



Core C++ 2026

20-21 Oct. 2026 :: Tel-Aviv

When a Broken Clock Is Right 23 Times a Day

Mike Lindner



More – in my bilingual blog: <https://sw-arch.blog/>
Architect in Slippers / ארכיטקט בכפכפים



Intro

What this is about





Intro

Who I am

Who you are

What this talk is about

“Severity” of Issues



Found in Dev

- Break compilation
- Break linkage
- Immediate crash
- Wrong logic/math



Found Too Late

- Found in prod
- *Not* found in prod



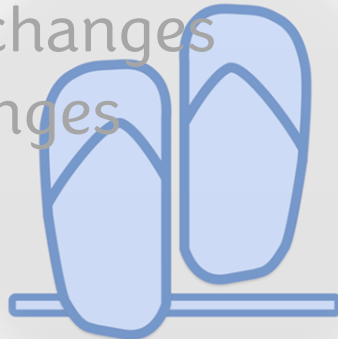
Found in Lab

- Occur sometimes
- Occur rarely



Waiting in the Shadows...

- Exist only in specific compilations
- Might occur if (other) code changes
- Might occur if platform changes



Clock Types

The fundamental clock categorization



Standard Clocks

By nature, clock and time are **not absolute** terms.

ANSI C / C++98 provided basic time facilities,
but no general clock abstraction or standard monotonic clock.

Time support was provided by other means.

- POSIX
- Other OSs APIs
- HW

Over the years, the domain become more standardized.



Clock Categories

Wall Clocks

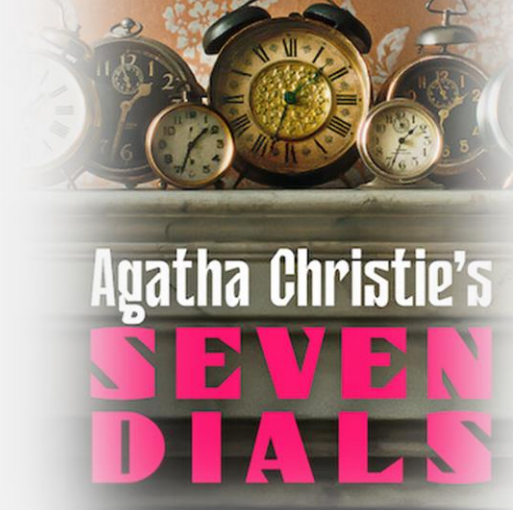
- `time()` [C90]
- `CLOCK_REALTIME` [POSIX]
- `TIME_UTC` [C11]
- `chrono::system_clock` [C++11]
- `chrono::utc_clock` [C++20]

Monotonic Clocks

- `CLOCK_MONOTONIC` [POSIX]
- `TIME_MONOTONIC` [C23]
- `chrono::steady_clock` [C++11]

Other Clocks

- `clock()` [C90]
- CPU / Core clock



C++ Clocks

C++'s `std::chrono` provides those clocks:

`system_clock`

`steady_clock`

`high_resolution_clock`

(and from C++20, a few additional ones).

What are they for?



The `chrono::steady_clock`

This clock represents some **internal time**.

E.g. – “3,002,478 tick updates from some random moment”.

But it *is* **monotonic**.

It can never be adjusted.



The `chrono::system_clock`

This clock represents the **system time**, or “wall clock”.

E.g. – “17-Mar-2030, 17:22:05.720”.

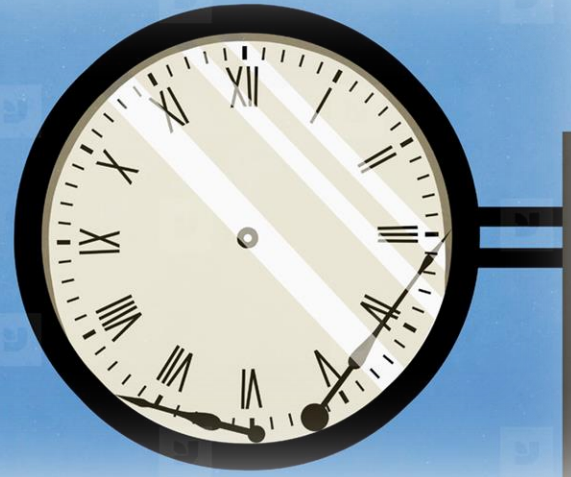
And it is **non-monotonic** –

[Or, to be precise: it is not *guaranteed* to be monotonic]

It might be adjusted (e.g., manually, or by NTP).



chrono::high_resolution_clock



Well, that's just a deceit:

It can be an alias to `system_clock`
or to `steady_clock`,
or be another clock.

Theoretically, it is the clock with the highest **representation** resolution.
It does not mean it really **counts** in that resolution.

So, it *may* or *may not* refer to the **system time**.
And it *may* or *may not* be **monotonic**.



What We're Focusing

If you want to schedule an event to
“Sunday at noon” –
use `system_clock`.

If you want to schedule an event to
“15ms from now” –
use a monotonic / steady clock.

So this is what relevant to RT systems.



The steady_clock's Limits

The standard steady_clock is not always enough for RT.

Granularity

Allow using finer timestamp or smaller intervals scheduling

Quality

Allow using a clock with higher accuracy and precision.

Synchronization

Be tightly coupled to other CPUs, devices, protocol or mechanisms.



External Clocks

To address those shortcomings,
Real-time systems widely use other clocks as well.

For example:

- Dedicated HW timers [e.g. very high-resolution timer]
- External reference timers [e.g. a reference clock on a multi-CPU SoC]
- Synchronization pulses [e.g. 1PPS or 10PPS]
- Calculating absolute time [e.g. extract GPS time from satellites]



External Clocks Meaning

“Real” clocks:

Devices that are fully-functional clocks and stand on their own.

Dedicated HW timers, external reference timers, ...

“Sync / adjust” clocks:

Means that help syncing and adjusting *other* existing clocks.

Synchronization pulses, calculating GPS time, ...



Sync / Adjust Clocks Examples



We use “clock” terms, but it refers to any other type of “time”.

E.g.: `time_point_from_sys_up()` may tell the time elapsed since a specific “epoch” when the system was up and fully initialized.

It may be as simple as `{ return now() - sys_up_epoch; }`.

E.g.: `pps_duration_to_local(ref_dur)` may take a duration in terms of some *ref clock* that signals every 1sec and translate it to actual duration in terms of the *local clock*.

So, if between any two consecutive signal (ref “real” second), the local clock counts *1003ms*, then calling it with *2000ms* will return *2006ms* – meaning that for counting two “real” secs, the local clock will count *2006ms*.



Handle with Care!

If your system have several different clocks...

Don't use the wrong clock.

More important: Don't mix-up clocks!





Lessons | Clock Types

- Use the **right** clock for the right purpose.
- For measuring / scheduling in specific periods – **monotonic** clocks
- There may be several “clocks” in a system, for **different needs**
- “**Real**” clocks use counters; “**adjusting**” clocks fix/align other’s time

Basics Terminology

What defines a clock



Read vs. Wait / Schedule

A “clock” can merely allow one to *read* its timestamp.

```
std::chrono::steady_clock::now()
```



Scheduling an event is a much more complicated action.

```
std::this_thread::sleep_for()
```



Clock Units

Representation resolution (or nominal resolution):

Smallest unit representable by the clock's type.

E.g., *1 μ s representation resolution* means that the values are represented in whole μ s.

