Quick user manual

J. Camilo Gómez C.

January 16, 2020

Contents

1 Background

1.1 About the RTOS

QuarkTS its a simple non-preemptive real-time OS with a quasi-static scheduler for embedded multitasking applications. Its simplified kernel uses a cooperative round-robin scheme with a linked chain approach and a queue to provide true FIFO priority-scheduling.

The most significant feature of the OS is that tasks are NOT preemptable. Once a task starts its execution, it owns the CPU at a base level (non-interrupt) until it returns. In other words, except when the CPU is servicing an event at interrupt-level, the CPU will be executing the task until returns, then, the control back to the task-scheduler. This has, however, significant benefits: you don't need to worry about concurrency inside the callback method since only one callback method runs at a time and could not be interrupted. All resources are yours for that period of time, no one can switch the value of variables (except interrupt functions of course...). It is a stable and predictable environment and it helps a lot with writing stable code.

Apart from these basic capabilities, it has a small memory footprint that makes it suitable for tiny embedded μControlled systems.

1.1.1 Hardware compatibility

QuarkTS has no direct hardware dependencies, so it is portable to many platforms and C compilers. The following cores have been powered with QuarkTS successfully:

- ARM cores(ATMEL, STM32, LPC, Kinetis, Nordic and others)
- 8Bit AVR, 8051, STM8
- HCS12, ColdFire, MSP430
- PIC (PIC24, dsPIC, 32MX, 32MZ)

1.1.2 Coding standard and naming convention

- All the QuarkTS implementation follows the C99 standard.
- The core source files try to conform most of the MISRA C 2004/2012 coding standard guidelines, allowing the OS to be used in safe-critical embedded systems.
- In line with MISRA guides, unqualified standard `char` and `char *` types are only permitted to hold ASCII characters and strings respectively.
- In line with MISRA guides and for portability between platforms, we use the `stdint.h` with typedefs that indicate size and signedness in place of the basic types.
- The `_t` suffix its used to denote a type definition (i.e `qBool_t, qTask_t, size_t, ...`).
- Functions, macros, `enum` values and data-types are prefixed `q`. (i.e. `qFunction`, `qEnumValue`, `QCONSTANT`, `qType_t`, ...)

- Other than the pre-fix, most macros used for constants are written in all upper case.
- Almost all functions returns a boolean value of type `qBool_t`, where a `qTrue - 1u` value indicates a successful procedure and `qFalse - 0u`, the failure of the procedure.

1.1.3 Memory usage

As a quasi-static scheduler is implemented here, dynamic memory allocation is not required and the assignment of tasks must be done before program execution begins. The kernel is designed to allow unlimited tasks and kernel objects (STimers, FSMs, queues, etc), but of course, the whole application will be constrained by the memory specifications of the embedded system.
The kernel's memory footprint can be scaled down to contain only the features required for your application, typically 2.5 KBytes of code space and less than 1 KByte of data space.

OS Memory Footprint (Measured in a 32bit MCU)	
Functionality	**Size**(bytes)
Kernel, scheduler and task management	2637
A task node (`qTask_t`)	64
Finite State-Machines(FSM) handling and related APIs	314
A FSM object (`qSM_t`)	37
STimers handling and related APIs	258
A STimer object (`qSTimer_t`)	8
qQueues handling and related APIS	544
A qQueue object (`qQueue_t`)	28
Memory management	407
A memory pool (`qMemoryPool_t`)	28
The AT Command Parser	1724
An AT-Parser instance (`qATParser_t`)	112
An AT command object (`qATCommand_t`)	24
Remaining utilities	2980

Although the kernel does not use dynamically-allocated resources internally, the application writer can create an object dynamically using the heap implementation provided by the memory management module described in section 3.5.

1.2 Timing approach

The kernel implements a Time-Triggered Architecture (TTA)[1], in which the tasks are triggered by comparing the corresponding task-time with a reference clock. The reference clock must be real-time and follow a monotonic behavior. Usually, all embedded systems can provide this kind of reference with a constant tick generated by a periodic background hardware-timer, typically, at 1Khz (1mS tick).

For this, the kernel allows you to select the reference clock source among these two scenarios:

- When tick already provided: The reference is supplied by the Hardware Abstraction Layer (HAL) of the device. It is the simplest scenario and it occurs when the framework or SDK of the embedded system includes a HAL-API that obtains the time elapsed since the system starts, usually in milliseconds and taking a 32-bit counter variable.
- When the tick is not provided: The application writer should use bare-metal code to configure the device and feed the reference clock manually. Here, a hardware timer should raise an interrupt periodically. After the *Interrupt Service Routine* (ISR) has been implemented using the platform-dependent code, the `qClock_SysTick()` API must be called inside. It is recommended that the reserved ISR should only be used by QuarkTS.

1.3 Setting up the kernel `qSchedulerSetup`

This function should be the first call to the OS APIs. `qSchedulerSetup()` prepares the kernel instance, sets the reference clock, defines the *Idle-Task* callback and allocates the stack for the internal queue.

```
void qSchedulerSetup(qGetTickFcn_t TickProvider,
                     qTime_t ISRTick,
                     qTaskFcn_t IDLE_Callback)
```

Parameters

- `TickProvider` : The function that provides the tick value. If the user application uses the `qClock_SysTick()` from the ISR, this parameter can be `NULL`.
  Note: Function should take `void` and return a 32bit unsigned integer.
  This argument must have this prototype : `qUINT32_t TickProviderFcn(void)`
- `ISRTick` :This parameter specifies the ISR background timer period in seconds(floating-point format).

  Note : This argument will be only available if `Q_SETUP_TIME_CANONICAL` is set to zero(0).
- `IDLE_Callback` : Callback function for the idle task. If not used, pass `NULL` as argument.

> This call is mandatory and must be called once in the application main thread before any kind of interaction with the OS.

Usage example:

Scenario 1: When tick is already provided

```
#include "QuarkTS.h"
#include "HAL.h"

#define TIMER_TICK   0.001   /* 1ms */

void main(void){
    HAL_Init();
    qSchedulerSetup(HAL_GetTick, TIMER_TICK, IdleTask_Callback, 10);

    /*
    TODO: add Tasks to the scheduler scheme and run the scheduler
    */
}
```

Scenario 2: When the tick is not provided

```
#include "QuarkTS.h"
#include "DeviceHeader.h"

#define TIMER_TICK   0.001   /* 1ms */

void Interrupt_Timer0(void){
    qClock_SysTick();
}

void main(void){
    MCU_Init();
    Setup_Timer0();
    qSchedulerSetup(NULL, TIMER_TICK, IdleTask_Callback, 10);

    /*
    TODO: add Tasks to the scheduler scheme and run the scheduler
    */
}
```

1.4 Tasks

Like many operating systems, the basic unit of work is the task. Tasks can perform certain functions, which could require periodic or one-time execution, update of specific variables or waiting for specific events. Tasks also could be controlling specific hardware or be triggered by hardware interrupts. In the QuarkTS OS, a task is seen as a node concept that links together:

- Program code performing specific task activities (callback function)
- Execution interval (time)
- Number of execution (iterations)
- Event-based data

The OS uses a *Task Control Block* (TCB) to represent each task, storing essential information about task management and execution. Part of this information also includes *link-pointers* that allows it to be part of a list.

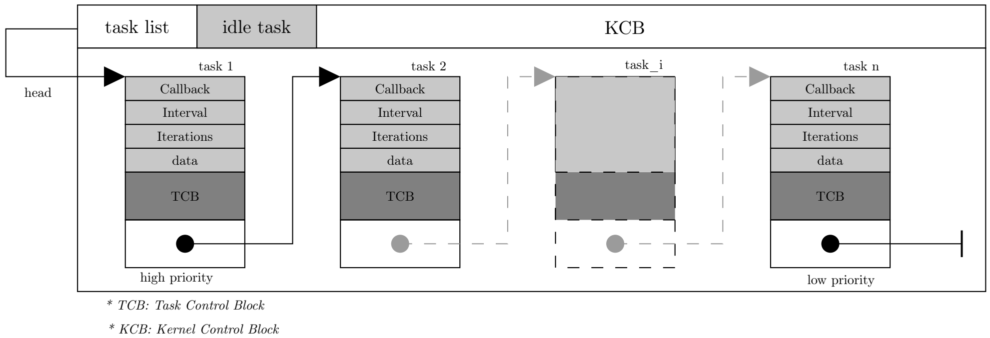

Figure 1: Tasks scheme

Each task performs its activities via a callback function and each of them is responsible for supporting cooperative multitasking by being "good neighbors", i.e., running their callback methods quickly in a non-blocking way and releasing control back to the scheduler as soon as possible (returning).

Every task node, must be defined using the `qTask_t` data-type and the callback is defined as a function that returns `void` and takes a `qEvent_t` data structure as its only parameter (This input argument can be used later to get event information, see section 2.3).

```
qTask_t UserTask;
void UserTask_Callback(qEvent_t eventdata){
    /*
    TODO : Task code
    */
}
```

Note: All tasks in QuarkTS must ensure their completion to return the CPU control back to the scheduler, otherwise, the scheduler will hold the execution-state for that task, preventing the activation of other tasks.

1.4.1 The idle task

Its a special task loaded by the OS scheduler when there is nothing else to do (no task in the whole scheme has reached the ready state). The idle task is already hard-coded into the kernel, ensuring that at least, one task is able to run. Additionally, the OS setup this task with the lowest possible priority to ensure that does not use any CPU time if there are higher priority application tasks able to run.
The idle task doesn't perform any active functions, but the user can decide if it should perform some activities defining a callback function for it. This could be done at the beginning of the kernel setup (as seen in section 1.3 with `qSchedulerSetup()`) or in run-time with `qSchedulerSetIdleTask()` .

```
void qSchedulerSetIdleTask(qTaskFcn_t Callback)
```

Of course, the callback must follow the same function prototype for tasks.

> To disable the idle-task activities, a `NULL` should be passed as argument.

1.4.2 Adding tasks to the scheme : `qSchedulerAdd`

After setting up the kernel with `qSchedulerSetup()`, the user can proceed to deploy the multitasking application by adding tasks. If the task node and their respective callback is already defined, the task can be added to the scheme using `qSchedulerAdd_Task()` . This API can schedule a task to run every `Time` seconds, `nExecutions` times and executing `CallbackFcn` method on every pass.

```
qBool_t qSchedulerAdd_Task(
                            qTask_t * const Task, qTaskFcn_t CallbackFcn,
                            qPriority_t Priority, qTime_t Time,
                            qIteration_t nExecutions, qState_t InitialState,
                            void* arg )
```

Parameters

- `Task` : A pointer to the task node.
- `CallbackFcn` : A pointer to a void callback method with a `qEvent_t` parameter as input argument..
- `Priority` : The priority value. [0(min) - `Q_PRIORITY_LEVELS`(max)]
- `Time` : Execution interval defined in seconds (floating-point format). For immediate execution use the `qTimeImmediate` definition.
- `nExecutions` : Number of task executions (Integer value). For indefinite execution use `qPeriodic` or the `qIndefinite` definition.
- `InitialState` : Specifies the initial operational state of the task (`qEnabled`, `qDisabled`, `qAsleep` or `qAwake`(implies `qEnabled`)).
- `arg` - Represents the task argument. This argument must be passed by reference and cast to `(void *)`.

Caveats

1. A task with `Time` argument defined in `qTimeImmediate`, will always get the *qReady* state in every scheduling cycle, as consequence, the idle task will never gets dispatched.

2. Tasks do not remember the number of iteration set initially by the `nExecutions` argument. After the iterations are done, the internal iteration counter decreases until reach the zero. If another set of iterations is needed, the user should set the number of iterations again and resume the task explicitly.

3. Tasks that performed all their iterations, put their own state to `qDisabled`. Asynchronous triggers do not affect the iteration counter.

4. The `arg` parameter can be used as storage pointer, so, for multiple data, create a structure with the required members and pass a pointer to that structure.

Invoking `qSchedulerAdd_Task()` is the most generic way to adding tasks to the scheme, supporting a mixture of time-triggered and event-triggered tasks, however, additional simplified API functions are also provided to add specific purpose tasks:

- Event-triggered only tasks → `qSchedulerAdd_EventTask()`
- State-machine tasks → `qSchedulerAdd_StateMachineTask()`. See section 3.2.4.
- AT-Command parser tasks → `qSchedulerAdd_ATParserTask()`. See section 3.4.7.

1.4.3 Event-triggered tasks

An event-triggered task reacts asynchronously to the occurrence of events in the system, such as external interrupts or changes in the available resources.

The API `qSchedulerAdd_EventTask()` is intended to add this kind of tasks, keeping it in a *qSuspended* state. Only asynchronous events followed by their priority value dictates when a task can change to the *qRunning* state.

```
qBool_t qSchedulerAdd_EventTask(
                    qTask_t * const Task, qTaskFcn_t CallbackFcn,
                    qPriority_t Priority, void* arg)
```

As seen above, arguments related to timing and iterations parameters are dispensed and the only required arguments become minimal, just needing: `CallbackFcn`, `Priority` and the related task arguments `arg`.

1.4.4 Removing a task : `qSchedulerRemoveTask`

As expected, the API removes the task from the scheduling scheme. This means the task node will be disconnected from the kernel chain, preventing additional overhead provided by the scheduler when it does checks over it and course, preventing from running.

```
qBool_t qSchedulerRemoveTask( qTask_t * const Task )
```

Caveats:

Task nodes are variables like any other. They allow your application code to reference a task, but there is no link back the other way and the kernel doesn't know anything about the variables, where the variable is allocated (stack, global, static, etc.) or how many copies of the variable you have made, or even if the variable still exists. So the `qSchedulerRemoveTask()` API cannot automatically free the resources allocated by the variable. If the task node has been dynamically allocated, the application writer it's responsible to free the memory block after a removal call.

1.5 Running the OS scheduler `qSchedulerRun`

After preparing the multitasking environment for your application, a call to `qSchedulerRun()` is required to execute the whole scheme. This function is responsible to run the following OS main components:

- **The Scheduler** : Select the tasks to be submitted into the system and decide with of them are able to run.
- **The Dispatcher** : When the scheduler completes its job of selecting ready tasks, it is the dispatcher which takes that task to the running state. This procedure gives a task control over the CPU after it has been selected by the scheduler. This involves the following:
  - Preparing the resources before the task execution
  - Execute the task activities (via the callback function)
  - Releasing the resources after the task execution

The states involved in the interaction between the scheduler and dispatcher are described in the section that follows.

> *Note*: After calling `qSchedulerRun()`, the OS scheduler will now be running, and the following line should never be reached, however, the user can optionally release it explicitly with `qSchedulerRelease()` API function.

1.5.1 Releasing the scheduler: `qSchedulerRelease`

This functionality must be enabled from the `Q_ALLOW_SCHEDULER_RELEASE` macro. This API stop the kernel scheduling. In consequence, the main thread will continue after the `qSchedulerRun()` call.

Although producing this action is not a typical desired behavior in any application, it can be used to handle a critical exception.

When used, the release will take place after the current scheduling cycle finish. The kernel can optionally include a release callback function that can be configured to get called if the scheduler is released. Defining the release callback, will help to take actions over the exception that caused the release action. To enable this functionality, the `qSchedulerSetReleaseCallback()` API should be used.

```
void qSchedulerSetReleaseCallback(qTaskFcn_t Callback)
```

When a scheduler release is performed, resources are not freed after this call. After released, the application can invoke the `qSchedulerRun()` again to resume the scheduling activities.

1.6 Global states and scheduling rules

A task can be in one of the four global states: *qRunning*, *qReady*, *qSuspended* or *qWaiting*. Each of these states is tracked implicitly by putting the task in one of the associated kernel lists.

These global states are described below:

- **qWaiting** : The task cannot run because the conditions for running are not in place. In other words, the task is waiting for the conditions for its execution to be met.
- **qReady** : The task has completed preparations for running, but cannot run because a task with a higher precedence is running.
- **qRunning** : The task is currently being executed.
- **qSuspended** : The task doesn't take part in what is going on. Normally this state is taken after the *qRunning* state or when the task doesn't reach the *qReady* state.

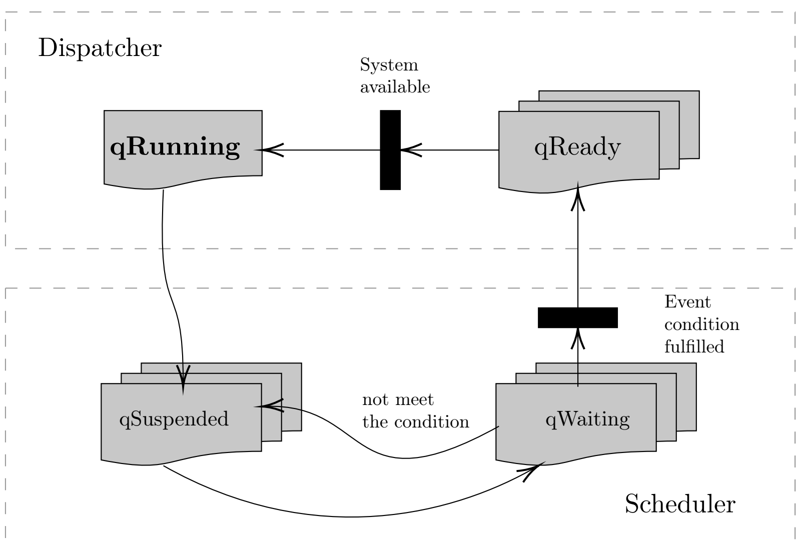

Figure 2: Task global states

The presence of a task in a particular list indicates the task's state. There are many ready-lists as defined in the `Q_PRIORITY_LEVELS` macro. To select the target ready list, the OS use the user-assigned priority between 0 (the lowest priority) and `Q_PRIORITY_LEVELS-1` (the highest priority). For instance, if `Q_PRIORITY_LEVELS` is set to 5, then QuarkTS will use 5 priority levels or ready lists: 0 (lowest priority), 1, 2, 3, and 4 (highest priority).

Except for the idle task, a task exists in one of these states. As the real-time embedded system runs, each task moves from one state to another(moving it from a list to another), according to the logic of a simple finite state machine (FSM). Figure 2 illustrates the typical flowchart used by QuarkTS to handle the task's states, with brief descriptions of state transitions, additionally you may also notice the interaction between the scheduler and the dispatcher.

The OS assumes that none of the tasks does a block anywhere during the *qRunning* state. Based on the *round-robin* fashion, each ready task runs in turn from every ready lists.

1.6.1 Rules

Task precedence is used as the task scheduling rule and precedence among tasks is determined based on the priority of each task. If there are multiple tasks able to run, the one with the highest precedence goes to *qRunning* state first. In determining precedence among tasks, of those tasks having different priority levels, that with the highest priority has the highest precedence. Among tasks having the same priority, the one that entered the scheduling scheme first has the highest precedence if the `Q_PRESERVE_TASK_ENTRY_ORDER` configuration is enabled, otherwise the OS will reserves for himself the order according to the dynamics of the kernel lists.

The scheduler also has an order of precedence for events, in this way, if events of different nature converge to a single task, these will be served according to the following flowchart:

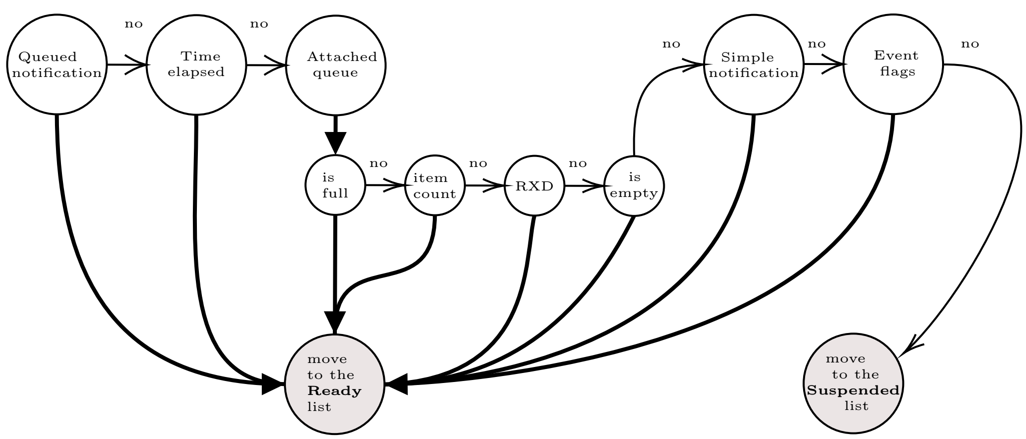

Figure 3: Event precedence

All the listed events are later detailed in section 2.

