Getting Started with Keleusma 0.2.2

Brendan Sechter

2026-07-11

Keleusma is a total functional stream processor that compiles to bytecode and runs on a stack-based virtual machine. The language ships with a static worst-case-execution-time bound and a static worst-case-memory-usage bound that a load-time verifier proves before any program runs. The 0.2.0 release covered in an earlier article introduced cryptographic module signing, information-flow labels, newtypes with refinement predicates, and a reset instruction-set architecture. The 0.1.1 pre-release was covered in the first article of this series.

Version 0.2.1 was tagged on 2026-07-08. It is a consolidation release that fills gaps in the surface syntax, adds a general const-generics facility, turns scripts into first-class shell citizens, provides source-level debugging support, tightens the load-time verifier, and adds an operator-configured deployment policy for signed and encrypted bytecode. Version 0.2.2 was tagged on 2026-07-09, the day after 0.2.1. It is a build-fix and tooling release on the self-hosting groundwork line. It repairs cross-target and continuous-integration regressions from 0.2.1 that broke the flagship Cortex-M embedded targets and the `verify`-without-`floats` feature combination, lands the learning guide as a bilingual mdbook that is now served at the hosted book URL, lands the initial scaffold of the self-hosted-compiler subproject that the 0.3.0 release will complete, and codifies the release process with a mandatory green-continuous-integration gate. The language surface is unchanged from 0.2.1, and no wire-format or bytecode-version change accompanies the release. The self-hosting concept was treated for the software case in the streaming compilers series conclusion and for silicon in the recent hardware article.

Readers who want to try the language without installing anything can use the browser-based playground, which compiles and verifies programs through a WebAssembly build of the compiler and reports worst-case execution time and memory bounds live. The playground is served alongside the hosted book.

This article walks through the material additions of the 0.2.x line with runnable examples, covering the 0.2.1 language features that 0.2.2 preserves and the 0.2.2 tooling additions that are relevant to a getting-started walkthrough. Every code listing below was executed with the version recorded in the Software Versions section, and every reported output is the actual output produced. The article is an on-ramp for readers already familiar with 0.2.0. Readers new to the language should start with the bundled guide and its installation chapter.

Software Versions

```
# Date (UTC)
$ date -u "+%Y-%m-%d %H:%M:%S +0000"
2026-07-10 22:05:01 +0000

# OS and Version
$ uname -vm
Darwin Kernel Version 25.5.0: Mon Apr 27 20:38:56 PDT 2026; root:xnu-12377.121.6~2/RELEASE_ARM6
```

```
# Keleusma
$ keleusma --version
keleusma 0.2.2
```

Installation

Keleusma 0.2.2 is published on crates.io as a library and as the separate command-line crate `keleusma-cli`. The source lives on GitHub and the application-programming-interface documentation is on docs.rs. The install path is unchanged from 0.2.0. The 0.2.2 release additionally repairs the 32-bit and `no_std` embedded builds that 0.2.1 broke, so a fresh install from source on the flagship Cortex-M targets now succeeds without a manual patch.

```
git clone https://github.com/sgeos/keleusma
cd keleusma
cargo install --path keleusma-cli --bin keleusma
```

Confirm the installation.

```
$ keleusma --version
keleusma 0.2.2
```

To embed the runtime in a Rust program rather than use the tool, add the library crates to a project.

```
[dependencies]
keleusma = "0.2"
keleusma-arena = "0.3.1"
```

The `keleusma-arena` version requirement tightens from `0.3` to `0.3.1` in 0.2.2 because the runtime now consumes additive helpers that `keleusma-arena` 0.3.1 exposes.

Boolean, Bitwise, and Shift Operators

Version 0.2.0 added the five bitwise opcodes `BitAnd`, `BitOr`, `BitXor`, `Shl`, and `Shr` without a grammar to reach them from source. Version 0.2.1 supplies the grammar and, in the same pass, rearranges the boolean operators so that the two families never disambiguate by operand type.

