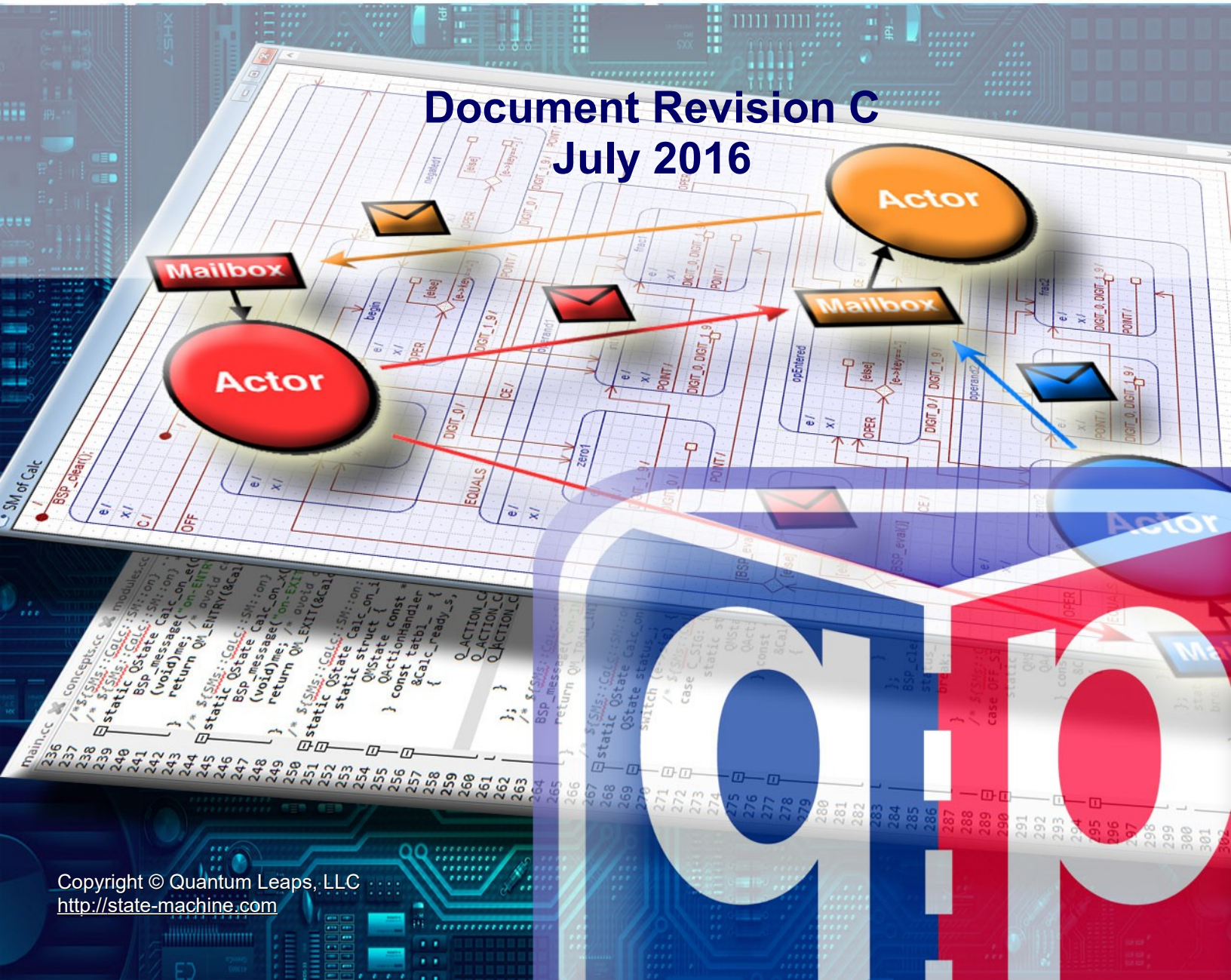
Quantum® LeaPs
innovating embedded systems

Application Note:
QP™/C++ Performance Tests and Results

Document Revision C
July 2016



Copyright © Quantum Leaps, LLC
http://state-machine.com

Table of Contents

1 Introduction 1
- 1.1 General Performance Measurement Strategy 2
- 1.2 CPU-Assisted Cycle Measurement 2
- 1.3 Aspects of Performance Measured 3

2 Test Environment 4
- 2.1 Target Hardware 4
- 2.2 Embedded Toolset 5
  - 2.2.1 Optimization Options 5
  - 2.2.2 FPU Options 6
- 2.3 Embedded Software 7
  - 2.3.1 Software Components 8
  - 2.3.2 Hierarchical State Machines 9
  - 2.3.3 Assertions in QP/C++ 10
  - 2.3.4 No Argument Checking in uC/OS-II 10
  - 2.3.5 Compile-Time Configuration of uC/OS-II 10

3 CPU-Cycle Measurements and Results 11
- 3.1 Group 1 Performance Tests 11
  - 3.1.1 Interrupt Latency, Response, and Recovery 11
  - 3.1.2 Message (Event) Posting 13
  - 3.1.3 Memory Pool Get and Put 13
  - 3.1.4 Group 1 Performance Measurement Results 14
- 3.2 Group 2 Performance Tests 15
  - 3.2.1 Semaphores 15
  - 3.2.2 Mutexes 16
  - 3.2.3 In-line delay() 16
  - 3.2.4 Group 2 Performance Measurement Results 17
- 3.3 Group 3 Performance Tests 18
  - 3.3.1 Hierarchical State Machines 18
  - 3.3.2 Dynamic Events 19
  - 3.3.3 Event Posting 19
  - 3.3.4 Event Publishing 19
  - 3.3.5 Time Events 19
  - 3.3.6 Group 3 Performance Measurement Results 20

4 Memory Size Measurements 21
- 4.1 Code Size Measurements 21
  - 4.1.1 Built-in Kernels (QV, QK, and QXK) 21
  - 4.1.2 Conventional RTOS (uC/OS-II) 23
- 4.2 Data Size Measurements 24
- 4.3 Stack Size Measurements 24
- 4.4 Memory Size Measurement Results 25

5 Summary 26

6 Contact Information 26

Copyright © Quantum Leaps, LLC. All Rights Reserved.

Legal Disclaimers

Information in this document is believed to be accurate and reliable. However, Quantum Leaps does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

Quantum Leaps reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

All designated trademarks are the property of their respective owners.

1 Introduction

This Application Note describes performance tests and results of the QP/C++ active object framework, which combines hierarchical state machines with the particular model of concurrency, known as active objects (a.k.a., actors). The framework provides a modern, event-driven, *reusable architecture* that is generally *safer*, more responsive and easier to understand than "naked threads" of a conventional Real-Time Operating System (RTOS).

In the resource-constrained embedded systems, the biggest concern has always been about the size and efficiency of any such "unconventional" solutions, especially that most existing active object frameworks (e.g., frameworks accompanying various modeling tools) have traditionally been built on top of a conventional RTOS, which can only add memory footprint and CPU overhead to the end-application.

However, in this respect QP/C++ differs from most other active object frameworks, because it can run standalone on bare-metal single-chip microcontrollers, completely replacing a conventional RTOS. The framework contains a selection of *built-in* real-time kernels, such as the cooperative QV kernel, the preemptive, non-blocking QK kernel, and the preemptive, blocking QXK kernel, which are all specifically custom-tailored and optimized to execute event-driven active objects. For compatibility with the existing software, the QP/C++ framework can also run on top of a conventional, third-party RTOS kernel, such as the µC/OS-II RTOS kernel from Micrium used in this study. All this creates a unique opportunity to compare the space (code) and time (CPU cycles) performance of these various options.

The main goals of this Application Note are as follows:

- To help you understand *what* performance aspects are being measured and *how* they are being measured, so that you can make meaningful apples-to-apples comparisons
- To help you evaluate the QP/C++ framework against other competitive offerings and your specific performance goals by presenting the specific test *results*
- To help you understand the overhead of various features and configuration options of the QP/C++ framework to choose the right options for your specific application
- To provide you with methods and tools to *repeat* the tests yourself, or to devise your *own* performance tests
- To help you debug timing and performance bottleneck problems in your code
- To help you perform CPU utilization of your own application to apply formal methods such as Rate Monotonic Analysis (RMA)

Copyright © Quantum Leaps, LLC. All Rights Reserved.

1.1 General Performance Measurement Strategy

The general problem with any performance measurements is that they strongly depend on a large number of factors, such as: the CPU type, the CPU memory interface (e.g., wait-states), the compiler quality, the specific compiler optimization, other compiler options, the specific initial state of the system under test (e.g., the priority of a thread to be preempted), the specific final state reached after the test (e.g., context switch was required or not), any instrumentation used to perform the test, and many, many more. Therefore, it is generally impossible to cover the whole test space and provide meaningful measurements for even the most common use scenarios that would be directly applicable to your specific system.

Instead, the general philosophy applied in this document is to carefully explain *what* is being measured and *how* it is being measured, so that **you** can repeat the tests yourself, with your specific hardware and software configuration.

Therefore, the general measurement strategy used in this Application Note is to keep it *simple* (so that you have a chance to actually repeat these tests). To this end, the measurements described in this documents have the following characteristics:

- **Non-intrusive** means that the performance measurements require no additional code on the target system. This is certainly a preferred method because intrusive methods always have some effect on the timing of the target system.
- **Non-repetitive** means that you don't need to construct special test harnesses to repeat the tests in a loop to improve the accuracy. You simply choose the interesting case in your code and perform a cycle-count measurement for this one case.

1.2 CPU-Assisted Cycle Measurement

In the past, performance testing often required instrumenting the target code (e.g., to toggle GPIO pins that could then be observed on an oscilloscope) and repeating the tests many times to improve the accuracy of the collected measurements. Consequently, most older benchmarks were designed to execute certain pieces of code repetitively millions of times.

However, modern MCUs often have a register that keeps track of the elapsed CPU cycles. For example, newer ARM Cortex-M3/4/7 devices provide a register called `CYCLECOUNTER`. Recording the values of the `CYCLECOUNTER` at two code points allows you can accurately measure the number of CPU cycles as the *difference* between those two points. With such measurement, you can always calculate the actual execution *time* for your specific CPU clock speed. For example, if your CPU is running at 100MHz, you can divide the number of CPU cycles by 100 to get the number of microseconds passed.

> **NOTE:** All timing measurements in this document are expressed in *CPU cycles* rather then seconds or microseconds.

