# RK0 Architecture and Services

A white paper on explicit progress and retained-state semantics in a small real-time kernel

**A small kernel is not automatically a simple kernel. RK0 treats recurring real-time coordination patterns as named kernel semantics: task eligibility, temporal release, ownership, availability, waiting, history, invocation, and latest state.**

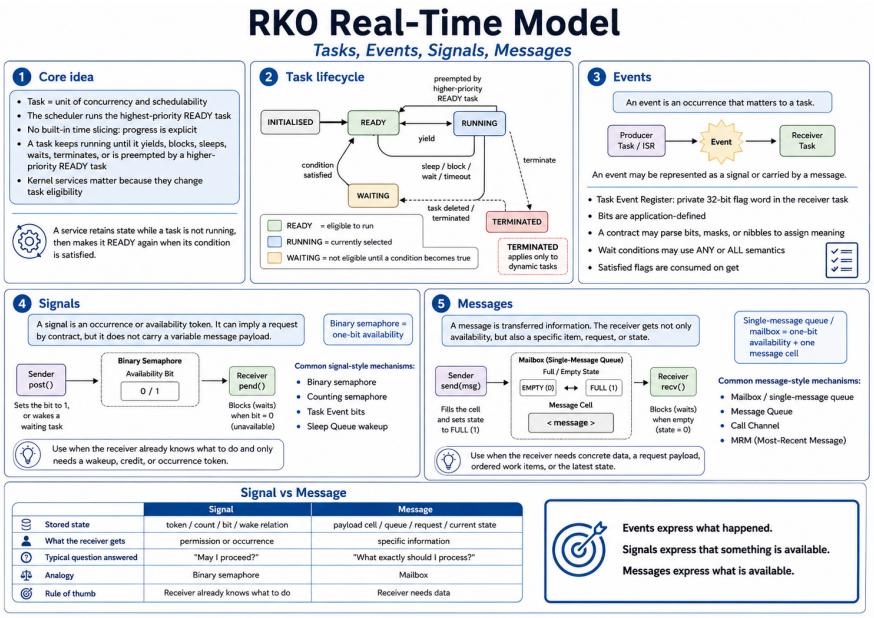


*Figure 1. RK0 real-time model overview: tasks, events, signals, and messages.*

White paper. References are listed at the end of the paper.

# Executive Summary

RK0 is a small deterministic real-time kernel designed around explicit progress. The central claim is that a task should run for a known reason, stop for a known reason, and become eligible again because a service condition has become true.

The design does not treat RTOS services as a catalogue of familiar functions. Each service is read as a retained-state contract: it defines what remains true while a task is not running, who can block, who can wake, what is preserved, what is coalesced, and which real-time mistake the mechanism prevents.

This framing is the main distinction between RK0 and more generic kernel presentations. Mainstream kernels often emphasise feature breadth, compatibility layers, fairness options, and API familiarity. RK0 instead emphasises semantic separation: availability is not ownership, a wakeup is not payload transfer, ordered history is not latest state, and synchronous invocation is not merely a queued message.

The result is not a claim that familiar kernels are wrong. The claim is narrower: for known embedded applications with tight timing constraints, some coordination patterns are common enough and subtle enough that they deserve explicit kernel-level semantics.

## Key claims

1. Execution progress should be explicit.
2. Services should be understood by what they retain and discard.
3. Fairness is not a hidden scheduler promise; it is expressed through priorities, yielding, blocking, and waiting.
4. Signals and messages differ by retained state: token versus transferred information.
5. The kernel should encode recurring real-time coordination semantics without becoming a broad platform.

## Scope and non-goals

This paper is not an API manual. It does not enumerate flags, parameter values, or call signatures. It explains the architecture and service model: what RK0 is trying to make explicit, why the services exist, and how the design differs from mainstream kernel expectations.

# 1. The Problem: Complexity Without Coordination Semantics

Embedded software is increasingly software-defined, connected, updated, layered, and security-sensitive. At the same time, many embedded targets remain small, closed, and timing-sensitive. They are built around known devices, known control loops, known timing budgets, and known failure modes.

This creates an architectural tension. More software abstraction is needed for portability, maintainability, and lifecycle management. Real-time correctness still depends on concrete execution properties: bounded latency, bounded allocation, clear ownership, clear task eligibility, and analysable communication semantics.

