QP / Qt
Actor Model for complex systems

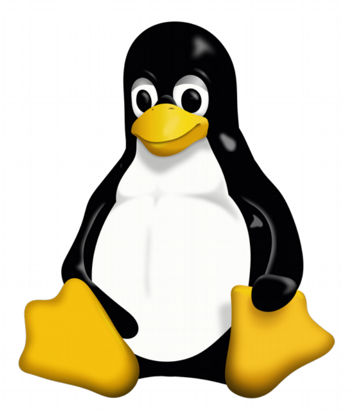
Alessio Lama

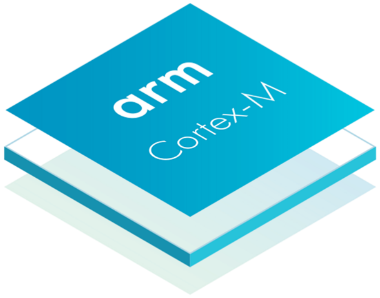
alessio.lama@yahoo.it
Linkedin

23 Maggio 2018

About me

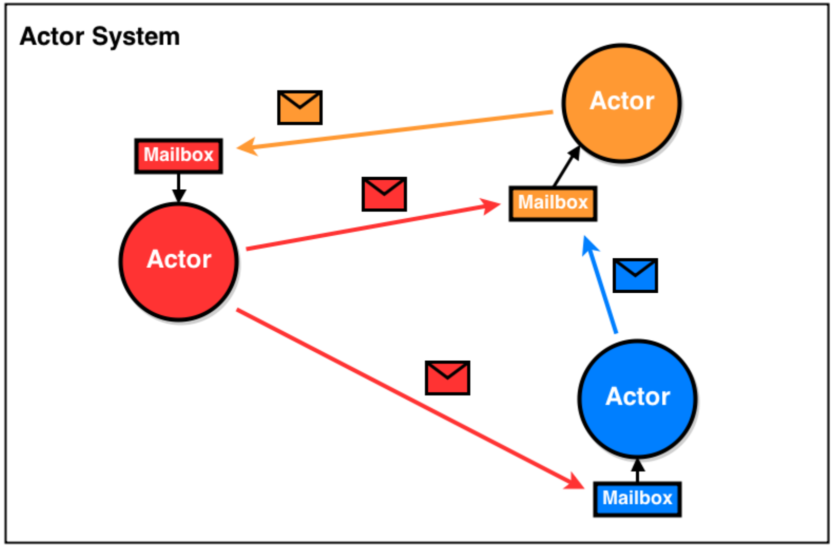
- Bare Metal
- Linux Kernel
- Qt Framework
- Safe Code
- Automotive
- Musical Algorithms





Actor Model

The actor model is a conceptual model to deal with concurrent computation.
It defines some general rules for how the system's components should behave and interact with each other.
The most famous language that uses this model is probably Erlang.



Active Object (Actor)

- Active Object (Actor) is an event-driven, strictly encapsulated software object running in its own thread and communicating asynchronously by means of events.

→ Not a real novelty. The concept known from 1970s, adapted to real-time in 1990s (ROOM actor), and from there into the UML (active class).

- The UML specification further proposes the UML variant of hierarchical state machines (UML statecharts) with which to model the behavior of event-driven active objects (active classes).

What is QP?

