ARMv7M Kochab RTOS User Manual

Manual Version: 0.0.1

Software Version: 0.0.2

Revision History

Version	Date	Description.
0.0.2	13 August 2013	Initial document release for partner evaluation.

Concepts

This chapter introduces the concepts that form part of the Kochab RTOS variant. The RTOS provides a core, as well as a set of services that build upon the core. Services are optional, and their functionality could be duplicated by application code if necessary. This document currently describes the core API.

The concepts defined in this chapter should be familiar to readers who have experience in other real-time operating system. Although most real-time operating systems provides a similar set of concepts and primitives the details for each often differ in significant aspects so the chapter is recommended for all readers.

Core

This section describes the core concepts of the RTOS. The RTOS core provides a set of data structures, algorithms and APIs that have been carefully selected to work together. Except as specifically noted it is not generally possible to change the core concepts.

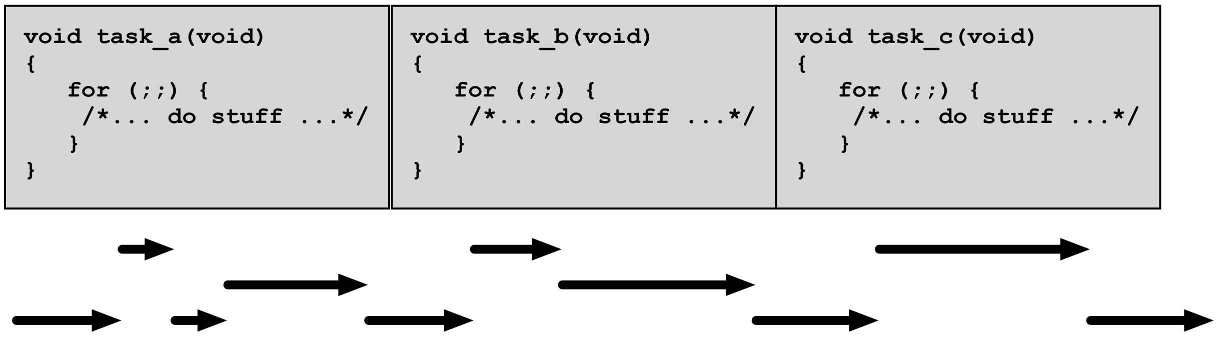
Tasks

The microcontroller is at its heart a device which executes an infinite stream[1] of instructions that modifies internal state and controls external devices. The challenge for an embedded programmer is work out which instructions should be executed to obtain the desired behaviour. For systems with simple requirements this can be easily achieved with a single infinite loop, however as the inherent complexity of requirements increase a single infinite loop becomes too complicated to effectively develop, reason about or debug.

```
void main(void)
{
   for (;;) {
    /*... do stuff ...*/
   }
}
```

[1] Strictly speaking the stream of instruction is finite due to physical limitations.

One approach for dealing with this complexity is to provide an abstraction multiple infinite streams of instructions which multiplex on to the single underlying stream. This allows the system designer to describe the system as multiple infinite loops, each of which is simple enough to effectively develop, reason about and debug. The ***task*** is the RTOS concept for this abstraction.



```
void task_a(void)
{
   for (;;) {
    /*... do stuff ...*/
   }
}
```

```
void task_b(void)
{
   for (;;) {
    /*... do stuff ...*/
   }
}
```

```
void task_c(void)
{
   for (;;) {
    /*... do stuff ...*/
   }
}
```

Each task in the system is defined to have an ***entry point***. The entry point is the location in program RAM where the task's execution starts from. In general this will be a function with a type signature `void fn (void)`.

Each task in the system has a defined ***stack***. The stack is primarily used by the code executing in the task for holding variables and return addresses during function calls. It is additionally used by the RTOS to save registers during a task switch. The size of each stack is chosen by the application programmer, and should be tailored to each task. Each task may have a different stack size.