1.6.2 Additional operational states

Each task has independent operating states from those globally controlled by the scheduler. These states can be handled by the application writer to modify the event-flow to the task and consequently, affecting the task transition to the *qReady* global state. These operating states are described below.

- *qAwake* : In this state, the task is conceptually in an alert mode, handling most of the available events. This operational state is available when the `SHUTDOWN` bit is set, allowing the next operational states to be available:
  - *qEnabled* : The task is able to catch all the events. This operational state is available when the `ENABLE` bit is set.
  - *qDisabled* : In this state the time events will be discarded. This operational state is available when the `ENABLE` bit is cleared.
- *qAsleep* : Task operability is put into a deep doze mode, so the task can't be triggered by the lower precedence events. This operational state is available when the `SHUTDOWN` bit is cleared. The task can exit from this operational state when it receives a high precedence event (a queued notification) or using the `qTaskSetState()` API.

The figure 4 shows a better representation of how the event flow can be affected by this operational states.

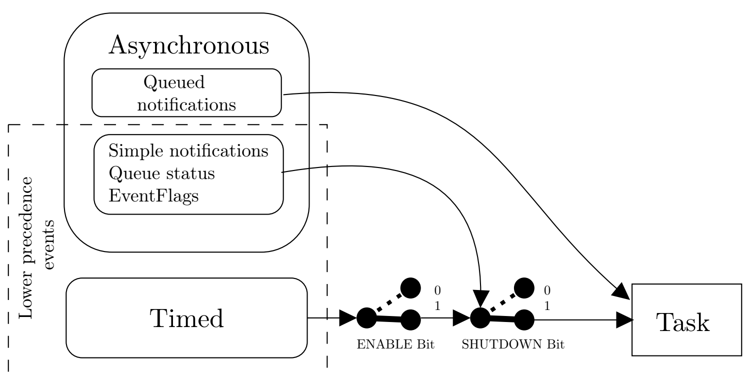

Figure 4: Event flow according operational states

> Queued notifications are the only event that can wake up sleeping tasks.

> The *qAsleep* operational state overrides the *qEnabled* and *qDisabled* State.

1.7 Getting started

Unpack the source files and copy them into your project folder. Then include the header file `QuarkTS.h` and setup the instance of the kernel using the `qSchedulerSetup()` inside the main thread. Additional configuration to the target compiler may be required to add the path to the directory of header files. The code below shows a common initialization in the main source file.

File: `main.c`

```
#include "QuarkTS.h"
#define TIMER_TICK   0.001   /* 1ms */

void main(void){
    /*start of hardware specific code*/
    HardwareSetup();
    Configure_Periodic_Timer_Interrupt_1ms();
    /*end of hardware specific code*/
    qSchedulerSetup(NULL, TIMER_TICK, IdleTask_Callback, 10);
    /*
    TODO: add Tasks to the scheduler scheme and run the scheduler
    */
}
```

In the above code, the following considerations should be taken

- The function `qSchedulerSetup()` must be called before any interaction with the scheduler.
- The procedure `HardwareSetup()` should be a function with all the hardware instructions needed to initialize the target system.
- The procedure `Configure_Periodic_Timer_Interrupt_1ms()` should be a function with all the hardware instructions needed to initialize and run a timer with an overflow tick of one millisecond.

Tasks can be later added to the scheduling scheme by simply calling `qSchedulerAdd_Task()` or any of the other available APIs for specific purpose tasks.

1.8 Recommended programming pattern

A multitasking design pattern demands a better project organization. To have this attribute in your solution, code the tasks in a separate source file. A simple implementation example is presented below.

To make the handling of the tasks available in other contexts, nodes should be globals(see the `extern` qualifier in `MyAppTasks.h` header file). Avoid implementing functionalities in a component that are not related to each other. Put any other components and globals resources in a separated source file, for example (`ScreenDriver.h`/`ScreenDriver.c`), (`Globals.h`/`Globals.c`). These simple design tips will allow you to have a better principle of abstraction and will maximize the cohesion of the solution, improving the code readability and maintenance.

File: `MyAppTasks.h`

```
#ifndef MYAPPTASK_H
    #define MYAPPTASK_H

    #include "Globals.h"
    #include "QuarkTS.h"

    extern qTask_t CommunicationTask, HardwareCheckTask,
                  CheckUserEventsTask, SignalAnalisysTask;

    void CommunicationTask_Callback(qEvent_t);
    void HardwareCheckTask_Callback(qEvent_t);
    void CheckUserEventsTask_Callback(qEvent_t);
    void SignalAnalisysTask_Callback(qEvent_t);}
#endif
```

File: `MyAppTasks.c`

```
#include "MyAppTasks.h"

qTask_t CommunicationTask, HardwareCheckTask,
        CheckUserEventsTask, SignalAnalisysTask;

void CommunicationTask_Callback(qEvent_t e){
    /*
    TODO: Communication Task code
    */
}

void HardwareCheckTask_Callback(qEvent_t e){
    /*
    TODO: Hardware Check Task code
    */
}

void CheckUserEventsTask_Callback(qEvent_t e){
    /*
    TODO: Check User Events Task code
    */
}

void SignalAnalisysTask_Callback(qEvent_t e){
    /*
    TODO: Signal Analisys Task code
    */
}

/*this task doesnt need an Identifier*/
void IdleTask_Callback(qEvent_t e){
    /*
    TODO: Idle Task code
    */
}
```

File: `main.c`

```
#include "QuarkTS.h"
#include "Globals.h"
#include "MyAppTasks.h"
```

```
void interrupt OnTimerInterrupt(){ //hardware specific code
    qClock_SysTick(); //
}

void main(void){
    /*start of hardware specific code*/
    HardwareSetup();
    Configure_Periodic_Timer_Interrupt_10ms();
    /*end of hardware specific code*/
    qSchedulerSetup(NULL, 0.01, IdleTask_Callback, 10);
    qSchedulerAdd_Task(HardwareCheckTask, HardwareCheckTask_Callback,
                       qLowest_Priority, 0.25, qPeriodic, qEnabled, NULL);
    qSchedulerAdd_Task(SignalAnalisysTask, SignalAnalisysTask_Callback,
                        qHigh_Priority, 0.1, 200, qEnabled, NULL);
    qSchedulerAdd_EventTask(CheckEventsTask, CheckEventsTask_Callback,
                        qMedium_Priority, NULL);
    qSchedulerAdd_Task(CommunicationTask, CommunicationTask_Callback,
                       qHigh_Priority, qTimeInmediate, qPeriodic,
                       qEnabled, NULL);
    qSchedulerRun();
    for(;;){}
}
```

1.9 Critical sections

Since the kernel is non-preemptive, the only critical section that must be handled are the shared resources accessed from the ISR context. Perhaps, the most obvious way of achieving mutual exclusion is to allow the kernel to disable interrupts before it enters its critical section and then, enable interrupts after it leaves its critical section.

By disabling interrupts, the CPU will be unable to change the current context. This guarantees that the currently running job can use a shared resource without another context accessing it. But, disabling interrupts, is a major undertaking. At best, the system will not be able to service interrupts for the time the current job is doing in its critical section, however, in QuarkTS, these critical sections are handled quickly as possible.

Considering that kernel is hardware-independent, the application writer should provide the necessary piece of code to enable and disable interrupts.

For this, the `qCritical_SetInterruptsED()` API should be used. In this way, communication between ISR and tasks using queued notifications or data queues becomes safely.

```
void qCritical_SetInterruptsED(void (*Restorer)(qUINT32_t),
                               qUINT32_t (*Disabler)(void));
```

Parameters:

- `Restorer` : The function with hardware specific code to enable or restore interrupts.
- `Disabler` : The function with hardware specific code that disables interrupts.

In some systems, disabling the global IRQ are not enough, as they don't save/restore state of interrupt, so here, the `qUINT32_t` argument and return value in both functions

(`Disabler` and `Restorer`) becomes relevant, because they can be used by the application writer to save and restore the current interrupt configuration. So, when a critical section is performed, the `Disabler`, in addition to disable the interrupts, return the current configuration to be retained by the kernel, later when the critical section finish, this retained value are passed to `Restorer` to bring back the saved configuration.

1.10 Task management APIs in run-time

Most of the scheduling parameters regarding task execution can be changed at run-time. The following APIs are intended for this purpose.

```
void qTaskSetTime( qTask_t * const Task, const qTime_t Value )
```

Set/Change the task execution interval.

Parameters:

- `Task` : A pointer to the task node.
- `Value` : Execution interval defined in seconds (floating-point format). For immediate execution use `qTimeImmediate`.

```
void qTaskSetIterations( qTask_t * const Task, qIteration_t Value)
```

Set/Change the number of task iterations.

Parameters:

- `Task` : A pointer to the task node.
- `Value` : Number of task executions (integer value). For indefinite execution user `qPeriodic` or `qIndefinite`. Tasks do not remember the number of iteration set initially.

```
void qTaskSetPriority( qTask_t * const Task, qPriority_t Value)
```

Set/Change the task priority value.

Parameters:

- `Task` : A pointer to the task node.
- `Value` : Priority value. `[0(min) - Q_PRIORITY_LEVELS(max)]`.

```
void qTaskSetCallback( qTask_t * const Task, qTaskFcn_t CallbackFcn )
```

Set/Change the task callback function. . Can be used to detach an state machine.

Parameters:

- `Task` : A pointer to the task node.
- `Callback` : A pointer to a void callback method with a `qEvent_t` parameter as input argument.

```
void qTaskSetState(qTask_t * const Task, const qState_t State )
void qTaskResume(qTask_t * const Task)
void qTaskSuspend(qTask_t * const Task)
void qTaskASleep(qTask_t * const Task)
void qTaskAwake(qTask_t * const Task)
```

Set the task operability state. .

Parameters:

- `Task` : A pointer to the task node.
- `State` : Use one of the following values:
  - `qEnabled` : Task will be able to catch all the events.
  - `qDisabled` : Time events will be discarded. The task can catch asynchronous events.
  - `qAsleep` : Put the task into a deep-doze operability mode. The task can't be triggered by the lower precedence events.
  - `qAwake` : Task will be able to catch all the events.

```
void qTaskSetData(qTask_t * const Task, void* UserData )
```

Set the task data.

Parameters:

- `Task` : A pointer to the task node.
- `UserData` : A pointer to the associated data.

```
void qTaskClearTimeElapsed( qTask_t * const Task )
```

Clear the elapsed time of the task. Restart the internal task tick.

Parameters:

- `Task` : A pointer to the task node.

```
qCycles_t qTaskGetCycles( const qTask_t * const Task )
```

Retrieve the number of task activation's.

Parameters:

- `Task` : A pointer to the task node.

Return value:

A `qUINT32_t` value containing the number of task activations.

```
qState_t qTaskGetState( const qTask_t * const Task)
```

Retrieve the task operational state.

Parameters:

- `Task` : A pointer to the task node.

Return value:

`qEnabled` or `qDisabled` if the task is `qAwaken`. `qAsleep` if the task is in a sleep operational state.

```
qTask_t* qTaskSelf(void)
```

Get current running task handle.

Return value:

A pointer to the current running task. `NULL` when the scheduler it's in a busy state or when idle-task is running.

1.11 Demonstrative examples

1.11.1 A simple scheduling

This example demonstrates a simple environment setup for multiple tasks. Initially, only `task1` and `task2` are enabled. `task1` runs every 2 seconds 10 times and then stops. `task2` runs every 3 seconds indefinitely. `task1` enables `task3` at its first run. `task3` run every 5 seconds. `task1` disables `task3` on its last iteration and changed `task2` to run every 1/2 seconds. In the end, `task2` is the only task running every 1/2 seconds.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#include "QuarkTS.h"
#define TIMER_TICK   0.001   /* 1ms */

qTask_t task1, task2, task3; /*task nodes*/
/*==================================================================*/
void interrupt Timer0_ISR(void) {
```

```
    qClock_SysTick();
}
/*==================================================================*/
void Task1_Callback(qEvent_t e){
    puts("Task1");

    if(e->FirstIteration){
        qTaskResume(&Task3);
    }

    if(e->LastIteration){
        qTaskSuspend(&Task3);
        qTaskSetTime(&Task2, 0.5);
    }
}
/*==================================================================*/
void Task2_Callback(qEvent_t e){
    puts("Task2");
}
/*==================================================================*/
void Task3_Callback(qEvent_t e){
    puts("Task3");
}
/*==================================================================*/
int main(void) {
    HardwareSetup();  /*hardware initialization function*/
    /*function to fire an interrupt at 1ms - timer tick*/
    Configure_Periodic_Timer0_Interrupt_1ms();

    qSchedulerSetup(NULL, TIMER_TICK, NULL, 10);

    qSchedulerAdd_Task(&Task1, Task1_Callback, 50, 2.0, 10,
                       qEnabled, NULL);
    qSchedulerAdd_Task(&Task2, Task2_Callback, 50, 3.0, qPeriodic,
                       qEnabled, NULL);
    qSchedulerAdd_Task(&Task2, Task3_Callback, 50, 5.0, qPeriodic,
                       qDisabled, NULL);
    qSchedulerRun();

    return 0;
}
```

1.11.2 Using the task argument

As seen in section 1.4.2, tasks can accept a parameter of type pointer to void (`void*`). This parameter could be used for multiple applications, including storage, task identification, duplication removal and others. The following example shows the usage of this argument to avoid callback duplication among tasks with the same behavior.

Consider a scenario where you have to build a digital controller for many physical variables, for example, a PID controller for temperature, humidity and light. The PID algorithm will be the same for all variables. The only difference will be the variable input, the controlled output action and the PID gains. In this case, each of the PID tasks will utilize the same callback methods. The only difference will be the I/O parameters (specific for each PID controller).

Let's define a PID data structure with the I/O variables and gains.

```
typedef struct{
     float yt; /*Measured variable (Controller Input)*/
     float ut; /*Controlled variable (Controller Output)*/
     float ie; /*Accumulated error*/
     float pe; /*Previous error*/
     float dt; /*Controller Time Step*/
     float sp; /*Set-Point*/
     float Kc, Ki, Kd; /*PID Gains*/
 }PID_Params_t;

 PID_Params_t TemperatureControl = {
    0,0,0,0, /*Initial IO state of yt and ut*/
    1.5, /*time step*/
    28.5, /*Set-Point*/
    0.89, 0.122, 0.001 /*Kc, Ki, Kd*/
 };

 PID_Params_t HumidityControl= {
    0,0,0,0, /*Initial IO state of yt and ut*/
    1, /*time step*/
    60.0, /*Set-Point*/
    2.5, 0.2354, 0.0015 /*Kc, Ki, Kd*/
 };

 PID_Params_t LightControl= {
    0,0,0,0, /*Initial IO state of yt and ut*/
    0.5, /*time step*/
    45.0, /*Set-Point*/
    5.36, 0.0891, 0.0 /*Kc, Ki, Kd*/
 };
```

A task will be added to the scheme to collect the sensor data and apply the respective control output.

```
qSchedulerAdd_Task(&IO_TASK, IO_TASK_Callback, qMedium_Priority,
                   0.1, qPeriodic, qEnabled, "iotask");
```

```
void IO_TASK_Callback(qEvent_t e){
    TemperatureControl.yt  = SampleTemperatureSensor();
    HumidityControl.yt  = SampleHumiditySensor();
    LightControl.yt  = SampleLightSensor();
    WriteTemperatureActuatorValue( TemperatureControl.ut );
    WriteHumidityActuatorValue( HumidityControl.ut );
    WriteLightActuatorValue( LightControl.ut );
}
```

Then, three different tasks are created to apply the respective PID controller. Note that these tasks refer to the same callback methods and we assign pointers to the respective variables.

```
qSchedulerAdd_Task(&TEMPERATURE_CONTROL_TASK, PIDControl_Callback,
                    qHigh_Priority, TemperatureControl.dt ,
```

```
                        qPeriodic, qEnabled, &TemperatureControl);
qSchedulerAdd_Task(&HUMIDITY_CONTROL_TASK, PIDControl_Callback,
                        qHigh_Priority, HumidityControl.dt,
                        qPeriodic, qEnabled, &HumidityControl);
qSchedulerAdd_Task(&LIGHT_CONTROL_TASK, PIDControl_Callback,
                        qHigh_Priority, LightControl.dt,
                        qPeriodic, qEnabled, &LightControl);
```

```
void PIDControl_Callback(qEvent_t e){
    float Error, derivative;

    /*
    Obtain the reference to the specific PID controller
    usign the TaskData field from the qEvent structure
    */
    PID_Params_t *Controller = (PID_Params_t *)e->TaskData;