The bitwise family uses the letter-prefixed names `band`, `bor`, `bxor`, and the prefix `bnot`. The operators apply to `Word`, `Byte`, and the parameterized `Multiword<N>`. On a `Multiword` the operation runs limb by limb with no cross-limb interaction. On a `Byte` the operation runs at the byte width, so `bnot 0Byte` is `255Byte`.

```
fn main() -> Word {
    let mask: Word = 0b1100 band 0b1010;
    let all: Word = 0b1100 bor 0b0011;
    let flipped: Word = 0b1010 bxor 0b1111;
    let inverted: Byte = bnot 0Byte;
    mask + all + flipped + (inverted as Word)
}
```

```
$ keleusma run 01_bitwise.kel
283
```

The shift family uses the assembly mnemonics `lsl` (logical left), `asl` (arithmetic left), `lsr` (logical right), and `asr` (arithmetic right). The `asl` and `lsl` operators produce the same bit pattern but `asl` denotes the multiplicative interpretation `x * 2^k` and therefore admits the `overflow`

and `underflow` arms of the checked-arithmetic construct. A variable shift amount is admissible and the worst-case bound is preserved, because the multi-word case is unrolled over the compile-time word count with runtime index arithmetic and branch-free bounds guards.

```
fn main() -> Word {
    let left: Word = 1 lsl 4;
    let arith_left: Word = 3 asl 2;
    let logical_right: Word = 128 lsr 3;
    let arith_right: Word = (0 - 8) asr 1;
    left + arith_left + logical_right + arith_right
}
```

```
$ keleusma run 02_shift.kel
40
```

The boolean family has two subfamilies. The eager `and`, `or`, `xor`, and prefix `not` always evaluate both operands. The short-circuit `andalso` and `orelse` skip the right operand when the left already decides the result. The eager forms are the branch-free default because a definitive worst-case-execution-time bound prefers branch-free code. The short-circuit forms remain available for cases where skipping the right operand is intended. Selection is by operator name and is never inferred from operand type.

```
fn main() -> Word {
    let a: bool = true and false;
    let b: bool = true or false;
    let c: bool = true xor true;
    let d: bool = not false;
    let e: bool = true andalso false;
    let f: bool = false orelse true;
    let count: Word =
        (if a { 1 } else { 0 })
        + (if b { 2 } else { 0 })
        + (if c { 4 } else { 0 })
        + (if d { 8 } else { 0 })
        + (if e { 16 } else { 0 })
        + (if f { 32 } else { 0 });
    count
}
```

```
$ keleusma run 03_boolean.kel
42
```

General Const Generics

Version 0.2.0 accepted a fixed-point width parameter on the `Multiword<N, F>` type as a special case. Version 0.2.1 replaces the special case with a general const-generics facility. A definition may be parameterized by a compile-time constant of type `Word` in addition to its type parameters. The parameter is introduced by the `const` keyword and serves in a type position as an array length or a `Multiword` parameter, and in a value position inside a function body as an ordinary `Word`.

```
fn tag_first<const n: Word>(buf: [Word; n]) -> Word {
    buf[0] + buf[n - 1] + n
}

fn main() -> Word {
    let five: [Word; 5] = [10, 20, 30, 40, 50];
```

```
    let three: [Word; 3] = [7, 14, 21];
    tag_first::<5>(five) + tag_first::<3>(three)
}
```

```
$ keleusma run 04_const_generics.kel
96
```

Const arguments are always explicit because they cannot be inferred from value arguments. A call writes a turbofish `f::<8>(...)`, a struct construction writes `Buf::<8> { ... }`, and a type reference writes `Buf<8>`. A const argument may be a total arithmetic expression over `+`, `-`, and `*`, so `Buf<n + 1>` and `Multiword<2 * n>` are admissible. Division and modulo are excluded from const arithmetic so evaluation is total.

Monomorphization substitutes every const parameter to a concrete literal before the load-time analyses run. Every array length, every `Multiword` parameter, and every loop bound in the specialized code is therefore a literal, and the worst-case-execution-time and worst-case-memory-usage analyses observe no symbolic constant. The static bounds are preserved unchanged.

Executable Scripts

A Keleusma script may begin with a shebang line and become a directly executable file on Unix.

```
#!/usr/bin/env keleusma

fn main() -> Word {
    12 * 42
}
```

Mark it executable and run it as a command.

```
$ chmod +x 05_shebang.kel
$ ./05_shebang.kel
504
```

The `keleusma` command-line frontend runs a bare path as a `run` invocation, and the lexer skips the shebang line while preserving source line numbers in diagnostics. Combined with the `shell` native bundle, a script becomes a portable command-line orchestrator that delegates work to POSIX tools, drives control flow on the returned `Word` exit codes, and sets its own process exit code with `shell::exit`.