RK0 addresses the semantic erosion that can occur when every coordination pattern is reconstructed from a few overloaded primitives. A semaphore becomes a wakeup, a lock, a request, or a credit depending on local convention. A queue becomes a command channel, a state

publisher, a service call, or a resource manager. A timer becomes a delay, a periodic release, or a deadline budget. Those protocols may work, but the meaning is hidden in application code.

> **RK0 is a real-time coordination project. It is not primarily a hardware abstraction project, a POSIX compatibility project, or a platform ecosystem project.**

# 2. Design Premise: Low Complexity Takes Work

Low complexity is not the same as low effort. A small kernel can be incomplete. A simple kernel must have fewer hidden behaviours, fewer accidental policies, fewer overloaded mechanisms, and fewer ways for an application to express a real-time requirement indirectly.

RK0 is designed around the idea that recurring real-time coordination needs should become kernel concerns when they are domain-independent, common, and difficult to implement correctly without scheduler or kernel-object participation.

This leads to a selective form of generality. RK0 provides generic mechanisms where generality is useful. It also provides services whose semantics directly encode common real-time relations: urgency, precedence, exclusion, availability, notification, state transfer, invocation, and latest-state publication.

The design deliberately avoids pursuing a standard API surface such as POSIX or CMSIS-RTOS as the primary objective. API familiarity is valuable, but a standard vocabulary can freeze the interface before the underlying real-time semantics are made explicit.

# 3. The Real-Time Model: Progress for a Known Reason

A task is the fundamental unit of concurrency and schedulability. In RK0, tasks follow the thread model and run inside a single firmware image. Static tasks use the states INITIALISED, READY, RUNNING, and WAITING. Dynamic tasks add TERMINATED.

The central invariant is simple: a task must be READY before it can be selected to run. READY does not mean running; it means eligible. RUNNING means currently selected. WAITING means not eligible until a condition becomes true.

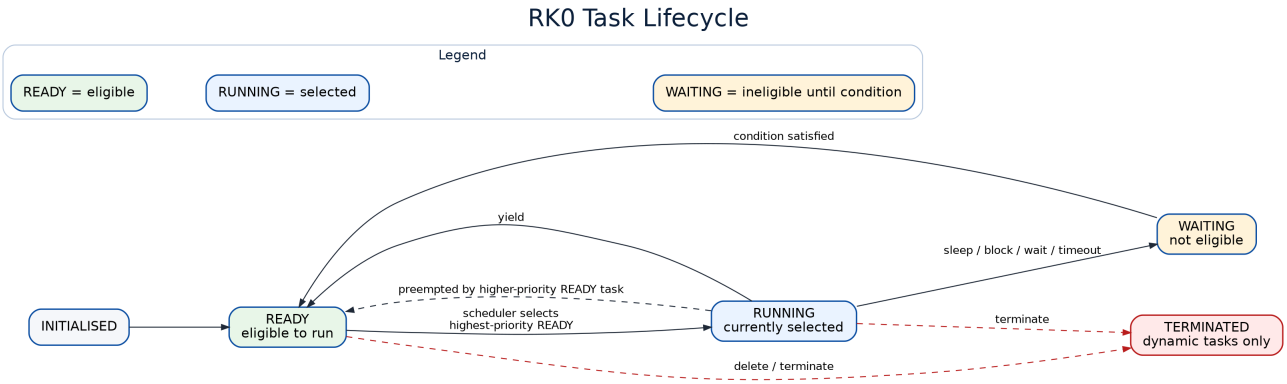


*Figure 2. Task lifecycle, including TERMINATED as a dynamic-task-only state.*

RK0 uses a priority-based preemptive scheduler, but has no built-in time slice. A task runs until it yields, blocks, waits, sleeps, terminates, or is preempted by a higher-priority READY task. Tasks with the same priority cooperate by yielding or waiting. If a dispatched task never yields or waits

and no higher-priority task becomes READY, the scheduler keeps it running because no reason to progress differently was expressed.

> **Execution progress is expressed in application code. The scheduler should not invent progress; the application should express it, and the kernel should preserve it predictably.**

# 4. Architecture: Executive Policy and Scheduler Mechanism

RK0 can be read as two cooperating layers. The Executive manages kernel objects and service semantics. The low-level scheduler acts as a software extension of the CPU: it selects, switches, blocks, and readies tasks.

