QuarkTS

Quick user manual

J. Camilo Gómez C.

January 4, 2023

Contents

1 Background

1.1 About the OS

QuarkTS is an open-source operating system that is built on top of a cooperative quasi-static scheduler. Its simplified kernel implements a specialized round-robin scheme using a linked-chain approach and an event-queue to provide true FIFO priority-scheduling.

Why cooperative?

Rather than having preemption, tasks manage their own life-cycle. This bring significant benefits, fewer re-entrance problems are encountered, because tasks cannot be interrupted arbitrarily by other tasks, but only at positions permitted by the programmer, so you mostly do not need to worry about pitfalls of the concurrent approach(resource-sharing, races, deadlocks, etc...).

What is it made for?

The design goal of QuarkTS is to achieve its stated functionality using a small, simple, and (most importantly) robust implementation to make it suitable on resource-constrained microcontrollers, where a full-preemptive RTOS is an overkill and their inclusion adds unnecessary complexity to the firmware development. In addition with a state-machines support, co-routines, time control and the inter-task communication primitives, QuarkTS provides a modern environment to build stable and predictable event-driven multitasking embedded software. Their modularity and reliability make this OS a great choice to develop efficiently a wide range of applications in low-cost devices, including automotive controls, monitoring and Internet of Things.

1.1.1 License

QuarkTS is licensed under the MIT License. You may copy, distribute and modify the software without any restriction, including without limitation the rights to to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the OS, and to permit persons to whom the OS is furnished to do so. This OS is provided as is in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE . For more details, see the MIT License in [1].

1.1.2 Hardware compatibility

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

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

1.1.3 Development process and Coding standard

QuarkTS is developed using a formal and rigorous process framed in compliance of the MISRA C 2012 and CERT coding standard guidelines and complemented with multiple static-analysis checks targered to safe critical applications.

Simply using QuarkTS in an application, does not mean developers can make a claim related to the development process and compliance of the OS to any requirements or product specification, without first, following a recognized system wide conformance verification process. Conformance evidence must then be presented, audited and accepted by a recognized and relevant independent assessment organization. Without undergoing this process of due diligence, no claim can be made as to the suitability of QuarkTS to be used in any safety or otherwise commercially critical application.

Besides the previous information, the OS sets the following clarifications regarding coding policies and naming convention:

- All the QuarkTS implementation follows the ANSI C99 standard strictly.
- Dynamic memory allocation is banned to conform the industry standards for safety-critical software: MISRA-C, DO178B, IEC 61508, ISO 26262 and so on.
- Because errors in string manipulation have long been recognized as a leading source of buffer overflows in C, a number of mitigation strategies have been devised. These include mitigation strategies designed to prevent buffer overflows from occurring and strategies designed to detect buffer overflows and securely recover without allowing the failure to be exploited.
- 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.
- In line with MISRA guides, unqualified standard **`char`** and **`char`** `*` types are only permitted to hold ASCII characters and strings respectively.
- The `_t` suffix its used to denote a type definition (i.e `qBool_t, qTask_t, size_t, ...`).
- Functions, macros, **`enum`** values and data-types are prefixed `q`. (i.e. `qFunction`, `qEnumValue`, `QCONSTANT`, `qType_t`, ...)
- Other than the pre-fix, most macros used for constants are written in all upper case.
- Almost all functions returns a boolean value of type `qBool_t`, where a `qTrue - 1u` value indicates a successful procedure and `qFalse - 0u`, the failure of the procedure.

1.1.4 Memory usage

As a quasi-static scheduler is implemented here, dynamic scheduling 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, 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 3.7 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`)	68
Finite State-Machines(FSM) handling and related APIs	314
A FSM object (`qSM_t`)	100
A state object (`qSM_State_t`)	36
STimers handling and related APIs	258
A STimer object (`qSTimer_t`)	8
Queues handling and related APIS	544
A queue object (`qQueue_t`)	28
Memory management	407
A memory pool (`qMemMang_Pool_t`)	28
The AT Command Line Interface	1724
An AT-CLI instance (`qATCLI_t`)	112
An AT-CLI command object (`qATCLI_Command_t`)	24
Remaining utilities	2980

☞Although the kernel does not use dynamically-allocated resources internally, the application writer can create an object in run-time using the safe-heap implementation provided by the memory management module, later described in section 3.5. Of course, additional checks must be performed to keep the solution safe.

1.2 Timing approach

The kernel implements a Time-Triggered Architecture (TTA)[2], in which the tasks are triggered by comparing the corresponding task-time with a reference clock. The reference clock must be real-time and follow a monotonic behavior. Usually, all embedded systems can provide this kind of reference with a constant tick generated by a periodic background hardware-timer, typically, at 1Khz (1mS tick).

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

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

API must be called inside. It is recommended that the reserved ISR should only be used by QuarkTS.

1.3 Setting up the OS kernel : `qOS_Setup`

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

```
qBool_t qOS_Setup( const qGetTickFcn_t TickProvider,
                   const qTimingBase_t BaseTimming,
                   qTaskFcn_t IdleCallback )
```

Parameters

- `TickProvider` : The function that provides the tick value. If the user application uses the `qClock_SysTick()` from the ISR, this parameter can be `NULL`.

  Note: Function should take **`void`** and return a 32bit unsigned integer.

  This argument must have this prototype : `qUINT32_t TickProviderFcn(void)`
- `BaseTimming` :This parameter specifies the ISR background timer period in seconds(floating-point format).

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

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

Usage example:

Scenario 1: When tick is already provided

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

#define TIMER_TICK    ( 0.001f )    /* 1ms */

void main( void ) {
    HAL_Init();
    qOS_Setup( HAL_GetTick, TIMER_TICK, IdleTask_Callback );
    /*
    TODO: add Tasks to the scheduler scheme and run the OS
    */
}
```

Scenario 2: When the tick is not provided

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

#define TIMER_TICK    ( 0.001f )    /* 1ms */

void Interrupt_Timer0( void ) {
    qClock_SysTick();
}

void main( void ) {
    MCU_Init();
    BSP_Setup_Timer0();
    qOS_Setup( NULL, TIMER_TICK, IdleTask_Callback );
    /* TODO: add Tasks to the scheduler scheme and run the OS */
}
```

1.4 Tasks

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

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

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

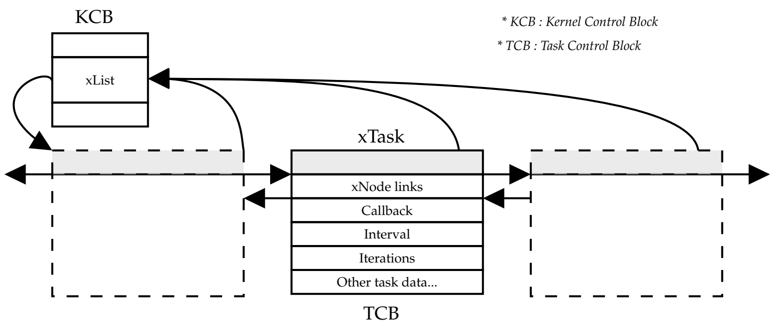

Figure 1: Task node illustration

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

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

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

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

1.4.1 The idle task

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

```
void qOS_Set_IdleTask(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 : `qOS_Add_Task`

After setting up the kernel with `qOS_Setup()`, 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 `qOS_Add_Task()` . This API can schedule a task to run every `Time` seconds, `nExecutions` times and executing `CallbackFcn` method on every pass.

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

Parameters

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

Return value

`qTrue` if the task was successfully added the scheme. Otherwise returns `qFalse`.

Caveats

1. A task with `Time` argument defined in `qTimeImmediate`, will always get the *qReady* state in every scheduling cycle, as consequence, the idle task will never gets dispatched.
2. Tasks do not remember the number of iteration set initially by the `nExecutions` argument. After the iterations are done, the internal iteration counter decreases until reach the zero. If another set of iterations is needed, the user should set the number of iterations again and resume the task explicitly.
3. Tasks that performed all their iterations, put their own state to `qDisabled`. Asynchronous triggers do not affect the iteration counter.
4. The `arg` parameter can be used as storage pointer, so, for multiple data, create a structure with the required members and pass a pointer to that structure.

Invoking `qOS_Add_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 → `qOS_Add_EventTask()`

- State-machine tasks → `qOS_Add_StateMachineTask()`. See section 3.2.7.
- AT Command Line Interface tasks → `qOS_Add_ATCLITask()`. 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 `qOS_Add_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 qOS_Add_EventTask( qTask_t * const Task, qTaskFcn_t CallbackFcn,
                           qPriority_t Priority, void* arg)
```

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

1.4.4 Removing a task : `qOS_Remove_Task`

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

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

Caveats:

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

1.5 Running the OS : `qOS_Run`

After preparing the multitasking environment for your application, a call to `qOS_Run()` is required to execute the scheduling 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 `qOS_Run()`, the OS scheduler will now be running, and the following line should never be reached, however, the user can optionally release it explicitly with `qOS_Scheduler_Release()` API function.

1.5.1 Releasing the scheduler: `qOS_Scheduler_Release`

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

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

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

```
qBool_t qOS_Set_SchedulerReleaseCallback( qTaskFcn_t Callback )
```

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

1.6 Global states and scheduling rules

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

These global states are described below:

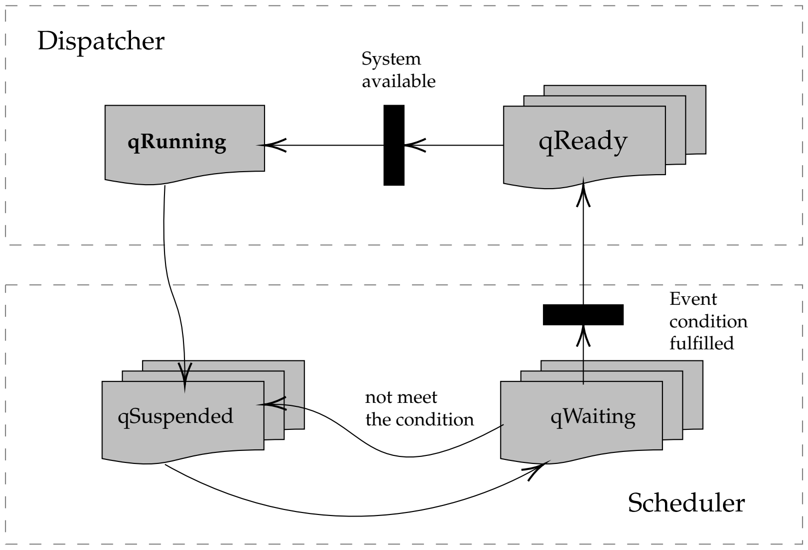

Figure 2: Task global states

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

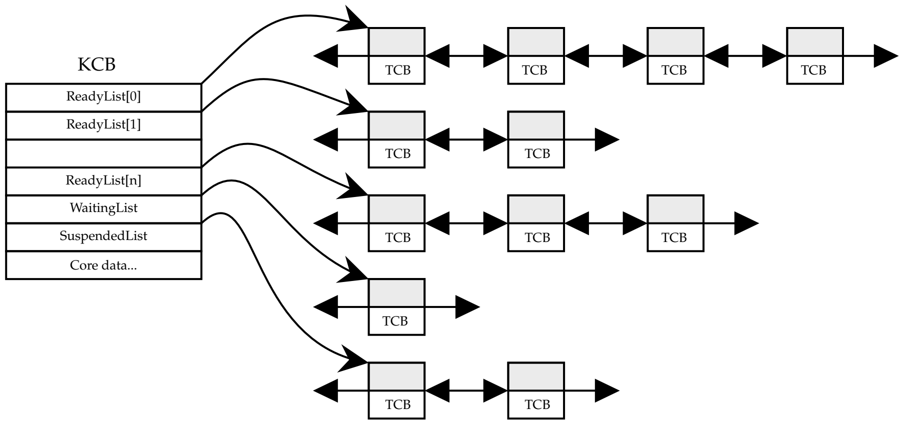

Figure 3: OS lists

The presence of a task in a particular list indicates the task's state. There are many ready-lists as defined in the `Q_PRIORITY_LEVELS` macro. To select the target ready list, the

OS use the user-assigned priority between 0 (the lowest priority) and `Q_PRIORITY_LEVELS-1` (the highest priority). For instance, if `Q_PRIORITY_LEVELS` is set to 5, then QuarkTS will use 5 priority levels or ready lists: 0 (lowest priority), 1, 2, 3, and 4 (highest priority).

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

The OS assumes that none of the tasks does a block anywhere during the *qRunning* state. Based on the *round-robin* fashion, each ready task runs in turn from every ready lists. The developer should take care to monitor their system execution times to make sure during the worst case, when all tasks have to execute, all of the deadlines are still met.

1.6.1 Rules

Task precedence is used as the task scheduling rule and precedence among tasks is determined based on the priority of each task. If there are multiple tasks able to run, the one with the highest precedence goes to *qRunning* state first.

In determining precedence among tasks, of those tasks having different priority levels, that with the highest priority has the highest precedence. Among tasks having the same priority, the one that entered the scheduling scheme first has the highest precedence if the `Q_PRESERVE_TASK_ENTRY_ORDER` configuration is enabled, otherwise the OS will reserves for himself the order according to the dynamics of the kernel lists.

1.6.1.1 Event precedence : The scheduler also has an order of precedence for incoming events (later detailed in section 2), in this way, if events of different nature converge to a single task, these will be served according to the following flowchart:

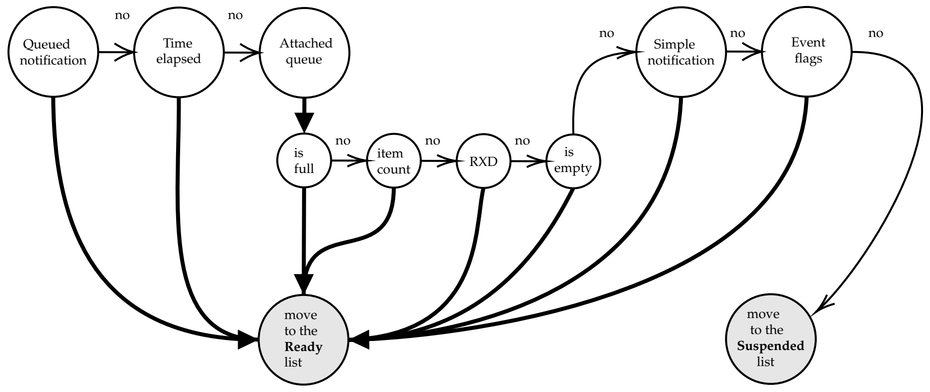

Figure 4: Event precedence

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

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

The figure 5 shows a better representation of how the event flow can be affected by this operational states.

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

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

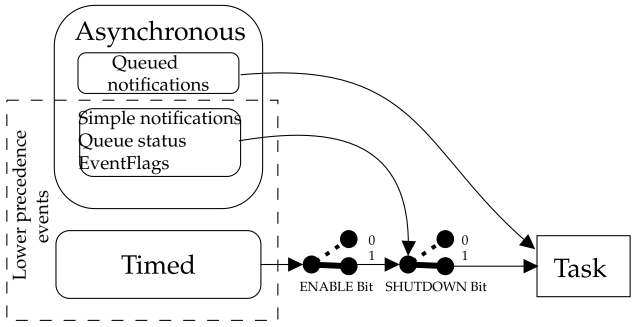

Figure 5: Event flow according operational states

1.7 Getting started

Unpack the source files and copy them into your project. Also, add a copy of the file `qconfig.h` and modify it according to your needs. Setup your compiler include path with the corresponding OS directory. Include the header file `QuarkTS.h` and setup the instance of the kernel using the `qOS_Setup()` inside the main thread. Additional configuration to the target compiler may be required to add the path to the directory of header files. The code below shows a common initialization in the main source file.

