VCL2FMXConverter v7.0 Atlas

Conversion Contracts Guide

Understanding, running and maintaining conversion contracts

Delphi VCL to FireMonkey conversion

Last changed October 18, 2026

V7 Atlas documentation

Contents

1. Purpose of This Guide....3
2. What a Contract Is....3
What a Contract Is Not....3
3. Why v7.0 Uses Contracts....4
4. How the Contract Runner Works....4
Normal Conversion Versus Contract Execution....4
5. Contract Folder Structure....4
6. Anatomy of an Expectation File....5
Example: A Simple Uses-Clause Guard....6
Example: A Positive Color Contract....6
7. What Contracts Prove....6
8. What Contracts Do Not Prove....6
9. How Contracts Interact With Mapping Packs....7
10. How Contracts Interact With Reports....7
11. How to Add a New Contract....7
Example Workflow for a New Bug....8
12. Good Contract Design....8
13. Weak Contracts and How to Repair Them....8
14. Public Source Distribution Notes....8
15. Recommended Review Instruction for Outside AI Reviewers....9
16. Current v7.0 Contract Baseline....9
17. Summary....9
Inherited maintenance changes....9
V7 Atlas language support and operating safeguards....10
Choosing a language....10
Reading translated Help....10
Editing and adding translations....10
Translation review....10
Source and output checks....10
Text encoding....11

1. Purpose of This Guide

This guide explains the v7.0 Atlas conversion-contract system in practical terms: what contracts are, what they contain, how the runner uses them, what they prove, what they do not prove, and how new contracts should be written when the converter learns a new rule.

The contract system exists because a source-to-source converter can look correct on one project and still fail in a different code shape. A contract captures the shape of a known problem and turns it into an executable rule. Once a problem has a contract, the converter either keeps satisfying that rule or the contract runner fails.

Plain-language definition: A conversion contract is a small source input case plus a written expectation file. The input case creates a specific conversion problem. The expectation file states exactly what the converted output and report must contain, and what must not appear.

2. What a Contract Is

A VCL2FMX conversion contract has two parts.

1. A fixture: one or more source files such as .pas, .dfm, .dpr, .inc, or project companion files. The fixture deliberately contains one behavior the converter must handle.
2. An expectation: a matching .expected.json file that records the expected conversion status, generated-code patterns, forbidden patterns, report messages, and uses-clause results.

The fixture is the test input. The expectation is the rule. Together, they become an executable contract.

Part	Example	Purpose
Fixture	contracts\colors\01_class_should_not_add_uiconsts.pas	Provides a small Pascal unit containing a class declaration and non-color words like class and declare.
Expectation	contracts\colors\01_class_should_not_add_uiconsts.expected.json	Forbids System.UIConsts and System.UITypes because the source does not use a real FMX cla color constant.
Generated output	tests\conversion_contract_output\...\output	Temporary output created by the runner so the output can be checked and discarded.
Report output	VCL_to_FMX_Conversion_Report.txt/html	The conversion report checked for required manual-review items, warnings, errors, or forbidden false positives.

What a Contract Is Not

- It is not runtime data loaded by the normal converter UI.
- It is not a mapping pack.
- It is not a sample project meant for users to convert by hand.
- It is not compared against every user project during a normal conversion.
- It is not a promise that the whole converted project is finished; it proves one rule or one defined cluster of rules.

3. Why v7.0 Uses Contracts

The converter performs many different tasks: DFM parsing, FMX generation, Pascal rewriting, uses-clause cleanup, include-file analysis, Windows-message handling, reporting, mapping-pack detection, dry-run output, and more. A small change in one area can accidentally break a distant area.

Contracts reduce that risk by turning each important behavior into a repeatable check. They also make outside reviews more useful. When a reviewer finds a bug, the next question is no longer just whether the source code is wrong. The next question is whether an existing contract should have caught it.

- If no contract exists, add one before trusting the fix.
- If a contract exists but still passed, strengthen the contract.
- If the contract fails after a change, decide whether the converter changed incorrectly or the expected behavior intentionally changed.

4. How the Contract Runner Works

The contract runner is an internal test tool. It scans the contracts folder, finds fixtures with matching expectations, runs the converter against each fixture, and compares generated files and reports against the expectation JSON.

The runner does not use a special database index. It discovers contracts directly from the folder tree. This is fast enough because each fixture is small and the directory structure is simple.