The task that is currently executing on the microcontroller is known as the ***active task***. To implement multiplexing the RTOS must provide a mechanism for changing the currently active task. This mechanism is called a ***task context switch***. The tasks's ***context*** refers to all the state associated with the task but for which the underlying hardware can only support one copy at a time. Specifically, the processor only supports a single program counter, stack[2] and register state. During a context switch the RTOS will save the active tasks state (primarily on the stack) and restore the state from the new task so that it may become active.[3]

[2] Switching the stack only requires changing the stack pointer register as stacks are defined in software and not fixed by the hardware.

[3] The context switch operation only switches core registers, customisations can be made to support switching other state such as stack protection registers.

A task switch may occur when the currently active task explicitly releases the processor through a ***yield*** operation (see also `yield` API). A task may also be preempted must explicitly yield it can never be preempted by another task in the system (although it can be preempted by exception handlers). When a task performs a yield operation the RTOS is responsible for choosing which task becomes active.[4] The chosen task must be runnable. In the case where the active task is the only runnable task then a yield operation does not change the currently active task. When more than one other task is runnable then the scheduler will choose the next task to become active.

There are often times when a task has no useful work to do. For example, it may be waiting for input from a device. Rather than forcing the the task to constantly yield it can instead ***wait***. A task can therefore be described as either ***waiting*** or ***runnable***. A waiting task will never become the active task (i.e: the active task is always runnable). A waiting task is sometimes described as being ***blocked*** or ***blocking***. The signal concept further describes the mechanism by which a task can become waiting or runnable. When a task becomes blocked the RTOS will choose a new task to be active in a similar process as described for the yield operation. There is however one key difference, in this case it is possible to have the case where there are zero runnable tasks. In this case no task can become active, so the RTOS enters an ***idle*** mode. When in idle mode the RTOS waits until a task becomes runnable[5]. Depending on the configuration of the RTOS idle mode may place the system in a low-power mode.

Tasks are reference by a `TaskId`. `TaskId`s are integers in the range **0** thru **n-1** where **n** is the total number of tasks in the system. Each task has a unique `TaskId`. Depending on the number of tasks in the system the TaskId will either be an 8-, 16- or 32-bit integer.

[4] While this may appear obvious there is an alternate design where the yield task can directly choose the next task itself.

[5] This can only happen due to an exception handler executing. The exact mechanism is described in the section on interrupt events.

Scheduler.

The scheduler is an important sub-component of the RTOS. The scheduler provides an algorithm for choosing which of the runnable tasks should be chosen to become active. The algorithm used in this RTOS variant is a ***strict-priority with inheritance*** algorithm.[6]

Each task in the system is assigned a priority (which is simply a positive, integral value). Priorities must be unique, that is, no two tasks may have the same priority.

The general rule of the scheduler is to pick the task with the highest effective priority from the set of runnable tasks.

Normally, a task's effective priority is the priority it has been explicitly assigned, however in some cases a task may be assigned a different priority based on the priority inheritance protocol.

When a task in the system is not runnable (i.e.: it is blocked), it may be blocked waiting for a specific task. Alternatively, it may be blocked waiting on an external event or no specific task. To reduce the occurrence of priority inversion, the scheduler will perform priority inheritance in the case where a task is blocked on another specific task. A tasks effective priority is the maximum from the set of the tasks assigned priority and the effective priority of any tasks that are blocked on the task.

As an example, consider three tasks, A, B and C with priorities 20, 10, and 5. If task A is blocked-on task C, then C's effective priority will be 20, rather than 5. In this case, assuming C is runnable, it would be selected. It is important to note that this inheritance relationship is transitive, so if C blocked on a task D with priority 1, then D's effective priority would be 20.

Signals.

Signals provide a low-level mechanism for controlling when tasks are runnable and when the are blocked. A task can wait to receive a signal, and it will become runnable when another task (or interrupt) sends it the signal it is waiting for.

Each task has a set of signals. The size of the set is a system wide configuration option and can be 8, 16, or 32. The recommended size is 8. A specific signal is identified by its `SignalId`. A `SignalId` is only unique within the context of a specific tasks. A specific signal must be referred to by a tuple `<TaskId, SignalId>`. A group of signals is called a signal set and specified by the `SignalSet` type.