File: `main.c`

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

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

In the above code, the following considerations should be taken

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

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

1.8 Recommended programming pattern

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

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

File: `MyAppTasks.h`

```
#ifndef MYAPPTASK_H
    #define MYAPPTASK_H
    #include "Globals.h"
    #include "QuarkTS.h"

    extern qTask_t CommunicationTask, HardwareCheckTask,
                  CheckUserEventsTask, SignalAnalisysTask;

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

File: `MyAppTasks.c`

```
#include "MyAppTasks.h"

qTask_t CommunicationTask, HardwareCheckTask,
        CheckUserEventsTask, SignalAnalisysTask;

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

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

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

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

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

File: `main.c`

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

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

void main( void ) {
    /*start of hardware specific code*/
    HardwareSetup();
    Configure_Periodic_Timer_Interrupt_10ms();
    /*end of hardware specific code*/
    qOS_Setup( NULL, 0.01f, IdleTask_Callback );
    qOS_Add_Task( HardwareCheckTask, HardwareCheckTask_Callback,
                  qLowest_Priority, 0.25, qPeriodic, qEnabled, NULL );
    qOS_Add_Task( SignalAnalisysTask, SignalAnalisysTask_Callback,
                  qHigh_Priority, 0.1, 200, qEnabled, NULL );
    qOS_Add_EventTask( CheckEventsTask, CheckEventsTask_Callback,
                       qMedium_Priority, NULL );
    qOS_Add_Task( CommunicationTask, CommunicationTask_Callback,
                  qHigh_Priority, qTimeImmediate, qPeriodic,
                  qEnabled, NULL );
    qOS_Run();
    for(;;){}
}
```

1.9 Critical sections

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

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

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

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

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

Parameters:

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

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

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

```
qBool_t qTask_Set_Time( qTask_t * const Task, const qTime_t Value )
```

Set/Change the task execution interval.

Parameters:

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

```
qBool_t qTask_Set_Iterations( qTask_t * const Task, qIteration_t Value )
```

Set/Change the number of task iterations.

Parameters:

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

```
qBool_t qTask_Set_Priority( qTask_t * const Task, qPriority_t Value )
```

Set/Change the task priority value.

Parameters:

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

```
qBool_t qTask_Set_Callback( qTask_t * const Task, qTaskFcn_t CallbackFcn )
```

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

Parameters:

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

```
qBool_t qTask_Set_State( qTask_t * const Task, const qState_t State )
qBool_t qTask_Resume( qTask_t * const Task )
qBool_t qTask_Suspend( qTask_t * const Task )
qBool_t qTask_ASleep( qTask_t * const Task )
qBool_t qTask_Awake( qTask_t * const Task )
```

Set the task operability state.

Parameters:

- `Task` : A pointer to the task node.
- `State` : Use one of the following values:
  - `qEnabled` : Task will be able to catch all the events.
  - `qDisabled` : Time events will be ignored. Only asynchronous events are allowed.
  - `qAsleep` : The task can't be triggered by lower precedence events. Only the queued notifications events can be caught.
  - `qAwake` : Task will be able to catch all the events.

```
qBool_t qTask_Set_Data( qTask_t * const Task, void* UserData )
```

Set the task data.

Parameters:

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

```
qBool_t qTask_Clear( qTask_t * const Task, const qTask_ClrParam_t param )
```

Clear the specified parameter for the task.

Parameters:

- `Task` : A pointer to the task node.
- `param` : Use one of the following values:
  - `qTask_ClearIterations` : Clear the number of iterations.
  - `qTask_ClearTimeElapsed` : Clear the time elapsed.
  - `qTask_ClearCycles` : Clear the number of task activation's.
  - `qTask_ClearSimpleNotifications` : Clear the notification value.
  - `qTask_ClearQueuedNotifications` : Clear all the queued notifications.
  - `qTask_ClearNotifications` : Clear all notifications (simple and queued).

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

Retrieve the number of task activation's.

Parameters:

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

Return value:

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

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

Retrieve the task operational state.

Parameters:

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

Return value:

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

```
qTask_t* qTask_Self( 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.

```
qTask_GlobalState_t qTask_Get_GlobalState( const qTask_t * const Task )
```

Retrieve the task global-state.

Parameters:

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

Return value:

One of the available global states : `qWaiting`, `qSuspended`, `qRunning` or `qReady`. Return `qUndefinedGlobalState` if the current task its passing through a current kernel transaction

---

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

Check if the task has pending notifications. Please read section 2.2.1.

Parameters:

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

Return value:

`qTrue` if the function asserts, otherwise returns `qFalse`.

1.11 Demonstrative examples

1.11.1 A simple scheduling

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

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

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

```c
qTask_t task1, task2, task3; /*task nodes*/
/*==================================================================*/
void interrupt Timer0_ISR( void ) {
    qClock_SysTick();
}
/*==================================================================*/
void Task1_Callback( qEvent_t e ) {
    BSP_UART1_WriteString( "Task1" );

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

    if ( e->LastIteration ) {
        qTask_Suspend( &Task3 );
        qTask_Set_Time( &Task2, 0.5f );
    }
}
/*==================================================================*/
void Task2_Callback( qEvent_t e ) {
    BSP_UART1_WriteString( "Task2" );
}
/*==================================================================*/
void Task3_Callback( qEvent_t e ) {
    BSP_UART1_WriteString( "Task3" );
}
/*==================================================================*/
int main( void ) {
    HardwareSetup();  /*hardware initialization function*/
    /*function to fire an interrupt at 1ms - timer tick*/
    Configure_Periodic_Timer0_Interrupt_1ms();

    qOS_Setup( NULL, TIMER_TICK, NULL );
    qOS_Add_Task( &Task1, Task1_Callback, 50, 2.0f, 10, qEnabled, NULL );
    qOS_Add_Task( &Task2, Task2_Callback, 50, 3.0f, qPeriodic, qEnabled, NULL )
    ;
    qOS_Add_Task( &Task2, Task3_Callback, 50, 5.0f, qPeriodic, qDisabled, NULL
    );
    qOS_Run();

    return 0;
}
```

1.11.2 Using the task argument

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

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

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

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

    PID_Params_t TemperatureControl = {
       0.0f, 0.0f, 0.0f, 0.0f, /*Initial IO state of yt and ut*/
       1.5f, /*time step*/
       28.5f, /*Set-Point*/
       0.89f, 0.122f, 0.001f /*Kc, Ki, Kd*/
    };
    PID_Params_t HumidityControl= {
       0.0f, 0.0f, 0.0f, 0.0f, /*Initial IO state of yt and ut*/
       1.0f, /*time step*/
       60.0f, /*Set-Point*/
       2.5f, 0.2354f, 0.0015f /*Kc, Ki, Kd*/
    };
    PID_Params_t LightControl= {
       0.0f, 0.0f, 0.0f, 0.0f, /*Initial IO state of yt and ut*/
       0.5f, /*time step*/
       45.0f, /*Set-Point*/
       5.36f, 0.0891f, 0.0f /*Kc, Ki, Kd*/
    };
```

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

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

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

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

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

```
void PIDControl_Callback( qEvent_t e ) {
    float Error, derivative;
    /*
    Obtain the reference to the specific PID controller
    using the TaskData field from the qEvent structure
    */
    PID_Params_t *Controller = (PID_Params_t *)e->TaskData;
    /*Compute the error*/
    Error = Controller->sp - Controller->yt;
    /*Compute the accumulated error using backward integral approximation*/
    Controller->ie += Error*Controller->dt;
    /*update and compute the derivative term*/
    derivative = (Error - Controller->pe)/Controller->dt;
    /*update the previous error*/
    Controller->pe = Error;
    /*compute the pid control law*/
    Controller->ut = Controller->Kc*Error +
                     Controller->Ki*Controller->ie +
                     Controller->Kd*derivative;
}
```

1.12 Configuration macros

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

`Q_PRIORITY_LEVELS`	*Default:* `3`. The number of priorities available for application tasks.
`Q_SETUP_TIME_CANONICAL`	*Default:* `0(disabled)`. If enabled, the kernel assumes the timing base to 1mS(1KHz). So all time specifications for tasks and STimers must be set in milliseconds(mS). Also can be used to remove the floating-point operations when dealing with time. In some systems, can reduce the memory usage.
`Q_SETUP_TICK_IN_HERTZ`	*Default:* `0(disabled)`. If enabled, the timing base will be taken as frequency(Hz) instead of period(S) by `qOS_Setup()` (In some systems, can reduce the memory usage).
`Q_PRIO_QUEUE_SIZE`	*Default:* `10`. Size of the priority queue for notifications. This argument should be an integer number greater than zero. A zero value can be used to disable this functionality.
`Q_PRESERVE_TASK_ENTRY_ORDER`	*Default:* `0(disabled)`. If enabled, kernel will preserve the tasks entry order every OS scheduling cycle.
`Q_MEMORY_MANAGER`	*Default:* `1(enabled)`. Used to enable or disable the memory management module.
`Q_BYTE_ALIGNMENT`	*Default:* `8`. Used by the memory management module to perform the byte-alignment.
`Q_DEFAULT_HEAP_SIZE`	*Default:* `2048`. The total amount of heap size for the default memory pool.
`Q_NOTIFICATION_SPREADER`	*Default:* `0(disabled)`. Used to enable or disable the spread notification functionality.
`Q_FSM`	*Default:* `1(enabled)`. Used to enable or disable the Finite State Machine (FSM) extension.
`Q_FSM_MAX_NEST_DEPTH`	*Default:* `5`. The max depth of nesting in Finite State Machines (FSM).
`Q_FSM_MAX_TIMEOUTS`	*Default:* `3`. Max number of timeouts inside a timeout specification for the Finite State machine (FSM) module.
`Q_QUEUES`	*Default:* `1(enabled)`. Used to enable or disable the queues APIs for communication to tasks.
`Q_TRACE_VARIABLES`	*Default:* `1(enabled)`. Used to enable or disable debug and trace macros.
`Q_DEBUGTRACE_BUFSIZE`	*Default:* `36`. The buffer size for debug and trace macros.
`Q_DEBUGTRACE_FULL`	*Default:* `1(enabled)`. Used to enable of disable the extended output of trace macros.
`Q_ATCLI`	*Default:* `1(enabled)`. Used to enable or disable the AT Command Line Interface (CLI) module.
`Q_TASK_COUNT_CYCLES`	*Default:* `0(disabled)`. Used to enable or disable the cycle count in tasks.
`Q_TASK_EVENT_FLAGS`	*Default:* `1(enabled)`. Used to enable or disable the task event flags.
`Q_MAX_FTOA_PRECISION`	*Default:* `10`. The default precision used to perform float to ASCII conversions.
`Q_ATOF_FULL`	*Default:* `0(disabled)`. Used to enable or disable the scientific notation in ASCII to float conversions.
`Q_ALLOW_SCHEDULER_RELEASE`	*Default:* `0(disabled)`. Used to enable or disable the scheduler release functionality.
`Q_RESPONSE_HANDLER`	*Default:* `1(enabled)`. Used to enable or disable the response handler functionality.
`Q_EDGE_CHECK_IOGROUPS`	*Default:* `1(enabled)`. Used to enable or disable the edge check functionality for I/O groups .
`Q_BYTE_SIZED_BUFFERS`	*Default:* `1(enabled)`. Used to enable or disable the usage of Byte-sized buffers.
`Q_USE_STDINT_H`	*Default:* `1(enabled)`. Use the `stdint.h` header to define kernel data-types.

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 6).
- 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 6). 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.

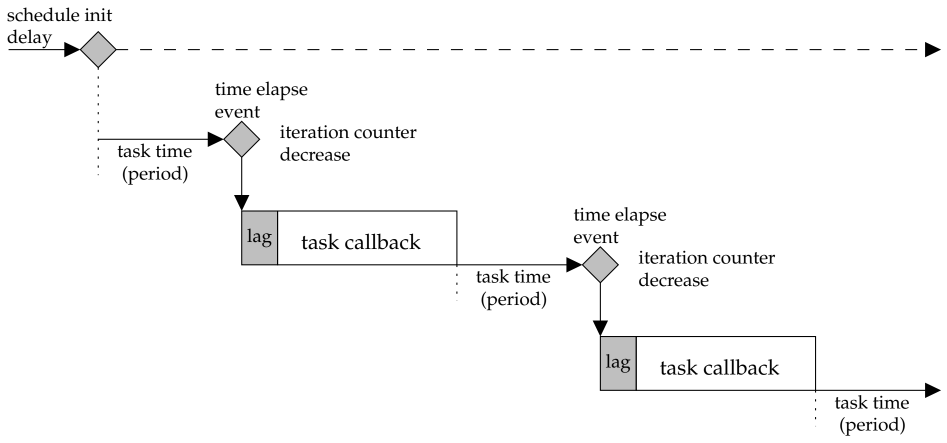

Figure 6: Inherit time lag

- 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 7).

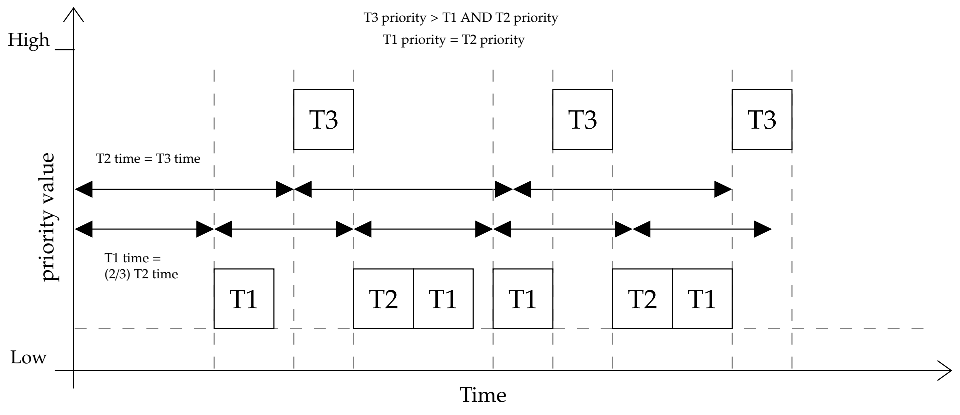

Figure 7: Priority scheduling example with three (3) tasks attached triggered by time-elapsed events

2.2 Asynchronous events and inter-task communication

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

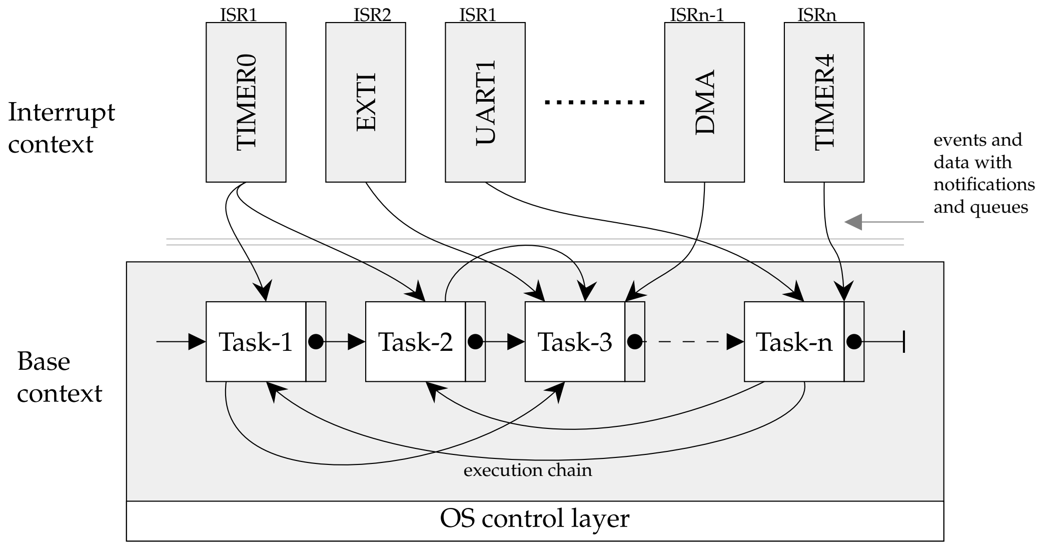

Figure 8: Heavy cooperative environment

As shown in figure 5, two main scenarios are presented, *ISR-to-task* and *task-to-task* interaction.

When using interrupts to catch external events, it is expected to be handled with fast and lightweight code to reduce the variable ISR overhead introduced by the code itself. If too much overhead is used inside an ISR, the system will tend to lose future events. In some specific situations, in the interest of stack usage predictability and to facilitate system behavioral analysis, the best approach is to synchronize the ISR with a task to leave the heavy job in the base-level instead of the interrupt-level, so interrupt handlers only collect event data and clear the interrupt source and therefore exit promptly by deferring the processing of the event data to a task, this is also called *Deferred Interrupt Handling*.

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

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

The OS provides the following features for task communication:

2.2.1 Notifications

The notifications allow tasks to interact with other tasks and to synchronize with ISRs without the need of intermediate variables or separate communication objects. By using notifications, a task or ISR can launch another task sending an event and related data to the receiving task. This is depicted in figure 9.

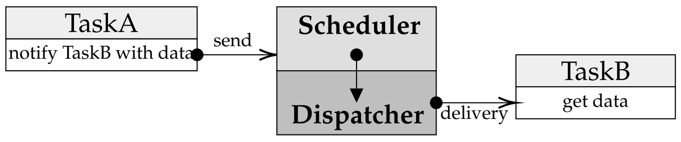

Figure 9: A notification used to send an event directly from one task to another

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

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

Parameters

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

Return value:

`qTrue` if the notification has been sent, or `qFalse` if the notification value reaches its max value.

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

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

Here, the `qTask_Notification_Queue()` 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 [3]. That is, the tasks are (conceptually) stored in the queue in priority order instead of the insertion order. If two tasks with the same priority are notified, they are served in the FIFO form according to their order inside the queue. Figure 10 illustrates this behavior.

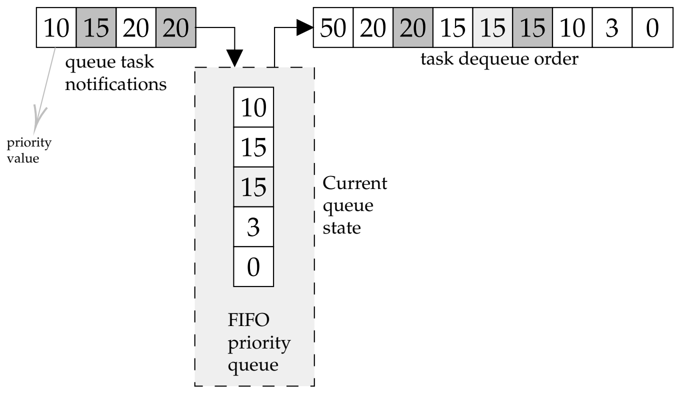

Figure 10: Priority-queue behavior

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

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

Parameters

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

Return value:

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

Figure 11, 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 `qTask_Notification_Queue()` 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`. Following the priority-queue behavior, the scheduler makes a new reordering, so the next queue extraction will be for `Task-D`, `Task-C`, and `Task-B` sequentially.

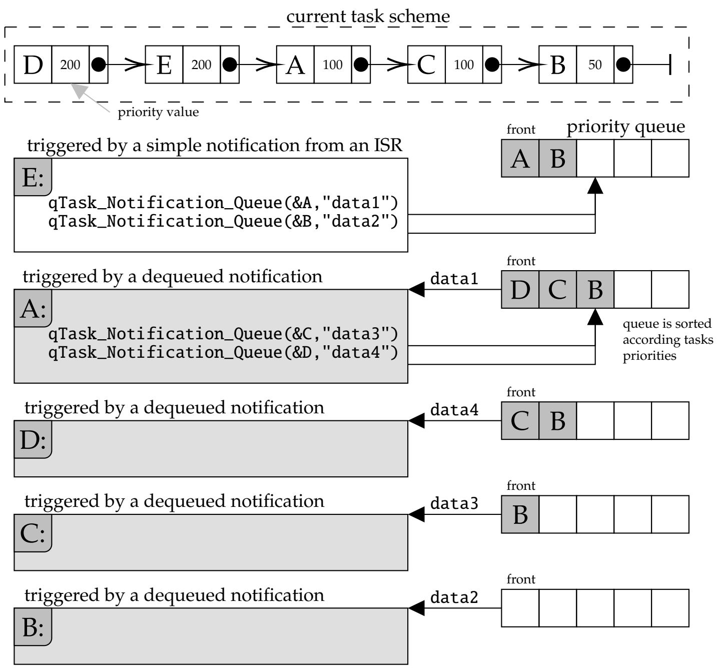
current task scheme
D 200
E 200
A 100
C 100
B 50
priority value
triggered by a simple notification from an ISR
E:
qTask_Notification_Queue(&A,"data1")
qTask_Notification_Queue(&B,"data2")
front
priority queue
A B
triggered by a dequeued notification
A:
qTask_Notification_Queue(&C,"data3")
qTask_Notification_Queue(&D,"data4")
data1
D C B
queue is sorted according tasks priorities
triggered by a dequeued notification
D:
data4
C B
triggered by a dequeued notification
C:
data3
B
triggered by a dequeued notification
B:
data2

Figure 11: Priority-queue example

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

☞Among all the provided events, queued notifications have the highest precedence.

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 `qOS_Notification_Spread()` can be used.

