This presentation highlights the new features in the new QP5 milestone release.

**What is it?**
QP™ is a family of lightweight, open source software frameworks for building responsive and modular real-time embedded applications as systems of cooperating, event-driven active objects (actors). The QP™ family consists of QP/C, QP/C++, and QP-nano frameworks, which are all strictly quality controlled, superbly documented, and commercially licensable.

**Where does it run?**
All QP™ frameworks can run on "bare-metal" single-chip microcontrollers, completely replacing a traditional Real-Time Operating System (RTOS). Ports and ready-to-use examples are provided for most major CPU families. QP/C and QP/C++ can also work with a traditional OS/RTOS, such as: POSIX (Linux, embedded Linux, QNX, INTEGRITY), Win32 (Windows, Windows embedded, Windows CE), ThreadX, MicroC/OS, etc.

**How does it handle behavior?**
The behavior of active objects is specified in QP by means of hierarchical state machines (UML statecharts). The frameworks support manual coding of UML state machines in C or C++ as well as fully automatic code generation by means of the free graphical QM™ modeling tool.

**Who is using it?**
The QP frameworks are used in millions of products worldwide in aerospace, robotics, consumer electronics, wired and wireless telecommunications, industrial automation, transportation, and many more. The QP™ frameworks and the QM™ modeling tool receive over 30,000 downloads a year (not even counting downloads of QP ports).
The main purpose of this milestone QP5 release is to enable the QM modeling tool to generate a new type of state machine code (requires QM version 3).

Specifically, QM can generate the complete transition-sequences (sequences of exit/entry/initial actions) at code-generation time instead of discovering the transition-sequences at run-time. This requires a different (much simpler) event processor than the one provided in the QHsm class. QP5 provides such new event processor in the new class QMsm.

**NOTE:** QP5 5.1 remains backwards-compatible with the existing QP 4.x applications, except the ARM Cortex-M applications need to adjust the interrupt priorities. Specifically, you need to set the interrupt priorities equal or numerically higher than QF_AWARE_ISR_CMSIS_PRI constant provided in the qf_port.h header file.
The new QMsm Base Class

The new type of state machine implementation in QP5 is based on the new **QMsm** class, which takes advantage of the QM tool as an advanced "state machine compiler". The QMsm class provides the new implementation of the **init()** and **dispatch()** functions. QMsm provides also the virtual pointer, which enables **polymorphism** (late binding).

Coding state machines based on QMsm requires the assistance of the QM tool to generate the complete transition-sequences (sequences of exit/entry/initial actions) at code-generation time. The resulting code can be still highly **human-readable**, but it will no longer be human-maintainable. The lab tests indicate that the new "housekeeping" code for executing hierarchical state machines can be about twice as fast as the previous code based on the QHsm class.

Additionally, the new code requires less run-time support (smaller event processor) and uses 70% less of stack space in the call to the QMsm_dispatch() operation than QHsm_dispatch().
New State Machine Code

typedef struct QMsmTstTag {
    QMsm super; /* inherits QMsm */
    uint8_t foo; /* some attributes */
} QMsmTst;

void QMsmTst_ctor(void);

static QState QMsmTst_initial(QMsmTst * const me);

static QMState const QMsmTst_s1_ = {
    (QMState const )0, /* the superstate (NULL==top) */
    Q_STATE_CAST(&QMsmTst_s1), /* state-handler function */
    Q_ACTION_CAST(&QMsmTst_s1_x) /* state-exit action (if present) */
};

static QState QMsmTst_s1(QMsmTst * const me, QEvt const * const e);
static QState QMsmTst_s1_e(QMsmTst * const me);
static QState QMsmTst_s1_x(QMsmTst * const me);
static QState QMsmTst_s1_i(QMsmTst * const me);

static QMState const QMsmTst_s11_ = {
    &QMsmTst_s1_, /* the superstate */
    Q_STATE_CAST(&QMsmTst_s11), /* state-handler function */
    Q_ACTION_CAST(&QMsmTst_s11_x) /* state-exit action (if present) */
};

static QState QMsmTst_s11(QMsmTst * const me, QEvt const * const e);
static QState QMsmTst_s11_e(QMsmTst * const me);
static QState QMsmTst_s11_x(QMsmTst * const me);
static QState QMsmTst_s11_i(QMsmTst * const me);

****

NOTE: The new state machine code is **not** intended to be written manually. This code is generated **automatically** by the QM tool from the state diagram. The explanation provided here is only intended to help you understand this code structure, so that you can work with it effectively (e.g., inside a traditional source-level debugger).