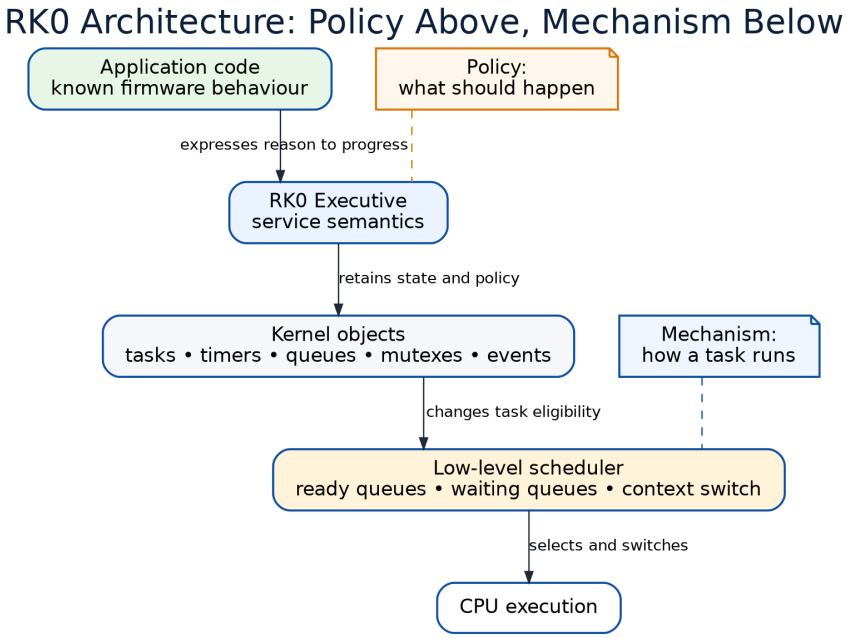


*Figure 3. Architecture layers: application intent becomes service semantics, then scheduler-visible task state.*

This split explains why services are not mere wrappers around scheduler operations. A service is the point where an application-level reason becomes a scheduler-visible condition. The application expresses a blocked condition. The service retains the state needed to represent that condition. The scheduler later observes that the task is READY again.

```
Application-level statement:
    This task cannot proceed until a condition becomes true.

Service-level statement:
    This object retains the state needed to represent that condition.

Scheduler-level statement:
    The task is no longer WAITING; it is READY again.
```

# 5. Events, Signals, and Messages

Real-time systems are reactive. They respond to changes in the environment: a timer tick, a sensor sample, an interrupt, a state transition, a command, a deadline, or a control phase. An event is the logical representation of such a change.

Given two observation instants, if the observed state differs, at least one event must have occurred. If the observed state is the same, it is still not possible to conclude that no event occurred; a value may have changed twice and returned to its original value. Computation samples reality, and real-time software must react while the result is still useful.

In RK0, an event can be represented as a signal or carried by a message. The distinction is not decorative; it determines what the kernel object retains.

## Signals: contract over tokens

A signal signifies occurrence or availability. It can imply a request by application contract, but it does not carry a variable message payload. A task may define an event bit as "sample the ADC", a semaphore as "run the control step", or a nibble in an event word as a compact command class. Those are valid contracts. The kernel still retains only a token, count, bit, or waiting relation.

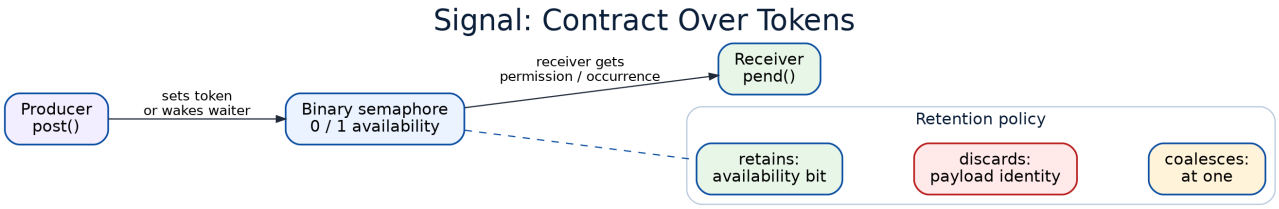


*Figure 4. A binary semaphore is one-bit availability: it can wake a receiver, but it does not carry a payload.*

## Messages: contract over transferred information

A message carries information across a task boundary. The receiver does not only learn that something is available; it receives a specific item, request, response relation, or state snapshot.

The boundary is most clearly explained by comparing a binary semaphore with a mailbox. A binary semaphore is one-bit availability. A single-message queue is one-bit availability plus one message cell.