```
qBool_t qOS_Notification_Spread( const void *eventdata,
                                 const qTask_NotifyMode_t mode )
```

Parameters

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

Return value:

`qTrue` on success. Otherwise `qFalse`.

☞This function spreads a notification event among all the tasks in the scheduling scheme, so, for tasks that are not part of the barrier, just discard the notification. This operation will be performed in the next scheduling cycle.

2.2.3 Queues

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

As showed in figure 12, the last position is connected back to the first position to make a circle. It is also called *ring-buffer* or *circular-queue*.

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

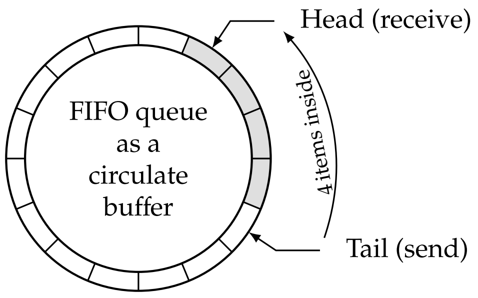

Figure 12: qQueues conceptual representation

The usage of this data structure is detailed in section 2.4.2.

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

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

2.2.4 Event Flags

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

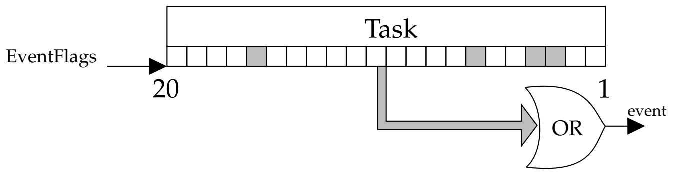

Figure 13: Task event flags

Up to twenty(20) bit-flags are available per task and whenever the scheduler sees that one event-flag is set, the kernel will trigger the task execution.

The function `qTask_EventFlags_Modify` is intended to modify the task event-flags:

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

Parameters

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

  `QEVENTFLAG_01 | QEVENTFLAG_02 | QEVENTFLAG_03 | ... | QEVENTFLAG_20`
- `action` : `QEVENTFLAG_SET` or `QEVENTFLAG_CLEAR`.

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

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

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

Parameters

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

Return value:

The EventFlag value of the task.

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

Parameters

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

- `CheckForAll` : Used to create either a logical AND test (where all flags must be set)or a logical OR test (where one or more flags must be set) as follows:

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

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

Return value:

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

2.3 Retrieving the event data

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

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

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

This data structure is defined as:

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

Each field of the structure is described as follows

- `Trigger` : The flag that indicates the event source that triggers the task execution. This flag can only have nine(9) possible values:
  - `byTimeElapsed` : When the time specified for the task elapsed.
  - `byNotificationQueued` : When there is a queued notification in the FIFO priority queue. For this trigger, the dispatcher performs a dequeue operation automatically. A pointer to the dequeued event-data will be available in the `EventData` field.

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

- `TaskData` : The storage pointer. Tasks can store a pointer to specific variable, structure or array, which represents specific user data for a particular task. This may be needed if you plan to use the same callback method for multiple tasks.
- `EventData` : Associated data of the event. Specific data will reside here according to the event source. This field will have a `NULL` value when the trigger gets one of this values: *byTimeElapsed*, *byEventFlags* and *byNoReadyTasks*.
- `FirstCall` : This flag indicates that a task is running for the first time. Can be used for data initialization purposes.
- `FirstIteration` : Indicates whether current pass is the first iteration of the task. This flag will be only set when time-elapsed events occurs and the iteration counter has been parameterized.
- `LastIteration` : Indicates whether current pass is the last iteration of the task. This flag will be only set when time-elapsed events occurs and the iteration counter has been parameterized.
- `StartDelay` : The number of epochs between current system time and point in time when the task was marked as Ready. Can be used to keep track when current task's execution took place relative to when it was scheduled. A value of 0 (zero) indicates that task started right on time per schedule. This parameter will be only available on timed tasks, when `Trigger == ` *byTimeElapsed*.

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

2.4 Implementation guidelines

2.4.1 Sending notifications

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

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

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

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

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

/*==================================================================*/
void interrupt Timer1Second_ISR( void ) {
    qTask_Notification_Send( &taskA, NULL );
    HAL_ClearInterruptFlags( HAL_TMR_ISR ); /*hardware dependent code*/
}
/*==================================================================*/
void interrupt ExternalInput_ISR( void ) {
    if ( RISING_EDGE == HAL_GetInputEdge() ) { /*hardware dependent code*/
        qTask_Notification_Queue( &taskA, app_events[ 1 ] );
    }
    else {
        qTask_Notification_Queue( &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( 0 == strcmp( e->EventData, "ButtonFallingEdge" ) ){
                press_counter++; /*count the button press*/
                if( 3 == press_counter ){ /*after 3 presses*/
                    /*send the notification of 3 presses to taskB*/
                    qTask_Notification_Send( &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.
         */
        qDebug( e->EventData, Message );
    }
}
/*==================================================================*/
int main( void ) {
    HAL_Setup_MCU(); /*hardware dependent code*/
    qTrace_Set_OutputFcn( HAL_OutPutChar );
    /* setup the scheduler to handle up to 10 queued notifications*/
    qOS_Setup( HAL_GetTick, 0.001, NULL );
    qOS_Add_EventTask( &taskA, taskA_Callback, qLowest_Priority, NULL );
    qOS_Add_EventTask( &taskB, taskB_Callback, qLowest_Priority, NULL );
    qOS_Run();
    return 0;
}
```

2.4.2 Setting up a queue : `qQueue_Setup`

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

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

```
qBool_t qQueue_Setup( qQueue_t * const q, void* dataArea,
                      size_t itemSize, size_t itemsCount )
```

Parameters

- `q` : A pointer to the queue object

- `dataArea` : Must point to a data block or array of data that is at least large enough to hold the maximum number of items that can be in the queue at any one time
- `itemSize` : Size of one element in the data block
- `itemsCount` : The maximum number of items the queue can hold at any one time.

2.4.3 Performing queue operations

```
qBool_t qQueue_Send( qQueue_t * const q, void *itemToQueue,
                     const qQueue_InsertMode_t pos );
```

Parameters

- `q` : 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.
- `pos` : Can take the value `QUEUE_SEND_TO_BACK` to place the item at the back of the queue, or `QUEUE_SEND_TO_FRONT` to place the item at the front of the queue (for high priority messages).

Return value

`qTrue` on the item was successfully added to the queue, `qFalse` if not.

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

```
qBool_t qQueue_Receive( qQueue_t * const q, void *dst )
```

Parameters

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

Return value

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

2.4.4 Attach a queue to a task

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

```
qBool_t qTask_Attach_Queue( qTask_t * const Task, qQueue_t * const q,
                            const qQueue_LinkMode_t mode,
                            const qUINT16_t arg )
```

Parameters

- `Task` : A pointer to the task node
- `q` : 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:
  - `qQueueMode_Receiver` : The task will be triggered if there are elements in the queue.
  - `qQueueMode_Full` : The task will be triggered if the queue is full.
  - `qQueueMode_Count` : The task will be triggered if the count of elements in the queue reach the specified value.
  - `qQueueMode_Empty` : The task will be triggered if the queue is empty.
- `arg` : This argument defines if the queue will be attached (`qLink`) or detached (`qUnLink`) from the task. If the `qQueueMode_Count` mode is specified, this value will be used to check the element count of the queue. A zero value will act as `qUnLink` action.

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

2.4.5 A queue example

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

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

/*----------------------------------------------------------------------*/
```

```
void interrupt Timer0_ISR( void ) {
    qClock_SysTick();
}
/*-----------------------------------------------------------------------*/
qTask_t TSK_PRODUCER, TSK_CONSUMER; /*task nodes*/
qQueue_t UserQueue; /*Queue Handle*/
/*-----------------------------------------------------------------------*/

/* The producer task puts data into the buffer if there is enough free
 * space in it, otherwise the task block itself and wait until the queue
 * is empty to resume. */
void TSK_Producer_Callback( qEvent_t e ) {
    static qUINT16_t unData = 0;
    unData++;
    /*Queue is empty, enable the producer if it was disabled*/
    if ( byQueueEmpty == e->Trigger ) {
        qTask_Resume( qTask_Self() );
    }

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

    qOS_Setup( NULL, TIMER_TICK, IdleTask_Callback );
    /*Setup the queue*/
    qQueue_Setup( &UserQueue, BufferMem /*Memory block used*/,
                  sizeof(qUINT16_t) /*element size*/,
                  16 /* element count*/ );

    /*  Append the producer task with 100mS rate. */
    qOS_Add_Task( &TSK_PRODUCER, TSK_Producer_Callback,
                  qMedium_Priority, 0.1f, qPeriodic, qEnabled,
                  "producer" );
    /* Append the consumer as an event task. The consumer will
     * wait until an event trigger their execution
     */
    qOS_Add_EventTask( &TSK_CONSUMER, TSK_Consumer_Callback,
                       qMedium_Priority, "consumer" );
    /* the queue will be attached to the consumer task
     * in qQueueMode_Count mode. This mode sends an event to the consumer
     * task when the queue fills to a level of 4 elements
     */
    qTask_Attach_Queue( &TSK_CONSUMER, &UserQueue, qQueueMode_Count, 4 );
    /* the queue will be attached to the producer task in
     * qQueueMode_Empty mode. This mode sends an event to the producer
     * task when the queue is empty
     */

    qTask_Attach_Queue( &TSK_PRODUCER, &UserQueue, qQueueMode_Empty, qATTACH );
    qOS_Run();
    return 0;
}
```

2.4.6 Other queue APIs

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

Resets a queue to its original empty state.

Parameters:

- `obj` : A pointer to the queue object

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

Returns the empty status of the queue.

Parameters:

- `obj` : A pointer to the queue object

Return value:

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

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

Returns the number of items in the queue.

Parameters:

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

Return value:

The number of elements in the queue.

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

Returns the full status of the queue.

Parameters:

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

Return value:

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

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

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

Parameters:

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

Return value:

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

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

Remove the data located at the front of the queue.

Parameters:

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

Return value:

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

2.4.7 Using the task Event-flags

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

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

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

qTask_t TaskDataProducer;
qUINT8_t dataToTransmit[ 10 ] = { 0u };

/*-----------------------------------------------------------------------*/
void interrupt Timer0_ISR( void ){
    qClock_SysTick();
}
/*-----------------------------------------------------------------------*/
void interrupt Timer1_ISR( void ) {
    qTask_EventFlags_Modify( &TaskDataProducer, TIMER_EXPIRED,
                             QEVENTFLAG_SET );
}
/*-----------------------------------------------------------------------*/
void interrupt EXTI_ISR( void ) {
    if ( EXTI_IsRisingEdge() ) {
        qTask_EventFlags_Modify( &TaskDataProducer, SWITCH_CHANGED,
                                 QEVENTFLAG_SET );
    }
}
/*-----------------------------------------------------------------------*/
void TaskDataProducer_Callback( qEvent_t e )
    qBool_t condition;
    condition = qTask_EventFlags_Check( &TaskDataProducer,
                                        DATA_TXMIT, qTrue, qTrue );
    if ( qTrue == condition) {
        GenerateData( dataToTransmit );
        qTask_EventFlags_Modify( &TaskDataProducer, DATA_READY,
                                 QEVENTFLAG_SET );
    }
    qTask_EventFlags_Check( &TaskDataProducer,
                            DATA_READY | SWITCH_CHANGED | TIMER_EXPIRED,
                            qTrue, qTrue );
}
/*-----------------------------------------------------------------------*/
void IdleTask_Callback( qEvent_t e ) {

    TransmitData( dataToTransmit );
    qTask_EventFlags_Modify( &TaskDataProducer,
                              DATA_TXMIT, QEVENTFLAG_SET );
}
/*-----------------------------------------------------------------------*/
int main( void ) {
    HardwareSetup();  //hardware specific code
    /* next line is used to setup hardware with specific code to fire
     * interrupts at 1ms - timer tick*/
    Configure_Periodic_Timer0_Interrupt_1ms();
    Configure_Periodic_Timer1_Interrupt_2s();
    Configure_External_Interrupt();
    /*
    Idle task will be responsible to transmit the generate the data after
    all conditions are meet
    */
    qOS_Setup( NULL, TIMER_TICK, IdleTask_Callback );
    /*
    The task will wait until data is transmitted to generate another set of
    data
    */
    qOS_Add_EventTask( &TaskDataProducer, TaskDataProducer_Callback,
                       qHigh_Priority, "DATAPRODUCER" );
    /*
    Set the flag DATA_TXMIT as initial condition to allow the data
    generation at startup
    */
    qTask_EventFlags_Modify( &TaskDataProducer, DATA_TXMIT, QEVENTFLAG_SET );
    qOS_Run();
    for(;;){}
    return 0;
}
```

3 Extensions

3.1 STimers

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

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

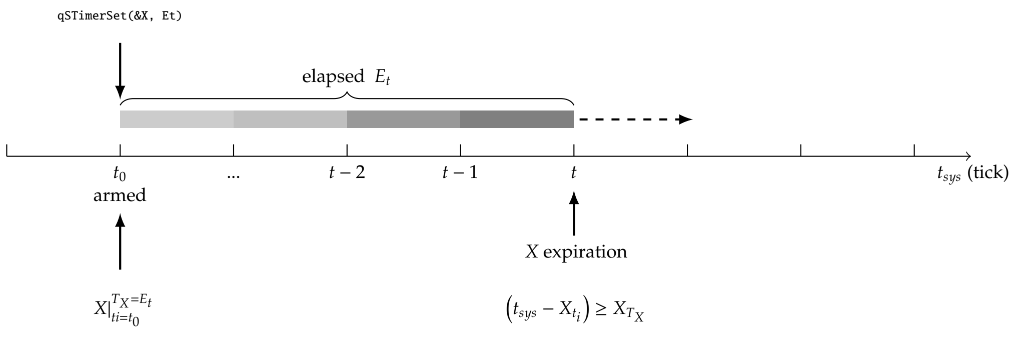

Figure 14: STimers operation

As illustrated in figure 14, 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 `(qSTimer_Set())`, (X_{t_i}). Thanks to modular arithmetic, both of these are guaranteed to work fine across the clock-source rollover(a 32bit unsigned-counter), at least, as long the delays involved are shorter than 49.7 days.