****

Each state is represented as:
1. QMState object (const in ROM)
2. state-handler function
3. optional entry-action function
4. optional exit-action function
5. optional initial-transition function

The QMState object provides the information about nesting of the state, the associated state-handler function, and the exit-action function, if present.

The state-handler function no longer handles entry/exit/initial transition. The entry/exit/initial transition are handled in separate functions associated with a state. These functions are only defined when needed.
The constructor is the same as before, except it calls the superclass’ ctor QMsm_ctor().

The top-most initial transition handler function requires a transition-action array, which defines the sequence of actions to execute upon the initial transitions. The generation of this array is difficult by hand, but it is easy for the QM tool. The transition-action array is static and const, which means that most compilers will place it in ROM. The array is zero-terminated.

The initial-transition must return through the macro QM_INITIAL().
The entry/exit actions and per-state initial transitions are separate action-handler functions. These functions are only defined if the given state actually has an entry action, exit action or an initial transition.

The per-state initial transition is coded identically to the top-most initial transition.
New State Machine Code (cont'd)

QsmTst_s1(QsmTst * const me, QEvt const * const e) { /* state-handler fun. */
    QState status_;  
    switch (e->sig) {
    case A_SIG: {
        static QActionHandler const act_[] = {
            Q_ACTION_CAST(&QMsmTst_s1_x),
            Q_ACTION_CAST(&QMsmTst_s2_e),
            Q_ACTION_CAST(0)
        };
        BSP_display("s1-A; ");
        status_ = QM_TRAN(&QMsmTst_s2_,
            act_);
        break;
    }
    case I_SIG: {
        BSP_display("s1-I; ");
        status_ = QM_HANDLED();
        break;
    }
    default: {
        status_ = QM_SUPER();
        break;
    }
    }
    return status_;  
}

The state-handler function handles only the explicit events, but it does not handle the special “events” Q_ENTRY_SIG, Q_EXIT_SIG, or Q_INIT_SIG.

Every transition requires its own transition-action array.
### Course Outline

- Enable QM to generate new type of SM code
  - **Provide virtual functions**
- Multiple system clock tick rates
- Non-asserting event allocation and posting
- Kernel-Unaware Interrupts for ARM Cortex-M3/M4
- Quick Summary of Other Changes

QP5 introduces polymorphism (virtual functions) for most important operations within the framework, such as state machine init() and dispatch() and active object start(), post(), and postLIFO() operations. Making theses functions "virtual" means that all these operations can be re-defined in subclasses of state machines and active objects.
QP5 provides virtual functions for the important framework operations. The QMsm base class provides the standard \texttt{vptr->vtable} constructs, which make up an indirection level for calling functions, as implemented in the macros \texttt{QMSM_INIT()} and \texttt{QMSM_DISPATCH()}.  

\textbf{****}  

NOTE: You don't need to know exactly how polymorphism is implemented in C to take advantage of the provided “virtual hooks”. Just call the respective operations using the macros, such as \texttt{QMSM_INIT()} or \texttt{QMSM_DISPATCH()} rather than the specific implementations, such as \texttt{QMsm_init()} or \texttt{QHsm_dispatch()}.  

\textbf{****}  

The \texttt{vptr} is initialized in the QMsm constructor, but each subclass of QMsm (direct or transitive) has an opportunity to define its own virtual table with different implementations and hook the \texttt{vptr} to this table.  