    /*Compute the error*/
    Error = Controller->sp - Controller->yt;
    /*Compute the accumulated error using backward integral approximation*/
    Controller->ie += Error*Controller->dt;
    /*update and compute the derivative term*/
    derivative = (Error - Controller->pe)/Controller->dt;
    /*update the previous error*/
    Controller->pe = Error;
    /*compute the pid control law*/
    Controller->ut = Controller->Kc*Error  +
                     Controller->Ki*Controller->ie +
                     Controller->Kd*derivative;
}
```

1.12 Configuration macros

Some OS features can be customized using a set of macros located in the header file `qconfig.h`. Here is the default configuration, followed by an explanation of each macro:

`Q_PRIORITY_LEVELS`	*Default:* `3`. The number of priorities available for application tasks.
`Q_SETUP_TIME_CANONICAL`	*Default:* `0(disabled)`. If enabled, the kernel assumes the timing base to 1mS(1KHz). So all time specifications for tasks and STimers must be set in milliseconds(mS). Also can be used to remove the floating-point operations when dealing with time. In some systems, can reduce the memory usage.
`Q_SETUP_TICK_IN_HERTZ`	*Default:* `0(disabled)`. If enabled, the timing base will be taken as frequency(Hz) instead of period(S) by `qSchedulerSetup()` (In some systems, can reduce the memory usage).
`Q_PRIO_QUEUE_SIZE`	*Default:* `10`. Size of the priority queue for notifications. This argument should be an integer number greater than zero. A zero value can be used to disable this functionality.
`Q_PRESERVE_TASK_ENTRY_ORDER`	*Default:* `0(disabled)`. Size of the priority queue for notifications. This argument should be an integer number greater than zero. A zero value can be used to disable this functionality.
`Q_MEMORY_MANAGER`	*Default:* `1(enabled)`. Used to enable or disable the memory management module.
`Q_BYTE_ALIGNMENT`	*Default:* `8`. Used by the memory management module to peform the byte-aligment.
`Q_DEFAULT_HEAP_SIZE`	*Default:* `2048`. The total amount of heap size for the default memory pool.
`Q_NOTIFICATION_SPREADER`	*Default:* `1(enabled)`. Used to enable or disable the spread notification functionality.
`Q_FSM`	*Default:* `1(enabled)`. Used to enable or disable the Finite State Machine (FSM) extension.
`Q_QUEUES`	*Default:* `1(enabled)`. Used to enable or disable the queues APIs for communication to tasks.
`Q_TRACE_VARIABLES`	*Default:* `1(enabled)`. Used to enable or disable debug and trace macros.
`Q_DEBUGTRACE_BUFSIZE`	*Default:* `36`. The buffer size for debug and trace macros.
`Q_DEBUGTRACE_FULL`	*Default:* `1(enabled)`. Used to enable of disable the extended output of trace macros.
`Q_ATCOMMAND_PARSER`	*Default:* `1(enabled)`. Used to enable or disable the CLI AT parser module.
`Q_TASK_COUNT_CYCLES`	*Default:* `1(enabled)`. Used to enable or disable the cycle count in tasks.
`Q_MAX_FTOA_PRECISION`	*Default:* `10`. The default precision used to perform float to ASCII conversions.
`Q_ATOF_FULL`	*Default:* `0(disabled)`. Used to enable or disable the scientific notation in ASCII to float conversions.
`Q_ALLOW_SCHEDULER_RELEASE`	*Default:* `0(disabled)`. Used to enable or disable the scheduler release functionality.
`Q_RESPONSE_HANDLER`	*Default:* `1(enabled)`. Used to enable or disable the response handler functionality.
`Q_EDGE_CHECK_IOGROUPS`	*Default:* `1(enabled)`. Used to enable or disable the edge check functionality for I/O groups .
`Q_BYTE_SIZED_BUFFERS`	*Default:* `1(enabled)`. Used to enable or disable the usage of Byte-sized buffers.

2 Events

2.1 Time elapsed

Running tasks at pre-determined rates is desirable in many situations, like sensory data acquisition, low-level servoing, control loops, action planning and system monitoring. As seen in section 1.4.2, you can schedule tasks at any interval your design demands, at least, if the time specification is lower than the scheduler tick. When an application consists of several periodic tasks with individual timing constraints, a few points must be taken:

- When the time interval of a certain task has elapsed, the scheduler triggers an event (*byTimeElapsed*) that put the task in a `qReady` state (see figure 5).
- If a task has a finite number iterations, the scheduler will disable the task when the number of iterations reaches the programmed value.
- Tasks always have an inherent time-lag that can be noticed even more, when the programmed time-interval is too low (see figure 5). In a real-time context, it is important to reduce this time-lag or jitter, to an acceptable level for the application.

> QuarkTS can generally meet a time deadline if you use lightweight code in the callbacks and there is a reduced pool of pending tasks, so it can be considered a soft real-time scheduler, however, it cannot meet a deadline deterministically like a hard real-time OS.

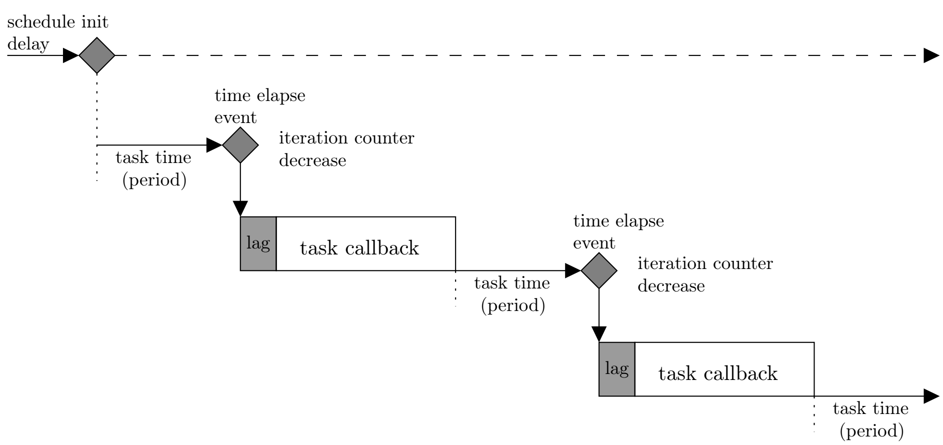

Figure 5: Inherit time lag

- The most significant delay times are produced inside the callbacks. As mentioned before, use short efficient callback methods written for cooperative scheduling.
- If two tasks have the same time-interval, the scheduler executes first, the task with the highest priority value (see figure 6).

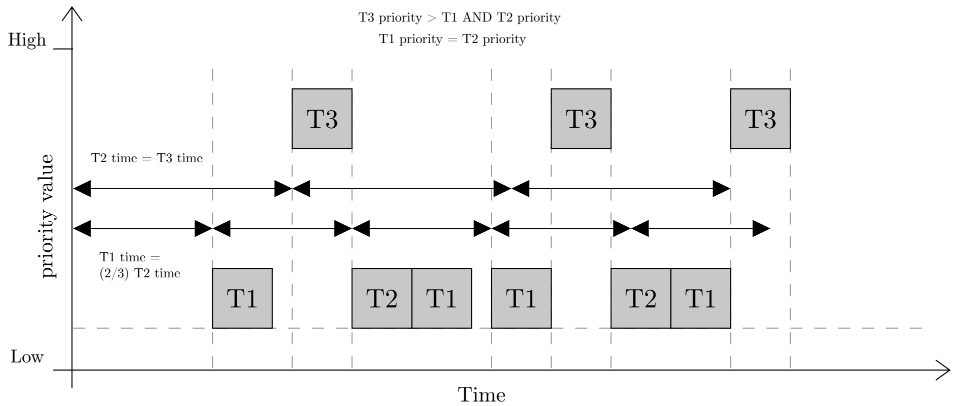

Figure 6: Priority scheduling example with three (3) tasks attached triggered by time-elapsed events

2.2 Asynchronous events and inter-task communication

Applications existing in heavy environments require tasks and ISR interacting with each other, forcing the application to implement some event model. Here, we understand events, as any identifiable occurrence that has significance for the embedded system. As such, events include changes in hardware, user-generated actions or messages coming from components of the application itself.

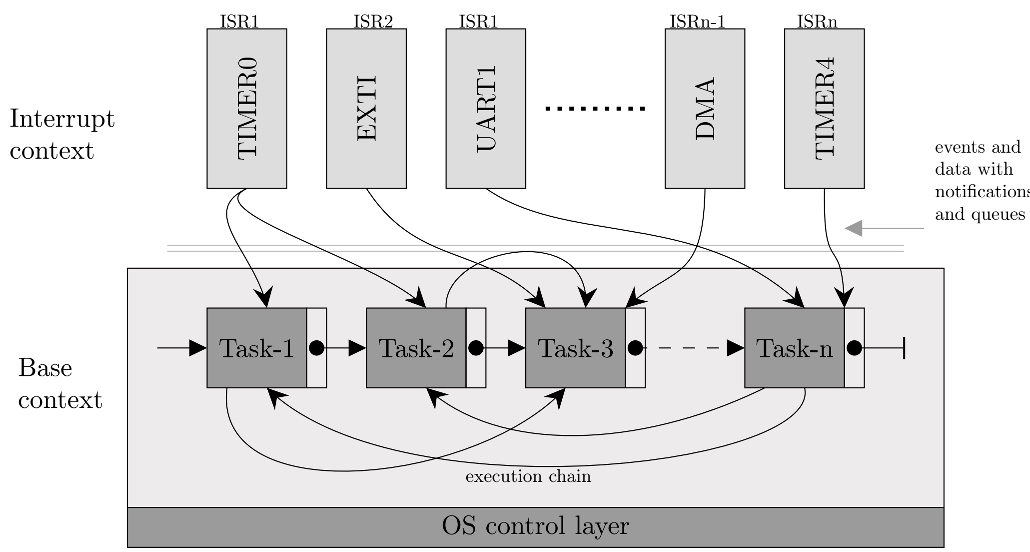

Figure 7: Heavy cooperative environment

As shown in figure 5, two main scenarios are presented, *ISR-to-task* and *task-to-task* interaction. When using interrupts to catch external events, it is expected to be handled with fast and lightweight code to reduce the variable ISR overhead introduced by the code itself. If too much overhead is used inside an ISR, the system will tend to lose future events. In some specific situations, the best approach is to synchronize the ISR with a task to leave the heavy job in the base-level instead of the interrupt-level.

The other scenario is when a task is performing a specific job and another task must be awakened to perform some activities when the other task finishes.

Both scenarios require some ways in which tasks can communicate with each other. For this, the OS does not impose any specific event processing strategy to the application designer but does provide features that allow the chosen strategy to be implemented in a simple and maintainable way. From the OS perspective, these features are just sources of asynchronous events with specific triggers and related data.

The OS provides the following features for task communication:

2.2.1 Notifications

The notifications allow tasks to interact with other tasks and to synchronize with ISRs without the need of intermediate variables or separate communication objects. By using notifications, a task or ISR can launch another task sending an event and related data to the receiving task. This is depicted in figure 8.

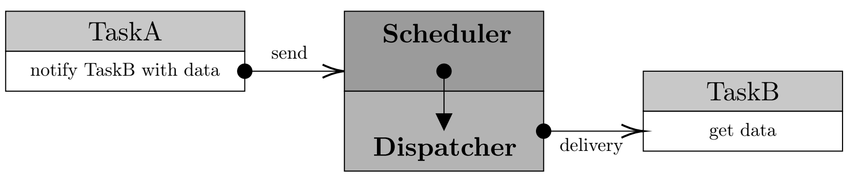

Figure 8: A notification used to send an event directly from one task to another

2.2.1.1 Simple notifications: Each task node has an 32-bit notification value which is initialized to zero when a task is added to the scheme. The API `qTaskSendNotification()` is used to send an event directly updating the receiving task's notification value increasing it by one. As long the scheduler sees a non-zero value, the task will be changed to a *qReady* state and eventually, the dispatcher will launch the task according to the execution chain. After served, the notification value is later decreased.

```
qBool_t qTaskSendNotification( qTask_t * const Task, void* eventdata)
```

Parameters

- `Task` : The pointer of the task node to which the notification is being sent
- `EventData` : Specific event user-data.

Return value:

`qTrue` if the notification has been sent, or `qFalse` if the notification value reach their max value.

> Sending simple notifications using `qTaskSendNotification()` is interrupt-safe, however, this only catches one event per task because the API overwrites the associated data.

2.2.1.2 Queued notifications : If the application notifies multiple events to the same task, queued notifications are the right solution instead of using simple notifications.

Here, the `qTaskQueueNotification()` take advantage of the scheduler FIFO priority-queue. This kind of queue, is somewhat similar to a standard queue, with an important distinction: when a notification is sent, the task is added to the queue with the corresponding priority level, and will be later removed from the queue with the highest priority task first [2]. That is, the tasks are (conceptually) stored in the queue in priority order instead of the insertion order. If two tasks with the same priority are notified, they are served in the FIFO form according to their order inside the queue. Figure 9 illustrates this behavior.

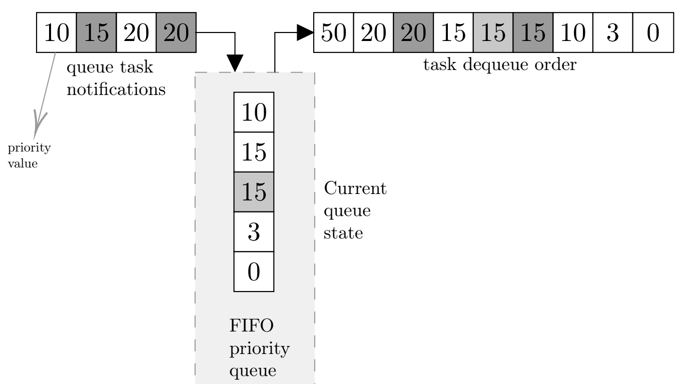

Figure 9: Priority-queue behavior

The scheduler always checks the queue state first, being this event the one with more precedence among the others. If the queue has elements, the scheduler algorithm will extract the data and the corresponding task will be launched with the trigger flag set in *byNotificationQueued*.

```
qBool_t qTaskQueueNotification( qTask_t * const Task, void* eventdata )
```

Parameters

- `Task` : The pointer of the task node to which the notification is being sent

- `EventData` : Specific event user-data.

Return value:

`qTrue` if the notification has been inserted in the queue, or `qFalse` if an error occurred or the queue exceeds the size.

> Among all the provided events, queued notifications have higher precedence.

Figure 10, shows a cooperative environment with five tasks. Initially, the scheduler activates `Task-E`, then, this task enqueues data to `Task-A` and `Task-B` respectively using the `qTaskQueueNotification()` function. In the next scheduler cycle, the scheduler realizes that the priority-queue is not empty, generating an activation over the task located at the beginning of the queue. In this case, `Task-A` will be launched and its respective data will be extracted from the queue. However, `Task-A` also enqueues data to `Task-C` and `Task-D`. As mentioned previously, this is a priority-queue, so the scheduler makes a new reordering. In this context, the next queue extraction will be for `Task-D`, `Task-C`, and `Task-B` sequentially.

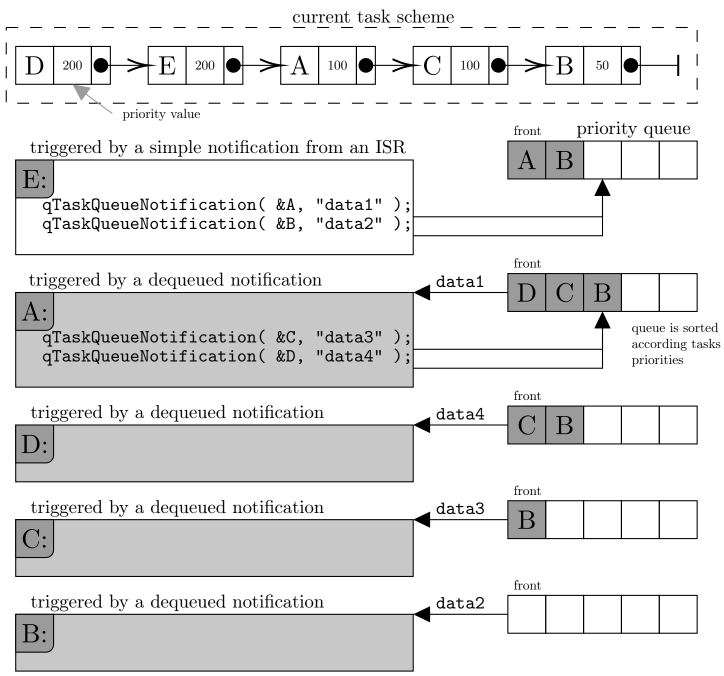

Figure 10: Priority-queue example

> Any queue extraction involves an activation to the receiving task. The extracted data will be available inside the `qEvent_t` structure.

2.2.2 Spread a notification

In some systems, we need the ability to broadcast an event to all tasks. This is often referred to as a *barrier*. This means that a group of tasks should stop activities at some point and cannot proceed until another task or ISR raise a specific event. For this kind of implementations, the `qSchedulerSpreadNotification()` can be used.

```
qBool_t qSchedulerSpreadNotification( const void *eventdata,
                                      const qTaskNotifyMode_t mode )
```

Parameters

- `eventdata` : Specific event user-data.
- `mode` : The method used to spread the event: `Q_NOTIFY_SIMPLE` or `Q_NOTIFY_QUEUED`

Return value:

`qTrue` on success. Otherwise `qFalse`.

This function spreads a notification event among all the tasks in the scheduling scheme, so, for tasks that are not part of the barrier, just discard the notification.

The spread operation will be performed in the next scheduling cycle.

2.2.3 Queues

A queue its a linear data structure with simple operations based on the FIFO (First In First Out) principle. It is capable to hold a finite number of fixed-size data items. The maximum number of items that a queue can hold is called its *length*. Both the length and the size of each data item are set when the queue is created.

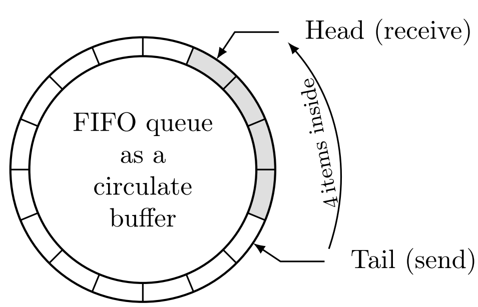

Figure 11: qQueues conceptual representation

As showed in figure 11, the last position is connected back to the first position to make a circle. It is also called *ring-buffer* or *circular-queue*.

In general, this kind of data structure is used to serialize data between tasks, allowing some elasticity in time. In many cases, the queue is used as a data buffer in interrupt service routines. This buffer will collect the data so, at some later time, another task can fetch the data for further processing. This use case is the single "task to task" buffering case. There are also other applications for queues as serializing many data streams into one receiving streams (multiple tasks to a single task) or vice-versa (single task to multiple tasks). The usage of this data structure is detailed in section 2.4.2.

> The OS uses the queue by copy method. Queuing by copy is considered to be simultaneously more powerful and simpler to use than queuing by reference.

Queuing by copy does not prevent the queue from also being used to queue by reference. For example, when the size of the data being queued makes it impractical to copy the data into the queue, then a pointer to the data can be copied into the queue instead.

2.2.4 Event Flags

Every task node has a set of built-in event bits called *Event-Flags*, that can be used to indicate if an event has occurred or not. They are somewhat similar to signals, but with greater flexibility, providing a low cost, but flexible means of passing simple messages between tasks. One task can set or clear any combination of event flags. Another task may read the event flag group at any time or may wait for a specific pattern of flags.

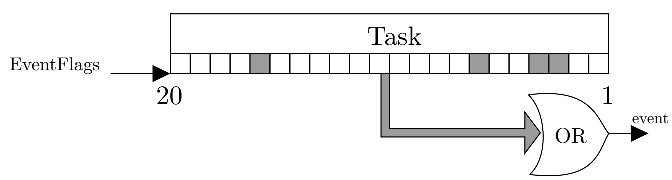

Figure 12: Task event flags

Up to twenty(20) bit-flags are available per task and meanwhile the scheduler sees that one event-flag is set, the kernel will trigger the task execution.
The function `qTaskModifyEventFlags` is intended to modify the task event-flags:

```
void qTaskModifyEventFlags( qTask_t * const Task, qTaskFlag_t flags,
                            qBool_t action )
```

Parameters

- `Task` : A pointer to the task node.
- `flags` : The flags to modify. Can be combined with a bitwise 'OR' ('|').

  `QEVENTFLAG_01 | QEVENTFLAG_02 | QEVENTFLAG_03 | ... | QEVENTFLAG_20`

- `action` : `QEVENTFLAG_SET` or `QEVENTFLAG_CLEAR`.

> The scheduler will put the task into a *qReady* state when any of the available event-flags is set. The flags should be cleared by the application writer explicitly.

To read or check the event flags, the application can use one of the following API functions:

```
qTaskFlag_t qTaskReadEventFlags( const qTask_t * const Task )
```

Parameters

- `Task` : A pointer to the task node.

Return value:

The EventFlag value of the task.

