User Manual

valid for v4.9.1

J. Camilo Gómez C.

July 30, 2019

Contents

1 Background

1.1 About the RTOS

QuarkTS its a simple non-preemptive real-time OS with a quasi-static scheduler for embedded multi-tasking applications. Its simplifed 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 tasks, its that they are NOT preemptable. Once a task starts its execution, it owns the CPU in base level (non-interrupt) until it returns. In other words, except when the CPU is servicing an event in 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 this basic capabilities, it has a small memory footprint that make 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. The following cores has been powered with QuarkTS successfully:

- ARM cores(ATMEL, STM32, LPC, Kinetis, Nordic and others)
- AVR
- ColdFire
- PIC (16F, 18F, PIC24, dsPIC, 32MX, 32MZ)
- MSP430
- 8051
- HCS12
- x86

1.1.2 Coding standard and naming convention

- All the QuarkTS implementation follow the C89 standard.
- The core source files try to conform most of the MISRA coding standard guidelines, however, certain deviations are allowed for reasons of performance.
- Functions, macros, enum values and data-types are prefixed `q.` (i.e. `qFunction`, `qEnumValue`, `QCONSTANT`, `qType_t`, ...)

- The _t suffix its used to denote a type name.
- In line with MISRA guides, unqualified standard char types are only permitted to hold ASCII characters.
- In line with MISRA guides, variables of type `char *` are only permitted to hold pointers to ASCII strings.
- Other than the pre-fix, macros used for constants are written in all upper case.
- 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.
- 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`)	52
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`)	9
qQueues handling and related APIS	544
A qQueue object (`qQueue_t`)	20
Memory management	407
A memory pool (`qMemoryPool_t`)	28
The AT Command Parser	1724
An AT-Parser instance (`qATParser_t`)	60
An AT command object (`qATCommand_t`)	16
Remaining utilities	2980

Although the kernel does not use dynamically-allocated resources internally, the application writer can create any 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 this two scenarios:

- When tick already provided: The reference is supplied by the HAL of the device. 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 `qSchedulerSysTick()` 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 your first call to the QuarkTS API. `qSchedulerSetup()` prepares the kernel instance, set the reference clock, define the Idle-Task callback and allocates the stack for the internal queue.

```
void qSchedulerSetup(qGetTickFcn_t TickProvider,
                     qTime_t ISRTick,
                     qTaskFcn_t IDLE_Callback,
                     uint8_t QueueSize)
```

Parameters

- `TickProvider` : The function that provides the tick value. If the user application uses the `qSchedulerSysTick()` 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 : `uint32_t TickProviderFcn(void)`

- `ISRTick` :This parameter specifies the ISR background timer period in seconds(floating-point format).

- `IDLE_Callback` : Callback function for the idle task. If not used, pass `NULL` as argument.

- `QueueSize` : Size of the priority queue. This argument should be an integer number greater than zero.

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 "DeviceHAL.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){
    qSchedulerSysTick();
}

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

A task is a node concept that links together:

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

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.

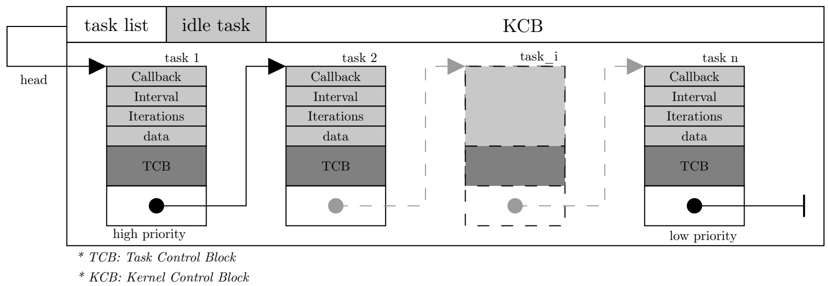

Figure 1: Tasks scheme

For execution purposes, the tasks are linked into execution chains (see figure 1), which are processed by the scheduler in priority order. Each task performs its function via a callback function and each of them are 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 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 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 scheduler, ensuring that at least, one task that is able to run. Aditionally, the scheduler 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 don'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 *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) - 255(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 state of the task (`qEnabled` or `qDisabled`).
- `arg` - Represents the task arguments. All arguments 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, internal iteration counter decreases until reach the zero. If another set of iterations is needed, user should set the number of iterations again and resume the task explicitly.
3. Tasks which 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 *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 becomes 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 he 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 *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 its 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 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.

The action will be performed 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 States and scheduling rules

Task states are classified into the four 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 higher precedence is running.
- **qRunning** : The task is currently being executed.
- **qSuspended** : The task doesn't take part on what is going on. Normally this state is taken after the *qRunning* state or when the task don't reach the *qReady* state.