The QP5 framework invokes the operations \texttt{init()} and \texttt{dispatch()} only via the macros \texttt{QMSM_INIT()} and \texttt{QMSM_DISPATCH()}, which means that these “hooks” are in place for customization in subclasses.
The **QActive** class in QP5 extends the vtable inherited from QMsm by adding virtual functions for the start(), post(), and postLIFO() operations. The polymorphic way of invoking these operations is provided my means of the macros QACTIVE_START(), QACTIVE_POST(), and QACTIVE_POST_LIFO(), respectively.

****

NOTE: The macros QACTIVE_START(), QACTIVE_POST(), and QACTIVE_POST_LIFO() are the recommended way of calling the respective operations. You should avoid calling the specific implementations directly (e.g., QActive_start(), QActive_post()).

****

The **QMActive** class inherits all these operations from QActive, but it overrides the init() and dispatch() operations to use the implementation from the QMsm base class.
QM to generate new type of SM code
• Provide “hooks” for customization (virtual functions)

Multiple system clock rates
• Non-asserting event allocation and posting
• Kernel-Unaware Interrupts for ARM Cortex-M3/M4
• Quick Summary of Other Changes

QP5 supports multiple, independent system clock tick rates for time events. The number of system tick rates can be now configured in the QP/C ports. For example, a digital watch can use a "fast" clock tick rate of 100Hz and a "slow" clock tick rate of only 1Hz. These clock tick rates can be managed independently, so for example, the fast clock tick rate can be shut down in the absence of time events assigned to this rate. This feature allows the applications to implement sophisticated power-saving policies.
QP5 adds support for multiple system clock tick rates. The number of system tick rates can be now configured in the QP/C ports by defining the macro \texttt{QF\_MAX\_TICK\_RATE} (default value is 1 rate).

To support this, QP5 adds an “extended” tick handler \texttt{QF\_tickX(tickRate)}, which takes the tickRate argument. The \texttt{QF\_tickX()} function has been carefully designed to be callable from ISR and from the task level, including low-priority tasks.

***

NOTE: The recommended way of calling \texttt{QF\_tickX()} is through the macro \texttt{QF\_TICK\_X(rate, sender)}, which supplies the sender of the tick for QS software tracing.

***

Every tick rate can be managed independently from the other rates, meaning that a given rate \texttt{n} can be shut-down (the corresponding \texttt{QF\_tickX}(\texttt{n}) can be stopped being called) while other rates can continue. Shutting down a tick rate is recommended only if there are no active time events assigned to this rate. The new API \texttt{QF\_noTimeEvtsActiveX(rate)} is provided to test for this. This function must be called from within a critical section.
The new “extended” TimeEvt constructor takes the active object, signal, and **tick-rate**.

```c
void QTimeEvt_ctorX(QTimeEvt * const me, QActive *me, enum_t sig, uint8_t tickRate);
```

NOTE: this constructor assigns a time event to a given active object and a given tick rate.

The “extended” arm operation serves both one-shot and periodic time events:

```c
void QTimeEvt_armX(QTimeEvt * const me, QTimeEvtCtr nTicks, QTimeEvtCtr interval);
```

Assigning 0-interval arms a **one-shot** time event. A non-zero interval arms a periodic time event. The first period and the interval could be different, which helps adjust the phasing of periodic time events on the first arm operation.
QP5 provides a low-latency implementation of QF_tickX(), and QTimeEvt_*( ) operations. The QF_tickX() can also be called from low-priority tasks, which means that it can be preempted and must handle correctly arming/disarming of time events during preemption.