Features:

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

3.1.1 Using a STimer

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

To use them, the code should follow a specific pattern that deals with the states of this object. All related APIs are designed to be non-blocking, this means there are ideal for use in cooperative environments as the one provided by the OS itself. To minimize the implementation, this object is intentionally created to behave like a binary object, this implies that it only handles two states, *Armed* and *Disarmed*.

An *Armed* timer means that it is already running with a specified preset value and a *Disarmed* timer is the opposite, which means that it doesn't have a preset value, so consequently, it is not running at all.

The arming action can be performed with `qSTimer_Set()` or `qSTimer_FreeRun()` and disarming with `qSTimer_Disarm()`.

Detailed APIs description is presented below. (For `qSTimer_Disarm()` ignore the `Time` argument.)

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

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

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

Parameters

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

Here, `qSTimer_FreeRun()` is a more advanced API, it checks the timer and performs 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 `qSTimer_Set()` and `qSTimer_Disarm()` `qTrue` on success, otherwise, returns `qFalse`.

For `qSTimer_FreeRun()` returns `qTrue` when the STimer expires, otherwise, returns `qFalse`. For a disarmed STimer, also returns `qFalse`.

All possible checking actions are also provided for this object, including `qSTimer_Elapsed()` , `qSTimer_Remaining()` and `qSTimer_Expired()` , with the last one being the most commonly used for timing applications. Finally, to get the current status of the STimer (check if is Armed or Disarmed) the `qSTimer_Status()` API should be used.

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

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

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

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

For this APIs, their only argument, is a pointer to the STimer object.

Return Value

For `qSTimer_Elapsed()`, `qSTimer_Remaining()` returns the elapsed and remaining time specified in epochs respectively.
For `qSTimer_Expired()`, returns `qTrue` when STimer expires, otherwise, returns `qFalse`. For a disarmed STimer, also returns `qFalse`.
For `qSTimer_Status()`, returns `qTrue` when armed, and `qFalse` for disarmed.

Usage example:

The example below shows a simple usage of this object. It is noteworthy that arming is performed once using the `FirstCall` flag. This prevents the timer from being re-armed every time the task runs. 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*/
        qSTimer_Set( &timeout, 3.5f );
    }

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

3.2 Finite State Machines (SM)

The state machine is one of the fundamental programming patterns that are most commonly used. This approach breaks down the design into a series of finite steps called "states" that perform some narrowly defined actions. Every state can change to another as a consequence of incoming stimuli also called events or signals. This elemental mechanism allows designers to solve complex engineering problems in a very straightforward way. Knowing the importance of this approach in the development of embedded applications, the OS adopts this design pattern as a kernel extension.

In an effort to maximize efficiency and minimize complexity, the module implements the basic features of the Harel statecharts to represent hierarchical state machines. This features form a proper subset that approaches in a very minimalist way, some of the specifications of the UML statecharts, including:

- Nested states with proper handling of group transitions and group reactions.
- Guaranteed execution of entry/exit actions upon entering/exiting states.
- Straightforward transitions and guards.

In addition to this, the provided implementation also features a powerful coding abstraction including transition tables and timeout signals, allowing to build scalable solutions from simple flat state-machines to complex statecharts.

3.2.1 The provided approach

In QuarkTS, a state-machine must be instantiated with an object of type `qSM_t`. States are represented as instances of the `qSM_State_t` object.

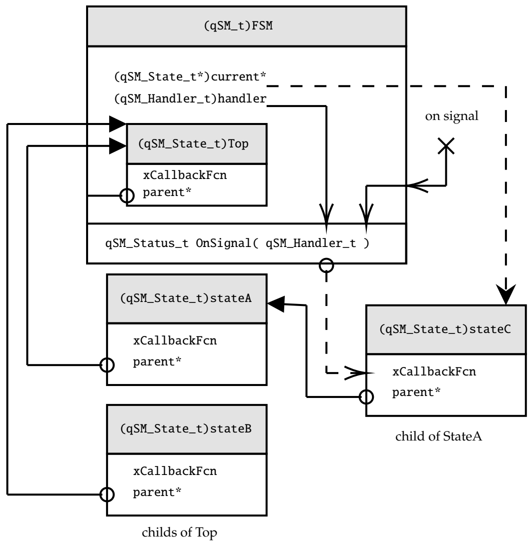

Figure 15: FSM module design

One important attribute of the `qSM_State_t` object is the callback function, which is used to describe the behavior specific to the state. Also there is a pointer to the parent state to define nesting of the state and its place in the hierarchical topology. As shown in figure 15, a state machine consist of a least one state, the "top level" state. So concrete state machine are built by adding an arbitrary number states and defining callback functions. The only purpose of the top state is to provide the root of the hierarchy, so that the highest level can return to top as their parent state.

3.2.2 Setting up a state machine : `qStateMachine_Setup`

Like any other OS object, a Finite State Machine (FSM) must be explicitly initialized before it can be used. The `qStateMachine_Setup()` API initializes the instance, sets the callback for the top state, sets the initial state and the surrounding callback function.

```
qBool_t qStateMachine_Setup(qSM_t * const m, qSM_StateCallback_t topCallback,
                            qSM_State_t * const initState,
                            qSM_SurroundingCallback_t surrounding, void *Data )
```

Parameters

- `m` : A pointer to the FSM object.
- `topCallback` : The callback for the "Top" state. This argument is a pointer to a callback function, returning `qSM_Status_t` and with a `qSM_Handler_t` variable as input argument.
- `initState` : The first state to be executed (initial-state).
- `Surrounding` : The surrounding callback function. To ignore pass `NULL`.
- `Data` : Represents the FSM arguments. To ignore pass `NULL`. All arguments must be passed by reference and cast to `(void *)`. Only one argument is allowed, so, for multiple arguments, create a structure that contains all of the arguments and pass a pointer to that structure.

➥*Note*: For the `Surrounding` argument, a `NULL` value will act as a "disable" action.

3.2.3 Subscribing states and defining callbacks:

State machines are constructed by composition, therefore, the topology of a state machine is determined upon construction. In this module implementation, there are not distinction between composite states(states containing substates) and leaf states. All states are potentially composite. The API `qStateMachine_StateSubscribe` should be used to initialize the state and define its position in the topology.

```
qBool_t qStateMachine_StateSubscribe( qSM_t * const m,
                                      qSM_State_t * const state,
                                      qSM_State_t * const parent,
                                      qSM_StateCallback_t StateFcn,
                                      qSM_State_t * const initState,
                                      void *Data )
```

Parameters

- `m` : a pointer to the FSM object.
- `state` : A pointer to the state object.
- `parent` : A pointer to the parent state. Pass `NULL` if this state its a child of the *Top* state.
- `StateFcn` : The handler function associated to the state.

  Prototype: `qSM_Status_t xCallback( qSM_Handler_t h )`
- `initState` : The first child-state to be executed if the subscribed state its a parent in an hierarchical pattern. To ignore pass `NULL` as argument.
- `Data` : State data. Storage pointer. To ignore pass `NULL` as argument.

A state callback-functions takes a `qSM_Handler_t` object as input argument and returns a `qSM_Status_t` value. An example is shown in the following code snippet:

```
qSM_Status_t ExampleState_Callback( qSM_Handler_t h ){
    /* TODO: State code */
    return qSM_STATUS_EXIT_SUCCESS;
}
```

3.2.4 The state callback/handler : Performing transitions and retrieving data :

Because callback functions are methods derived from the state-machine object, they have direct access to some attributes via the `qSM_Handler_t` argument. The usage of this object it's required to make the FSM moves between states and additionally get extra data. The provided attributes are:

- `NextState` : Desired next state. The application writer should change this field to another state to produce a state transition in the next FSM's cycle. Changing this field will only take effect when the state is executed under user custom-defined signals or in the absence of signals `QSM_SIGNAL_NONE`.
- `StartState` : Desired nested initial state (substate). The application writer should change this field to set the initial transition if the current state its a parent(or composite state). Changing this field attribute only take effect when the state is executed under the `QSM_SIGNAL_START` signal.
- `Signal` (read-only): Received signal. Can have any of the following values:
  - `QSM_SIGNAL_NONE` if no signal available.
  - `QSM_SIGNAL_ENTRY` if the current state has just entered from another state.

- `QSM_SIGNAL_START` to set nested initial transitions by using the `StartState` attribute.
- `QSM_SIGNAL_EXIT` if the current state has just exit to another state.
- Any other user-defined signal will reside here, including the `QSM_SIGNAL_TIMEOUT(#)` signals.

- `TransitionHistory` : Use this option if the transition is to a composite state. This attribute defines how the story should be handled. If this field is not established, `qSM_TRANSITION_NO_HISTORY` is assumed. The possible values for this attribute are:
  - `qSM_TRANSITION_NO_HISTORY` : History is not preserved. Composite states will start according to their default transition.
  - `qSM_TRANSITION_SHALLOW_HISTORY` : History will be kept to allow the return to only the top-most sub-state of the most recent state configuration, which is entered using the default entry rule.
  - `qSM_TRANSITION_DEEP_HISTORY` : History will be kept to allow full state configuration of the most recent visit to the containing region.
- `Status` (read-only): The exit(or return) status of the last state. Should be used in the *Surrounding* callback to perform the corresponding actions for every value. On states callback will take the value `qSM_STATUS_NULL`;
- `machine` (read-only): A generic pointer to the container state machine.
- `Data` (read-only): State-machine associated data. If the FSM is running as a task, the associated event data can be queried through this field. (here, a cast to `qEvent_t` is mandatory).
- `StateData` (read-only): State associated data. Storage-pointer.

Within the callback function of every state, only one level of dispatching (based on the signal) is necessary. Typically this is archived using a single-level switch statement. Callback functions communicate with the state machine engine through the `qSM_Handler_t` and the return value of type `qSM_Status_t`.

The semantic is simple, if a signal is processed, the callback functions returns the status value `qSM_STATUS_SIGNAL_HANDLED`. otherwise it throws the signal for further processing by higher-level states. Also, this returning mechanism can be used to handle exceptions by using the surrounding callback.

Entry/Exit actions and default transitions are also implemented inside the callback function in response of pre-defined signals. `QSM_SIGNAL_ENTRY`, `QSM_SIGNAL_EXIT` and `QM_SIGNAL_START`. The state machine generates and dispatches this signals to appropriates handlers upon state transitions.

The example below shows what a status callback should look like including the use of the handler.

```
qSM_Status_t ExampleState_Callback( qSM_Handler_t h ){
    switch( h ){
        case QSM_SIGNAL_START:
            break;
        case QSM_SIGNAL_ENTRY:
            break;
        case QSM_SIGNAL_EXIT:
            break;
        case USER_DEFINED_SIGNAL:
            h->NextState  =  &OtherState; /*transition*/
            break;
        default:
            break;
    }
    return qSM_STATUS_EXIT_SUCCESS;
}
```

As shown above, the return value represents the exit status of the state, and it can be handled with an additional *surrounding* callback function (S_u) established at the moment of the FSM setup. The values allowed to return are listed below.

- `qSM_STATUS_EXIT_SUCCESS`.
- `qSM_STATUS_EXIT_FAILURE`.
- `qSM_STATUS_SIGNAL_HANDLED`.
- Any other integer value between `-32762` and `32767`.

To code initial transitions, application writer should catch the `QSM_SIGNAL_START`, perform the required actions and then designate the target sub-state by assigning the `StartState` attribute of the `qSM_Handler_t` argument. Regular transitions are coded in a very similar way, except that here, you catch the custom-defined signal and then assign the `NextState` attribute of the `qSM_Handler_t` argument. The developer is free to write and control state transitions. Transitions are only allowed under the availability of user custom-defined signals. Regular transitions are not allowed at an entry point (`QSM_SIGNAL_ENTRY`), exit point (`QSM_SIGNAL_EXIT`), or a start point (`QSM_SIGNAL_START`).

➥*Note*: User should not target the top state in a transition and use it as transition source either. The only customizable aspect of the top state is the initial transition.

3.2.5 The surrounding callback

It is a checkpoint before and after each state executes its activities through its state-callback. The behavior of this surrounding callback must be defined by the programmer.

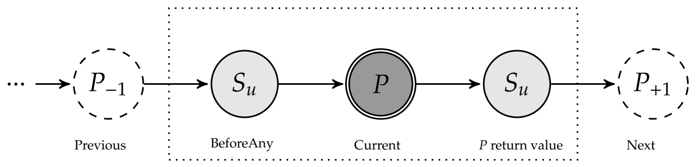

Figure 16: Surrounding callback (S_u) invocation after and before the *current* state (P)

When the surrounding callback is executed, it indicates its own checkpoint through the `Status` attribute of the `qSM_Handler_t` argument.

Unlike a state callback, the *Surrounding* callback should not return anything, thus, the callback should be written as:

```
void SurroundingCallback_Example( qSM_Handler_t m ){
    switch( h->Status ){
        case qSM_STATUS_BEFORE_ANY:
            /* TODO: before any code */
            break;
        case qSM_STATUS_EXIT_FAILURE:
            /* TODO: failure code */
            break;
        case qSM_STATUS_EXIT_SUCCESS:
            /* TODO: success code */
            break;
        case qSM_STATUS_SIGNAL_HANDLED:
            /* TODO: signal handled code */
            break;
        case 5: /*user defined return value*/
            /* TODO: used defined*/
            break;
        default:
            /*handle the unexpected*/
            break
    }
}
```

As you can see in the example below, the surrounding execution case its checked through the FSM handle by reading the `Status` field.

3.2.6 Running a state machine : `qStateMachine_Run`

This API is used to execute the Finite State Machine. Only a single cycle is performed propagating the input signal until it is handled.

```
qBool_t qStateMachine_Run( qSM_t * const m, qSM_Signal_t xSignal )
```

Parameters

- `obj` : a pointer to the FSM object.
- `xSignal` : User-defined signal (this value will be ignored if the installed queue has items available)

3.2.7 Adding a state machine as a task : `qOS_Add_StateMachineTask`

The best strategy to run a FSM is delegating it to a task. For this, the provided API should be used. Here, the task doesn't have a specific callback, instead, it will evaluate the active state of the 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_Run()` can be omitted.

```
qBool_t qOS_Add_StateMachineTask( qTask_t * const Task, qSM_t *m,
                                  qPriority_t Priority,
                                  qTime_t Time, qState_t InitialTaskState,
                                  void *arg )
```

Parameters

- `Task` : A pointer to the task node.
- `m` : A pointer to Finite State-Machine (FSM) object.
- `Priority` : The priority value. [0(min) - `Q_PRIORITY_LEVELS`(max)]
- `Time` : Execution interval defined in seconds (floating-point format). For immediate execution use the `qTimeImmediate` definition.
- `InitialTaskState` : Specifies the initial operational state of the task (`qEnabled`, `qDisabled`, `qAsleep` or `qAwake`).
- `arg` - Represents the task arguments. All arguments must be passed by reference and cast to `(void *)`.