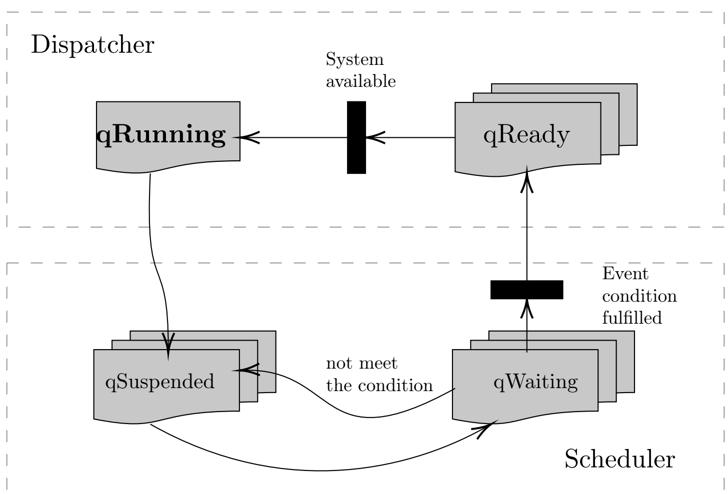

Figure 2: Task states

Except for the idle task, a task exists in one of this states. As the real-time embedded system runs, each task moves from one state to another, according to the logic of a simple finite state machine (FSM). Figure 2 illustrates the typical flowchart used by QuarkTS to handle 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 task does a block anywhere during the *qRunning* state. Based in the *round-robin* fashion, each ready task runs in turn only from the linked list or the priority-queue, here, the priority parameter set its position in the list statically and in the queue in a dynamic way.

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. 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 to the scheduling scheme first has the highest precedence.

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 in the following order:

1. Queued notifications (higher precedence)
2. Time elapsed
3. Events from an attached queue
   (a) Is full
   (b) Element count
   (c) Receiver
   (d) Is empty
4. Simple notifications (lower precedence)

All the listed events are later detailed in section 2.

1.7 Getting started

Unpack the source files (`QuarkTS.h` and `QuarkTS.c`) and copy it into your project folder. Then include the header file and setup the instance of the kernel using the `qSchedulerSetup()` function inside the main thread. The code bellow shows a common initialization in the main source file.

File: `main.c`

```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.

Task 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 proyect 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 tasks handling 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 tip, 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
    qSchedulerSysTick(); //
}

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,
                      120, 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 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 current 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, this critical sections are handled quickly as possible.

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

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

```
void qSchedulerSetInterruptsED(void (*Restorer)(uint32_t),
                               uint32_t (*Disabler)(void));
```

Parameters:

- `Restorer` :The function with hardware specific code that enables or restores 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 `uint32_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 *Task, 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 *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 *Task, qPriority_t Value)
```

Set/Change the task priority value.

Parameters:

- `Task` : A pointer to the task node.
- `Value` : Priority Value. [0(min) - 255(max)].

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

Set/Change the task callback function.

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

Set the task state (Enabled or Disabled). .

Parameters:

- `Task` : A pointer to the task node.
- `State` : `qEnabled` or `qDisabled`

```
void qTaskSetData(qTask_t *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 *Task)
```

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

Parameters:

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

```
uint32_t qTaskGetCycles(const qTask_t *Task)
```

Retrieve the number of task activation's.

Parameters:

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

Return value:

A `uint32_t` value containing the number of task activations.

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

Retrieve the enabled/disabled state.

Parameters:

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

Return value:

