思考了 6m 49s

Yes. After looking at the current GUC machinery and the history behind it, I think your broader conclusion is justified:

> **The next-smaller-unit rounding rule is a poor semantic model for existing timeout GUCs too. The zero-sentinel bug is only its most severe manifestation.**

Current PostgreSQL explicitly documents the behavior: a fractional value with a unit is first rounded to a multiple of the next smaller unit, and an integer GUC is then rounded again to an integer. PostgreSQL +1 Current master implements exactly that: `convert_to_base_unit()` computes the exact base-unit value, then immediately applies an `rint()` at the next-smaller-unit granularity; `parse_int()` later performs another `rint()`. Doxygen +1

The problem is deeper than `positive → 0`.

The real violated invariant

For a quantity such as a duration, I think the fundamental invariant should be:

```
same physical duration
        ↓
same value in the parameter's native unit
        ↓
same final quantization
        ↓
same semantics
```

Unit syntax should be **representation**, not semantics.

Therefore all of these should be equivalent before the final millisecond quantization:

```
0.0001d
0.0024h
0.144min
8.64s
8640ms
8640000us
```

They all mean exactly:

```
8640ms
```

But the present GUC conversion intentionally introduces a quantization that depends on the **input suffix**.

For example:

```
0.144min
    = 8.64s
    ↓ round to next smaller unit ("s")
9s
    ↓
9000ms
```

while:

```
8.64s
    ↓ round to next smaller unit ("ms")
8640ms
```

That is fundamentally strange for a duration parser.

And it gets much worse around the half-unit boundaries. Consider:

```
0.020d = 28.8 minutes
0.021d = 30.24 minutes
```

The first-stage GUC quantization is to hours:

```
0.020d
  = 0.48h
  → 0h
  → 0ms

0.021d
  = 0.504h
  → 1h
  → 3600000ms
```

A tiny change from about 28.8 minutes to 30.24 minutes can therefore change the stored timeout from **zero to one hour**.

For something such as `statement_timeout`, zero is not merely another approximation: zero disables the timeout. PostgreSQL's documentation explicitly assigns that sentinel meaning. PostgreSQL

So I would now characterize the problem as:

```
                    current GUC time conversion

physical duration
       |
```

```
      ▼
input-unit-dependent quantization      ← unjustified loss
      |
      ▼
native-unit quantization               ← unavoidable loss
      |
      ▼
runtime semantics
```

The first quantization should not exist for timeout semantics.

The history actually strengthens this argument

The 2019 change was not introduced because a timer required this behavior.

The commit explanation uses:

```
work_mem = 30.1GB
```

and says that directly converting to the native unit would yield the rather unattractive:

```
31562138kB
```

so PostgreSQL instead rounds to:

```
30822MB
```

The stated concern was obtaining a “reasonable approximation” rather than a “silly-looking” setting. PostgreSQL

That is understandable for configuration presentation. It is much less compelling as a rule for interpreting a deadline.

In fact, the discussion that led into this work started around allowing units and fractional values for `vacuum_cost_delay`, and Tom Lane explicitly mentioned wanting inputs such as `0.1GB`; the generic unit machinery then ended up serving both memory and time quantities. PostgreSQL

So there is a mismatch of abstraction:

```
historical requirement:
    produce human-friendly configuration approximations

timeout requirement:
    preserve requested duration until forced
    to quantize by actual timer resolution
```

Those are different contracts.

There are actually three separate issues

I would distinguish them, because that gives you a much stronger argument on hackers.

1. **Validation after lossy conversion is a correctness bug.** Current configuration processing calls `parse_int()` / `parse_real()` first and performs the GUC's min/max check afterward. Doxygen Consequently, a value that is genuinely below the minimum can be rounded into the valid range. `-0.0001d → 0` for a `min = 0` GUC is the obvious example. This is difficult to defend even as legacy behavior.
2. **Next-smaller-unit rounding is a semantic-design problem for durations.** `8.64s` and `0.144min` should not mean different durations after parsing. Even when neither encounters a sentinel, 8640ms versus 9000ms demonstrates the abstraction problem.
3. **Native-resolution quantization needs its own policy.** Once you have exactly `1.1ms` but the runtime interface accepts integral milliseconds, some information genuinely must be lost. For a deadline/timeout, `ceil()` is the natural rule because it preserves:

   ```
   effective timeout >= requested timeout
   ```

   and preserves the much more important invariant:

   ```
   requested > 0  =>  effective > 0
   ```

Those three issues should not be conflated.

