Roc platform template for Rust

Project documentation
unreleased

Contents

Run a Roc application 3
- Use a released platform 3
- Use a working checkout 3
- Choose an example 3

Understand the platform interface 4
- Application entry point 4
- Host services 4
- Targets and validation 4

Contribute to the platform 5
- Contribute 5
- Tooling and product compilers 5
- Build and bundle 5
- Update host bindings 5
- Test the interface 5

Maintain independently released linker inputs 7
- Contents and producer identity 7
- Producing and publishing a change 7
- Routine consumption 8
- Component inventories 8

Publish and maintain a release 9
- Prepare a version 9
- Publish the tested files 9
- Recover an interrupted publication 9
- Compatibility and documentation 9
- Deferred shared-tooling work 10

Report security concerns 11
- Report a vulnerability 11
- Supported releases 11
- Verify what you consume 11

Release history 12
- 1.2.1 12
- 1.2.0 12

Roc platform template for Rust

:doctype: book :toc: left

Build a native Roc platform with a Rust host. Start with the released examples if you want to write a Roc application. Start with the contributor chapter if you want to change the host or the platform interface.

The latest release contains a platform bundle and examples that name that exact bundle. The API reference describes the latest published platform; this manual describes repository maintenance at version unreleased.

Run a Roc application

Use a released platform

Install the latest Roc nightly, then download and extract `roc-platform-template-rust-examples-<version>.zip` from the latest release. The archive includes a README and an executable specification of the examples' arguments, standard input, output and exit status.

```
roc examples/hello_world/main.roc
roc build examples/hello_world/main.roc
```

The archive's applications name the release's immutable platform URL. Keep that URL when creating your own application. The hash in its filename verifies the bundle content. The platform does not pin your compiler; the daily compatibility workflow checks the published examples against the latest nightly.

Use a working checkout

Install Blueprint, Nix and a current Roc nightly available as `roc`. Run the same named tasks locally and in CI:

```
blueprint run build-platform
roc examples/hello_world/main.roc
blueprint run check
```

Committed examples depend on `../../platform/main.roc`, so they exercise your changes. They require a host built from this checkout. Downloaded release examples instead use their release's bundle and need no local Rust host build.

Choose an example

`hello_world` demonstrates output. `cli_args` receives arguments, `echo` reads one line, `echo_multiline` reads multiple lines, and `exit` demonstrates a nonzero application exit status. `tests/spec.json` records their exact behavior. The syntax and debug examples also serve as integration fixtures; they are not additional platform services.

Understand the platform interface

Application entry point

The application provides `main!`, receiving `List(Str)` and returning `Try({}, [Exit(I32), ..])`. An `Ok({})` result exits successfully; `Err(Exit(code))` uses the supplied process exit status. Other errors are reported on standard error and return a failure status.

Host services

Service	Behavior
`Stdout.line!`	Write a line to standard output.
`Stderr.line!`	Write a line to standard error.
`Stdin.line!`	Read a line from standard input.

The generated API reference documents the public Roc types. The Rust host owns allocation, process I/O and the generated ABI boundary.

Targets and validation

Bundle target	Host architecture	Native compatibility coverage
`x64musl`	Linux x86_64 with musl	Ubuntu x86_64 runner
`arm64musl`	Linux ARM64 with musl	Ubuntu ARM64 runner
`arm64mac`	Apple Silicon macOS	Apple Silicon macOS runner
`x64mac`	Intel macOS	Cross-compiled only; native validation disabled pending upstream support

Every host is stored at `platform/targets/<target>/libhost.a` in the bundle. Cross-compilation is evidence that the host compiles; it does not establish native execution. Do not interpret the compatibility badge as Intel macOS validation. Native coverage is tracked in roc-automation#101.

Contribute to the platform

Contribute

Report a reproducible bug or propose a change through the issue tracker. For a bug, include the operating system, target, `roc version`, the platform release or source commit, the command and its output. Report security concerns through the private channel.

Open a pull request from current main. Keep commits focused and signed. Run `blueprint run check` before review; CI also tests the exact release candidate archives. Changes to behavior need meaningful tests, including error cases.

Tooling and product compilers

Blueprint declares the tools and native toolchains; its lock records the reviewed inputs. `roc-stable` runs source automation under `scripts/`. Its tag matches those scripts' header fields. The plain `roc` on your PATH checks, tests and bundles the product with the current nightly. Product source and examples carry no compiler pin.

When updating tooling, change the Blueprint setting and affected script pins together. Format source scripts with `roc-stable fmt`; format platform source and examples with `roc fmt`. Never refresh a lock merely to silence a failure: review the selected dependency identities.