`qTrue` if the task in on Enabled state, otherwise returns `qFalse`.

```
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 Configuration macros

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

`Q_SETUP_TIME_CANONICAL`	*Default:* `undef`. If defined, the kernel assumes the timing base to 1mS(1KHz). So all time specifications for tasks and STimers must be set in mS. Can be used to remove the floating-point operations when dealing with time. (In some systems, can significantly reduce the memory usage).
`Q_SETUP_TICK_IN_HERTZ`	*Default:* `undef`. If defined, the timing base will be taken as frequency(Hz) instead of period(S) by `qSchedulerSetup()` (In some systems, can significantly reduce the memory usage).
`Q_BYTE_SIZED_BUFFERS`	*Default:* `defined`. Used to enable or disable the usage of Byte-sized buffers.
`Q_MEMORY_MANAGER`	*Default:* `defined`. 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_QUEUES`	*Default:* `defined`. Used to enable or disable the queues APIs for communication to tasks.
`Q_PRIORITY_QUEUE`	*Default:* `defined`. Used to enable or disable the kernel priority queue for notifications.
`Q_AUTO_CHAINREARRANGE`	*Default:* `defined`. If defined, the kernel perform a new scheme sorting when task priorities are changed in run-time.
`Q_TRACE_VARIABLES`	*Default:* `defined`. 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:* `defined`. Used to enable of disable the extended output of trace macros.
`Q_ATCOMMAND_PARSER`	*Default:* `defined`. Used to enable or disable the CLI AT parser module.
`Q_TASK_COUNT_CYCLES`	*Default:* `defined`. 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:* `undef`. Used to enable or disable the scientific notation in ASCII to float conversions.

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 3).
- 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 3). 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.

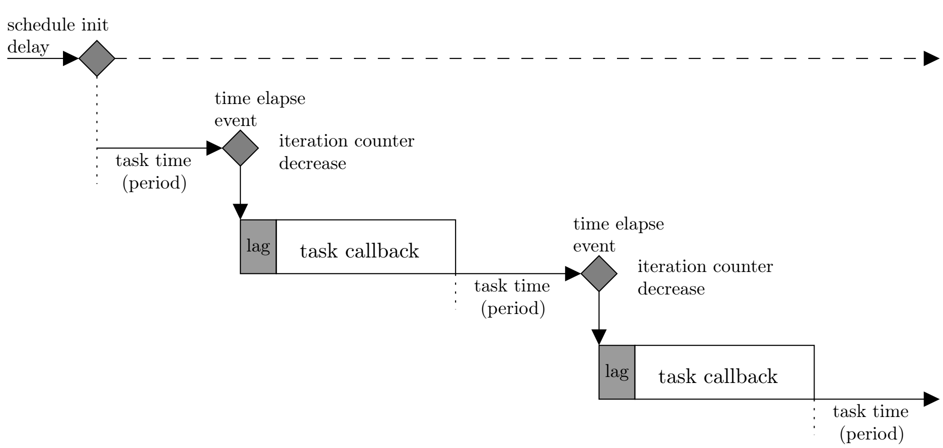

Figure 3: Inherit time lag

- 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 4).

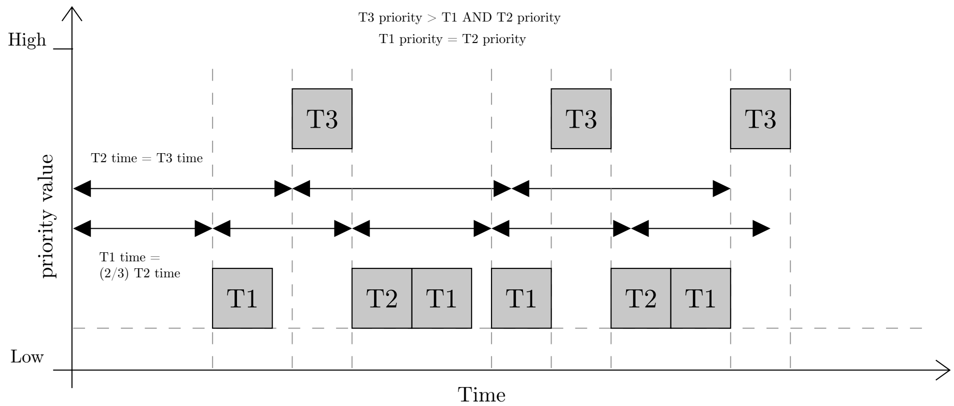

Figure 4: Priority scheduling example with three (3) tasks attached triggered by time-elapsed events

2.2 Asynchronous events and inter-task communication

Applications existing in heavy environments(like handling multiple peripherals and I/Os) or tasks with a high level of interaction, must 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 and system-generated events.

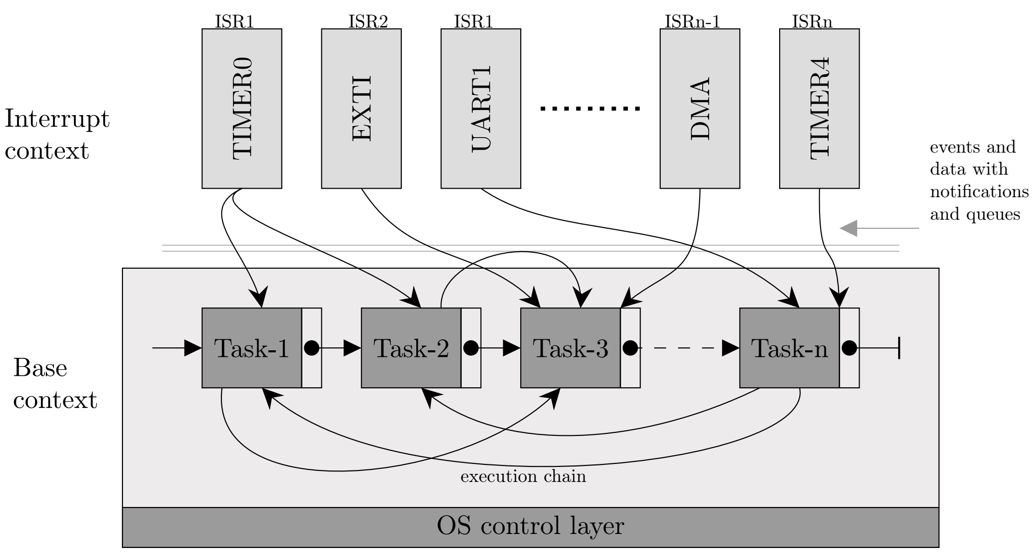

Figure 5: Heavy cooperative environment

If you take a look to figure 5, you can see this kind of heavy interaction, where a tasks and ISRs gets involved by sending events and data to each other.

For example, the interrupts catch external events, but sometimes the event-handling requires heavy processing that can overload the ISR and consequently, the system can lose future events. In this scenario, a task can handle events that need high processing, so we are taking the notification from the interrupt-level and making the actions in the base-level. Another common scenario is when a task is performing a specific job and another task must be awaken to perform some activities when the other task finish.
This kind of scenarios requires some ways in which tasks can communicate with each other. For this, QuarkTS 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. If used, they comes as asynchronous events with specific triggers and event data.

The OS API provides the following features for task communication:

2.2.1 Notifications

The notifications allow tasks 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 data to the receiving task. This is depicted in figure 6.

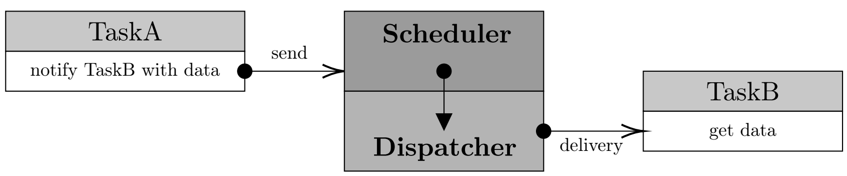

Figure 6: A notification used to send an event directly from one task to another

2.2.1.1 Simple notifications: Each task note has a 8-bit notification value which is initialised to zero when task is added to the scheme. The `qTaskSendNotification()` API 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, 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 are later decreased.

