K0BA Documentation (v0.3.1L)

This is a high-level system description. For API details look at `kapi.h`

K0BA Lite (a.k.a *K0*) is flat: there is no *user-space* and *kernel-space*, and therefore *no system calls*. Low-end MCUs based on the supported architecture (*ARMv7M*) often do not come with an MPU - so privilege levels are limited to the CPU scope: instructions and registers. The added safety does not cover the cost of a system call for a real-time application. For MPU-based hardware the *Minim'All* version will be made public eventually.

The K0 design approach is to reduce the level of indirections the most: ***"as modular as possible, rarely opaque."***

[layeredkernel] | *images/layeredkernel.png*

K0BA does not aim be an application framework. Its goal is to provide a deterministic multitask engine, with a balanced set of services.

Process, Threads and Tasks

A process is composed by an execution image and an address space - the modern process concept is about program execution and about memory isolation. A thread is a logical sequence of instructions. Several threads co-exist in the same address space, within the same process.

On the embedded realm, probably because we lack a better abstraction, we use *multithreading* to fine-tune our load balance, and therefore the responsiveness to achieve real-time.

When we assign to a thread a running stack and a record to keep its meta-data (execution context mainly), now we have an operating system entitity we can handle as an *execution unit*: in K0 we name it a *Task* - and the terms task and thread are used interchangeably on this document.

Thus, the K0 multitasking engine is deployed within a single process on a single address space - as the majority of small kernels you see around.

Multithreading within a single-process is an arrangement: instead of of having a single super-loop, we have many - each one running on its own execution stack.

The cost is the complexity to deal with shared memory, resources, and everything that comes with making every task (thread) to believe it owns the entire processor.

For those attached to operating system concepts, in K0 as every thread is a *fork* from the main thread, conceptually every thread is a kernel thread - and *user threads* are just a label for threads running on a different stack pointer from the *main stack pointer* used by the kernel handlers.

Core Mechanisms

In this section a high-level description of the kernel core mechanisms is provided. These mechanisms are always present: scheduler, timers and memory allocator.

1. O(1) scheduler

K0 employs a priority rate-monotonic scheduler. A higher priority task always preempts a low priority task. The algorithm is supposed to work up to a load of around 70%.

Tasks with shorter periods ideally are assigned to higher priorities.

The scheduler supports 255 tasks and 32 priorities. Remarkably, it is an O(1) scheduler: no matter the number of tasks to choose from, the scheduler will always spend the same amount of (logical) time.

1.1. Data Structures

1.1.1. Task Control Block

Threads are represented as Tasks. Every task is associated to a Task Control Block structure. This is a record for stack, resources and time management:

Task Control Block
Task name
Saved Stack Pointer
Stack Adddress
Stack Size
Status (ready, running, sending…)
Assigned Priority
Current Priority
User Assigned ID
Kernel Assigned ID
Time-Slice
Remaining time-slice
Last wake-time
Flags: run-to-completion, timed-out, yielded
Pending resources: semaphores, mutexes, timers, message passing, etc.
Monitoring: dispatch counter, lost signals, number of preemptions, preempted by
Aggregated list node

Tasks are static - they cannot be created on runtime, to be destroyed, to fork or join.

On practice, tasks are either running or waiting for its turn to run. When there is no blocking condition, we say a task is `READY` - it is just waiting for the scheduler. When there is a blocking condition the task is `WAITING`. A tasks needs to be READY to be picked up by the kernel scheduler and switch to `RUNNING`.

[taskstates] | *images/taskstates.png*

We extend the WAITING state logically as:

- PENDING : the task suspended itself waiting for a direct signal.
- SUSPENDED: the task has been suspended by another task, and will switch to *READY* when signalled.
- SLEEPING: a task is normally sleeping for an *event*. This is a broad concept we explore a bit later.
- BLOCKED: a task is blocked on a critical region, when trying to access a busy resource.
- SENDING/RECEIVING: same as blocked, but the busy resource is a kernel object for message passing (or similar) mechanism.

1.1.2. Task Queues

The backbone of the queues where tasks will wait for its turn to run is a circular doubly linked list: removing any item from a doubly list takes O(1) (provided we don't need to search the item). As the kernel is aware of each task's address, adding and removing is always O(1). Singly linked lists, can't achieve O(1) for removal in any case.

A circular doubly linked-list ADT is employed:

```
   HEAD                                        TAIL
   _____     _____     _____     _____          _____
  |     |-->|     |-->|     |-->|     |-->   <--|     |
  |DUMMY|   |  H  |   | H+1 |   | H+2 |  . . .  |  T  |
<-|_____|<--|_____|<--|_____|<--|_____|         |_____|-->
|_____________________________________________________|


 - INITIALISE

 The list is initialised by declaring a node, and assigning its previous
 and next pointers to itself. This is the anchored reference.

        _____
      __|__   |
     |     |-->
     |DUMMY|
   <-|_____|
    |____|
```

```
    - INSERT AFTER

       When list is empty we are inserting the head (that is also the tail).
       If not empty:
        To insert on the head, reference node is the dummy.
        To insert on the tail, reference node is the current tail
        (dummy's previous node).
            _____     _____     _____
           | ref |-> | new |-> | old |->
        . .|     |   |next |   | next| . . .
         <-|_____| <-|_____| <-|_____|

    - REMOVE A NODE

        To remove a node we "cut off" its next and previous links, rearranging
        as: node.prev.next = node.next; node.next.prev = node.prev;


                      ________________
              _____  |  _____     ___|_
             |     |-> |     x   |     |->
          . . .|prev |   |node |   | next| . . .
             <-|_____|   x_____| <-|_____|
              |_______________|


      To remove the head, we remove the dummy's next node
      To remove the tail, we remove the dummy's previous node
      In both cases, the list adjusts itself
```

Thus, comes another design choice towards achieving O(1). The global ready queue is a table of FIFO queues—each queue dedicated to a priority—and not a single ordered queue. So, enqueuing a ready task is always O(1). If tasks were placed on a single ready queue, the time complexity would be O(n), given the sorting needed.