This is achieved by implementing the following rules:

1. **Only** QF_tickX() function can change the main list of the time events (originating from QF_timeEvtHead[n].next pointer).
2. Disarming of a time event does **not** remove it from the list, it merely marks the time event for removal.
3. Arming a time event does **not** add it to the main list. Instead, arming a time event adds a time event to the “newly armed” list, based on the QF_timeEvtHead[n].act pointer (reused here as a link pointer).
4. The **QF_tickX()** function appends the “newly armed” list **atomically** to the main list, so that rule #1 is not violated.
QP5 adds a new "extended" API for non-asserting event allocation and posting. This feature is intended for situations, when an application is hammered with external events that at times arrive too fast for processing, but that can be ignored under the overload conditions.
Non-Asserting Event Allocation

- New macro Q_NEW_X() requires a `margin` argument
- Margin specifies the # unused events that must exist in the pool

```c
MyEvt *myevt;
Q_NEW_X(myevt, MyEvt, 5U, MY_EVT_SIG); /* "extended" event allocator */
if (myevt != (MyEvt *)0) { /* mevt might be NULL; have to test! */
    myevt->foo = 123;
    myevt->bar = 0xABC;
    ~~~~
}
```

The new “extended” event allocator Q_NEW_X() requires a **non-zero** margin argument, which specifies the number of unused events in the pool (of the given size) for the allocation to succeed. If the pool contains less than the margin of events, Q_NEW_X() will set the event argument to NULL, but it will **not assert**.

You must always test the event argument for NULL.

****

**NOTE:** The margin parameter allows you to keep sufficient number of still unused events in the pool for the “normal” event allocation, which asserts when the pool runs out of events.

****

The internal implementation of the native memory pool QMPool_get() operation has been augmented to take the margin argument.

When the QP port is configured to use “event constructors”, the macro Q_NEW_X() takes the variadic form and calls the event constructor only when the allocation succeeds.
Non-Asserting Event Posting

• New macro QACTIVE_POST_X() requires **margin** argument

Margin specifies the # unused entries that must exist in the event queue

The new “extended” event posting QACTIVE_POST_X() requires a **non-zero** margin argument, which specifies the number of unused entries in the event queue for the posting to succeed. If the queue contains less than the margin of unused entries, QACTIVE_POST_X() will **not** post the event and it will return 0 (FALSE), but it will **not** assert.

*****

NOTE: The margin parameter allows you to keep sufficient number of unused entries in the queue for the “normal” event posting, which asserts when the queue overflows.

*****

The QActive_post() operation has been augmented to take the margin argument and return the status. The operation asserts if the margin is zero and the posting cannot succeed.

The non-asserting event posting is not available for time events or for publishing events.

*****

NOTE: This non-asserting event posting should be reserved for special situations and should be used with caution. The main mechanisms for event allocation and posting should remain the **asserting** versions (Q_NEW, QACTIVE_POST, QF_PUBLISH), because they provide the event-delivery guarantee.

*****

The new “extended” event posting QACTIVE_POST_X() requires a **non-zero** margin argument, which specifies the number of unused entries in the event queue for the posting to succeed. If the queue contains less than the margin of unused entries, QACTIVE_POST_X() will **not** post the event and it will return 0 (FALSE), but it will **not** assert.

*****

NOTE: The margin parameter allows you to keep sufficient number of unused entries in the queue for the “normal” event posting, which asserts when the queue overflows.

*****

The QActive_post() operation has been augmented to take the margin argument and return the status. The operation asserts if the margin is zero and the posting cannot succeed.

The non-asserting event posting is not available for time events or for publishing events.

*****

NOTE: This non-asserting event posting should be reserved for special situations and should be used with caution. The main mechanisms for event allocation and posting should remain the **asserting** versions (Q_NEW, QACTIVE_POST, QF_PUBLISH), because they provide the event-delivery guarantee.

*****
The QP 5.1 port to ARM Cortex-M3/M4 never completely disables interrupts, even inside the critical sections. The highest-urgency interrupts (prioritized in the NVIC hardware) that are never disabled are “unaware” of the real-time kernel and run completely freely (this is called sometimes “zero interrupt latency”). Such “kernel unaware” interrupts can be useful for handling very fast (sub microsecond) deadlines, for example quickly re-arming a DMA channel, fast communication, etc.

****

NOTE: “Kernel-unaware” interrupts cannot call any QP services and specifically they cannot post events.