↪*Note*: For the `Surrounding` 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 `qSM_Handler_t` argument. Check the example below:

```
qSM_Status_t Example_State( qSM_Handler_t h ) {
    qEvent_t e = h->Data;
    /* Get the event info of the task that owns this state-machine*/
    switch ( h->Signal ) {
        case QSM_SIGNAL_ENTRY:
            break;
        case QSM_SIGNAL_EXIT:
            break;
        default:
            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 */
            break;
    }
    return qSM_STATUS_EXIT_SUCCESS;
}
```

3.2.8 A demonstrative example

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

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

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

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

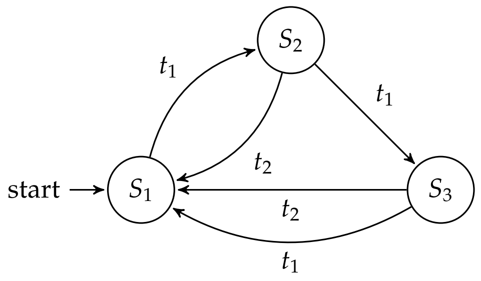

Figure 17: A simple FSM example with three states

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

```
qTask_t LED_Task; /*The task node*/
qSM_t LED_FSM; /*The state-machine handle*/
qSM_State_t State_LEDOff, State_LEDOn, State_LEDBlink;
```

Then, we define our states as the flow-diagram showed in figure 17.

```
qSM_Status_t State_LEDOff_Callback( qSM_Handler_t h ) {
    switch ( h->Signal ) {
        case QSM_SIGNAL_ENTRY:
            BSP_LED_OFF();
            break;
        case QSM_SIGNAL_EXIT: case QSM_SIGNAL_START: /*Ignore*/
            break;
        default:
            if ( BUTTON_PRESSED ) {
                h->NextState = &State_LEDOn;
            }
            break;
    }
    return qSM_STATUS_EXIT_SUCCESS;
}
/*----------------------------------------------------------------------*/
qSM_Status_t State_LEDOn_Callback( qSM_Handler_t h ) {
    static qSTimer_t timeout;
    switch ( h->Signal ) {
        case QSM_SIGNAL_ENTRY:
            qSTimer_Set( &timeout, 10.0f ); /*STimer gets armed*/
            BSP_LED_ON();
            break;
        case QSM_SIGNAL_EXIT: case QSM_SIGNAL_START: /*Ignore*/
            break;
        default:
            if ( qSTimer_Expired( &timeout) ) { /*check if the timeout expired
    */
                h->NextState = &State_LEDOff;
            }
            if ( BUTTON_PRESSED ) {
                h->NextState = &State_LEDBlink;
            }
            break;
    }
    return qSM_STATUS_EXIT_SUCCESS;
}
/*----------------------------------------------------------------------*/
qSM_Status_t State_LEDBlink_Callback( qSM_Handler_t h ) {
    static qSTimer_t timeout;
    static qSTimer_t blinktime;

    switch ( m->Signal ) {
        case QSM_SIGNAL_ENTRY:
            qSTimer_Set( &timeout, 10.0f );
            break;
        case QSM_SIGNAL_EXIT: case QSM_SIGNAL_START: /*Ignore*/
            break;
        default:
            if ( qSTimer_Expired( &timeout ) || BUTTON_PRESSED ) {
                h->NextState = &State_LEDOff;
            }
            if ( qSTimer_FreeRun( &blinktime, 0.5f ) ) {
               BSP_LED_TOGGLE();
            }
            break;
    }
    return qSM_STATUS_EXIT_SUCCESS;
}
```

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

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

```
qStateMachine_Setup( &LED_FSM, NULL, &State_LEDOff, NULL, NULL );
qStateMachine_StateSubscribe( &LED_FSM, &State_LEDOff, NULL,
                              State_LEDOff_Callback, NULL, NULL );
```

```
qStateMachine_StateSubscribe( &LED_FSM, &State_LEDOn, NULL,
                              State_LEDOn_Callback, NULL, NULL );
qStateMachine_StateSubscribe( &LED_FSM, &State_LEDBlink, NULL,
                              State_LEDBlink_Callback, NULL, NULL );
qOS_Add_StateMachineTask( &LED_Task, &LED_FSM, qHigh_Priority,
                          0.1, qEnabled, NULL );
```

3.2.9 Sending signals

To communicate within and between state-machines or even other contexts, use signals. A signal is a simple value who can be used to abstract an incoming event. In the receiving state-machine, a queue or a exclusion variable receives the signal and holds it until the state-machine can evaluate it.

When coding state-machines, the application writer can benefit from this simple event-abstraction mechanism. On the one hand, there would be a more uniform programming when writing states callbacks and on the other hand, the communication of the state machine from other contexts becomes easier.

To send a signal to a state machine, use the following API:

```
qBool_t qStateMachine_SendSignal( qSM_t * const m, qSM_Signal_t xSignal,
                                  void *sigData, const qBool_t isUrgent )
```

Parameters

- `m` : A pointer to the FSM object.
- `xSignal` : The user-defined signal.
- `sigData` : The data associated to the signal.
- `isUrgent` : If `qTrue`, the signal will be sent to the front of the queue. (only if the there is a signal-queue available)

Return Value

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

`qFalse` if there is a queue, and the signal cannot be inserted because it is full or the exclusion variable has not been freed.

This API can manage their delivery to one of these possible destinations: an *exclusion variable* or a *signal queue*:

- An *exclusion variable* its a variable with an important distinction, it can only be written if it is empty. The empty situation only happens, if the engine has already propagated the signal within the state machine. If the signal has not yet propagated, the signal sending cannot be carried out.

- When a *signal queue* is used, the signal is put into a FIFO structure and the engine takes care of dispatching the signal in an orderly manner. The only situation where the signal cannot be delivered is if the queue is full. This its the preferred destination, as long as there is a previously installed signal queue (for this, please review the next section.

➥*Note*: If the signal-queue its available, the `qStateMachine_SendSignal` will always select it as destination.

➥If a state-machine, a task, or another context sends a signal to a full queue, an queue-overflow occurs. The result of the queue overflow it that the state-machine drops the new signal.

3.2.10 Installing a signal queue

A state machine can have a FIFO queue to allow the delivery of signals from another contexts. If the signal queue its installed, the state-machine engine constantly monitors the queue for available signals. The engine then propagates the signal through the hierarchy until it is processed. To enable this functionality in your state machine, the queue must be installed by using the `qStateMachine_InstallSignalQueue` API.

The install operation should be performed after both, the queue and the FSM are correctly initialized by using `qQueue_Setup` and `qStateMachine_Setup` respectively.

```
qBool_t qStateMachine_InstallSignalQueue( qSM_t * const m, qQueue_t *queue );
```

Parameters

- `m` : A pointer to the FSM object.
- `queue` : A pointer to the queue instance.

Return Value

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

➥*Note*: Make sure that queues are enabled in the `qconfig.h`.

☞When configuring a signal queue with `qQueue_Setup`, remember to size it based on the type `qSM_Signal_t`.

☞If the state-machines its delegated to a task, make sure to install the queue prior to setting up the task. In this way, a kernel connection can be performed between the FSM signal-queue and the FSM-task, allowing the OS to catch signals to produce a task event, this prevents the wait of the task for the specified period, resulting in a faster handling of incoming signals.

3.2.11 Using a transition table

In this approach, the FSM is coded in tables with the outgoing transitions of every state, where each entry relates signals, actions and the target state. This is an elegant method to translate the FSM to actual implementation as the handling for every state and event combination is encapsulated in the table.

Signal Id	*Signal action/guard*	*Target state*	*History mode*	*Signal data*
`Signal1`	`NULL`	`StateB`	0	`NULL`
`Signal3`	`DoOnSignal3`	`StateD`	0	`sig3_data`
...	...	...	...	...
`Signal6`	`NULL`	`StateA`	0	`NULL`

Table 1: Transition table layout for a state

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

The API `qStateMachine_Set_StateTransitions()`, should be used to perform the transition table installation to a specific state.

```
qBool_t qStateMachine_Set_StateTransitions( qSM_State_t * const state,
                                            qSM_Transition_t * const table,
                                            const size_t n )
```

Parameters

- `m` : A pointer to the state object.
- `table` : The array of entries (`qSM_Transition_t[]`) that describes the outgoing transitions.
- `n` : The number of transitions available within the `table`.

Return Value

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

Caveats

- State transitions are not limited to the specification of the transition table. A state callback owns the higher precedence to change a state. The application writer can use both, a transition table and direct `NextState` field manipulation in state callbacks to perform a transition to the FSM.
- Special care is required when the table grows very large, that is, when there are many invalid state/event combinations, leading to a waste of memory. There is also a memory penalty as the number of states and events grow. The application writer need to accurately account for this during initial design. A statechart pattern can be used to improve the design and reduce the number of transition entries.
- The user is responsible for defining the transitions according to the topology of the state machine. Undefined behaviors can occur if the topology is broken with poorly defined transitions.

➥*Note*: When a transition entry is defined. the event-signal should be located as the first parameter of the entry. Please see the transition layout in table 1.

3.2.12 Signal actions and guards:

Transition tables allow the usage of this feature. When an event-signal is received from the queue, the signal-action, if available, is evaluated before the transition is triggered. This action is user-defined and should be coded as a function that takes a `qSM_Handler_t` object and returns a value of type `qBool_t`.

```
qBool_t Signal_Action( qSM_Handler_t h ){
    /* TODO : Event-signal action*/
    return qTrue; /*allow the state transition*/
}
```

The return value is checked after to allow or reject the state transition. The application writer can code a boolean expression to implement statechart guards or perform some pre-transition procedure.

☞If a signal-action returns `qFalse`, the event-signal is rejected, preventing the state transition to be performed in the calling FSM.

➥*Note*: When a transition entry is defined. the signal-action should be located as the third parameter of the entry. Please see the transition layout in table 1. A `NULL` value will act as a NOT-defined, always allowing the state-transition.

3.2.13 FSM Timeout specification

A timeout specification is mechanism to simplify the notion of time passage inside states. The basic usage model of the timeout signals is as follows:

An timeout specification allocates one or more timer objects. The user relates in a table each specific timeout operations within the state where are they going to operate. So, according to the table, when a state needs to arrange for a timeout, the engine can set or reset the given timer. When the FSM engine detects that the appropriate moment has arrived (a timer expiration occurs), it inserts the timeout signal directly into the recipient's event queue. The recipient then processes the timeout signal just like any other signal.

Given the above explanation, it is evident that for its operation, the state machine requires an installed signal queue.

A timeout specification is referenced by an object of type `qSM_TimeoutSpec_t` and must be installed inside the state machine using the API `qStateMachine_InstallTimeoutSpec()`.

```
qBool_t qStateMachine_InstallTimeoutSpec( qSM_t * const m,
                                          qSM_TimeoutSpec_t * const ts );
```

Parameters

- `m` : A pointer to the FSM object.
- `ts` : A pointer to the timeout specification object.

Return Value

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

Then, timeout operations can be defined in a table for each state using the following API:

```
qBool_t qStateMachine_Set_StateTimeouts( qSM_State_t * const state,
                                         qSM_TimeoutStateDefinition_t *tdef,
                                         const size_t n  );
```

Parameters

- `state` : A pointer to the state object.
- `tdef` : The lookup table matching the requested timeout values with their respective options.
- `n` : the number of elements inside `tdef`.

Return Value

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

A timeout specification element is defined as an structure of type `qSM_TimeoutStateDefinition_t` and should follow this layout:

Timeout value	*Options*

Table 2: Timeout specification layout

The options for every timeout its a bitwise value that indicates which timeout should be used and the operations than should be performed internally by the state-machine engine. This options can be combined with a bitwise OR and are detailed as follows:

- `QSM_TSOPT_INDEX(index)` : To select the timeout to be used in the specification. Should be a value between `0` and `(Q_FSM_MAX_TIMEOUTS-1)`
- `QSM_TSOPT_SET_ENTRY` : To set the timeout when the specified state its entering.
- `QSM_TSOPT_RST_ENTRY` : To reset the timeout when the specified state its entering.
- `QSM_TSOPT_SET_EXIT` : To set the timeout when the specified state its exiting.
- `QSM_TSOPT_RST_EXIT` : To reset the timeout when the specified state its exiting.
- `QSM_TSOPT_KEEP_IF_SET` : To apply the Set operation only if the timeout its in a reset state.
- `QSM_TSOPT_PERIODIC` : To put the timeout in periodic mode.

☞Data associated to timeout signals should be set to `NULL`. Any other value will be ignored and will be passed as `NULL` to the FSM handler.

☞The user is responsible for writing timeout specifications correctly. Care must be taken that the specifications do not collide between hierarchical states to avoid overwriting operations.

☞You can increase the number of available timeouts instances by changing the `Q_FSM_MAX_TIMEOUTS` configuration macro inside `qconfig.h`.

3.2.14 Demonstrative example using transition tables

The following example shows the implementation of the FSM presented in section 3.2.8 using the transition table approach with signal-queue and a timeout specification.

Before getting started, the required variables should be defined:

```
/*define the FSM application event-signals*/
#define SIGNAL_BUTTON_PRESSED   ( (qSM_SigId_t)1 )
#define SIGNAL_TIMEOUT          ( QSM_SIGNAL_TIMEOUT(0) )
#define SIGNAL_BLINK            ( QSM_SIGNAL_TIMEOUT(1) )

qTask_t LED_Task; /*The task node*/
qSM_t LED_FSM; /*The state-machine handler*/
qSM_State_t State_LEDOff, State_LEDOn, State_LEDBlink;
qQueue_t LEDsigqueue; /*the signal-queue*/
qSM_Signal_t led_sig_stack[ 5 ];  /*the signal-queue storage area*/
qSM_TimeoutSpec_t tm_spectimeout;

/*create the transition tables for every state*/
qSM_Transition_t LEDOff_transitions[] = {
    { SIGNAL_BUTTON_PRESSED, NULL, &State_LEDOn    ,0, NULL}
};

qSM_Transition_t LEDOn_transitions[] = {
    { SIGNAL_TIMEOUT,        NULL, &State_LEDOff   ,0, NULL},
    { SIGNAL_BUTTON_PRESSED, NULL, &State_LEDBlink ,0, NULL},
};

qSM_Transition_t LEDBlink_transitions[] = {
    { SIGNAL_TIMEOUT,        NULL, &State_LEDOff   ,0, NULL},
    { SIGNAL_BUTTON_PRESSED, NULL, &State_LEDOff   ,0, NULL}
};

/*define the timeout specifications */
qSM_TimeoutStateDefinition_t LedOn_Timeouts[]={
    { 10.0f,  QSM_TSOPT_INDEX(0) | QSM_TSOPT_SET_ENTRY | QSM_TSOPT_RST_EXIT  },
};

qSM_TimeoutStateDefinition_t LEDBlink_timeouts[]={
    { 10.0f,  QSM_TSOPT_INDEX(0) | QSM_TSOPT_SET_ENTRY | QSM_TSOPT_RST_EXIT  },
    { 0.5f,   QSM_TSOPT_INDEX(1) | QSM_TSOPT_SET_ENTRY | QSM_TSOPT_RST_EXIT |
    QSM_TSOPT_PERIODIC  },
};
```

Then, we define the callback for the states.

```
qSM_Status_t State_LEDOff_Callback( qSM_Handler_t h ) {
    switch ( h->Signal ) {
        case QSM_SIGNAL_ENTRY:
            BSP_LED_OFF();
            break;
        default:
            break;
    }
    return qSM_STATUS_EXIT_SUCCESS;
}
/*-------------------------------------------------------------------*/
qSM_Status_t State_LEDOn_Callback( qSM_Handler_t h ) {
    switch ( h->Signal ) {
        case QSM_SIGNAL_ENTRY:
            BSP_LED_ON();
            break;
        default:
            break;
    }
    return qSM_STATUS_EXIT_SUCCESS;
}
/*-----------------------------------------------------------------------*/
qSM_Status_t State_LEDBlink_Callback( qSM_Handler_t h ) {
    switch ( h->Signal ) {
        case SIGNAL_BLINK:
            BSP_LED_TOGGLE();
            break;
        default:
            break;
    }
    return qSM_STATUS_EXIT_SUCCESS;
}
```

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

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

```
qStateMachine_Setup( &LED_FSM, NULL, &State_LEDOff, NULL, NULL );
qStateMachine_StateSubscribe( &LED_FSM, &State_LEDOff, QSM_STATE_TOP,
    State_LEDOff_Callback, NULL, NULL );
qStateMachine_StateSubscribe( &LED_FSM, &State_LEDOn, QSM_STATE_TOP,
    State_LEDOn_Callback, NULL, NULL );
qStateMachine_StateSubscribe( &LED_FSM, &State_LEDBlink, QSM_STATE_TOP,
    State_LEDBlink_Callback, NULL, NULL );

qQueue_Setup( &LEDsigqueue, led_sig_stack, sizeof(qSM_Signal_t), qFLM_ArraySize
    (led_sig_stack) );
qStateMachine_InstallSignalQueue( &LED_FSM, &LEDsigqueue );

qStateMachine_InstallTimeoutSpec( &LED_FSM, &tm_spectimeout );
qStateMachine_Set_StateTimeouts( &State_LEDOn, LedOn_Timeouts, qFLM_ArraySize(
    LedOn_Timeouts) );
qStateMachine_Set_StateTimeouts( &State_LEDBlink, LEDBlink_timeouts,
    qFLM_ArraySize(LEDBlink_timeouts) );

qStateMachine_Set_StateTransitions( &State_LEDOff, LEDOff_transitions,
    qFLM_ArraySize(LEDOff_transitions) );
qStateMachine_Set_StateTransitions( &State_LEDOn, LEDOn_transitions,
    qFLM_ArraySize(LEDOn_transitions) );
qStateMachine_Set_StateTransitions( &State_LEDBlink, LEDBlink_transitions,
    qFLM_ArraySize(LEDBlink_transitions) );

qOS_Add_StateMachineTask(  &LED_Task, &LED_FSM, qMedium_Priority, 0.1f,
    qEnabled, NULL  );
```

3.2.15 Demonstrative example using the hierarchical approach

In conventional state machine designs, all states are considered at the same level. The design does not capture the commonality that exists among states. In real life, many states handle most transitions in similar fashion and differ only in a few key components.

Even when the actual handling differs, there is still some commonality. It is in these situations where the hierarchical designs makes the most sense.

