

decimal64 — Project Status Report

May 2026 · 7 Cycles Complete

2026-05-30

Contents

decimal64 — Project Status Report	1
1. Current State	1
Features	2
Test counts (as run, 2026-05-30)	2
2. Per-Cycle Delivery	2
Cycle 01 — Design & Foundation (2026-05-26)	2
Cycle 02 — UDecimal64 (2026-05-26)	2
Cycle 03 — Parse Speed (2026-05-27)	3
Cycle 04 — Math-Ops Performance (2026-05-27)	3
Cycle 05 — Scientific Notation (2026-05-28)	3
Cycle 06 — Serde Feature (2026-05-28)	3
Cycle 07 — no_std Compatibility (2026-05-28)	3
3. Test Summary	4
4. Open Items	4
5. Architecture Summary	4

decimal64 — Project Status Report

Date: 2026-05-30

Version: 0.1.0

Repository: git@github.com:dunnock/decimal64.git (local; remote push pending)

Toolchain: Rust stable (1.95.0)

Host: 12th Gen Intel Core i9-12900K · Linux 6.17.0-29-generic

1. Current State

decimal64 is a Rust library crate providing a **64-bit fixed-point decimal type** with a compile-time const-generic scale parameter:

```
pub struct Decimal64<const S: u32>(i64);    // signed
pub struct UDecimal64<const S: u32>(u64);    // unsigned
pub struct Scientific<D>(D, i32);            // scientific-notation wrapper
```

All 7 planned development cycles are complete.

Features

Cargo feature	Default	Notes
std	yes	enables <code>std::error::Error</code> on <code>ParseError</code>
alloc	no	<code>no_std</code> with heap allocation
serde	no	<code>Serialize/Deserialize</code> for all three types; implies <code>alloc</code>

Bare `no_std` (no `alloc`) is fully supported: the core arithmetic and parse logic lives in `core` with no heap dependency.

Test counts (as run, 2026-05-30)

```
$ cargo test
test result: ok. 144 passed; 0 failed; 0 ignored; finished in 0.01s
Doc-tests: ok. 1 passed; 0 failed; finished in 0.10s
```

```
$ cargo build --no-default-features
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.13s
```

Under `--no-default-features --features alloc`: **133 tests pass** (11 fewer because `from_f64/from_f64_round` require `std` float intrinsics on stable Rust and are `#[cfg(feature = "std")]` gated).

2. Per-Cycle Delivery

Cycle 01 — Design & Foundation (2026-05-26)

Established the full API surface and verified it met performance targets before committing to implementation:

- `Decimal64<S>` with `from_raw/raw`, `ZERO/ONE/MAX/MIN` constants
- `FromStr / Display`, `from_f64 / to_f64`, `from_i64 / to_i64`
- `checked_add`, `checked_sub`, `checked_mul`, `checked_div`
- `Ord`, `Eq`, `Hash`
- Baseline parse benchmarks confirmed 100 M/s parse throughput from day one:

Input	decimal64 (ns)	f64 (ns)	rust_decimal (ns)
"0"	4.66	6.36	9.81
"123.4567"	6.42	8.63	11.98
"9999999999.9999"	8.46	8.81	13.54

Cycle 02 — UDecimal64 (2026-05-26)

Added `UDecimal64<S>` (unsigned counterpart): identical API surface to `Decimal64<S>`, stores a `u64` raw value, rejects negative parse input and underflowing arithmetic. Benchmarked at parity with the signed variant.

Cycle 03 — Parse Speed (2026-05-27)

Introduced SWAR/u128-based digit scanning for the fast-parse path, replacing the character-at-a-time loop. Added `#[inline]` to the parse chain and enabled thin-LTO in the bench profile. Post-optimisation, `decimal64` parses ~ 1.3 – $1.9\times$ faster than `f64` and $\sim 2\times$ faster than `rust_decimal` across the benchmark corpus.

Cycle 04 — Math-Ops Performance (2026-05-27)

Introduced the **H2 fast path** in `checked_mul` / `checked_div`: when the intermediate result fits in `i64`/`u64`, use integer arithmetic and the compiler’s magic-constant divide instead of `i128`. Fall through to `i128` only on overflow.