Granularity (or actual/effective resolution):

Smallest observable increment of the clock.

E.g., *granularity of 15 μ s* means that if the current timestamp is 3060 μ s, the next reads will be 3075 μ s, 3090 μ s etc.

Note: The term *resolution* is commonly used for both concepts!



Clock Correctness

Accuracy:

Closeness to “true” or reference time.

E.g., if 1,000.000s pass and the clock measures 1,000.003s, its error over that interval is 3ms.

This corresponds to an *average rate error* of 3 ppm.

Precision:

Repeatability of measurements.

E.g., if you measure the same nominal 1ms interval many times and get 1000.2 μ s, 1000.1 μ s, 1000.3 μ s, 1000.2 μ s, ... you could say that *precision* is: $\sigma = 0.08\mu$ s, where σ is the standard deviation.



Scheduling Delays

Scheduling events suffer of some overhead before running.



For example, `sleep_for(10ms)` may actually resume after 11.6ms.

Latency:

How late the action actually occurs relative to its deadline.

E.g., 1.6ms in the above example.

Jitter:

Variation in that latency over repeated runs.



Meaning of “Tick”

The term may refer to one of those, according to context:

- Hardware counter tick
- `std::chrono::duration/clock` tick
- OS/scheduler tick

For example, on a given system, their values can be:

- hardware counter tick: 10ns
- chrono tick: 1ns
- RTOS scheduler tick: 1ms



Terminology of `std::chrono`



- **time_point**
 - a position on a particular clock's timeline (analogous to “timestamp” like 17:32:05)
- **duration**
 - a relative amount of time (e.g. 130ms or 50,226μs)
- **count()**
 - the *raw* number stored in a duration.
- **period**
 - compile-time ratio of 1sec – the unit of one count (e.g. `std::milli` = 1/1000s)
- **rep**
 - the underlying numeric type used for the count (e.g. `int64_t`, `double`)
- **Epoch**
 - the clock-specific instant corresponding to `time_point{}` / time zero.





Lessons | Basic Terminology

- Use the right clock for the right purpose.
- For measuring / scheduling in specific periods – monotonic clocks
- There may be several “clocks” in a system, for different needs
- “Real” clocks use counters; “adjusting” clocks fix/align other’s time
- A “clock” allows only to **read** time
- **Scheduling / waiting** are different mechanism, that *use* clocks

Timer Correctness

What defines the difference between clocks



Reading Time Factors

- Granularity of the (HW) timer
- Representation resolution of the (SW) timer
- Number of bits in the HW counter and in SW timer types

Larger types allow calculations on larger periods

E.g., assume a signed counter where tick is $1\mu\text{s}$.

A 32bit counter wraps around after about half an hour;
a 64bit counter will keep steady for more than 290 millennia.



Reading Time Factors (cont.)

- **SW access time for reading the timer**
 - Does it require system call or not?
 - Does it require syncing mechanism?
 - How complex is the calculation from HW counter to SW time?
- **HW architecture**
 - Clock distribution, e.g., is it “pushed” to CPU or CPU access external HW?
 - Internal CPU arch, e.g., can each core directly read time w/o sync?
- **Accuracy**
- **Precision**



Setting Timers Factors

- Minimum programmable deadline

E.g., it might take $10\mu\text{s}$ to set a deadline even if granularity is $2\mu\text{s}$

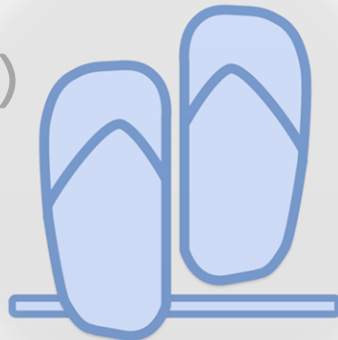
- Scheduler/OS tick [mainly in RTOSs]

E.g., if it is 1ms , then even sleeping for $50\mu\text{s}$ might wait close to 1ms

- Actual task wakeup latency/jitter

Depends on HW (comparator, interrupt, ...)

and SW (IRQs priorities and masking, task/thread priorities, ...)





Lessons | Timer Correctness

- Use the right clock for the right purpose.
- For measuring / scheduling in specific periods – monotonic clocks
- There may be several “clocks” in a system, for different needs
- “Real” clocks use counters; “adjusting” clocks fix/align other’s time
- A “clock” allows only to **read** time
- **Scheduling / waiting** are different mechanism, that *use* clocks
- **Quality** of clocks/schedulers includes several aspects

Arithmetic

Time math



Overflow/wraparound

The size of HW and SW counters define the maximum duration they can deal with.

If the HW counter overflows:

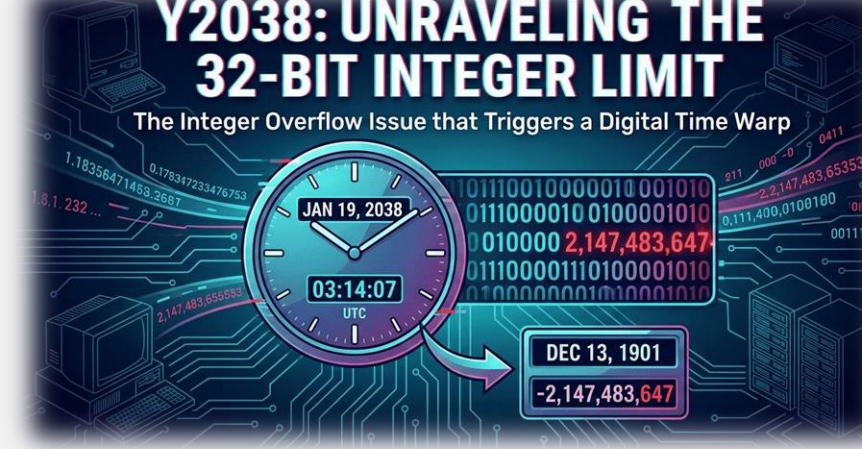
Two counter reads at t_1 and t_2 cannot be directly compared!

This can be handled by the SW clock mechanism.

If the SW counter overflows:

Two counter reads at t_1 and t_2 cannot be compared.

Comparing two time-points – up to **half range**.



Unix Epoch:

00:00:00

January 1, 1970

Signed or Unsigned?

It makes sense that the HW counter will be unsigned.

It's one job is to tick forward.

Making the SW clock unsigned might sound tempting, but...

- Using unsigned **time points** –
no way to represent a time before the epoch.
- Using unsigned **durations** –
calculating the delta $d_1 - d_2$ may *underflow*.
- One is signed and the other unsigned –
all the common issues of mismatching types.



Type Safety

Embedded systems code-base **last many years**.

They tend to get newer-standard compilers **very late**.

They usually use **their own** special clocks.

HW-specific API and OS API are usually **provided in C**.

And – the secret ingredient –

the background of many RT/Embedded programmers
is from early C++, C, or even electric engineering.

The result:

Many systems do not have **type safety** means.



Type Safety Importance

Type safety is important in any system.
But clocks and times in special risk when not keeping it.



Many years of experience prove that this is not only in theory.
Clocks and time bugs are very common. Still.

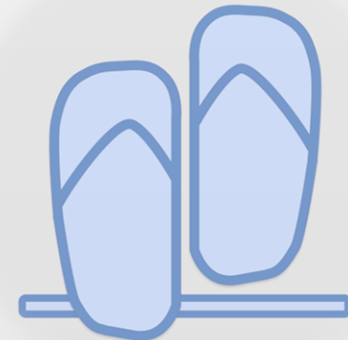
Using *native types*, or *poorly crafted ones*,
for **times** and **clocks** is practically dangerous.