```
qBool_t qTaskCheckEventFlags( qTask_t * const Task, qTaskFlag_t
                              FlagsToCheck, qBool_t ClearOnExit,
                              qBool_t CheckForAll )
```

Parameters

- `Task` : A pointer to the task node.
- `FlagsToCheck` : A bitwise value that indicates the flags to test inside the EventFlags. Can be combined with a bitwise 'OR' (`'|'`).
- `ClearOnExit` : If is set to `qTrue` then any flags set in the value passed as the `FlagsToCheck` parameter will be cleared in the event group before this function returns only when the condition is meet.
- `CheckForAll` : Used to create either a logical AND test (where all flags must be set)or a logical OR test (where one or more flags must be set) as follows:

  If is set to `qTrue` this API will return `qTrue` when either all the flags set in the value passed as the `FlagsToCheck` parameter are set in the task's EventFlags.

  If is set to `qFalse` this API will return `qTrue` when any of the flags set in the value passed as the `FlagsToCheck` parameter are set in the task's EventFlags.

Return value:

`qTrue` if the condition is meet, otherwise return `qFalse`.

2.3 Retrieving the event data

As you can read in the previous sections, tasks can be triggered from multiple event sources (time-elapsed, notifications, queues and event-flags). This can lead to several situations that must be handled by the application writer from the task context, for example:

- What is the event source that triggers the task execution?
- How to get the event associated data?
- What is the task execution status?

The OS provides a simple approach for this, a data structure with all the regarding information of the task execution. This structure, that is already defined in the callback function as the `qEvent_t` argument, is filled by the kernel dispatcher, so the application writer only needs to read the fields inside.

This data structure is defined as:

```
typedef struct{
    qTrigger_t Trigger;
    void *TaskData;
    void *EventData;
    qBool_t FirstCall, FirstIteration, LastIteration;
}qEvent_t;
```

Every field of the structure are described as follows

- `Trigger` : The flag that indicates the event source that triggers the task execution. This flag can only have nine(9) possible values:
  - `byTimeElapsed` : When the time specified for the task elapsed.
  - `byNotificationQueued` : When there is a queued notification in the FIFO priority queue. For this trigger, the dispatcher performs a dequeue operation automatically. A pointer to the extracted event-data will be available in the `EventData` field.
  - `byNotificationSimple` : When the execution chain does, according to a requirement of asynchronous notification event prompted by `qSendEvent`. A pointer to the dequeued data will be available in the `EventData` field.
  - `byQueueReceiver` : When there are elements available in the attached queue, the scheduler makes a data dequeue(auto-receive) from the front. A pointer to the received data will be will be available in the `EventData` field.

  - `byQueueFull` : When the attached queue is full. A pointer to the queue will be available in the `EventData` field.
  - `byQueueCount` : When the element-count of the attached queue reaches the specified value. A pointer to the queue will be available in the `EventData` field.
  - `byQueueEmpty` : When the attached queue is empty. A pointer to the queue will be available in the `EventData` field.
  - `byEventFlags` : When any of the available event flags is set. Flags should be cleared by the application writer.
  - `byNoReadyTasks` : Only when the idle task is triggered

- `TaskData` : The storage pointer. Tasks can store a pointer to specific variable, structure or array, which represents specific user data for a particular task. This may be needed if you plan to use the same callback method for multiple tasks.

- `EventData` : Associated data of the event. Specific data will reside here according to the event source. This field will have a `NULL` value when the trigger gets one of this values: *byTimeElapsed*, *byEventFlags* and *byNoReadyTasks*.

- `FirstCall` : This flag indicates that a task is running for the first time. Can be used for data initialization purposes.

- `FirstIteration` : Indicates whether current pass is the first iteration of the task. This flag will be only set when time-elapsed events occurs and the iteration counter has been parameterized.

- `LastIteration` : Indicates whether current pass is the last iteration of the task. This flag will be only set when time-elapsed events occurs and the iteration counter has been parameterized.

> Asynchronous events never change the task iteration counter, consequently, it has no effect on related fields, `FirstIteration` and `LastIteration`.

2.4 Implementation guidelines

2.4.1 Sending notifications

The kernel handles all the notifications by itself (simple or queued), so intermediate objects aren't needed. Just calling `qTaskSendNotification()` or `qTaskQueueNotification()` is enough to send notifications. After the task callback is invoked, the notification is cleared by the dispatcher. Here the application writer must read the respective fields of the event-data structure to check the received notification.

The next example shows an ISR to task communication. Two interrupts send notifications to a single task with specific event data. The receiver task (`taskA`) after further processing, send an event to `taskB` to handle the event generated by the transmitter (`taskA`).

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "HAL.h" /*hardware dependent code*/
#include "QuarkTS.h"

qTask_t taskA, taskB;
void taskA_Callback(qEvent_t e);
void taskB_Callback(qEvent_t e);

const char *app_events[] = {
                            "Timer1seg",
                            "ButtonRisingEdge",
                            "ButtonFallingEdge",
                            "3Count_ButtonPush"
                            };

/*==================================================================*/
void interrupt Timer1Second_ISR(void){
    qTaskSendNotification(&taskA, NULL);
    HAL_ClearInterruptFlags(HAL_TMR_ISR); /*hardware dependent code*/
}
/*==================================================================*/
void interrupt ExternalInput_ISR(void){
    if( HAL_GetInputEdge() == RISING_EDGE ){ /*hardware dependent code*/
        qTaskQueueNotification(&taskA, app_events[1]);
    }
    else{
        qTaskQueueNotification(&taskA, app_events[2]);
    }
    HAL_ClearInterruptFlags(HAL_EXT_ISR); /*hardware dependent code*/
}
/*==================================================================*/
void taskA_Callback(qEvent_t e){
    static int press_counter = 0;

    switch(e->Trigger){ /*check the source of the event*/
        case byNotificationSimple:
            /*
            * Do something here to process the timer event
            */
            break;
        case byNotificationQueued:
            /*here, we only care the Falling Edge events*/
            if( strcmp( e->EventData, "ButtonFallingEdge" )==0 ){
                press_counter++; /*count the button press*/
                if( press_counter == 3){ /*after 3 presses*/
                    /*send the notification of 3 presses to taskB*/
                    qTaskSendNotification(taskB, app_events[3]);
                    press_counter = 0;
                }
            }
            break;
        default:
            break;
    }
}
/*==================================================================*/
```

```c
void taskB_Callback(qEvent_t e){
    if( byNotificationSimple == e->Trigger){
        /*
         * we can do more here, but this is just an example,
         * so, this task will only print out the received
         * notification event.
         */
        qDebugMessage( e->EventData );
    }
}
/*================================================================*/
int main(void){
    HAL_Setup_MCU(); /*hardware dependent code*/
    qSetDebugFcn( HAL_OutPutChar );
    /* setup the scheduler to handle up to 10 queued notifications*/
    qSchedulerSetup(HAL_GetTick,0.001 NULL, 10);
    qSchedulerAdd_EventTask(&taskA, taskA_Callback, qLowest_Priority, NULL);
    qSchedulerAdd_EventTask(&taskB, taskB_Callback, qLowest_Priority, NULL);
    qSchedulerRun();
}
```

2.4.2 Creating a queue : `qQueueCreate`

A queue must be explicitly created before it can be used.
These objects are referenced by handles, which are variables of type `qQueue_t`. The `qQueueCreate()` API function creates a queue and initialize the instance.

The required RAM for the queue data should be provided by the application writer and could be statically allocated at compile time or in run-time using the memory management module.

```c
qBool_t qQueueCreate( qQueue_t * const obj, void* DataArea,
                      size_t ItemSize, size_t ItemsCount )
```

Parameters

- `obj` : A pointer to the queue object
- `DataArea` : Must point to a data block or array of data that is at least large enough to hold the maximum number of items that can be in the queue at any one time
- `ItemSize` : Size of one element in the data block
- `ItemsCount` : The maximum number of items the queue can hold at any one time.

2.4.3 Performing queue operations

```c
qBool_t qQueueSendToBack( qQueue_t * const obj, void *ItemToQueue )
```

```c
qBool_t qQueueSendToFront( qQueue_t * const obj, void *ItemToQueue )
```

As might be expected, `qQueueSendToBack()` is used to send data to the back (tail) of a queue, and `qQueueSendToFront()` is used to send data to the front (head) of a queue. `qQueueSend()` is equivalent to, and exactly the same as, `qQueueSendToBack()`.

Parameters

- `obj` : A pointer to the queue object
- `ItemToQueue` : A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `ItemToQueue` into the queue storage area.

Return value

`qTrue` if data was retrieved from the queue, otherwise returns `qFalse`.

The API `qQueueReceive()` is used to receive (read) an item from a queue. The item that is received is removed from the queue.

```
qBool_t qQueueReceive (qQueue_t * const obj, void *dest )
```

Parameters

- `obj` : A pointer to the queue object
- `ItemToQueue` : Pointer to the buffer into which the received item will be copied.

Return value

`qTrue` if data was retrieved from the queue, otherwise returns `qFalse`.

2.4.4 Attach a queue to a task

Additional features are provided by the kernel when the queues are attached to tasks; this allows the scheduler to pass specific queue events to it, usually, states of the object itself that needs to be handled, in this case by a task. For this, the following API is provided:

```
qBool_t qTaskAttachQueue( qTask_t * const Task, qQueue_t * const Queue,
                          const qQueueLinkMode_t Mode, const qUINT16_t arg )
```

Parameters

- `obj` : A pointer to the task node
- `obj` : A pointer to the queue object
- `Mode` : Attach mode. This implies the event that will trigger the task according to one of the following modes:

  - qQUEUE_DEQUEUE : The task will be triggered if there are elements in the queue.
  - qQUEUE_FULL : The task will be triggered if the queue is full.
  - qQUEUE_COUNT : The task will be triggered if the count of elements in the queue reach the specified value.
  - qQUEUE_EMPTY : The task will be triggered if the queue is empty.

- arg : This argument defines if the queue will be attached (qATTACH) or detached (qDETACH) from the task. If the qQUEUE_COUNT mode is specified, this value will be used to check the element count of the queue. A zero value will act as qDETACH action.

For the qQUEUE_DEQUEUE mode, data from the front of the queue will be received automatically in every trigger, this involves a data removal after the task is served. During the respective task execution, the EventData field of the qEvent_t structure will be pointing to the extracted data.
For the other modes, the EventData field will point to the queue that triggered the event.

2.4.5 A queue example

This example shows the usage of QuarkTS queues. The application is the classic producer/consumer example. The producer task puts data into the queue. When the queue reaches a specific item count, the consumer task is triggered to start fetching data from the queue. Here, both tasks are attached to the queue.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include "QuarkTS.h"
#define TIMER_TICK   0.001   /* 1ms */

/*---------------------------------------------------------------------*/
void interrupt Timer0_ISR(void) {
    qClock_SysTick();
}
/*---------------------------------------------------------------------*/
qTask_t TSK_PRODUCER, TSK_CONSUMER; /*task nodes*/
qQueue_t UserQueue; /*Queue Handler*/
/*---------------------------------------------------------------------*/
/* The producer task puts data into the buffer if there is enough free
 * space in it, otherwise the task block itself and wait until the queue
 * is empty to resume.
 */
void TSK_Producer_Callback(qEvent_t e) {
    static qUINT16_t unData = 0;
    unData++;
    /*Queue is empty, enable the producer if it was disabled*/
    if(e->Trigger == byQueueEmpty){
        qTaskResume(qTaskSelf());
    }
```

```
    if (!qQueueSendToBack(&UserQueue, &unData)){ /*send data to the queue*/
        /*
        * if the data insertion fails, the queue is full
        * and the task disables itself
        */
      qTaskSuspend(qTaskSelf());
    }
}
/*-----------------------------------------------------------------------*/
/* The consumer task gets one element from the queue.*/
void TSK_Consumer_Callback(qEvent_t e) {
    qUINT16_t unData;
    qQueue_t *ptrQueue; /*a pointer to the queue that triggers the event*/
    if(e->Trigger == byRBufferCount){
      ptrQueue= (qQueue_t *)e->EventData;
      qQueueReceive(ptrQueue, &unData);
      return;
    }
}
/*-----------------------------------------------------------------------*/
void IdleTask_Callback(qEvent_t e){
    /*nothing to do...*/
}
/*-----------------------------------------------------------------------*/
int main(void) {
    qUINT8_t BufferMem[16*sizeof(qUINT16_t)] = {0};
    HardwareSetup();  //hardware specific code
    /* next line is used to setup hardware with specific code to fire
     * interrupts at 1ms - timer tick*/
    Configure_Periodic_Timer0_Interrupt_1ms();

    qQueueCreate(&UserQueue, BufferMem /*Memory block used*/,
                 sizeof(qUINT16_t) /*element size*/,
                 16 /* element count*/ );
    qSchedulerSetup(NULL, TIMER_TICK, IdleTask_Callback, 10);

    /*  Append the producer task with 100mS rate. */
    qSchedulerAdd_Task(&TSK_PRODUCER, TSK_Producer_Callback,
                       qMedium_Priority, 0.1, qPeriodic, qEnabled,
                       "producer");
    /* Append the consumer as an event task. The consumer will
     * wait until an event trigger their execution
     */
    qSchedulerAdd_EventTask(&TSK_CONSUMER, TSK_Consumer_Callback,
                            qMedium_Priority, "consumer");
    /* the queue will be attached to the consumer task
     * in qQUEUE_COUNT mode. This mode sends an event to the consumer
     * task when the queue fills to a level of 4 elements
     */
    qTaskAttachQueue(&TSK_CONSUMER, &UserQueue, qQUEUE_COUNT, 4);
    /* the queue will be attached to the producer task in
     * qQUEUE_EMPTY mode. This mode sends an event to the producer
     * task when the queue is empty
     */
    qTaskAttachQueue(&TSK_PRODUCER, &UserQueue, qQUEUE_EMPTY, qATTACH);
    qSchedulerRun();
    return 0;
}
```

2.4.6 Other queue APIs

```
void qQueueReset( qQueue_t * const obj )
```

Resets a queue to its original empty state.

Parameters:

- `obj` : A pointer to the queue object

```
qBool_t qQueueIsEmpty( const qQueue_t * const obj );
```

Returns the empty status of the queue.

Parameters:

- `obj` : A pointer to the queue object

Return value:

`qTrue` if the queue is empty, `qFalse` if it is not.

```
size_t qQueueCount( const qQueue_t * const obj )
```

Returns the number of items in the queue.

Parameters:

- `obj` : A pointer to the queue object.

Return value:

The number of elements in the queue.

```
qBool_t qQueueIsFull( const qQueue_t * const obj )
```

Returns the full status of the queue.

Parameters:

- `obj` : A pointer to the queue object.

Return value:

`qTrue` if the queue is full, `qFalse` if it is not.

```
void* qQueuePeek( const qQueue_t * const obj )
```

Looks at the data from the front of the queue without removing it.

Parameters:

- `obj` : A pointer to the queue object.

Return value:

Pointer to the data, or `NULL` if there is nothing in the queue.

```
qBool_t qQueueRemoveFront( qQueue_t * const obj )
```

Remove the data located at the front of the queue.

Parameters:

- `obj` : A pointer to the queue object.

Return value:

`qTrue` if data was removed from the queue, otherwise returns `qFalse`.

2.4.7 Using the task Event-flags

This example demonstrate the usage of *Event-flags*. The idle task will transmit data generated from another task only, when the required conditions are meet, including two events from an ISR (A timer expiration and the change of a digital input) and when a new set of data are generated. The task that generates the data should wait until the idle task transmission is done to generate a new data set.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include "QuarkTS.h"
#define TIMER_TICK   0.001   /* 1ms */

/*event flags application definitions */
#define SWITCH_CHANGED  QEVENTFLAG_01
#define TIMER_EXPIRED   QEVENTFLAG_02
#define DATA_READY      QEVENTFLAG_03
#define DATA_TXMIT      QEVENTFLAG_04

qTask_t TaskDataProducer;
uint8_t dataToTransmit[10] = {0};
```

```
/*-----------------------------------------------------------------------*/
void interrupt Timer0_ISR(void) {
    qClock_SysTick();
}
/*-----------------------------------------------------------------------*/
void interrupt Timer1_ISR(void) {
    qTaskModifyEventFlags( &TaskDataProducer, TIMER_EXPIRED,
                           QEVENTFLAG_SET );
}
/*-----------------------------------------------------------------------*/
void interrupt EXTI_ISR(void){
    if( EXTI_IsRisingEdge() ){
        qTaskModifyEventFlags( &TaskDataProducer, SWITCH_CHANGED,
                               QEVENTFLAG_SET );
    }
}
/*-----------------------------------------------------------------------*/
void TaskDataProducer_Callback(qEvent_t e){
    qBool_t condition;
    condition = qTaskCheckEventFlags( &TaskDataProducer,
                                      DATA_TXMIT, qTrue, qTrue );
    if( qTrue == condition){
        GenerateData( dataToTransmit );
        qTaskModifyEventFlags( &TaskDataProducer, DATA_READY,
                                 QEVENTFLAG_SET );
    }
}
/*-----------------------------------------------------------------------*/
void IdleTask_Callback(qEvent_t e){
    qBool_t condition;
    condition = qTaskCheckEventFlags( &TaskDataProducer,
                                       DATA_READY | SWITCH_CHANGED |
                                       TIMER_EXPIRED,
                                       qTrue, qTrue );
    if( qTrue == condition){
        TransmitData( dataToTransmit );
        qTaskModifyEventFlags( &TaskDataProducer,
                                DATA_TXMIT, QEVENTFLAG_SET );
    }
}
/*-----------------------------------------------------------------------*/
int main(void) {
    HardwareSetup();  //hardware specific code
    /* next line is used to setup hardware with specific code to fire
     * interrupts at 1ms - timer tick*/
    Configure_Periodic_Timer0_Interrupt_1ms();
    Configure_Periodic_Timer1_Interrupt_2s();
    Configure_External_Interrupt();
    /*
    Idle task will be responsible to transmit the generate the data after
    all conditions are meet
    */
    qSchedulerSetup(NULL, TIMER_TICK, IdleTask_Callback, 10);
    /*
    The task will wait until data is transmitted to generate another set of
    data
    */
    qSchedulerAdd_EventTask(&TaskDataProducer, TaskDataProducer_Callback,
                            qHigh_Priority, "DATAPRODUCER");
```

```
    /*
    Set the flag DATA_TXMIT as initial condition to allow the data
    generation at startup
    */
    qTaskModifyEventFlags( &TaskDataProducer, DATA_TXMIT, QEVENTFLAG_SET );
    qSchedulerRun();
    for(;;){}
    return 0;
}
```

3 Modules

3.1 STimers

There are several situations where the application doesn't need such hard real-time precision for timing actions and we just need that a section of code will execute when at least, some amount of time has elapsed. For these purposes, STimers (Software-Timers) is the right module to use.

The STimers implementation doesn't access resources from the interrupt context, does not consume any significant processing time unless a timer has actually expired, does not add any processing overhead to the *sys-tick* interrupt, and does not walk any other data structures. The timer service just takes the value of the existing kernel clock source for reference (t_{sys}), allowing timer functionality to be added to an application with minimal impact.

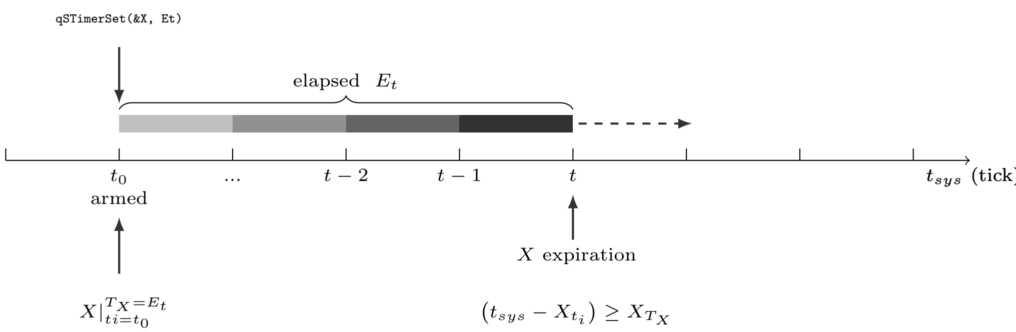

Figure 13: STimers operation

As illustrated in figure 13, the time expiration check is roll-over safe by restricting it, to the only calculation that make sense for timestamps, $t_{sys} - X_{T_x}$, that yields a duration namely the amount of time elapsed between the current instant(t_{sys}) and the later instant, specifically, the tick taken at the arming instant with (`qSTimerSet()`), (X_{t_i}). Thanks to modular arithmetic, both of these are guaranteed to work fine across the clock-source rollover(a 32bit unsigned-counter), at least, as long the delays involved are shorter than 49.7 days.

Features:

- Provides a non-blocking equivalent to delay function.
- Each STimer encapsulates its own expiration (timeout) time.
- Provides elapsed and remaining time APIs.
- As mentioned before, STimers uses the same kernel clock source, this means the time-elapsed calculation use the `qClock_GetTick()` API, therefore, the time resolution has the same value passed when the scheduler has been initialized with `qSchedulerSetup()`.

3.1.1 Usign a STimer

A STimer is referenced by handle, a variable of type `qSTimer_t` and preferably, should be initialized by the `QSTIMER_INITIALIZER` constant before any usage.

To use them, the code should follow a specific pattern that deals with the states of this object. All related APIs are designed to be non-blocking, this means there are ideal for use in cooperative environments as the one provided by the OS itself. To minimize the implementation, this object is intentionally created to behave like a binary object, this implies that it only handles two states, *Armed* and *Disarmed*.
An *Armed* timer means that is already running with a specified preset value and a *Disarmed* timer, is their opposite, this means it doesn't have a preset value, so consequently, it is not running at all.

The arming actions can be performed with `qSTimerSet()` or `qSTimerFreeRun()` and disarming with `qSTimerDisarm()`.

A detailed API description are presented below. (For `qSTimerDisarm()` ignore the `Time` argument.)

```
qBool_t qSTimerSet( qSTimer_t * const obj, const qTime_t Time )
```

```
qBool_t qSTimerFreeRun( qSTimer_t * const obj, const qTime_t Time )
```

```
void qSTimerDisarm( qSTimer_t * const obj )
```

Parameters

- `obj` : A pointer to the STimer object.
- `Time` : The expiration time(must be specified in seconds).

Here, `qSTimerFreeRun()` it's a more advanced API, it does the check and the arming. If disarmed, it gets armed immediately with the specified time. If armed, the time argument is ignored and the API only checks for expiration. When the time expires, the STimer gets armed immediately taking the specified time.

Return Value

For `qSTimerSet()` `qTrue` on success, otherwise, returns `qFalse`.
For `qSTimerFreeRun()` returns `qTrue` when the STimer expires, otherwise, returns `qFalse`. For a disarmed STimer, also returns `qFalse`.
None for `qSTimerDisarm()`.