1.2. The scheduling algorithm

It goes like this: as the ready queue table is indexed by priority - the index 0 points to the queue of ready tasks with priority 0, and so forth, and there are 32 possible priorities - a 32-bit integer can represent the state of the ready queue table. It is a BITMAP:

```
The BITMAP computation: ((1a) OR (1b)) AND (2), s.t.:

(1a) Every Time a task is readied, update: BITMAP |= (1U << task->priority );
(1b) Every Time an empty READY QUEUE becomes non-empty, update: BITMAP |= (1U <<
queueIndex)
(2): Every Time READY QUEUE becomes empty, update: BITMAP &= ~(1U << queueIndex);
EXAMPLE:
```

```
  Ready Queue Index :     (6)5 4 3 2 1 0
          Not empty :      1 1 1 0 0 1 0
                           ------------->
                 (LOW)  Effective Priority  (HIGH)
In this case, the scenario is a system with 7 priority task levels. Queues with
priorities 6, 5, 4, and 1 are not empty.
```

The idle task priority is assigned by the kernel, during initialisation taking into account all priorities the system programmer has defined. Unless user-tasks are occupying all 32 piorities, the Idle Task is treated as an ordinary lowest-priority and has a position in the queue. If not, the idle task on practice will have no queue position, and will be selected when the BITMAP is 0. In the above bitmap, the idletask is in readyQueue[6] .

Given this mask, we know that we shall start inspecting on the LSBit and stop when the first 1 is found. There are uncountable manners of doing this. The approach I chose is:

(1) Isolate the rightmost '1':

```
RBITMAP = BITMAP & -BITMAP. (- is the bitwise operator for two's complement: ~BITMAP +
1) `
```

In this case:

```
                          [31]       [0]  :  Bit Position
                            0...1110010   :  BITMAP
                            1...0001110   : -BITMAP
                           =============
                            0...0000010   :  RBITMAP
                                    [1]
```

The rationale here is that, for a number N, its 2's complement -N, flips all bits - except the rightmost '1' (by adding '1') . Then, N & -N results in a word with all 0-bits except for the less significant '1'.

(2) Extract rightmost '1' position:

Within GCC, the _builtin_ctz() function does the trick: it returns the number of _trailing 0-bits within an integer, starting from the LSbit. The number of 'trailing zeroes' equals the position where the first '1' is found, that is also the ready queue index (and hence the priority) of the next task to be dispatched.

Using ARMv7M instructions, a possible solution is to use the CLZ (count lead zeros):

```
.global __getReadyPrio
.type __getReadyPrio, %function
.thumb_func
__getReadyPrio:
CLZ  R12, R0
```

```
MOV  R0, R12
NEG  R0, R0
ADD  R0, #31

BX LR
```

Thus, we subtract 31 from the number of leading zeroes, and get the index.

The source code in K0BA looks like:

```
static inline PRIO kCalcNextTaskPrio_()
{
    if (readyQBitMask == 0U)
    {
        return (idleTaskPrio);
    }
    readyQRightMask = readyQBitMask & -readyQBitMask;
    PRIO prioVal = (PRIO) (__getReadyPrio(readyQRightMask));
    return (prioVal);
    // or GCC builtin (more portable)
    //return (PRIO)(__builtin_ctz(readyQRightMask));
}

VOID kSchSwtch(VOID)
{
    nextTaskPrio = calcNextTaskPrio_();
    K_TCB* nextRunPtr = NULL;
    K_ERR err = kTCBQDeq( &readyQueue[nextTaskPrio], &nextRunPtr);
    if ((nextRunPtr == NULL) || (err != K_SUCCESS))
    {
        kErrHandler(FAULT_READYQ);
    }
    runPtr = nextRunPtr;
}
```

1.3. Scheduler Determinism

This is a simple test to establish some evidence the scheduler obeys the preemption criteria: a higher priority task always preempts a lower priority task.

Task1, 2, 3, 4 are in descending order of priority. If the scheduler is well-behaved, we shall see counters differing by "1".

```
volatile UINT32 counter1;
volatile UINT32 counter2;
volatile UINT32 counter3;
volatile UINT32 counter4;
```

```
VOID Task1(VOID)
{
    while(1)
    {
        counter1++;
        kPend();
    }
}

VOID Task2(VOID)
{
    while(1)
    {
        counter2++;
        kSignal(1); /* shall immediately be preempted by task1 */
        kPend();    /* suspends again */

    }
}


VOID Task3(VOID)
{
    while(1)
    {
        counter3++;
        kSignal(2);  /* shall immediately be preempted by task2 */
        kPend();     /* suspends again */
    }


}

VOID Task4(VOID)
{
    while(1)
    {
        counter4++;
        kSignal(3); /* shall immediately be preempted by task3 */
                        /* as the lowest priority task it will only be resumed
                           after all higher priority tasks are suspended */
    }

}
```

This is the output after some time running:

[signaldet] | *images/signaldet.png*

In the above example, we use direct signals. For semaphores:

```
K_SEMA sema1;
K_SEMA sema2;
K_SEMA sema3;
K_SEMA sema4;
volatile UINT32 counter1, counter2, counter3, counter4; /*dont code like this*/
VOID kApplicationInit(VOID)
{
    kSemaInit(&sema1, 0);
    kSemaInit(&sema2, 0);
    kSemaInit(&sema3, 0);
    kSemaInit(&sema4, 0);
    counter1=counter2=counter3=counter4=0; /*dont code like this*/
}

VOID Task1(VOID)
{

    while (1)
    {
        counter1++;
        kSemaWait(&sema1, K_WAIT_FOREVER);
    }
}

VOID Task2(VOID)
{

    while (1)
    {
        counter2++;
        kSemaSignal(&sema1);
        kSemaWait(&sema2, K_WAIT_FOREVER);
    }
}

VOID Task3(VOID)
{
    while (1)
    {
        counter3++;
        kSemaSignal(&sema2);
        kSemaWait(&sema3, K_WAIT_FOREVER);
    }
}

VOID Task4(VOID)
{
    while (1)
    {
```

```
        counter4++;
        kSemaSignal(&sema3);
    }
}
```

