

RK0 User Manual

Real-Time Kernel '0'

Antonio Giacomelli

Draft for RK0 0.60.0 - 9 August 2026

Contents

Section	Topic
1	About this manual
2	Run RK0 on QEMU
3	A minimal RK0 application
4	Tasks and execution progress
5	Waiting, timeouts and execution context
6	Time services
7	Signals and waiting
8	Shared-state coordination
9	Buffered message passing
10	Synchronous message passing
11	Latest-state publication
12	Deterministic memory
13	Dynamic kernel objects
14	Dynamic tasks
15	Trace and fault handling
16	Choosing a communication service
17	Compact API index
18	Further reading

1 About this manual

This manual explains how to use RK0 services and what an application observes when it uses them. It is deliberately shorter and more operational than the RK0 DocBook.

For each service, the manual answers three questions:

1. What relationship does the service express?
2. What does a successful call mean to the caller?
3. When should the service be selected instead of a nearby alternative?

The public prototypes and exhaustive return values remain authoritative in `core/inc/kapi.h`. The [RK0 DocBook](#) explains design rationale and implementation. The [Service Map](#) compares the contracts.

This draft describes RK0 0.60.0 at repository commit `0f5c6b4`. In this version:

- a Mailbox is a one-slot Message Queue;
- a Port is a Message Queue bound to one owner task;
- a Channel endpoint belongs to its server task; there is no `RK_CHANNEL` object;
- a Rendezvous endpoint belongs to its receiver task; there is no `RK_RENDEZVOUS` object;
- selected non-task kernel objects may be created and destroyed at runtime from bounded kernel-owned pools.

1.1 Notation

RK0 priorities run in the direction used by many fixed-priority kernels:

Priority 0 is the highest user priority. A smaller number means a more urgent task.

The figures use the following vocabulary:

- **RUNNING** - the task currently owns the processor;
- **READY** - the task may be selected by the scheduler;
- **WAITING** - the task cannot run until a time or coordination condition is satisfied;
- a solid arrow - an operation or state transition;
- a dashed arrow - a notification or return path that does not carry stored history.

1.2 Coding Style

RK0 coding style might look odd for those with a PC Linux background, but indeed it has a very traditional embedded systems programming style.

- **Kernel Objects:** Kernel objects are any data structure provided by the kernel along with a service (Queues, Semaphores, etc.). Kernel objects are on the form `RK_OBJECT`. Eg.: `RK_SEMAPHORE`, `RK_MSG_QUEUE`.
- **Object Handles:** In RK0, a handle is an object whose type is a pointer to another object. For example, `RK_TASK_HANDLE` is a pointer to a **Task Control Block** structure.
- **Declaration macros and function macros:** some macros, mainly those used to declare and/or initialise some objects, will not use a trailing ``;` or even a `()`.

```

#include <kapi.h>

#define WORKER_STACK_WORDS 160U
#define WORKER_PRIORITY 2U
typedef struct
{
    ULONG mesgSize;
    CHAR *stringPtr;
}MyMesg_t;

#define N_MESG 15U

    /* declare task objects: handle name, function, buffer name, size in words */
RK_DECLARE_TASK(workerHandle, WorkerTask,
    workerStack, WORKER_STACK_WORDS)

    /* declare Message Queue: Object name: mesgqA; Memory buffer: qBuf; Type of Message MyMesg_t
that has 2 words. Number of Messages: 15 */
RK_DECLARE_MESG_QUEUE(mesgqA, qBuf, MyMesg_t, N_MESG)

int main(void)
{

    kCoreInit(); /* init core (sysclk, peripherals, interrupts, etc.) */
    kInit(); /* init kernel */

    while (1)
    {
        kErrorHandler(RK_FAULT_APP_CRASH);
    }
}

VOID kApplicationInit(VOID)
{

    RK_INIT_OBJ_PARTITIONS /* Required if using dynamic objects constructors/destructors */

    RK_ERR err = kTaskInit(&workerHandle,
        WorkerTask,
        RK_NO_ARGS,
        "Worker",
        workerStack,
        WORKER_STACK_WORDS,
        WORKER_PRIORITY,
        RK_PREEMPT);
    K_ASSERT(err == RK_ERR_SUCCESS);
    err = kMesgQueueInit(&mesgqA, qBuf, RK_TYPE_WORD_COUNT(MyMesg_t), N_MESG);
    K_ASSERT(err == RK_ERR_SUCCESS);

}

VOID WorkerTask(VOID *args)
{
    RK_UNUSEARGS

    while (1)
    {
        DoOneCycle();
        kSleepPeriodic(RK_MS_TO_TICKS(20U));
    }
}

```

1.2 Return values

RK0 separates successful operations, well-defined unsuccessful outcomes and invalid use:

Value	Meaning	Examples
0	Operation completed	RK_ERR_SUCCESS
positive	Valid operation, condition not met	queue full, timeout, no waiter
negative	Invalid object, parameter, state or context	null object, wrong owner, ISR misuse

Do not treat every non-zero result as a kernel fault. A non-blocking receive from an empty queue is a normal observation. In contrast, unlocking another task's mutex is invalid use.

API preconditions remain the application's responsibility when `RK_CONF_ERR_CHECK` is disabled.

2 Run RK0 on QEMU

RK0 currently builds with the GNU Arm Embedded toolchain and runs the repository demonstration on QEMU.

Required commands:

```
git clone https://github.com/antoniogiacomelli/RK0.git
cd RK0
make ARCH=armv7m qemu
```

Use `ARCH=armv6m` for the Cortex-M0 QEMU target. The default application is `app/src/application.c`. It contains selectable examples intended as executable documentation.

The principal build outputs are placed below `build/<architecture>/`:

- `rk0_demo.elf` - executable and debugger image;
- `rk0_demo.bin` - raw binary;
- `rk0_demo.hex` - Intel HEX image;
- `rk0_demo.map` - linker map.

For a debug server instead of immediate execution:

```
make ARCH=armv7m qemu-debug
```

QEMU then waits for GDB on TCP port 1234.

3 A minimal RK0 application

An RK0 application provides ordinary platform initialisation, static kernel storage and a mandatory `kApplicationInit()` function.

```
#include <kapi.h>

#define WORKER_STACK_WORDS 160U
#define WORKER_PRIORITY 2U

RK_DECLARE_TASK(workerHandle, WorkerTask,
                workerStack, WORKER_STACK_WORDS)

int main(void)
{
    kCoreInit();
    kInit();

    /* kInit() does not return during normal operation. */
    while (1)
    {
        kErrorHandler(RK_FAULT_APP_CRASH);
    }
}

VOID kApplicationInit(VOID)
{
    /* Required if using dynamic objects constructors/destructors */
    RK_INIT_OBJ_PARTITIONS

    RK_ERR err = kCreateTask(&workerHandle,
                            WorkerTask,
                            RK_NO_ARGS,
                            "Worker",
                            workerStack,
                            WORKER_STACK_WORDS,
                            WORKER_PRIORITY,
                            RK_PREEMPT);
    K_ASSERT(err == RK_ERR_SUCCESS);
}

VOID WorkerTask(VOID *args)
{
    RK_UNUSEARGS

    while (1)
    {
        DoOneCycle();
        kSleepPeriodic(RK_MS_TO_TICKS(20U));
    }
}
```

`RK_DECLARE_TASK` declares the entry prototype, an 8-byte-aligned stack and a task handle. Stack size is expressed in 32-bit words, not bytes. The handle and stack must remain valid for the application lifetime; file-scope storage is normal.

`kInit()` calls `kApplicationInit()`, completes kernel initialisation, makes the created tasks ready, enables interrupts and starts the highest-priority task. Create startup tasks and initialise the objects they need inside `kApplicationInit()`.

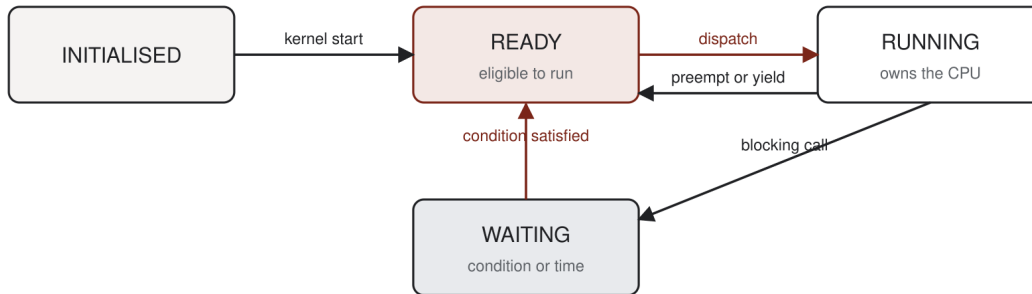
`RK_INIT_OBJ_PARTITIONS` initialises the fixed pools used by runtime-created kernel objects. It is a no-op when `RK_CONF_DYNAMIC_OBJECTS` is disabled, so it may remain in common startup code. Call it once before any dynamic-object **Create** operation. It does not create an object.