Mismatching Time Elements

Time elements should follow pointer-like arithmetic.

- `duration + duration → duration`
- `duration - duration → duration`
- `time_point + duration → time_point`
- `time_point - duration → time_point`
- `time_point - time_point → duration`
- `time_point + time_point → has no meaning!`



Mismatching Units

Consider this code:

```
8  // Sleep for <interval> microseconds
9  void deep_sleep(int64_t interval);
10
11 void go_sleep(int32_t element_count)
12 {
13     int32_t min_waiting = get_min_wait();
14     int32_t waiting_per_element = get_per_element();
15
16     int32_t total_time = min_waiting + element_count * waiting_per_element;
17
18     deep_sleep(total_time);
19 }
```

Apparently, `deep_sleep()` uses microseconds.

But what about `get_min_wait()` and `get_per_element()`?



Mismatching Clocks

Combining values from two different clocks.

For example:

- Adding a duration in terms of one clock to a time-point of another clock.
- Setting a callback on some timer, with values taken from reading another.



Double Adjustment

Recall the sync / adjust “clocks” we mentioned, like

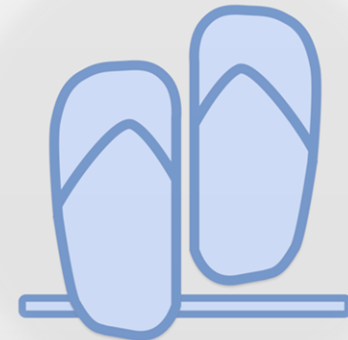
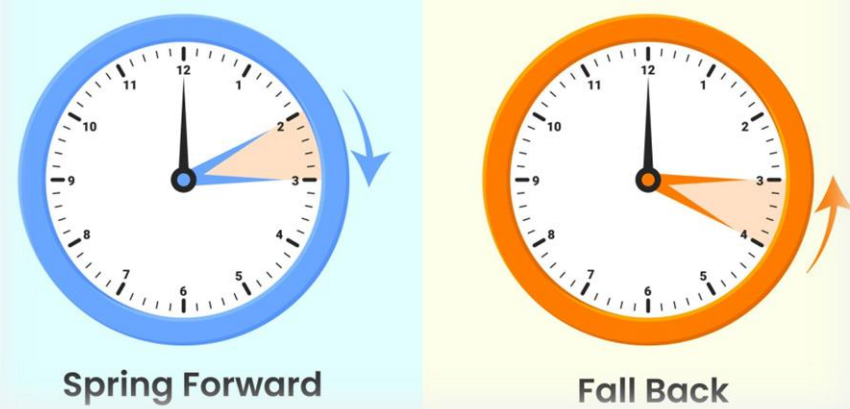
- `pps_duration_to_local(ref_time)`

```
5 void do_something(int32_t dur)
6 {
7     int32_t pps_dur = pps_duration_to_local(dur);
8
9     // ...
10 }
```

If someone calls `do_something()` with an already adjusted value, the result will be wrong.

We want the compiler in our side.

Without well-crafted time types, bugs can be easily added.



“Illegal” Time

We should deal with “illegal” time values.

Time vars/objs that are –

- Uninitialized
- The result of some bad calc
- No longer reliable (e.g. drifted too much)

They may hold some random value,
or some predefined “illegal” value, like -1 or -99999.

Both may end up in a disaster if used where time is required.





Lessons | Arithmetic

- “Real” clocks use counters; “adjusting” clocks fix/align other’s time
- A “clock” allows only to **read** time
- **Scheduling / waiting** are different mechanism, that *use* clocks
- Quality of clocks/schedulers includes several aspects
- Type-safety design of clocks and timers is essential
- Time elements should adhere to pointer-like arithmetic
- Time units should be clear, safe and work right in expressions
- Different clocks/time types must not mix w/o conversion
- Beware of redundant conversion (“double adjustment”)
- Illegal time representation / behavior must be considered

std::chrono

What it does and what it does not

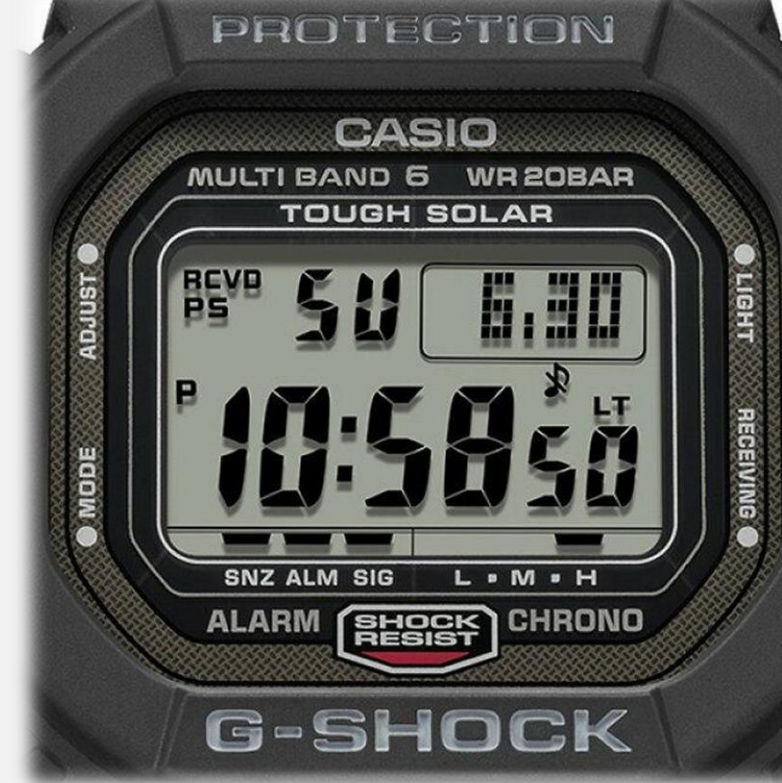


std::chrono

We have seen some std::chrono elements.

The library provides:

- **Built-in clocks** (system_clock, steady_clock, ...)
- Categorizing clocks to **steady** or **not steady**
- Clear definition of **time-unit types**
- Keeping the **arithmetic** of those types
- Implicit and explicit **casting** when relevant
- Additional “**goodies**”, like formatters etc.
- Means to define **new clocks**



A chrono Clock Structure



Any clock provided by `std::chrono` includes the following:

- `rep` underlying numeric counter type
- `period` duration of one counter unit (as a ratio of 1sec)
- `duration` a time interval (using `rep` and `period`)
- `time_point` point on this clock's timeline
- `is_steady` whether it can ever go backward
- `now()` returns the current `time_point`



Keep duration / time_point math

`std::chrono` prevents wrong expressions,
like adding a `time_point` to another `time_point`.

So, it solves the **time element mismatching** issue.