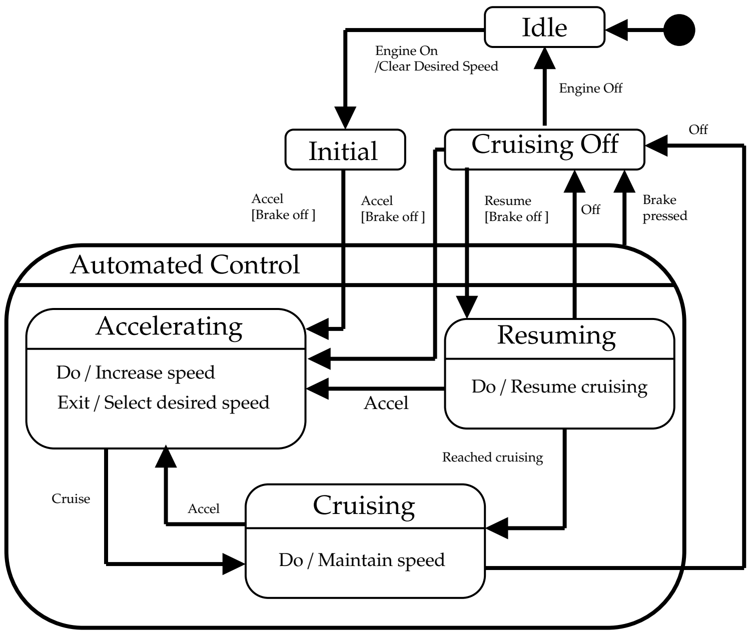

Figure 18: Cruise control FSM example

A hierarchical state-machine is characterized by having compound states. A composite state is defined as state that has inner states and can be used as a decomposition mechanism that allows factoring of common behaviors and their reuse. And this is the biggest advantage of this design, because it captures the commonality by organizing the states as a hierarchy. The states at the higher level in hierarchy perform the common handling, while the lower level states inherit the commonality from higher level ones and perform the state specific functions.

This example takes the "Cruise Control" study case from [4], a real-time system that manages the speed of an automobile based on inputs from the driver.

The behavior of this system is state-dependent in that the executed actions correspond not only to the driver input, but also on the current state of the system and with the status of the engine and the brake.

The figure 18 illustrate the modeling of this system with the "Automated Control" state acting as composite.

Before getting started, the required user-defined signals, variables, and entries of the transition table should be defined:

```
#define  SIGNAL_ENGINE_ON          ( (qSM_SigId_t)(1) )
```

```
#define  SIGNAL_ACCEL               ( (qSM_SigId_t)(2) )
#define  SIGNAL_RESUME              ( (qSM_SigId_t)(3) )
#define  SIGNAL_OFF                 ( (qSM_SigId_t)(4) )
#define  SIGNAL_BRAKE_PRESSED       ( (qSM_SigId_t)(5) )
#define  SIGNAL_CRUISE              ( (qSM_SigId_t)(6) )
#define  SIGNAL_REACHED_CRUISING    ( (qSM_SigId_t)(7) )
#define  SIGNAL_ENGINE_OFF          ( (qSM_SigId_t)(8) )

qTask_t CruiseControlTask;
qSM_t Top_SM;

/*highest level states*/
qSM_State_t state_idle, state_initial, state_cruisingoff,
    state_automatedcontrol;
/*states inside the state_automatedcontrol*/
qSM_State_t state_accelerating, state_cruising, state_resuming;

qQueue_t top_sigqueue;
qSM_Signal_t topsm_sig_stack[ 10 ];

/*======================================================================*/
/*                              TRANSITION TABLES                        */
/*======================================================================*/
qSM_Transition_t idle_transitions[] =
{
{ SIGNAL_ENGINE_ON, SigAct_ClearDesiredSpeed, &state_initial      ,0, NULL }
};

qSM_Transition_t initial_transitions[] =
{
{ SIGNAL_ACCEL,     SigAct_BrakeOff,          &state_accelerating ,0, NULL }
};

qSM_Transition_t accel_transitions[] =
{
{ SIGNAL_CRUISE,     NULL,                    &state_cruising     ,0, NULL }
};

qSM_Transition_t cruising_transitions[] =
{
{ SIGNAL_OFF,       NULL,                     &state_cruisingoff  ,0, NULL }
{ SIGNAL_ACCEL,     NULL,                     &state_accelerating ,0, NULL }
};

qSM_Transition_t resuming_transitions[] =
{
{ SIGNAL_ACCEL,     NULL,                     &state_accelerating ,0, NULL }
};

qSM_Transition_t cruisingoff_transitions[] =
{
{ SIGNAL_ACCEL,      SigAct_BrakeOff,         &state_accelerating ,0, NULL },
{ SIGNAL_RESUME,     SigAct_BrakeOff,         &state_resuming     ,0, NULL },
{ SIGNAL_ENGINE_OFF, NULL,                    &state_idle         ,0, NULL }
};

qSM_Transition_t automated_transitions[] =
{
{ SIGNAL_BRAKE_PRESSED,   NULL,               &state_cruisingoff  ,0, NULL }
};
/*---------------------------------------------------------------------*/
```

Then, signal-actions and state callbacks are later defined:

```
/*=======================================================================*/
/*                      EVENT-SIGNAL ACTIONS AND GUARDS                  */
/*=======================================================================*/
qBool_t SigAct_ClearDesiredSpeed( qSM_Handler_t h ) {
    (void)h;
    Speed_ClearDesired();
    return qTrue;
}
/*---------------------------------------------------------------------*/
qBool_t SigAct_BrakeOff( qSM_Handler_t h ) {
    (void)h; /*unused*/
    return ( BSP_BREAK_READ() == OFF ) ? qTrue : qFalse;  /*check guard*/
}
/*=======================================================================*/
/*                  STATE CALLBACK FOR THE TOP FSM                     */
/*=======================================================================*/
qSM_Status_t state_top_callback( qSM_Handler_t h ) {
    qSM_Status_t RetVal = qSM_STATUS_EXIT_SUCCESS;
    switch ( h->Signal ) {
        case QSM_SIGNAL_ENTRY:
            break;
        case QSM_SIGNAL_EXIT:
            break;
    }
    return RetVal;
}
/*=======================================================================*/
/*                  CALLBACKS FOR THE STATES ABOVE TOP                     */
/*=======================================================================*/
qSM_Status_t state_idle_callback( qSM_Handler_t h ) {
    /*TODO : state activities*/
    return qSM_EXIT_SUCCESS;
}
/*---------------------------------------------------------------------*/
qSM_Status_t state_initial_callback( qSM_Handler_t h ) {
    /*TODO : state activities*/
    return qSM_EXIT_SUCCESS;
}
/*---------------------------------------------------------------------*/
qSM_Status_t state_cruisingoff_callback( qSM_Handler_t h ) {
    /*TODO : state activities*/
    return qSM_EXIT_SUCCESS;
}
/*---------------------------------------------------------------------*/
qSM_Status_t state_automatedcontrol_callback( qSM_Handler_t h ) {
    /*TODO : state activities*/
    return qSM_EXIT_SUCCESS;
}
/*=======================================================================*/
/*          STATE CALLBACKS FOR THE AUTOMATED CONTROL FSM                */
/*=======================================================================*/
qSM_Status_t state_accelerating_callback( qSM_Handler_t h ) {
    switch( h->Signal ) {
        case QSM_SIGNAL_EXIT:
            Speed_SelectDesired();
            break;
        default:
            Speed_Increase();
            break;
    }
    return qSM_EXIT_SUCCESS;
}
/*---------------------------------------------------------------------*/
qSM_Status_t state_resuming_callback( qSM_Handler_t h ) {
    Cruising_Resume();
    return qSM_EXIT_SUCCESS;
}
/*---------------------------------------------------------------------*/
qSM_Status_t state_cruising_callback( qSM_Handler_t h ) {
    Speed_Maintain();
    return qSM_EXIT_SUCCESS;
}
```

Finally, the dedicated task for the FSM and related objects are configured.

```
qStateMachine_Setup( &Top_SM, state_top_callback, &state_idle, NULL, NULL );
/*subscribe to the highest level states*/
qStateMachine_StateSubscribe( &Top_SM, &state_idle, QSM_STATE_TOP,
    state_idle_callback, NULL, NULL );
qStateMachine_StateSubscribe( &Top_SM, &state_initial, QSM_STATE_TOP,
    state_initial_callback, NULL, NULL );
qStateMachine_StateSubscribe( &Top_SM, &state_cruisingoff, QSM_STATE_TOP,
    state_cruisingoff_callback, NULL, NULL );
qStateMachine_StateSubscribe( &Top_SM, &state_automatedcontrol, QSM_STATE_TOP,
    state_automatedcontrol_callback, NULL, NULL );
/*subscribe to the states within the state_automatedcontrol*/
qStateMachine_StateSubscribe( &Top_SM, &state_accelerating, &
    state_automatedcontrol, state_accelerating_callback, NULL, NULL );
qStateMachine_StateSubscribe( &Top_SM, &state_resuming, &state_automatedcontrol
    , state_resuming_callback, NULL, NULL );
qStateMachine_StateSubscribe( &Top_SM, &state_cruising, &state_automatedcontrol
    , state_cruising_callback, NULL, NULL );

qQueue_Setup( &top_sigqueue, topsm_sig_stack, sizeof(qSM_Signal_t),
    qFLM_ArraySize(topsm_sig_stack) );
qStateMachine_InstallSignalQueue( &Top_SM, &top_sigqueue );

qStateMachine_Set_StateTransitions( &state_idle, idle_transitions,
    qFLM_ArraySize(idle_transitions) );
qStateMachine_Set_StateTransitions( &state_initial, initial_transitions,
    qFLM_ArraySize(initial_transitions) );
qStateMachine_Set_StateTransitions( &state_cruisingoff, cruisingoff_transitions
    , qFLM_ArraySize(cruisingoff_transitions) );
qStateMachine_Set_StateTransitions( &state_automatedcontrol,
    automated_transitions, qFLM_ArraySize(automated_transitions) );
qStateMachine_Set_StateTransitions( &state_accelerating, accel_transitions,
    qFLM_ArraySize(accel_transitions) );
qStateMachine_Set_StateTransitions( &state_resuming, resuming_transitions,
    qFLM_ArraySize(resuming_transitions) );
qStateMachine_Set_StateTransitions( &state_cruising, cruising_transitions,
    qFLM_ArraySize(cruising_transitions) );
```

```
qOS_Add_StateMachineTask(  &CruiseControlTask, &Top_SM, qMedium_Priority, 0.1f,
     qEnabled, NULL  );
```

3.2.16 Demonstrative example with history pseudo-states

State transitions defined in high-level composite states often deal with events that require immediate attention; however, after handling them, the system should return to the most recent substate of the given composite state. UML statecharts address this situation with two kinds of history pseudostates: *shallow history* and *deep history*(denoted as the circled H and H* icon respectively in figure).

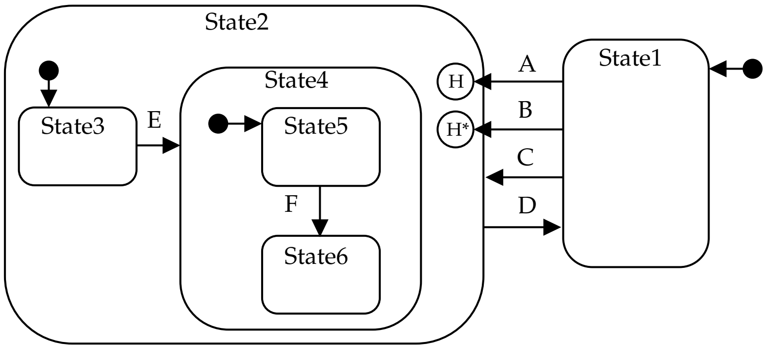

Figure 19: Example with history pseudo-states

Shallow history A transition to the shallow history state in a composite state invokes the last state that was active, at the same depth as the history state itself, prior to the most recent exit of the composite state.

Deep history A transition to the deep history state within a composite state invokes the state that was active, immediately before the most recent exit of the composite state. The last active state can be nested at any depth.

Here, the way to specify this type of transitions in QuarkTS is very straightforward, you only need to assign the history-mode in the last entry of the transition as shown below:

```
#define SIGNAL_A        ( (qSM_SigId_t)(1) )
#define SIGNAL_B        ( (qSM_SigId_t)(2) )
#define SIGNAL_C        ( (qSM_SigId_t)(3) )
#define SIGNAL_D        ( (qSM_SigId_t)(4) )
#define SIGNAL_E        ( (qSM_SigId_t)(5) )
#define SIGNAL_F        ( (qSM_SigId_t)(6) )

qQueue_t sigqueue;
qSM_Signal_t topsm_sig_stack[ 10 ];
qSM_t super;
qSM_State_t state1, state2, state3, state4, state5, state6;

qSM_Transition_t state1_transitions[] = {
```

```
{ SIGNAL_A, NULL,   &state2,  qSM_TRANSITION_SHALLOW_HISTORY, NULL  },
{ SIGNAL_B, NULL,   &state2,  qSM_TRANSITION_DEEP_HISTORY, NULL  },
{ SIGNAL_C, NULL,   &state2,  qSM_TRANSITION_NO_HISTORY, NULL   }
};

qSM_Transition_t state2_transitions[] = {
{ SIGNAL_D, NULL,   &state1, 0, NULL }
};

qSM_Transition_t state3_transitions[] = {
{ SIGNAL_E, NULL,   &state4,  0, NULL }
};

qSM_Transition_t state5_transitions[] = {
{ SIGNAL_F, NULL,   &state6,  0, NULL }
};
```

And here, the configuration and topology of the state-machine is presented, including the default transitions (the small circles filled with black). Please don't forget to define the callbacks for each state.

```
qStateMachine_Setup( &super, state_top_callback, &state1, NULL, NULL );
qStateMachine_StateSubscribe( &super, &state1, QSM_STATE_TOP, state1_callback,
    NULL, NULL );
qStateMachine_StateSubscribe( &super, &state2, QSM_STATE_TOP, state2_callback,
    &state3, NULL );
qStateMachine_StateSubscribe( &super, &state3, &state2, state3_callback, NULL,
    NULL );
qStateMachine_StateSubscribe( &super, &state4, &state2, state4_callback, &
    state5, NULL );
qStateMachine_StateSubscribe( &super, &state5, &state4, state5_callback, NULL,
    NULL );
qStateMachine_StateSubscribe( &super, &state6, &state4, state6_callback, NULL,
    NULL );

qQueue_Setup( &sigqueue, topsm_sig_stack, sizeof(qSM_Signal_t), qFLM_ArraySize(
    topsm_sig_stack) );
qStateMachine_InstallSignalQueue( &super, &sigqueue );

qStateMachine_Set_StateTransitions( &state1, state1_transitions, qFLM_ArraySize
    (state1_transitions) );
qStateMachine_Set_StateTransitions( &state2, state2_transitions, qFLM_ArraySize
    (state2_transitions) );
qStateMachine_Set_StateTransitions( &state3, state3_transitions, qFLM_ArraySize
    (state3_transitions) );
qStateMachine_Set_StateTransitions( &state5, state5_transitions, qFLM_ArraySize
    (state5_transitions) );

qOS_Add_StateMachineTask(  &SMTask, &super, qMedium_Priority, 0.1f, qEnabled,
    NULL  );
```

3.3 Co-Routines

As showed in figure 20, 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.

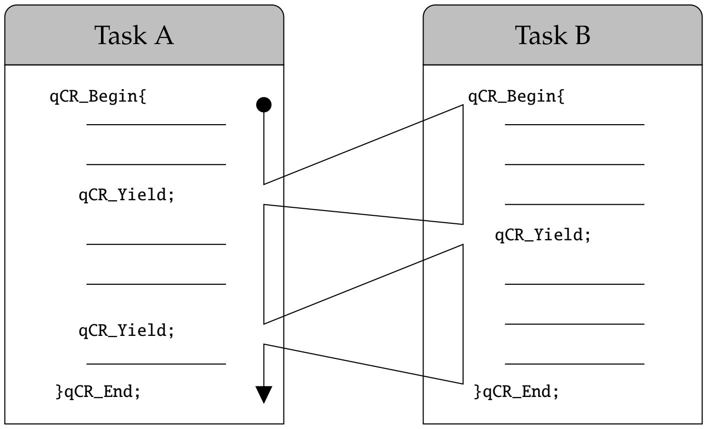

Figure 20: Coroutines in QuarkTS

The QuarkTS implementation uses the Duff's device approach, and is heavily inspired by the Knuth method[5], Simon Tatham's Co-Routines in C [6] and Adam Dunkels Protothreads [7]. This means that a *local-continuation* variable is used to preserve the current state of execution at a particular place of the Co-Routine scope but without any call history or local variables. This brings benefits to lower RAM usage, but at the cost of some restrictions on how a Co-routine can be used.