Benchmark	Before (ns)	After (ns)	Improvement
<code>decimal64_mul</code> (scale 4)	4.73	2.81	−36.6%
<code>decimal64_div</code> (scale 4)	4.46	2.59	−42.5%
<code>udecimal64_mul</code> (scale 4)	3.88	3.36	−11.6%
<code>udecimal64_div</code> (scale 4)	3.97	3.19	−17.2%

Cycle 05 — Scientific Notation (2026-05-28)

Added `Scientific<D>` wrapping any `Decimal64`/`UDecimal64` with an `i32` exponent. Parse uses an OR-mask digit-validity scan and `#[inline(always)]` helpers. Benchmarks (scale 4):

Benchmark	Median (ns)
<code>scientific_parse/noexp</code>	12.66
<code>scientific_parse/pos_exp</code>	14.46
<code>scientific_parse/neg_exp</code>	8.93

Cycle 06 — Serde Feature (2026-05-28)

Optional `serde` feature implements `Serialize` / `Deserialize` for `Decimal64`, `UDecimal64`, and `Scientific<D>`. String-path serialization leverages the existing `Display` impl; deserialization uses `FromStr`. Actual performance far exceeded initial budgets:

Operation	Budget	Actual
JSON serialize (<code>serde_json</code>)	500 ns	44 ns
JSON deserialize (<code>serde_json</code>)	100 ns	14 ns
Postcard serialize (raw <code>i64</code>)	—	22 ns

Cycle 07 — `no_std` Compatibility (2026-05-28)

Ported the library to `#![no_std]`. Key decisions:

- `core::` imports replace `std::` throughout
- `from_f64/from_f64_round` demoted to `#[cfg(feature = "std")]` — `f64::powi` requires `libm` intrinsics only available with `std` on stable Rust
- `const_pow10` (a `const fn`) replaces runtime `f64::powi` for scale factors in `to_f64`

- Alloc-dependent code (`Display`, `String`-returning methods) gated on `cfg(any(feature = "std", feature = "alloc"))`

Zero performance impact: all benchmarks within $\pm 5\%$ of pre-`no_std` baseline.

Final benchmark highlights (cycle 07, all median times):

Benchmark	Time
<code>decimal64_add</code>	307.7 ps
<code>decimal64_mul</code>	589.7 ps
<code>decimal64_div</code>	1.954 ns
<code>parse_decimal64/"123.4567"</code>	6.47 ns
<code>parse_f64/"123.4567"</code>	8.66 ns
<code>parse_rust_decimal/"123.4567"</code>	12.55 ns

3. Test Summary

Command	Tests passing
<code>cargo test</code> (default features)	144 + 1 doc-test
<code>cargo test --no-default-features --features alloc</code>	133 + 1 doc-test
<code>cargo build --no-default-features</code>	PASS (0.13 s)
<code>cargo bench --no-run</code>	PASS (4 executables)

The 11-test delta between configurations is fully accounted for by the `#[cfg(feature = "std")]` gating of float-math methods.

4. Open Items

Item	Status
Push to GitHub (<code>git@github.com:dunnock/decimal64.git</code>)	Pending operator action
Crates.io publish	Blocked on GitHub push
<code>no_std</code> bare metal (no alloc) validation	Accepted as out-of-scope for v0.1
SIMD parse optimisation	Deferred; current ~6 ns parse already beats f64
Arbitrary-scale <code>*/÷</code> with automatic scale promotion	Design deferred to v0.2

5. Architecture Summary

```
src/
  lib.rs      - public re-exports, crate-level #![no_std]
  decimal64.rs - Decimal64<S>: parse, arithmetic, display, conversions
```

udecimal64.rs – UDecimal64<S>: unsigned counterpart
scientific.rs – Scientific<D>: scientific-notation wrapper
parse.rs – shared SWAR fast-parse helpers
parse_unsigned.rs
serde_impls.rs – Serialize/Deserialize (feature = "serde")
serde_as.rs – helper for raw i64/u64 serde path

benches/

 parse.rs, arithmetic.rs, scientific.rs, serde.rs

All hot paths are `#[inline(always)]`; the library is `repr(transparent)` throughout so raw i64/u64 storage is zero-overhead at the type boundary.

Report generated 2026-05-30. Cross-checked against git log --oneline and live cargo test / cargo build --no-default-features runs in worktree 08-status-report.