A task can ***wait*** on one or more signals[7] (see `signal_wait_set` API). If one or more of the requested signals has been sent to the task, one of the sent signals will be ***delivered*** to the task. If none of the requested signals have been sent to the task, then the task will block until one of the signals is delivered. The wait operation always acts as a yield point, even in the case where a requested signal is immediately available. Two optional operations are also available. A task can ***peek*** (see `signal_peek_set` API) to determine if a signal is available without delivering the signal. Additionally a task can ***poll*** (see `signal_poll_set` API) which will deliver a signal if available, but otherwise return immediately with an error code. Neither the peek or poll operation cause a yield to occur.

Signal delivery occurs when a users waits or polls for a signal. Although a user can request multiple signals, only one signal is delivered each time. In the case where multiple signals are available the signal with the lowest SignalId is delivered.

A task can **send** one or more signals to a specific task[8] (see `signal_send_set` API). Sending a signal does not cause a yield to occur, a task must explicitly yield after sending a signal if that is the desired behaviour. If a task has already has a specific signal pending then it is lost; there is no queuing of signals.

[7] Strictly a task can also wait on zero signals, however this will block a task permanently.

[8] A task is permitted to send signals to itself, although this is likely of limited utility.

Mutex.

Mutexes provide a low-level mechanism for implementing critical sections. A task can *lock* a mutex via the `mutex_lock` API. Once the `mutex_lock` function returns the task is the holder of the mutex. Any other tasks that attempt to obtain the lock will block until the current holder has relinquished the lock via the `mutex_unlock` API. Only the current holder may unlock the mutex.

When a task is blocked waiting for a lock it will be considered to be *blocked-on* the current holder of the mutex. In this manner the holder will inherit the priority of any tasks waiting for the lock. See the description of priority inheritance in the scheduler section for more information.

A task may attempt to acquire a lock without the possibility of blocking by using the `mutex_try_lock` API.

When multiple tasks are waiting to acquire a mutex, the scheduling algorithm will choose the appropriate task.

Semaphore.

Semaphores provide an alternative signalling mechanism. Conceptually a semaphore is an integral value with two operations: *post* and *wait*. Post is sometimes called V, signal, or up. Wait is sometimes called P or down. The post operation increments the underlying value, whereas the wait operation decrements the underlying value, if (and only if) the current value of the semaphore is greater than zero. The wait operation will block until the semaphore value becomes greater than zero.

Unlike a mutex, a semaphore has no concept of a *holder*. A consequence of this is that when a task waits on a semaphore, no task will inherit the waiter's priority.

The post operation is made available through the `sem_post` API. The wait operation is made available through the `sem_wait` API. Additionally, a `sem_try_wait` API is made available. This allows a task to attempt to decrement a semaphore without blocking.

Interrupt (Exception) Handlers.

Tasks in the system run in the standard mode. Applications can additionally register ***exception handlers*** that execute when a particular exception occurs. This is registration is done through RTOS configuration. Exception handlers will preempt any application tasks and the RTOS itself. Effectively exception handlers run at a higher priority than all tasks, and do not need to wait until a task explicitly yields. In return for these advantages there are certain important constraints that are placed upon exception handlers.

The exception handler must be finite, that is the function must return. As the exception handler executes at a higher priority than the rest of the system, the system is effectively blocked until the exception handler finishes processing.

Exception handlers execute on the stack of the currently active task. For this reason the exception handler should limit stack usage. As the exception handler may interrupt any task, each task must have enough stack for both its own usage and that of the largest exception handler.

As the system may be interrupted at any time it is important that the exception handler is careful with what data structures it uses, as the interrupted code may be in the process of updating the data structure. For this reason exception handlers are not generally able to call any RTOS APIs. There is of course the need for exception handler to notify tasks when certain events occur. The RTOS provides a very specialised interrupt event mechanism (see next section) to allow an exception handler to send a signal to a task.

The RTOS support two types of exception handlers, handlers that can request task preemption, and handlers than never cause a task preemption. An exception handler that never causes task preemption will not return any value. On completion of this exception handler execution will continue with the task that was interrupted by the exception handler.