The benefits of such CPU-assisted cycle measurements are:

- No need to modify the target code
- The accuracy is one CPU cycle unit (see NOTE below)
- No need to repeat the tests multiple times to increase accuracy
- The method requires support from the MCU and the debugger you are using

> **NOTE:** The `CYCLECOUNTER` is easy to use and provides precise measurements, but to achieve the single CPU cycle accuracy you need to be careful to always use the **Step-Over** or **Continue** to the next breakpoint debugger command (as opposed to **Step-Into**). Also, the value of `CYCLECOUNTER` can be influenced by the CPU pipeline or cache conditions. It is recommended not to measure only one or two instructions, but instead ten or more instructions to get more precise data.

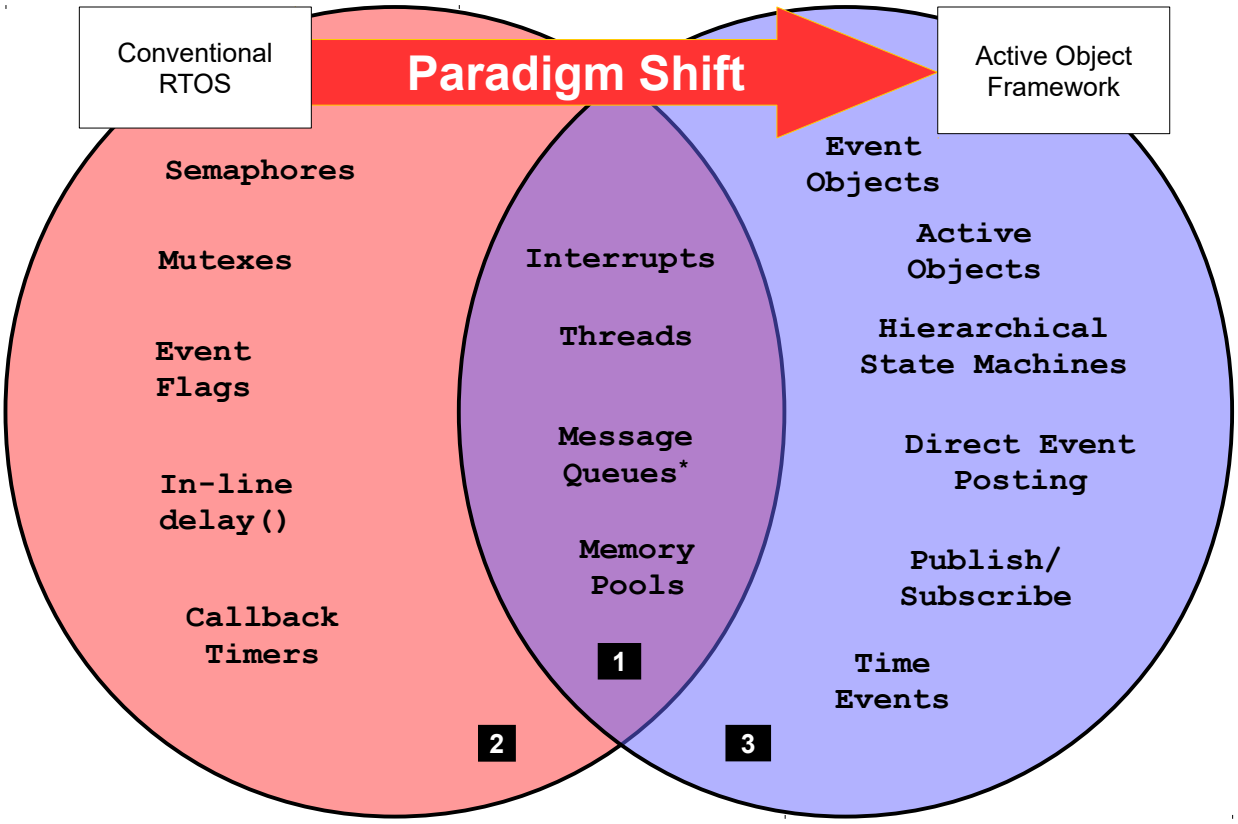
Copyright © Quantum Leaps, LLC. All Rights Reserved.

1.3 Aspects of Performance Measured

The active object model of execution underlying QP/C++ represents a *paradigm shift* compared to a conventional RTOS (see Figure 1). This means that many features of QP/C++, such as hierarchical state machines, are not provided in conventional RTOS, and conversely, many features provided in a conventional RTOS, such as semaphores, are not provided in QP/C++. For these reasons, the comparative performance measurements are not possible for all QP/C++ features and some performance measurements cannot be compared to any existing RTOS offerings.

NOTE: The exception is the QXK built-in kernel of QP/C++, which does provide blocking mechanisms such as semaphores, mutextes, and in-line delay.

Figure 1: Paradigm shift from a conventional RTOS to an active object framework



According to Figure 1, the performance measurements are divided into three groups with different participants:

1. Features supported by both a conventional RTOS (µC/OS-II) and the built-in QP/C++ kernels: QV, QK and QXK (e.g., interrupt processing, context switch, queues, memory partitions).
2. Features supported only by a conventional RTOS (µC/OS-II) and the QXK built-in kernel (semaphores, in-line delay, etc.)
3. Features supported only by the QP/C++ framework and not available in conventional RTOS (publish/subscribe, time events, hierarchical state machines, etc.)

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2 Test Environment

To focus the discussion, this Application Note uses a specific selection of the target hardware, embedded toolset, and the embedded software to test. This section describes all these components and their configuration used in the performance tests.

NOTE: The selected hardware and software should be generally representative for the intended QP/C++ applications, which are low-end 32-bit single-chip microcontrollers.

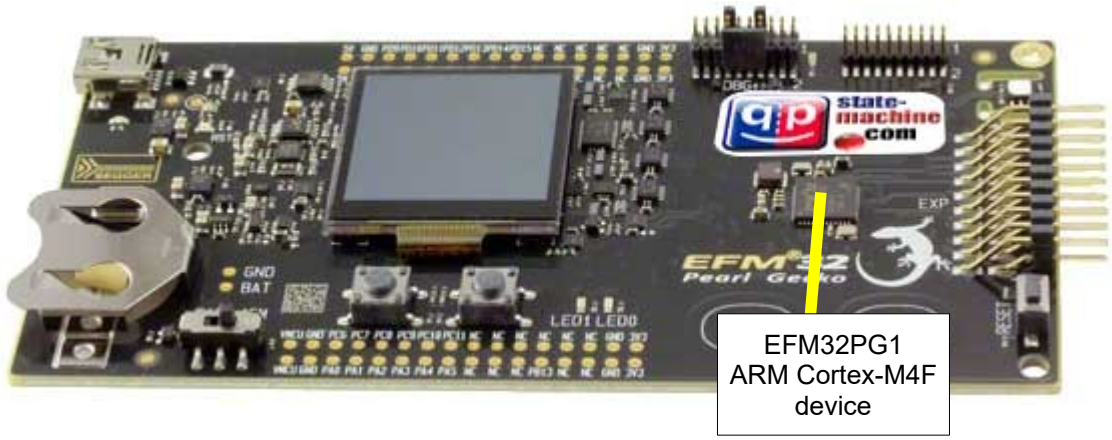
2.1 Target Hardware

The target hardware used in this Application Note is the EFM32-SLSTK3401A Pearl Gecko Starter Kit from Silicon Labs. The board is based on the EFM32PG1B200F256GM48 MCU with **ARM Cortex-M4F** core surrounded by 256KB of no-waitstate Flash ROM and 32KB of SRAM. The CPU is capable of clock speeds up to 40MHz, but is clocked only at 19MHz in the tests.

The CPU provides the `CYCLECOUNTER` register for measuring execution times. The board also provides the built-in J-Link debugger, two user LEDs (LED0 and LED1) and two user switches (BTN0 and BTN1) that are used in the tests. The board is inexpensive ($29.99 at the time of this writing) and is available directly from Silicon Labs as well as many online distributors.

NOTE: The FPU inside the ARM Cortex-M4 core is NOT used in the performance tests (see also the Toolset configuration), so the CPU behaves very similarly to the ARM Cortex-M3 core.

Figure 2: EFM32-SLSTK3401A (Perl Gecko Starter Kit) board



Copyright © Quantum Leaps, LLC. All Rights Reserved.

2.2 Embedded Toolset

The embedded toolset used in this Application Note is the IAR EWARM version **7.60** running under the free size-limited KickStart license. The C-SPY debugger included in this toolset supports viewing the `CYCLECOUNTER` register for measuring execution times (see Figure 3).

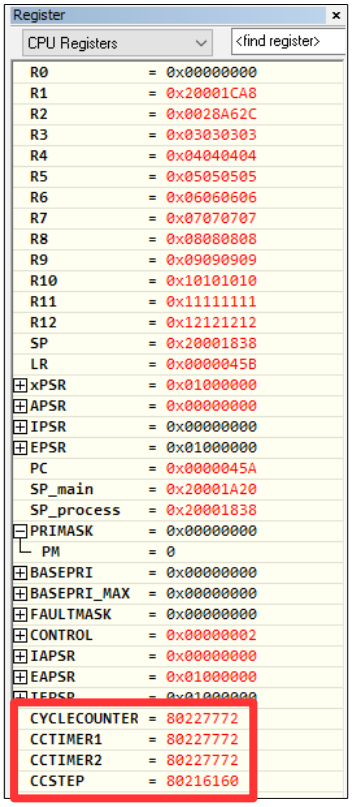
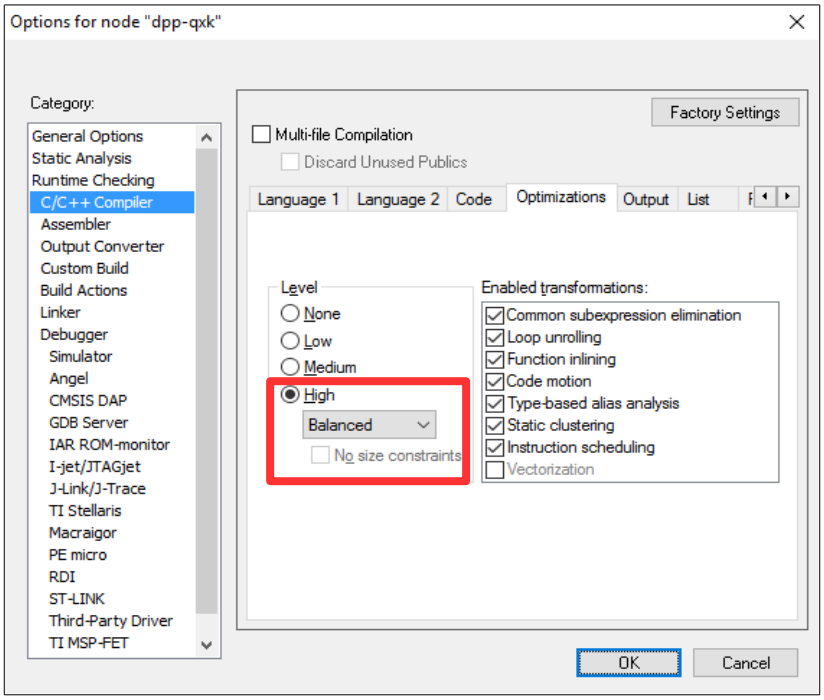


Figure 3: The CYCLECOUNTER register in the IAR C-SPY debugger view

2.2.1 *Optimization Options*

The choice of compiler optimization has significant impact on the performance measurements. Therefore it has to be specified and kept the same in all tests. The optimization chosen for all performance measurements is **High/Balanced** (see Figure 4).