4 Tasks and execution progress

4.1 Task states

The scheduler is the only component that changes a task from READY to RUNNING.

Only the scheduler changes READY to RUNNING



RK0 task lifecycle

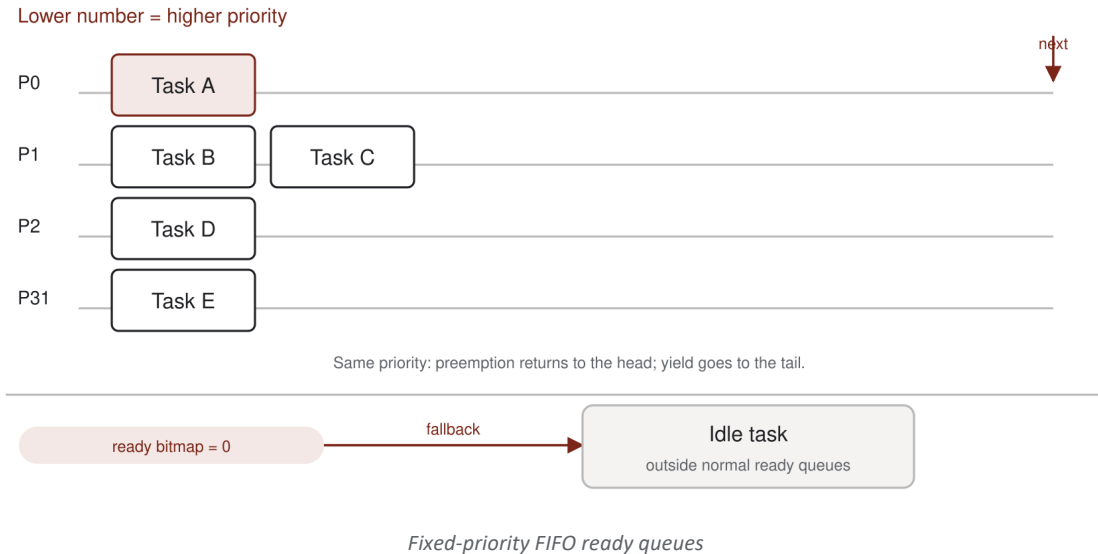
A static task normally moves among INITIALIZED, READY, RUNNING and a family of WAITING states. A dynamic task may additionally become TERMINATED.

- A task becomes **READY** when creation completes after scheduler start, or when its waiting condition is satisfied.
- The scheduler changes one READY task to **RUNNING**.
- A higher-priority ready task may preempt a preemptible running task.
- `kYield()` voluntarily returns the caller to READY.
- A blocking service changes the caller to a service-specific WAITING state.

Returning from a task entry function is not a task-termination operation. Task functions should normally contain a loop. Dynamic tasks terminate through the dynamic-task API described later.

4.2 Fixed-priority FIFO scheduling

RK0 maintains one FIFO ready queue for each normal priority from 0 to 31 and does not perform automatic time slicing. A 32-bit bitmap records which of those queues are non-empty.



When the bitmap is non-zero, its least-significant set bit identifies the highest-priority non-empty ready queue. Within the same priority:

- initial order follows task creation order;
- a preempted task returns to the head, so it resumes before equal-priority peers;
- a yielded task goes to the tail;
- a task made ready after waiting goes to the tail.

The Idle task is not on the normal ready queues and is outside their bitmap. It is dispatched as the fallback when the ready bitmap is zero; it is not the priority-31 task and it does not compete with application work.

Two equal-priority tasks therefore cooperate by yielding or waiting. If the running task does neither, there is no expressed reason for another equal-priority task to run.

`RK_NO_PREEMPT` creates an exceptional non-preemptible task. Once dispatched, it continues until it blocks, yields or otherwise leaves RUNNING. Use this only for short, bounded service routines: it can delay every user task, including more urgent ones.

4.3 Yielding is not sleeping

`kYield()` says that the task has completed its present turn among ready peers. It does not express a time condition and does not make the task WAITING.

If no equal- or higher-priority task is ready, yielding does not manufacture work for a lower-priority task. The caller remains the correct task to run.

4.4 Scheduler lock

`kSchLock()` and `kSchUnlock()` (also exposed as `kPreemptDisable()` and `kPreemptEnable()`) defer user-task preemption. Locks are nested and belong to the task.

The scheduler lock does **not** disable interrupts and is not a mutex. It does not establish shared-resource ownership or priority inheritance. Use it only where delaying dispatch is itself the required, bounded operation.

5 Waiting, timeouts and execution context

Most services that may wait accept one of three timeout forms:

Timeout	Caller behaviour
<code>RK_NO_WAIT</code>	Inspect or attempt the operation and return immediately
finite ticks	Wait up to the requested kernel time
<code>RK_WAIT_FOREVER</code>	Wait until the service condition is satisfied

Finite values must not exceed `RK_MAX_PERIOD`. Convert deliberately between milliseconds and ticks:

```
RK_TICK ticks = RK_MS_TO_TICKS(25U);
```

The conversion truncates. A duration shorter than one kernel tick becomes zero and therefore changes the operation into `RK_NO_WAIT` for timeout-bearing APIs.

5.1 What a timeout guarantees

A timeout removes a task's dependency on the object and makes it READY. Actual return to application code still depends on fixed-priority scheduling. A newly ready task cannot execute while a more urgent task legitimately remains RUNNING.

The operation's return code tells the caller which completion occurred. After `RK_ERR_TIMEOUT`, application code must assume that the requested operation did not complete unless that service explicitly defines an abandoned in-progress state, as Channel does.

5.2 Interrupt service routines

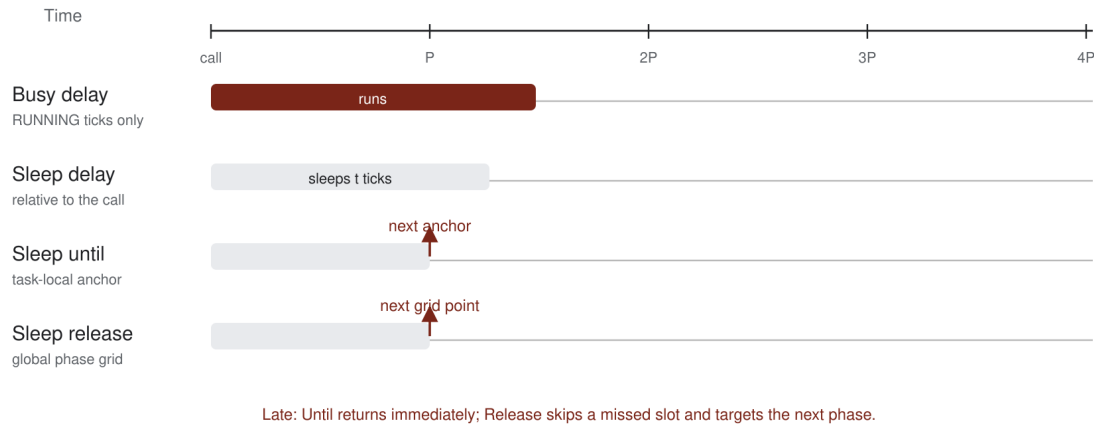
An ISR must never request an operation that can block. Some services provide a non-blocking ISR path, such as `kEventSet()`, `kSemaphorePost()` and no-wait Message Queue operations. Multi-task wake operations may be deferred to the PostProc system task.

Task-backed endpoints - Port receive, Channel and Rendezvous - should be used from task context. Their meaning depends on the current task, not only on whether the implementation can reject a particular call.

When ISR use matters, check the allowed context in `kapi.h` for the exact operation. Do not infer ISR safety merely from the absence of a timeout parameter.

6 Time services

RK0 keeps different meanings of time in separate calls.



Comparison of RK0 time services

6.1 Choose the intended time relation

Service	Meaning	Best use
<code>kBusyDelay(t)</code>	Consume t ticks while RUNNING	Simulated or deliberately active workload
<code>kSleepDelay(t)</code>	Sleep relative to this call	One-off delay or back-off
<code>kSleepUntil(&anchor, p)</code>	Advance one task-local deadline by p	Periodic work that must account for every activation
<code>kSleepRelease(p)</code>	Target the next scheduler-start phase slot	Phase-aligned periodic work that may skip missed releases

6.2 Busy Delay

`kBusyDelay(t)` leaves the caller RUNNING. Only ticks observed while the caller executes count towards completion. Time spent preempted does not reduce the remaining busy delay.