Script Arguments

The `shell` bundle exposes a script's own arguments through `shell::arg(i)` and `shell::arg_count()`. Index zero is the script path, and indices one and above are the positional arguments that the launcher passed after it, mirroring the shell variables `$0` and `$1`. The `shell::arg` native returns `Option<Text>` so an out-of-range index is a total operation that the script destructures with `match`.

```
#!/usr/bin/env keleusma

use shell::arg
use shell::arg_count

fn label(a: Option<Text>) -> Word {
    match a {
```

```
        Option::Some(_name) => 1,
        Option::None => 0,
    }
}

fn main() -> Word {
    let n: Word = shell::arg_count();
    let first_present: Word = label(shell::arg(1));
    n * 10 + first_present
}
```

```
$ chmod +x 06_args.kel
$ ./06_args.kel alpha beta
31
```

The output encodes the argument count `n = 3` in the tens digit and the presence of index one in the ones digit. The command-line frontend also accepts a `--` terminator after which every token is treated as a script argument regardless of shape. A companion native `shell::run_full(cmd) -> (Word, Text, Text)` returns the exit code together with both standard-output and standard-error streams, complementing `shell::run` which captures and discards the error stream.

Debug Assertions

The new `assert` statement expresses a compile-out debug assertion over a `bool` condition. An optional message argument attaches a diagnostic string.

```
fn safe_scale(base: Word, factor: Word) -> Word {
    assert factor >= 0, "factor must be non-negative";
    base * factor
}

fn main() -> Word {
    safe_scale(6, 7)
}
```

Under a debug build the compiler emits a runtime check that traps with `VmError::AssertionFailed` when the condition is false, together with a strippable debug record carrying the source span and the optional message.

```
$ keleusma compile 07_assert.kel --debug -o 07_assert.bin
wrote 07_assert.bin (3812 bytes)
$ keleusma run 07_assert.bin
42
```

The next example flips the argument sign and shows the diagnostic.

```
fn safe_scale(base: Word, factor: Word) -> Word {
    assert factor >= 0, "factor must be non-negative";
    base * factor
}

fn main() -> Word {
    safe_scale(6, 0 - 7)
}
```

```
$ keleusma compile 07b_assert_fail.kel --debug -o 07b_assert_fail.bin
wrote 07b_assert_fail.bin (3892 bytes)
```

```
$ keleusma run 07b_assert_fail.bin
error: vm: AssertionFailed
```

Under an ordinary compile the statement compiles out entirely and contributes no opcodes. The `assert` keyword is not reserved outside statement position, so `assert(x)` at expression position remains a call to a user-defined function. The virtual machine also records the source position of the trap through the internal `fault_location` field, which a host program consumes through the `Vm::fault_source_location()` application-programming-interface to map the trap to a source span. The command-line frontend used for the demonstration above prints the `VmError` alone and leaves the span-resolution step to a host program that consumes the library directly.

Partial Operation Handling

Every mathematically partial operation now has a defined contract and an opt-in source-level handling construct, so a program can be made total at the source level rather than relying on a runtime trap. Checked arithmetic over `Word`, `Byte`, `Float`, and `Fixed<N>` uses the arm family `ok`, `overflow`, `underflow`, and `zero_divisor`. An omitted `overflow` or `underflow` arm defaults to two's-complement wrapping. Inside an arm body the keywords `saturate_max` and `saturate_min` stand for the largest and smallest value of the operand type and let a program clamp an out-of-range result to the edge of the range. Array indexing uses `invalid_index`. Refinement-newtype construction uses `invalid_newtype`. The discriminant-to-enum conversion uses `ok`, `payload_discriminant`, and `invalid_discriminant`. Fallible native calls use `error(code)`, with the host reporting the `Word` code through the new `KeleusmaError` derive.

```
fn safe_div(a: Word, b: Word) -> Word {
    a / b {
        ok(q) => q,
        zero_divisor(_n) => 0,
    }
}

fn saturate_add_byte(a: Byte, b: Byte) -> Byte {
    a + b {
        ok(v) => v,
        overflow(_) => saturate_max,
    }
}

fn main() -> Word {
    let q: Word = safe_div(84, 2);
    let z: Word = safe_div(10, 0);
    let s: Byte = saturate_add_byte(200Byte, 100Byte);
    q + z + (s as Word)
}
```

```
$ keleusma run 09_partial.kel
297
```

The three checked operations produce `safe_div(84, 2) = 42`, `safe_div(10, 0) = 0` through the `zero_divisor` arm, and `saturate_add_byte(200Byte, 100Byte) = 255Byte` through the saturating `overflow` arm, summing to 297. The virtual machine traps recoverably on an unhandled partial operation through specific `VmError` variants. The specification of every runtime fault lives in the reference document Handling Partial Operations.