```
qBool_t qTaskSendNotification(qTask_t *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 catch 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 is the right solution instead 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 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 7 illustrates this behavior.

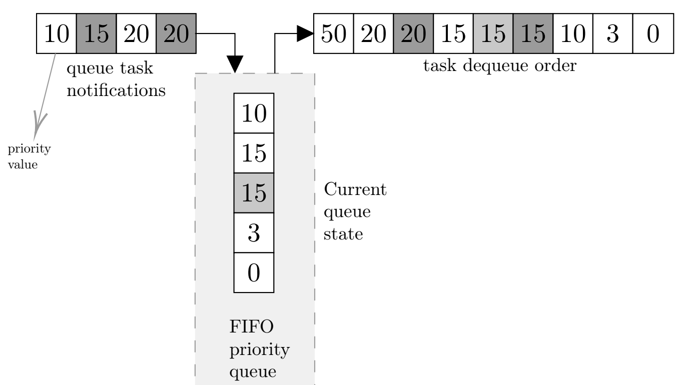

Figure 7: Priority-queue behavior

The scheduler always checks the queue state, 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*.

```
void qTaskQueueNotification(qTask_t *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 has the higher precedence.

Figure 8, 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.

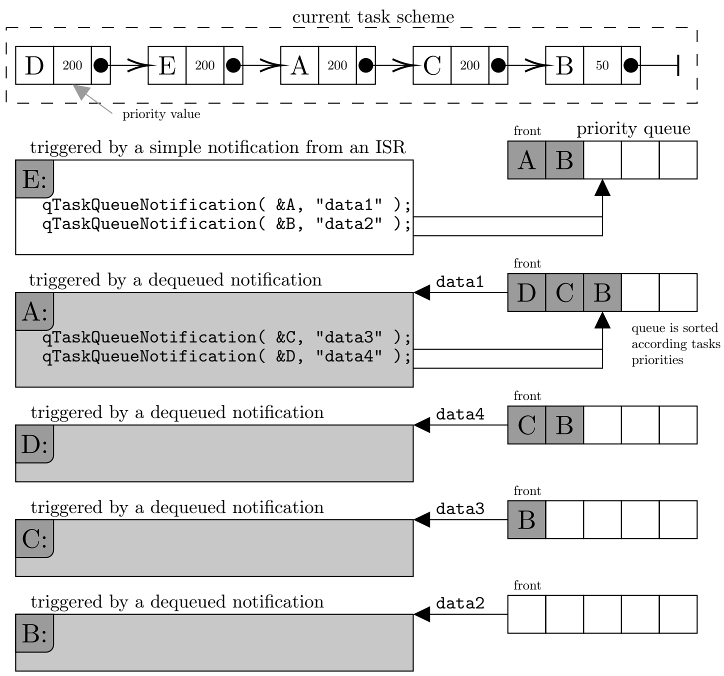

Figure 8: Priority-queue example

> 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(void *eventdata,
                                    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` if the event could be spread in all tasks. Otherwise `qFalse`.

This function spread 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.

2.2.3 Queues

A queue its a linear data structure with simple operations based on the FIFO (First In First Out) principle. 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.

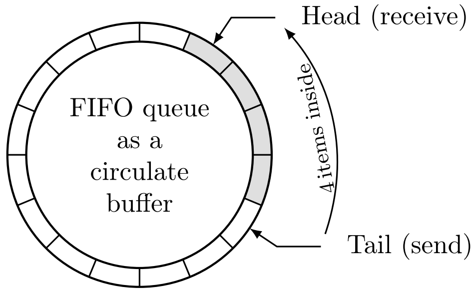

Figure 9: qQueues conceptual representation

As showed in figure 9, 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 is also another applications for queues as serialize many data streams into one receiving streams (multiple tasks to a single task) or vice-versa (single task to multiple tasks)

> 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.

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 and queues). 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 is already defined in the callback function, and 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 eight(8) 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.
- `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 only have a `NULL` value when the trigger is *byTimeElapsed* or *byPriority*.
- `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 handle all the notifications by itself (simple or queueded), 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 show a ISR to task communication. Two interrupts send notifications to a single task with specific event data. The receiver task (taskA) after a 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 "Core_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;
    }
}
/*==================================================================*/
```

```
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, NULL, 10);
    qSchedulerAdd_EventTask(&taskA, taskA_Callback, qLowest_Priority, NULL);
    qSchedulerAdd_EventTask(&taskA, taskA_Callback, qLowest_Priority, NULL);
    qSchedulerRun();
}
```