Use it to represent CPU work in an example or for a genuinely active wait. It does not free the processor for a lower-priority task.

6.3 Sleep Delay

`kSleepDelay(t)`, also named `kSleep(t)`, suspends the task for a relative duration measured from the call.

```
DoWork();
kSleep(RK_MS_TO_TICKS(50U));
```

In a loop, work time, preemption and release jitter accumulate. Do not use relative sleep to define a periodic task.

6.4 Sleep Until

`kSleepUntil()` uses a reference owned by the task:

```
RK_TICK anchor = kTickGet();

while (1)
{
    AcquireAndProcessSample();

    RK_ERR err = kSleepUntil(&anchor,
                           RK_MS_TO_TICKS(10U));
    if (err == RK_ERR_ELAPSED_PERIOD)
    {
        RecordLateActivation();
    }
}
```

Each call advances the anchor by one period. If the deadline has already elapsed, the call returns immediately with `RK_ERR_ELAPSED_PERIOD`. It does not skip forward over multiple releases. This favours execution count over absolute phase recovery.

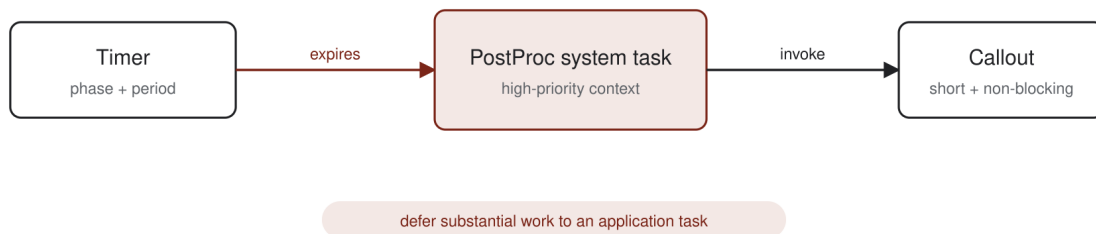
6.5 Sleep Release

`kSleepRelease()`, also named `kSleepPeriodic()`, uses the phase grid established when scheduling begins. A late task sleeps less to reach the next phase. If an entire release has been missed, RK0 records an overrun, skips the missed slot and targets the next valid one.

Use it for known startup tasks that should remain aligned to a system phase. Do not use the global startup phase of `kSleepRelease()` for a task spawned later at an arbitrary time; use a task-local `kSleepUntil()` anchor instead.

6.6 Application timer callouts

An application timer defers a callback from tick expiry to the PostProc system task.



Application timer callout context

`kTimerInit()` arms a one-shot or reloading timer with an initial phase, a period, a callback and an argument pointer. `kTimerCancel()` cancels an active timer. When dynamic objects are enabled, `kTimerCreate()` allocates and arms a timer, and `kTimerDestroy()` cancels and releases it.

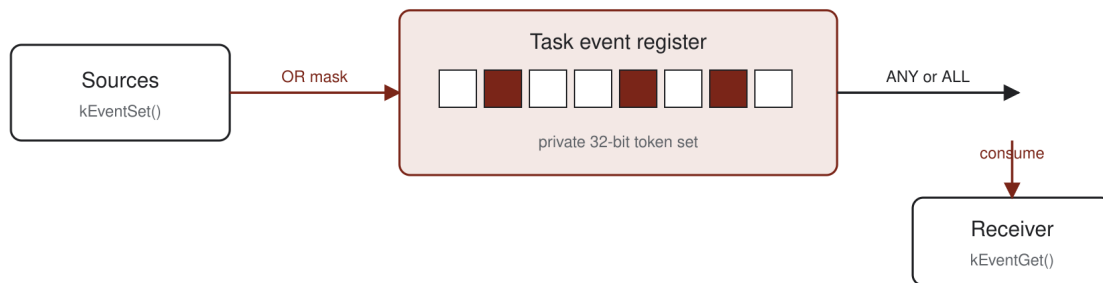
Callouts execute in high-priority, non-preemptible system-task context. They must be short and non-blocking. A normal pattern is to set a task event or post a semaphore and let an application task perform the work.

7 Signals and waiting

Signals express occurrence or availability. They do not carry an application payload.

7.1 Task Event Register

Every task has a private 32-bit Event Register. Other tasks or an ISR set bits; only the receiving task waits for and consumes them.



Repeated sets of the same bit collapse into one pending token.

Task Event Register

`kEventSet(task, mask)` ORs the supplied mask into the current register. Setting the same bit repeatedly before it is consumed still leaves one bit set. If each occurrence matters, use a counting Semaphore or a Message Queue.

The receiver selects `RK_EVENT_ANY` or `RK_EVENT_ALL`:

```
#define EVT_SAMPLE RK_EVENT_1
#define EVT_SETPOINT RK_EVENT_2

RK_EVENT_FLAG got = 0UL;

RK_ERR err = kEventGet(EVT_SAMPLE | EVT_SETPOINT,
    RK_EVENT_ALL,
    &got,
    RK_WAIT_FOREVER);
K_ASSERT(err == RK_ERR_SUCCESS);
```

On success, RK0 optionally copies the complete register state to `got`, then clears the requested mask. Bits outside that mask remain pending.

Use Task Events when:

- the receiver is known;
- no payload is needed;
- bitwise ANY or ALL composition is useful;
- repeated occurrences may coalesce.

`kEventQuery()` observes the current register. `kEventClear()` explicitly clears selected bits. Passing `NULL` as their task handle selects the calling task; an ISR must name a target task explicitly.

7.2 Semaphores

A Semaphore stores an unsigned count of available units or pending occurrences. It has no owner.

```
static RK_SEMAPHORE samplesReady;

VOID kApplicationInit(VOID)
{
    RK_ERR err = kSemaphoreInit(&samplesReady,
                                0U, /* initially no sample */
                                8U); /* retain up to 8 */
    K_ASSERT(err == RK_ERR_SUCCESS);
}
```

`kSemaphorePend()` consumes a stored count. If the count is zero, it returns immediately for `RK_NO_WAIT` or places the task in the priority-ordered wait queue.

`kSemaphorePost()` first satisfies the most urgent waiting task. With no waiter, it increments the count up to the configured maximum. Posting at the maximum leaves the value unchanged and returns `RK_ERR_SEMA_FULL`.

Convenience initialisers are available:

```
kSemaBinInit(&doorbell, 0U); /* maximum 1 */
kSemaCountInit(&credits, 12U); /* maximum UINT */
```

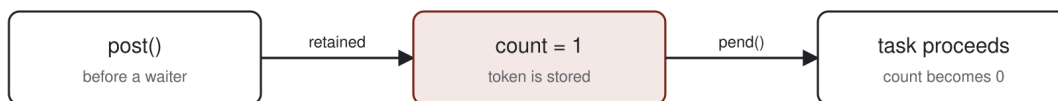
A binary Semaphore can guard a simple region only by application convention. It is not a Mutex: any task may post it, there is no owner and priority inversion is not handled.

Dynamic use is `RK_SEMAPHORE_HANDLE` with `kSemaphoreCreate()` and `kSemaphoreDestroy()`. Destroy requires no waiters; a stored count is discarded.

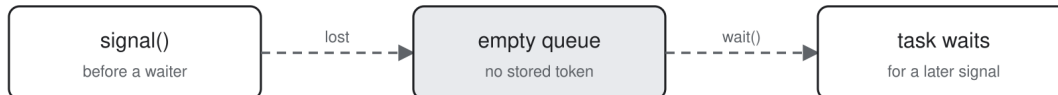
7.3 Sleep Queues

A Sleep Queue is a pure, non-latching waiting relation. It stores waiting tasks, not a condition and not an occurrence token.

Semaphore



Sleep Queue



Semaphore token retention compared with a Sleep Queue

`kSleepQueueWait()` waits for a future signal. `kSleepQueueSignal()` wakes the highest-priority waiter; `kSleepQueueWake(q, n, left)` wakes up to `n` (0 means all). `kSleepQueueReady()` wakes a named task from that queue, while `kSleepQueueBlockReadyTask()` moves a non-running READY task into it.