Automatic Unit Conversion

`std::chrono` **implicitly** converts one unit to another, if it is **lossless**.
If the conversion is **lossy**, the compilation will break.
For a **deliberate** lossy conversion, there is **explicit** casting.

```
4  using namespace std::chrono;
5
6  // OK: exact implicit conversion
7  milliseconds ms = seconds{2};
8  // ERROR: would lose 500ms
9  seconds s = milliseconds{2500};
10 // OK: explicit lossy casting; s2 will be 2sec
11 seconds s2 = duration_cast<seconds>(milliseconds{2500});
```

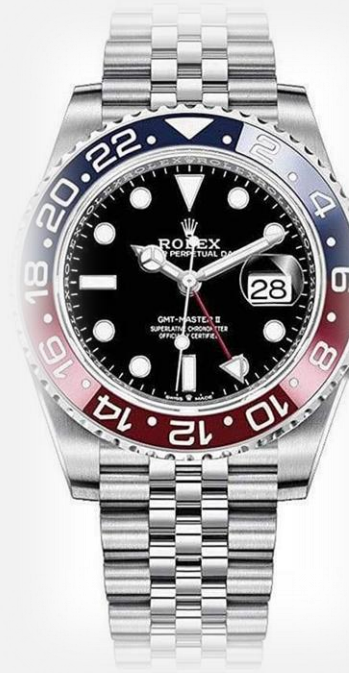
So, it solves the “*unit mismatching*” issue.



Automatic Unit Conversion (cont.)

The same is true for time_points.

```
4 using namespace std::chrono;
5
6 time_point<steady_clock, milliseconds> t_ms =
7     time_point<steady_clock, seconds>{}; // OK: lossless
8
9 time_point<steady_clock, seconds> t_s =
10     time_point<steady_clock, milliseconds>{}; // ERROR: lossy
11
12 auto t_s2 =
13     time_point_cast<seconds>(
14         time_point<steady_clock, milliseconds>{}); // OK: explicit
```



Automatic Unit Conversion (cont.)



Implicit conversion is not only to a known type.

If a lossless conversion is possible, the compiler will detect the destination type automatically.

```
4  using namespace std::chrono;
5
6  // type: milliseconds; value: 2500ms
7  auto d1 = seconds{2} + milliseconds{500};
8
9  // type: microseconds; value: 2500us
10 auto d2 = milliseconds{2} + microseconds{500};
11
12 // type: microseconds; value: 1'000'001us
13 auto d3 = seconds{1} + microseconds{1};
```

But it cannot protect from **overflow**!



Automatic Unit Conversion (cont.)



Note:

The `std::chrono` implicit casts can protect from **lossy** conversion;

```
4  using namespace std::chrono;
5
6  using sec16 = duration<uint16_t>;
7  using ms16  = duration<uint16_t, std::milli>;
8
9  sec16 sec{100};
10 ms16 milli = sec;      // implicit conversion: seconds -> milliseconds
11
12 // mathematically: 100,000 ms
13 // but uint16_t can hold only 0..65,535
```

But they *cannot* protect from **overflow**!



Illegal Values

- Uninitialized duration may have unknown value.

- `std::chrono::milliseconds d; // unknown value`

- Uninitialized `time_point` defaults to the epoch.

- `std::chrono::steady_clock::time_point t;`
`t.time_since_epoch() == steady_clock::duration::zero()`

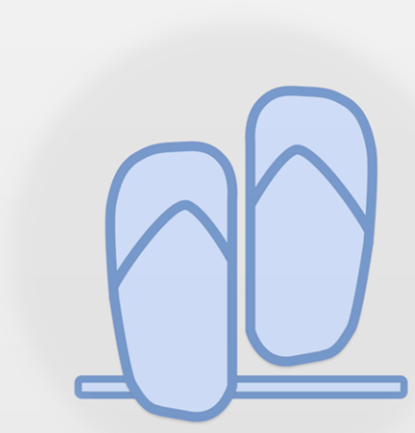
- Both are **not** protected from issues like unsigned overflow, UB on signed overflow, or values like NaN in FP.



Defining New Clocks

New clocks that follow `std::chrono` defs will automatically benefit from its advantages.

However, this is not always enough, and sometime, does not suit at all, for the needs of RT embedded clocks and timers.



Mismatching Clocks – What Works

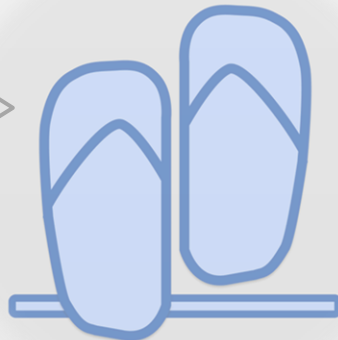
`time_points` in `chrono` are related to their clock.

Therefore, `time_points` of different clocks are type safe.

- They cannot be assigned by mistake to the wrong clock type
- They cannot be used together in expressions like `tp_c1 - tp_c2`

Explicit conversion can be done.

- using special `from_x()` / `to_x()` funcs
- by providing a `std::chrono::clock_time_conversion<Dest, Source>` specialization and using `clock_cast<>` (since C++20).



Mismatching Clocks – What's Not

Conversion can be done for

- different epochs
- different periods.

Durations (unlike time-points) are independent units – not tied to any clock.

Therefore, in chrono semantics,

“30ms period measured on **clock X**”

cannot be different than

“30ms period measured on **clock Y**”.



Clock Types

Whether you take advantage of `std::chrono` clock definition, or define a whole new type – make sure to keep type safety principles.

This is the key to avoid some of the nastiest bugs.





Lessons | `std::chrono`

- Time elements should adhere to pointer-like arithmetic
- Time units should be clear, safe and work right in expressions
- Different clocks/time types must not mix w/o conversion
- Beware of redundant conversion (“double adjustment”)
- Illegal time representation / behavior must be considered
- The `std::chrono` provides a **standard interface** for clocks
- It provides **type-safety** for time elements math and usage
- It is **not bullet-proof**, and can still result in wrong values
- RT systems might have clock/time needs that **do not fit it**
- Clock/time **type-safety** is crucial for a robust system

Periodic Timers

The bread and butter of RT/Embedded systems



A Simple Requirement

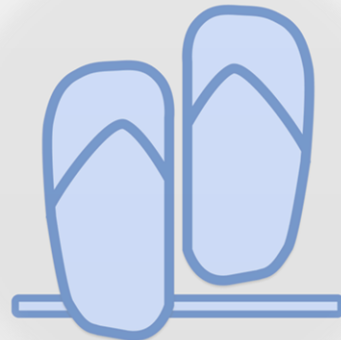
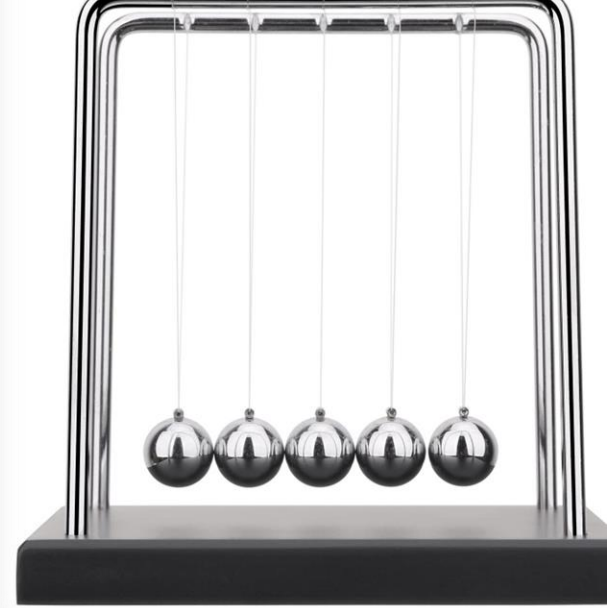
Periodically repeated actions are all over the place.
Especially in RT/Embedded systems.

So, the requirement is quite simple.