Figure 4: IAR EWARM optimization options (High/Balanced) used in all performance tests



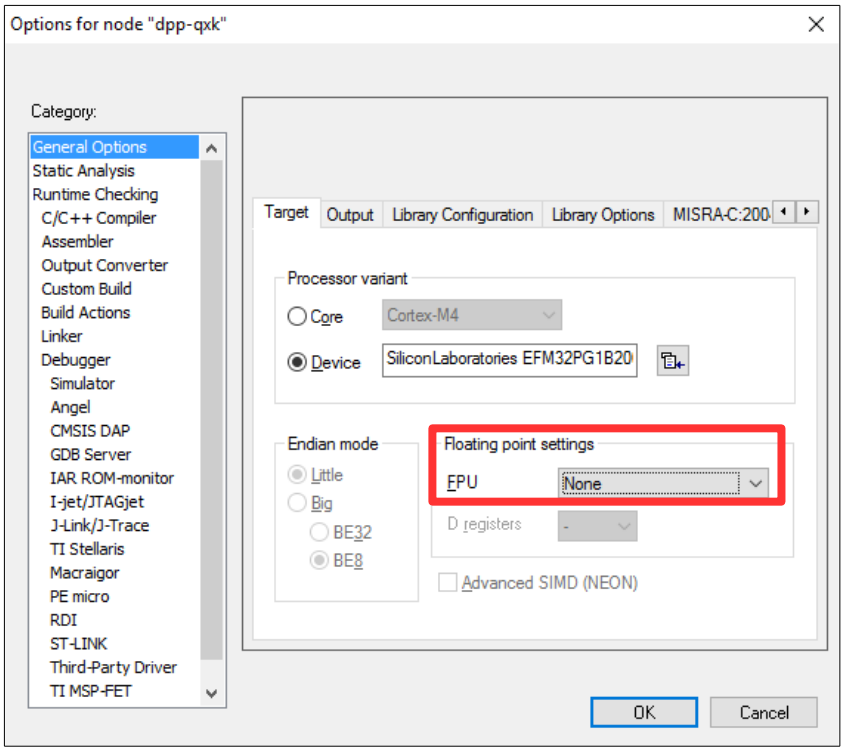
Copyright © Quantum Leaps, LLC. All Rights Reserved.

2.2.2 *FPU Options*

The presence of the hardware FPU (which is available in ARM Cortex-M4F) can have an impact on the interrupt entry/exit times, context switch times, and the size of stack required. Therefore, to simplify the measurements, the hardware FPU is **not used** in any of the tests.

To disable the FPU, the Floating Point settings are set to "None" (see Figure 5). Additionally, for the µC/OS-II tests, the command-line macro `__TARGET_FPU_SOFTVFP` is defined to the compiler. (The µC/OS-II port to ARM Cortex-M4 internally uses this macro to conditionally omit the code for the FPU.)

Figure 5: IAR EWARM FPU options



NOTE: The FPU is the main difference between ARM Cortex-M3 and M4, so with FPU disabled the Cortex-M4 core behaves very similarly to the Cortex-M3 core.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2.3 Embedded Software

As mentioned before, the performance testing strategy applied in this Application Note does not require creating specific test harnesses. Instead, the simplicity of CPU cycle measurements allows you to use any QP/C++ application (e.g., your own project) for testing performance. Of course, the application must contain at least one instance of every performance aspect you wish to measure.

This Application Note uses a slightly modified Dining Philosophers Problem (DPP) example, which is provided in the examples directory of the QP/C++ distribution (see Listing 1).