8 Shared-state coordination

8.1 Mutex Locks

A Mutex expresses exclusive ownership of shared state. Once locked, only its owner may unlock it. RK0 Mutexes are non-recursive: attempting to lock the same Mutex twice is invalid.

```
static RK_MUTEX stateLock;

VOID kApplicationInit(VOID)
{
    RK_ERR err = kMutexInit(&stateLock,
                           RK_PRIO_INHERITANCE);
    K_ASSERT(err == RK_ERR_SUCCESS);
}

VOID UpdateState(VOID)
{
    RK_ERR err = kMutexLock(&stateLock, RK_WAIT_FOREVER);
    K_ASSERT(err == RK_ERR_SUCCESS);

    sharedState.value++;

    err = kMutexUnlock(&stateLock);
    K_ASSERT(err == RK_ERR_SUCCESS);
}
```

The dynamic form uses `RK_MUTEX_HANDLE`, `kMutexCreate()` and `kMutexDestroy()`. The Mutex must be unlocked, unowned and have no waiters before it can be destroyed.

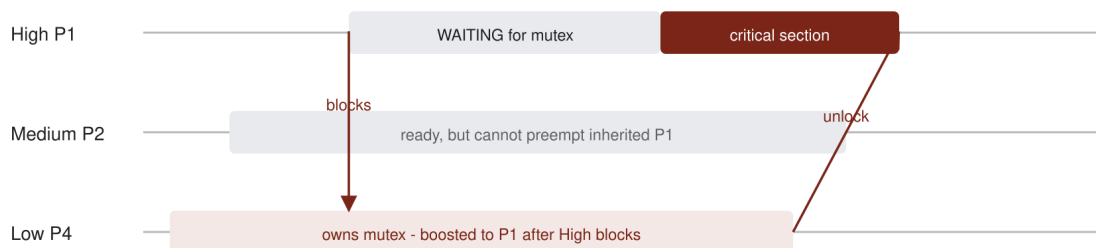
The protocol is selected at initialisation:

- `RK_PRIO_NONE` - mutual exclusion only;
- `RK_PRIO_INHERITANCE` - transitive priority inheritance.

8.1.1 Priority inheritance

When a high-priority task waits for a Mutex owned by a lower-priority task, the owner inherits the waiter's effective priority. This prevents unrelated medium-priority work from extending the inversion.

Priority inheritance keeps the dependency moving



Transitive priority inheritance on a Mutex

RK0 recomputes a task's effective priority across all PI-enabled Mutexes it owns. If an owner itself waits on another Mutex, the inherited priority propagates through the dependency chain.

Priority inheritance bounds interference; it does not make long critical sections inexpensive. Keep the protected region bounded and do not perform unrelated blocking operations while owning a Mutex.

8.1.2 Mutex versus binary Semaphore

Property	Mutex	Binary Semaphore
Stored state	locked/unlocked	count 0/1
Owner	yes	no
Who may release	owner only	any task or ISR permitted by API
Priority inheritance	optional	no
Intended meaning	exclusive shared-state access	occurrence or availability

8.2 Condition-variable helpers

RK0 provides helpers that compose a Sleep Queue with a Mutex:

```
while (!PredicateIsTrue())
{
    RK_ERR err = kCondVarWait(&condition,
                             &stateLock,
                             RK_WAIT_FOREVER);
    K_ASSERT(err == RK_ERR_SUCCESS);
}
```

The caller tests an application predicate while owning the Mutex. Waiting releases the Mutex and joins the Sleep Queue without allowing another task to slip between those two actions. After a successful wake, the helper reacquires the Mutex before returning.

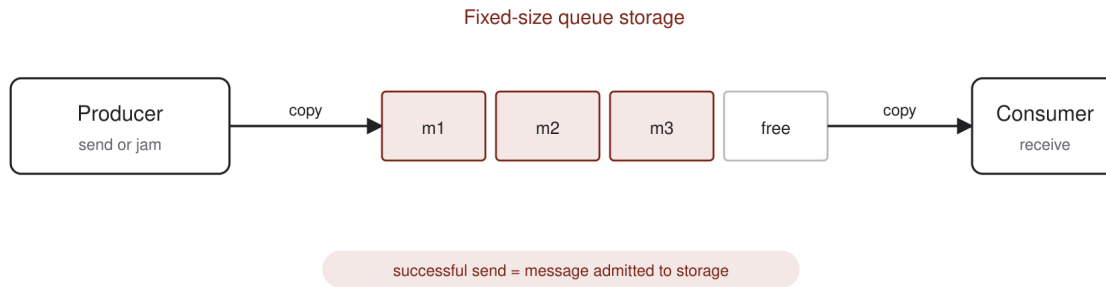
Always retest the predicate in a loop. A wake means that the condition may have changed; it is not proof that this particular task may proceed.

`kCondVarSignal()` wakes one waiter. `kCondVarBroadcast()` wakes all. The application decides which signalling discipline preserves its invariant.

9 Buffered message passing

9.1 Message Queues

A Message Queue transfers fixed-size messages by copy and preserves their order up to the configured queue capacity.



Capacity absorbs bursts. It does not increase long-term throughput.

Fixed-size Message Queue

Use a Message Queue when every accepted message remains relevant and a bounded buffer should absorb short differences between producer and consumer timing.

A successful send means one of two things:

- the message was copied directly into an already-waiting receiver's buffer; or
- the message was copied into queue storage.

It does not mean that the receiver has processed the message.

9.1.1 Declare and initialise

```

typedef struct
{
    ULONG timestamp;
    UINT value;
} Sample;

#define SAMPLE_DEPTH 4U

static RK_MESG_QUEUE sampleQueue;
RK_DECLARE_MESG_QUEUE_BUF(sampleQueueStorage,
    Sample,
    SAMPLE_DEPTH)

VOID InitSampleQueue(VOID)
{
    RK_ERR err = kMsgQueueInit(&sampleQueue,
        sampleQueueStorage,
        RK_MESGQ_MESG_SIZE(Sample),
        SAMPLE_DEPTH);
    K_ASSERT(err == RK_ERR_SUCCESS);
}
  
```

Message size is rounded to one of the supported sizes: 1, 2, 4 or 8 words. The declaration macro allocates matching word-aligned storage. Payloads larger than 8 words should normally be placed in a fixed-block Memory Partition and transferred through a one-word queue of pointers.

`kMesgQueueCreate()` allocates only the queue control object. Queue storage remains application-owned and must stay valid until `kMesgQueueDestroy()` succeeds. Destruction requires an empty, unowned queue with no waiting senders, receivers or broadcast receivers.

9.1.2 Send and receive

```
Sample sample = ReadSample();

RK_ERR err = kMesgQueueSend(&sampleQueue,
                           &sample,
                           RK_NO_WAIT);
if (err == RK_ERR_BUFFER_FULL)
{
    RecordDroppedSample();
}
```

`kMesgQueueSend()` appends. `kMesgQueueJam()` inserts at the front. A sender may wait when the queue is full. A receiver may wait when it is empty. Waiting senders and receivers are selected by effective priority, FIFO among equal priorities.

`kMesgQueuePeek()` copies the front message without consuming it. `kMesgQueueReset()` discards buffered messages and releases current waiters. `kMesgQueueQuery()` reports buffered messages and waiting senders or receivers.

A queue does not create throughput. Capacity absorbs bursts and jitter. Over time, either the producer waits, messages are intentionally dropped, or the consumer must sustain the arrival rate.

9.1.3 Send callback

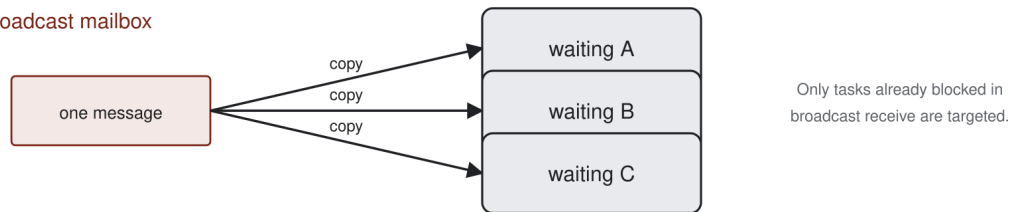
When enabled, `kMesgQueueInstallSendCbK()` installs a callback after a successful normal send or jam. The callback runs synchronously inside the sending operation. It must be short and non-blocking, normally setting a Task Event or posting a Semaphore.

The callback is notification, not delivery. The queue remains the authority for the message.

9.2 Mailboxes

`RK_MBOX` is an alias for a Message Queue whose depth is exactly one. Ordinary post and pend operations retain queue semantics.

The dynamic aliases are `RK_MBOX_HANDLE`, `kMboxCreate()` and `kMboxDestroy()`; their lifecycle is the one-slot Message Queue lifecycle.