Step	What Happens	Why It Matters
1	Discover expectation files under contracts.	Each expectation identifies the input fixture to run.
2	Copy the fixture into an isolated temporary source folder.	The original contract fixture remains unchanged.
3	Run the converter through RunConversionEngine.exe.	The same core engine is exercised without clicking the UI.
4	Read generated Pascal, FMX, DPR, and reports.	The runner checks behavior, not just process completion.
5	Apply expected and forbidden regex patterns.	A contract can require correct output and forbid dangerous leftovers.
6	Print pass/fail summary.	Any unexpected behavior stops the release gate.

Normal Conversion Versus Contract Execution

A normal user conversion does not compare the user's project to every contract. The converter applies its built-in rules, mapping packs, and analysis logic directly to the user's source. The contracts are only used when the internal contract runner is executed.

In other words, if there are 19 user projects and 150 contracts, the converter does not run 19 times 150 comparisons. The contracts are only test cases for the converter itself.

5. Contract Folder Structure

The v7.0 contracts are organized by behavior family so a maintainer can find the right place for a new fixture quickly.

Folder	Purpose
colors	Color constants, UI constants, modal results, and false positives such as class and clarity.
comments_and_boundaries	Comments, strings, disabled blocks, routine boundaries, initialization/finalization boundaries, and false-positive guards.
components_and_events	Component declarations, lifecycle events, media events, image/property assignments, shape conversion, and event wiring.
dfm_fidelity	Checks for missing DFM/FMXL objects, properties, and events after generation.
dfm_pairs	Paired Pascal/DFM fixtures for FMX form generation.
graphics	Canvas/GDI conversion and visual-review reporting.
include_analysis	Pascal include directives, missing/outside/nested/recursive includes, and UTF-8 include files.
project_integration	Whole mini-project fixtures that exercise multiple systems together.
uses_clause	Unit cleanup, Windows/VCL unit preservation/removal, and conditional uses behavior.
winapi_messages	Windows messaging APIs, message constants, WndProc, message declarations, system commands, and false positives.

6. Anatomy of an Expectation File

Most expectations use the same fields. Fields can be omitted when they are not relevant to a fixture.

Field	Meaning
input_file	The fixture path relative to the project root.
expected_status	The conversion status expected by the runner, usually converted.
expected_conversions	Conversion categories expected to be reported.
expected_manual_reviews	Manual-review categories that must be present.
expected_units_added	Units that must appear in the generated uses clause.
expected_units_absent	Units that must not appear in the generated uses clause.
expected_output_patterns	Regex patterns that must appear in generated output files.
forbidden_output_patterns	Regex patterns that must not appear in generated output files.
expected_report_patterns	Regex patterns that must appear in the text or HTML report.
forbidden_report_patterns	Regex patterns that must not appear in reports.
copy_case_directory	Optional project-level mode that copies an entire fixture directory before conversion.
skip_generated_compile	Optional flag for fixtures that intentionally preserve missing or manual-review dependencies.

Example: A Simple Uses-Clause Guard

This contract proves that ordinary words beginning with cla do not inject FMX UI color units.

```
{
  "input_file": "contracts\\colors\\01_class_should_not_add_uiconsts.pas",
  "expected_status": "converted",
  "expected_output_patterns": ["unit ContractClassShouldNotAddUIConsts"],
  "forbidden_output_patterns": [
    "System\\.UIConsts",
    "System\\.UITypes"
  ],
  "expected_report_patterns": [],
  "forbidden_report_patterns": []
}
```

This expectation is deliberately small. It does not care about every line of generated Pascal. It cares only that the unit still converts and that two false-positive units are absent.

Example: A Positive Color Contract

A good negative contract should usually be paired with a positive contract. The negative contract says class should not trigger color units. The positive contract says a real FMX color constant should trigger them.

```
procedure TContractRealClaColorShouldAddUIConsts.Run;
var
  ColorValue: TAlphaColor;
begin
  ColorValue := claBlack;
  Writeln(Integer(ColorValue));
end;
```

The expectation for this fixture requires both System.UIConsts and System.UITypes. Together, the positive and negative contracts prevent both over-matching and under-matching.