All possible checking actions are also provided for this object, including `qSTimerElapsed()` , `qSTimerRemaining()` and `qSTimerExpired()` , being the last one, the recurrent for most of the common timing applications. Finally, to get the current status of the STimer (check if is Armed or Disarmed) the `qSTimerStatus()` API should be used.

```
qClock_t qSTimerElapsed( const qSTimer_t * const obj )
```

```
qClock_t qSTimerRemaining( const qSTimer_t * const obj )
```

```
qBool_t qSTimerExpired( const qSTimer_t * const obj )
```

```
qBool_t qSTimerStatus( const qSTimer_t * const obj )
```

For this APIS, their only argument, its a pointer to the STimer object.

Return Value

For `qSTimerElapsed()`, `qSTimerRemaining()` returns the elapsed and remaining time specified in epochs respectively.
For `qSTimerExpired()`, returns `qTrue` when STimer expires, otherwise, returns `qFalse`. For a disarmed STimer, also returns `qFalse`.
For `qSTimerStatus()`, returns `qTrue` when armed, and `qFalse` for disarmed.

Usage example:

The example below shows a simple usage of this object, it is noteworthy that arming is performed once using the `FirstCall` flag. This prevents the timer gets re-armed every time the task run. After the timer expires, it should be disarmed explicitly.

```
void Example_Task(qEvent_t e){
    static qSTimer_t timeout = QSTIMER_INITIALIZER;
    if(e->FirstCall){
        /*Arming the stimer for  3.5 seg*/
         qSTimerSet(&timeout, 3.5);
    }

    /*non-blocking delay, true when timeout expires*/
    if(qSTimerExpired(&timeout)){
        /*
        TODO: Code when STimer expires
        */
        qSTimerDisarm(&timeout);
    }
    else return; /*Yield*/
}
```

3.2 Finite State Machines (FSM)

The state machine is one of the fundamental programming patterns. Designers use this approach frequently for solving complex engineering problems. State machines break down the design into a series of finite steps called "states". Each state performs some narrowly defined actions. Events, on the other hand, are the stimuli that cause the state to move or produce a transition between states.

In QuarkTS, states must be defined as functions taking a `qSMData_t` object and returning a `qSM_Status_t` value. An example it's shown in the following code snippet:

```
qSM_Status_t Example_State(qSMData_t m){
    /*
    TODO: State code
    */
    return qSM_EXIT_SUCCESS;
}
```

Here, the return value represents the finish status of the state, allowing only the values listed below.

- `qSM_EXIT_SUCCESS(-32768)`.
- `qSM_EXIT_FAILURE(-32767)`.
- Any other integer value between `-32766` and `32767`.

Every state machine has the concept of *current state* (P). This is the state that FSM currently occupies. At any given moment in time, the state machine can be in only a single state. The exit status can be handled with additional *sub-states*($S_{(b,s,u,f)}$) established at the moment of the FSM initialization. This workflow between the *current state* and the *sub-states* are better showed in the graph below:

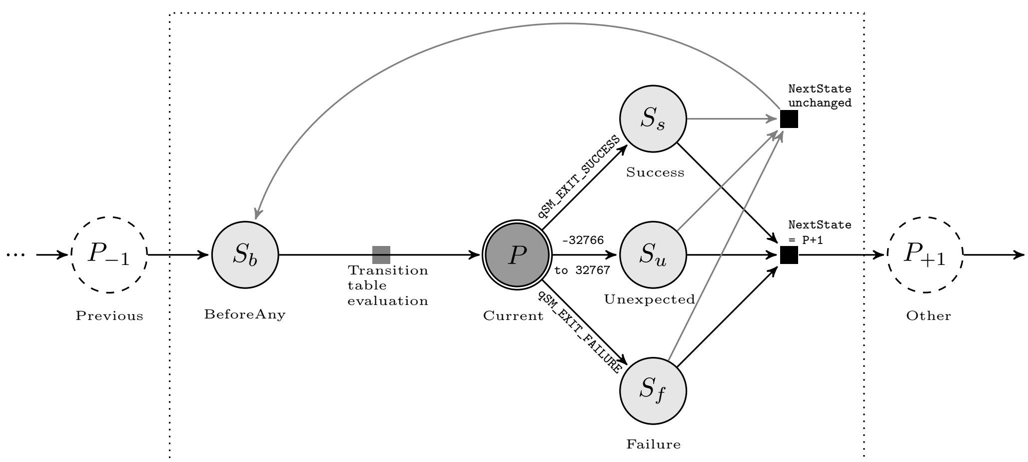

Figure 14: Sub-states evaluated after and before the *current* state

Figure 14 shows that this *sub-states* (if enabled) are implicitly linked in the workflow, to every state in the FSM. This *sub-states*, and the way they are called are described below:

- *BeforeAny* (S_b) : Performed in every FSM cycle. As its name implies, it is launched before the *current state* executes.

- *Success* (S_s) : This *sub-state* is conditioned to the exit status of the *current state*. If *current state* has exited with `qSM_EXIT_SUCCESS`, this sub-state will be launched after. Can be used to assert the actions of the *current state*, for example, if a certain state if controlling an output, the value over it will be written only, if the state has ended correctly.
- *Failure* (S_f): Same behavior as the previous one, but instead, is conditioned to the exit value `qSM_EXIT_FAILURE`. It can be used to handle exceptions.
- *Unexpected* (S_u): Same behavior as *Failure* and *Success*, but instead, is conditioned to an exit value between `-32766` and `32767`. The applications for this sub-state should be defined by the application writer.

3.2.1 Setting up a state machine : `qStateMachine_Init`

Like any other OS objects, a finite state machine (FSM) must be explicitly initialized before it can be used. FSMs are referenced by handles, which are variables of type `qSM_t`.

The `qStateMachine_Init()` API initialize the instance, set the initial state and conditioned sub-states.

```
qBool_t qStateMachine_Init( qSM_t * const obj, qSM_State_t InitState,
                            qSM_SubState_t SuccessState,
                            qSM_SubState_t FailureState,
                            qSM_SubState_t UnexpectedState,
                            qSM_SubState_t BeforeAnyState )
```

Parameters

- `obj` : A pointer to the FSM object.
- `InitState` : The first state to be performed. This argument is a pointer to a callback function, returning `qSM_Status_t` and with a `qSMData_t` variable as input argument.
- `SuccessState` : Sub-State performed after a state exits with `qSM_EXIT_SUCCESS`.
- `FailureState` : Sub-State performed after a state exits with `qSM_EXIT_FAILURE`.
- `UnexpectedState` : Sub-State performed after a state exits with any value between `-32766` and `32767`.
- `BeforeAnyState` : A state called before the normal state machine execution.

> *Note*: For every sub-state argument, a `NULL` value will act as a "disable" action.

Unlike normal states, a sub-state should not return anything, thus, the callback for substates should be written as:

```
void SubState_Example(qSMData_t m){
    /*
    TODO: Sub-State code
    */
}
```

3.2.2 Running a state machine : `qStateMachine_Run`

This API is used to execute the finite state machine. Only a single cycle is performed invoking the callback of the current active state, and of course, the available sub-states according to figure 14.

```
void qStateMachine_Run( qSM_t * const obj, void *Data )
```

Parameters

- `obj` : a pointer to the FSM object.
- `Data` : FSM associated data. Also can be used to represents the FSM arguments. All arguments must be passed by reference and cast to (`void *`). For multiple data, create a structure and pass a pointer to that structure.

3.2.3 Changing states and retrieving FSM data

Both, states and sub-states callbacks takes a `qSMData_t` object as input argument, that is basically a pointer to the FSM that invoke the state. The usage of this object its underlying to make the FSM move between states and additionally, get extra execution data. The provided fields are:

- `NextState` : The next state to be performed after the current state finish. The application writer should change this field to another state callback to produce a state transition in the next FSM's cycle.
- `PreviousState` (read-only): Last state seen in the flow chart.
- `LastState` (read-only): The last state executed.
- `PreviousReturnStatus` (read-only): The exit(or return) status of the previous state.
- `StateFirstEntry` (read-only): A flag that indicates that the state has just entered from another state.
- `Data` (read-only): State-machine associated data. If the FSM is running as a task, the associated event data can be queried thought this field. (here, a cast to `qEvent_t` is mandatory)

The developer is free to write and control state transitions. A state can jump to another by changing the `NextState` field. There is no way to enforce the state transition rules. Any transition is allowed at any time.

The code snippet below, show how this input argument should be used to produce a state transition and obtain additional information:

```
qSM_Status_t Some_State(qSMData_t m){
    qSM_Status_t RetValue = qSM_EXIT_SUCCESS;
    if( m->StateFirstEntry ){
        /*
            TODO : Do something at the first entry
        */
    }

    if ( EVENT_A_RECEIVED() ){ /*this its a state transition*/
        m->NextState = Other_State;
        /*it will be executed in the next cycle*/
        /*this transition have the higher priority*/
    }
    else if ( EVENT_B_RECEIVED() ){ /*this its a state transition*/
        m->NextState = m->PreviousState; /*return to the previous state*/
        /*it will be executed in the next cycle*/
    }
    else{
        RetValue = qSM_EXIT_FAILURE;
    }

    /*
        TODO : Whatever this state does
    */
    return RetValue;
}
```

3.2.4 Adding a state machine as a task : `qSchedulerAdd_StateMachineTask`

The best FSM running strategy is delegating it to a task. For this, the provided API should be used. Here, the task doesn't have a specific callback, instead, it will evaluate the active state of FSM, and later, all the other possible states in response to events that mark their own transition. The task will be scheduled to run every `Time` seconds in `qPeriodic` mode.

Using this API, the kernel will take care of the FSM by itself, so the usage of `qStateMachine_Init()` and `qStateMachine_Run()` can be omitted.

```
qBool_t qSchedulerAdd_StateMachineTask(
            qTask_t * const Task, qPriority_t Priority, qTime_t Time,
            qSM_t * const StateMachine, qSM_State_t InitState,
            qSM_SubState_t BeforeAnyState, qSM_SubState_t SuccessState,
            qSM_SubState_t FailureState, qSM_SubState_t UnexpectedState,
            qState_t InitialTaskState, void *arg )
```

Parameters

- `Task` : A pointer to the task node.
- `Priority` : The priority value. [0(min) - `Q_PRIORITY_LEVELS`(max)]
- `Time` : Execution interval defined in seconds (floating-point format). For immediate execution use the `qTimeImmediate` definition.
- `StateMachine` : A pointer to Finite State-Machine (FSM) object.
- `InitState` : The first state to be performed.
- `BeforeAnyState` : The sub-state called before any current state.
- `SuccessState` : Sub-State performed after a state exits with `qSM_EXIT_SUCCESS`.
- `FailureState` : Sub-State performed after a state exits with `qSM_EXIT_FAILURE`.
- `UnexpectedState` : Sub-State performed after a state exits with any value between `-32766` and `32767`.
- `TaskInitState` : Specifies the initial operational state of the task (`qEnabled`, `qDisabled`, `qAsleep` or `qAwake`).
- `arg` - Represents the task arguments. All arguments must be passed by reference and cast to `(void *)`.

> *Note*: For every sub-state argument, a `NULL` value will act as a disable action.

Now that a task is running a dedicated state-machine, the specific task event-info can be obtained in every state callback through the `Data` field of the `qSMData_t` argument. Check the example below:

```
qSM_Status_t Example_State(qSMData_t m){
    qEvent_t e = m->Data;
    /* Get the event info of the task that handle this state-machine*/
    if(e->FirstCall){
        /*
        TODO: Task initialization
        */
    }
    switch(e->Trigger){
        case byTimeElapsed:
            /* TODO: Code for this case */
        break;
        case byNotificationSimple:
            /* TODO: Code for this case */
        break;
        case byQueueCount:
            /* TODO: Code for this case */
        break;
        default: break;
    }
```

```
    /*
    TODO: State code
    */
    return qSM_EXIT_SUCCESS;
}
```

3.2.5 A demonstrative example

In this example, one press of the button turn on the LED, a second push of the button will make the LED blink and if the button is pressed again, the LED will turn off. Also, our system must turn off the LED after a period of inactivity. If the button hasn't been pressed in the last 10 seconds, the LED will turn off.

States (S_i) and transitions (t_i) are described as follows:

- S_1 : LED Off
- S_2 : LED On
- S_3 : LED Blink

- t_1 : Button press (rising edge)
- t_2 : Timeout (10S)

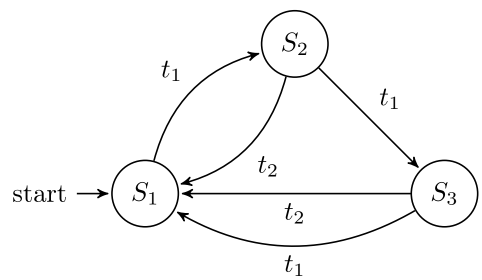

Figure 15: A simple FSM example with three states

To start the implementation, let's define the neccesary global variables...

```
qTask_t LED_Task; /*The task node*/
qFSM_t LED_FSM; /*The state-machine handler*/
```

Then, we define our states as the flow-diagram showed in figure 15.

```
qSM_Status_t State_LED_Off( qSMData_t m ){
    if( m->StateFirstEntry ){
        LED_OUTPUT = 0;
    }
    if( BUTTON_PRESSED ){
        m->NextState = State_LED_On;
    }
    return qSM_EXIT_SUCCESS;
}
/*---------------------------------------------------------------------*/
qSM_Status_t State_LED_On( qSMData_t m ){
    static qSTimer_t timeout;
    if( m->StateFirstEntry ){
        qSTimerSet( &timeout, 10.0 ); /*STimer gets armed*/
```

```
        LED_OUTPUT = 1;
    }
    if( qSTimerExpired( &timeout) ){ /*check if the timeout expired*/
        m->NextState = State_LED_Off;
    }
    if( BUTTON_PRESSED ){
        m->NextState = State_LED_Blink;
    }
    return qSM_EXIT_SUCCESS;
}
/*-----------------------------------------------------------------------*/
qSM_Status_t State_LED_Blink( qSMData_t m ){
    static qSTimer_t timeout;
    static qSTimer_t blinktime;
    if( m->StateFirstEntry ){
        qSTimerSet( &timeout, 10.0 );
    }
    if( qSTimerExpired( &timeout ) || BUTTON_PRESSED ){
        m->NextState = State_LED_Off;
    }
    if( qSTimerFreeRun( &blinktime, 0.5 ) ){
       LED_OUTPUT = !LED_OUTPUT;
    }
    return qSM_EXIT_SUCCESS;
}
```

Finally, we add the task to the scheduling scheme running the dedicated state machine.

Note: Remember that you must set up the scheduler before adding a task to the scheduling scheme.

```
qSchedulerAdd_StateMachineTask(&LED_Task, qHigh_Priority, 0.1, &LED_FSM,
                                State_LED_Off, NULL, NULL, NULL, NULL,
                                qEnabled, NULL);
```

3.2.6 Using a transition table

In this approach, the FSM is coded in a table relating states and signal-events. This is an elegant method to translate the FSM to actual implementation as the handling for every state and event combination is encapsulated in the table.

Current	*Signal*	*Next*
StateA	Signal1	StateB
StateB	Signal3	StateD
...	...	...
StateD	Signal6	StateA

Table 1: Transition table example

Here, the application writer get a quick picture of the FSM and the embedded software maintenance is also much more under control. A transition table is referenced through an object of type `qSM_TransitionTable_t` and should be explicitly installed in the target FSM with the corresponding entries, an n-sized array of `qSM_Transition_t` elements.

Events by the other hand, are represented in the table with an unsigned number of type `qSM_Signal_t`. Here, the designer is free to name event signals according the FSM application.

The API `qStateMachine_TransitionTableInstall()`, should be used to perform the transition table installation to the target FSM.

```
qBool_t qStateMachine_TransitionTableInstall( qSM_t * const obj,
            qSM_TransitionTable_t *table, qSM_Transition_t *entries,
            size_t NoOfEntries, qSM_Signal_t *AxSignals, size_t MaxSignals)
```

Parameters

- `obj` : A pointer to the FSM object.
- `table` : A pointer to the transition table instance.
- `entries` : The array of transitions (`qSM_Transition_t[]`).
- `NoOfEntries` : The number of transitions inside `entries`.
- `AxSignals` : A pointer to the memory area used for queuing signals. `qSM_Signal_t[]`.
- `MaxSignals` : The number of items inside `AxSignals`.

Return Value

Returns `qTrue` on success, otherwise returns `qFalse`;

State transitions are no limited to the specification of the transition table. A State callback owns the higher precedence to change an state. The application writer can use both, a transition table and direct `NextState` field manipulation in state callbacks to perform a transition to the FSM.

A special take care should be taken when the table becomes sparse, that is, there are many invalid state/event combinations, driving to a wastage of memory. There is also a memory penalty as the number of states and events grow. The application writer need to accurately account for this during initial design.

As shown in figure 14, an installed transition table is evaluated after the *Before-Any* sub-state, and later, the current state handler will be evaluated after the table sweep is completed. In brief, the state handler has a higher precedence over the table to perform state transitions.

3.2.6.1 Signals in transition tables:

When using transitions tables, signals are the selected event-abstraction to produce a transition from one state to another. Signals can be asynchronously delivered to the FSM, as they are handled internally with an exclusive FIFO queue.

To perform a signal delivering to FSMs with an installed transition table, the following API should be used:

```
qBool_t qStateMachine_SendSignal( qSM_t * const obj, qSM_Signal_t signal,
                                  qBool_t isUrgent )
```

Note : The FSM instance must have a transition table previously installed.

Parameters

- `obj` : A pointer to the FSM object.
- `signal` : The user-defined signal.
- `isUrgent` : If qTrue, the signal will be sent to the front of the transition-table queue.

Return Value

`qTrue` if the provided signal was successfully delivered to the FSM, otherwise return `qFalse`.

> A FSM signal-queue is only available through a transition table.

3.2.6.2 Signal responsiveness in state-machine tasks :

When using state-machine tasks, there is no relationship between transition-table signals and task events by default, this means that tasks cant be changed to the *qReady* if a signal is received, as consequence, the FSM can lose responsiveness to incoming signals because they are only served when the task is triggered by another kind of events.

To overcome this special problematic, the following OS API should be invoked:

```
qBool_t qScheduler_StateMachineTask_SignalConnect( qTask_t * const Task )
```

This API performs a kernel connection between the built-in signal-queue of a transition table and a task, allowing the OS to catch signals to produce a task event, allowing fast handling to incoming signals and of course, improving the transition-table functionality.

> This API should be only invoked after a transition-table installation.

> This API doesn't have any effect if the FSM is not attached to a task.

3.2.6.3 Demostrative example using a transition table The following example shows the implementation of the FSM presented in section 3.2.5 using a transition table approach.

Before getting started, the required variables should be defined:

```
/*define the FSM application event-signals*/
#define SIGNAL_BUTTON_PRESSED  ( (qSMSignal_t)1 )
#define SIGNAL_TIMEOUT         ( (qSMSignal_t)2 )

qTask_t LED_Task; /*The task node*/
qFSM_t LED_FSM; /*The state-machine handler*/
qSM_TransitionTable_t LED_FSM_TransTable; /*the FSM transition table*/
/*create the transition table entries with the desired FSM behavior*/
qSM_Transition_t LED_FSM_TransEntries[] = {
    { State_LED_Off, SIGNAL_BUTTON_PRESSED, State_LED_On },
    { State_LED_On, SIGNAL_TIMEOUT, State_LED_Off },
    { State_LED_On, SIGNAL_BUTTON_PRESSED, State_LED_Blink },
    { State_LED_Blink, SIGNAL_TIMEOUT, State_LED_Off },
    { State_LED_Blink, SIGNAL_BUTTON_PRESSED, State_LED_Off }
};
/*create the memory area for the signal-queue*/
qSMSignal_t LED_FSM_SignalArea[4];
/*the timeout object*/
qSTimer_t LED_FSM_Timeout = QSTIMER_INITIALIZER;
```

Then, we define the callback for the states. Here, the `BeforAny` sub-state is defined to send the timeout signal.

```
void SubState_LED_BeforeAny( qSMData_t m ){
    if( qSTimerExpired( &LED_FSM_Timeout ) ){
        qStateMachine_SendSignal( m, SIGNAL_TIMEOUT, qFalse );
    }
    return qSM_EXIT_SUCCESS;
}
/*---------------------------------------------------------------------*/
qSM_Status_t State_LED_Off( qSMData_t m ){
    if( m->StateFirstEntry ){
        LED_OUTPUT = 0;
    }
    return qSM_EXIT_SUCCESS;
}
/*---------------------------------------------------------------------*/
qSM_Status_t State_LED_On( qSMData_t m ){
    if( m->StateFirstEntry ){
        qSTimerSet( &LED_FSM_Timeout, 10.0 ); /*STimer gets armed*/
        LED_OUTPUT = 1;
    }
    return qSM_EXIT_SUCCESS;
}
/*---------------------------------------------------------------------*/
qSM_Status_t State_LED_Blink( qSMData_t m ){
    static qSTimer_t blinktime;
    if( m->StateFirstEntry ){
        qSTimerSet( &LED_FSM_Timeout, 10.0 );
    }
    if( qSTimerFreeRun( &blinktime, 0.5 ) ){
```

```
        LED_OUTPUT = !LED_OUTPUT;
    }
    return qSM_EXIT_SUCCESS;
}
```

In the previous code sniped, we assumed that `SIGNAL_BUTTON_PRESSED` can be delivered from either the interrupt context or another task.