In contrast, an exception handler that may cause preemption must return a boolean value. When this return value is false, the currently executing task will resume. However, when the return value is true, the RTOS will interrupt the currently executing task and run the scheduler. The scheduler will then choose the task with the highest effective priority, which may or may not be the currently executing task.

Note, the only reason why the currently executing task would not have the highest effective priority is if the exception handler caused a task with a higher effective priority to become runnable via an interrupt event.

Interrupt Events.

Interrupt events provide the bridge between the other distinct task and exception handler world. The system can be configured with a number of ***interrupt events***. Each interrupt event is associated with a `TaskId` and `SignalSet`.

An exception handler is able to ***raise*** an interrupt event (See `irq_event_raise` API). This is done in an atomic manner so it does not cause any data race problems. When an interrupt event is raised it indirectly causes the associated `SignalSet` to be sent to the associated task. To ensure the integrity of data structures the `SignalSet` is not sent to the task until the next yield point. When a yield occurs any interrupt events are processed before consulting the scheduler, in this manner if any interrupt event causes a task to become runnable that task will be eligible for selection at the next yield point.

Startup.

The RTOS `start` API should be called form the system's main function. This API ensures that all data structures are correctly initialised. After this occurs all tasks are automatically started by the RTOS.

API Reference.

Illegal API Usage.

Some APIs describe API usages that are illegal. Users of the API are responsible for ensuring that the API is used correctly. The specific behaviour that occurs when an API is used incorrectly is undefined. Illegal usage may cause a task crash, a software reset, or it may *appear* to work. Some illegal usages may be detected by static analysis tools.

APIs.

The APIs described include both function and function-like macros. To ensure performance some APIs are implemented as macros. Unfortunately the primary compiler does not support static inline functions, so macros must be used. This means that the compiler will not necessarily be performing the optimum level of type-checking to ensure program correctness. The types described in the API must be adhered to even in the case where the API is implemented as a macro.

Types.

`TaskId`

A `TaskId` is used to refer to a specific task. The underlying type is an unsigned integer of a size large enough to represent all task IDs. The `TaskId` should generally be treated an an opaque value. Arithmetic operations should not be used on a `TaskId`. For specialised purposes the programmer can assume that `TaskId` are consecutive integers from `TASK_ID_ZERO` thru `TASK_ID_MAX`. For example if a per task data structure is required it is valid to use a fixed size array and index the array by using the `TaskId`. For iterating over such an array it is valid use to increment a `TaskId`, however care must be taken to ensure the resulting value is in range. `TaskId` can be tested for equality, however other logical operations (e.g: comparison) should not be used.

`SignalId`

A `SignalId` refers to a signal within the context of a specific task. The underlying type is an 8-bit unsigned integer. The `SignalId` should be treated as an opaque value and should not be used in arithmetic or logical operations other than direct equality comparisons.

`SignalIdOption`

A `SignalIdOption` type contains all values as described for `SignalId`, but has an additional value called `SIGNAL_ID_NONE`. In all other respects this type is identical to the `SignalId` type. This type is used only as a return value for the `signal_poll_set` API. A `SignalIdOption` value can be compared for equality against a `SignalId` value, otherwise it can not be used in place of a `SignalId`. If a `SignalIdOption` has a value other than the `SIGNAL_ID_NONE` value, then it can be legally cast to a `SignalId`.

`SignalSet`

A `SignalSet` is used to refer to multiple `SignalId`-s at the same time. The underlying type of a `SignalSet` is an 8-, 16- or 32-bit unsigned integer large enough to represent all possible signals.

Constant defines.

The following pre-processor macro definitions are made available as constants. Ideally these would be made available as typed *static const* variables, however the compiler does not provide optimal code generation in that case, so pre-processor *#define* is used instead.

`TASK_ID_ZERO`

The constant has the type `TaskId`. This value represents the task ID for the zeroth task in the system.

`TASK_ID_MAX`

The constant has the type `TaskId`. This value represent the maximum task ID in the system.