[semadet] | *images/semadet.png*

Here tick is running @ 1ms (1KHz)

1.4. Common scheduling pitfalls

To avoid the most common pitfalls when scheduling tasks the system programmer should be aware that:

- The scheduler behaviour is to choose the highest priority READY task to run. Always
- ***IMPORTANT***: Don't overlook the READY state of a task. If no time slice is enabled, the only way a task will switch from *RUNNING* to *READY* is by yielding. Otherwise it can only go from *RUNNING* to *WAITING* (equivalent to) and then to *READY*.
- So with no time-slice and no blocking conditions a good practice is to *yield* or to *sleep*(1) at the end of a task loop so other tasks will have the opportunity to run.
- A time slice is not a burst. A higher priority task when ready will pause a lower priority task in the middle of its time slice, resuming from where it left.
- Make sure the number of tasks and the highest (lowest effective) assigned priority is correct in `kconfig.h`. If wrong, the scheduler might not run one or more tasks or hard fault when switching.

2. Timers

2.1. Task Delays

The primitive `sleep(t)` suspends a task on a SLEEPING state, for t ticks starting to count when called.

For periodic activations, use `sleepuntil(p)` in which p is an absolute suspension period of time in ticks. The kernel adjusts any time drift/jitters that might happen in-between calls. If time-slice scheduler is enabled, this primitive is not available.

To busy-wait (active delay within a task) you can use the `busy(t)` primitive.

2.2. Application Timers

Time Control Block
Mode: Reload/OneShot
Callout Function

Time Control Block
Callout Arguments
Timeout
Current Delta Tick
Next Timer Address

K0 offers two types of application timers - one-shot and auto-reload. Both have the system tick as the time reference and are countdown timers.

The system programmer needs to be aware that the callout will run within a deferred handler, that is a run-to-completion system-task.

One-shot timers that are within a task loop will naturally be activated periodically. A remark is that timers leverage a delta queue. Suppose you have a set of timers T1, T2, T3. They will countdown from 8, 6 and 10 ticks, respectively. For a regular queue, the node sequence is RQ = <(T1, 8), (T2, 6), (T3, 10)> (at tick=0). The delta queue would have pairs ordered as a sequence (tick=0): DQ = <(T2, 6), (T1, 2), (T3, 2)>. So having to decrease only the list head for any amount of timers, yields O(1) time-complexity within the interrupt handler.

3. Memory Allocator

Memory Allocator Control Block
Associated Block Pool
Number of Blocks
Block Size
Number of Free Blocks
Free Block List

Bear in mind that the standard `malloc()` leads to fragmentation and (also, because of that) is highly undeterministic. Unless we are using it once - to allocate memory before starting up, it doesn’t fit.But often we need to 'multiplex' memory amongst tasks over time, that is to dynamic allocate and deallocate.

To avoid fragmentation we use fixed-size memory blocks. A simple approach would be a static table marking each block either as free or taken. With this pattern you will need to 'search' for the next available block, if any - the time for searching changes - what is indeterministic.

A suitable approach is to keep track of what is free using a linked list of addresses - a dynamic table. We use "meta-data" to initialise the linked-list - every address holds the "next" address value.

This approach limits that the minimal size of a block is the size of a memory address - 32-bit for our supported architecture. Yet, this is the cheapest way to store meta-data. If not storing on the empty address itself, an extra 32-bit variable would be needed to each block, so it could have a size that is less than 32-bit.

When a routine calls `alloc()` the address to be returned is the one free list is pointing to, say addr1. Before, we update free list to point to the value stored within addr1, say addr8.

When a routine calls `free(addr1)`, we overwrite whatever has been written in addr1 with the value freelist points to (if no more `alloc()` were issued, it still is addr8), and addr1 is the freelist head again.

A major hazard is having a routine writing to non-allocated memory within a pool, as it will spoil the meta-data.

Inter-task synchronisation and communication (ITC)

In this section a high-level description of the mechanisms used for synchronisation and communication between tasks is presented. The most part of these mechanisms are components that can be enabled/disabled and configured - exception is for *direct signals*.

The design has event-driven systems in mind, and great effort has been put on the design of the mechanisms here presented. K0 handle synchronisation by events using *direct signals*, *semaphores* and *event sleep/wake*.

As fully synchronous message passing is an elegant way of controlling the flow of several embedded applications the idea of *message-as-a-signal* is used for *mailboxes*. As mailboxes do not use copy semantics, formally they are a shared memory protocol.

True message passing is provided by *message queues*, which rely on copy semantics. Blocking and unblocking behaviour is avaialable for sending and receiving.

For applications like servoloops, the limitation of traditional message queues is handled by *Cyclical Asynchronous Buffers*: *Pump-Drop* messages - they provide a fully asynchronous one-to-many communication channel, that is concerned on keeping readers updated at the pace of a given producer.

Finally, stream of bytes can be passed by *Pipes*. Pipes within K0 are very flexible and rely on events to synchronise.

1. Direct Signals

Some tasks are entirely reactive and have a single responsibility. It is cheaper and more effective to allow them to pend themselves - and be signalled without an associated kernel object - such as a semaphore. The `signal(id)` primitive takes a Task ID as a parameter. The `pend()` primitive takes no parameters, acts on the caller task. If a task is signalled when not pending, the error counter "lostSignals" is increased in its task control block. This counter is meant for some diagnostics. The standard mechanism does not check for lost signals - to catch up with. Doing so would turn it into in (degenerated) private (weak) semaphore. A *PENDING* task is placed on the global sleeping queue and removed when signalled—its position in the queue does not matter.

The main use case is for deferred handlers. The lack of a control block associated with a direct signal makes it rudimentary but sufficient, justified by the negligible cost and the ubiquity of single-purpose event-driven tasks sitting idle until notified: after readied, the scheduler will take care of its dispatching, respecting the priority. No shared resource, no contention, no priority inversion.