7. What Contracts Prove

- The converter recognizes a known input shape.
- The converter produces required generated-code patterns.
- The converter avoids known dangerous or noisy output patterns.
- The report contains required manual-review, warning, blocking, or informational messages.
- The generated uses clause includes needed units and excludes unused or false-positive units.
- A past bug stays fixed after future changes.

8. What Contracts Do Not Prove

- They do not prove every real project will compile without manual review.
- They do not replace beta testing against real applications.
- They do not make unsupported VCL or Windows behavior automatically safe.
- They do not prove visual correctness beyond the defined output/report patterns.

- They do not prove third-party component behavior unless a mapping rule and fixture explicitly cover that behavior.
- They do not run during normal user conversion unless the internal runner is invoked.

9. How Contracts Interact With Mapping Packs

Mapping packs describe third-party components and conversion actions. Contracts verify converter behavior. They are related, but not the same.

Mapping Pack	Contract
Contains JSON rules for known third-party component classes.	Contains fixtures and expectations that test converter behavior.
Used during normal conversion.	Used by the internal test runner.
Can identify, preserve, substitute, or review a component.	Can prove that mapping-pack behavior appears in generated output or reports.
Ships in the mapping_packs folder.	Ships in the contracts folder as test evidence.

When a mapping rule changes, a contract should be added or updated. The contract should prove the expected generated component, preserved property, manual-review message, or forbidden false positive.

10. How Contracts Interact With Reports

The report is part of the converter's output. For migration work, a truthful report can be as important as generated code. Contracts can therefore assert report behavior.

- A real unsupported item should appear in the report.
- A false positive should not appear in the report.
- A blocking item should be marked as blocking when generated output cannot be trusted.
- A visual-review item should still report visual review even when a safe code conversion is possible.
- Informational dry-run notices should not inflate actionable issue totals.

This distinction helped v7.0 separate warnings, manual-review items, blockers, informational notices, and dry-run notices.

11. How to Add a New Contract

Add a new contract when a behavior is important enough that losing it would damage conversion quality, compile readiness, report honesty, or user trust.

3. Choose the correct behavior folder under contracts.
4. Create the smallest fixture that demonstrates the problem.
5. Create a matching .expected.json file.
6. Use expected patterns for behavior that must appear.
7. Use forbidden patterns for behavior that must never appear.
8. Run the focused contract folder first.
9. Run the full contract suite before trusting the change.
10. If the converter change touches shared logic, run regression guards too.

Example Workflow for a New Bug

Suppose a converted project reveals that TShape.Shape = stCircle becomes TEllipse correctly, but the report still asks for manual review. The proper v7.0 workflow is:

11. Create a fixture with a TShape whose DFM Shape property is stCircle.
12. Add Pascal code that assigns shpLED.Shape := stCircle.
13. Expect TEllipse in the generated Pascal and FMX.
14. Forbid Shape = stCircle in generated FMX.
15. Forbid FMX manual review for shpLED.Shape.
16. Fix the converter.
17. Run contracts and regression guards.

12. Good Contract Design

Good contracts are precise. They do not try to check everything. They check the behavior that matters.

Good Practice	Reason
Keep fixtures small.	Small fixtures isolate the behavior and make failures easy to understand.
Use one main rule per contract.	A contract that checks too much can fail for the wrong reason.
Include forbidden patterns.	Many converter bugs are leftovers, not missing additions.
Pair positive and negative tests.	This prevents over-matching and under-matching.
Assert report behavior.	A clean compile is not enough if the report hides migration risk.
Use project-level fixtures when needed.	Some behavior only appears when multiple units, includes, or DPR files interact.

13. Weak Contracts and How to Repair Them

A weak contract is a contract that passes even when the converter is wrong. The v7.0 color issue is a good example. The contract name said class should not add UI constants, but the expectation did not forbid System.UIConsts or System.UITypes. The runner passed because the expectation was too vague.

Rule: A contract name is not a test. Only the expectation file is a test. If the expectation does not require or forbid the right pattern, the runner cannot catch the bug.

To repair a weak contract, update the expectation so it would have failed before the fix. Then run the old behavior mentally or with the old executable if available. A repaired contract should prove that the bug is genuinely guarded.

14. Public Source Distribution Notes

The public source-distribution zip includes the contracts folder because contracts document the expected converter behavior. The public source zip intentionally excludes internal tools and generated outputs. That means the contracts are present as authoritative fixtures, but the runner scripts and RunConversionEngine.exe are not included in that compile-only zip.