****

NOTE: The only way “kernel-unaware” interrupts can communicate with the QP framework is to trigger a “kernel-aware” interrupt, which can post events to QP.

****
### Changed Structure of ARM Cortex-M Ports

Renamed port/examples directory to “arm-cm”
- The previous name “arm-cortex” could be confusing for Cortex-A

Moved the CMSIS directory to “ports\arm-cm\cmsis”
- Instead of replicating it in each example project
- Updated CMSIS to version 3.20

Updated toolset versions, added ARM/Keil to the QP

Baseline code

The QP 5.1 port to ARM Cortex-M has changed the directory structure to avoid any confusion with other CPU cores in the ARM Cortex family (such as Cortex-R or Cortex-A)

CMSIS directory with core CMSIS header files is now inside the ARM Cortex-M port
**Kernel-Unaware and Kernel-Aware Interrupts**

• QP 5.1 port to ARM Cortex-M3/M4 never completely disables interrupts

*Interrupts are disabled selectively using the BASEPRI register

<table>
<thead>
<tr>
<th>Interrupt type</th>
<th>NVIC priority bits</th>
<th>Priority for CMSIS NVIC_SetPriority( )</th>
<th>CMSIS_PRI</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel-unaware</td>
<td>0 0 0 0 0 0</td>
<td>0</td>
<td>0</td>
<td>Never disabled</td>
</tr>
<tr>
<td>Kernel-aware</td>
<td>0 0 0 0 0 1</td>
<td>1 = QF_AWARE_ISR_CMSIS_PRI</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 0 0 0 0</td>
<td>4</td>
<td></td>
<td>Disabled in critical sections</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0 0 0</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 0 0 0 0</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PendSV interrupt</td>
<td>1 1 0 0 0 0</td>
<td>7</td>
<td></td>
<td>Should not be used for regular interrupts</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Only Kernel-Aware Interrupts are allowed to call QP services!

The QP 5.1 port to ARM Cortex-M3/M4 never completely disables interrupts, even inside the critical sections. On Cortex-M3/M4 (ARMv7-M architectures), the QP port disables interrupts selectively using the BASEPRI register.

As shown in the picture, this policy divides interrupts into “kernel-unaware” interrupts, which are never disabled, and “kernel-aware” interrupts, which are disabled in the QP critical sections.

****

NOTE: Only “kernel-aware” interrupts are allowed to call QP services. “Kernel-unaware” interrupts are not allowed to call any QP services and they can communicate with QP only by triggering a “kernel-aware” interrupt (which can post or publish events).

*****

NOTE: The number of interrupt priority bits in the NVIC is “implementation dependent”, meaning that various silicon vendors can provide different number of bits. The minimum number of bits is 3, as shown in the picture.

*****

INFO: The BASEPRI register is not implemented in the ARMv6-M architecture (Cortex-M0/M0+), so Cortex-M0/M0+ need to use the PRIMASK register to disable interrupts globally. In other words, in Cortex-M0/M0+ ports, all interrupts are “kernel-aware”.