A task can suspend another task using the `suspend(id)` primitive. Note that only one task can be running, so normally the target will be a task that is `READY` - it will transition to `SUSPENDED` until `signal(id)` happens. A task with priority n cannot suspend a task with priority $m < n$ (effective higher). The use of this primitive is to be an exception as it subverts the event-driven nature of real-time systems, in which tasks themselves trigger a suspension (by pending on a resource for instance) to later be notified to switch to ready, not the opposite.

2. Counter Semaphores

Semaphore Control Block
Signed Counter
Acquirer Task
Waiting Queue
Timeout

K0 semaphores can be classified as 'strong counter' semaphores. Counter means the primitives `signal()` and `wait()` will increase and decrease, respectively, the initial signed value 'N' a semaphore is assigned (a semaphore cannot be initialised with a negative value).

When wait() results in a negative value, the caller will be blocked within the semaphore queue. The negative value of a counter semaphore inform us straight away how many tasks are blocked waiting for a signal.

When signalled only one task is unblocked. If a semaphore has a dedicated queue it is 'strong', because it establishes an order. Not having a queue, make a 'weak' semaphore, as the task to be released depends on the position of the task who signals. In K0BA, tasks block and resume within a semaphore-guarded region, ordered by their priority or on a FIFO discipline (that is configured in kconfig.h). Switching to a FIFO discipline might be useful if you notice low-priority tasks starving.

Semaphores can be used to share resources, leading to an unavoidable priority inversion. The owner of a guarded region as the highest priority task able to pass a `wait()`. When a task blocks on a semaphore, if its priority is higher than the semaphore owner, the owner has its priority raised to the same as the blocking task. When a semaphore signals it is leaving the guarded region so its priority is restored. We recognise this mechanism is more suitable for mutexes semaphores since they can have a single owner and we are working on a better mechanism for semaphores. This feature is also optional.

3. Mutex Semaphores

Mutex Control Block
Locked state
Owner
Waiting Queue
Timeout

Mutexes are a semaphore specialisation. They lock a critical section under strict ownership: only the task that owns a mutex can act on it again. If a task that is not an owner tries to lock it, it switches to a `BLOCKED` state, until the mutex is unlocked - as on semaphores. When a not-owner tries to unlock a mutex, the behaviour is implementation-defined. As it can only mean a programming mistake or a serious system bug, in K0 it will hard fault.

Note that, although K0 provides mutex as a distinct mechanism, "1"-semaphores can provide the same mutual exclusive behaviour, handling priority inversion with the same efficiency - although not providing strict ownership. It is up to the system programmer to decide if the last is a need.

In more resourceful operating systems the mutex ownership also extends to other features, as providing recursive locks and constraining the behaviour of dynamic tasks that own a mutex - they cannot be destroyed or terminate, to name a couple. Both are not relevant features in K0 targets.

4. Events (Sleep/Wake)

Event Control Block
Event ID (self-assigned)
Sleeping Queue
Timeout

An event is a condition in time that will trigger a reaction: a countdown timer reaching zero, or the 7th bit of the 7th received stream on the 7th pin in the 7th fullmoon night in the 7th year of system uptime - being 0. These are 2 events. The reactions we don't bother, it can even be 'to ignore the event'.

Events are *pure signals*, they are absent or present - a counter semaphore is an event recorder - and as it needs to count every `signal()` operation is mapped to a single `wait()` operation - *semaphore signals do not broadcast*. The ability to broadcast the occurrence of an event is paramount on reactive systems.

Say a modern car has its speed increased by the driver. Different groups of mechanisms need to be notified to take some action. The radio so that it might boost the volume. The windshield wiper control will increase its speed. The cruiser will switch off - as it might indicate a need for the driver to regain control, and I'd bet the cruiser switches off before the radio volume is boosted.

So, tasks `sleep(event)` for an event. They rely on another task/ISR to trigger a `wake(event)` that will `READY` *all tasks sleeping on that event at once*. What happens later, is up to the scheduler and the application design. It is not uncommon to wake tasks right before a guarded region by mutex

semaphore - tasks will access the critical region one after another, on the order they arrived.

Within K0BA, an event is associated to a kernel object. The kernel assigns only an ID and a sleeping queue within an' event' object. The associated sleeping queue is used to diminish the overhead of the `wake()` primitive - with a global queue the kernel would need to check if the task is sleeping on a specific event.

As on `VER0.3.1` the current design decision is not providing signal()/wait() for events, that is, the ability to enqueue/dequeue a single task sleeping for an event. Condition variables and monitor-like synchronisation schemes can be constructed with events associated to mutex semaphores.

5. Remarks on synchronisation services

5.1. Installed callbacks for signals

It is tempting to become UNIX-minded and install callbacks on signals. When resuming, a task first checks for any pending signals. If they exist, the task deviates from its normal flow (as within any software interrupt), runs the callback, and deals with any side effects. The overhead is excessive—both in time and space—a need to reserve a space on the stack or provide a stack for the callback; a need to tune the context-switching to happen when there is a signal pending and to get back to the deviated point.

5.2. Time-out on blocking primitives

Blocking primitives within K0BA have a timeout parameter, given in ticks. Only kPend() does not given its logical simplistic nature, and application. If the task cannot perform access to a resource within a time given in system ticks, it will switch back to READY, eventually be dispatched, and force a return K_ERR_TIMEOUT.

5.3. Event Groups (multi-condition synchronisation)

As on VER0.3.1 the decision is not to provide event groups as another abstraction. The ways to leverage multicondition synchronisation are too many, there is no 'enough' abstraction. The current mechanisms when combined can provide multicondition synchronisation in the form of Monitors, Condition Variables, Event Groups and Flags, etc., as demonstrated on the usage patterns at the end of this document.

6. Mailbox

Mailbox Control Block
Message Address
Mailbox status
Waiting queue
Owner task

Mailbox Control Block
Timeout

While in GPOS jargon mailboxes are queues of messages - as a distinction from pipes (that are stream buffers) - in embedded system software, often mailboxes are said to have a capacity of a a single messsage, and more recently you will not find it as a distinct mechanism - you use a 1-message queue.