2.4.2 Creating a queue : `qQueueCreate`

A queue must be explicitly created before it can be used.
This 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 must be statically allocated at compile time and should be provided by the application writer.

```
qBool_t qQueueCreate(qQueue_t *obj, void* DataArea,
                     qSize_t ItemSize, qSize_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
- `ElementSize` : Size of one element in the data block
- `ElementCount` : The maximum number of items the queue can hold at any one time.

2.4.3 Performing queue operations

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

```
qBool_t qQueueSendToFront(qQueue_t *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 *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 by a task. For this, the following API is provided:

```
qBool_t qTaskAttachQueue(qTask_t *Task, qQueue_t *Queue,
                         const qRBLinkMode_t Mode, uint8_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 front of the queue will be received automatically in every trigger and a pointer to it, will be available in the `EventData` field of `qEvent_t` structure. For the other modes, the `EventData` field will be a pointer to the queue that triggered the event.

2.4.5 A queue example

This example shows the usage of the 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.

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

/*-----------------------------------------------------------------------*/
void interrupt Timer0_ISR(void) {
    qSchedulerSysTick();
}
/*-----------------------------------------------------------------------*/
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 uint16_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) {
    uint16_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) {
    uint8_t BufferMem[16*sizeof(uint16_t)] = {0};
    HardwareSetup();  //hardware specific code
    /* next line is usted 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(uint16_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, 50, 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,
                            50, "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 *obj)
```

Resets a queue to its original empty state.

Parameters:

- `obj` : a pointer to the queue object

```
qBool_t qQueueIsEmpty(qQueue_t *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.

```
qSize_t qQueueCount(qQueue_t *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(qQueue_t *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(qQueue_t *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 *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`.

3 Modules

3.1 STimers

There are several situations were the application don'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 this purposes, STimers (Software-Timers) is the right module to use.

The STimers implementation don'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 take the value of the existing kernel clock source for reference (t_{sys}), allowing timer functionality be added to an application with minimal impact.

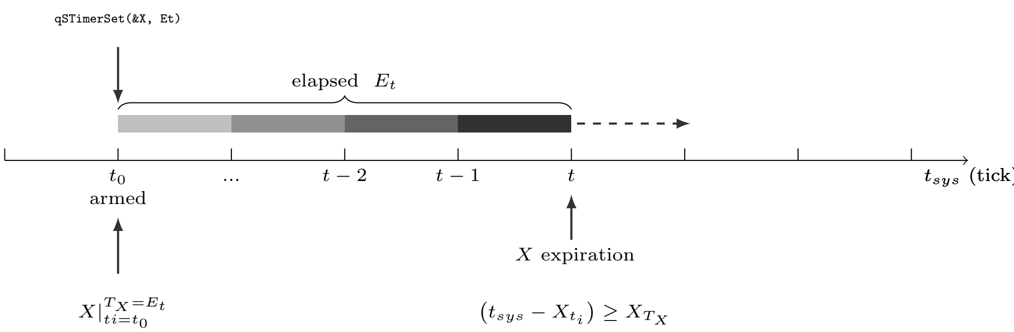

Figure 10: STimers operation

Features:

- Provides a non-blocking equivalent to delay function.
- Each STimer encapsulate 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 `qSchedulerGetTick()` API, therefore, the time resolution has the same value passed when scheduler has being initialized with `qSchedulerSetup()`.
- As illustrated in figure 10, 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 as the delays involved are shorter than 49.7 days.