(c) Quantum Leaps, LLC
Assigning ARM Cortex-M Interrupt Priorities

/*!!!!!!!!!!!!!!!!!!!!!!!!!!!! CAUTION !!!!!!!!!!!! !!!!!!!!!!!!!!!!!!!!!!!
 * Assign a priority to EVERY ISR explicitly by calling NVIC_SetPriority().
 * DO NOT LEAVE THE ISR PRIORITIES AT THE DEFAULT VALUE!
 */
enum KernelUnawareISRs {
    MAX_KERNEL_UNAWARE_CMSIS_PRI /* keep always last */
};
/* "kernel-unaware" interrupts can't overlap "kernel-aware" interrupts */
Q_ASSERT_COMPILE(MAX_KERNEL_UNAWARE_CMSIS_PRI <= QF_AWARE_ISR_CMSIS_PRI);
enum KernelAwareISRs {
    GPIOPORTA_PRIO = QF_AWARE_ISR_CMSIS_PRI, /* see NOTE00 */
    ADCSEQ3_PRIO,
    SYSTICK_PRIO,
    MAX_KERNEL_AWARE_CMSIS_PRI /* keep always last */
};
/* "kernel-aware" interrupts should not overlap the PendSV priority */
Q_ASSERT_COMPILE(MAX_KERNEL_AWARE_CMSIS_PRI
                  <= (0xFF >> (8-__NVIC_PRIO_BITS)));
The enumeration KernelUnawareISRs lists the priority numbers for the “kernel-unaware”
interrupts. These priorities start with zero (most urgent). The priorities are suitable as the
argument for the NVC_SetPriority() CMSIS function.
*****
NOTE: The NVIC allows you to assign the same priority level to multiple interrupts, so you
can have more ISRs than priority levels running as “kernel-unaware” or “kernel-aware”
interrupts.
*****
* The last value in the enumeration MAX_KERNEL_UNAWARE_CMSIS_PRI keeps track of
  the maximum priority used for a “kernel-unaware” interrupt.
* The compile-time assertion ensures that the “kernel-unaware” interrupt priorities do not
  overlap the “kernel-aware” interrupts, which start at QF_AWARE_ISR_CMSIS_PRI.
* The enumeration KernelAwareISRs lists the priority numbers for the “kernel-aware”
  interrupts.
* The “kernel-aware” interrupt priorities start with the QF_AWARE_ISR_CMSIS_PRI offset,
  which is provided in the qf_port.h header file.
* The last value in the enumeration MAX_KERNEL_AWARE_CMSIS_PRI keeps track of
  the maximum priority used for a “kernel-aware” interrupt.
* The compile-time assertion ensures that the “kernel-aware” interrupt priorities do not
  overlap the lowest priority level reserved for the PendSV exception.
The QF_onStartup() callback function is where you set up the interrupts.

* The CMIS function NVIC_SetPriorityGrouping() assigns all the priority bits to be preempt priority bits, leaving no priority bits as subpriority bits to preserve the direct relationship between the interrupt priorities and the ISR preemption rules. This is the default configuration out of reset for the ARM Cortex-M3/M4 cores, but it can be changed by some vendor-supplied startup code. To avoid any surprises, the call to NVIC_SetPriorityGrouping(0U) is recommended.

* The interrupt priorities fall all interrupts ("kernel-unaware” and “kernel-aware” alike) are set explicitly by calls to the CMSIS function NVIC_SetPriority().

* All used IRQ interrupts need to be explicitly enabled by calling the CMSIS function NVIC_EnableIRQ().
## Course Outline

- Enable QM to generate new type of SM code
- Provide “hooks” for customization (virtual functions)
- Multiple system clock tick rates
- Non-asserting event allocation and posting
- Kernel-Unaware Interrupts for ARM Cortex-M3/M4

**Quick Summary of Other Changes**

QP5 provides also a number of smaller features and enhancements
**QS/QSPY Updates**

**QS updates to generate new information**

- QS_QF_TICK, QS_QF_TIME_EVT_* records send tick rate byte
- New trace records for failed event allocation/posting: QS_QF_ACTIVE_POST_ATTEMPT, QS_MPOOL_GET_ATTEMPT
- New trace records for unit testing: QS_TEST_RUN, QS_TEST_FAIL

**Dictionary records generate less code by using functions**

- Added new source file qs_dict.c with QS_*_dict() functions

**More efficient QS code by grouping attributes into struct**

---

QS/QSPY software tracing required updating to accommodate the new features.

Also the size of the QS tracing instrumentation and its performance have been improved.
## Documentation/Examples

Added simple “Blinky” example requested by users
- For Windows, Linux, ARM Cortex-M with GNU, IAR, and Keil

Added “Getting Started with QP” guide in PDF
- Instead of replicating it in each example project

Updated Doxygen documentation
- Updated the code examples and missing references
- HTML documentation has the left-side summary panel and its own search box