- Let the action do its thing for the first time.
- Run it again after some fixed interval of time.
- And again, and again...

A common, simple pattern is to just put it in a loop:

```
while(true) {  
    do_your_thing();  
    sleep(time_interval);  
}
```



But This Does Not Achieve The Goal

If this is what we do (using `std::chrono`, `std::this_thread`):

```
while(true)
{
    do_your_thing();
    sleep_for(time_interval);
}
```

Then the actual time between any call to `do_your_thing()` includes both the **time of sleep** and the **time of work**.



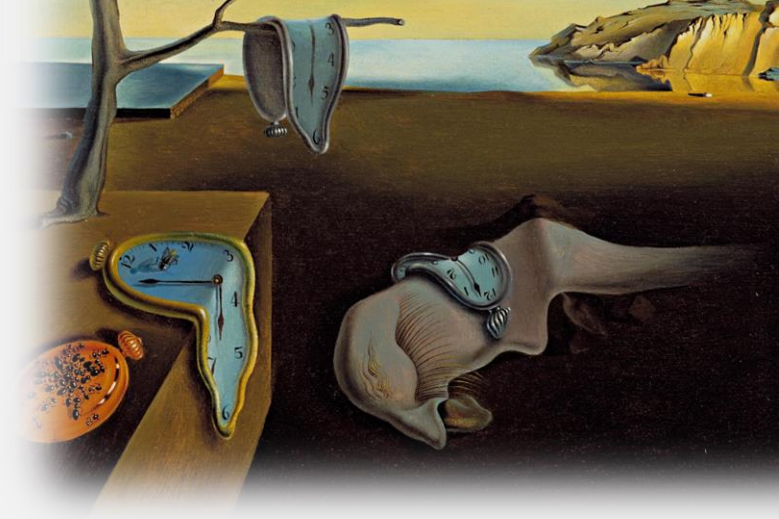
Is This Bad?

Not necessarily.

If the work is not required to run in exact periods,
but only “not too often”,
this may be okay.
Though not very elegant.

But

- we lose control of its de-facto behavior
- over time, it drifts more and more
- at some point, it may fall on another activity and interfere it.



Fixed Interval Solution

A better version of this loop might be:

```
auto next = std::chrono::steady_clock::now();
while (true)
{
    do_your_thing();
    next += time_interval;
    std::this_thread::sleep_until(next);
}
```

This version is much more **deterministic**,
and it keeps the phase over time.

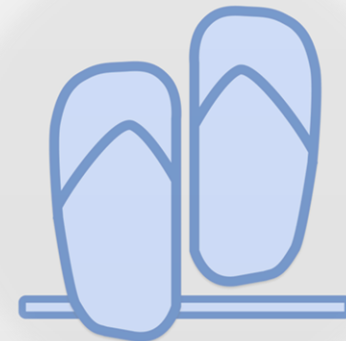


Timer Setting Limitations

Even if the next deadline is known exactly,
the actual deadline set in the HW timer might be a bit late.

For example, due to issues like –

- timer granularity / quantization
- minimum programmable delay
- other hardware/programming limitations



Execution Delay



In addition,
there's a delay between the timer expiration and the actual run.

For example, due to –

- interrupt latency
- scheduler latency
- core wakeup latency
- higher-priority work
- contention / OS overhead



Latency and Jitter

Combining the **timer limitation** and the **execution delay**, provides the **latency** and **jitter** of the periodic work.

Latency:

The whole combined delay

We can define the average latency, worst-case latency, etc.

Jitter:

The statistical difference between measured latencies

In critical tasks, it is important to experimentally measure the actual latency and jitter to verify time bounds.



Missed Deadlines



What happens if at some point, the deadline is already missed?
Ignoring that case may result in an unreliable system.

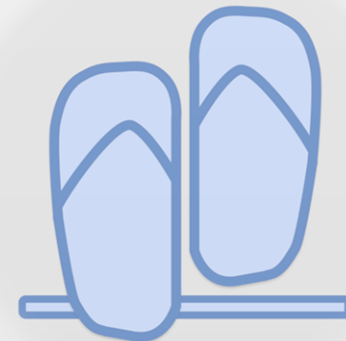
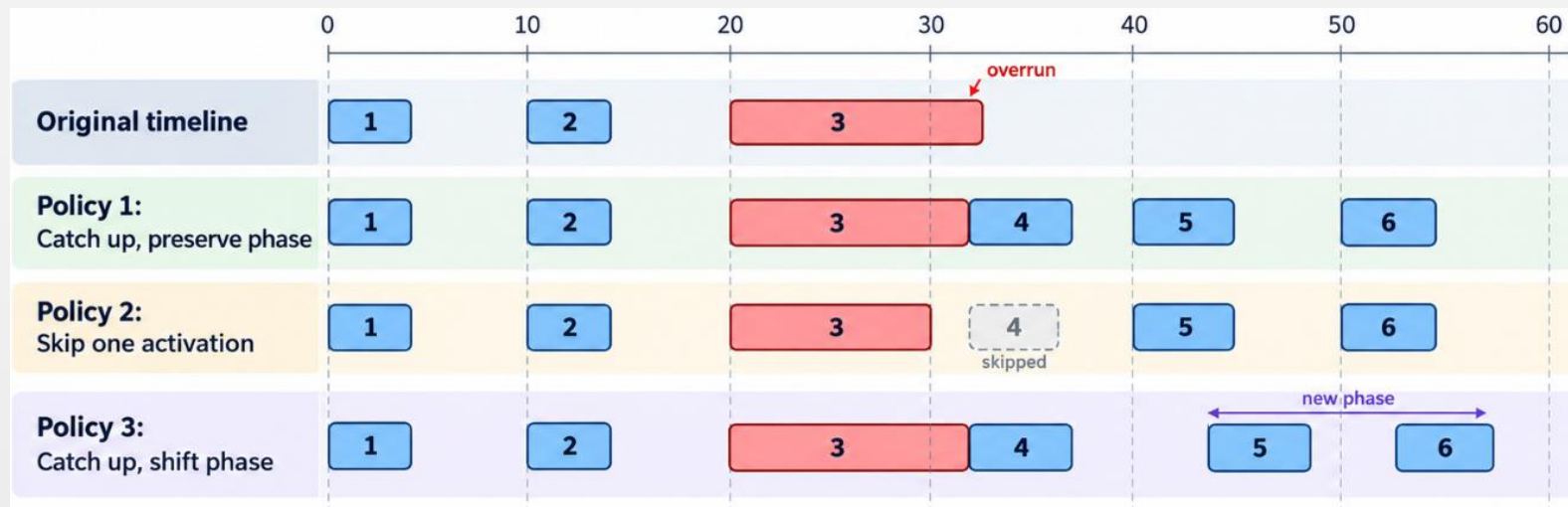
A clear policy must be applied, according to the specific case.



Missed Deadlines

Possible policies are –

1. Execute immediately, and keep next period as it is
2. Skip the action this time, and wait for the next period
3. Execute immediately, and re-schedule the next period – effectively, changing the period phase





Lessons | Periodic Timers

- It provides type-safety for time elements math and usage
- It is not bullet-proof, and can still result in wrong values
- RT systems might have clock/time needs that do not fit it
- Clock/time type-safety is crucial for a robust system
- Naïve `sleep()` loops do not keep the period and **drift** over time
- In RT systems, **predictability** is not a luxury; drift may be risky
- Use `sleep_until()` to keep the period fixed
- Know your timer **limits**: granularity, min. programming time, ...
- Understand and measure **the latency and jitter** of task execution
- Do not ignore **missed deadlines** – implement a clear policy

Timer Storms

That's where the 23 hours a day are not enough



What is a Timer Storm?