Listing 1: Selected directories and files in the performance test code

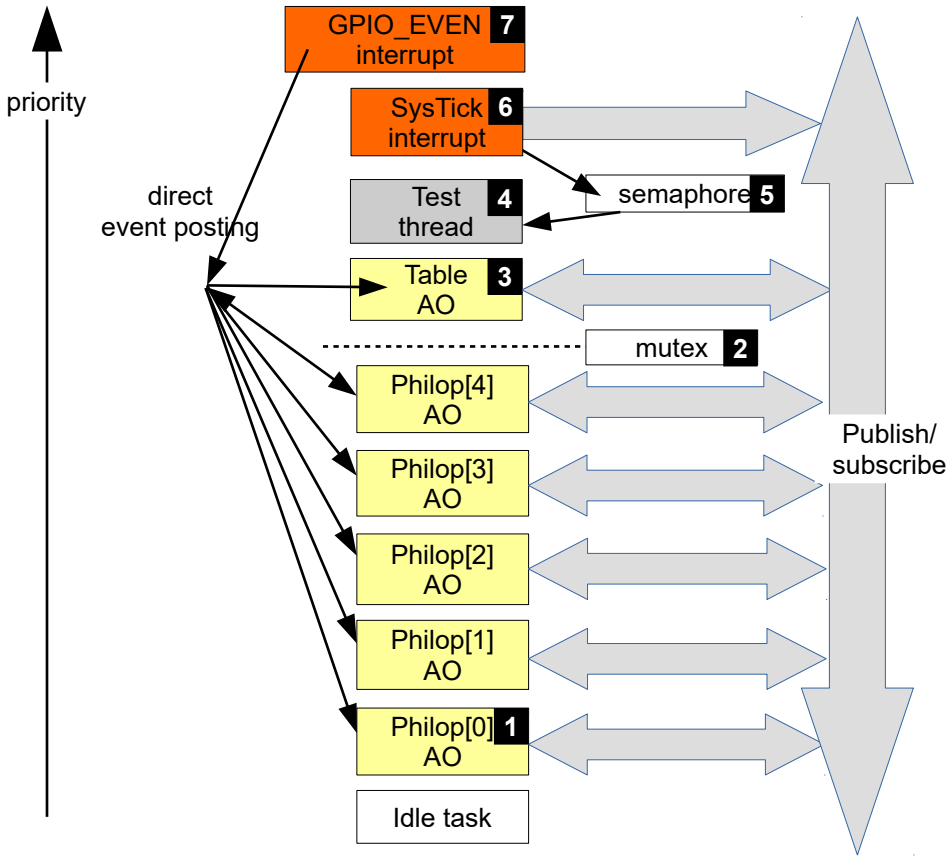
```
<qpcpp>/                - QP/C++ installation directory
+-3rd_party/            - 3rd-party software
| +-efm32pg1b/          - EFM32PG support software
+-examples/
| +-performance/
| | +-dpp_efm32-slstk3401a/ - DPP example for EFM32-SLSTK3401A board
| | | +-qk/             - QK preemptive non-blocking kernel
| | | | +-iar/          - IAR toolset
| | | | | +-dpp-qk.eww  - IAR workspace to build the example
| | | | | | +-. . .
| | | | +-bsp.cpp       - Board Support Package implementation for QK
| | | | +-main.cpp      - main() function for QK
| | | +-qv/             - QV cooperative kernel
| | | | +-iar/          - IAR toolset
| | | | | +-dpp-qv.eww  - IAR workspace to build the example
| | | | | | +-. . .
| | | | +-bsp.cpp       - Board Support Package implementation for QV
| | | | +-main.cpp      - main() function for QV
| | | +-qxk/            - QXK preemptive, blocking kernel
| | | | +-iar/          - IAR toolset
| | | | | +-dpp-qxk.eww - IAR workspace to build the example
| | | | | | +-. . .
| | | | +-bsp.cpp       - Board Support Package implementation for QXK
| | | | +-main.cpp      - main() function for QXK
| | | | +-test.h        - Test of a "naked" thread for QXK interface
| | | | +-test.cpp      - Test of a "naked" thread for QXK implementation
| | | +-ucos-ii/        - uC/OS-II conventional RTOS kernel
| | | | +-iar/          - IAR toolset
| | | | | +-dpp-ucos2.eww - IAR workspace to build the example
| | | | | | +-. . .
| | | | +-bsp.cpp       - Board Support Package implementation for uC/OS-II
| | | | +-main.cpp      - main() function for uC/OS-II
| | | | +-os_cfg.h      - uC/OS-II kernel configuration file
| | | | +-app_cfg.h     - uC/OS-II application configuration file
| | | | +-test.h        - Test of a "naked" thread for uC/OS-II interface
| | | | +-test.cpp      - Test of a "naked" thread for uC/OS-II implementation
| | | |
| | | +-bsp.h           - Board Support Package interface
| | | +-dpp.h           - DPP application interface (generated code)
| | | +-dpp_qhsm.qm/    - DPP application QM model (QHsm strategy)
| | | +-dpp_qmsm.qm/    - DPP application QM model (QMsm strategy)
| | | +-philo.cpp       - Philosopher active object (generated code)
| | | +-table.cpp       - Table active object (generated code)
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2.3.1 Software Components

As shown in Figure 6, the DPP example application consists of 6 active objects (`Philo[0..4]` and the `Table` AO) and a couple of interrupts that publish or send directly events to the active objects. The active objects also communicate with one another by exchanging events both directly and by means of publish/subscribe. Additionally, the examples for QXK and uC/OS-II have an extra "naked" thread Test. This naked thread uses a semaphore and in-line delay() to exercise these features.

Figure 6: Components of the DPP test example



1 5 identical `Philo[0..4]` active objects have priorities 1..5.

2 A mutex at priority 6 to protect the random number generator shared among the `Philo` active objects (this feature is not available in the QV kernel)

3 `Table` active object at priority 7

4 `Test` "naked thread" at priority 8 (available only in the QXK and uC/OS-II kernels). This thread exercises such features as semaphore (see [5] and in-line `delay()`).

5 A semaphore to signal the `Test` "naked thread" (available only in the QXK and uC/OS-II kernels)

6 `SysTick` interrupt handler that runs at 100Hz and posts and publishes events to the active objects

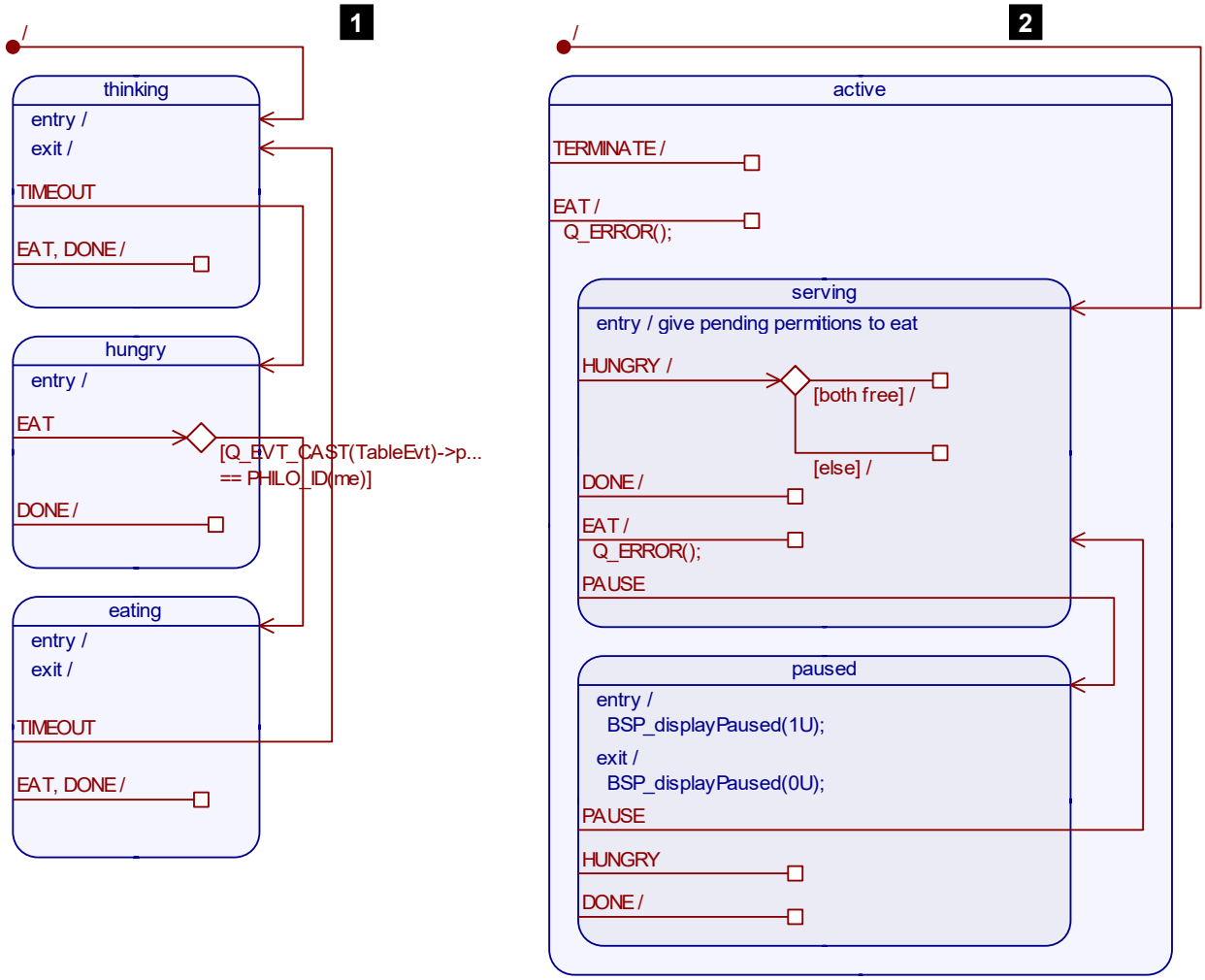
7 `GPIO_EVEN` interrupt handler that posts event directly to the `Table` active object. This interrupt can be triggered from the debugger and is used for testing interrupt handling.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2.3.2 *Hierarchical State Machines*

Figure 7 shows the state machines in the DPP test application.

Figure 7: State Machines in the DPP test example



1 The state machine associated with the Philo active objects is simple without state nesting. It clearly shows the life cycle consisting of states “thinking”, “hungry”, and “eating”. This state machine generates the HUNGRY event on entry to the “hungry” state and the DONE event on exit from the “eating” state because this exactly reflects the semantics of these events. The state machine is driven by time events (TIMEOUT) and EAT signals from the Table active object.

2 The state machine associated with the Table active objects contains one level of state nesting. It has a composite state “active” with two sub-states: “serving” and “paused”. The event PAUSE triggers transitions between these two states. This event is generated upon pressing and releasing the BTN0 switch on the Pearl Gecko board.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2.3.3 Assertions in QP/C++

The QP/C++ framework extensively uses Design-by-Contract (assertions) to check various conditions, such as parameters passed to the framework, consistency of the internal data, etc. These assertions obviously introduce some overhead, but they are considered an integral part of the framework. Therefore they are left *enabled* and the results published here reflect the additional overhead of checking assertions.

2.3.4 No Argument Checking in uC/OS-II

In contrast, the uC/OS-II code is built with the configuration macro `OS_ARG_CHK_EN` set to zero (*disabled*), so the results published here do *not* contain additional overhead of parameters checking in uC/OS-II.

> **NOTE:** uC/OS-II contains many checks that are not under the control of the `OS_ARG_CHK_EN` macro. These integral checks are still active, because removing them would require modifying the uC/OS-II source code.