Limitations and Restrictions:

- The stack of a Co-Routine is not maintained when a yield is performed. 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`.
- Calls to API functions that could cause the Co-Routine to block, can only be made from the Co-Routine function itself - not from within a function called by the Co-Routine .
- The implementation does not permit yielding or blocking calls to be made from within a `switch` statement.

3.3.1 Coding a Co-Routine

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

```
void CoroutineTask_Callback( qEvent_t e ) {
    qCR_Begin {
        if ( EventNotComing() ) {
            qCR_Yield;
        }
        DoTheEventProcessing();
        qCR_Delay( WAIT_TIME_S );
        PerformActions();
    } qCR_End;
}
```

The `qCR_Begin` statement should be placed at the start of the task function in which the Co-routine runs. All C statements above the `qCR_Begin` will be executed as if they were in an endless-loop each time the task is scheduled.

A `qCR_Yield` statement return the CPU control back to the scheduler but saving the execution progress, thereby allowing other processing tasks to take place in the system. With the next task activation, the Co-Routine will resume the execution after the last `qCR_Yield` statement.

☞All the Co-routine statements has the *qCR* appended at the beginning of their name.

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

☞Do not use an endless-loop inside a Co-routine ,this behavior it's already hardcoded within the segment definition.

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 is to wait for a fixed period of time. For this, the `qCR_Delay()` should be used .

```
qCR_Delay( qTime_t tDelay )
```

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

Another common blocking call is `qCR_WaitUntil()` :

```
qCR_WaitUntil( Condition )
```

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

An additional wait statement is also provided that sets a timeout for the logical condition to be met, with a similar behavior of `qCR_WaitUntil`.

```
qCR_TimedWaitUntil( Condition, qTime_t Timeout )
```

Optionally, the `Do-Until` structure gives to application writer the ability to perform a multi-line job before the yield, allowing more complex actions to being performed after the Co-Routine resumes:

```
qCR_Do{
    /* Job : a set of instructions*/
}qCR_Until( Condition );
```

Usage example:

```
void Sender_Task( qEvent_t e ) {
    static qSTimer_t timeout;
    qCR_Begin {
        Send_Packet();
        /*
           Wait until an acknowledgment has been received, or until
           the timer expires. If the timer expires, we should send
           the packet again.
        */
        qSTimer_Set( &timeout, TIMEOUT_TIME );
        qCR_WaitUntil( PacketACK_Received() ||
                       qSTimer_Expired(&timeout));
    } qCR_End;
}
/*====================================================================*/
void Receiver_Task( qEvent_t e ) {
    qCR_Begin {
        /* Wait until a packet has been received*/
        qCR_WaitUntil( Packet_Received() );
        Send_Acknowledgement();
    } qCR_End;
}
```

3.3.3 Positional jumps

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

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

```
qCR_PositionGet( qCR_Position_t CRPos )
```

```
qCR_PositionRestore( qCR_Position_t CRPos )
```

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

```
qCR_PositionReset( qCR_Position_t CRPos )
```

3.3.4 Semaphores

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

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

```
qCR_SemInit( qCR_Semaphore_t *sem, qUINT16_t Value )
```

To perform the *wait* operation, the `qCR_SemWait()` 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.

```
qCR_SemWait( qCR_Semaphore_t *sem )
```

Finally, `qCR_SemSignal()` 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.

```
qCR_SemSignal( qCR_Semaphore_t *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 `qCR_SemWait` statement.

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

#define NUM_ITEMS 32
#define BUFSIZE 8

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

  qCR_Begin{
     for ( produced = 0 ; produced < NUM_ITEMS; ++produced ) {
        qCR_SemWait( &full );
        qCR_SemWait( &mutex );

        add_to_buffer( produce_item() );

        qCR_SemSignal( &mutex );
        qCR_SemSignal( &empty );
     }
  }qCR_End;
}
/*===================================================================*/
void ConsumerTask_Callback( qEvent_t e ) {
  static int consumed;

  qCR_Begin {
      for( onsumed = 0 ; consumed < NUM_ITEMS; ++consumed ) {
        qCR_SemWait( &empty );
        qCR_SemWait( &mutex );

        consume_item( get_from_buffer() );

        qCR_SemSignal( &mutex );
        qCR_SemSignal( &full );
      }
  } qCR_End;
}
/*===================================================================*/
void IdleTask_Callback( qEvent_t e ) {
    /*nothing to do*/
}
/*===================================================================*/
int main(void) {
  HAL_Init();

  qOS_Setup( HAL_GetTick, 0.001, IdleTask_Callback );
  qCR_SemInit( &empty, 0 );
  qCR_SemInit( &full, BUFSIZE );
  qCR_SemInit( &mutex, 1 );
  qOS_Add_Task( &ProducerTask, ProducerTask_Callback,
                qMedium_Priority, 0.1f, qPeriodic, qEnabled, NULL );
  qOS_Add_Task( &ConsumerTask, ConsumerTask_Callback,
                qMedium_Priority, 0.1f, qPeriodic, qEnabled, NULL );
  qOS_Run();
  return 0;
}
```

3.3.5 External control

There are several circumstances where becomes necessary to control the flow of execution outside the segment that defines the Co-routine itself. This is usually used to defer the job of the Co-routine or resume it in response to specific occurrences that arises in other contexts, either tasks or interrupts.

To code this specific situations, a handler to the Co-routine should be defined, a variable of type `qCR_Handle_t`. In addition to this, the scope of the target Co-routine must be started with the `qCR_BeginWithHandle` statement instead of `qCR_Begin`.

```
qCR_Handle_t xHandleCR = NULL; /*NULL initialization are strictly necessary*/

/*==================================================================*/
void AnotherTask_Callback( qEvent_t e ) {
    int UserInput = 0;
    if ( e->FirstIteration ) {
        qCR_ExternControl( xHandleCR, qCR_RESUME, 0 );
    }
    if ( e->LastIteration ) {
        qCR_ExternControl( xHandleCR, qCR_SUSPEND, 0 );
    }
    UserInput = GetTerminalInput( );
    if ( UserInput == USR_RESTART ) {
        qCR_ExternControl( xHandleCR, qCR_RESTART, 0 );
    }
    Perform_AnotherTask_Activities();
}
/*==================================================================*/
void CoroutineTask_Callback( qEvent_t e ) {
    qCR_BeginWithHandle( xHandleCR ){ /*externally controlled*/
        if ( EventNotComing() ) {
            qCR_Yield;
        }
        RunFirstJob();
        qCR_Delay( WAIT_TIME );
        SecondJobStatus = RunSecondJob();
        qCR_TimedWaitUntil( JobFlag == JOB_SUCCESS, JOB_TIMEOUT );
        CleanUpStatus = CleanupJob();
        qCR_WaitUntil( SomeVar > SomeValue );
    } qCR_End;
}
```

As seen in the code snippet above, the Co-routine handle its globally declared to allow other contexts to access it. The example shows that another task can control the Co-routine using the `qCR_ExternControl` API. The actions performed by this API can be only be effective after the handle instantiation, an operation that takes place once on the first call of the Co-routine.

```
qBool_t qCR_ExternControl( qCR_Handle_t h, const qCR_ExternAction_t action,
                           const qCR_ExtPosition_t action )
```

Parameters

- `h` : The Co-routine handle.
- `action` : The specific action to perform, should be one of the following:
  - `qCR_RESTART` : Restart the Co-routine execution at the place of the `qCR_BeginWithHandle` statement.
  - `qCR_SUSPEND` : Suspend the entire Co-routine segment. The task will still running instructions outside the segment.
  - `qCR_RESUME` : Resume the entire Co-routine segment at the point where it had been left before the suspension.
  - `qCR_POSITIONSET` : Force the coroutine execution at the position specified in `pos`. If a non-valid position is supplied, the Co-routine segment will be suspended.
- `pos` : The requested position if action = `qCR_POSITIONSET`. For other actions this argument its ignored.

☞A `NULL` initialization its mandatory on `qCR_Handle_t` variables. Undefined behavior may occur if this step is ignored.

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 seen as a typed set of commands to produce a result, but here, the commands are typed in real-time by a user through a specific interface, for example, UART, USB, LAN, etc.

A CLI is often developed to aid initial driver development and debugging. This CLI might 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 components of the CLI

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

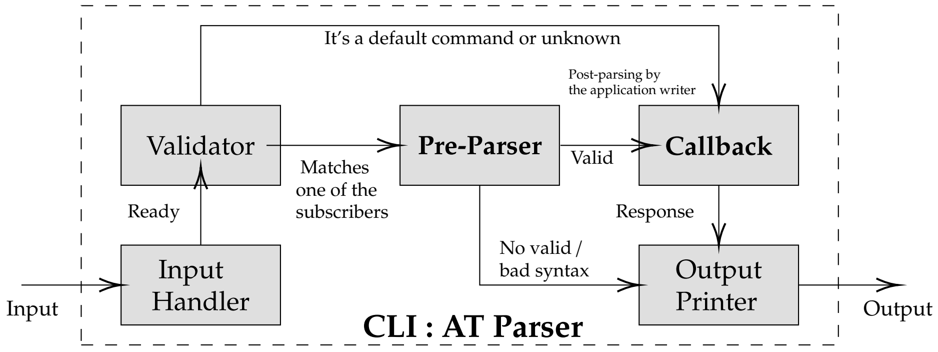

Figure 21: AT parser for a CLI implementation

As seen in figure 21, the CLI has a few components described below:

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

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

3.4.2 Supported syntax

The syntax is straightforward and the rules are provided below:

- All command lines must start with `AT` and end with an `EOL` character. By default, the CLI 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** (`qATCLI_CMDTYPE_ACT`) : This is the simplest type of commands that can be subscribed. Its normally used to execute the action that the command should do. This type doesn't take arguments or modifiers, for example,

    ```
    AT+CMD
    ```

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

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

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

    ```
    AT+CMD=?
    ```

  - **Parameter Set** (`qATCLI_CMDTYPE_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 output responses are:
  - `OK`: Indicates the successful execution of the command.
  - `ERROR`: A generalized message to indicate failure in executing the command.
  - `UNKNOWN` : The input command its not subscribed.
  - `NOT ALLOWED` : The command syntax is not one of the allowed types.
  - `User-defined`: A custom output message defined by the application writer.
  - `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-CLI instance

Before starting the CLI development, the corresponding instance must be defined; a data structure of type `qATCLI_t` . The instance should be initialized using the `qATCLI_Setup()` API . A detailed description of this function is shown bellow:

```
qBool_t qATCLI_Setup( qATCLI_t * const cli, const qPutChar_t OutputFcn,
                      char *Input, const size_t SizeInput,
                      char *Output, const size_t SizeOutput )
```

Parameters

- `cli` : A pointer to the AT Command Line Interface instance.
- `OutputFcn` : The basic output-char wrapper function. All the CLI 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`.

3.4.4 Subscribing commands to the parser

The AT CLI is able to subscribe any number of custom AT commands. For this, the `qATCLI_CmdSubscribe()` API should be used.

This function subscribes the CLI instance to a specific command with an associated callback function, so that next time the required command is sent to the CLI input, the callback function will be executed. The CLI module only analyzes commands that follow the simplified AT-Commands syntax already described in section 3.4.2.

```
qBool_t qATCLI_CmdSubscribe( qATCLI_t * const cli,
                             qATCLI_Command_t * const Command,
                             const char *TextCommand,
                             qATCLI_CommandCallback_t Callback,
                             qATCLI_Options_t CmdOpt, void *param )
```

Parameters

- `cli` : A pointer to the AT Command Line Interface 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: `qATCLI_Response_t xCallback(qATCLI_Handler_t, qATCLI_PreCmd_t )`
- `CmdOpt` : This flag combines with a bitwise 'OR' ('|') the following information:
  - `qATCLI_CMDTYPE_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' : `qATCLI_CMDTYPE_PARA | 0xXY` , where `X` which defines maximum argument number for incoming command and `Y` which defines minimum argument number for incoming command.
  - `qATCLI_CMDTYPE_TEST` : `AT+cmd=?` is allowed.
  - `qATCLI_CMDTYPE_READ` : `AT+cmd?` is allowed.
  - `qATCLI_CMDTYPE_ACT` : `AT+cmd` is allowed.
- `param` : User storage pointer.

3.4.5 Writing a command callback

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

```
qATCLI_Response_t CMD_Callback( qATCLI_Handler_t h ){
    /* TODO : The command callback */
}
```

The callback takes one argument of type `qATCLI_Handler_t` and returns a single value. The

input argument it's just a pointer to public data of the CLI instance where the command it subscribed to. From the callback context, can be used to print out extra information as a command response, parse the command parameters, and query properties with crucial information about the detected command, like the type, the number of arguments, and the subsequent string after the command text. The members are described as follows:

- `Command` (read-only) : A pointer to the calling AT Command object.
- `Type` (read-only) : The command type.
- `StrData` (read-only) : The string data after the command text.
- `StrLen` (read-only) : The length of `StrData`.
- `NumArgs` (read-only) : Number of arguments, only available if `Type = qATCLI_CMDTYPE_PARA`.
- `Output` : Points to the output buffer storage area.
- `UserData` : Points to the user-defined data - Storage Pointer

  This argument also includes helper methods to write the output and retrieve command arguments:

- `void puts(const char* s)` : Writes the string `s` to the output.

- `void putch(const char c)` : Writes the byte `c` to the output.
- `char* GetArgPtr( qIndex_t n )` : Get the pointer where the desired argument starts.
- `int GetArgInt( qIndex_t n )` : Get the `n` argument parsed as integer.
- `qUINT32_t GetArgHex( qIndex_t n )` : Get the `n` HEX argument parsed as `qUINT32_t`.
- `qFloat32_t GetArgFlt( qIndex_t n )` : Get the `n` argument parsed as float.
- `char* GetArgString( qIndex_t n, char *out )` : get the argument `n` by copying the string into `out`.

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

- `qATCLI_OK` : as expected, print out the `OK` string.
- `qATCLI_ERROR` : as expected, print out the `ERROR` string.
- `qATCLI_ERROR_CODE(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 `QATCLI_ERROR_CODE(15)`, will print out the string `ERROR:15`.
- `qATCLI_NORESPONSE` : No response will be printed out.

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

```c
qATCLI_Response_t CMD_Callback( qATCLI_Handler_t h ) {
  qATCLI_Response_t Response = qATCLI_NORESPONSE;
  int arg1 = 0;
  float arg2 = 0;
  /*check the command-type*/
  switch ( h->Type ) {
    case qATCLI_CMDTYPE_PARA:
        if( h->NumArgs > 0 ){
          arg1 = h->GetArgInt( 1 ); /*get the first argument as integer*/
          if( h->NumArgs > 1){
              arg2 = h->GetArgFlt( 2 ); /*get the second argument as float*/
          }
        }
      sprintf( h->Output, "arg1 = %d arg2 = %f", arg1, arg2);
      Response = qATCLI_NORESPONSE;
      break;
    case qATCLI_CMDTYPE_TEST:
      h->puts( "inmediate message" );
      Response = qATCLI_OK;
      break;
    case qATCLI_CMDTYPE_READ:
      strcpy( h->Output , "Test message after the callback");
      Response = qATCLI_OK;
      break;
    case qATCLI_CMDTYPE_ACT:
      Response = qATCLI_OK;
      break;
    default:
      Response = qATCLI_ERROR;
      break;
  }
  return Response;
}
```

3.4.6 Handling the input

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

```
qBool_t qATCLI_ISRHandler( qATCLI_t * const cli, char c )
```

```
qBool_t qATCLI_ISRHandlerBlock( qATCLI_t * const cli, char *data,
                                const size_t n )
```

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

Parameters

- `cli` : Both APIs take a pointer to AT Command Line Interface instance.

for `qATCLI_ISRHandler` :

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

for `qATCLI_ISRHandlerBlock` :

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

Return Value

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

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

```
qBool_t qATCLI_Raise( qATCLI_t * const cli, const char *cmd )
```

```
qATCLI_Response_t qATCLI_Exec( qATCLI_t * const cli, const char *cmd )
```

As expected, both functions send the string to the specified CLI. The difference between both APIs is that `qATCLI_Raise()` sends the command through the input, marking it as ready for parsing and acting as the *Input handler* component.

The `qATCLI_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:

- `cli` : A pointer to the AT Command Line Inteface instance.
- `cmd` : The command string, including arguments if required.

Return value:

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

For `qATCLI_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 `qATCLI_Run()` API. Almost all the components that make up the CLI are performed by this API, except for the *Input Handler*, that should be managed by the application writer itself.

```
qBool_t qATCLI_Run( qATCLI_t * const Parser )
```

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

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

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

```
qBool_t qOS_Add_ATCLITask( qTask_t * const Task,
                           qATCLI_t *cli,
                           qPriority_t Priority, void *arg )
```

Parameters

- `Task` : A pointer to the task node.
- `Parser` : A pointer to the AT Command Line Interface instance.

- `Priority` : Task priority Value. [0(min) - `Q_PRIORITY_LEVELS`(max)]
- `arg` : Storage pointer. To ignore pass `NULL`.