A and B are two unrelated periodic tasks.

Each of them is designed to run on its own.

Each of them assumes “normal” system load when triggered.

Say that A intensively works for 2ms, and B for 3ms.

If each of them run at a different point,
and the system load is reasonable –
Everything's fine.

But what happens if they try to run simultaneously?



What Does The Storm Cause?

Two intensive tasks running in parallel may cause –

- latency/jitter spikes
- missed deadlines
- queue buildup
- lock/cache/memory/bus/I/O contention
- collateral damage to unrelated tasks

In the “best” case, the two tasks will fail to keep their bounds.
In worse cases, the damage may spread to other tasks,
and even bring the whole system down.

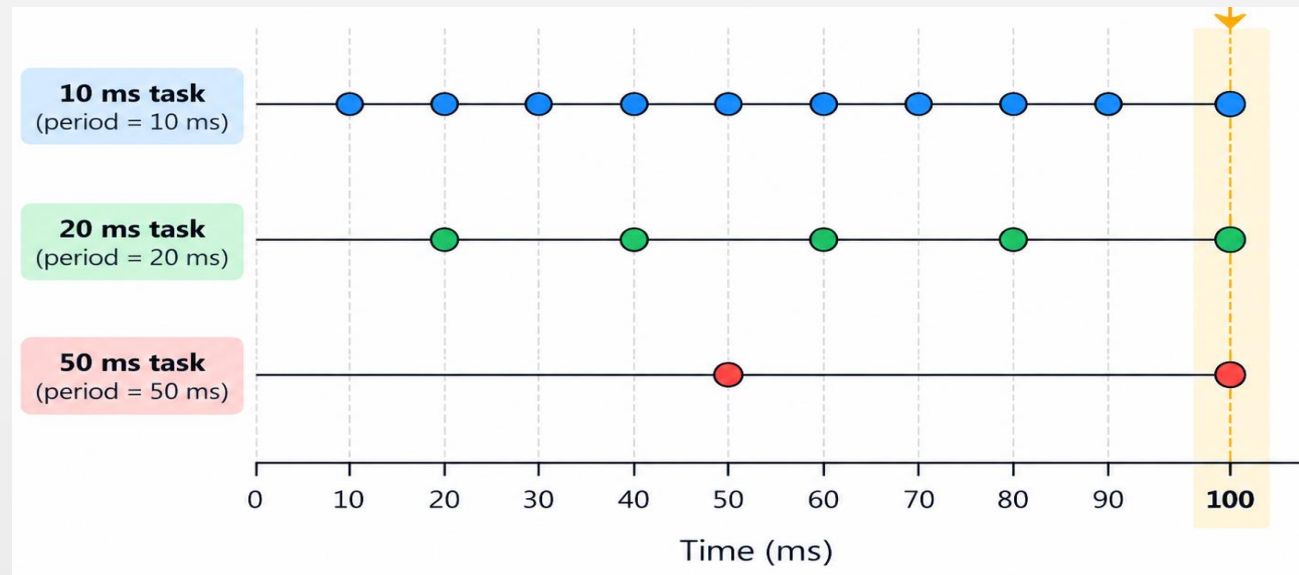


Why Do Timer Storm Happen? ^[1]

The most direct reason:

Having several periodic tasks with the **same phase** and the **same interval** – or with different intervals that have a **common multiple**.

For example:
10ms, 20ms, 50ms –
will all run together at $t = 100ms$.



Why Do Timer Storm Happen? [2]



Another case:

When several tasks are triggered by the same event.

For example:

Tasks A, B and C are all triggered by the PPS signal.

These cases are usually simpler:

They will happen all the time,
and they are easier to see at design.



Why Are They So Nasty? [3]

Some types of storms may only happen after a lot of time, in unpredictable times, and easily sneak from the programmer eyes.

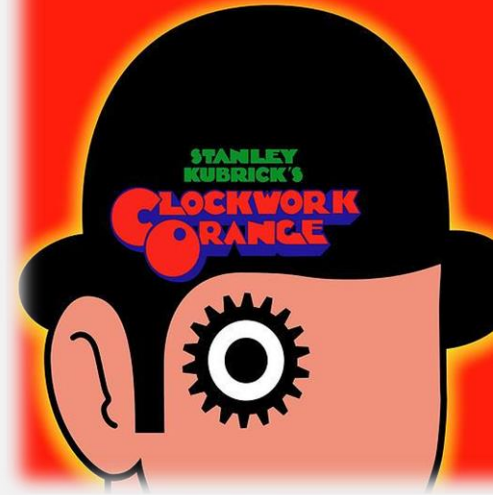
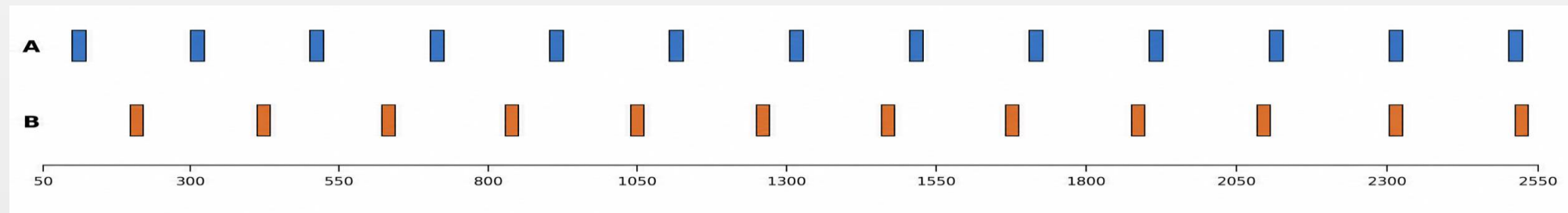
That's because they are not related to **nominal periods**, but to **drifted** ones.

For example:

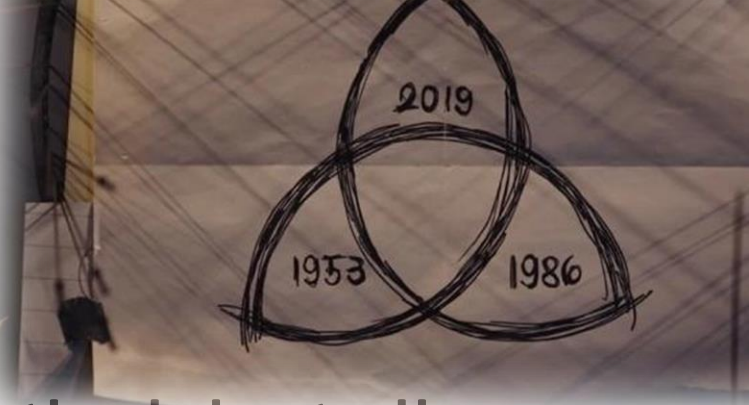
Tasks A and B both run every 200ms. A starts at $t = 100\text{ms}$, B at $t = 200\text{ms}$. They **should** run perfectly without interfering.

But what if B **drifts** a little – e.g. 1ms per hour?

After close to 100 hours, B will run together with A for several cycles.



And Even Nastier?



Some potential storms might not be detected in the lab at all.

A common case:

Two periodic tasks that are not tied to the same timeline.