Build and bundle

```
blueprint run build-platform -- --target all
blueprint run bundle -- --output-dir dist
```

The first task cross-compiles the host and installs hash-verified linker inputs. The second assembles the bundle without rebuilding native code. For local validation, `--target native` builds or bundles only the machine's target. Generated hosts, bundles and documentation are not committed.

Update host bindings

Bindings in `src/roc_platform_abi.rs` are committed and consumed by host builds. The bindings check regenerates them using the selected nightly's verified source specification and compares them with the committed file. An interface change requires reviewing both the generated diff and matching host changes. Use the bindings task's explicit update operation, then build and run the examples on all supported runners. A stale binding must fail CI; host builds must not silently regenerate it.

Test the interface

`blueprint run check` covers formatting, source policy, unit tests, working-tree examples and examples against a loopback-served bundle. `tests/spec.json` records arguments,

input, exact output and exit status. A check never rewrites expectations. Review explicit expectation updates as behavior changes.

Register every application under `examples/` in `tests/spec.json`. Validation rejects unlisted applications, duplicate application paths and empty case lists; adding a source file must not leave its behavior untested.

```
blueprint run examples
blueprint run examples -- --operation test --archive <examples.zip> --bundle
<bundle.tar.zst>
```

Tooling and compiler commands have a ten-minute safety deadline; individual example processes have ten seconds. A timeout, missing result or skipped case does not establish compatibility. Inspect retained diagnostics on successful as well as failed compiler calls.

Maintain independently released linker inputs

The Linux startup, libc, unwinding, and compiler-runtime files are linker inputs, not Rust host outputs. They therefore have an independent release cycle selected by `runtime/link-inputs.lock.json`. `libhost.a` remains part of the platform host cycle: changing host source can require a new host without rebuilding stable operating-system inputs.

This separation matters because ordinary pull requests should validate the code under review, not repeatedly download Zig and reconstruct an already reviewed dependency. Each locked target archive has an immutable release name, byte size, and SHA-256. CI restores archives from GitHub Actions cache by lock identity and rehashes every cache hit. A cache miss downloads only the selected release asset. The lock authorizes bytes; the cache merely avoids network traffic.

Contents and producer identity

`source.json` pins the Zig 0.16.0 distribution and source archive by SHA-256. The producer builds `x64musl` and `arm64musl` in separate empty caches and selects the five files from Zig's final link command: `crt1.o`, `libc.a`, `libunwind.a`, `libzigc.a`, and `libcompiler_rt.a`. Each target archive also contains licenses and an inner manifest with per-file hashes.

The producer fingerprint covers the builder, native validator, source and test toolchain pins, smoke source, and workflow. This makes a material producer change fail as stale instead of silently substituting newly built bytes during routine CI. Archive hashes identify the exact consumer bytes; the fingerprint answers whether their recipe is still current. These are related but distinct checks.

Producing and publishing a change

The unprivileged `runtime-release.yml` workflow is dispatch-only. It builds both targets, tests each archive on its native architecture, emits canonical `build-input-release.json`, and attests that manifest and both archives. It has no repository or release write authority.

Use this order when the producer fingerprint changes:

1. Stage the material change in a same-repository PR. Ordinary validation may report a stale lock until the new release is adopted.
2. Optionally dispatch `runtime-release.yml` on the PR branch with `release_candidate=true` and `expected_sha` set to its exact head. Review the native tests, manifest, archive hashes, and attestations. This unprivileged preview creates no release and changes no lock.
3. From `main`, dispatch **Publish linker inputs** (`publish-linker-inputs.yml`) with `pull-request` set to that PR number. Its full-SHA-pinned trusted wrapper reruns the producer at the exact PR head, verifies workflow and source identity, hashes, target inventory, and attestations, publishes an immutable manifest-hash-derived release, and appends a lease-guarded, GitHub-signed `runtime/link-inputs.lock.json`-only commit to the PR.
4. Review the lock as a dependency update and rerun routine CI against the released bytes. Merge with a merge commit after all required checks pass. Build and publish the

   Roc platform bundle later through its separate release workflow, using the adopted lock.

The optional preview does not replace the publisher's own producer run.

The split authority is intentional: PR code may compile, but cannot publish or write its own asserted lock; trusted default-branch code may publish inert tested bytes, but never executes scripts from the candidate artifact. Keeping the PR-built release after merge preserves the exact source-to-tests-to-attestation chain. Do not rebuild or promote it on `main`.

The merge commit keeps both the attested producer commit and signed lock-only commit visible. Changed candidate bytes require a new content identity; never replace a published asset or move its tag.