Strippable Debug Metadata

Compiled bytecode can carry optional per-chunk debug metadata that maps op-stream positions back to source. The compile flag `--debug` emits it, and the new subcommand `keleusma strip` removes it, producing a release artefact byte-identical to a non-debug compile of the same source.

```
$ keleusma compile 08_strip.kel -o 08_release.bin
wrote 08_release.bin (2456 bytes)

$ keleusma compile 08_strip.kel --debug -o 08_debug.bin
wrote 08_debug.bin (3372 bytes)

$ keleusma strip 08_debug.bin -o 08_stripped.bin
stripped 08_debug.bin -> 08_stripped.bin (2456 bytes)

$ cmp 08_release.bin 08_stripped.bin && echo "IDENTICAL"
IDENTICAL
```

The metadata lives in a chunk-local pool in the wire format's auxiliary body and never in the opcode stream. A debug build and a release build therefore share an identical opcode stream for the same source, and stripping is a pure subtraction rather than a transform. The subcommand refuses signed or encrypted input, because rewriting the body invalidates a signature. The supported ordering is compile, then strip, then sign.

The metadata catalogue covers twelve record kinds including source-span records for statements, line-number records, variable-name records, call-site records, type annotations, information-flow-label annotations, generic-instantiation records, verifier witnesses, worst-case-execution-time markers, breakpoint candidates, assertion contexts, and optimisation markers at refinement-elision sites.

Deployment Policy for Signed and Encrypted Bytecode

Building on the 0.2.0 module-signing facility, the command-line frontend gains an operator-configured execution policy that constrains which bytecode may run, analogous to an enrolled-key model in a firmware trust framework.

Strict signing activates when a trust store is in force. Configuration is by the environment variable `KELEUSMA_TRUSTED_KEYS_DIR` naming a directory of `*.pub` verifying keys, or by the platform-conventional directory `/etc/keleusma/trusted_keys`, or by the force flag `KELEUSMA_REQUIRE_SIGNED=1`. When strict signing is active, the frontend rejects source files and unsigned bytecode, admits signed bytecode only when the signature validates against an enrolled signer, and rejects the command-line flag `--verifying-key` so an unprivileged operator cannot relax the system-managed trust list. Strict encryption activates symmetrically through `KELEUSMA_DECRYPTION_KEYS_DIR` and `KELEUSMA_REQUIRE_ENCRYPTED`.

The two modes compose into four policy states from the permissive default through fully locked-down. The operator manual with air-gapped, production-fleet, and kiosk deployment scenarios is the reference document Security Policy.

Under the Hood

Three internal changes in 0.2.1 are worth naming even though the source language is not directly affected.

The composite runtime representation is now flat bytes resident in the host arena rather than heap-allocated `Vec` and `String` graphs. The runtime `Value` slot is thirty-two bytes, down from

forty, pinned by a compile-time size assertion. Composite construction is a bump-pointer allocation in the arena's transient region with no global allocator, so a composite-building script runs on a `no_std` target without a global heap. Worst-case-memory-usage bounds are correspondingly tighter and now reflect the language's fixed-size guarantee rather than the previous `Vec` and `String` over-approximation.

The load-time verifier gains a typed operand-stack pass after the manner of the Java Virtual Machine and WebAssembly verifiers. The pass reconstructs the flat shape of every operand-stack entry and local slot by a bytecode-level type-preservation abstract interpretation. It validates every compiler-baked composite, field, and array-element offset against the canonical flat layout of the accessed type, closing several audit findings that were previously trusted at runtime under a debug assertion. It upgrades the operand-depth pass from max-of-branch-depths to an exact-height join. It enforces loop back-edge operand-stack neutrality. And it validates wire-carried shared-slot offsets against the shared-data buffer.

Trait methods on generic structs and enums now resolve on a concrete, type-generic, or const-generic receiver alike. Monomorphization specializes each generic implementation once per concrete instantiation of its target type, substituting the implementation's type and const parameters through the method signatures and bodies, so a call on a `Cell<Word>` receiver reaches the specialized method even when the source implementation was written against a generic `Cell<T>`.