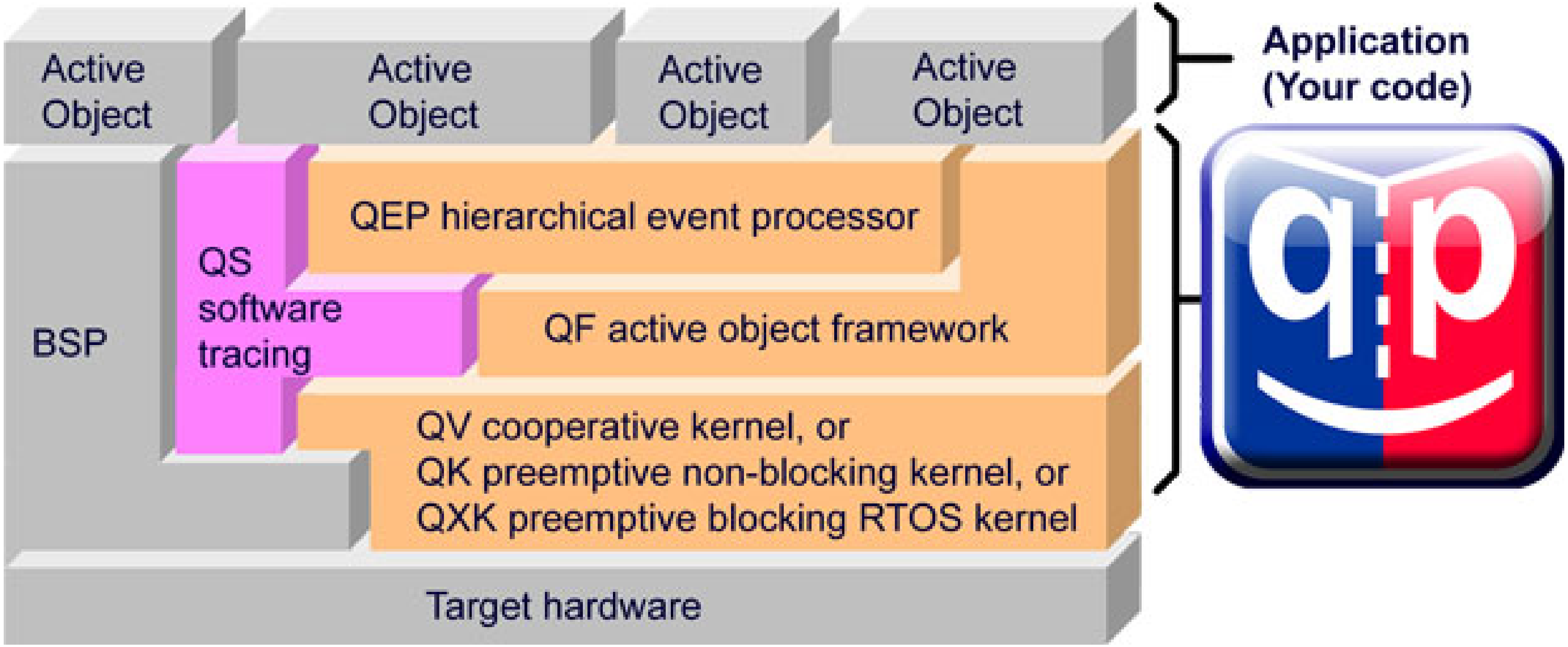
QP/C++™ (Quantum Platform in C++) is a lightweight, open source active object (actor) framework for building responsive and modular real-time embedded applications as systems of asynchronous event-driven active objects (actors).

The QP/C++™ framework is a member of a larger family consisting of QP/C++, QP/C, and QP-nano frameworks, which are all strictly quality controlled, thoroughly documented, and available under dual licensing model.