2.3.5 Compile-Time Configuration of uC/OS-II

The uC/OS-II kernel is designed to be configured by means of pre-processor macros, which are all defined in the `os_cfg.h` and `app_cfg.h` header files (see Listing 1). The settings in these header files are carefully chosen to minimize the code size and overhead of the uC/OS-II kernel for the DPP example. In particular, only services used by the DPP example are enabled, and the number of kernel objects is set such that it just fits the actual use of the project without over-allocating anything.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

3 CPU-Cycle Measurements and Results

This section describes the performed tests and the obtained results. The following sub-sections cover sets of related features. Each sub-section starts with an explanation of *what* is being measured and *how* it is being measured, followed by the measurements.

NOTE: Please remember that the CPU-cycle numbers depend on many factors and might be significantly different for different CPUs, compilers, compiler options, etc.

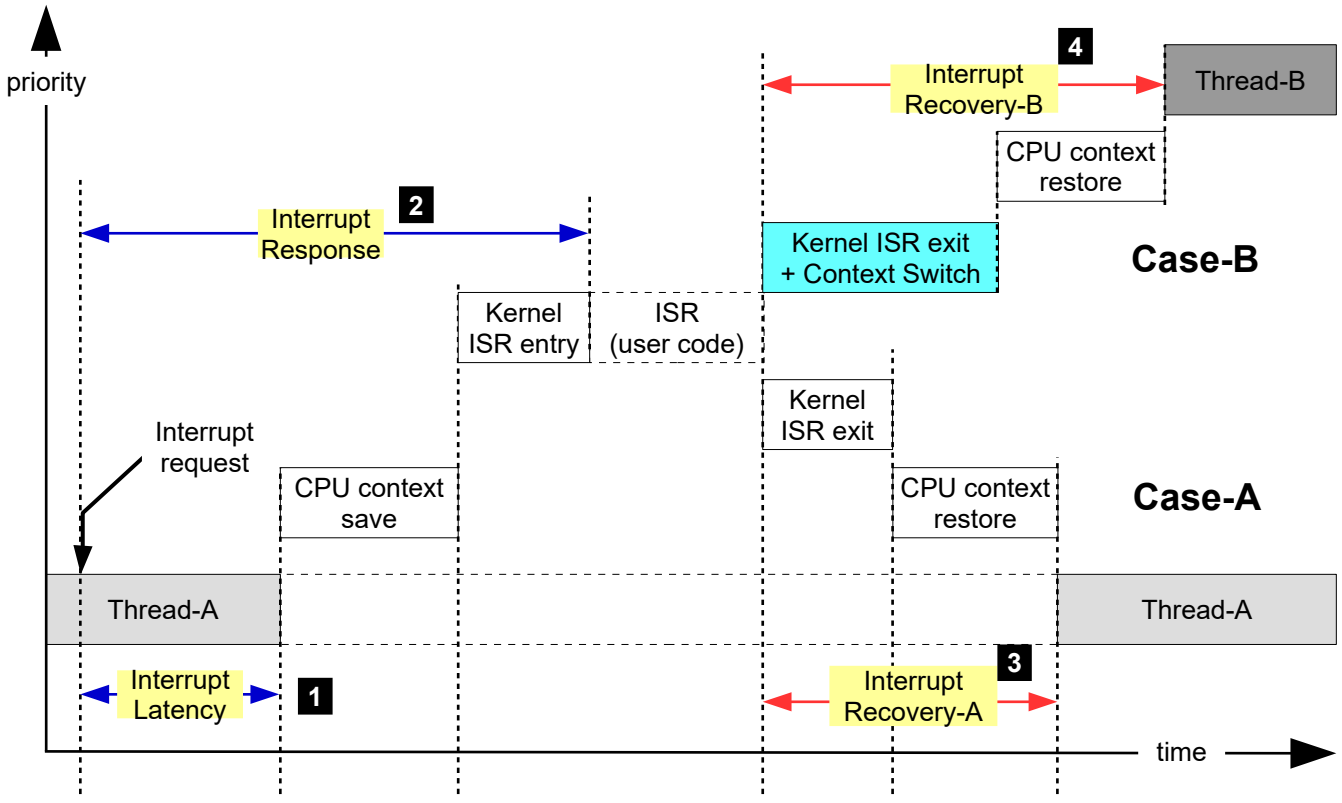
3.1 Group 1 Performance Tests

As described in Section 1.3, Group 1 features are supported by both a conventional RTOS (µC/OS-II) and the built-in QP/C++ kernels: QV, QK and QXK. These features include interrupt processing including context switch, queues, and memory partitions.

3.1.1 Interrupt Latency, Response, and Recovery

Probably the most important performance specification of a real-time kernel is how fast it handles *interrupts*. The following Figure 8 shows the most important aspects of interrupt handling that are measured and reported in this document.

Figure 8: Interrupt latency, response, and recovery



Copyright © Quantum Leaps, LLC. All Rights Reserved.

1 **Interrupt Latency** is the time between the arrival of an interrupt request and the beginning of interrupt processing. This latency is caused by briefly disabling interrupts that most real-time kernels (including uC/OS-II and all the built-in kernels in QP/C++) to perform critical operations atomically. Sections of code that run with interrupts disabled are called *Critical Sections*.

> **NOTE:** For the ARM Corex-M3/M4 CPUs with the BASEPRI register, the built-in kernels in QP/C++ **never disable** the highest-priority interrupts prioritized above the specified threshold, so consequently such "kernel-unaware" interrupts run with **"zero latency"**. The standard uC/OS-II to ARM Cortex-M4 does not support this feature. (See also Application Note: QP and ARM Cortex-M).

In general, measurement of interrupt latency requires finding the longest Critical Section in the system. In case of QP/C++, some of the longest Critical Section is inside the `QMActive::post_()` function (83 CPU cycles). The QK port to ARM Cortex-M has an even longer Critical Section inside the `PendSV` exception handler.

The uC/OS-II code has not been scrutinized sufficiently to determine for sure which Critical Section is the longest. For the purposes of this analysis, the longest critical section in uC/OS-II code has been found in the `OSQPost()` function.

> **NOTE:** The determination which Critical Section in the code is the longest depends on the kernel configuration (enabled features), CPU type, compiler type, compiler optimization, and many other factors. Therefore, the analysis should be repeated for the specific system configuration at hand.

2 **Interrupt Response** is the time between the arrival of an interrupt request and the start of the user code that handles the interrupt. The additional contribution to Interrupt Response comes from the hardware CPU context save (9-12 CPU cycles on Cortex-M3/M4) plus the special function "Kernel ISR Entry" that the real-time kernel must call to notify the kernel that the ISR is starting.

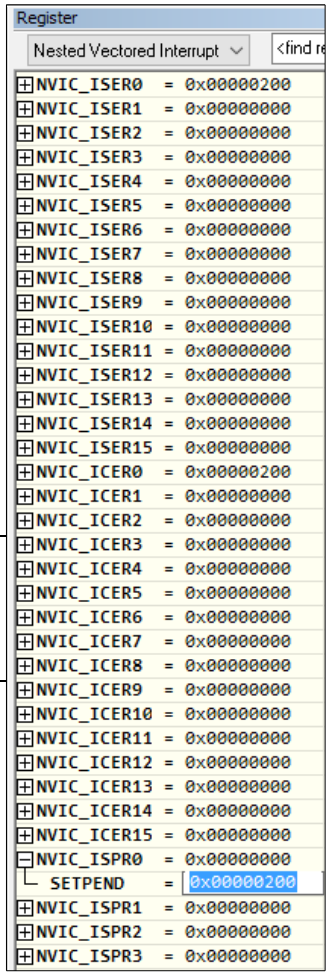
Interrupt Recovery is the time required by the CPU to return to the thread context. There could be two cases here:

3 Case-A occurs when the interrupt returns to the *same* Thread-A.

4 Case-B occurs when the interrupt returns to a *different* Thread-B, which can happen under a preemptive kernel, when the ISR makes the higher-priority Thread-B ready-to-run.

> **NOTE:** Case-B involves the **Context-Switch**, which requires saving the context of Thread-A and restoring the context of Thread-B. Such Context-Switch typically takes more time than simple return to the same thread. Therefore the performance results (see Table 1) list *Interrupt Recovery-A* and *Interrupt Recovery-B* separately for preemptive kernels (QK, QXK, and uC/OS-II).

Testing of interrupt handling requires triggering an interrupt at will. The DPP test application specifically contains an interrupt handler `GPIO_EVEN`, which is specifically designed to be used for testing. In the IAR EWARM debugger you can trigger this interrupt by writing 0x200 to the `NVIC_ISPR0` register.

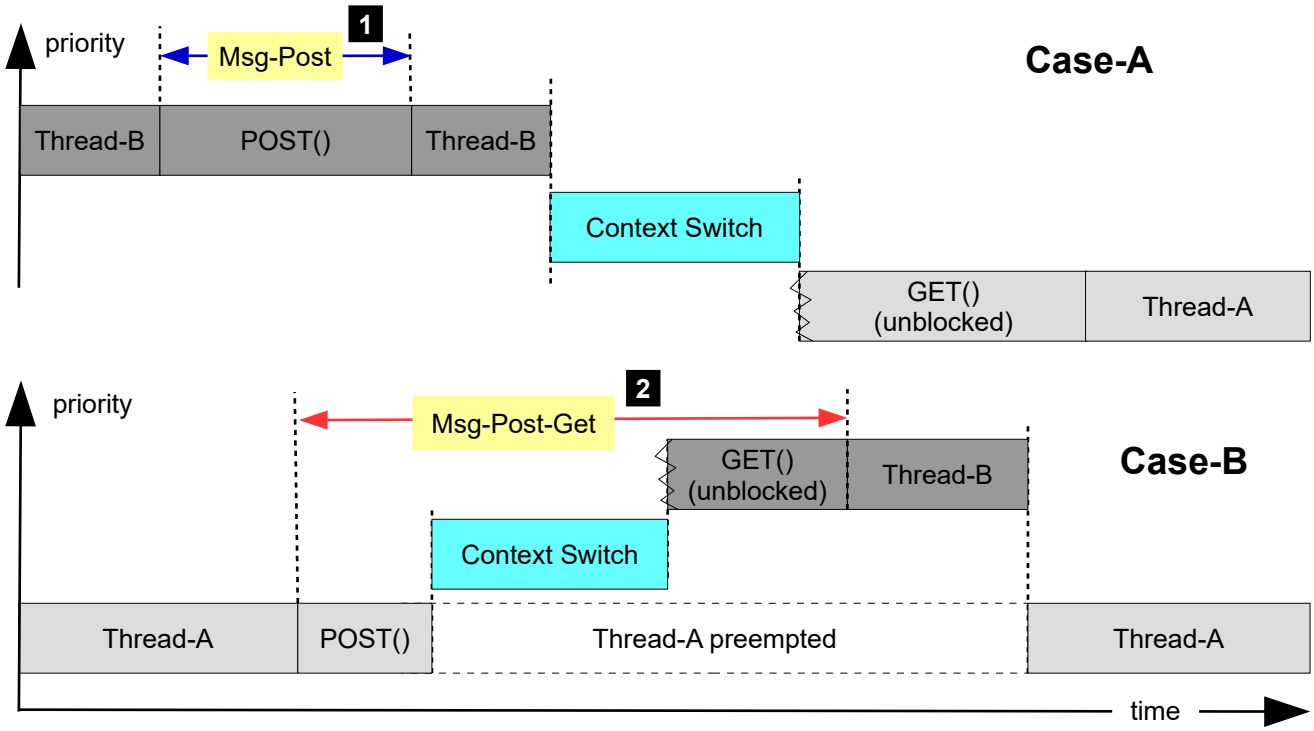


Copyright © Quantum Leaps, LLC. All Rights Reserved.

3.1.2 *Message (Event) Posting*

Thread-safe event-delivery is one of the most important jobs of an event-driven framework, such as QP/C++. The cornerstone of event-delivery is asynchronous event posting from one active object thread to another, or from ISRs to an active object. Event posting is also available in conventional RTOS kernels in the form of posting messages to message queues.

Figure 9: Message (event) posting scenarios



As shown in Figure 9, message posting can lead to two scenarios:

1 **Msg-Post:** Event posting occurs asynchronously and the thread that posts the event continues undisturbed. The time overhead of event posting is just the time spent inside the `POST()` function.

