is connected to the developer environment through its host application and tool layer. In this environment, Codex can:
- read local files in the active workspace
- edit allowed local files
- run shell commands
- inspect reports, logs, and screenshots
- create generated outputs such as notes, reports, or patch files

Codex does not automatically gain unrestricted access to the entire machine. Access is mediated by the user, the host environment, tool permissions, sandbox rules, and any escalation/approval model configured for the session.

What Codex Can Place On The Computer

Codex can create or modify local artifacts when that is part of the requested task. Typical outputs include:

- modified source files
- updated documentation
- generated reports
- audit notes
- backup archives
- generated output files used by the project
- a sandbox for coding and staging of code modifications

Codex does not independently install background services or general-purpose resident tooling as part of normal usage. It works through the session and the files it is asked to manage.

What Codex May Store Or Rely On

Codex may rely on:

- the current thread or conversation context
- the active workspace files
- generated local artifacts from the task itself

Important qualification:

The exact handling of prompts, conversation history, telemetry, retention, and any cloud-side processing depends on the deployment model and host product configuration. Engineering, compliance, privacy, and

security teams should consult the official OpenAI or host-platform documentation for authoritative retention and policy details.

Expected Developer Experience

Developers should expect Codex to function as a fast, context-building assistant rather than as an autonomous replacement for engineering judgment.

In a healthy workflow, Codex will:

- inspect relevant files
- identify likely causes or implementation paths
- explain its reasoning
- apply focused changes when requested
- summarize the outcome and remaining verification needs

Codex performs best when requests include:

- file paths
- exact compiler or runtime error text
- expected behavior
- actual observed behavior
- constraints on scope and risk

Effective Prompting Guidance

Codex responds best to direct, bounded engineering requests. Good prompts are concrete and outcome-oriented.

Examples:

- "Inspect this compiler error and identify the root cause."
- "Fix only the generated FMX translation problem in this unit."
- "Review this change for regressions. Findings first."
- "Update this guide without breaking formatting or graphics."
- "Do a readiness sweep and report only consequential issues."

Useful prompt ingredients include:

- exact file names
- absolute or clearly scoped paths
- exact error messages

- the last known good behavior
- scope controls such as "do not edit unrelated files" or "no code changes, report only"

What Developers Should Tell Codex

To get reliable help, developers should tell Codex:

- what the problem is
- where the problem is
- what the code should do
- what happened instead
- what level of intervention is wanted

Examples of high-value instructions:

- "Analyze only. Do not patch anything."
- "Fix this, but do not widen scope."
- "Use the live converter code, not stale generated output."
- "Back up the project after the fix."
- "Treat this as a release-readiness review and report only consequential items."

Capabilities

- Codex is particularly effective at:
- tracing compiler failures back to generator logic
- finding regressions introduced by shared converter rules
- inspecting large Pascal units for risky rewrites
- preparing release notes and engineering documentation
- checking packaging and distribution logic
- validating that cleanup code restores external/shared handlers correctly
- comparing generated output against source intent

Limitations

Codex has important technical and operational limitations:

- it can make incorrect inferences
- it can miss project-specific intent that is not represented in code or documentation
- it is limited by the files and tools available in the session
- it may be blocked by sandboxing, permissions, or unavailable compilers
- it cannot guarantee runtime correctness without actual rebuild and execution

- it is not a substitute for QA, release testing, or human review

Risk Management Guidance

Codex should be integrated into engineering practice with explicit controls:

- require focused scope on risky fixes
- require real build/test verification after code changes
- use backups before significant shared-rule changes, a very important rule
- treat generated output and release packaging as separate validation targets
- do not accept "compiles" as full proof of runtime correctness until everything is tested

For release work, Codex is most reliable when used as part of a disciplined cycle:

- inspect the code
- make narrow generic fixes
- rebuild
- run targeted regression tests
- perform readiness and packaging review

Security And Privacy Considerations

Codex should be treated as a tool operating inside the developer environment, not as an invisible background observer. It works on the files and inputs made available to it. However, engineering teams should avoid giving Codex unnecessary secrets or sensitive material unless the deployment model has been reviewed and approved for that data type.

Teams should define local policy for:

- source-code sensitivity
- secrets handling
- customer data exposure
- conversation retention expectations
- audit and approval practices for tool-driven changes

Bottom Line

Codex is a practical AI engineering assistant that can materially reduce effort in debugging, converter maintenance, documentation work, and release review. It is most valuable when used as a disciplined collaborator: clear request, narrow scope, real verification, and human release ownership. It should accelerate engineering judgment, not replace it. Codex us offered by subscription of ChatGPT and requires at least a ‘Plus’ subscription, priced at US $20.00 per month. If you code a lot, this is a worthwhile investment.

Index

This manual index is intentionally simple and uses section references rather than page references so it remains stable while the document is still evolving.

Manual-Review Comment Guardrails

Manual-review output must be bounded. Rules that flag unsafe VCL or WinAPI behavior may comment the affected declaration, statement, or method body, but must never consume Delphi structural boundaries such as visibility sections, var/const/type sections, implementation, uses clauses, initialization/finalization, or the unit terminator.

Unsupported message-handler declarations must recognize Delphi method directives after the parameter list, including message WM_*, override, virtual, dynamic, reintroduce, abstract, overload, stdcall, cdecl, safecall, and register.

Regression tests should include VCL message handlers, nested handler bodies, malformed or multi-line WinAPI calls, UTF-8 comments, Windows-1252 comments, and representative non-English comments. The generated Pascal must retain an active final end. and must not prefix normal unit structure with FMX manual review.

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

Architecture completed through V6.0

Report data is owned by a shared model and rendered independently as text, HTML, and deterministic vcl2fmx.report.v1 JSON. DFM parsing and FMX generation are separate units. Component mapping resolution and the static mapping catalog are separate units. The legacy oversized conversion and reporting procedures now have guarded decomposition seams.

Mapping-pack discipline

All 22 bundled packs and 493 rules declare vcl2fmx.mapping-pack.v1. Runtime and command-line validation reject unsupported schemas and actions, malformed classes, invalid confidence, duplicate pack identifiers, and class collisions. Mapping-gap export produces a complete conservative starter pack.

Validation baseline

V6.0 is guarded by 227 executable conversion contracts, 66 regression guards, 63 acceptance checks, mapping-pack positive and negative suites, French beta regression, mapping smoke coverage, Win32 and Win64 Release builds, and compilation of every eligible generated contract fixture. Synthetic malformed, missing, external, dry-run, and manual-review fixtures explicitly declare when compilation is not applicable.

V6.0 conversion corrections

Compiled fixtures now enforce Pascal unit and file-name agreement in isolated staging, normalize conditional and implementation uses clauses, add required FMX and System units, preserve case-insensitive DFM property conversion, translate VCL alpha blend and dialog border assignments, and use FMX-compatible canvas text and rectangle helpers.