Overwrite mailbox**Broadcast mailbox***One-slot Mailbox operations***9.2.1 Overwrite**

`kMsgQueuePostOvw()` replaces the pending value instead of waiting for the slot to become free. Use this when only the newest not-yet-consumed value matters to one receiver.

Overwrite is not the same as MRM. A Mailbox has one slot and normal receivers compete for it; MRM retains versions safely while multiple readers finish independently.

9.2.2 Broadcast

Tasks first block in `kMboxBroadcastRecv()`. A later `kMboxBroadcast()` copies one message to every task that is already waiting in broadcast receive.

If no broadcast receiver is waiting, no message is deposited and the current API returns `RK_ERR_BUFFER_EMPTY`. A broadcast is therefore a simultaneous handoff to present waiters, not retained publication for future readers.

9.3 Ports

A Port is a Message Queue bound to one owner task. It is not another kernel object.

```

static RK_MSG_QUEUE dacPort;
RK_DECLARE_PORT_BUF(dacPortStorage, DacRequest, 3U)

VOID InitDacPort(VOID)
{
    RK_ERR err = kPortInit(&dacPort,
        dacPortStorage,
        RK_MSGQ_MSG_SIZE(DacRequest),
        3U,
        dacManagerHandle);
    K_ASSERT(err == RK_ERR_SUCCESS);
}
  
```

Clients name the owner task when sending:

```
kPortSend(dacManagerHandle, &request, RK_WAIT_FOREVER);
```

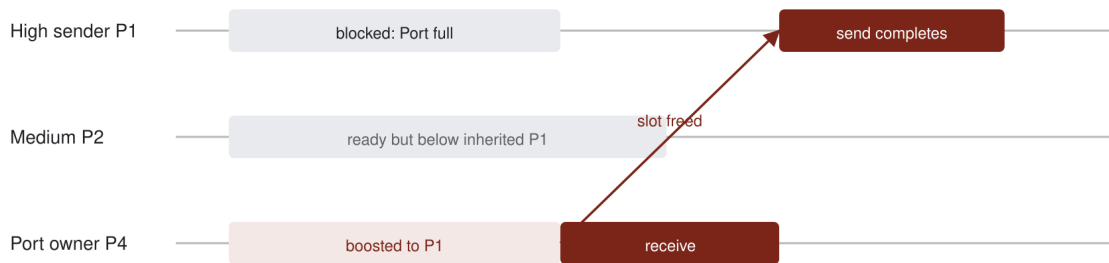
The owner receives from its own endpoint:

```
kPortRecv(&request, RK_WAIT_FOREVER);
```

Only the owner may receive. A task can own one Port.

9.3.1 Backpressure and owner priority

When the Port is full and a sender blocks, the owner can inherit the most urgent blocked sender priority so that it can receive and relieve the full-queue dependency.



The boost lets the owner receive and relieve a blocked-send dependency.

Port backpressure and owner boost

No boost occurs merely because a message is buffered. Successful `kPortSend()` still means only that the message was admitted. If a client needs an answer or completion result, use a Channel or design an explicit return path.

9.3.2 Single coordination authority

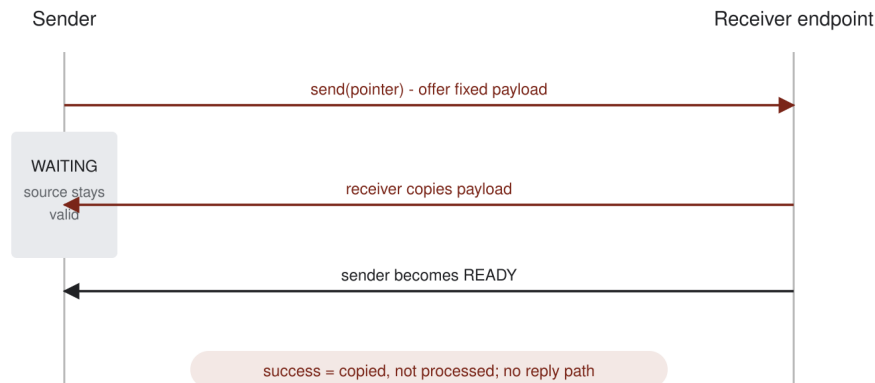
Do not expose the same resource through both direct Mutex-protected access and a Port-owned server path. Those are competing ownership authorities and can form dependency cycles that priority inheritance cannot safely resolve.

RK0 enforces the rule on ownership-transfer paths: Port send, receive, jam, overwrite and reset reject a running task that owns any Mutex.

10 Synchronous message passing

10.1 Rendezvous

Rendezvous is unbuffered task-to-task message passing. The sender names a receiver and waits until that receiver copies the fixed-size payload.



Unbuffered Rendezvous handoff

The receiver task owns an implicit endpoint configured once:

```

typedef struct
{
    UINT sequence;
    UINT sample;
} Sample;

RK_ERR err = kRendezvousInit(controllerHandle,
                             sizeof(Sample)); /* establish a link with a synchronous comm of max
sizeof(Sample) bytes */
K_ASSERT(err == RK_ERR_SUCCESS);
  
```

The byte size must be non-zero and a multiple of **RK_WORD_SIZE**. There is no standalone Rendezvous object and no kernel payload buffer.

10.1.1 Send-and-wait-until-copied

```

Sample sample = MakeSample();

RK_ERR err = kRendezvousSend(controllerHandle,
                             &sample, sizeof(sample),
                             RK_WAIT_FOREVER);
  
```

If the receiver is waiting, the payload is copied immediately. Otherwise, the sender waits and its source pointer remains part of the pending offer. Multiple senders may wait, selected by effective priority, but their messages are not buffered in a queue.

The receiver calls:

```

Sample sample;
/* receiver to a local sample. nBytes stores the number of received bytes */
RK_ERR err = kRendezvousRecv(&sample, &nBytes,
                             RK_WAIT_FOREVER);
  
```

A successful send means that the receiver copied the payload into its own storage. It does not mean that the receiver processed it, and there is no reply path.

When an urgent sender waits for a less-urgent receiver, the receiver inherits the highest waiting sender priority until the handoff dependency is removed.

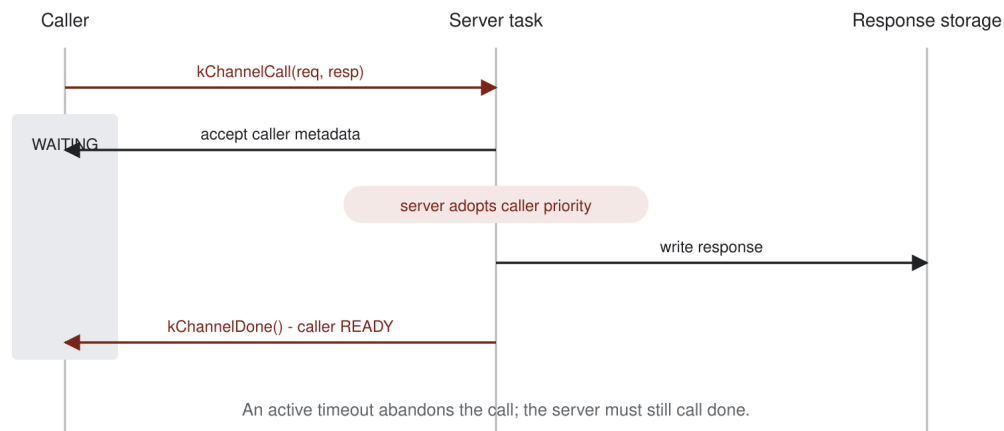
Buffer lifetime: the sender's source storage must remain valid until `kRendezvousSend()` returns. A sender timeout invalidates the offer, so a later receiver cannot consume a stale pointer.

Use Rendezvous when delivery acknowledgement is sufficient. Use a Channel when the caller needs a computed response. Use a Port or Message Queue when backlog is part of the contract.

Rendezvous send and receive reject a task that owns a Mutex, preserving the single-authority rule.

10.2 Call Channels

A Channel is a synchronous N:1 procedure call to a server task. The client provides request and response storage, calls the server and waits for `kChannelDone()`.



Channel request, accept and completion

There is no `RK_CHANNEL` object and no initialisation call. Every task has an implicit server endpoint. Pending call metadata lives in the caller's Task Control Block; accepted metadata is copied into stack-local `RK_CALL_DATA` on the server.

10.2.1 Client

```

ChannelRequest request = {
    .sample = 123U,
    .scale = 3U
};
ChannelResponse response = {0U};

RK_ERR err = kChannelCall(serverHandle,
    &request,
    &response,
    sizeof(request),
    RK_WAIT_FOREVER);
if (err == RK_ERR_SUCCESS)
{
    UseResult(response.result);
}
  
```

`kChannelCall()` always waits; `RK_NO_WAIT` is not a valid Channel call timeout.