https://www.state-machine.com

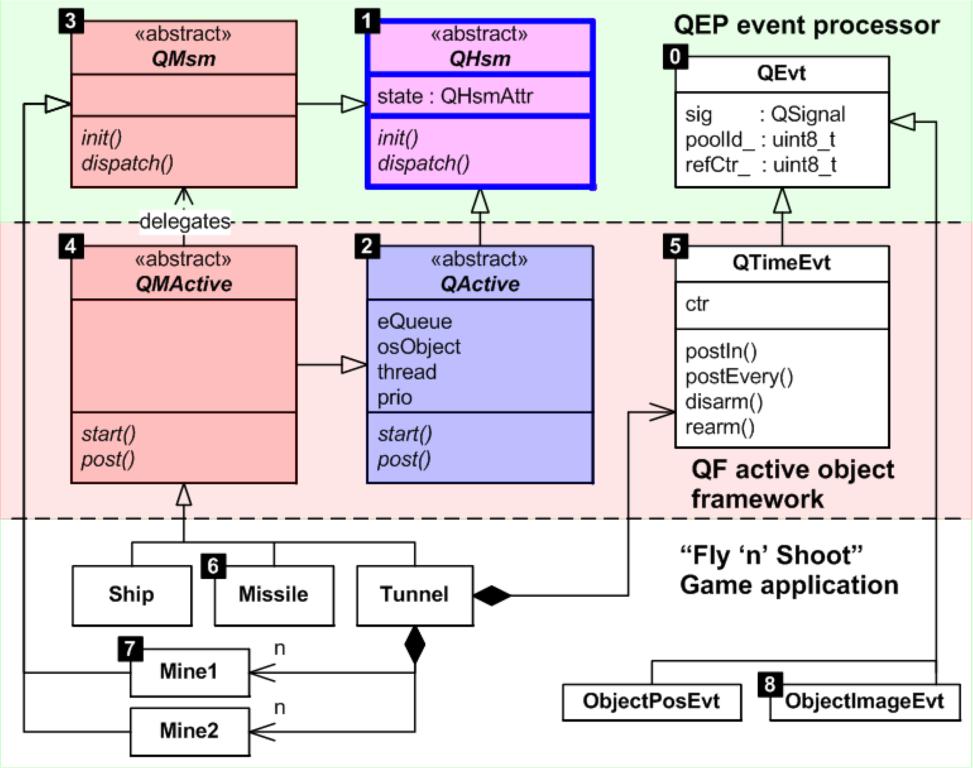
The behavior of active objects is specified in QP/C++ by means of hierarchical state machines (UML statecharts).
The framework supports manual coding of UML state machines in C++ as well as automatic code generation by means of the free QM™ modeling tool.

QP Architecture



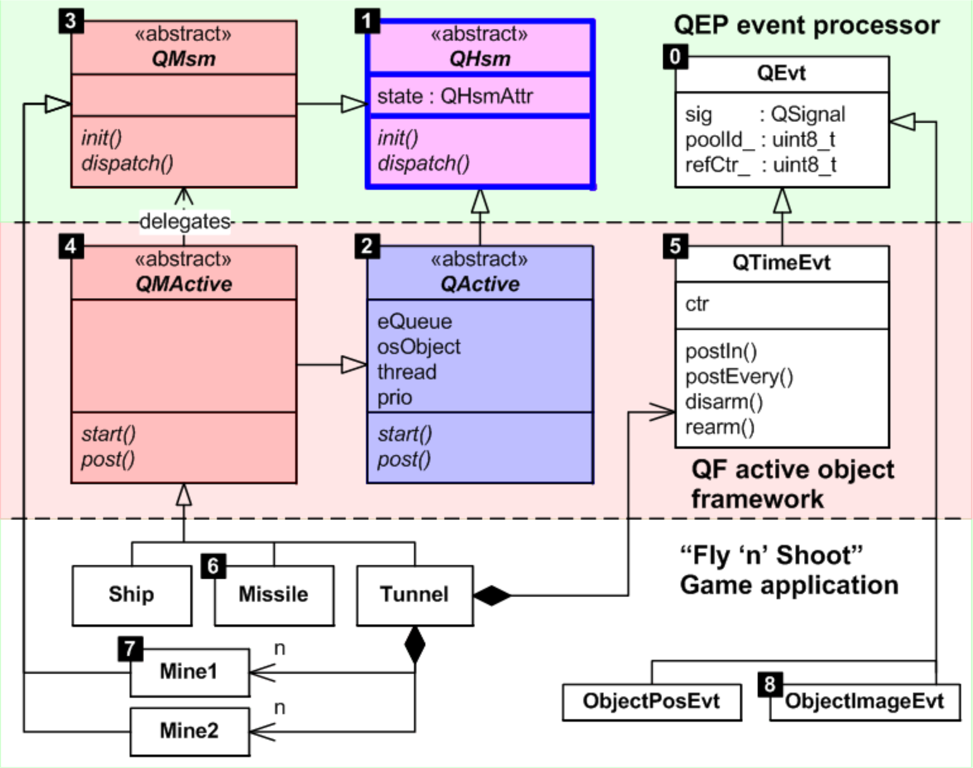
Active Object
Active Object
Active Object
Active Object
Application (Your code)
BSP
QS software tracing
QEP hierarchical event processor
QF active object framework
QV cooperative kernel, or
QK preemptive non-blocking kernel, or
QXK preemptive blocking RTOS kernel
Target hardware
q:p