In K0BA a mailbox is a message-as-signal mechanism. Strictly it is not a message passing - it is a shared memory protocol. The mail passed is the address of an object that contains a message - there is no copy. This object is application-dependent, so sender and receiver must agree on the concrete type and mantain the message scope.

Mailboxes can initialise full/empty, have their own waiting queue and handle priority propagation - when passing a token/message between tasks, they synchronise on a turnstile - what is a handy use for them.

A mailbox interface contract can be defined as follows:

1. A mailbox starts either EMPTY or FULL.
2. If starting FULL, it is initialised with an address.
3. A mailbox will always have an assigned 'owner' after the first send() or recv() primitive. The owner is the task who succeded gaining access to the box.
4. When a producer send() to a FULL mailbox, it is enqueued on the mailbox waiting queue, and its task will now be on state SENDING.
5. Likewise, a consumer blocks when issuing a recv() on an empty mailbox. Task status switches to RECEIVING and it is enqueued on the mailbox waiting queue.
6. A mailbox implements a priority propagation protocol, boosting the owner priority on the occasion a higher priority task blocks waiting.
7. The waiting queue for a mailbox, has a discipline that can either be priority or FIFO. Default is by priority, to be coherent with the scheduler.

You need to be aware of how the priority of tasks and mailbox queue discipline impact the message exchange, and therefore the program flow. Say you have a bad idea: two producers and a consumer with priorities High, Medium and Low, respectively. Producers send() and consumer recv() - no other blocking conditions. If the queue discipline for the mailbox is ordered by priority, the medium priority producer will block once and deadlock - since when the consumer empties the box, who is released is the higher priority - always. If it is a FIFO, it will receive the two messages serially - if you configure as FIFO, mind you are choosing to delay a higher priority task.

6.1. Send-receive

The optional sendrecv() primitive sends a message and waits for a reply - its straight for any synchronous client-server communication - a client sends a request and wait for answer.

6.2. Aynchronous mailboxes methods

As an extension for the same synchronous mailboxes there are primitives asend() and arecv(). Instead of blocking they will return an error.

There two other asynchronous methods: asendovw() - to deposit a message on a mailbox even if it is FULL, and arecvkeep() - to retrieve a message from a mailbox but the box will not switch to EMPTY - so other receivers can grab the same message.

7. Message Queue

Message Queue Control Block
Storage address
Write Index
Read Index
Message Size
Max of Messages
Message Count
Owner task
Timeout

As mentioned, Mailboxes and Message Queues are quite distinct not because of the message size. A queue is indeed a message passing mechanism, taking full data ownership - it is a data-centric mechanism.

For a message queue, user will provide a buffer with enough capacity (number of messages x message size), that will be handled as a circular buffer. Message queues always start empty and they pass by copy.

The primitives for the message queue are send(), recv(), asend(), arecv(),jam() and peek(). Opposed to mailboxes which asynchronous methods are an extension, queues can be configured as either synchronous, asynchronous or both.

Message Queues also propagate priority when blocking, and the blocking queue discipline is configured either as by priority or FIFO. This is not to be confused with the message buffering, the first message placed is the first extracted - except for the jam() that places a messages on the queue front. But who sends/receives a message is a matter of task precedence when accessing a queue.

Note, a queue will behave asynchronously until a recv() is issued to an empty queue and when a send() is issued on a full queue. A task that blocks on a full queue will be released as soon as a reader extracts a message and vice-versa.

This behaviour might be desirable or not. The result, anyway, is that tasks with different periods end up synchronising to the lowest rate - (usually, undesirable for real-time). After a queue blocks full, when it eventually unblocks you might had lost data - after all, a producer when blocked won’t

be able to update its data. Is it the problem? Yes and no. The problem with a blocking queue is that by buffering you are reading from the past.

Between a message that does not arrive, and one that arrives with information that is not useful, there is no practical difference. But if a message arrives with information that will deceive your control loop to perform a wrong action - this is the worse.

7.1. Pump-Drop Messages

Pump-Drop Message Control Block
Allocator
Most Recent Buffer Address
Pump-Drop Buffer
Data Address
Data Size
Readers Count

Pump-Drop Buffers are meant to overcome the drawbacks exposed above for message queues, based on the concept of Cyclical Asynchronous Buffers (CAB) (essentialy as found in the HARTIK kernel). This is a fully asynchronous, one-to-many mechanism, that has an implicit memory allocator for the PD-Buffers.

Primitives for Pump-Drop buffers (PDB) are:

- For writer: reserve() and pump() .
- For readers: fetch() and drop().

The semantics is as follows - remember it is one writer to many readers:

1. When the producer needs to write a new message, first it reserves a PD-buffer. This might already be allocated - if it is allocated but has readers, a new PD-buffer is allocated, to ensure data consistency.
2. Writer now appends a message to be sent (application-dependent) and pump the buffer.
3. With a new buffer pumped, the last one will eventually drop to 0 readers and be deallocated. If it already has 0, it is deallocated.
4. After pumped, the PD buffer is marked as the current buffer. From now on, the former pumped PDB cannot be fetched and will eventually drop to 0 readers and be deallocated.
5. A reader first 'fetches' a message PD-buffer address - what will increase the user count within a it.
6. After 'having' the message, a reader 'drops the PD-buffer': the kernel checks if it was the last reader and it is not the current pd-buffer - if these two conditions are met, the buffer is deallocated. (Checking it is NOT the current PDB guarantees there is already a new PDB in the circuit so readers won’t starve).

This mechanism guarantees the information is always updated, but no message is corrupted. The ideal pool size, thus, is simply the number of tasks + 1.

8. Pipes

Pipe Control Block
Circular Buffer
Write pointer
Read pointer
Event "room"
Event "data"
Timeout

Pipes transmit streams of bytes - unlike queues or mailboxes that transmit 'objects' with a format. They are fast because data does not move, what moves is the read/write pointer within the pipe buffer. A writer stops its operation when the N specified bytes have been written, or there is no more room. In the last case it goes to sleep until a reader wakes it up after 'extracting' from the pipe.