10.2.2 Server

```
while (1)
{
    RK_CALL_DATA call = {0};

    RK_ERR err = kChannelAccept(&call,
                               RK_WAIT_FOREVER);
    K_ASSERT(err == RK_ERR_SUCCESS);

    ChannelRequest const *request = call.reqPtr;
    ChannelResponse *response = call.respPtr;

    response->result = request->sample * request->scale;

    err = kChannelDone(&call);
    K_ASSERT(err == RK_ERR_SUCCESS);
}
```

Many callers may wait on one server, ordered by effective priority. Only one call may be active. After accepting a call, the server adopts the caller's effective priority until `kChannelDone()`.

Successful client completion means that the server called `kChannelDone()`. The kernel does not interpret the request or response payload.

10.2.3 Timeout after acceptance

A queued timeout removes the caller before acceptance. A timeout after acceptance creates an **abandoned call**:

- the caller returns `RK_ERR_TIMEOUT`;
- the server may still be executing;
- the server must still call `kChannelDone()`;
- the caller cannot start another Channel call until that completion clears the abandoned call.

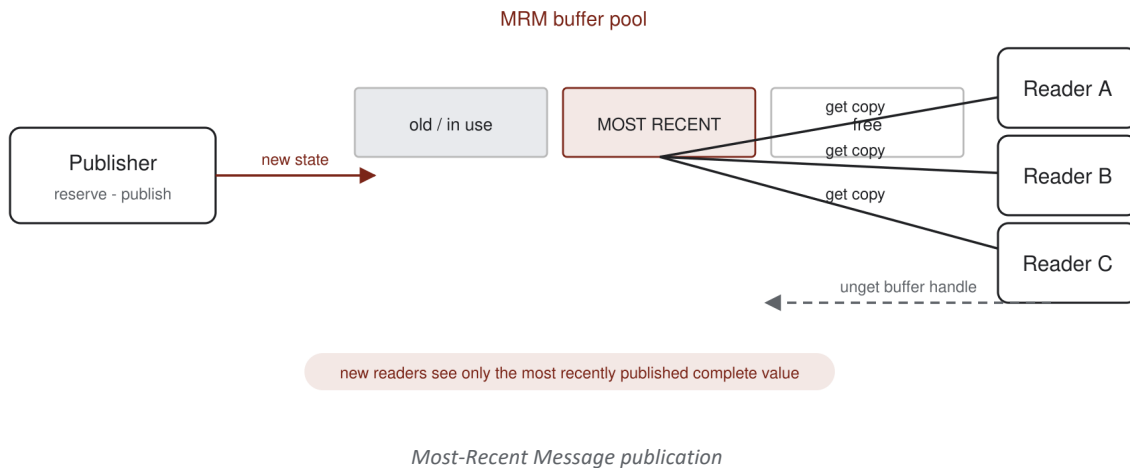
The accepted `RK_CALL_DATA` contains raw application pointers. If a bounded call can time out after acceptance, request and response storage must remain valid until the server calls `kChannelDone()`. Caller-stack objects are therefore unsafe for that case unless their lifetime is otherwise extended.

`kChannelCall()`, `kChannelAccept()` and `kChannelDone()` reject a running task that owns a Mutex. Always pair every successful accept with done, including abandoned calls.

11 Latest-state publication

11.1 Most-Recent Message

MRM is a non-blocking 1:N protocol for publishing the latest complete state. New readers never walk through a backlog of obsolete samples.



Use it when:

- one writer publishes state;
- several readers operate at independent rates;
- the latest complete value is more relevant than every intermediate value;
- readers must never observe a partially updated object.

11.1.1 Configure both pools

MRM uses two equal-length pools:

1. **RK_MRM_BUF** control buffers, which hold reader counts and payload addresses;
2. application payload buffers.

For one writer and R readers, configure $R + 2$ elements: one per communicating task plus one spare. For an application type **PlantState**:

```
#define N_READERS 3U
#define N_MRM_BUFS (N_READERS + 2U)

static RK_MRM stateMrm;
static RK_MRM_BUF stateControlPool[N_MRM_BUFS];
static PlantState statePayloadPool[N_MRM_BUFS] K_ALIGN(4);

RK_ERR err = kMRMInit(&stateMrm, stateControlPool,
                     statePayloadPool, N_MRM_BUFS,
                     RK_TYPE_WORD_COUNT(PlantState));
K_ASSERT(err == RK_ERR_SUCCESS);
```

Use static zero-initialised storage for the **RK_MRM_BUF** pool.

kMRMCreate() allocates only the MRM control object. The control-buffer and payload pools remain application-owned. Before **kMRMDestroy()**, the application must ensure that no reader retains a version and

no writer reservation remains outstanding. An unreferenced current publication is reclaimed during destruction.

11.1.2 Writer: reserve and publish

```
RK_MRM_BUF *buffer = kMRMReserve(&stateMrm);  
if (buffer != NULL)  
{  
    PlantState state = ReadPlantState();  
    RK_ERR err = kMRMPublish(&stateMrm,  
                             buffer,  
                             &state);  
    K_ASSERT(err == RK_ERR_SUCCESS);  
}
```

Reservation is private to the writer until publication. Publishing copies the payload and makes that buffer the only version visible to new readers.

11.1.3 Reader: get and unget

```
PlantState state;  
RK_MRM_BUF *buffer = kMRMGet(&stateMrm, &state);  
  
if (buffer != NULL)  
{  
    ProcessState(&state);  
  
    RK_ERR err = kMRMUnget(&stateMrm, buffer);  
    K_ASSERT(err == RK_ERR_SUCCESS);  
}
```

kMRMGet() copies the current payload and returns a handle to the version from which it was taken. Pair every successful get with exactly one unget of that same handle. The handle lets RK0 retain an old version while readers that acquired it are still active.

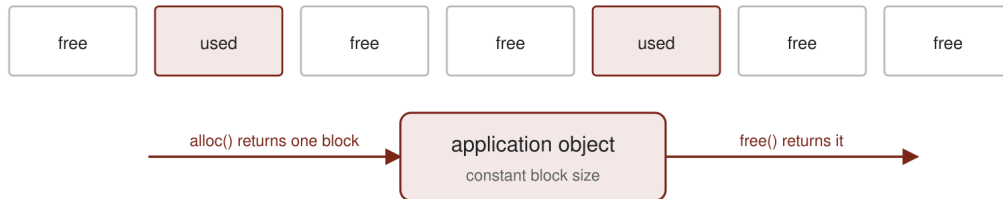
MRM never waits, wakes a task or donates priority. **kMRMReserve()** returns **NULL** when no buffer is available; **kMRMGet()** returns **NULL** before the first publication.

12 Deterministic memory

12.1 Memory Partitions

A Memory Partition allocates homogeneous fixed-size blocks from application-supplied storage.

Fixed-size block pool



No variable-size search, coalescing, or heap fragmentation model.

Fixed-block Memory Partition

Allocation and free are immediate; they do not wait or affect task priority. There is no variable-size search, block splitting, coalescing or general heap-fragmentation model.

```
typedef struct
{
    RK_TICK timestamp;
    CHAR text[60];
} LogRecord;

#define LOG_BLOCKS 8U

static RK_MEM_PARTITION logMemory;
RK_DECLARE_MEM_POOL(LogRecord, logPool, LOG_BLOCKS)

VOID InitLogMemory(VOID)
{
    RK_ERR err = kMemPartitionInit(&logMemory,
                                   logPool,
                                   sizeof(LogRecord),
                                   LOG_BLOCKS);
    K_ASSERT(err == RK_ERR_SUCCESS);
}
```

Allocate and test for exhaustion:

```
LogRecord *record = kMemPartitionAlloc(&logMemory);
if (record != NULL)
{
    FillRecord(record);
}
```

Return the block exactly once to its original partition:

```
kMemPartitionFree(&logMemory, record);
```

Block size is rounded up to a word boundary. The pool must be aligned, sized for the rounded block geometry and contain at least one block. Allocation does not clear a reused block; initialise application fields before use.