This changes how I would revise your report

I would no longer recommend the report's conservative end state in which legacy nonzero rounding remains accepted indefinitely.

In particular, an expected result like:

```
0.0001d   → 8640ms
0.0024h   → 8640ms
0.144min  → 9000ms    ← preserve legacy behavior
8.64s     → 8640ms
```

is architecturally inconsistent.

If the premise of the report is now “the time-GUC conversion contract itself is wrong,” then the target should instead be:

```
0.0001d   ┐
0.0024h   │
```

```
0.144min  ├── exact conversion ──→ 8640ms
8.64s     │
8640ms    │
8640000us ┘
```

That is a much cleaner specification:

> **Convert time units exactly. Quantize only once, at the resolution of the parameter's actual representation.**

For a timeout whose zero means “disabled,” I would add:

> **When positive durations must be quantized to integral time units, round upward so that quantization cannot shorten the requested timeout or turn it into the zero sentinel.**

That is much stronger than saying merely “prevent positive values from becoming zero.”

I would also change the implementation architecture slightly

Rather than making “rounded conversion” the primitive:

```c
convert_to_base_unit(..., bool round, ...)
```

I think the cleanest primitive is the lossless one:

```c
convert_to_base_unit_exact(...)
```

Then any configuration-specific approximation is explicitly layered on top:

```
                ┌─ memory/config presentation policy
parse numeric ──┤
                └─ exact duration semantics
```

Conceptually:

```c
double exact;

parse_numeric(...);
convert_to_base_unit_exact(value, unit, base_unit, &exact);

/* Semantic validation operates on what the user actually specified. */
validate_range(exact);
```

```
if (timeout_semantics)
    stored = exact > 0 ? ceil(exact) : exact;
else if (legacy_guc_rounding)
    stored = ...;
```

This matters because the exact quantity is the **semantic value**. Rounding is a policy imposed on that quantity. A conversion routine should not destroy the semantic value before anybody has had a chance to validate or interpret it.

I therefore agree with your earlier instinct that `GUC_NO_ROUND` is an awkward abstraction. The strange operation is not “exact conversion”; the strange operation is the legacy approximation. Exact conversion should ideally be the primitive.

Would I change all existing time GUCs at once?

This is where I would be more conservative strategically than architecturally.

I think there is a solid case for eliminating next-smaller-unit rounding for **time-unit GUCs**. But this behavior has been documented and shipped since PostgreSQL 12, so changing all existing settings is a compatibility change, even if the existing behavior is poor. The 2019 commit deliberately established it as user-visible behavior. PostgreSQL +1

I would therefore separate patchability from design correctness:

```
WAIT FOR LSN
    exact units + ceil
    └─ unquestionably appropriate; new interface

generic GUC range validation
    validate exact converted value before lossy rounding
    └─ correctness fix, potentially discuss backpatching

existing time GUC unit conversion
    remove next-smaller-unit quantization
    └─ master-only behavioral change

timeout-specific final quantization
    nearest → ceil
    └─ separate discussion because not every time-valued
       GUC necessarily has deadline semantics
```

That last distinction matters. I would **not** argue that every `GUC_UNIT_MS` variable must automatically use `ceil()`. A polling interval, cost delay, logging interval, replication grace period, and hard timeout do not necessarily have identical rounding contracts.

But I *would* argue that all of them should receive an **exact unit conversion first**. Whether their eventual native-resolution conversion uses nearest, floor, or ceiling is a second

semantic decision.

That decomposition makes the proposal much harder to object to.

The sentence I would build the broader argument around

I think the essence is:

> The existing GUC rounding rule conflates unit conversion with value quantization. For time-valued settings, the unit suffix should only describe the same physical duration in a different representation; it should not determine the precision at which that duration is interpreted. Conversion should therefore preserve the exact duration in the parameter's base unit, with any necessary rounding applied only once at the parameter's actual storage or execution resolution.

And for timeout GUCs specifically:

> For timeout parameters, that final quantization should not shorten a positive requested duration, and in particular must never transform it into a sentinel such as zero that has qualitatively different semantics.

I think that is a substantially stronger first-principles argument than the current report's “protect zero from rounding” framing. **The zero collision is evidence of the abstraction error; it isn't the abstraction error itself.**