QP Architecture



- **0.** The QP::QEvt class represents events without parameters and serves as the base class for derivation of time events and any events with parameters.
- **1.** The abstract QP::QHsm class represents a Hierarchical State Machine (HSM) with full support for hierarchical nesting of states, entry/exit actions, initial transitions, and transitions to history in any composite state.
- **2.** The abstract QP::QActive class represents an active object that uses the QP::QHsm style implementation strategy for state machines.
- **3.** The abstract QP::QMsm class (QM State Machine) derives from QP::QHsm and implements the fastest and the most efficient strategy for coding hierarchical state machines, but this strategy is not human-maintainable and requires the use of the QM modeling tool.

QP Architecture

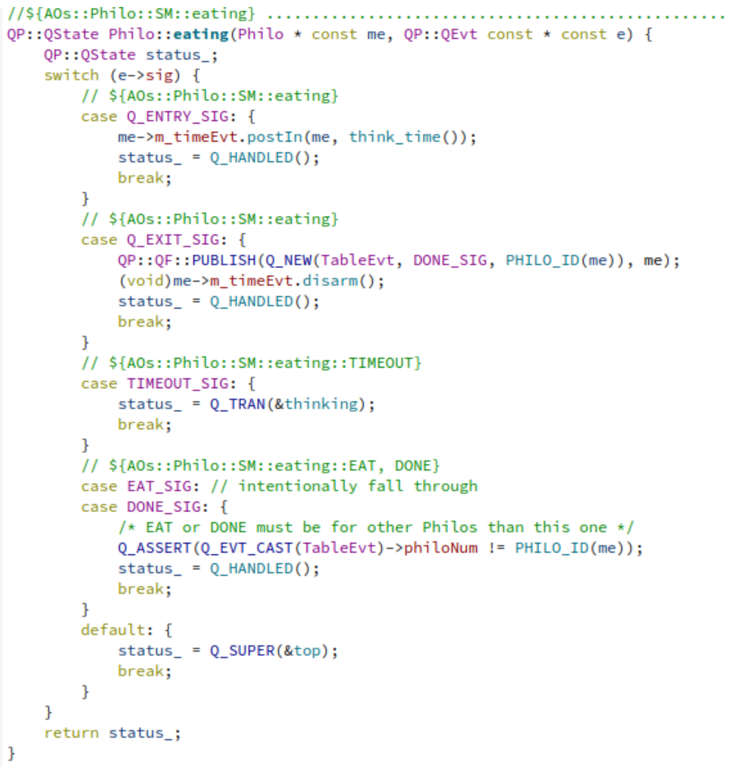


- **4.** The abstract QP::QMActive class represents an active object that uses the QP::QMsm state machine implementation strategy.
- **5.** The QP::QTimeEvt class represents time events in QP. Time events are special QP events equipped with the notion of time passage. The basic usage model of the time events is as follows. An active object allocates one or more QP::QTimeEvt objects (provides the storage for them). When the active object needs to arrange for a timeout, it arms one of its time events to fire either just once (one-shot) or periodically.
- **6.** Active Objects in the application derive either from the QP::QActive or QP::QMActive base class.
- **7.** Applications can also use classes derived directly from the QP::QHsm or QP::QMsm base classes to represent "raw" state machines that are not active objects, because they don't have event queue and execution thread.
- **8.** Application-level events with parameters derive from the QP::QEvt class.