12.1.1 Mail queue pattern

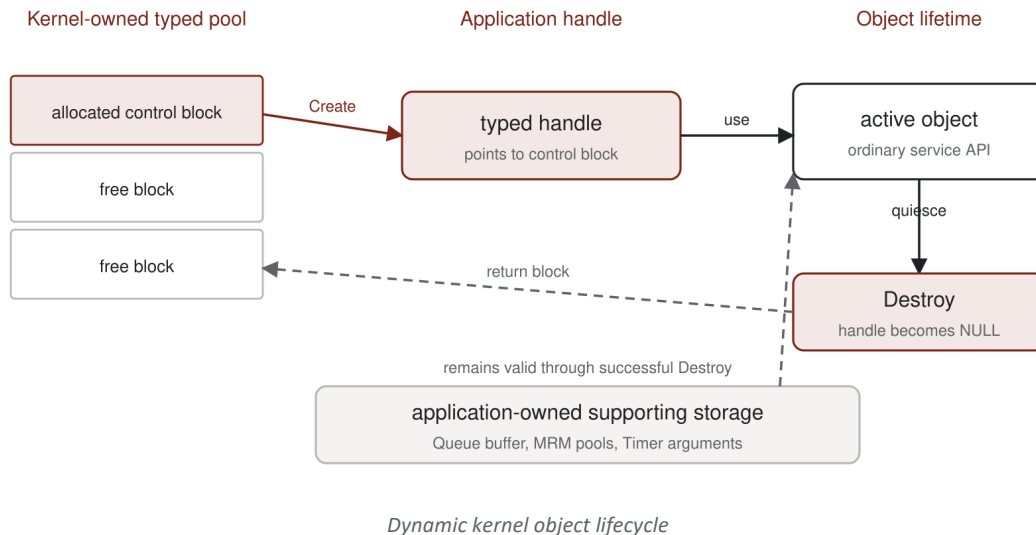
For variable or larger payloads, combine a Memory Partition with a one-word Message Queue of pointers:

1. sender allocates a block;
2. sender fills it and enqueues its pointer;
3. receiver dequeues and processes the block;
4. receiver returns it to the same partition.

This keeps kernel copy cost bounded to one word while making lifetime and ownership explicit. The sender must free the block if a non-blocking queue send fails.

13 Dynamic kernel objects

Dynamic creation changes an object's storage and lifetime; it does not change the service's completion, waiting or priority semantics. RK0 still uses fixed-capacity, kernel-owned pools rather than a general heap.



13.1 Configure bounded pools

Enable runtime creation of non-task objects in `kconfig.h` and set the maximum count for each required family:

```
#define RK_CONF_DYNAMIC_OBJECTS      ON
#define RK_CONF_DYNAMIC_SEMAPHORES_MAX 4U
#define RK_CONF_DYNAMIC_MUTEXES_MAX  4U
#define RK_CONF_DYNAMIC_SLEEP_QUEUES_MAX 2U
#define RK_CONF_DYNAMIC_MESG_QUEUES_MAX 4U
#define RK_CONF_DYNAMIC_TIMERS_MAX    2U
#define RK_CONF_DYNAMIC_MRMS_MAX      1U
```

When `RK_CONF_DYNAMIC_OBJECTS` is not set explicitly, it follows `RK_CONF_DYNAMIC_TASK`: dynamic task support enables it; otherwise it is disabled. The two facilities use separate pools and may be configured independently.

Each maximum reserves a kernel-owned Memory Partition for that control-object type. A maximum of zero keeps that family unavailable even when dynamic objects are enabled. Exhausting a family returns `RK_ERR_BUFFER_EMPTY`; it does not fall back to a heap or wait for another object to be destroyed.

Initialise the partitions once before the first create operation:

```
VOID kApplicationInit(VOID)
{
    RK_INIT_OBJ_PARTITIONS
    /* Create startup tasks and static objects. */
}
```

The statement-style macro is a no-op when the feature is disabled. It intentionally discards the initialisation result. Startup code that must inspect failure may call `kObjPartitionsInit()` directly inside an `RK_CONF_DYNAMIC_OBJECTS` feature guard.

13.2 Static and dynamic forms

Property	Static object	Dynamic object
Control-block storage	declared by the application	allocated from a typed kernel pool
Start of lifetime	matching Init call	matching Create call
Reference	address of the object	typed handle such as <code>RK_MUTEX_HANDLE</code>
End of lifetime	normally application lifetime	matching Destroy call after quiescence
Service operations	unchanged	unchanged; pass the handle as the object pointer

`Create` allocates the kernel control block, not every buffer referenced by that control block:

Dynamic family	Storage that remains application-owned
Semaphore, Mutex, Sleep Queue	none
Message Queue or Mailbox	message buffer supplied to <code>kMsgQueueCreate()</code> or <code>kMboxCreate()</code>
Timer	callback argument and anything it references
MRM	<code>RK_MRM_BUF</code> array and payload pool supplied to <code>kMRMCreate()</code>

Application-owned storage must remain valid until the corresponding `Destroy` operation succeeds.

For example, a dynamically allocated Mutex is used by the ordinary Mutex operations:

```
static RK_MUTEX_HANDLE stateLock = NULL;

RK_ERR err = kMutexCreate(&stateLock,
                        RK_PRIO_INHERITANCE);
K_ASSERT((err == RK_ERR_SUCCESS) && (stateLock != NULL));

err = kMutexLock(stateLock, RK_WAIT_FOREVER);
K_ASSERT(err == RK_ERR_SUCCESS);
UpdateSharedState();
err = kMutexUnlock(stateLock);
K_ASSERT(err == RK_ERR_SUCCESS);

err = kMutexDestroy(&stateLock);
K_ASSERT((err == RK_ERR_SUCCESS) && (stateLock == NULL));
```

Create and Destroy take the address of the typed handle variable. A failed Create leaves that output handle **NULL**. A successful Destroy clears the handle variable passed to it and returns the control block to its family pool.

For a Message Queue, only the control block is dynamic:

```
#define WORK_DEPTH 4U

typedef struct
{
    ULONG command;
    ULONG argument;
} WorkItem;

static RK_MSG_QUEUE_HANDLE workQueue = NULL;
RK_DECLARE_MSG_QUEUE_BUF(workStorage, WorkItem, WORK_DEPTH)

RK_ERR err = kMsgQueueCreate(&workQueue,
                            workStorage,
                            RK_MSGQ_MSG_SIZE(WorkItem),
                            WORK_DEPTH);
K_ASSERT((err == RK_ERR_SUCCESS) && (workQueue != NULL));

/* Use workQueue with the ordinary Message Queue operations. */

/* First stop users and drain the queue. */
err = kMsgQueueDestroy(&workQueue);
K_ASSERT((err == RK_ERR_SUCCESS) && (workQueue == NULL));
```

workStorage is not released by Destroy. It is application storage and may be reused only after the queue has been destroyed successfully.

13.3 Destruction requires quiescence

Destroy is not a close or broadcast-cancellation operation. The application must first stop new users and allow current operations to finish.

Dynamic family	Required state before Destroy
Semaphore	no waiting tasks; a stored count may be discarded
Mutex	unlocked, no owner and no waiting tasks
Sleep Queue	no waiting tasks
Message Queue or Mailbox	empty, no waiting senders or receivers, no broadcast receivers and no Port owner

Dynamic family	Required state before Destroy
Timer	may be armed; Destroy cancels it before release
MRM	no outstanding reader reference and no reserved unpublished buffer

The following rules apply to every dynamic family:

- Create and Destroy are task-context operations and must not be called from an ISR.
- A Destroy function accepts only an object produced by the matching Create function. Do not pass the address of a static object.
- The kernel clears only the handle variable passed to Destroy. Copied handles and raw pointer aliases are not tracked; after successful destruction they are stale.
- Quiescence is an application protocol. Prevent another task from starting an operation while the object is being destroyed.
- Give one component authority over the owning handle and the Create/Destroy operations. Other tasks may borrow the handle only while that component guarantees the object's lifetime.
- Application-owned supporting storage must outlive the control object: Message Queue buffers, MRM control and payload pools, and Timer callback arguments remain valid through successful destruction.
- Do not destroy a Timer from its own callback. The PostProc timer path still uses the Timer state after the callback returns.

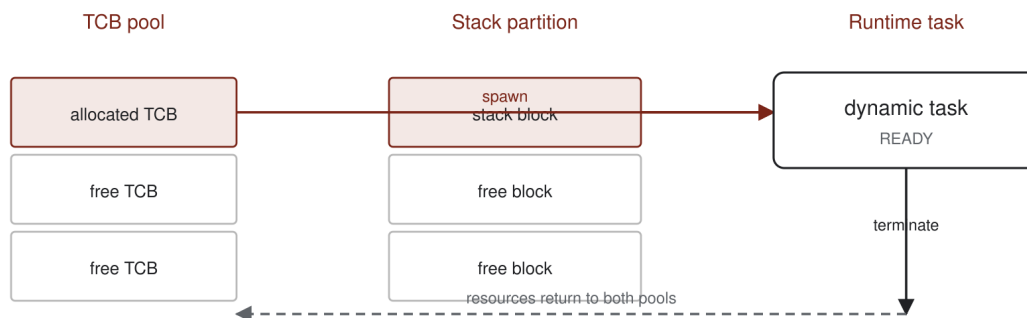
There are no dynamic-object APIs for Memory Partitions, Ports or Task Event Registers. Channel and Rendezvous endpoints remain task-backed. A dynamically created Message Queue cannot be destroyed while it is bound as a Port.