A reader reads until N specified bytes were read or when there is no more data. In the last case it will sleep, to be waken by a writer.

Both one-to-many and many-to-one channels are possible - those who wake need to themselves ensure precedence.

As on V0.3.1, they depend on sleep-wake mechanism.

You cannot usem them within ISRs, in no occasion - if in need, a simple ring buffer is a replacement.

Usage Patterns

In this section some simple usage patterns are being attached. The board used to run these snippets is a Nucleo-F103RB (ARM Cortex-M3 based).

1. Exchange with rendez-vous

Usage: synchronise through message-passing

```
/* simple exchange using semaphores and a variable */
K_SEMA sendSema;
K_SEMA recvSema;
UINT32 mail;

VOID kApplicationInit(VOID)
```

```
{
    kSemaInit(&sendSema, 0);
    kSemaInit(&recvSema, 0);
    mail = 4;
}

VOID RecvTask(VOID)
{
    UINT32 mailCpy;

    while (1)
    {
        kSemaWait(&sendSema, K_WAIT_FOREVER);
        mailCpy = mail;
        kprintf("Received mail: %lu \n\r", mailCpy);
        kSemaSignal(&recvSema);
    }
}

VOID SendTask(VOID)
{
    while (1)
    {
        mail = 2*mail;
        kSemaSignal(&sendSema);
        kSemaWait(&recvSema, K_WAIT_FOREVER);

    }
}
```

2. Notify with a mailbox

Usage: coordinate a notified task action on a combination of events

```
/* @file application.c */

#include "application.h"

/* waiting 4 flags to be active before going */

INT stack1[STACKSIZE];
INT stack2[STACKSIZE];
INT stack3[STACKSIZE];

typedef enum
{
    TEMPERATURE = 1, HUMIDITY, CO2, FLOW
} UPDATE_t;
```

```
K_MBOX mbox;

/*** Init kernel objects here */
VOID kApplicationInit(VOID)
{
    kMboxInit(&mbox, EMPTY, NULL);
}

/* this task notifies which sensors had been updated */
VOID NotifyTask(VOID)
{
    UINT32 sendFlag = 0;
    UPDATE_t updateType = 0;

    while (1)
    {   /* simple sum to switch sensor type */

            updateType = (updateType + 1);
            if (updateType > 4)
            {
                updateType = 1;
            }
            switch (updateType)
            {
            case (TEMPERATURE):
                sendFlag = FLAG_TEMP_SENSOR_UPDATE;
                break;
            case (HUMIDITY):
                sendFlag = FLAG_HUM_SENSOR_UPDATE;
                break;
            case (CO2):
                sendFlag = FLAG_CO2_SENSOR_UPDATE;
                break;
            case (FLOW):
                sendFlag = FLAG_FLOW_SENSOR_UPDATE;
                break;
            default:
                break;
            }
            K_ERR err = kMboxSend(&mbox, (ADDR) &sendFlag, K_WAIT_FOREVER);
            kSleepUntil(2); /* every 10ms */
        }
}

/* this task stores events by OR'ing. when it matches the expected event record it
expects to take an action, the storage is reset.
alternatively it could take an action whenever a single flag was active */

VOID NotifiedTask(VOID)
{
    UINT32 *rcvdFlag = 0;
```

```
    TID senderPid = 0;
    UINT32 wantedFlags = FLAG_TEMP_SENSOR_UPDATE | FLAG_HUM_SENSOR_UPDATE |
    FLAG_CO2_SENSOR_UPDATE |
    FLAG_FLOW_SENSOR_UPDATE;
    UINT32 rcvdFlags = 0;
    while (1)
    {
         K_ERR err = kMboxRecv(&mbox, (ADDR) &rcvdFlag, &senderPid,
                 K_WAIT_FOREVER);
        if (err == 0)
        {

            if (senderPid == 1)
            {

                rcvdFlags |= *rcvdFlag;
                if (rcvdFlags == wantedFlags)
                {
/* the term 'synchronisation' is a bit loose here. the tasks are synch'ed
since they are running back-and-forth, but the NotifiedTask() will take
an specific action after being notified 4 different sensors had eventually been
updated. it is more like ''coordination'' */

                    rcvdFlags = 0; /* clear rcvd flags */
                    kprintf("Task synchronized\n\r");
                    /* do work */
                }
                else
                {
                    kprintf("Still missing a flag\n\r");
                    kBusyDelay(1);
                }
            }

        }
    }
}


VOID Task3(VOID)
{
    while (1)
    {
        kSleep(1);

    }

}
```