In the DPP example application, this case can be tested when the high-priority `Table` active object posts event to the lower-priority `Philo[n]` in the entry action to state "serving".

2 **Msg-Post-Get:** Under a preemptive kernel, event posting to a higher-priority thread leads to preemption of the lower-priority thread and activation of the higher priority thread. The overhead of the `POST()` and `GET()` operations cannot be easily measured separately, because unblocking happens inside the `GET()` function. Instead, the time overhead of event posting is the time from the entry to the `POST()` function to the return from the `GET()` function.

In the DPP example application, this case can be tested when the low-priority `Philo` active object posts event to the high-priority `Table` in the entry action to state "hungry".

3.1.3 *Memory Pool Get and Put*

QP/C++ implements deterministic fixed-size memory pools (`QMPool`) that are used as event pools, but can also be used for other purposes. uC/OS-II RTOS also implements fixed-size memory pools. Therefore this feature can be compared directly in Table 1.

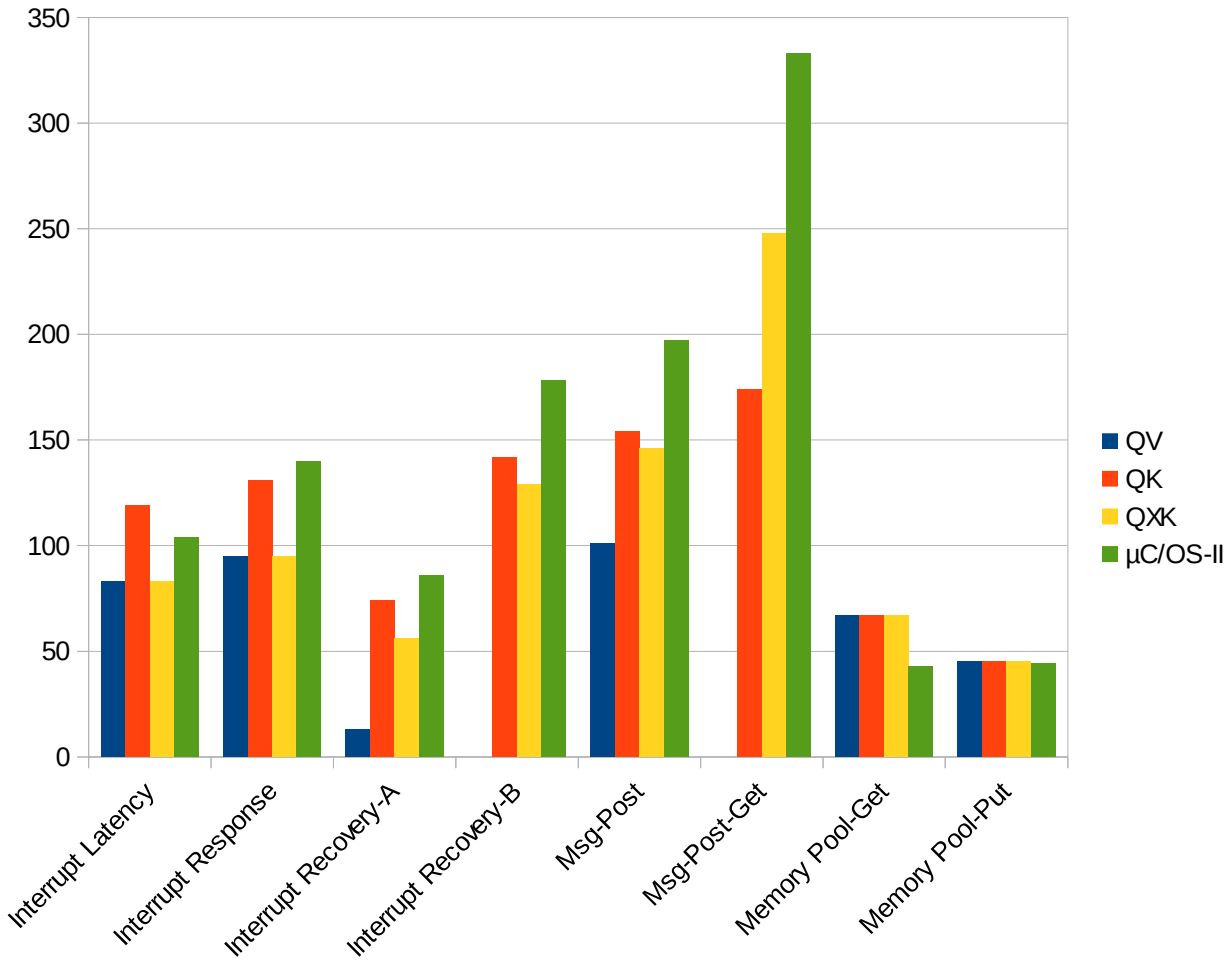
Copyright © Quantum Leaps, LLC. All Rights Reserved.

3.1.4 Group 1 Performance Measurement Results

Table 1: Group-1 measurements in CPU cycles (less is better)

Feature	QV	QK	QXK	µC/OS-II
Interrupt Latency	83	119	83	104
Interrupt Response	95	131	95	140
Interrupt Recovery-A	13	74	56	86
Interrupt Recovery-B	N/A	142	129	178
Msg-Post	101	154	146	197
Msg-Post-Get	N/A	174	248	333
Memory Pool-Get	67	67	67	43
Memory Pool-Put	45	45	45	44

Figure 10: Group-1 measurements in CPU cycles (less is better)



Copyright © Quantum Leaps, LLC. All Rights Reserved.

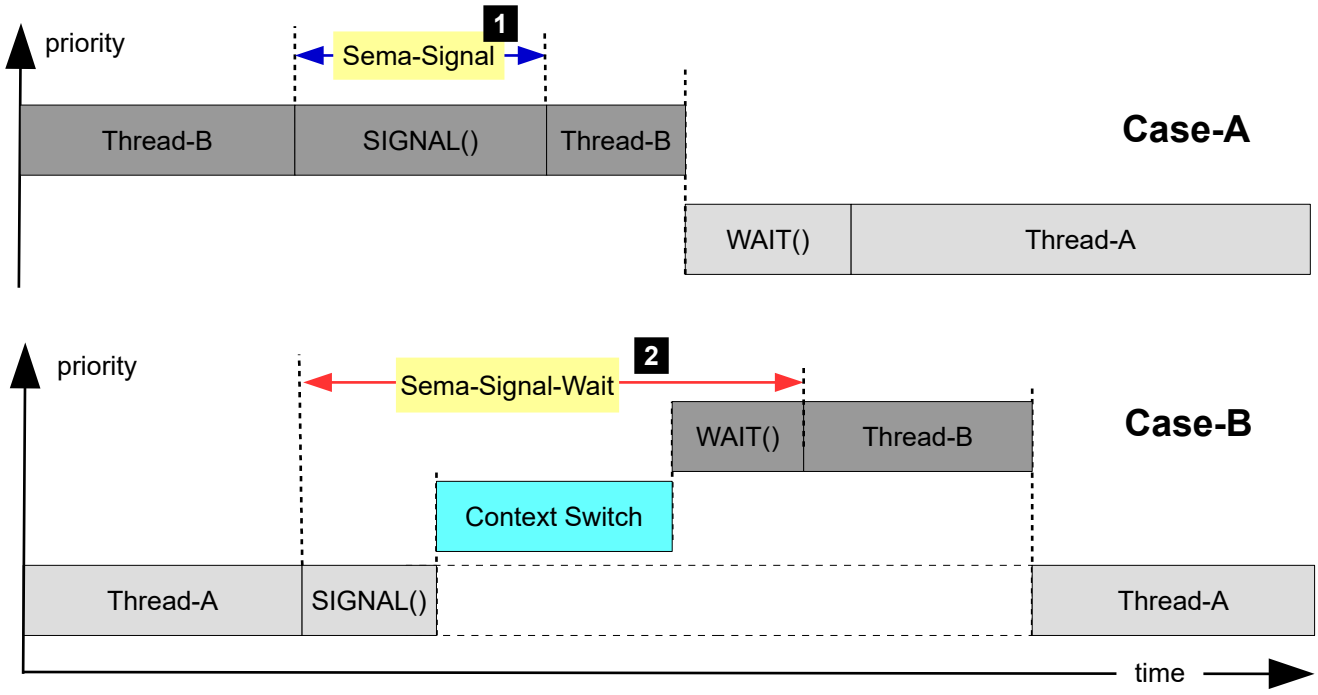
3.2 Group 2 Performance Tests

As described in Section 1.3, Group 2 features are supported only by a conventional RTOS (uC/OS-II) and the QXK kernel.

3.2.1 Semaphores

Virtually all conventional RTOS kernels, including uC/OS-II and QXK, support semaphores as a mechanism to synchronize thread execution. Semaphores are important, because many existing pieces of software, such as commercial middleware and legacy code use them.

Figure 11: Semaphore signaling scenarios



As shown in Figure 11, signaling a semaphore can lead to two scenarios:

1 **Signal-Asynch:** Semaphore signaling occurs asynchronously and the thread (or ISR) that signals the semaphore continues undisturbed. The time overhead of event posting is just the time spent inside the `SIGNAL()` function.

In the DPP example application (QXK or uC/OS-II version), this case can be tested when the BTN1 is depressed and `SysTick_Handler` (in QXK) or `App_TimeTickHook` (in uC/OS-II) BSP signals the semaphore to the `Test` naked thread.

2 **Signal-Wait:** Under a preemptive kernel, signaling a semaphore held by a higher-priority thread leads to preemption of the lower-priority thread and activation of the higher priority thread. The overhead of the `SIGNAL()` and `WAIT()` operations cannot be easily measured separately, because unblocking happens inside the `WAIT()` function. Instead, the time overhead of event posting is the time from the entry to the `SIGNAL()` function to the return from the `WAIT()` function.

In the DPP example application (QXK or uC/OS-II version), this case can be tested when the BTN0 is depressed and `BSP::displayPaused` signals the semaphore to the `Test` naked thread.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

3.2.2 *Mutexes*

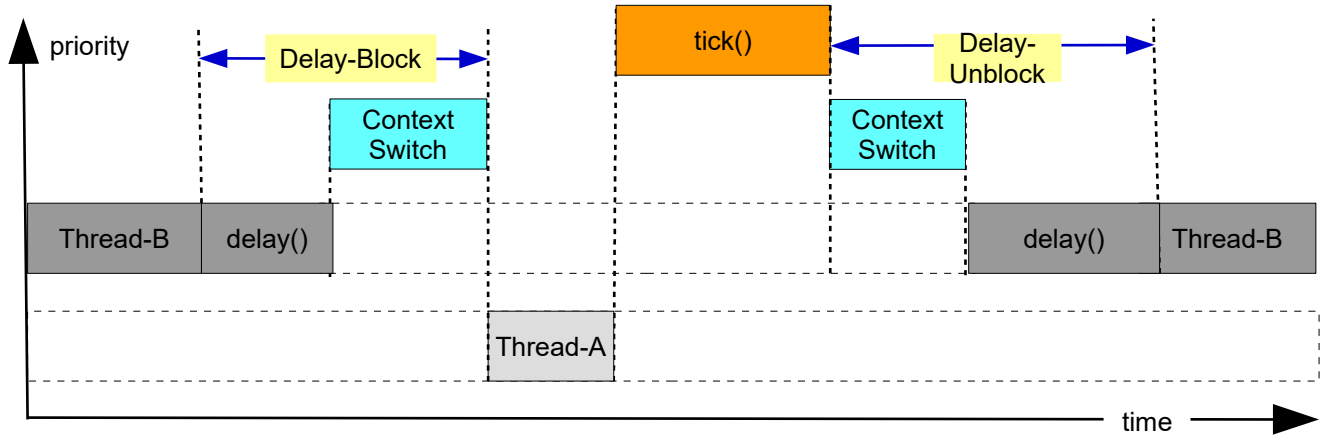
Mutexes (a.k.a. Mutual Exclusion Semaphores) are also frequently used to protect shared resources among RTOS threads. Both QXK and uC/OS-II support priority-ceiling mutex type, so the comparisons can be quite direct.

In the DPP example application (QXK or uC/OS-II version) mutex can be tested inside the `BSP::random()` function.

3.2.3 *In-line delay()*

In-line blocking delay is another very frequently used mechanism to "throttle" thread execution. Both QXK and uC/OS-II support in-line delay, so the comparisons can be quite direct.

Figure 12: In-line delay() measurements



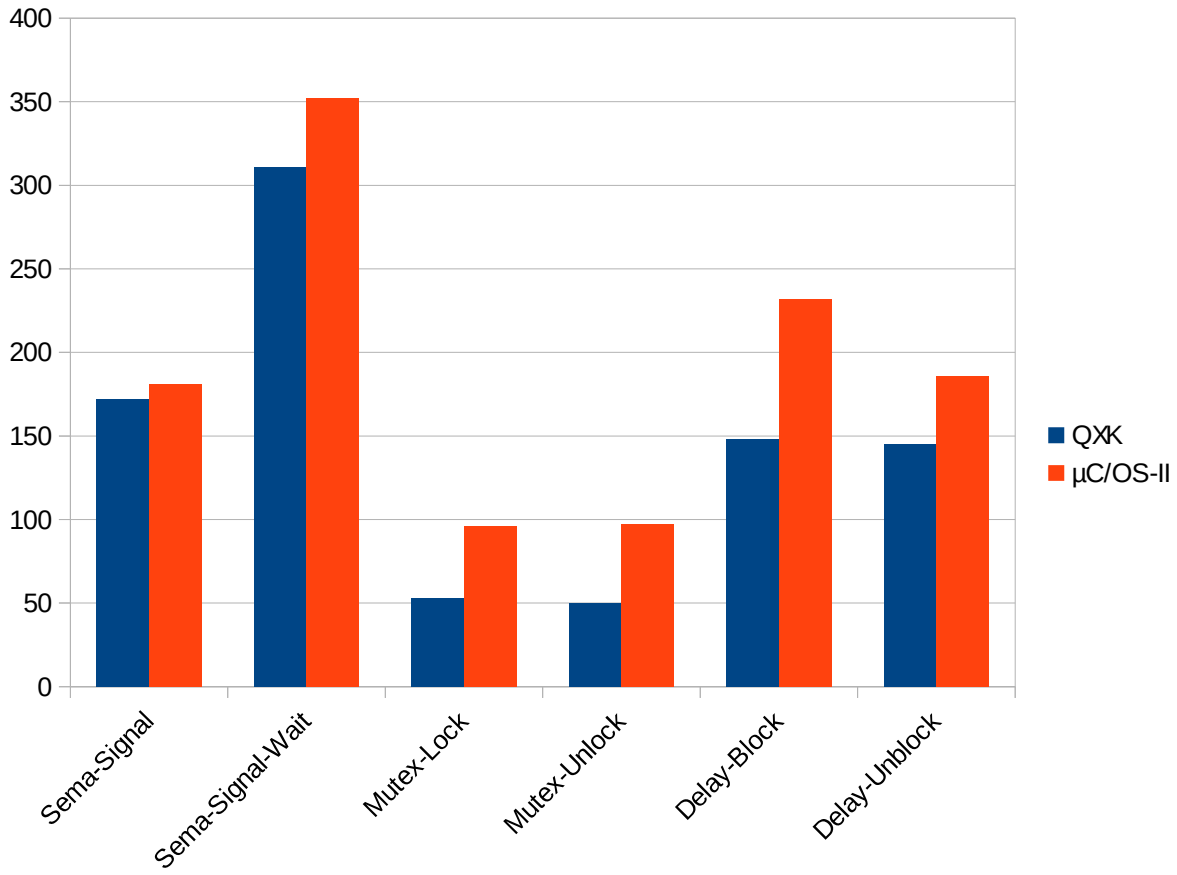
Copyright © Quantum Leaps, LLC. All Rights Reserved.

3.2.4 Group 2 Performance Measurement Results

Table 2: Group-2 measurements in CPU cycles (less is better)

Feature	QXK	µC/OS-II
Sema-Signal	172	181
Sema-Signal-Wait	311	352
Mutex-Lock	53	96
Mutex-Unlock	50	97
Delay-Block	148	232
Delay-Unblock	145	186

Figure 13: Group-2 measurements in CPU cycles (less is better)



Copyright © Quantum Leaps, LLC. All Rights Reserved.

3.3 Group 3 Performance Tests

As described in Section 1.3, Group 3 tests pertain to features only by the QP/C++ and not supported by conventional RTOS. The main purpose of this section is to give you an idea about the overhead of various QP services.

NOTE: With the exception of event posting, Group-3 features use the same QP/C++ code, which does not depend on the underlying real-time kernel. For simplicity, the results in this section are measured only for the **QK kernel**.

3.3.1 Hierarchical State Machines

A complete performance testing of hierarchical state machines could, in principle, require you to test all possible state transition topologies, among states with or without entry/exit actions, across many levels of state nesting, with various guard conditions, etc. Such exhaustive testing is out of scope of this Application Note. Instead, this Application Note tests only the simple state machines available in the DPP example application (see Section 2.3.2). However, this document will present test results for the two implementation strategies of hierarchical state machines used in QP/C++.