Routine consumption

`blueprint run build-platform` requests only the native target unless `--target all` is used. The consumer:

1. recomputes the producer fingerprint;
2. locates the target archive in the local content cache;
3. downloads the exact locked release asset only on a miss;
4. checks size and SHA-256 even for a cache hit;
5. rejects links, traversal, duplicates, unexpected files, and inner hash errors;
6. installs through the declared `platform/targets/` and metadata paths.

Routine builds do not query attestations. Attestations are admission evidence checked by the trusted publisher; the committed content hashes provide cheap, offline integrity checks on every use. This avoids turning GitHub's attestation service into a routine availability dependency while retaining provenance at the release boundary.

Component inventories

The selected historical linker-input releases have no artifact-bound supplier SBOM sidecars. Do not substitute an unpublished inventory or claim complete supplier coverage. Product inventories record this as an explicit gap. The trusted publisher currently rejects extra sidecars; its versioned manifest, attestation, transfer verification and generated lock need shared support before this step can be enabled. The upstream TODO is linked from roc-automation#62.

Publish and maintain a release

Prepare a version

A platform version is independent of the compiler version. Compare the changed public interface with the latest bundle using `roc bump`, and choose a version that is large enough for the change. Author `docs/releases/<version>.adoc` before dispatching the release. Explain user-visible changes and any migration, not just a list of merged commits.

Dispatch Release from main with the unprefixed version. The workflow validates the version and notes, runs checks, builds or restores the hosts, bundles once, and tests the exact archived examples against the exact bundle with a clean package cache on every supported runner.

Publish the tested files

Generation and validation run with read-only permissions. The protected `release` environment permits publication from main only. Its publisher treats artifacts as inert bytes: stage a draft, upload the explicit asset set, download and verify names, sizes and hashes, then publish once. Privileged jobs never execute candidate source or build scripts.

A release retains its platform bundle, frozen examples, API documentation archive, manual PDF, checksums, artifact-bound SBOM sidecars and available attestation evidence. Notes identify the source commit and product/tooling compilers. Coverage gaps remain explicit in the inventories.

Recover an interrupted publication

If publication fails before a release is public, inspect the draft and the run's evidence. Delete the failed draft before retrying the supported publisher path; do not publish it by hand. Never move a version tag, replace assets or repair a published immutable release. Changed software bytes require a new version.

A corrected SBOM uses a new immutable location and document revision bound to the unchanged software digest, with links to the retained original.

Compatibility and documentation

Nightly compatibility tests the latest published examples against the latest compiler. A failing release is fixed with a new platform release, not by editing its archive. A diagnostic run naming historical versions must not close a tracking issue for the latest pair.

The site leads with the latest final release's API and example archive. Development documentation is labelled unreleased. Releases before documentation archives existed can have reconstructed documentation labelled with their exact source/compiler identity; reconstruction never adds assets to an old release. The manual is available as HTML, PDF and static Markdown under `manual/`.

Deferred shared-tooling work

The shared platform release adapter is tracked in roc-automation#100. Until it supports native bundle contents, this repository retains a small platform-specific adapter. Intel native validation remains disabled pending roc-automation#101. Linker-input SBOM publication remains incomplete pending roc-automation#62.

Report security concerns

Report a vulnerability

Use GitHub's private vulnerability reporting form. Do not put sensitive reproduction details in a public issue. Include the release or source commit, affected target, impact and a minimal reproducer. See the repository's `SECURITY.md` for the same reporting entry point.

Supported releases

Security fixes target the latest platform release. Older releases do not receive automatic backports. Upgrade to a fixed release when one is available. There is no promised response time. Public reports and advisories identify which releases and targets are affected when that information is established.

Verify what you consume

Use immutable release URLs and verify downloaded hashes. Product release sidecars describe artifact composition and documented gaps; provenance records the producer and source. Neither establishes that the software is free of vulnerabilities. Missing supplier coverage is reported as incomplete, never as proof that a component is absent.

Release history

1.2.1

This maintenance release strengthens source-policy checks, generated binding validation and release evidence. It retains the platform interface and example paths. Product release inventories describe the delivered files and retain explicit missing supplier coverage.

The manual brings build, contribution, release and security instructions into one place. Intel macOS remains cross-compiled only; the native compatibility result covers Linux x86_64, Linux ARM64 and Apple Silicon macOS.

The Rust host and committed bindings now match the selected nightly glue. Host implementers will see explicit ownership requirements on generated allocation and reference-count helpers; the supplied host has been updated with them. The public Roc interface remains unchanged.

1.2.0

This release removes product compiler pins and makes downloaded example archives use the exact published platform bundle. Daily compatibility validates the published examples against the latest nightly on Linux x86_64, Linux ARM64 and Apple Silicon macOS. Committed examples instead follow the working tree.

The original immutable release has no documentation archive or product SBOM sidecars. Reconstructed API documentation is labelled as such and does not change the release's original asset inventory.