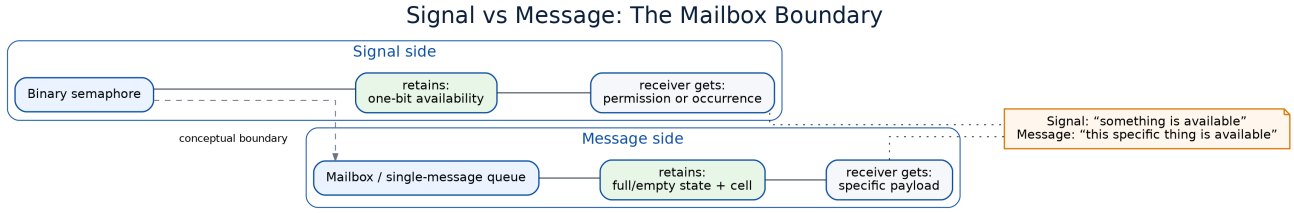


*Figure 5. Signal versus message: the mailbox adds a retained message cell to the availability state.*

```
Signal:
    something is available.

Message:
    this specific thing is available.
```

This distinction prevents a common real-time mistake: confusing "wake a task" with "transfer work". A wakeup may be sufficient when the receiver already knows what to do. A message is needed when the receiver must process concrete data, a request payload, ordered work, or a latest state.

# 6. Services as Retained-State Contracts

Once the real-time model is clear, RK0 services can be read as retained-state contracts rather than as an API catalogue. Each service answers the same questions: what does it retain, what does it discard, who may block, who may wake, what is coalesced, what is preserved, and what real-time mistake is prevented.


*Figure 6. Service retention map: each service preserves a different real-time relation.*

| Service | Retains | Discards | Real-time mistake prevented |
|---|---|---|---|
| Scheduler / ready queues | Eligibility and priority order. | Implicit fairness and hidden round-robin progress. | Assuming that time passing implies progress. |
| Timers | Relative delay, phase relation, local periodic reference, or timeout budget. | The assumption that all delays mean the same thing. | Using a relative delay where the real requirement is periodic phase or bounded waiting. |
| Partition memory | Fixed-size bounded allocation pool. | General heap behaviour and unbounded fragmentation. | Pretending that dynamic allocation is free because it is convenient. |
| Semaphore | Availability count and blocked waiters. | Payload, ownership, and identity beyond the retained count. | Building notifications from ad hoc polling or shared variables. |
| Task Event | Private consumable bits in the receiver task. | Payload and public shared-variable semantics. | Using shared flags without a clear consumption rule. |
| Sleep Queue | Waiters and their priority-ordered waiting relation. | Occurrence history and predicate truth. | Believing that every wakeup primitive must latch occurrences. |
| Mutex | Ownership and priority dependency. | Anonymous availability. | Treating a lock as a binary semaphore and hiding priority inversion. |
| Message Queue | Ordered message history up to queue capacity. | Latest-only semantics and infinite producer-consumer mismatch. | Losing work items or hiding payloads in fragile side channels. |
| Port | Owned queue endpoint and sender-owner priority relation. | N:N receive ambiguity. | Building a resource-manager task without making ownership visible to the scheduler. |

| Service | Retains | Discards | Real-time mistake prevented |
|---|---|---|---|
| Call Channel | Synchronous invocation, response path, and client-server priority coupling. | Anonymous message backlog and fire-and-forget interpretation. | Modelling procedure calls as ordinary queued messages. |
| MRM | Newest valid state and reader-safe buffer lifetime. | Historical backlog for new readers. | Forcing control loops to drain stale history before seeing current state. |

## Sleep Queue example: retaining waiters, not events

The Sleep Queue illustrates the retained-state method particularly well. It retains waiters. It does not retain occurrences. A signal sent to an empty Sleep Queue is lost. That behaviour is not a defect when the primitive is understood correctly: the queue says "from now on, wait until signalled", not "remember that an event happened".