Trace registers a created object normally and unregisters it after successful destruction, so it no longer appears in object listings.

14 Dynamic tasks

Dynamic tasks support middleware that must create and terminate tasks after scheduling has started. They should still come from a bounded, known configuration. In RK0 0.60.0, `RK_CONF_DYNAMIC_TASK` is disabled by default and must be enabled deliberately.

```
#define RK_CONF_DYNAMIC_TASK ON
```



Dynamic task resource lifecycle

Each dynamic task uses:

- one TCB slot from the fixed kernel task pool;
- one stack block from an application-selected Memory Partition.

```
#define DYNAMIC_STACK_WORDS 256U
#define DYNAMIC_TASKS      2U

RK_DECLARE_DYNAMIC_TASK(workerHandle, WorkerTask)
RK_DECLARE_DYNAMIC_STACK_POOL(workerStackMemory,
                               workerStackPool,
                               DYNAMIC_TASKS,
                               DYNAMIC_STACK_WORDS)

static RK_DYNAMIC_TASK_ATTR workerAttributes = {
    .taskFunc = WorkerTask,
    .argsPtr = RK_NO_ARGS,
    .taskName = "DynWork",
    .priority = 3U,
    .preempt = RK_PREEMPT,
    .stackMemPtr = &workerStackMemory
};
```

Initialise the stack partition before spawning:

```
kMemPartitionInit(&workerStackMemory,
                  workerStackPool,
                  sizeof(workerStackPool[0]),
                  DYNAMIC_TASKS);

kTaskSpawn(&workerAttributes, &workerHandle);
```

A successfully spawned task becomes READY immediately and may preempt its creator.

Only runtime-spawned tasks may be terminated. `kTaskTerminate(&handle)` terminates another eligible dynamic task and sets that handle variable to `NULL`. `kTaskTerminateSelf()` defers cleanup to `PostProc` because the running task cannot reclaim its own active stack.

Termination is rejected while the task owns or waits for a resource, remains in an unsupported blocking relation, or has dependants through Port, Channel or Rendezvous state. Copied stale handles remain the application's responsibility even after the handle passed to terminate is cleared.

Real-time rule: runtime creation changes the active task set. Bound when and how many tasks may exist. A dynamic task should use `kSleepUntil()` with a local anchor for periodic work; it did not participate in the scheduler-start phase used by `kSleepRelease()`.

15 Trace and fault handling

15.1 Trace console

When `RK_CONF_TRACE` is enabled, `kTraceInit()` starts an interactive UART-backed trace task. The board supplies a non-blocking receive hook and signals input from its UART ISR.

Name initialised objects so console output is readable:

```
kTraceNameObject(&sampleQueue, "SampleQ");
kTraceNameObject(&stateLock, "StateLk");
kTraceInit();
```

Names are limited by `RK_NAME_SIZE`; with the default size, use at most seven visible characters plus the terminating NUL.

Useful commands include:

Command	Use
top	Task state, effective and nominal priority, dispatches, CPU accounting and stack watermark
list kobjects	Registered kernel objects and last operation
list kmesg	Queue depth, owner and waiting senders or receivers
list kipc	Task-backed Channel and Rendezvous state
list ksema	Semaphore and Mutex state
list kmem	Partition size and free blocks
list ktimers	Application timers
hist <name>	Recent operations for one object
hist task/<name>	Effective-priority changes for one task

Trace CPU percentage is diagnostic accounting, not a cycle-accurate profiler. Use the history with the task and object snapshots to understand causality.

15.2 Error handling

In checked builds, invalid API use can call `kErrorHandler()` in addition to returning an error. Configure the handler for the application's fault policy: halt, capture evidence, reset or hand control to a safety mechanism as appropriate.

Treat these as programming errors rather than routine branch conditions:

- using an uninitialised or wrong object;
- using a stale handle after dynamic object destruction;
- destroying a dynamic object while another task can still use it;
- blocking from an ISR;
- recursive Mutex lock;
- unlocking a Mutex without ownership;
- mixing Mutex ownership with Port, Channel or Rendezvous authority;
- freeing a block to the wrong partition;
- letting a task return from its entry function.

Stack overflow and exhausted bounded resources should produce evidence that is useful after reset. The trace stack watermark, object history and application-specific retained fault record are complementary.

16 Choosing a communication service

The services differ mainly in what they retain and where completion occurs.

	Queue	Port	Rendezvous	Channel	MRM
Buffers history	yes	yes	no	no	latest only
Success means	admitted	admitted	copied	reply done	published
Receiver	any	owner	named task	server	many
Priority relation	none*	backpressure	handoff	call duration	none

* A plain owned queue can be configured as a Port; ownership changes the scheduling contract.

Comparison of RK0 message-passing services

Need	Choose	Success means
Count occurrences or units	Semaphore	one token was consumed or posted
Private bitwise occurrences	Task Event Register	requested bits were present and consumed
Wait for a future notification	Sleep Queue	the task was signalled after it began waiting
Preserve a bounded message history	Message Queue	message copied to receiver or queue storage
Keep only one pending newest value	overwrite Mailbox	one-slot value replaced or delivered
Deliver one value to present waiters	broadcast Mailbox	every targeted waiting receiver obtained a copy
Send buffered work to one owner	Port	message admitted to the owner's queue
Deliver directly and wait until copied	Rendezvous	receiver copied the fixed-size payload
Invoke a server and receive a result	Channel	server called kChannelDone()
Publish latest state to many readers	MRM	complete value became current

Ask these questions in order:

1. Is this an occurrence, shared-state ownership or a message?
2. If it is a message, should history be preserved, discarded or never created?
3. Does the sender complete on admission, on copy or on server reply?
4. Is there one owner, one receiver, one server or many readers?
5. What priority dependency appears when a more urgent task waits?

Do not emulate a stronger contract with a weaker service unless the application explicitly accepts the changed completion, timeout and scheduling semantics.

17 Compact API index

This index is for orientation. See [kapi.h](#) for exact parameters, conditional compilation and return values.

Family	Principal operations
Kernel and tasks	kInit, kTaskInit/kCreateTask, kYield, task accessors
Dynamic object pools	RK_INIT_OBJ_PARTITIONS, kObjPartitionsInit
Dynamic tasks	kTaskSpawn, kTaskTerminate, kTaskTerminateSelf
Scheduler control	kSchLock, kSchUnlock
Time	kSleepDelay, kSleepRelease, kSleepUntil, kBusyDelay, kTickGet
Callout timer	kTimerInit/kTimerCreate, kTimerCancel, kTimerDestroy
Task Events	kEventSet, kEventGet, kEventQuery, kEventClear
Semaphore	kSemaphoreInit/kSemaphoreCreate, kSemaphorePend, kSemaphorePost, kSemaphoreQuery, kSemaphoreDestroy
Sleep Queue	kSleepQueueInit/kSleepQueueCreate, kSleepQueueWait, kSleepQueueSignal, kSleepQueueWake, kSleepQueueReady, kSleepQueueBlockReadyTask, kSleepQueueQuery, kSleepQueueDestroy
Mutex	kMutexInit/kMutexCreate, kMutexLock, kMutexUnlock, kMutexQuery, kMutexDestroy
Condition helper	kCondVarWait, kCondVarSignal, kCondVarBroadcast
Message Queue	kMesgQueueInit/kMesgQueueCreate, kMesgQueueSend, kMesgQueueRecv, kMesgQueueJam, kMesgQueuePeek, kMesgQueueReset, kMesgQueueQuery, kMesgQueueDestroy
Mailbox	kMboxInit/kMboxCreate, kMboxPost, kMboxPend, broadcast operations, overwrite, kMboxDestroy
Port	kPortInit, kPortSend, kPortRecv, kPortJam, kPortPostOvw, kPortReset
Rendezvous	kRendezvousInit, kRendezvousSend, kRendezvousRecv
Channel	kChannelCall, kChannelAccept, kChannelDone
MRM	kMRMInit/kMRMCreate, kMRMReserve, kMRMPublish, kMRMGet, kMRMUnget, kMRMDestroy
Memory Partition	kMemPartitionInit, kMemPartitionAlloc, kMemPartitionFree
Trace	kTraceInit, kTraceNameObject, snapshot and history APIs
•	