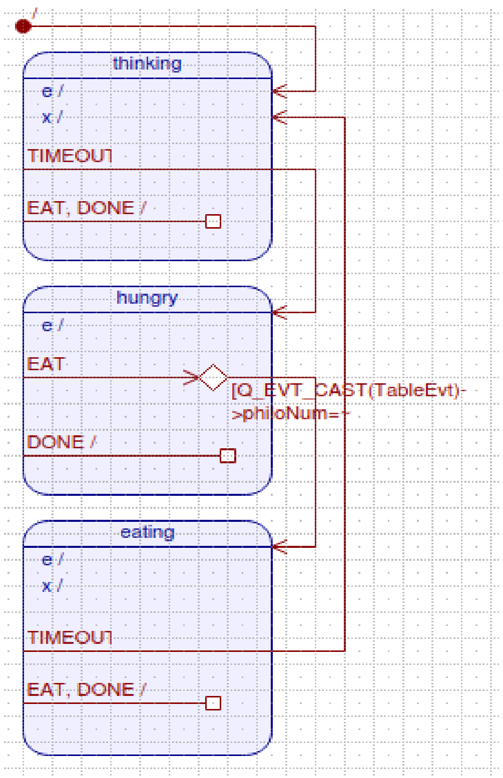
To provide ...

- To provide a reusable architecture based on active objects (actors), which is safer and easier to understand than "free-threading" with a traditional RTOS.
- To provide a simple-to-use coding techniques for hierarchical state machines, with which to implement the behavior of active objects.
- To provide efficient and thread-safe event-driven mechanisms for active objects to communicate, such as direct event passing and publish-subscribe.
- To provide event-driven timing services (time events).
- To provide a selection of built-in real-time kernels to run the QP applications, such as the cooperative QV kernel, the preemptive non-blocking QK kernel, and the preemptive blocking QXK kernel.
- To provide testing support for applications based on software tracing (Q-Spy).
- To provide portability layer and ready-to-use ports to 3rd-party RTOSes and desktop operating systems such as Linux and Windows.
- To provide a target for modeling and automatic code generation from the QM modeling tool.

QEP – Event Processing



```
//${AOs::Philo::SM::eating} ..................................................
QP::QState Philo::eating(Philo * const me, QP::QEvt const * const e) {
    QP::QState status_;
    switch (e->sig) {
        // ${AOs::Philo::SM::eating}
        case Q_ENTRY_SIG: {
            me->m_timeEvt.postIn(me, think_time());
            status_ = Q_HANDLED();
            break;
        }
        // ${AOs::Philo::SM::eating}
        case Q_EXIT_SIG: {
            QP::QF::PUBLISH(Q_NEW(TableEvt, DONE_SIG, PHILO_ID(me)), me);
            (void)me->m_timeEvt.disarm();
            status_ = Q_HANDLED();
            break;
        }
        // ${AOs::Philo::SM::eating::TIMEOUT}
        case TIMEOUT_SIG: {
            status_ = Q_TRAN(&thinking);
            break;
        }
        // ${AOs::Philo::SM::eating::EAT, DONE}
        case EAT_SIG: // intentionally fall through
        case DONE_SIG: {
            /* EAT or DONE must be for other Philos than this one */
            Q_ASSERT(Q_EVT_CAST(TableEvt)->philoNum != PHILO_ID(me));
            status_ = Q_HANDLED();
            break;
        }
        default: {
            status_ = Q_SUPER(&top);
            break;
        }
    }
    return status_;
}
```