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, the STimer 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, 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 obj, const qTime_t Time)
```

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

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

Parameters

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

Here, `qSTimerFreeRun()` its 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 STimer expires, otherwise, returns `qFalse`. For a disarmed STimer, also return `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 *obj)
```

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

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

```
qBool_t qSTimerStatus(const qSTimer_t *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 bellow, shows a simple usage of this object, it is noteworthy that arming is performed once using the `FirstCall` flag. This prevents 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 which 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 the finish status of the state as it's shown in the following:

```
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:

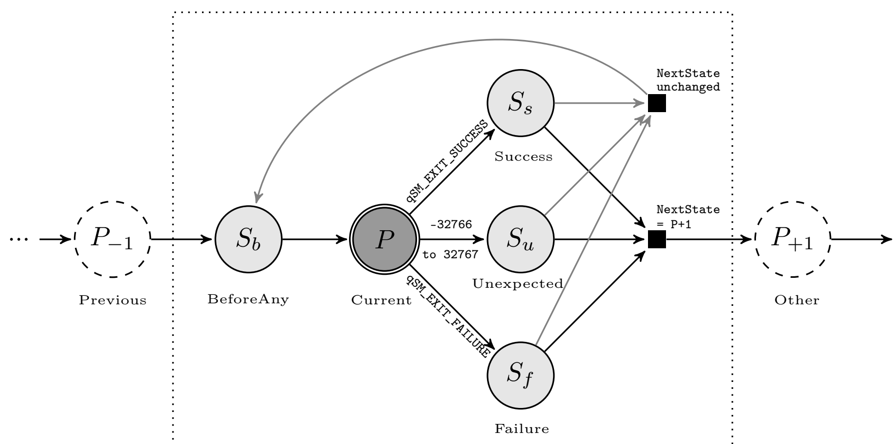

Figure 11: Sub-states evaluated after and before the *current* state

Figure 11 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 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`. 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`

As 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 *obj, qSM_State_t InitState,
                           qSM_ExState_t SuccessState,
                           qSM_ExState_t FailureState,
                           qSM_ExState_t UnexpectedState);
```

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`.

> *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 11.

```
void qStateMachine_Run(qSM_t *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 is fundamental to make the FSM move between states and additionally, get extra execution data. The fields provided 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 the 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 snipped bellow, 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){
    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*/
        return qSM_EXIT_SUCCESS;
        /*returning here make this transition to have higher priority*/
    }

    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*/
    }

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

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

The best FSM running strategy, is delegating it to a task using this API . 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 *Task, qPriority_t Priority, qTime_t Time,
                qSM_t *StateMachine, qSM_State_t InitState,
                qSM_ExState_t BeforeAnyState, qSM_ExState_t SuccessState,
                qSM_ExState_t FailureState, qSM_ExState_t UnexpectedState,
                qState_t InitialTaskState, void *arg)
```

Parameters

- `Task` : A pointer to the task node.
- `Priority` : The priority value. [0(min) - 255(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 state of the task (`qEnabled` or `qDisabled`).
- `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 system, one press of the button turns 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)

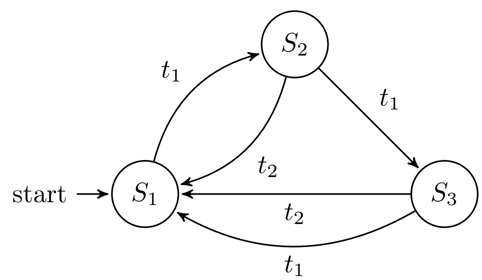

Figure 12: A simple FSM example with three states

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 12.