`SIGNAL_ID_NONE`

The constant has the type `SignalIdOption`. This value can be compared against the return value of the `signal_poll_set` API.

Function APIs.

start

```
void start(void);
```

The `start` initialises the RTOS, and jumps to the first task in the scheduler. This function should be called from the system's main function. This function does not return.

yield

```
void yield(void);
```

This function will yield to another runnable task. Each task must yield frequently to ensure all tasks have a share of execution. When yielding any raised interrupt events will be processed. In the case where the current task is the only runnable task, the caller can rely on the operation being fast.

signal_wait_set

```
SignalId signal_wait_set(SignalSet sigset);
```

The `signal_wait_set` API allows a task to simultaneously wait for one or more signals. The task will block until one of the requested signals is available. Only one signal will be delivered for each call. The actual signal delivered is returned. If a signal is available immediately (without needing to block) this API implies a yield operation.

signal_poll_set

```
SignalIdOption signal_poll_set(SignalSet sigset);
```

The `signal_poll_set` API work in a similar manner to the `signal_wait_set` API, however instead of blocking if a signal is unavailable the API will return immediately. `SIGNAL_ID_NONE` will be returned if no signal is delivered. The `signal_poll` APIs do not imply a yield operation, even in the case where a signal is delivered.

signal_peek_set

```
bool signal_peek_set(SignalSet sigset);
```

The `signal_peek_set` API can be used to determine if a signal is available for delivery. In all cases true is returned if the requested signal could be delivered, false otherwise. If `signal_peek_set` is called before a call to `signal_wait_set` with the same arguments and returns `true`, then `signal_wait_set` is guaranteed not to block. If `signal_peek_set` is called before a call to `signal_poll_set` with the same argument and returns `true`, then `signal_poll_set` is guaranteed to return a valid `SignalId`. This function does not imply a yield operation.

signal_send_set

```
void signal_send_set(TaskId task, SignalSet sigset);
```

The `signal_send`_set API makes the specified signal set available on the specific task. If the signal was already available on the specified task, then this operation does not change the state of the system. The API does not imply a yield. The caller must explicitly yield if that is the intended behaviour.

mutex_lock

```
void mutex_lock(MutexId mutex);
```

The `mutex_lock` API makes the specified signal set available on the specific task. If the signal was already available on the specified task, then this operation does not change the state of the system. The API does not imply a yield. The caller must explicitly yield if that is the intended behaviour.

mutex_try_lock

```
bool mutex_try_lock(MutexId mutex);
```

The `mutex_lock` API makes the specified signal set available on the specific task. If the signal was already available on the specified task, then this operation does not change the state of the system. The API does not imply a yield. The caller must explicitly yield if that is the intended behaviour.

mutex_unlock

```
void mutex_unlock(MutexId mutex);
```

The `mutex_lock` API makes the specified signal set available on the specific task. If the signal was already available on the specified task, then this operation does not change the state of the system. The API does not imply a yield. The caller must explicitly yield if that is the intended behaviour.

sem_wait

```
void sem_wait(SemId sem);
```

The `mutex_lock` API makes the specified signal set available on the specific task. If the signal was already available on the specified task, then this operation does not change the state of the system. The API does not imply a yield. The caller must explicitly yield if that is the intended behaviour.

sem_try_wait

```
bool sem_try_wait(SemId sem);
```

The `mutex_lock` API makes the specified signal set available on the specific task. If the signal was already available on the specified task, then this operation does not change the state of the system. The API does not imply a yield. The caller must explicitly yield if that is the intended behaviour.

sem_post

```
void sem_post(SemId sem);
```

The `mutex_lock` API makes the specified signal set available on the specific task. If the signal was already available on the specified task, then this operation does not change the state of the system. The API does not imply a yield. The caller must explicitly yield if that is the intended behaviour.

irq_event_raise

```
void irq_event_raise(IrqEventId);
```

The `irq_event_raise` API provides the interface that enables an exception handler to raise an interrupt event. While it is possible for a task to call this API, it is expected to only be called from an exception handler. Tasks can directly call `signal_send_set` API so do not need to indirect through the interrupt event.