After invoked, both CLI 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 CLI components. As the task is event-triggered, there is no additional overhead and the writer of the application can assign a priority value to balance the application against other tasks in the scheduling scheme.

3.4.8 CLI Example

The following example demonstrates the usage of a simple command-line interface using the UART peripheral with two subscribed commands :

- A command to write and read the state of a GPIO pin `at+gpio`.
- A command to retrieve the compilation timestamp `at+info`.

First, lets get started defining the required objects to setup the CLI module:

```c
#define CLI_MAX_INPUT_BUFF_SIZE         ( 128 )
#define CLI_MAX_OUTPUT_BUFF_SIZE        ( 128 )

qTask_t CLI_Task;
qATCLI_t CLI_Object;
qATCLI_Command_t AT_GPIO, AT_INFO;

char CLI_Input[ AT_CLI_MAX_INPUT_BUFF_SIZE ];
char CLI_Output[ AT_CLI_MAX_OUTPUT_BUFF_SIZE ];

/*Command callbacks*/
qATCLI_Response_t AT_GPIO_Callback( qATCLI_Handler_t h );
qATCLI_Response_t AT_INFO_Callback( qATCLI_Handler_t h );
```

Then the CLI instance its configured by subscribing commands and adding the task to the OS. A wrapper function its required here to make the UART output-function compatible with the CLI API.

```c
void CLI_OutputChar_Wrapper( void *sp, const char c ) { /*CLI output function*/
    (void)sp; /*unused*/
    HAL_UART_WriteChar( UART1, c );
}
/*==================================================================*/
int main( void ) {
    HAL_Setup();
    qOS_Setup( HAL_GetTick, TIMER_TICK, NULL );
    qATCLI_Setup( &CLI_Object, BSP_UART_PUTC, CLI_Input, sizeof(CLI_Input),
                  CLI_Output, sizeof(CLI_Output) );

    qATCLI_CmdSubscribe( &CLI_Object, &AT_GPIO, "at+gpio", AT_GPIO_Callback,
                         QATCLI_CMDTYPE_ACT | QATCLI_CMDTYPE_READ |
                         QATCLI_CMDTYPE_TEST | QATCLI_CMDTYPE_PARA | 0x22, NULL
     );
    qATCLI_CmdSubscribe( &CLI_Object, &AT_GPIO, "at+gpio", AT_GPIO_Callback,
                         QATCLI_CMDTYPE_ACT, NULL );
    qOS_Add_ATCLITask( &CLI_Task, &CLI_Object, qLowest_Priority );
    qOS_Run();

    return 0;
}
```

The CLI input its feeded from the interrupt context by using the UART receive ISR:

```
void interrupt HAL_UART_RxInterrupt( void ) {
    char received;

    received = HAL_HUART_GetChar( UART1 );
    qATCLI_ISRHandler( &CLI_Object, received ); /*Feed the CLI input*/
}
```

Finally, the command-callbacks are later defined to perform the requested operations.

```
qATCLI_Response_t AT_GPIO_Callback( qATCLI_Handler_t h ) {
  qATCLI_Response_t RetValue = qATCLI_ERROR;
    int pin, value;

  switch ( h->Type ) {
      case qATCLI_CMDTYPE_ACT: /*< AT+gpio */
        RetValue = qATCLI_OK;
        break;
        case qATCLI_CMDTYPE_TEST: /*< AT+gpio=? */
            h->puts( "+gpio=<pin>,<value>\r\n" );
            h->puts( "+gpio?\r\n" );
        RetValue = qATCLI_NORESPONSE;
        break;
      case qATCLI_CMDTYPE_READ: /*< AT+gpio? */
          sprintf( h->Output, "0x%08X", HAL_GPIO_Read( GPIOA ) );
        RetValue = qATCLI_NORESPONSE;
        break;
        case qATCLI_CMDTYPE_PARA: /*< AT+gpio=<pin>,<value> */
          pin = h->GetArgInt( 1 );
          value = h->GetArgInt( 2 );
          HAL_GPIO_WRITE( GPIOA, pin, value );
        RetValue = qATCLI_OK;
        break;
      default : break;
    }

  return RetValue;
}
/*==================================================================*/
qATCLI_Response_t AT_INFO_Callback( qATCLI_Handler_t h ) {
  qATCLI_Response_t RetValue = qATCLI_ERROR;

  switch ( param->Type ) {
      case qATCLI_CMDTYPE_ACT: /*< AT+info */
        strcpy( h->Output, "Compilation: " __DATE__ " " __TIME__ );
        RetValue = qATCLI_NORESPONSE;
        break;
      default : break;
  }

  return RetValue;
}
/*=================================================================*/
```

3.5 Memory Management

As the OS is targered to build safe-critical embedded applications, dynamic memory allocation its not allowed for the kernel design, because can lead to out-of-storage run-time failures, which are undesirable. However, some applications can be easily deployed using this allocation scheme, so a safe and portable implementation becomes relevant in the scope of the user-code.

In a typical C environment, memory can be allocated using the standard library functions *malloc()* and *free()*, 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. Also, there is a range of unspecified and implementation-defined behaviour associated with dynamic memory allocation, as well as a number of other potential pitfalls. Additionally, some implementations can suffer from fragmentation.

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

3.5.1 Principle of operation

The allocation scheme works by subdividing a static array into smaller blocks and using the *First-Fit* approach (see figure 22).

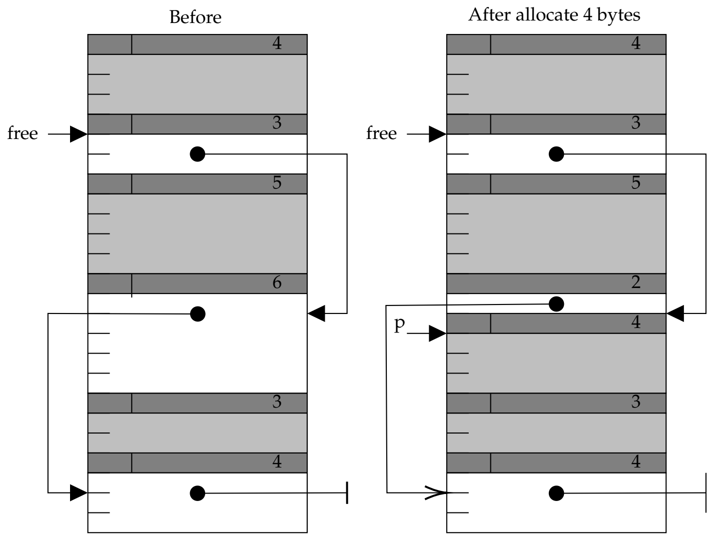

Figure 22: First-fit allocation policy

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

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

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

✱*Warning*: The application is not exempt from memory leaks if the user does not perform adequate memory management. Here, the worst case scenario can occur in the absence of free memory.

3.5.2 Memory pools

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

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

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

```
qBool_t qMemMang_Pool_Setup( qMemMang_Pool_t * const mPool , void* Area ,
                             size_t size )
```

Parameters

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

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

```
qBool_t qMemMang_Pool_Select( qMemMang_Pool_t * const mPool )
```

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

```
size_t qMemMang_Get_FreeSize( void )
```

Usage example:

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

qTask_t taskA;
qMemMang_Pool_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) );
    qMemMang_Pool_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 {
        qTrace_Message("ERROR:ALLOCATION_FAIL");
    }

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

int main(void) {
    char area_another_heap[ 512 ] = { 0 };
    qTrace_Set_OutputFcn( OutPutChar );
    /*Create a memory heap*/
    qMemMang_Pool_Setup( &another_heap, area_another_heap, 512);
    qOS_Setup( HAL_GetTick, 0.001f, IdleTaskCallback );
    qOS_Add_Task( &taskA, taskA_Callback, qLowest_Priority, 0.1f,
                  qPeriodic, qEnabled, NULL);
    qOS_Run();
    return 0;
}
```

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, a single-char output function must be defined using the `qTrace_Set_OutputFcn()` macro.

```
qTrace_Set_OutputFcn( qPutChar_t fcn )
```

Where `fcn` is 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 using the
    selected peripheral.
    */
}
```

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

3.6.1 Viewing variables

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

```
qTrace_Var( Var, DISP_TYPE_MODE )
qTrace_Variable( Var, DISP_TYPE_MODE )
```

```
qDebug_Var( Var, DISP_TYPE_MODE )
qDebug_Variable( Var, DISP_TYPE_MODE )
```

Parameters:

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

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

3.6.2 Viewing a memory block

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

```
qTrace_Mem( Pointer, BlockSize )
qTrace_Memory( Pointer, BlockSize )
```

Parameters:

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

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

3.6.3 Usage

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

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

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

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

After that, trace macros will be available for use.

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

4 Utility APIs

4.1 Generic lists

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

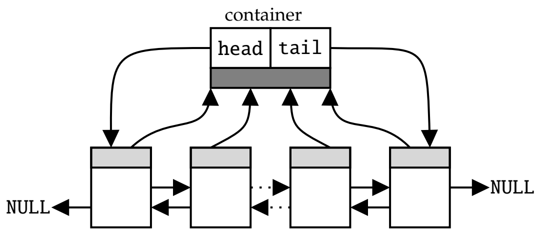

Figure 23: Doubly-linked list implementation

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

```
typedef struct{
    qNode_MinimalFields; /*< required for lists*/
    int a;
    int b;
    float y;
}userdata_t;
```

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

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

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

Parameters:

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

Return value:

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

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

Insert an item into the list.

Parameters:

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

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

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

Return value:

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

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

Remove an item from the list.

Parameters:

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

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

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

Return value:

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

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

Check if the node is member of the list.

Parameters:

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

Return value:

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

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

Get a pointer to the front item of the list.

Parameters:

- `list` : Pointer to the list.

Return value:

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

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

Get a pointer to the back item of the list.

Parameters:

- `list` : Pointer to the list.

Return value:

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

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

Check if the list is empty.

Parameters:

- `list` : Pointer to the list.

Return value:

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

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

Get the number of items inside the list.

Parameters:

- `list` : Pointer to the list.

Return value:

The number of items of the list.

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

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

Parameters:

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

Return value:

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

```
qBool_t qList_ForEach( qList_t *const list, qList_NodeFcn_t Fcn,
                       void *arg, qList_Direction_t dir, void *NodeOffset )
```

Operate on each element of the list.

Parameters:

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

  Should have this prototype:

  ```
  qBool_t Function( qList_ForEachHandle_t h )
  ```

  where the `h` its a handle of the list iterator with the following members:

  - `node` : the k-element of the list that is currently being processed in the loop.
  - `arg` : same argument of the calling function.
  - `stage` : indicates the loop progress and should be checked by the application writer to perform the specific operations over the list. This variable can take the following values:
    * `QLIST_WALKINIT` : When the loop is about to start. In this case, A `NULL` value will be passed inside the `node` member.
    * `QLIST_WALKTHROUGH` : When the loop is traversing the list.
    * `QLIST_WALKEND` : When the loop has finished. In this case, A `NULL` value will be passed in the `node` member.

  By default, `Function` should return `qFalse`. If a `qTrue` value is returned, the walk through loop will be terminated.
- `arg` : Argument passed to `Fcn` through the iterator handle.
- `dir` : Use one of the following options:
  - `QLIST_FORWARD` : to walk through the list forwards.
  - `QLIST_BACKWARD` to walk through the list backwards.
- `NodeOffset` : If available, the list walk through will start from this node. To ignore, pass `NULL`.

Return value:

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

```c
qBool_t qList_Sort( qList_t * const list,
                    qBool_t (*CompareFcn)( qList_CompareHandle_t h ) )
```

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

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

Parameters:

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

  Taking `h` as the conpare-handle argument with two members inside, `n1` and `n2`, pointing to the nodes being compared (both converted to (`const void*`). The function defines the order of the elements by returning a Boolean data, where a `qTrue` value indicates that element pointed by `n1` goes after the element pointed to by `n2`

Return value:

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

```
qBool_t qList_IteratorSet( qList_Iterator_t *iterator, qList_t *const list,
                           void *NodeOffset, qList_Direction_t dir )
```

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

Parameters:

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

Return value:

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

```
void* qList_IteratorGetNext( qList_Iterator_t *iterator )
```

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

Parameters:

- `iterator` : Pointer to the iterator instance.

Return value:

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

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

Swap two nodes that belongs to the same list by changing its own links.

Note: The container list will be updated if any node is part of the boundaries.

Parameters:

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

Return value:

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

4.2 Byte sized buffers

An interrupt-safe byte-sized ring buffer.

```
qBool_t qBSBuffer_Setup( qBSBuffer_t * const obj, volatile qUINT8_t *buffer,
                     const size_t length )
```

Initialize the byte-sized buffer.

Parameters:

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

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

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

Parameters:

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

Return value:

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

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

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

Parameters:

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

Return value:

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

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

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

Parameters:

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

Return value:

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

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

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

Parameters:

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

Return value:

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

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

Query the empty status of the byte-sized buffer.

Parameters:

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

Return value:

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

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

Query the full status of the byte-sized buffer.

Parameters:

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

Return value:

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

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

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

Parameters:

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

Return value:

Number of elements in the byte-sized buffer.

4.3 Input groups for edge-checking

An interface to manage and simplify the value(with edge-checking) of incoming digital signals groups.

```
qBool_t qEdgeCheck_Setup( qEdgeCheck_t * const Instance,
                          const qCoreRegSize_t RegisterSize,
                          const qClock_t DebounceTime )
```

Initialize an I/O edge-check instance.

Parameters:

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

Return value:

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

```
qBool_t qEdgeCheck_Add_Node( qEdgeCheck_t * const Instance,
                             qEdgeCheck_IONode_t * const Node,
                             void *PortAddress,
                             const qBool_t PinNumber )
```

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

Parameters:

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

Return value:

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

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

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

Parameters:

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

Return value:

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

```
qBool_t qEdgeCheck_Get_NodeStatus( const qEdgeCheck_IONode_t * const Node )
```

Query the status of the specified input-node.

Parameters:

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

Return value:

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

4.4 Response handler

An API to simplify the handling of requested responses from terminal interfaces.

```
qBool_t qResponse_Setup( qResponse_t * const obj, char *xLocBuff,
                         size_t nMax )
```

Initialize the instance of the response handler object.

Parameters:

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

Return value:

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

```
void qResponse_Reset( qResponse_t * const obj )
```

Reset the response handler.

Parameters:

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

```
qBool_t qResponse_Received( qResponse_t * const obj,
                            const char *Pattern, size_t n )
```

Non-blocking response check.

Parameters:

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

Return value:

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

```
qBool_t qResponse_ReceivedWithTimeout( qResponse_t * const obj,
                                       const char *Pattern,
                                       size_t n, qTime_t t )
```

Non-blocking response check with timeout.

Parameters:

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

Return value:

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

```
qBool_t qResponse_ISRHandler( qResponse_t * const obj,
                              const char rxchar )
```

ISR receiver for the response handler. . This call should be placed inside the ISR in charge of receiving the incoming data (one byte at the time).

Parameters:

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

Return value:

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

4.5 Miscellaneous

```
qTime_t qClock_Convert2Time( 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 qClock_Convert2Clock( 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.

```
qBool_t qIOUtil_SwapBytes( 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 qIOUtil_CheckEndianness( void )
```

Check the system endianess.

Return value:

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

```
qBool_t qIOUtil_OutputRaw( qPutChar_t fcn, void* storagep, void *data,
                           size_t n, qBool_t AIP )
qBool_t qIOUtil_InputRaw( qGetChar_t fcn, void* storagep, void *data,
                          size_t n, qBool_t AIP )
```

Wrapper methods to write(`qIOUtil_OutputRaw`) or read(`qIOUtil_InputRaw`) `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.

```
qBool_t qIOUtil_OutputString( 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.

```
char* qIOUtil_U32toX( qUINT32_t value, char *str, int8_t n )
```

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

Parameters:

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

Return value:

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

```
qUINT32_t qIOUtil_XtoU32( const char *s )
```

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

Parameters:

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

Return value:

The numeric value in `qUINT32_t`.

```
qFloat64_t qIOUtil_AtoF( 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 qIOUtil_AtoI( 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* qIOUtil_UtoA( qUINT32_t num, char* str, qUINT8_t base )
```

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

Parameters:

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

Return value:

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

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

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

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

Parameters:

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

Return value:

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

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

Converts a float value to a formatted string.

Parameters:

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

Return value:

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

```
char* qIOUtil_BtoA( 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`.

4.6 Additional macros

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

References

[1] Kyle E Mitchell. The mit license line by line - /dev/lawyer. `https://writing.kemitchell.com/2016/09/21/MIT-License-Line-by-Line.html`. Accessed: 2021-04-13.

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

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

[4] H. Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Object Technology Series. Addison Wesley, 2000.

[5] Donald Knuth. *The Art of Computer Programming, Volume 1*. Addison Wesley, 1998.

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

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

Index