```
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;
}
```

Finnaly we add the task to the scheduling scheme running a dedicated state machine.

Note: Remember that you must setup 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 Changing the FSM attributes in run-time

For this, use the API that are detailed below:

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

- `obj` : A pointer to the FSM object.
- `Flag` : 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 failure sub-state.
  - `qSM_UNEXPECTED_STATE` : Set the unexpected sub-state.
  - `qSM_BEFORE_ANY_STATE` : Set the sub-state executed before any state.
- `s` : The new value for state (only apply 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 13, 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.

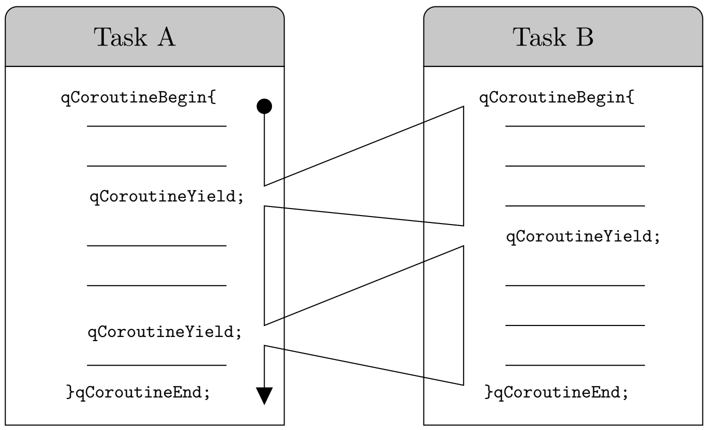

Figure 13: Coroutines in QuarkTS

The QuarkTS implementation uses the Duff's device approach, and is heavily inspired from 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

Application writer just need 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 precisely, 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 clearly 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 provide 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 signalled, 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 "SysHAL.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 be see it as a typed set of commands to produce a result, but here, the commands are typed in real-time by an 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.

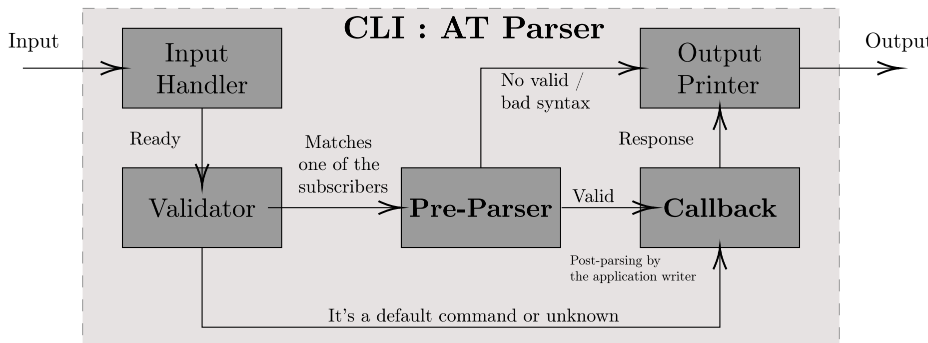

Figure 14: AT parser for a CLI implementation

As seen in figure 14, 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 a EOL(End-Of-Line) byte, 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 its valid, the respective command-callback is invoked. Here, the application writer its 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 an UART interface is choosen, the input should take the received bytes from a 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 a 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 dont take arguments of modifiers, for example,

    ```
    AT+CMD
    ```

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

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

  - **Test** (`QATCMDTYPE_TEST`) : These types of commands allow you to get the values which 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 *Parser, qPutChar_t OutputFcn,
                         char *Input,  qSize_t SizeInput, char *Output,
                         qSize_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 analyze commands that follows the simplified AT-Commands syntax already described in section 3.4.2.

```
qBool_t qATParser_CmdSubscribe(qATParser_t *Parser, qATCommand_t *Command,
                               const char *TextCommand,
                               qATCommandCallback_t Callback,
                               uint16_t CmdOpt)
```

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.

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 take two arguments and return a single value. The first argument, its just a pointer to the parser instance where the command its subscribed. From the callback context, can be used to print out extra information as 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. This 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_SET`.

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 avoid any kind of polling implementation and allow the CLI application to be designed using an event-driven pattern.

```
qBool_t qATCommandParser_ISRHandler(qATParser_t *Parser, char c)
```

```
qBool_t qATParser_ISRHandlerBlock(qATParser_t *Parser, char *data,
                                  qSize_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 this functions from the desired hardware interface, for example, from an 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 *Parser, const char *cmd)
```

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

As expected, both functions sends 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, with execption of the *Input Handler*, that should be managed by the application writer itself. However, this only take place, if the input is ready.

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

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

The simple approach for this, is 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 its leave this job be handled by task instead the application writer should code 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 will be 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 *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) - 255(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 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(qATParser_PreCmd_t *param, int8_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(qATParser_PreCmd_t *param, int8_t n)
```

To get the pointer where the desired argument starts.

```
int qATParser_GetArgInt(qATParser_PreCmd_t *param, int8_t n)
```

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

```
float qATParser_GetArgFlt(qATParser_PreCmd_t *param, int8_t n)
```

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

and

```
uint32_t qATParser_GetArgHex(qATParser_PreCmd_t *param, int8_t n)
```

To get the HEX argument parsed as `uint32_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 `uint32_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, QuarkTS provides their 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 has the same prototype as the standard C library counterparts.

3.5.1 Principle of operation

The allocation scheme works by subdividing an static array into smaller blocks. Because the array is statically declared, it will make the application appear to consume a lot of RAM, even before any memory has actually been allocated from the array.