- **QMsm/QMActive-based state machines** provide a state machine implementation strategy that requires the assistance of the QM™ tool (as an advanced "state machine compiler") to generate the complete transition-sequences at code-generation time. The resulting code is *significantly more efficient* than the code based on the `QHsm` class and is still highly human-readable, but is not suitable for manual coding or maintaining.

  To test the `QMsm` state machine implementation strategy with the DPP example application, you need to generate the code from the `dpp_qmsm.qm` QM model.

- **QHsm/QActive-based state machines** provide an alternative state machine implementation strategy that was originally designed for manual coding of HSMs, but now can also benefit from automatic code generation by QM™. The older `QHsm/QActive`-style state machines are less efficient in time (CPU cycles) and space (e.g., stack usage) than the newer `QMsm/QMActive`-style State Machines. This is because the `QHsm/QActive`-style implementation strategy requires discovering the transition-sequences (sequences of exit/entry/initial actions) at run time as opposed to code-generation time.

  To test the `QHsm` state machine implementation strategy with the DPP example application, you need to generate the code from the `dpp_qhsm.qm` QM model.

The performance measurements published in Table 3 are labeled as follows:

QMsm-1 denotes the cycle count for dispatching the `EAT_SIG` event to the Philo state machine (state “hungry”) with QMsm implementation strategy and **QHsm-1** for the same event with the QHsm strategy. This event triggers the execution of the following actions: guard condition, transition to “eating”, entry action to “eating” including call to `QTimeEvt::armX()`. This case is heavily impacted by the overhead of arming the time event.

QMsm-2 denotes the cycle count for dispatching the `MAX_PUB_SIG` event to the Table state machine with QMsm implementation strategy and **QHsm-2** for the same event with the QHsm strategy. This event is actually not recognized by the Table state machine, so it “bubbles up” through two levels of state nesting and is eventually silently discarded (per UML semantics).

Copyright © Quantum Leaps, LLC. All Rights Reserved.

3.3.2 *Dynamic Events*

The QP/C++ framework supports dynamic events, which are events that are dynamically allocated from a deterministic event pool and that the framework automatically recycles after they are used. The framework determines when to recycle a dynamic event by means of the reference counter maitained in each event. The performance measurements published in Table 3 are labeled as follows:

Q_NEW denotes the cycle count for allocating a new dynamic event (from the first (small) event pool).

QF_gc denotes the cycle count for recycling (garbage-collect) of a dynamic event

> **NOTE:** The garbage-collect function has much lower overhead when it merely decrements the reference counter, but it does not actually recycle the event. The QF_gc measurement in Table 3 is for the high-overhead case of actually recycling a dynamic event to the pool.

3.3.3 *Event Posting*

The overhead of direct event posting has been already covered in Group 1. The performance measurements published in Table 3 are labeled as follows:

QActive_POST denotes asynchronous event posting from a high-priority active object (or ISR)

QActive_POST-GET denotes the overhead of delivering event from low-priority AO to high-priority AO.

3.3.4 *Event Publishing*

The QP/C++ framework supports the publish/subscribe event delivery, which involves mutlicasting of events to multiple subscribers. Consequently, the overhead of event publishing depends on the number of subscribers *n*. The dependency is approximately linear:

*QF_PUBLISH(n) = a * n + b*, where *a, b* are constants and *n* is the number of subscribers

The performance measurements published in Table 3 are labeled as follows:

QF_PUBLISH-A denotes the overhead of publishing an event to one subscriber

QF_PUBLISH-B denotes the constant overhead of publishing

> **NOTE:** The QF_PUBLISH-A and QF_PUBLISH-B coefficients have been calculated for the two data points: when Philo active object publishes DONE event to the Table active object (n==1), and when Table publishes EAT event to all Philo objects (n==5).

3.3.5 *Time Events*

QP/C++ provides an event-driven time-management mechanism in form of time events (`QTimeEvt` class). The most frequently used services in this category include arming and disarming time events. The performance measurements published in Table 3 are labeled as follows:

QTime-Arm denotes the overhead of arming a time event

QTime-Disarm denotes the overhead of disarming a time event

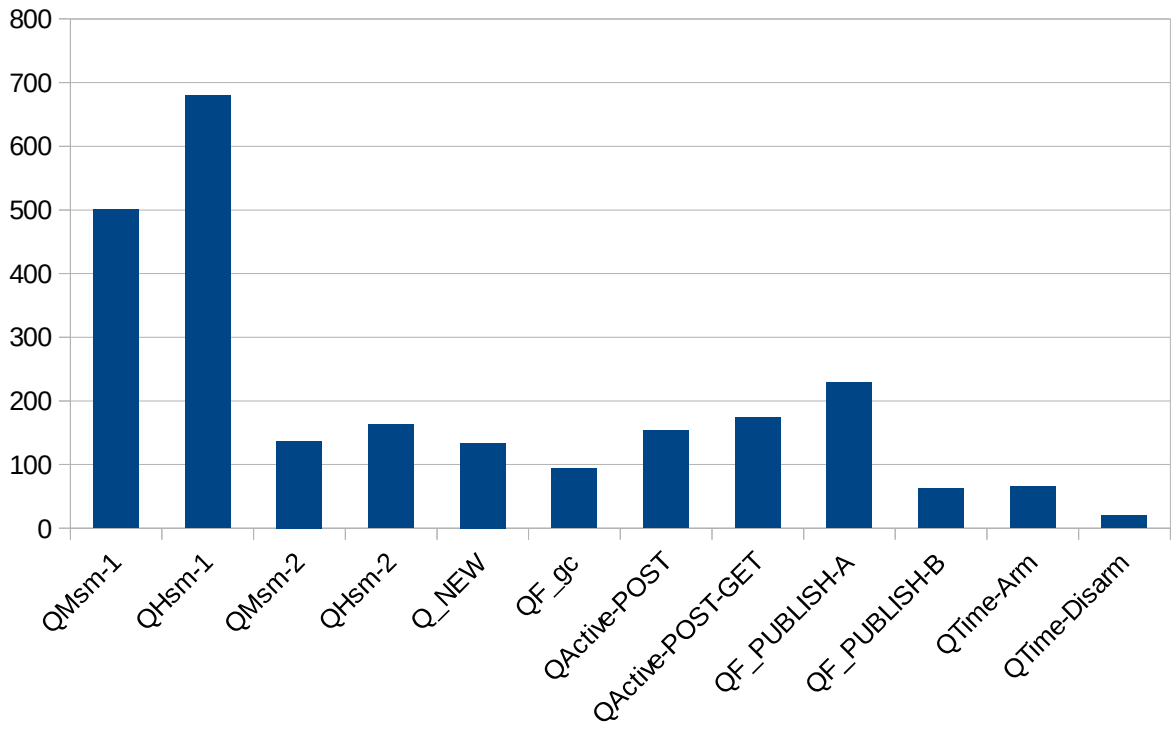
Copyright © Quantum Leaps, LLC. All Rights Reserved.

3.3.6 Group 3 Performance Measurement Results

Table 3: Group-3 measurements in CPU cycles (less is better)

Feature	CPU cycles (QK kernel)
QMsm-1	501
QHsm-1	680
QMsm-2	137
QHsm-2	163
Q_NEW	134
QF_gc	94
QActive-POST	154
QActive-POST-GET	174
QF_PUBLISH-A	229
QF_PUBLISH-B	63
QTime-Arm	66
QTime-Disarm	20

Figure 14: Group-3 measurements in CPU cycles (less is better)



Copyright © Quantum Leaps, LLC. All Rights Reserved.

4 Memory Size Measurements

This section describes the code size (ROM) and data size (RAM) measurements for various configuration options of QP/C++ and compares them to the results for uC/OS-II.

4.1 Code Size Measurements

The code size measurements are based on the linker map files produced for all 4 tested configurations (QV, QK, QXK, and uC/OS-II).

4.1.1 Built-in Kernels (QV, QK, and QXK)

The following Listing 2 shows an example of the map file for the DPP project with the QK kernel. The code size of QP/C++ is calculated as the sum of all modules comprising QP/C++ (highlighted in the listing).

Listing 2: Module Summary section of the linker map file for the DPP example with QK. The highlighted part corresponds to the QP/C++ framework