Toward a Self-Hosted Compiler

Version 0.2.2 lands the initial scaffold of the self-hosted-compiler subproject that the 0.3.0 release will complete. The scaffold lives at `compiler/` in the repository and comprises the three-stage `loop` pipeline skeleton, namely `lexer`, `parser`, and `codegen`, along with a Rust host driver and a release-by-release implementation plan. No stage is implemented in 0.2.2. The V0.2.x line lands its prerequisites across the operator surface, the const-generics facility, the flat-byte composite representation, the typed operand-stack verifier pass, the debug metadata, and the strict-mode deployment policy that the earlier sections of this article walk through. Version 0.3.0 will turn the scaffold into a working compiler written in Keleusma that compiles Keleusma.

The self-hosting concept was treated at length for the software case in the streaming compilers series conclusion, which discusses the coalgebraic fixed-point condition that a self-hosted compiler satisfies. It was treated for the silicon case in the recent article on self-hosted silicon compilation. The Keleusma standardization effort sits on the software side of that boundary and offers a candidate example of a compact-toolchain language design that a self-hosting compiler could reasonably compile itself with.

Going Deeper

This article covers the material additions of the 0.2.x line that are relevant to a getting-started walkthrough. The complete language reference is the hosted book, whose 0.2.2 release migrated the previously loose Markdown guide into an mdbook served at `https://sgeos.github.io/keleusma/`. The book is bilingual with English as the source and Japanese as a gettext-based translation that 0.2.2 also ships. It teaches Keleusma from first principles in a forty-chapter track and covers the embedding surface for Rust hosts in a second track. Readers who prefer to try the language without installing anything can use the browser-based playground, which runs the compiler as WebAssembly in the reader's browser and is served from the hosted book site. The reference for handling partial operations is the partial-operations chapter. The reference for information-flow labels is the labels chapter. The reference for deployment-policy configuration is the Security Policy document. The

reference for shebang-executable scripts is the Automation and Scripting document. The bundled example scripts are the seed material the guide builds on.

Two companion articles apply Keleusma to specific problem shapes. A verifiable control kernel develops the language around a small runtime kernel. Information-flow control, a deep dive develops the language around the security-labelled data model that Version 0.2.0 introduced.

Conclusion

Neither 0.2.1 nor 0.2.2 changes the central promise the language makes. Every accepted program still carries a static proof of bounded execution time and bounded memory usage. What 0.2.1 adds is completeness on the surface syntax where 0.2.0 left gaps, a general const-generics facility that supersedes the earlier special case, first-class scripting ergonomics that turn a Keleusma file into a command-line tool, source-level debugging support that a debugger or host program can consume, a tightened load-time verifier that closes several audit findings, and an operator-configured deployment policy for signed and encrypted bytecode. What 0.2.2 adds is the initial scaffold of the self-hosted-compiler subproject, the learning guide as a bilingual mdbook that is now hosted online, a browser-based playground that runs the compiler as WebAssembly, and a codified release process with a mandatory green-continuous-integration gate. The 0.2.2 release also repairs build regressions that 0.2.1 introduced on 32-bit and `no_std` embedded targets and in the `verify`-without-`floats` feature combination. The pattern is consolidation and tooling maturation, not a change of direction. The direction is set by the 0.3.0 self-hosted-compiler goal that the 0.2.2 scaffold lays the groundwork for.

References

- Keleusma, Crate on crates.io
- Keleusma, Command-Line Crate on crates.io
- Keleusma, Application-Programming-Interface Documentation on docs.rs
- Keleusma, Example Scripts
- Keleusma, GitHub Repository
- Keleusma, Hosted Book (mdbook)
- Keleusma, Browser-Based Playground
- Keleusma, Guide, Installing and Running
- Keleusma, Guide, Handling Partial Operations
- Keleusma, Guide, Information-Flow Labels
- Keleusma, Reference, Automation and Scripting
- Keleusma, Reference, Security Policy
- Related Post, Getting Started with Keleusma 0.1.1
- Related Post, Getting Started with Keleusma 0.2.0
- Related Post, A Verifiable Control Kernel in Keleusma
- Related Post, Information-Flow Control, A Deep Dive with Keleusma
- Related Post, Streaming Compilers Series Conclusion
- Related Post, The Self-Hosted Silicon Compiler