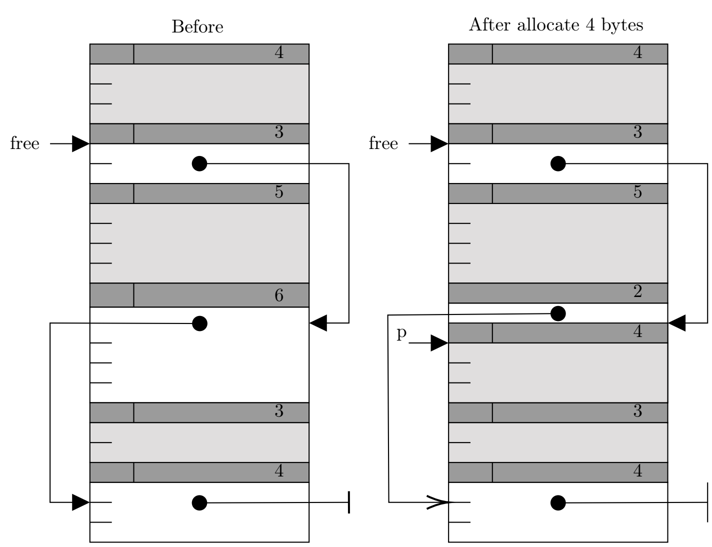

Figure 15: First-Fit allocation policy

The allocation scheme use the *First-Fit* approach (see figure 15). For better reliability, the implementation combines adjacent free blocks into a single larger block, minimizing the risk of fragmentation, and making it suitable for applications that repeatedly allocate and free different sized blocks of RAM.

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 `QuarkTS.h`.

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)
```

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 typical pools with fixed-size blocks allocation, in QuarkTS can be of any size, thereby user is responsible of selecting the appropriate memory pool to allocate data with the same size.

Besides the *default* pool, that is already defined, 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 *mPool, void* Area, size_t size)
```

Parameters

- `obj` : A pointer to the memory pool instance.
- `Area` : A pointer to a memory region (`uint8_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, beside 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 new 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 *mPool)
```

Usage example:

```
#include <stdio.h>
#include <stdlib.h>
#include "QuarkTS.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){
    qSetDebugFcn( OutPutChar );
    /*Create a memory heap*/
    qMemoryHeapCreate(mtxheap, 50, qMB_4B);
    qSchedulerSetup(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 hardware-dependant 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 an 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 its used to format the output of this 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 prototype follow the required prototype. Later, in the main thread, a call to the `qSetDebugFcn()` is used to setup 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 uint32_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 *obj, volatile uint8_t *buffer,
                    const qSize_t length)
```

Initialize the BSBuffer(Byte-sized Buffer).

Parameters:

- `obj` : A pointer to the qBSBuffer(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 *obj, uint8_t data)
```

Adds an element of data to the BSBuffer(Byte-sized Buffer).

Parameters:

- `obj` : A pointer to the qBSBuffer(Byte-sized Buffer) object
- `data` : The data to be added.

Return value:

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

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

Gets `n` data from the BSBuffer(Byte-sized Buffer) and removes them.

Parameters:

- `obj` : A pointer to the qBSBuffer(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 *obj, uint8_t *dest)
```

Gets one data-byte from the front of the BSBuffer(Byte-sized Buffer), and remove it.

Parameters:

- `obj` : A pointer to the qBSBuffer(Byte-sized Buffer) object
- `dest` : The location where the data will be written.

Return value:

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

```
uint8_t qBSBuffer_Peek(qBSBuffer_t const *obj)
```

Looks for one byte from the head of the BSBuffer(Byte-sized Buffer) without removing it.

Parameters:

- `obj` : A pointer to the qBSBuffer(Byte-sized Buffer) object

Return value:

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

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

Query the empty status of the BSBuffer(Byte-sized Buffer).

Parameters:

- `obj` : A pointer to the qBSBuffer(Byte-sized Buffer) object

Return value:

`qTrue` if the BSBuffer(Byte-sized Buffer) is empty, `qFalse` if it is not.

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

Query the full status of the BSBuffer(Byte-sized Buffer).

Parameters:

- `obj` : A pointer to the qBSBuffer(Byte-sized Buffer) object

Return value:

`qTrue` if the BSBuffer(Byte-sized Buffer) is full, `qFalse` if it is not.

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

Query the number of elements in the BSBuffer(Byte-sized Buffer).

Parameters:

- `obj` : A pointer to the qBSBuffer(Byte-sized Buffer) object

Return value:

Number of elements in the BSBuffer(Byte-sized Buffer).

4.2 Input groups for edge-checking

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

Initialize a 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 *Instance, qIONode_t *Node,
                              void *PortAddress, qBool_t PinNumber)
```

Insert a 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 *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(qIONode_t *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 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(uint32_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`.

```
uint32_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 `uint32_t`.

```
double 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(uint32_t num, char* str, uint8_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(int num, char* str, uint8_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(float f, char *str, uint8_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(float 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(float f)
```

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

Parameters:

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

Return value:

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

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