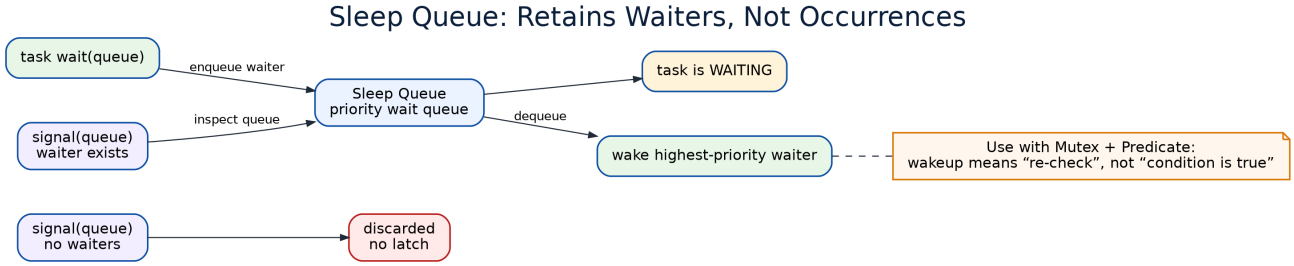


*Figure 7. Sleep Queue mechanism: the object retains waiters, not event history.*

Composed with a mutex and a protected predicate, the same primitive supports monitor-like coordination. The mutex protects the state, the predicate defines whether progress is valid, and the Sleep Queue parks tasks until some producer says that the predicate should be re-checked.

# 7. Where RK0 Drifts from Mainstream Kernels

RK0 does not drift from mainstream kernels by inventing unfamiliar concepts. It uses familiar terms: task, semaphore, mutex, queue, timer, event, and message. The drift is in the refusal to blur their meanings.

General-purpose kernels and broad RTOS platforms often optimise for feature breadth, compatibility, isolation, and configurable scheduling policy. For example, Zephyr documents configurable scheduling behaviour including time slicing, while Linux CFS is explicitly designed around fair CPU-time distribution. RK0 chooses a smaller target: deterministic coordination for known embedded applications.

| Mainstream framing | RK0 framing |
|---|---|
| Kernel as a collection of RTOS services. | Kernel as a set of schedulability contracts. |
| Fairness may be provided by scheduling policy. | Progress is explicit in application code. |
| Queues, semaphores, and flags are broad tools. | Retention policy determines the service. |
| Dynamic abstraction is often convenient. | Bounded behaviour is the first constraint. |
| API familiarity is valuable. | Semantic clarity is more valuable than surface compatibility. |

This is not an argument against generic mechanisms. RK0 still provides generic mechanisms where they are useful. The point is that some recurring real-time patterns are common enough, and their worst cases are subtle enough, that they deserve kernel-level semantics rather than informal application conventions.

# 8. Profiling, Release Claims, and the Cost of Simplicity

A credible real-time kernel claim depends on reproducible measurement. Fast paths are easy to advertise; worst cases require deliberate scenarios. Meaningful profiling asks which path was forced, which objects were contended, which priority relation existed, which timeout or queue state applied, and what upper bound follows from that scenario.

The RK0 project therefore treats release claims as dependent not only on implementation maturity, but also on reproducible debugging, tracing, profiling, and test-harness methods. A small implementation is not sufficient. The reasoning surface must also be explicit and externally repeatable.

> **A simple kernel is not one that avoids hard cases. It is one whose hard cases are named, bounded, and testable.**

# 9. Conclusion: Progress for a Known Reason

RK0 follows one rule: execution progress has a reason that the programmer can express once the service semantics are understood.

A task becomes eligible because a condition became true. A timer expired. A resource was released. A semaphore count became available. An event bit satisfied a mask. A queue received data. A channel completed a call. A new state was published. A protected predicate should be re-checked. Every service exists to make one of those reasons concrete.

```
Tasks express schedulability.
Events express what happened.
Signals express that something is available.
Messages express what is available.
Timers express when eligibility may resume.
Memory partitions express bounded allocation.
Mutexes express ownership.
Queues express preserved history.
Channels express invocation.
MRM expresses latest state.
```

RK0 is small because it rejects unnecessary machinery. RK0 is demanding because the machinery it keeps must carry clear real-time meaning.

# References


[1] RK0 Docbook, "RK0: The Real-Time Kernel 0". https://antoniogiacomelli.github.io/RK0/

[2] RK0 Wiki, "Service Semantics". https://github.com/antoniogiacomelli/RK0/wiki/Service-Semantics

[3] Zephyr Project Documentation, "Scheduling". https://docs.zephyrproject.org/latest/kernel/services/scheduling/index.html

[4] Linux Kernel Documentation, "CFS Scheduler". https://docs.kernel.org/scheduler/sched-design-CFS.html

[5] IAR, "Software-defined everything in embedded systems: Staying in control as complexity grows". https://www.iar.com/blog/software-defined-everything-in-embedded-systems-staying-in-control-as-complexity-grows