Consider this:

Task A takes 2ms.

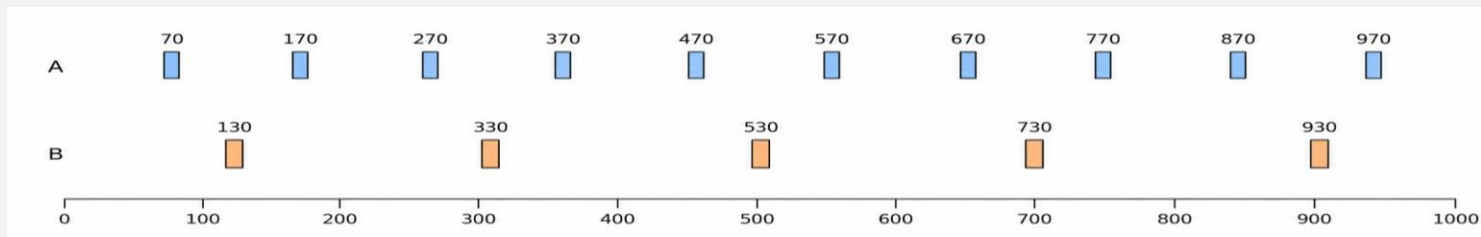
It runs exactly 35ms after each 10PPS signal.

Task B takes 3ms.

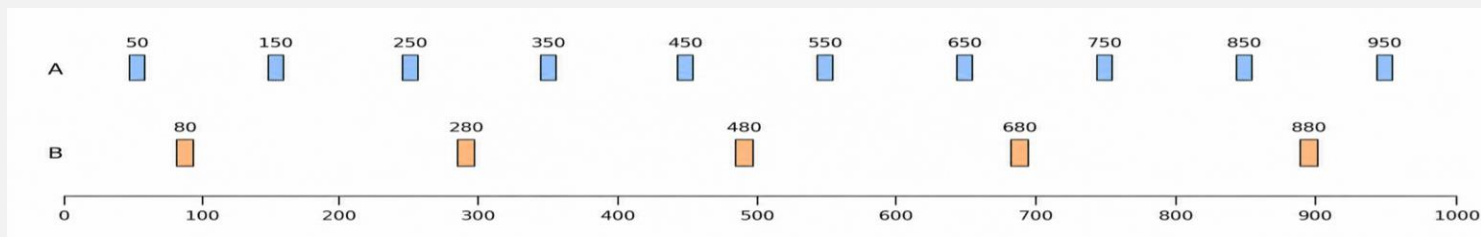
It runs when first connected to some other device,
and every 200ms from that moment.



The timeline in the first lab test:



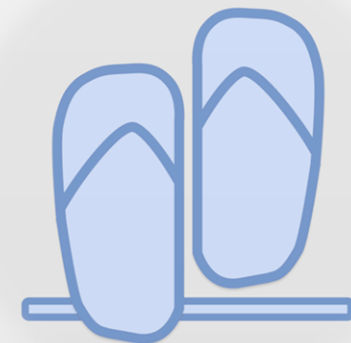
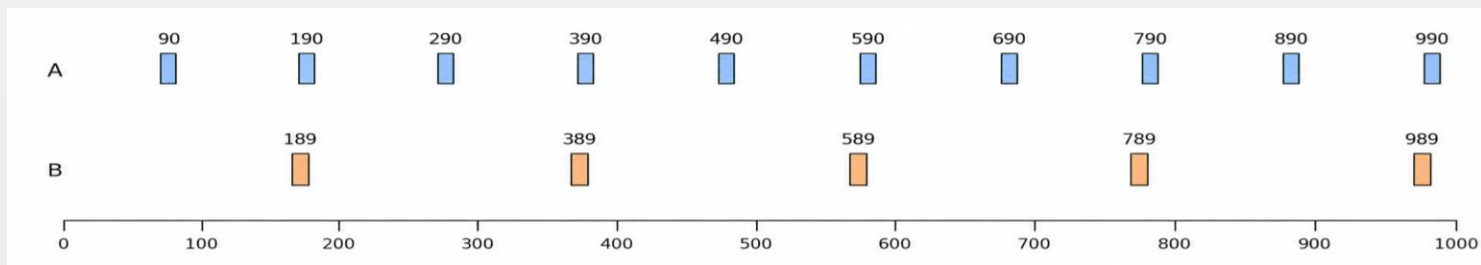
The timeline in the second lab test:



... and in the 99th, and in the 100th.

And in 1st production run. And in the 2nd. And the 300th.

And then, just when we put our newest satellite on the rocket:

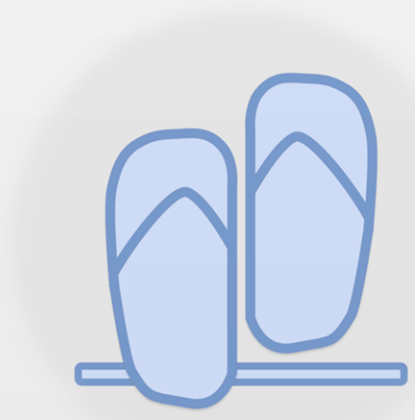


In this case, it does not matter
how long was the system tested.



If the two cycles do not collide **once**, they will **never** will.
And the chances that they **do** is very little, and not predictable.

Testing systems over a long time is not enough.
They should be “shaken” by all means.



October 2 1988

Identifying Storms

Catastrophic storms will expose themselves brutally...

But in most cases the storm may not be easily visible.

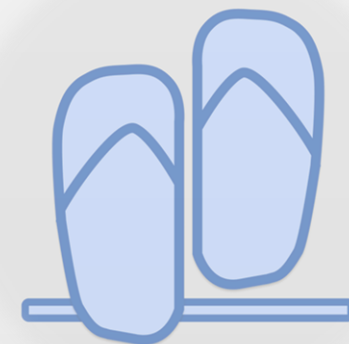
Looking at CPU load alone may not be useful.

The load is averaged over intervals (e.g., 100ms).

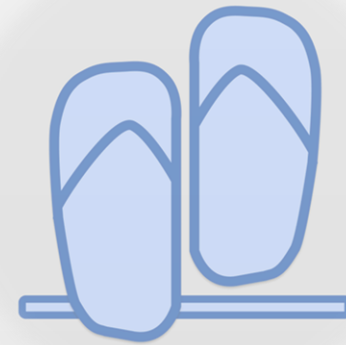
If the storm took 20ms, it is merely up to 20%.

And they might even not consume CPU but wait on contention.

28 days, 6 hours,
42 minutes and 12 seconds



So, What Do We Do?



Design To Avoid Storms [1]

The best way to win a fight is not to have one.
Ask the *periodical cicada*.

- Stagger periodic tasks.

If tasks run in periods of, say, 20ms, 50ms, 100ms – shift their phases:

- 0, 100, 200, 300, ...
- 5, 55, 105, 155, 205, ...
- 10, 30, 50, 70, 90, 110, ...



Design To Avoid Storms [2]

If the tasks are not tied to the same timeline –
For example:

- some are PPS related
- some starts at a specific events
- some use the internal system clock

Try to align them.

E.g., instead of starting the period immediately after some event –
schedule the new periodic task to a known “slot” in the PPS cycle.



Design To Avoid Storms [3]

Avoid period drifting.

For example:

- replace `sleep_for()` by `sleep_until()` with fixed period.

Fixed intervals are much easier to maintain.