For internal development, the runner remains in the tools folder of the working repository. Before trusting contract results after engine changes, rebuild both the main converter executable and the contract runner executable.

15. Recommended Review Instruction for Outside AI Reviewers

When sending the project to another code reviewer, include the contracts folder and explain how to interpret it. A useful instruction is:

> Review the converter source together with the contracts folder.
> Each contract is a fixture plus an .expected.json file describing required
> output, forbidden output, report expectations, and uses-clause expectations.
> If you find a bug, check whether a contract already covers it.
> If a contract exists but would not fail, report that the contract is weak.
> If no contract exists, propose a new contract fixture and expected JSON.

16. Current v7.0 Contract Baseline

Metric	Current Value
Contract expectations	227
Mapping packs	22
Last verified full contract run	227 pass, 0 fail
Regression guard status	41 pass, 0 blockers, 0 warnings
Normal conversion dependency	Contracts are not loaded during normal user conversion.

17. Summary

The v7.0 contract system is the converter's memory. It records known migration problems as executable examples, checks generated output and reports, and prevents the same class of error from silently returning. The contracts do not remove the need for real-project testing, but they make real-project discoveries permanent. Every important converter rule should eventually have a fixture, an expectation, and a regression guard.

Inherited maintenance changes

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.

V7 retains the established contract cases. Current validation results are recorded in the V7 Application Inventory; historical build results are not evidence of a current release build.

V7 Atlas language support and operating safeguards

Last changed October 15, 2026

VCL2FMXConverter v7.0 Atlas retains the established conversion workflow and adds 35 selectable interface language packs, translated display messages and reports, and online translation of Help. Language packs do not add conversion rules or translate generated applications. V7 also includes input, output and encoding safeguards described in this edition; it is not strictly a language-only code change.

Choosing a language

Choose a language in the header. The 35 bundled interface packs work offline and the preference is saved. On first use the language manager can select a matching system language. Missing packs fall back to English. Display messages and reference descriptions are translated; technical identifiers retain their spelling. The language selected when conversion starts is used for saved text and HTML reports. Existing reports are not rewritten when the interface language changes. Canonical JSON reports retain their English schema and data.

Reading translated Help

Help opened from Atlas uses a local HTTP server and passes the selected language to the manual. Keep Atlas running while browsing that Help session. Automatic Help translation requires Internet access; English Help remains available offline. Help hosted as ordinary web pages uses its language selector and can inherit the website selection. Opening local HTML directly with file:// does not provide the same online translation behavior. PDF and DOCX guides are supplied in English.

Editing and adding translations

Use the Delphi App Translation Studio with the matching V7 source workspace. Open the translation project/catalog under Localization/Development, scan the application, edit translations and glossary entries, run Validation, then Export Runtime Pack. Install the locale JSON under Localization/Languages beside the executable and restart Atlas. Back up customized packs before replacement. For a new language, create a catalog for that target locale rather than relabeling an existing pack. Check placeholders, technical names, long labels and right-to-left layout. A new Help language also needs a supported language mapping in the separate Help selector. Editing an interface pack does not change the online Help translation service.

Translation review

The included language packs are machine-generated and have not been reviewed by native speakers. Translation problems can be corrected using the Delphi App Translation Studio or reported to the developer with the locale, current wording and proposed correction. For a complete procedure, see the V7 Translation Guide and the Help topic Using and maintaining translations.

Source and output checks

Select one original VCL project and a separate output folder. V7 resolves paths before conversion, checks project selection before output cleanup, rejects ambiguous or mismatched DPR/DPROJ selections, and blocks empty input and unreadable source files. Scanned directory links and reparse points, and hard-linked output files, are rejected. Dry-run preview may write reports and create the output directory, but does not write converted source or remove existing build/history folders. Report actions use only paths produced by the current run.

Text encoding

Input text detection supports UTF-8, UTF-16 and explicit supported Pascal code-page directives. Without a byte-order mark or directive, valid UTF-8 is preferred; other legacy bytes use Windows-1252. Convert other legacy encodings to UTF-8 in Delphi before conversion. Generated Pascal output is UTF-8, and active code-page directives are normalized to {$CODEPAGE UTF8}. This can change generated text without changing the intended conversion rule.