To finish the setup, a task is added to handle the FSM and then, the transition table can be installed.

```
qSchedulerAdd_StateMachineTask( &LED_Task , qHigh_Priority , 0.1, &LED_FSM
                              ,State_LED_Off ,SubState_LED_BeforeAny,
                              NULL , NULL , NULL ,qEnabled , NULL);
qStateMachine_TransitionTableInstall( &LED_FSM, &LED_FSM_TransTable,
                                     LED_FSM_TransEntries, 5,
                                     LED_FSM_SignalArea, 4 );
/*improve the transition table signal-response*/
qScheduler_StateMachineTask_SignalConnect( &LED_Task );
```

3.2.7 Changing the FSM attributes in run-time

For this, use the API that are detailed below:

```
void qStateMachine_Attribute(
                        qSM_t * const obj, const qFSM_Attribute_t Flag ,
                        qSM_State_t  s, qSM_SubState_t subs )
```

- `obj` : A pointer to the FSM object.
- `Flag` : The attribute/action to be taken. Should be one of the following:
  - `qSM_RESTART` : Restart the FSM (here, the `s` argument must correspond to the init-state)
  - `qSM_CLEAR_STATE_FIRST_ENTRY_FLAG` : clear the entry flag for the current state if the `NextState` field doesn't change.
  - `qSM_FAILURE_STATE` : Set the failure sub-state.
  - `qSM_SUCCESS_STATE` : Set the success sub-state.
  - `qSM_UNEXPECTED_STATE` : Set the unexpected sub-state.
  - `qSM_BEFORE_ANY_STATE` : Set the sub-state executed before any state.
  - `qSM_UNINSTALL_TRANSTABLE` : To uninstall the transition table if available.
- `s` : The new value for state (only applies in `qSM_RESTART`). If not used, pass `NULL`.
- `subs` : The new value for sub-state (only apply in `qSM_FAILURE_STATE`, `qSM_SUCCESS_STATE`, `qSM_UNEXPECTED_STATE` and `qSM_BEFORE_ANY_STATE`. If not used, pass `NULL`.

3.3 Co-Routines

As showed in figure 16, a task coded as a Co-Routine, is just a task that allows multiple entry points for suspending and resuming execution at certain locations, this feature can bring benefits by improving the task cooperative scheme and providing a linear code execution for event-driven systems without complex state machines or full multi-threading.

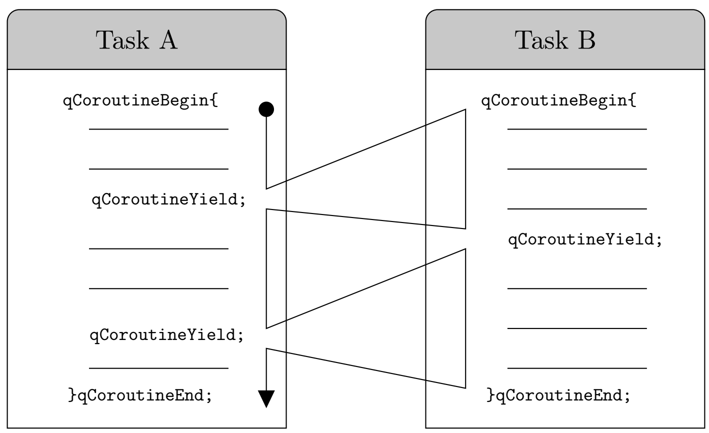

Figure 16: Coroutines in QuarkTS

The QuarkTS implementation uses the Duff's device approach, and is heavily inspired by the Simon Tatham's Co-Routines in C [3] and Adam Dunkels Protothreads [4]. This implementation does, however, impose slightly restrictions that are listed below:

Limitations and Restrictions:

- The stack of a Co-Routine is not maintained when a Co-Routine yields. This means variables allocated on the stack will loose their values. To overcome this, a variable that must maintain its value across a blocking call must be declared as `static`.
- Another consequence of sharing a stack, is that calls to API functions that could cause the Co-Routine to block, can only be made from the Co-Routine function itself - not from within a function called by the Co-Routine .
- The implementation does not permit yielding or blocking calls to be made from within a `switch` statement.

3.3.1 Coding a Co-Routine

The application writer just needs to create the body of the Co-Routine . This means starting a Co-Routine segment with `qCoroutineBegin` and end with `qCoroutineEnd` statements . From now on, yields and blocking calls from the Co-Routine scope are allowed.

```
void CoroutineTask_Callback(qEvent_t e){
    qCoroutineBegin{
        if( EventNotComing() ){
            qCoroutineYield;
        }
        DoTheEventProcessing();
    }qCoroutineEnd;
}
```

A `qCoroutineYield` call return the CPU control back to the scheduler but saving the execution progress. With the next task activation, the Co-Routine will resume the execution after the last `qCoroutineYield` statement.

All the Co-routine statements had *qCoroutine* or *qCR* appended at the beginning of their name.

Co-Routine statements can only be invoked from the scope of the Co-Routine.

3.3.2 Blocking calls

Blocking calls inside a Co-Routine should be made with the provided statements, all of them with a common feature: an implicit yield.

A widely used procedure, its wait for a fixed period of time. For this, the `qCoroutineDelay()` should be used .

```
qCoroutineDelay(qTime_t tDelay)
```

As expected, this statement makes an apparent blocking over the application flow, but to be precise, a yield is performed until the requested time expires, this allows other tasks to be executed until the blocking call finish. This *"yielding until condition meet"* behavior its the common pattern among the other blocking statements.

Another common blocking call is `qCoroutineWaitUntil()` :

```
qCoroutineWaitUntil( Condition )
```

This statement takes a `Condition` argument, a logical expression that will be performed when the coroutine resumes their execution. As mentioned before, this type of statement exposes the expected behavior, yielding until the condition met.

Optionally, the `Do-Until` structure are also provided to perform a multi-line job before the yield, allowing more complex actions to be performed after the Co-Routine resumes:

```
qCoroutineDo{
    /* Job : a set of instructions*/
}qCoroutineUntil( Condition );
```

Usage example:

```
void Sender_Task(qEvent_t e){
    static STimer_t timeout;
    qCoroutineBegin{
        Send_Packet();
        /*
           Wait until an ackowledgement has been received, or until
           the timer expires. If the timer expires, we should send
           the packet again.
        */
        qSTimerSet(&timeout, TIMEOUT_TIME);
        qCoroutineWaitUntil( PacketACK_Received() ||
                             qSTimerExpired(&timeout));
    }qCoroutineEnd;
}

void Receiver_Task(qEvent_t e){
    qCoroutineBegin{
        /* Wait until a packet has been received*/
        qCoroutineWaitUntil(Packet_Received());
        Send_Acknowledgement();
    }qCoroutineEnd;
}
```

3.3.3 Positional jumps

This feature provides positional local jumps, control flow that deviates from the usual Co-Routine call.

The complementary statements `qCoroutinePositionGet()` and `qCoroutinePositionRestore()` provide this functionality. The first, saves the Co-Routine state, at some point of their execution, into a `CRPos`, a variable of type `qCRPosition_t` , that can be used at some later point of program execution by `qCoroutinePositionRestore()` to restore the Co-Routine state to that saved by `qCoroutinePositionGet()` into `CRPos`. This process can be imagined to be a "jump" back to the point of program execution where `qCoroutinePositionGet()` saved the Co-Routine environment.

```
qCoroutinePositionGet( CRPos )
```

```
qCoroutinePositionRestore( CRPos )
```

And to resets the `CRPos` variable to the beginning of the Co-Routine, use :

```
qCoroutinePositionReset( CRPos )
```

3.3.4 Semaphores

This module implements counting semaphores on top of Co-Routines. Semaphores are a synchronization primitive that provide two operations: *wait* and *signal*. The *wait* operation checks the semaphore counter and blocks the Co-Routine if the counter is zero. The *signal* operation increases the semaphore counter but does not block. If another Co-Routine has blocked waiting for the semaphore that is signaled, the blocked Co-Routines will become runnable again.

Semaphores are referenced by handles, a variable of type `qCoroutineSemaphore_t` and must be initialized with `qCoroutineSemaphoreInit()` before any usage. Here, a value for the counter is required. Internally, semaphores uses an `unsigned int` to represent the counter, therefore the `Value` argument should be within range of this data-type.

```
qCoroutineSemaphoreInit( sem, Value )
```

To perform the *wait* operation, the `qCoroutineSemaphoreWait()` statement should be used. The wait operation causes the Co-routine to block while the counter is zero. When the counter reaches a value larger than zero, the Co-Routine will continue.

```
qCoroutineSemaphoreWait( sem )
```

Finally, `qCoroutineSemaphoreSignal()` carries out the *signal* operation on the semaphore. This signaling increments the counter inside the semaphore, which eventually will cause waiting Co-routines to continue executing.

```
qCoroutineSemaphoreSignal( sem )
```

Usage example:

The following example shows how to implement the bounded buffer problem using Co-Routines and semaphores. The example uses two tasks: one that produces items and other that consumes items.

Note that there is no need for a mutex to guard the `add_to_buffer()` and `get_from_buffer()` functions because of the implicit locking semantics of Co-Routines, so it will never be preempted and will never block except in an explicit `qCoroutineSemaphoreWait` statement.

```
#include "HAL.h"
#include "QuarkTS.h"
#include "AppLibrary.h"

#define NUM_ITEMS 32
#define BUFSIZE 8

qTask_t ProducerTask, ConsumerTask;
qCoroutineSemaphore_t mutex, full, empty;
/*============================================================*/
void ProducerTask_Callback(qEvent_t e){
  static int produced;

  qCoroutineBegin{
     for(produced = 0; produced < NUM_ITEMS; ++produced) {
        qCoroutineSemaphoreWait( &full );
        qCoroutineSemaphoreWait( &mutex );

        add_to_buffer( produce_item() );
```

```
        qCoroutineSemaphoreSignal( &mutex );
        qCoroutineSemaphoreSignal( &empty );
     }
  }qCoroutineEnd;
}
/*====================================================================*/
void ConsumerTask_Callback(qEvent_t e){{
  static int consumed;

  qCoroutineBegin{
      for(consumed = 0; consumed < NUM_ITEMS; ++consumed) {
        qCoroutineSemaphoreWait(pt, &empty);
        qCoroutineSemaphoreWait(pt, &mutex);

        consume_item(get_from_buffer());

        qCoroutineSemaphoreSignal(pt, &mutex);
        qCoroutineSemaphoreSignal(pt, &full);
      }
  }qCoroutineEnd;
}
/*====================================================================*/
void IdleTask_Callback(qEvent_t e){
    /*nothing to do*/
}
/*====================================================================*/
int main(void){
  HAL_Init();

  qSchedulerSetup( HAL_GetTick, 0.001, IdleTask_Callback, 10);
  qCoroutineSemaphoreInit(&empty, 0);
  qCoroutineSemaphoreInit(&full, BUFSIZE);
  qCoroutineSemaphoreInit(&mutex, 1);

  qSchedulerAdd_Task( &ProducerTask, ProducerTask_Callback,
                      qMedium_Priority, 0.1, qPeriodic, qEnabled,
                      NULL );
  qSchedulerAdd_Task( &ProducerTask, ProducerTask_Callback,
                      qMedium_Priority, 0.1, qPeriodic, qEnabled,
                      NULL );
  qSchedulerRun();
}
```

3.4 AT Command Line Interface

A command-line interface (CLI) is a way to interact directly with the software of an embedded system in the form of text commands and responses. It can see it as a typed set of commands to produce a result, but here, the commands are typed in real-time by a user through a specific interface, for example, UART, USB, LAN, etc.

A CLI is often developed to aid initial driver development and debugging. This CLI may become the interface (or one of the interfaces) used by a sophisticated end-user to interact with the product. Think of typing commands to control a machine, or perhaps for low-level access to the control system as a development tool, tweaking time-constants and monitoring low-level system performance during testing.

3.4.1 The parser

The provided development API, parse and handle input commands, following a simplified form of the extended AT-commands syntax.

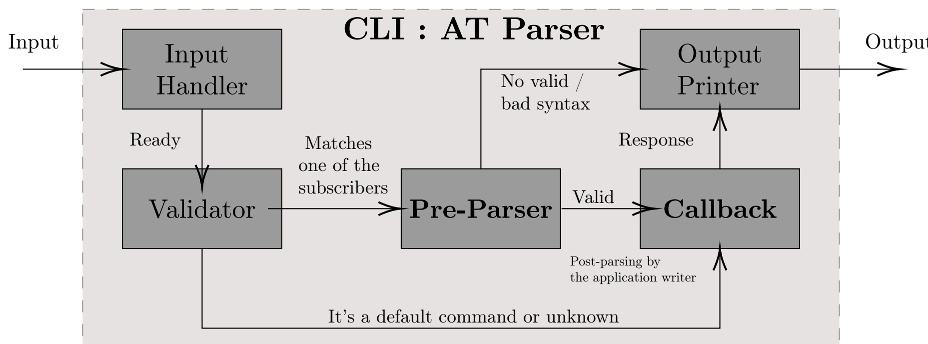

Figure 17: AT parser for a CLI implementation

As seen in figure 17, the parser has a few components described below:

- *Input Handler* : It is responsible for collecting incoming data from the *input* in the form of ASCII characters inside a buffer. When this buffer is ready by receiving an EOL(End-Of-Line) byte, it notifies the *validator* to perform the initial checks.
- *Validator*: Take the input string and perform three checks over it:
  1. The input matches one of the subscribed commands.
  2. The input matches one of the default commands.
  3. The input is unknown
- *Pre-Parser*: Takes the input if the *validator* asserts the first check. It is responsible for syntax validation and classification. Also prepares the input argument for the next component.
- *Callback or Post-Parser*: If input at the *pre-parser* it's valid, the respective command-callback is invoked. Here, the application writer is free to handle the command execution and the output response.
- *Output printer* : Takes all the return status of the previous components to print out a response at the output.

> Here, *Input* and *Output* should be provided by the application writer, for example, if a UART interface is chosen, the input should take the received bytes from an ISR and the output its a function to print out a single byte.

3.4.2 Supported syntax

This syntax is straightforward and the rules are provided below:

- All command lines must start with `AT` and end with an EOL character. By default, the parser uses the carriage return character. (We will use `<CR>` to represent a carriage return character in this document).
- AT commands are case-insensitive
- Only four types of AT commands are allowed:
  - **Acting** (`QATCMDTYPE_ACT`) : This is the simple type of commands which can be subscribed. Its normally used to execute the action that the command should do. This type don't take arguments or modifiers, for example,

    ```
    AT+CMD
    ```

  - **Read** (`QATCMDTYPE_READ`) : This type of command allows you to read or test a value already set for the parameter. Only one argument is allowed.

    ```
    AT+CMD?
    AT+CMD?PARAM1
    ```

  - **Test** (`QATCMDTYPE_TEST`) : These types of commands allow you to get the values that can be set for its parameters. No parameters are allowed here.

    ```
    AT+CMD=?
    ```

  - **Parameter Set** (`QATCMDTYPE_PARA`) : These types of commands allow n arguments to be passed for setting parameters, for example:

    ```
    AT+CMD=x,y
    ```

  If none of the types is given at the input, the command response will be `ERROR`

- The possible responses that can be output are:
  - **OK**: Indicates the successful execution of the command.
  - **ERROR**: A generalized message to indicate failure in executing the command.
  - **User-defined**: A custom output message defined by the application writer.
  - **UNKNOWN** : The input command its not subscribed.
  - **NOT ALLOWED** : The command syntax is not one of the allowed types.
  - **NONE** : No response.

All responses are followed by a `<CR><LF>`.

Errors generated during the execution of these AT commands could be due to the following reasons:

- Incorrect syntax/parameters of the AT command
- Bad parameters or not allowed operations defined by the application writer.

In case of an error, the string `ERROR` or `ERROR:<error_no>` are displayed.

3.4.3 Setting up an AT parser instance for the CLI

Before starting the CLI development, an AT parser instance must be defined; a data structure of type `qATParser_t` . The instance should be initialized using the `qATParser_Setup()` API .

A detailed description of this function is showed bellow:

```
qBool_t qATParser_Setup(
                    qATParser_t * const Parser, const qPutChar_t OutputFcn,
                    char *Input, const size_t SizeInput, char *Output,
                    const size_t SizeOutput, const char *Identifier,
                    const char *OK_Response, const char *ERROR_Response,
                    const char *NOTFOUND_Response, const char *term_EOL )
```

Parameters

- `Parser` : A pointer to the AT Command Parser instance.
- `OutputFcn` :The basic output-char wrapper function. All the parser responses will be printed-out through this function.
- `Input` : A memory location to store the parser input (mandatory)
- `SizeInput` : The size of the memory allocated in `Input`.
- `Output` : A memory location to store the parser output. If not used, pass `NULL`.
- `SizeOutput` : The size of the memory allocated in `Output`.
- `Identifier` : The device identifier string. This string will be printed-out after a call to the `AT_DEFAULT_ID_COMMAND`.
- `OK_Response` : The output message when a command callback returns `QAT_OK`. To use the default, pass `NULL`.
- `ERROR_Response` : The output message when a command callback returns `QAT_ERROR` or any `QAT_ERRORCODE(#)`. To use the default, pass `NULL`.
- `NOTFOUND_Response` : The output message when input doesn't match with any of the available commands. To use the default, pass `NULL`.
- `term_EOL` - The End Of Line string printed out after any of the parser messages. To use the default, pass `NULL`.

3.4.4 Subscribing commands to the parser

The AT Parser its able to subscribe any number of custom AT commands. For this, the `qATParser_CmdSubscribe()` API should be used.

This function subscribes the parser to a specific command with an associated callback function, so that next time the required command is sent to the parser input, the callback function will be executed.

The parser only analyzes commands that follow the simplified AT-Commands syntax already described in section 3.4.2.

```
qBool_t qATParser_CmdSubscribe( qATParser_t * const Parser,
                                qATCommand_t * const Command,
                                const char *TextCommand,
                                qATCommandCallback_t Callback,
                                qATParserOptions_t CmdOpt, void *param )
```

Parameters

- `Parser` : A pointer to the AT Command Parser instance.
- `Command` : A pointer to the AT command object.
- `TextCommand` : The string (name) of the command we want to subscribe to. Since this service only handles AT commands, this string has to begin by the `"at"` characters and should be in lower case.
- `Callback` : The handler of the callback function associated to the command. Prototype: `qATResponse_t xCallback(qATParser_t*, qATParser_PreCmd_t* )`
- `CmdOpt` : This flag combines with a bitwise 'OR' ('|') the following information:
  - `QATCMDTYPE_PARA` : `AT+cmd=x,y` is allowed. The execution of the callback function also depends on whether the number of argument is valid or not. Information about number of arguments is combined with a bitwise 'OR' : `QATCMDTYPE_PARA | 0xXY` , where `X` which defines maximum argument number for incoming command and `Y` which defines minimum argument number for incoming command.
  - `QATCMDTYPE_TEST` : `AT+cmd=?` is allowed.
  - `QATCMDTYPE_READ` : `AT+cmd?` is allowed.
  - `QATCMDTYPE_ACT` : `AT+cmd` is allowed.
- `param` : User storage pointer.