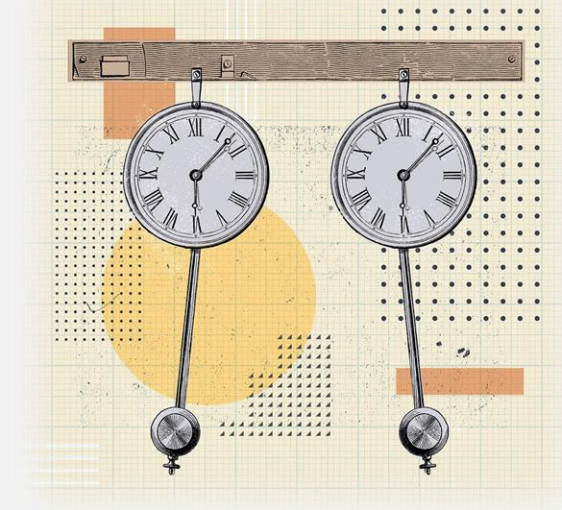
And they make much more stable systems.



Design To Avoid Storms [4]

If the system requirements do not allow that –
E.g, the period **must** be aligned with another cycle

Try to at least define “safe periods”.
For example, align it to the relevant cycle,
but explicitly verify that there is no collision



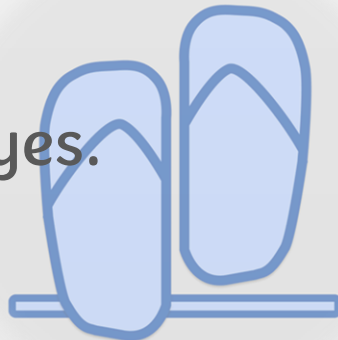
Make Collision Results Predictable

If in case of collision the results are predictable and acceptable – the problem is much less critical.

For example:

- Each task runs on a different core and does not use common resources.
- One of the tasks is of a low priority and will be late due to the other task without consequences.

But remember –
in complex systems, things tend to be more complex than meet the eyes.



Instrument the Timing

Add timing measurements wherever possible, so that it is possible to analyze system runs.

If there was a storm, this will help identifying it post-mortem.
For successful runs, it can help analyzing whether there are “edge” zones that might collide due to little changes.

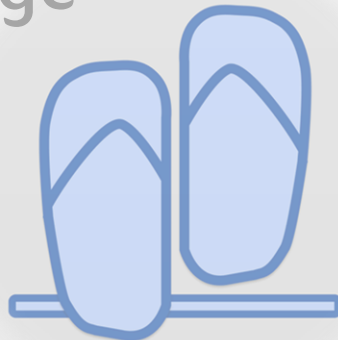


Worst-Case Testing

If requirements or design do not allow absolute preventing of task collisions or storms – make sure that edge-cases of colliding tasks are **deliberately “engineered” and tested.**

For example, use **phase-sweeping**.

This way, it will be possible to estimate the results of the edge cases, and add mitigations wherever required.



Trace Resource Contention

CPU load alone does not say much;
use a full suite of measurements to

- **uncover** the existence of storms
- **understand** their consequences
- **Identify** high-risk timing regions before failure

Typical suspects:

- mutex wait time
- queue depth
- memory/bus/DMA pressure
- ISR latency
- ...





Lessons | Timer Storms

- Know your timer limits: granularity, min. programming time, ...
- Understand and measure the latency and jitter of task execution
- Do not ignore missed deadlines – implement a clear policy
- Beware of **timer storms** in your system!
- Avoid same phase for all periods: **stagger** them in safe distance
- If one event triggers several tasks – consider **various delays**
- Period **drift** may result in a late timer storm
- Two unrelated “timelines” may create a storm **once in many runs**
- **CPU load** will not show a storm; measure **resource contention**
- **Deliberately** test collisions to understand risk and mitigation

Summary

Bad timers are our unfair advantage over AI agents!



The End | Q & A

When a Broken Clock Is Right 23 Times a Day

Mike Lindner



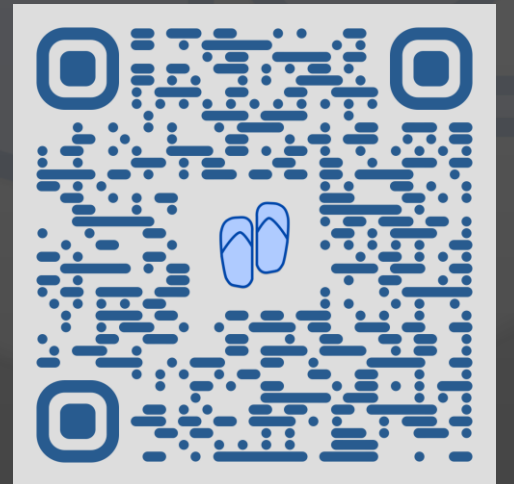
mike.Lindner@gmail.com



<https://linkedin.com/in/mike-lindner>



<https://sw-arch.blog/>
Architect in Slippers | ארכיטקט בכפכפים





Lessons | Summary

| Clock Types

- Use the **right** clock for the right purpose.
- For measuring / scheduling in specific periods – **monotonic** clocks
- There may be several “clocks” in a system, for **different needs**
- “**Real**” clocks use counters; “**adjusting**” clocks fix/align other’s time

| Basic Terminology

- A “clock” allows only to **read** time
- **Scheduling / waiting** are different mechanism, that *use* clocks

| Timer Correctness

- **Quality** of clocks/schedulers includes several aspects

| Arithmetic

- Type-safety design of clocks and timers is essential
- Time elements should adhere to pointer-like arithmetic
- Time units should be clear, safe and work right in expressions
- Different clocks/time types must not mix w/o conversion
- Beware of redundant conversion (“double adjustment”)
- Illegal time representation / behavior must be considered

| `std::chrono`

- The `std::chrono` provides a **standard interface** for clocks

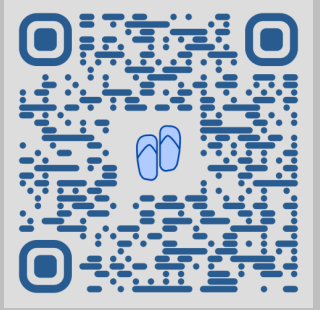
- It provides **type-safety** for time elements math and usage
- It is **not bullet-proof**, and can still result in wrong values
- RT systems might have clock/time needs that **do not fit it**
- Clock/time **type-safety** is crucial for a robust system

| Periodic Timers

- Naïve `sleep()` loops do not keep the period and **drift** over time
- In RT systems, **predictability** is not a luxury; drift may be risky
- Use `sleep_until()` to keep the period fixed
- Know your timer **limits**: granularity, min. programming time, ...
- Understand and measure **the latency and jitter** of task execution
- Do not ignore **missed deadlines** – implement a clear policy

| Timer Storms

- Beware of **timer storms** in your system!
- Avoid same phase for all periods: **stagger** them in safe distance
- If one event triggers several tasks – consider **various delays**
- Period **drift** may result in a late timer storm
- Two unrelated “timelines” may create a storm **once in many runs**
- **CPU load** will not show a storm; measure **resource contention**
- **Deliberately** test collisions to understand risk and mitigation



The End_[2] | Q & A

When a Broken Clock Is Right 23 Times a Day

Mike Lindner



mike.Lindner@gmail.com



<https://linkedin.com/in/mike-lindner>



<https://sw-arch.blog/>
Architect in Slippers | ארכיטקט בכפכפים