[mboxflags] | *images/mboxflags.png*

3. Multi-Client Server with mailboxes

Usage: Many-to-one command-response

```
#include "application.h"


#define MAX_PAYLOAD 36

K_MBOX serverMbox;
K_MBOX clientMbox1;
K_MBOX clientMbox2;

/* Application Protocol Data Unit */
typedef struct
{
    BYTE length; /* Length of the APDU payload */
    BYTE payload[MAX_PAYLOAD]; /* APDU payload */
    K_MBOX *replyMbox; /* Pointer to the client's reply mailbox */
} APDU;


void kApplicationInit(VOID)
{
    kMboxInit(&serverMbox, EMPTY, NULL);
    kMboxInit(&clientMbox1, EMPTY, NULL);
    kMboxInit(&clientMbox2, EMPTY, NULL);
}

/* Hello-server */
VOID Server(VOID)
{
    APDU *request, response;

    while (1)
    {
        /* Wait for a request */
        if (kMboxRecv(&serverMbox, &request, NULL, K_WAIT_FOREVER) == K_SUCCESS)
        {
            kprintf("Server received request: %s\n\r", request->payload);

            /* Process the request */
            response.length = snprintf((char*) response.payload,
                    sizeof(response.payload), "Response to: %s",
                    request->payload);

            /* Send the response back to the client's reply mailbox */
            if (kMboxSend(request->replyMbox, &response, K_WAIT_FOREVER) != K_SUCCESS)
            {
                kprintf("ACK fail\n\r");
```

```
            }
        }
    }
}

/* Hello-clients */
/
VOID Client1(VOID)
{
    APDU request, *response;

    while (1)
    {
        /* Prepare the request */
        snprintf((char*) request.payload, sizeof(request.payload),
                "Hello from Client 1");
        request.length = strlen((char*) request.payload);
        request.replyMbox = &clientMbox1; /* Specify the reply mailbox */

        /* Send the request to the server */
        if (kMboxSend(&serverMbox, &request, K_WAIT_FOREVER) == K_SUCCESS)
        {

            /* Wait for the response */
            if (kMboxRecv(&clientMbox1, (ADDR*)&response, NULL, K_WAIT_FOREVER)
                    == K_SUCCESS)
            {
                kprintf("C1 ACK'ed %s\n\r", response->payload);
            }
            else
            {
                kprintf("1F\n\r");
            }
        }
        else
        {
            kprintf("1F\n\r");
        }
    }
    kSleepUntil(10); /* every 50ms */
}

VOID Client2(VOID)
{
    APDU request, *response;

    while (1)
    {
        /* Prepare the request */
        snprintf((char*) request.payload, sizeof(request.payload),
                "Hello from Client 2");
```

```
        request.length = strlen((char*) request.payload);
        request.replyMbox = &clientMbox2; /* Specify the reply mailbox */

        /* Send the request to the server */
        if (kMboxSend(&serverMbox, &request, K_WAIT_FOREVER) == K_SUCCESS)
        {

            /* Wait for the response */
            if (kMboxRecv(&clientMbox2, (ADDR*)&response, NULL, K_WAIT_FOREVER)
                    == K_SUCCESS)
            {
                kprintf("C2 ACK'ed: %s\n\r", response->payload);
            }
            else
            {
                kprintf("2FAIL\n\r");
            }
        }
        else
        {
            kprintf("2FAIL\n\r");
        }

    }
    kSleep(10); /* 50ms */
}
```

[multiclient] | *images/multiclient.png*

4. Monitor-like: Synchronisation Barrier

Usage: resource access/tasks coordination

```
#include "application.h"

K_EVENT syncEvent; /* state event                   */
UINT32 syncCounter; /* state representation        */
K_MUTEX syncMutex; /* monitor lock                  */
K_MUTEX resourceLock; /* if there is a resource */
#define SYNC_CONDITION (syncCounter>=3) /* needed tasks in the barrier */

/* only one task can be active within a monitor
 they are enqueued either on the mutex or on the event
 */
static VOID synch(VOID)
{

    kMutexLock(&syncMutex, K_WAIT_FOREVER);
    kprintf("Task %d  entered monitor...\n\r", K_RUNNING_TID);
```

```
    syncCounter += 1;
    if (!(SYNC_CONDITION))
    {
        kMutexUnlock(&syncMutex);

        kEventSleep(&syncEvent, K_WAIT_FOREVER);
        kMutexLock(&resourceLock, K_WAIT_FOREVER);
        kprintf("Task %d is active in the monitor...\n\r", K_RUNNING_TID);
        kMutexUnlock(&resourceLock);
    }
    else
    {
        syncCounter = 0;
        kprintf("Task %d frees the waiting queue. \n\r", K_RUNNING_TID);
        kEventWake(&syncEvent);
        kMutexUnlock(&syncMutex);

    }
    kprintf("Task %d leaves monitor...\n\r", K_RUNNING_TID);
}

VOID kApplicationInit(VOID)
{
    kMutexInit(&syncMutex);
    kEventInit(&syncEvent);
    kMutexInit(&resourceLock);
    syncCounter = 0;
}

VOID Task1(VOID)
{

    while (1)
    {
        kSleep(5);
        synch();
    }
}

VOID Task2(VOID)
{
    while (1)
    {
        kSleep(8);
        synch();
    }
}

VOID Task3(VOID)
{
    while (1)
```

```
    {
        kSleep(3);
        synch();
    }

}
```

[syncbarr] | *images/syncbarr.png*

5. Queueing Pattern

Usage: Asynch (like) comm, serialising access, logging, monitoring

```
#include "application.h"
#include <stdlib.h>
#include <stdio.h>

INT stack1[STACKSIZE];
INT stack2[STACKSIZE];

/* sensor types */
typedef enum
{
    TEMPERATURE = 0, HUMIDITY, CO2, FLOW
} SensorType_t;

/* message sent through the queue */
typedef struct sensorMsg
{
    SensorType_t sensorType;
    INT32 sensorValue;
} Mesg_t;

#define N_MESSAGE 10
#define MESSAGE_SIZE sizeof(Mesg_t)

K_MESGQ mesgQueue;

BYTE queueBuffer[N_MESSAGE * MESSAGE_SIZE];

VOID kApplicationInit(VOID)
{
    kMesgQInit(&mesgQueue, (ADDR) queueBuffer, MESSAGE_SIZE, N_MESSAGE);
}

VOID SensorRead(VOID)
{
    SensorType_t sensorType = 0;
    Mesg_t msg;
```

```
    while (1)
    {
        sensorType = rand() % 4;
        switch (sensorType)
        {
        case (TEMPERATURE):

            msg.sensorValue = rand() % 50;
            msg.sensorType = TEMPERATURE;
            break;
        case (HUMIDITY):
            msg.sensorValue = rand() % 100;
            msg.sensorType = HUMIDITY;
            break;
        case (CO2):
            msg.sensorValue = rand() % 1000;
            msg.sensorType = CO2;
            break;
        case (FLOW):
            msg.sensorValue = rand() % 10;
            msg.sensorType = FLOW;
            break;
        default:
            break;
        }
        kMesgQSend(&mesgQueue, &msg, K_WAIT_FOREVER);
        kSleepUntil(2); /* every 10ms */
    }
}

VOID SensorLogging(VOID)
{
    Mesg_t recvMesg;
    while (1)
    {
        kMesgQRecv(&mesgQueue, &recvMesg, K_WAIT_FOREVER);
        {
        /* pretend printf is the logging method */
            if (recvMesg.sensorType == TEMPERATURE)
                kprintf("Temperature Sensor update: %lu C\n\r",
                        recvMesg.sensorValue);
            if (recvMesg.sensorType == HUMIDITY)
                kprintf("Humidity Sensor update: %lu%% \n\r",
                        recvMesg.sensorValue);
            if (recvMesg.sensorType == CO2)
                kprintf("CO2 Sensor update: %lu ppm\n\r", recvMesg.sensorValue);
            if (recvMesg.sensorType == FLOW)
                kprintf("FLOW Sensor update: %lu liters/min\n\r",
recvMesg.sensorValue);

        }
```

```
    }
}


/* why enqueue and not log when reading? to not loose track, events are buffered and
log happens on another task */
```