3.4.5 Writing a command callback

The command callback should be coded by the application writter. Here, the following prototype should be used:

```
qATResponse_t CMD_Callback(qATParser_t *parser, qATParser_PreCmd_t *param){

}
```

The callback takes two arguments and returns a single value. The first argument, it's just a pointer to the parser instance where the command its subscribed. From the callback context, it can be used to print out extra information as a command response.

The second, its the main parameter from the point of view of the application writer, and correspond to a data structure of type `qATParser_PreCmd_t` . The fields inside, are filled by the *Pre-Parser* component and gives information about the detected command, like type, number of arguments and the subsequent string after the command text. These fields are described as follows:

- `Command` : A pointer to the calling AT Command object.
- `Type` : The command type.
- `StrData` : The string data after the command text.
- `StrLen` : The length of `StrData`.
- `NumArgs` : Number of arguments, only available if `Type = QATCMDTYPE_PARA`.

The return value (an enum of type `qATResponse_t`) determines the response showed by the *Output printer* component. The possible allowed values are:

- `QAT_OK` : as expected, print out the `OK` string.
- `QAT_ERROR` : as expected, print out the `ERROR` string.
- `QAT_ERRORCODE(no)` : Used to indicate an error code. This code is defined by the application writer and should be a value between 1 and 32766. For example, a return value of `QAT_ERRORCODE(15)`, will print out the string `ERROR:15`.
- `QAT_NORESPONSE` : No response will be printed out.

A simple example of how the command callback should be coded is showed below:

```
qATResponse_t CMD_Callback(qATParser_t *parser, qATParser_PreCmd_t *param){
  qATResponse_t Response = QAT_NORESPONSE;
  switch(param->Type){
    case QATCMDTYPE_PARA:
      Response = QAT_OK;
      break;
    case QATCMDTYPE_TEST:
      Response = QAT_OK;
      break;
    case QATCMDTYPE_READ:
        strcpy( parser->Output , "Test");
      Response = QAT_OK;
      break;
    case QATCMDTYPE_ACT:
      Response = QAT_OK;
      break;
    default:
      Response = QAT_ERROR;
      break;
  }
  return Response;
}
```

3.4.6 Handling the input

Input handling are simplified using the provided APIs. The `qATParser_ISRHandler()` and `qATParser_ISRHandlerBlock()` functions are intended to be used from the interrupt context. This avoids any kind of polling implementation and allows the CLI application to be designed using an event-driven pattern.

```
qBool_t qATParser_ISRHandler( qATParser_t * const Parser, char c )
```

```
qBool_t qATParser_ISRHandlerBlock( qATParser_t * const Parser, char *data,
                                   const size_t n )
```

Both functions feed the parser input, the first one with a single character and the second with a string. The application writer should call one of these functions from the desired hardware interface, for example, from a UART receive ISR.

Parameters

- `Parser` : Both APIs take a pointer to the AT parse instance.

for `qATCommandParser_ISRHandler` :

- `c` : The incoming byte/char to the input.

for `qATParser_ISRHandlerBlock` :

- `data` : The incoming string.
- `n` : The length of the `data` argument.

Return Value

`qTrue` when the parser is ready to process the input, otherwise return `qFalse`.

If there are no intention to feed the input from the ISR context, the APIs `qATParser_Raise` or `qATParser_Exec` can be called at demand from the base context.

```
qBool_t qATParser_Raise( qATParser_t * const Parser, const char *cmd )
```

```
qATResponse_t qATParser_Exec( qATParser_t * const Parser, const char *cmd )
```

As expected, both functions send the string to the specified parser. The difference between both APIs is that `qATParser_Raise()` sends the command through the input, marking it as ready for parsing and acting as the *Input handler* component. The `qATParser_Exec()`, on the other hand, executes the components of *Pre-parsing* and *Post-parsing* bypassing the other components, including the *Output printer*, so that it must be handled by the application writer.

Parameters:

- `Parser` : A pointer to the ATParser instance.
- `cmd` : The command string, including arguments if required.

Return value:

For `qATParser_Raise()`, `qTrue` if the command was successfully raised, otherwise returns `qFalse`.

For `qATParser_Exec()`, the same value returned by the respective callback function. If the input string doesn't match any of the subscribed commands, returns `QAT_NOTFOUND`. If the input syntax is not allowed, returns `qAT_NOTALLOWED`.

> *Note*: All functions involved with the component *Input-handler*, ignores non-graphic characters and cast any uppercase to lowercase.

3.4.7 Running the parser

The parser can be invoked directly using the `qATParser_Run()` API. Almost all the components that make up the parser are performed by this API, except for the *Input Handler*, that should be managed by the application writer itself.

```
qBool_t qATParser_Run( qATParser_t * const Parser )
```

In this way, the writer of the application must implement the logic that leads this function to be called when the *input-ready* condition is given.

The simple approach for this is to check the return value of any of the input feeders APIs and set a notification variable when they report a ready input. Later in the base context, a polling job should be performed over this notification variable, running the parser when their value is true, then clearing the value after to avoid unnecessary overhead.

The recommended implementation is to leave this job be handled by a task instead of coding the logic to know when the parser should run. For this, the `qSchedulerAdd_ATParserTask()` is provided. This API add a task to the scheduling scheme running an AT Command Parser and treated as an event-triggered task. The address of the parser instance will be stored in the `TaskData` storage-Pointer.

```
qBool_t qSchedulerAdd_ATParserTask( qTask_t * const Task,
                                    qATParser_t *Parser,
                                    qPriority_t Priority )
```

Parameters

- `Task` : A pointer to the task node.
- `Parser` : A pointer to the AT Command Parser.
- `Priority` : Task priority Value. [0(min) - `Q_PRIORITY_LEVELS`(max)]

After invoked, both parser and task are linked together in such a way that when an *input-ready* condition is given, a notification event is sent to the task launching the parser components. As the task is event-triggered, there is no additional overhead and the writer of the application can assign a priority value to balance the application against other tasks in the scheduling scheme.

3.4.8 Retrieving arguments inside a command-callback

The following APIs should be only invoked from a command-callback context:

```
char* qATParser_GetArgString( const qATParser_PreCmd_t *param, qINT8_t n,
                              char* out)
```

Get the argument parsed as string from the incoming AT command.

Parameters:

- `param` : A pointer to the pre-parser instance(only available from the AT-Command callback).
- `n` : The number of the argument.
- `out` : Array in memory where to store the resulting null-terminated string.

Return value:

Same as `out` on success, otherwise returns `NULL`.

Next APIs takes the same arguments:

```
char* qATParser_GetArgPtr( const qATParser_PreCmd_t *param, qINT8_t n )
```

To get the pointer where the desired argument starts.

```
int qATParser_GetArgInt( const qATParser_PreCmd_t *param, qINT8_t n )
```

To get the argument parsed as integer from the incoming AT command.

```
qFloat32_t qATParser_GetArgFlt( const qATParser_PreCmd_t *param, qINT8_t n )
```

To get the argument parsed as float from the incoming AT command.

```
qUINT32_t qATParser_GetArgHex( const qATParser_PreCmd_t *param, qINT8_t n )
```

To get the HEX argument parsed as `qUINT32_t` from the incoming AT command.

Parameters:

- `param` : A pointer to the pre-parser instance(only available from the AT-Command callback).
- `n` : The number of the argument.

Return value:

For `qATParser_GetArgPtr()`, a pointer to the desired argument. `NULL` pointer if the argument is not present.
For `qATParser_GetArgInt()`, the argument parsed as integer. Same behavior of `qAtoI`. If argument not found returns 0.
For `qATParser_GetArgFlt()`, the argument parsed as `float`. Same behavior of `qAtoF`. If argument not found returns 0.
For `qATParser_GetArgHex()`, the HEX argument parsed as `qUINT32_t`. Same behavior of `qXtoU32`. If argument not found returns 0.

3.5 Memory Management

Memory can be allocated using the standard C library malloc() and free() functions, but they may not be suitable in most embedded applications because they are not always available on small microcontrollers or their implementation can be relatively large, taking up valuable code space. Additionally, some implementations can suffer from fragmentation.

To get around this problem, the OS provides its own memory allocation API. When the application requires RAM, instead of calling *malloc()*, call `qMalloc()` . When RAM is being freed, instead of calling *free()*, use `qFree()` . Both functions have the same prototype as the standard C library counterparts.

3.5.1 Principle of operation

The allocation scheme works by subdividing a static array into smaller blocks and using the *First-Fit* approach (see figure 18).

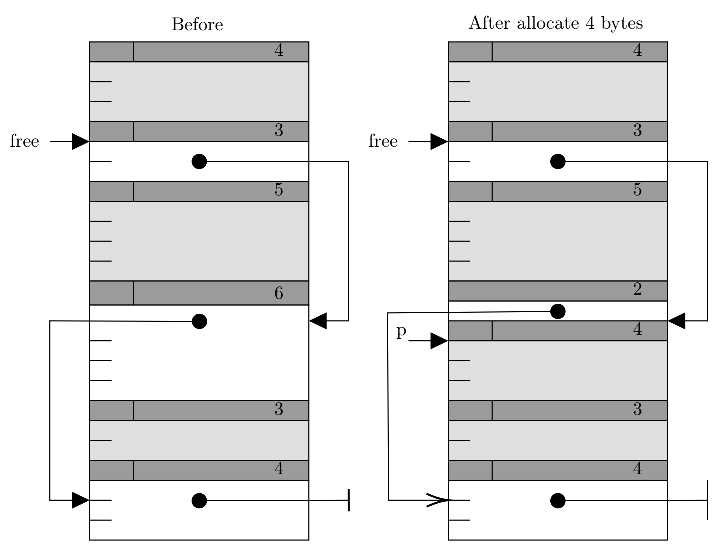

Figure 18: First-fit allocation policy

If Adjacent free blocks are available, the implementation combines them into a single larger block, minimizing the risk of fragmentation, making it suitable for applications that repeatedly allocate and free different sized blocks of RAM.

> *Note*: Because memory is statically declared, it will make the application appear to consume a lot of RAM, even before any memory has been allocated from it.

> *Warning*: All the memory management APIs are NOT interrupt-safe. Use these APIs only from the base context.

3.5.2 Memory pools

A memory pool is a kernel object that allows memory blocks to be dynamically allocated from a user-designated memory region. Instead of typical of pools with fixed-size blocks allocation, the pools in QuarkTS can be of any size, thereby the user is responsible for selecting the appropriate memory pool to allocate data with the same size.

The *default* memory management unit resides in a memory pool object. Also called the *default pool*. The total amount of available heap space in the default memory pool is set by `Q_DEFAULT_HEAP_SIZE`, which is defined in `qconfig.h`.

Besides the *default* pool, any number of additional memory pools can be defined. Like any other kernel object in QuarkTS, memory pools are referenced by handles, a variable of type `qMemoryPool_t` and should be initialized before any usage with the `qMemoryPool_Init()` API function.

```
qBool_t qMemoryPool_Init( qMemoryPool_t * const mPool, void* Area,
                          size_t size )
```

Parameters

- `mPool` : A pointer to the memory pool instance.
- `Area` : A pointer to a memory region (`qUINT8_t`) statically allocated to act as Heap of the memory pool. The size of this block should match the `size` argument.
- `size` : The size of the memory block pointed by `Area`.

To perform operations in another memory pool, besides the *default* pool, an explicit switch should be performed using `qMemoryPool_Select()` . Here, a pointer to the target pool should be passed as input argument. From now on, every call to `qMalloc()`, or `qFree()` will run over the newly selected memory pool. To return to the *default pool*, a new call to `qMemoryPool_Select()` is required passing `NULL` as input argument.

```
void qMemoryPool_Select( qMemoryPool_t * const mPool )
```

To keep track of the memory usage, the `qHeapGetFreeSize()` API function returns the number of free bytes in the current memory pool at the time the function is called.

```
size_t qHeapGetFreeSize(void)
```

Usage example:

```
#include <stdio.h>
#include <stdlib.h>
#include "QuarkTS.h"
#include "HAL.h"
#include "Core.h"

qTask_t taskA;
qMemoryPool_t another_heap;
void taskA_Callback(qEvent_t e);

void taskA_Callback(qEvent_t e){
    int *xdata = NULL;
    int *ydata = NULL;
    int *xyoper = NULL;
    int n = 20;
    int i;

    xyoper = (int*)qMalloc( n*sizeof(int) );
    xdata = (int*)qMalloc( n*sizeof(int) );
    qMemoryPool_Select( &another_heap ); /*change the memory pool*/
    /*ydata will point to a segment allocated in another pool*/
    ydata = (int*)qMalloc( n*sizeof(int) );

    /*use the memory if could be allocated*/
    if( xdata && ydata && xyoper ){
        for(i=0; i<n; i++){
            xdata[i] = GetXData();
            ydata[i] = GetYData();
            xyoper[i] = xdata[i] * ydata[i];

        }
        UseTheMemmory(xyoper);
    }
    else{
        qTraceMessage("ERROR:ALLOCATION_FAIL");
    }

    qFree( ydata );
    qMemoryPool_Select( NULL ); /*return to the default pool*/
    qFree( xdata );
    qFree( xyoper );
}


int main(void){
    char area_another_heap[512]={0};
    qSetDebugFcn( OutPutChar );
    /*Create a memory heap*/
    qMemoryPool_Init( &another_heap, area_another_heap, 512);
    qSchedulerSetup(HAL_GetTick, 0.001, IdleTaskCallback, 10);
```

```
    qSchedulerAddxTask(&taskA, taskA_Callback, qLowest_Priority,
                       0.1, qPeriodic, qEnabled, &mtxheap);
    qSchedulerRun();
}
```

3.6 Trace and debugging

QuarkTS include some basic macros to print out debugging messages. Messages can be simple text or the value of variables in specific base-formats. To use the trace macros, an single-char output function must be defined using the `qSetDebugFcn()` macro.

```
qSetDebugFcn(qPutChar_t fcn)
```

Where `fcn`, its a pointer to the single-char output function following the prototype:

```
void SingleChar_OutputFcn(void *sp, const char c){
    /*
    TODO : print out the c variable usign the
    selected peripheral.
    */
}
```

The body of this user-defined function should have a hardware-dependent code to print out the `c` variable through a specific peripheral.

3.6.1 Viewing variables

For viewing or tracing a variable (up to 32-bit data) through debug, one of the following macros are available:

```
qTraceVar(Var, DISP_TYPE_MODE)
qTraceVariable(Var, DISP_TYPE_MODE)
```

```
qDebugVar(Var, DISP_TYPE_MODE)
qDebugVariable(Var, DISP_TYPE_MODE)
```

Parameters:

- `Var` : The target variable.
- `DISP_TYPE_MODE` : Visualization mode. It must be one of the following parameters(case sensitive): `Bool`, `Float`, `Binary`, `Octal`, `Decimal`, `Hexadecimal`, `UnsignedBinary`, `UnsignedOctal`, `UnsignedDecimal`, `UnsignedHexadecimal`.

The only difference between `qTrace_` and `Debug`, is that `qTrace_` macros, print out additional information provided by the `__FILE__`, `__LINE__` and `__func__` built-in preprocessing macros, mostly available in common C compilers.

3.6.2 Viewing a memory block

For tracing memory from a specified target address, one of the following macros are available:

```
qTraceMem(Pointer, BlockSize)
qTraceMemory(Pointer, BlockSize)
```

Parameters:

- `Pointer` : The target memory address.
- `Size` : Number of bytes to be visualized.

Hexadecimal notation it's used to format the output of these macros.

3.6.3 Usage

In the example below, an UART output function is coded to act as the printer. Here, the target MCU is an ARM-Cortex M0 with the UART1 as the selected peripheral for this purpose.

```
void putUART1(void *sp, const char c){
    /* hardware specific code */
    UART1_D = c;
    while ( !(UART1_S1 & UART_S1_TC_MASK) ) {} /*wait until is done*/
}
```

As seen above, the function follows the required prototype. Later, in the main thread, a call to the `qSetDebugFcn()` is used to set up the output-function.

```
int main(void){
   qSetDebugFcn(putUART1);
   ...
   ...
}
```

After that, trace macros will be available for use.

```
void IO_TASK_Callback(qEvent_t e){
   static qUINT32_t Counter = 0;
   float Sample;
   ...
   ...
   qTraceMessage("IO TASK running...");
   Counter++;
   qTraceVariable( Counter, UnsignedDecimal );
   Sample = SensorGetSample();
   qTraceVariable( Sample, Float);
   ...
   ...
}
```

4 Utility APIs

4.1 Byte sized buffers

```
void qBSBuffer_Init( qBSBuffer_t * const obj, volatile qUINT8_t *buffer,
                     const size_t length )
```

Initialize the byte-sized buffer.

Parameters:

- `obj` : A pointer to the byte-sized buffer object
- `buffer` : Block of memory or array of data.
- `length` : The size of `buffer`(Must be a power of two)

```
qBool_t qBSBuffer_Put( qBSBuffer_t * const obj, const qUINT8_t data )
```

Adds an element of data to the byte-sized buffer.

Parameters:

- `obj` : A pointer to the byte-sized buffer object
- `data` : The data to be added.

Return value:

`qTrue` on success, otherwise returns `qFalse`.

```
qBool_t qBSBuffer_Read( qBSBuffer_t * const obj, void *dest, const size_t n )
```

Gets `n` data from the byte-sized buffer and removes them.

Parameters:

- `obj` : A pointer to the byte-sized buffer object
- `dest` : The location where the data will be written.

Return value:

`qTrue` on success, otherwise returns `qFalse`.

```
qBool_t qBSBuffer_Get( qBSBuffer_t * const obj, qUINT8_t *dest )
```

Gets one data-byte from the front of the byte-sized buffer, and remove it.

Parameters:

- `obj` : A pointer to the byte-sized buffer object
- `dest` : The location where the data will be written.

Return value:

`qTrue` on success, otherwise returns `qFalse`.

```
qUINT8_t qBSBuffer_Peek( const qBSBuffer_t * const obj )
```

Looks for one byte from the head of the byte-sized buffer without removing it.

Parameters:

- `obj` : A pointer to the byte-sized buffer object

Return value:

Byte of data, or zero if nothing in the buffer.

```
qBool_t qBSBuffer_Empty( const qBSBuffer_t * const obj )
```

Query the empty status of the byte-sized buffer.

Parameters:

- `obj` : A pointer to the byte-sized buffer object

Return value:

`qTrue` if the byte-sized buffer is empty, `qFalse` if it is not.

```
qBool_t qBSBuffer_IsFull( const qBSBuffer_t * const obj )
```

Query the full status of the byte-sized buffer.

Parameters:

- `obj` : A pointer to the byte-sized buffer object

Return value:

`qTrue` if the byte-sized buffer is full, `qFalse` if it is not.

```
size_t qBSBuffer_Count( const qBSBuffer_t * const obj )
```

Query the number of elements in the byte-sized buffer.

Parameters:

- `obj` : A pointer to the byte-sized buffer object.

Return value:

Number of elements in the byte-sized buffer.

4.2 Input groups for edge-checking

```
qBool_t qEdgeCheck_Initialize( qIOEdgeCheck_t * const Instance,
                               const qCoreRegSize_t RegisterSize,
                               const qClock_t DebounceTime )
```

Initialize an I/O edge-check instance.

Parameters:

- `Instance` : A pointer to the I/O edge-check object.
- `RegisterSize` : The specific-core register size: `QREG_8BIT`, `QREG_16BIT` or `QREG_32BIT`(default).
- `DebounceTime` : The specified time (in epochs) to bypass the bounce of the input nodes.

Return value:

`qTrue` on success, otherwise returns `qFalse`.

```
qBool_t qEdgeCheck_InsertNode( qIOEdgeCheck_t * const Instance,
                               qIONode_t * const Node, void *PortAddress,
                               const qBool_t PinNumber )
```

Inserts an I/O node to the edge-check instance.

Parameters:

- `Instance` : A pointer to the I/O edge-check object.
- `Node` : A pointer to the input-node object.
- `PortAddress` : The address of the core PORTx-register to read the levels of the specified `PinNumber`.
- `PinNumber` : The specified pin to read from `PortAddress`.

Return value:

`qTrue` on success, otherwise returns `qFalse`.

```
qBool_t qEdgeCheck_Update( qIOEdgeCheck_t * const Instance )
```

Update the status of all nodes inside the I/O edge-check instance (non-blocking call).

Parameters:

- `Instance` : A pointer to the I/O edge-check object.

Return value:

`qTrue` on success, otherwise returns `qFalse`.

```
qBool_t qEdgeCheck_GetNodeStatus( const qIONode_t * const Node )
```

Query the status of the specified input-node.