```
*******************************************************************************
*** MODULE SUMMARY
***

    Module                ro code  ro data  rw data
    ------                -------  -------  -------
C:\qp_lab\qpcpp\examples\performance\dpp_efm32-slstk3401a\qk\iar\dbg: [1]
    bsp.o                     572        8       20
    em_cmu.o                  144
    em_gpio.o                 120
    main.o                    188              236
    philo.o                   648       68      341
    qep_hsm.o                 660
    qep_msm.o                 628       28
    qf_act.o                   76      204      132
    qf_actq.o                 336
    qf_dyn.o                  292        4       64
    qf_mem.o                  336
    qf_ps.o                   320                 8
    qf_qact.o                  40       44
    qf_qeq.o                   42
    qf_qmact.o                 60
    qf_time.o                 364        4       32
    qk.o                      488       44       44
    qk_mutex.o                156
    qk_port.o                 112
    startup_efm32pg1b.o       284
    system_efm32pg1b.o         84       16       16
    table.o                 1 166       52       64
    ----------------------------------------------
    Total:                  7 116      472      957

dl7M_tln.a: [3]
    cppinit.o                 148               20
    exit.o                      4
    low_level_init.o            4
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
    ------------------------------------------------
    Total:                     156                20

dlpp7M_tl_ne.a: [4]
    cxxabi.o                    82
    ------------------------------------------------
    Total:                      82

m7M_tl.a: [5]
    FltAdd.o                   132
    FltSub.o                   214
    ------------------------------------------------
    Total:                     346

rt7M_tl.a: [6]
    ABImemclr.o                  6
    ABImemset.o                 94
    cexit.o                     14
    cmain.o                     26
    cmain_call_ctors.o          32
    copy_init3.o                44
    cstart_call_dtors.o
    cstartup_M.o                12
    data_init.o                 40
    zero_init3.o                64
    ------------------------------------------------
    Total:                     332

shb_l.a: [7]
    exit.o                      20
    ------------------------------------------------
    Total:                      20

    Gaps                        12
    Linker created                          40    1 420
----------------------------------------------------------
    Grand Total:             8 064         512    2 397
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

4.1.2 *Conventional RTOS (uC/OS-II)*

The following Listing 3 shows an example of the map file for the DPP project with the uC/OS-II RTOS kernel. The code size of uC/OS-II is calculated as the sum of all modules comprising uC/OS-II (highlighted in the listing).

Listing 3: Module Summary of the linker map file for the DPP example with uC/OS-II. The highlighted part corresponds to the uC/OS-II kernel

```
*******************************************************************************
*** MODULE SUMMARY
***

    Module                ro code  ro data  rw data
    ------                -------  -------  -------
C:\qp_lab\qpcpp\examples\performance\dpp_efm32-slstk3401a\ucos-ii\iar\dbg: [1]
    bsp.o                     592        8       16
    em_cmu.o                  144
    em_gpio.o                 120
    main.o                    440            4 296
    os_core.o               1 384            1 780
    os_cpu_a.o                172
    os_cpu_c.o                272              516
    os_dbg.o                                     2
    os_mem.o                  220
    os_mutex.o                860
    os_q.o                    716
    os_sem.o                  334
    os_task.o                 444
    os_time.o                  92
    philo.o                   636      128      201
    qep_msm.o                 624       28
    qf_act.o                  124      460       56
    qf_dyn.o                  332               16
    qf_port.o                 376       44
    qf_ps.o                   396                8
    qf_qmact.o                 44
    qf_time.o                 368        4       16
    startup_efm32pg1b.o       284
    system_efm32pg1b.o         84       16       16
    table.o                 1 074      112       36
    test.o                    132                4
    ----------------------------------------------
    Total:                 10 264      802    6 961

    . . .

    Gaps                       10
    Linker created                      40      512
-----------------------------------------------------
    Grand Total:           10 750      842    7 473
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

4.2 Data Size Measurements

The data size measurements are also based on the linker map files, which contain that information as well.

4.3 Stack Size Measurements

The memory required for the stack(s) is one of the most important contribution to the RAM footprint of a real-time kernel. To make the comparison between the QP/C++ build-in kernels and the uC/OS-II 3rd-party RTOS more fair, the stack size cannot be simply listed as the total pre-allocated stack space, because it contains arbitrary “padding”. Instead, the stack size listed in the memory size measurements (Table 4) is the actual stack space used by the application. This stack space is measured by the standard method of pre-filling the stack RAM with a known bit-pattern, running the application for a while, and inspecting in memory how much stack space has been overwritten (used).

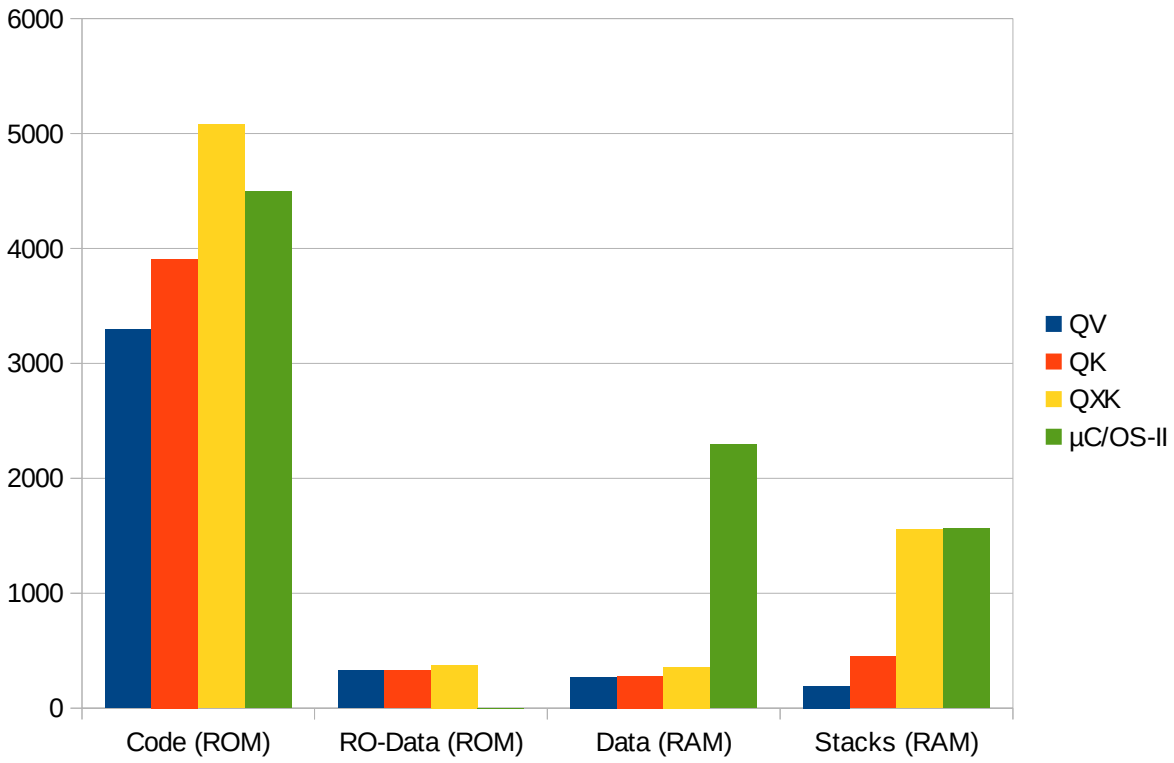
Copyright © Quantum Leaps, LLC. All Rights Reserved.

4.4 Memory Size Measurement Results

Table 4: Memory size measurement results in bytes (less is better)

Memory Type	QV	QK	QXK	µC/OS-II
Code (ROM)	3294	3910	5080	4494
RO-Data (ROM)	328	328	376	2
Data (RAM)	272	280	356	2296
Stacks (RAM)	192	448	1552	1564

Figure 15: Memory size measurement results in bytes (less is better)



Copyright © Quantum Leaps, LLC. All Rights Reserved.

5 Summary

In the resource-constrained embedded systems, the biggest concern has always been about the size and efficiency of any piece of code, especially if the new software introduces a paradigm shift with respect to established conventions.

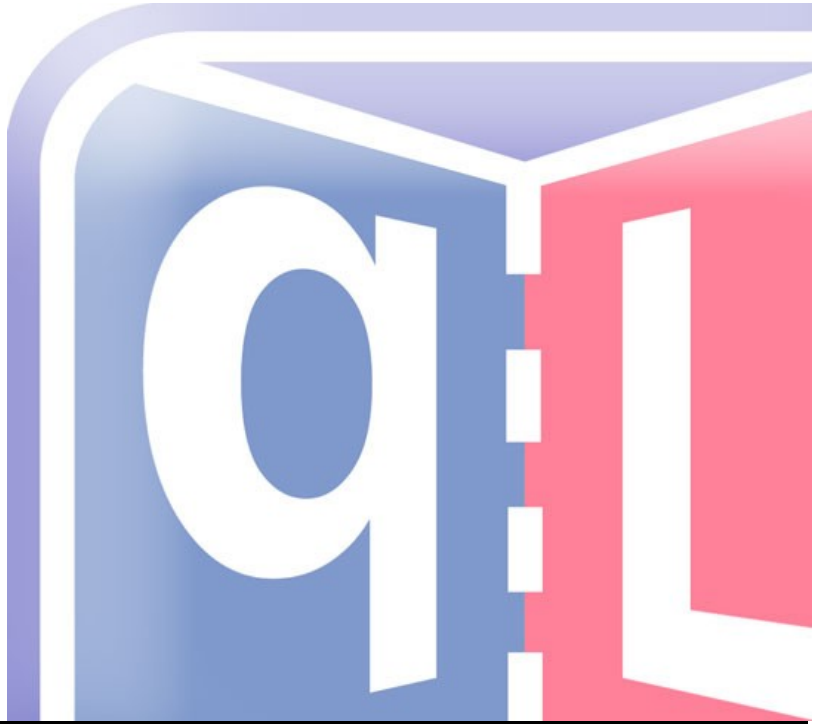
However, as it turns out an active object framework, like QP/C++, can be both smaller (in code size) and more efficient (in CPU cycles) than a conventional RTOS, such as uC/OS-II. This is possible, because event-driven programming paradigm is known to require less resources, especially RAM for the stacks, than sequential programming based on shared-state concurrency and blocking.

All these characteristics make event-driven active objects a perfect fit for resource-constrained embedded systems, such as single-chip microcontrollers (MCUs), systems on chip (SoCs), etc.. Not only you get the productivity boost by working at a **higher level of abstraction** than “naked” RTOS threads, but you get it at a lower memory utilization and better CPU efficiency.

6 Contact Information

Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
USA

WEB : http://www.state-machine.com
Email: mailto:info@state-machine.com



Copyright © Quantum Leaps, LLC. All Rights Reserved.