[queuing] | *images/queuing.png*

6. Event Flags/Event Groups

Usage: multi-condition broadcast, notification, observer pattern

Here is one of the many ways to implement multi-condition synchronisation, so-called event groups - with set/wait (or get)/clear. This approach use sleep/wake associated to an integer.

```
#include "application.h"
#include <stdlib.h>
#include <stdio.h>

INT stack1[STACKSIZE];
INT stack2[STACKSIZE];
INT stack3[STACKSIZE];
INT stack4[STACKSIZE];

#define MAX_FLAGS 32
#define FLAG_1 (1U << 0) /* subject 1 */
#define FLAG_2 (1U << 1) /* subject 2 */
#define FLAG_3 (1U << 2) /* subject 3 */

K_EVENT eventGroup;
K_MUTEX flagMutex; /* lock for protecting the flag variable */
static volatile UINT32 eventFlags = 0U; /*  event state representation */

VOID kApplicationInit(VOID)
{
    kEventInit(&eventGroup);
    kMutexInit(&flagMutex);
}

/* set flags (publish/notify) */
VOID setEventFlags(uint32_t flagMask)
{
    kMutexLock(&flagMutex, K_WAIT_FOREVER);
    eventFlags |= flagMask;
    kMutexUnlock(&flagMutex);

    kEventWake(&eventGroup);
}
```

```
/* subscribe/observe */
VOID waitForEventFlags(UINT32 flagMask, BOOL all) /* ALL or ANY */
{
        /*go to sleep */
        SLEEP:
        kEventSleep(&eventGroup, K_WAIT_FOREVER);

        /* a taks wakes here  */
        /* CR: either use a mutex or a disable all irqs (preferable) */
        kMutexLock(&flagMutex, K_WAIT_FOREVER);
        UINT32 currentFlags = eventFlags; /* temp copy */
        /* check */
        if ((all && ((currentFlags & flagMask) == flagMask)) || /* all flags */
        (!all && (currentFlags & flagMask))) /* any flags */
        {
            kMutexUnlock(&flagMutex);
            return; /* good to go */
        }
        kMutexUnlock(&flagMutex); /* nope, get back sleeping */
        goto SLEEP;
}

VOID clearEventFlags(UINT32 flagMask)
{
    kMutexLock(&flagMutex, K_WAIT_FOREVER);
    eventFlags &= ~flagMask;
    kMutexUnlock(&flagMutex);
}

VOID Task1(VOID) /* the setter, highest priority task  */
{
    while (1)
    {
        kSleep(1);
        setEventFlags(FLAG_1);
        setEventFlags(FLAG_2);
        kSleep(20);
        setEventFlags(FLAG_3);
    }
}

VOID Task2(VOID) /* waits for all flags */
{

    while (1)
    {
        waitForEventFlags(FLAG_1 | FLAG_2 | FLAG_3, TRUE); /* all flags */
        /* callback(); */
        kprintf("T2: got all flags \n\r");
        clearEventFlags(FLAG_1 | FLAG_2 | FLAG_3); /*clear*/
```

```
    }
}

VOID Task3(VOID) /* waits for any flags */
{
    while (1)
    {
        waitForEventFlags(FLAG_1 | FLAG_2, FALSE);
        /* callback(FLAG_1); or callback(FLAG_2); */
        kprintf("T3: At least one flag is set!\n\r");
        clearEventFlags(FLAG_1 | FLAG_2);
    }
}


VOID Task4(VOID) /* just want flag 3*/
{
    while (1)
    {
        waitForEventFlags(FLAG_3, TRUE);
        /* callback(); */
        kprintf("T4: FLAG_3 is up!\n\r");
        clearEventFlags(FLAG_3);
    }

}
```

[eventflags] | *images/eventflags.png*

Fueled by the craft...

The K0 design and implementation you see is the by-product of an (ongoing) research on operating systems design - either real-time or general-purpose.

I feel the need to declare some systems from where I borrowed ideas, for the sake of intellectual honesty, ethical technical conduct and peer recognition.

Shall I list the systems that inspired K0 design they would boil down to: uCOS, (former) ThreadX and 4.4BSD. SVR4 and its original UNIX traits on the GPOS realm, and the niche research hard real-time HARTIK are also worthy mentioning. Some details:

- From (pre-Azure) ThreadX and uCOS I borrowed insights for a highly-modular but low indirection design approach. I consider both present best-in-class embedded C code - which I still couldn't achieve.
- The doubly-list ADT is inspired on Linux 2.6.x.
- The scheduler data structures is also inspired on 4.4BSD. The scheduler bitmap computation borrows from uCOS the idea of counting lead zeros as a possible scheduling algorithm implementation..

- The delta timer approach is inspired on 4.4BSD.
- The priority propagation on turnstile synchronisation is inspired on 4.4BSD.
- The mailbox idea resembles the one found in uCOS/II.
- The memory allocator design is very similar to uCOS/II, in the sense it implements linked lists using arrays, an old-school approach to diminish the impact of meta-data.
- The initial user tasks stack assemble using the register numbers for easy debugging is borrowed from uCOS/II.
- Pipes design is inspired on the original UNIX named Pipes, or FIFOs.
- Sleep-Wake Events as 'pure signals' are also inspired on the original UNIX.
- HARTIK provided the idea of Cyclical Asynchronous Buffers and indirectly, several lessons on real-time systems by the writings of its main author, Buttazzo.

---

tony@kernel0.org