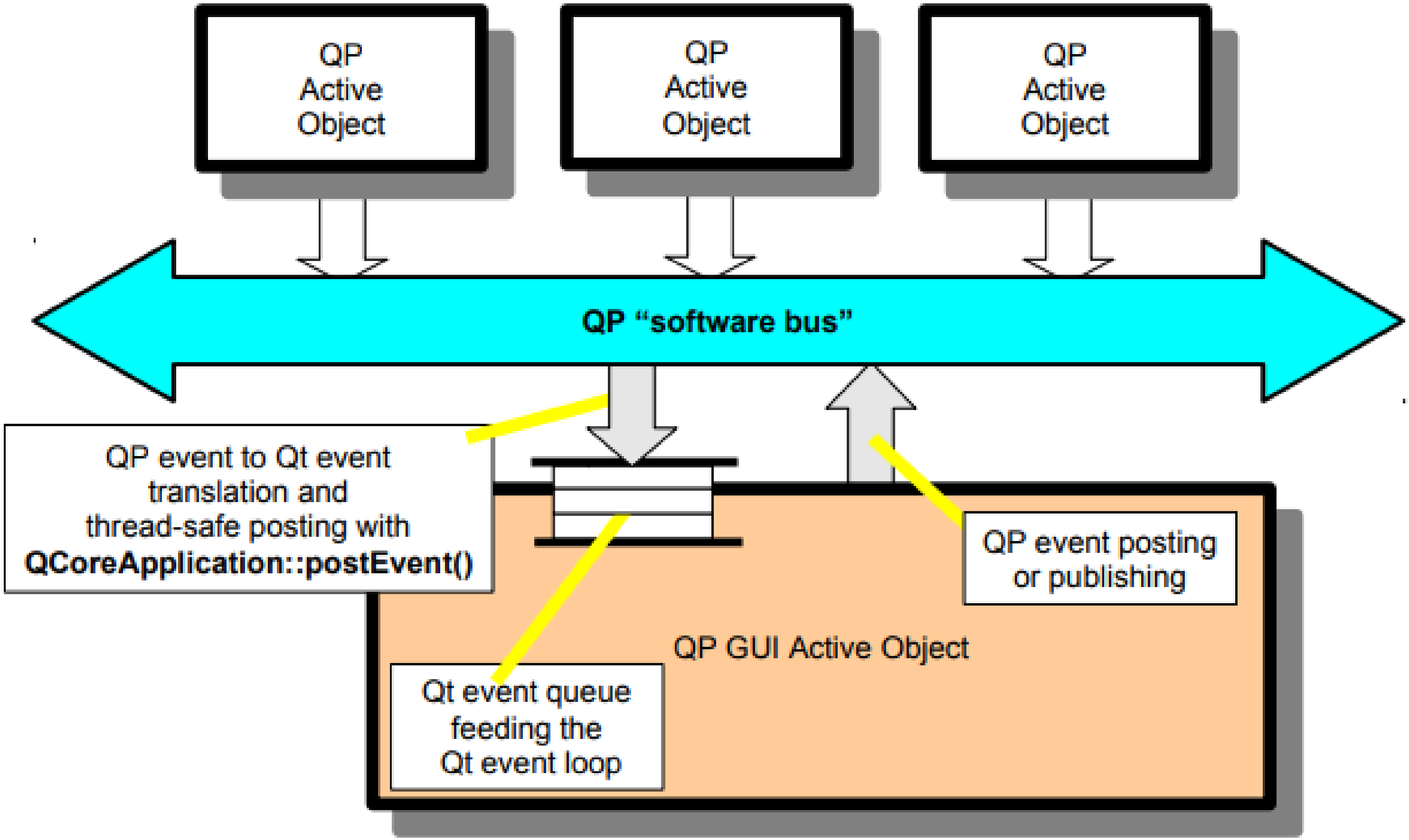
Target

- **ARM Cortex-M**
- **ARM Cortex-R**
- ARM7/ARM9
- **FreeRTOS**
- embOS
- **SYS/BIOS**
- ThreadX
- POSIX
- **Qt Framework**
- Win32 API

Qt/QP Integration

- First, you might use QP/C++ to build highly modular, well structured, multithreaded desktop or mobile Qt applications based on the concept of active objects (a.k.a. actors).
- Can be also useful for rapid prototyping (virtual prototyping), simulation, and testing of embedded software on the desktop, including building realistic user interfaces consisting of buttons, knobs, LEDs, dials, and LCD displays (both segmented and graphical).
- Moving embedded software development from an embedded target to the desktop eliminates the target system bottleneck and dramatically shortens the development time while improving the quality of the software.

QT Porting


QP Active Object
QP Active Object
QP Active Object
QP "software bus"
QP event to Qt event translation and thread-safe posting with **QCoreApplication::postEvent()**
QP event posting or publishing
QP GUI Active Object
Qt event queue feeding the Qt event loop