Parameters:

- `Instance` : A pointer to the I/O edge-check object.

Return value:

The status of the input node : `qTrue`, `qFalse`, `qRising`, `qFalling` or `qUnknown`.

4.3 Generic lists

The provided list implementation uses a generic *doubly-linked* approach in which each node apart from storing its data has two links. The first link points to the previous node in the list and the second link points to the next node in the list. The first node of the list has its previous link pointing to `NULL`, similarly the last node of the list has its next node pointing to `NULL`.
The list data-structure, referenced thought an object of type `qList_t` also has a *head* and a *tail* pointer, to allow fast operations on boundary nodes.

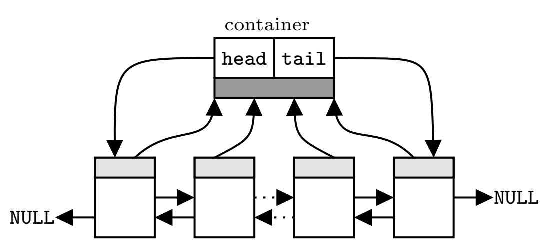

Figure 19: Doubly-linked list implementation

Nodes should be an user-defined data structure of any number of members, however, they must be specially defined to be compatible with the provided APIs. All the user-defined nodes must have the `qNode_MinimalFields` definition on top of the structure. An example is shown below:

```
typedef struct{
    qNode_MinimalFields;
    int a;
    int b;
    float y;
}userdata_t;
```

With this special type definition on all custom data, the application writer can take advantage of this powerful data structure. The following APIs are provided for lists management:

```
void qList_Initialize( qList_t * const list )
```

Must be called before a list is used. This initializes all the members of the list object.

Parameters:

- `list` : Pointer to the list being initialised.

```
qBool_t qList_Insert( qList_t *const list, void * const node,
                      const qListPosition_t position ){
```

Insert an item into the list.

Parameters:

- `list` : Pointer to the list.
- `node` : A pointer to the node to be inserted.
- `position` : The position where the node will be inserted. Could be `qList_AtFront`, `qList_AtBack` or any other index number where the node will be inserted after.

  Note: If the index exceeds the size of the list, the node will be inserted at the back.

  Note: If the list is empty, the node will be inserted as the first item.

Return value:

`qTrue` if the item was successfully added to the list, otherwise returns `qFalse`.

```
void* qList_Remove( qList_t * const list, void * const node,
                    const qListPosition_t position )
```

Remove an item from the list.

Parameters:

- `list` : Pointer to the list.
- `node` : A pointer to the node to be deleted (to ignore pass `NULL`).
- `position` : The position of the node that will be deleted. Could be `qList_AtFront`, `qList_AtBack` or any other index number.

  Note: If the `node` argument is supplied, the removal will be only effective if the data is member of the list. If ignored or the data is not a member of the list, this function will use the `position` instead as index for removal.

  Note: If the index exceeds the size of the list, the last node will be removed.

Return value:

A pointer to the removed node. `NULL` if removal can be performed.

```
qBool_t qList_IsMember( const qList_t *const list, const void *const node )
```

Check if the node is member of the list.

Parameters:

- `list` : Pointer to the list.
- `node` : A pointer to the node .

Return value:

`qTrue` if the node belongs to the list, `qFalse` if it is not.

```
void* qList_GetFront( const qList_t *const list ){
```

Get a pointer to the front item of the list.

Parameters:

- `list` : Pointer to the list.

Return value:

A pointer to the front node. `NULL` if the list is empty.

```
void* qList_GetBack( const qList_t *const list )
```

Get a pointer to the back item of the list.

Parameters:

- `list` : Pointer to the list.

Return value:

A pointer to the front node. `NULL` if the list is empty.

```
qBool_t qList_IsEmpty( const qList_t * const list )
```

Check if the list is empty.

Parameters:

- `list` : Pointer to the list.

Return value:

`qTrue` if the list is empty, `qFalse` if it is not.

```
size_t qList_Length( const qList_t * const list )
```

Get the number of items inside the list.

Parameters:

- `list` : Pointer to the list.

Return value:

The number of items of the list.

```
qBool_t qList_Move( qList_t *const destination, qList_t *const source,
                    const qListPosition_t position )
```

Moves(or merge) the entire list pointed by `source` to the list pointed by `destination` at location specified by `position`. After the move operation, this function leaves empty the list pointed by `source`.

Parameters:

- `destination` : Pointer to the list where the `source` nodes are to be moved.
- `source` : Pointer to the source list to be moved.
- `position` : The position where `source` list will be inserted. Could be `qList_AtFront`, `qList_AtBack` or any other index number where the list will be inserted after.

Return value:

`qTrue` if the move operation is performed successfully, otherwise returns `qFalse`.

```
qBool_t qList_ForEach( qList_t *const list, qListNodeFcn_t Fcn,
                       void *arg, qListDirection_t dir, void *NodeOffset )
```

Operate on each element of the list.

Parameters:

- `list` : Pointer to the list.
- `Fcn` : The function to perform over the node.

  Should have this prototype:

  `qBool_t Function( void* Node, void *arg, qList_WalkStage_t stage )`

  where the `stage` argument indicates the loop progress and should be checked by the application writer to perform the specific operations over the list. This variable can take the following values:

  - `QLIST_WALKINIT` : When the loop is about to start. In this case, A `NULL` value will be passed in the node pointer.
  - `QLIST_WALKTHROUGH` : When the loop is traversing the list.
  - `QLIST_WALKEND` : When the loop has finished. In this case, A `NULL` value will be passed in the node pointer

  By default, `Function` should return `qFalse`. If a `qTrue` value is returned, the walk through loop will be terminated.

- `arg` : Argument passed to `Fcn`.
- `dir` : Use one of the following options:
  - `QLIST_FORWARD` : to walk through the list forwards.
  - `QLIST_BACKWARD` to walk through the list backwards.
- `NodeOffset` : If available, the list walk through will start from this node. To ignore, pass `NULL`.

Return value:

`qTrue` if the walk through was early terminated, otherwise returns `qFalse`.

```
qBool_t qList_Sort(qList_t * const list,
                   qBool_t (*CompareFcn)(const void *n1, const void *n2) )
```

Sort the double linked list using the `CompareFcn` function to determine the order. The sorting algorithm used by this function compares pairs of adjacent nodes by calling the specified `CompareFcn` function with pointers to them as arguments. The sort is performed only modifying node's links without data swapping, improving performance if nodes have a large storage.

Note: The function modifies the content of the list by reordering its elements as defined by CompareFcn.

Parameters:

- `list` : Pointer to the list.
- `CompareFcn` : Pointer to a function that compares two nodes. This function is called repeatedly by `qList_Sort` to compare two nodes. It shall follow the following prototype: `qBool_t (*CompareFcn)(void *node1, void *node2)`

  Taking two pointers as arguments (both converted to (`const void*`). The function defines the order of the elements by returning a Boolean data, where a `qTrue` value indicates that element pointed by `node1` goes after the element pointed to by `node2`

Return value:

`qTrue` if at least one reordering is performed over the list.

```
qBool_t qList_IteratorSet( qListIterator_t *iterator, qList_t *const list,
                           void *NodeOffset, qListDirection_t dir)
```

Setup an instance of the given iterator to traverse the list.

Parameters:

- `iterator` : Pointer to the iterator instance.
- `list` : Pointer to the list.
- `NodeOffset` : The start offset-node. To ignore, pass `NULL`.
- `dir` : Use one of the following options:
  - `QLIST_FORWARD` : to go in forward direction.
  - `QLIST_BACKWARD` : to go in backward direction.

Return value:

`qTrue` on success. Otherwise returns `qFalse`.

```
void* qList_IteratorGetNext( qListIterator_t *iterator )
```

Get the current node available in the iterator. After invoked, iterator will be updated to the next node.

Parameters:

- `iterator` : Pointer to the iterator instance.

Return value:

Return the next node or `NULL` when no more nodes remain in the list.

```
qBool_t qList_Swap( void *node1, void *node2 )
```

Swap two nodes that belongs to the same list by changing its own links.
Note: The container list will be updated if any node is part of the boundaries.

Parameters:

- `node1` : Pointer to the first node.
- `node2` : Pointer to the second node.

Return value:

`qTrue` if the swap operation is performed. Otherwise returns `qFalse`.

4.4 Response handler

```
void qResponseInitialize( qResponseHandler_t * const obj, char *xLocBuff,
                          size_t nMax )
```

Initialize the instance of the response handler object.

Parameters:

- `obj` : A pointer to the response handler object
- `xLocBuff` : A pointer to the memory block where the desired response will remain.
- `nMax` : The size of memory block pointed by `xLocBuff`

```
void qResponseReset( qResponseHandler_t * const obj )
```

Reset the response handler.

Parameters:

- `obj` : A pointer to the response handler object

```
qBool_t qResponseReceived( qResponseHandler_t * const obj,
                           const char *Pattern, size_t n )
```

Non-blocking response check.

Parameters:

- `obj` : A pointer to the response handler object.
- `Pattern` : The data to be checked in the receiver ISR
- `n` : The length of the data pointer by `Pattern` . If `Pattern` its string, set `n` to zero(0) to auto-compute the length.

Return value:

`qTrue` if there is a response acknowledge, otherwise returns `qFalse`.

```
qBool_t qResponseReceivedWithTimeout( qResponseHandler_t * const obj,
                                      const char *Pattern,
                                      size_t n, qTime_t t )
```

Non-blocking response check with timeout.

Parameters:

- `obj` : A pointer to the response handler object.
- `Pattern` : The data to be checked in the receiver ISR
- `n` : The length of the data pointer by `Pattern` . If `Pattern` its string, set `n` to zero(0) to auto-compute the length.
- `obj` : The timeout value in seconds.

Return value:

`qTrue` if there is a response acknowledge and `qTimeoutReached` if timeout expires, otherwise returns `qFalse`.

```
qBool_t qResponseISRHandler( qResponseHandler_t * const obj,
                             const char rxchar )
```

ISR receiver for the response handler.

Parameters:

- `obj` : A pointer to the response handler object.
- `rxchar` : The byte-data from the receiver.

Return value:

`qTrue` when the response handler object match the request from `qResponseReceived()`.

4.5 Miscellaneous

```
qTime_t qClock2Time(const qClock_t t)
```

Convert the specified input time(epochs) to time(seconds).

Parameters:

- `t` : The time in epochs

Return value:

Time `t` in seconds.

```
qClock_t qTime2Clock(const qTime_t t)
```

Convert the specified input time(seconds) to time(epochs).

Parameters:

- `t` : The time in seconds

Return value:

Time `t` in epochs.

```
void qSwapBytes(void *data, size_t n)
```

Invert the endianess for `n` bytes of the specified memory location.

Parameters:

- `data` : A pointer to block of data.
- `n` : Number of bytes to swap.

```
qBool_t qCheckEndianness(void)
```

Check the system endianess.

Return value:

`qTrue` if little-endian, otherwise returns `qFalse`.

```
void qOutputRaw(qPutChar_t fcn, void* storagep, void *data,
                size_t n, qBool_t AIP)
void qInputRaw(qGetChar_t fcn, void* storagep, void *data,
                size_t n, qBool_t AIP)
```

Wrapper methods to write(`qOutputRaw`) or read(`qInputRaw`) `n` RAW data through `fcn`.

Parameters:

- `fcn` : The basic output or input byte function.
- `storagep` : The storage pointer passed to `fcn`.
- `data`: The data to be written or readed.
- `n` : Number of bytes to be writte or readed.
- `AIP` : Pass `qTrue` to auto-increment the storage-pointer.

```
void qOutputString(qPutChar_t fcn, void* storagep, const char *s,
                   qBool_t AIP)
```

Wrapper method to write a string through `fcn`.

Parameters:

- `fcn` : The basic output byte function.
- `storagep` : The storage pointer passed to `fcn`.
- `s` : The string to be written.
- `AIP` : Pass `qTrue` to auto-increment the storage-pointer.

```
void qU32toX(qUINT32_t value, char *str, int8_t n)
```

Converts an unsigned integer value to a null-terminated string using the 16 base and stores the result in the array given by `str` parameter. `str` should be an array long enough to contain any possible value.

Parameters:

- `value` : Value to be converted to string.
- `str` : Array in memory where to store the resulting null-terminated string.
- `n` : The number of chars used to represent the value in `str`.

```
qUINT32_t qXtoU32(const char *s)
```

Converts the input string `s` consisting of hexadecimal digits into an unsigned integer value. The input parameter `s` should consist exclusively of hexadecimal digits, with optional whitespaces. The string will be processed one character at a time, until the function reaches a character which it doesn't recognize (including a null character).

Parameters:

- `s` : The hex string to be converted.

Return value:

The numeric value in `qUINT32_t`.

```
qFloat64_t qAtoF(const char *s)
```

Parses the C string `s`, interpreting its content as a floating point number and returns its value as a double. The function first discards as many whitespace characters (as in `isspace`) as necessary until the first non-whitespace character is found. Then, starting from this character, takes as many characters as possible that are valid following a syntax resembling that of floating point literals, and interprets them as a numerical value. The rest of the string after the last valid character is ignored and has no effect on the behavior of this function.

Parameters:

- `s` : The string beginning with the representation of a floating-point number.

Return value:

On success, the function returns the converted floating point number as a double value.
If no valid conversion could be performed, the function returns zero (0.0).
If the converted value would be out of the range of representable values by a `double`, it causes undefined behavior.

```
int qAtoI(const char *s)
```

Parses the C-string `s` interpreting its content as an integral number, which is returned as a value of type `int`. The function first discards as many whitespace characters (as in `isspace`) as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many base-10 digits as possible, and interprets them as a numerical value. The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function. If the first sequence of non-whitespace characters in `s` is not a valid integral number, or if no such sequence exists because either `s` is empty or it contains only whitespace characters, no conversion is performed and zero is returned.

Parameters:

- `s` : The string beginning with the representation of a integer number.

Return value:

On success, the function returns the converted integral number as an int value.
If the converted value would be out of the range of representable values by an `int`, it causes undefined behavior.

```
char* qUtoA(qUINT32_t num, char* str, qUINT8_t base)
```

Converts an unsigned value to a null-terminated string using the specified base and stores the result in the array given by `str` parameter. `str` should be an array long enough to contain any possible value: `(sizeof(int)*8+1)` for radix=2, i.e. 17 bytes in 16-bits platforms and 33 in 32-bits platforms.

Parameters:

- `num` : Value to be converted to a string.
- `str` : Array in memory where to store the resulting null-terminated string.
- `base` : Numerical base used to represent the value as a string, between 2 and 36, where 10 means decimal base, 16 hexadecimal, 8 octal, and 2 binary.

Return value:

A pointer to the resulting null-terminated string, same as parameter `str`.

```
char* qItoA(qINT32_t  num, char* str, qUINT8_t base)
```

Converts an integer value to a null-terminated string using the specified `base` and stores the result in the array given by `str` parameter. If `base` is 10 and value is negative, the resulting string is preceded with a minus sign (-). With any other base, value is always considered unsigned.

`str` should be an array long enough to contain any possible value: `(sizeof(int)*8+1)` for radix=2, i.e. 17 bytes in 16-bits platforms and 33 in 32-bits platforms.

Parameters:

- `num` : Value to be converted to a string.
- `str` : Array in memory where to store the resulting null-terminated string.
- `base` : Numerical base used to represent the value as a string, between 2 and 36, where 10 means decimal base, 16 hexadecimal, 8 octal, and 2 binary.

Return value:

A pointer to the resulting null-terminated string, same as parameter `str`.

```
char* qFtoA(qFloat32_t f, char *str, qUINT8_t precision)
```

Converts a float value to a formatted string.

Parameters:

- `f` : Value to be converted to a string.
- `str` : Array in memory where to store the resulting null-terminated string.
- `precision` : Desired number of significant fractional digits in the string. (The max allowed precision is `MAX_FTOA_PRECISION=10`)

Return value:

A pointer to the resulting null-terminated string, same as parameter `str`.

```
char* qBtoA(qBool_t num, char *str)
```

Converts a boolean value to a null-terminated string. Input is considered true with any value different to zero (0). `str` should be an array long enough to contain the output

Parameters:

- `num` : Value to be converted to a string.
- `str` : Array in memory where to store the resulting null-terminated string.

Return value:

A pointer to the resulting null-terminated string, same as parameter `str`.

```
qBool_t qIsNan(qFloat32_t f)
```

Determines if the given floating point number `f` is a not-a-number(`NaN`) value.

Parameters:

- `f` : Floating point value(32bits).

Return value:

`qTrue` if input is `NaN`, otherwise `qFalse`.

```
qBool_t qIsInf(qFloat32_t f)
```

Determines if the given floating point number `f` is positive or negative infinity.

Parameters:

- `f` : Floating point value(32bits).

Return value:

`qTrue` is argument has an infinite value, otherwise `qFalse`.

4.6 Additional macros

`qBitsSet(Register, Bits)` : Sets (writes a 1 to) the bits indicated by the `Bits` mask in the numeric variable `Register`.

`qBitsClear(Register, Bits)` : Clears (writes a 0 to) the bits indicated by the `Bits` mask in the numeric variable `Register`.

`qBitSet(Register, Bit)` : Sets (writes a 1 to) the `n-Bit` in the numeric variable `Register`.

`qBitClear(Register, Bit)` : Clears (writes a 0 to) the `n-Bit` in the numeric variable `Register`.

`qBitRead(Register,Bit)` : Reads the `n-Bit` of the numeric variable `Register`.

`qBitToggle(Register,Bit)` : Invert the state of the `n-Bit` in the numeric variable `Register`.

`qBitWrite(Register, Bit, Value)` : Writes the `Value` to the `n-Bit` in the numeric variable `Register`.

`qByteMakeFromBits(b7,b6,b5,b4,b3,b2,b1,b0)` : Merge the bits from the most significant bit `b7` to the least significant bit `b0` into a single byte.

`qByteHighNibble(Register)` : Extracts the high-order (leftmost) nibble of a byte.

`qByteLowNibble(Register)` : Extracts the low-order (rightmost) nibble of a byte.

`qByteMergeNibbles(H,L)` : Merge the high(`H`) and low(`L`) nibbles into a single byte.

`qWordHighByte(Register)` : Extracts the high-order (leftmost) byte of a word.

`qWordLowByte(Register)` : Extracts the low-order (rightmost) byte of a word.

`qWordMergeBytes(H,L)` : Merge the high(`H`) and low(`L`) bytes into a single word.

`qDWordHighWord(Register)` : Extracts the high-order (leftmost) word of a Dword.

`qDWordLowWord(Register)` : Extracts the low-order (rightmost) word of a Dword.

`qDWordMergeWords(H,L)` : Merge the high(`H`) and low(`L`) words into a single DWord.

`qClip(X, Max, Min)` : Gives `X` for `Min<=X<=Max`, `Min` for `X<Min` and `Max` for `X>Max`.

`qClipUpper(X, Max)` : Gives `X` for `X<=Max` and `Max` for `X>Max`.

`qClipLower(X, Min)` : Gives `X` for `X>=Min` and `Min` for `X<Min`.

`qIsBetween(X, Low, High)` : Returns `true` if the value in `X` is between `Low` and `High`, otherwise returns `false`.

`qMin(a,b)` : Returns the smaller of `a` and `b`.

`qMax(a,b)` : Returns the greater of `a` and `b`.

`qMins2Time(t)` : Converts to seconds the time value in `t` given in minutes.

`qHours2Time(t)` : Converts to seconds the time value in `t` given in hours.

`qDays2Time(t)` : Converts to seconds the time value in `t` given in days.

`qWeeks2Time(t)` : Converts to seconds the time value in `t` given in weeks.

References

[1] M.J. Pont. *Patterns for Time-Triggered Embedded Systems.* Addison-Wesley. ACM Press, 2001.

[2] Charles E.; Rivest Ronald L.; Stein Clifford Cormen, Thomas H.; Leiserson. *"Section 6.5: Priority queues". Introduction to Algorithms (2nd ed.).* MIT Press. McGraw-Hill, 1990.

[3] Simon Tatham. Coroutines in C. `https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html`. Accessed: 2010-09-30.

[4] Adam Dunkels. Protothreads. `http://dunkels.com/adam/pt/index.html`. Accessed: 2010-09-30.

Index