



4/30/2003 3:04 AM

“ After so many years of C++ oriented development all over the world, one would think that it would be impossible to come up with a significantly better implementation of a state machine in C++. It was profoundly refreshing (and surprising almost to the point of embarrassment) to see that your book contained a state machine implementation superior to anything I has seen before. Thank you!

I particularly like the way each state becomes its own method, the simple way hierarchical states are implemented, and the way entry/exit actions are generated automatically.

On the other hand, I have some constructive criticism on the other parts of the QF, if you are interested.

We all evaluate new information based on our previous experiences. One of the more significant experiences of my professional life as an embedded/real-time developer was a project where we were hired to develop the control system for a system of communication cable laying machinery. This was installed on a specially built ship, to lay fiber optic Internet cables on the sea floor.

The system had two cable-laying lines, as well as a 100-ton winch to tow a plough, which is used to bury the cable on the sea floor. The system had of thousands of I/O signals, many of which were analog. A large part of the cable-laying lines consisted of "linear cable laying engines" consisting of wheel pairs that could open and close hydraulically on the cable. Each wheel had its own motor with closed loop control. In fact everything, even the 100-ton winch, had closed loop control of both speed and tension. The system had both PC-based and more traditional "buttons and sticks" control consoles, duplicated in two locations, and could also be remotely controlled by the ship's control system.

The amount of I/O and the requirements for, and complexity of the control algorithms had forced our customer (the producer of the machinery) to abandon PLCs and start developing the system in C++.





Interestingly, the architecture (initially developed by the customer, later refined by us) had several things in common with the QF. It used communicating state machines to control everything, but the state machine was the more traditional double switch/case blocks.

This architecture had the same advantages as you describe in [Practical Statecharts in C/C++] Section 7.3.1: The conceptual integrity made a huge system that should have been difficult to design much simpler because the design of all classes followed the same basic architecture. If you were familiar with one part of the system, you could easily understand any other. This is very different from the more random collection of design patterns that most applications consist of today, where the internal architecture of each major module will be different and depend of the actual developer and his moods and ideas at the time.

The QF is superior to the architecture of our system in many ways, especially the state machine implementation. There are several other differences, and our system was superior in some ways. I will attempt to describe these here, not to boast (our architecture was raw and unrefined in many ways), but since they might serve as ideas for a future version of the QF:

1) Scalable lightweight scheduling:

The QF uses one thread per object, and each object must have a unique priority. It only allows a very limited number of active objects. Our system shared each thread among several objects, in much the same way as ROOM and Rational RealTime. Each thread had a list of objects and called a virtual method on all object in the list every time it was activated. This reduces context-switching overhead significantly in a system with a large number of active objects (ours had more than a hundred). The QF seems to restrict the system to a low number of objects, ours had no such restrictions. This restriction seems to arise from a desire to keep the data structure for the publish/subscribe system very small (your example is 200 bytes). This is not really relevant for a C++ system, which will typically have quite a bit of memory available. I would prefer to use a proper list of subscriber pointers to remove this scalability problem, even if it would cost more memory. The initial list could be short, and you could use the same reallocation trick as `std::vector` to grow each list as needed. This will not cause fragmentation problems, as it typically only happens during application initialization. We did not implement periodic execution using events. This was considered, but we





decided against it due to the amount of legacy customer code that would have to be changed. The QF approach using timers is the better in this respect.

2) Both publish/subscribe and point-to-point message passing, even to other computers: All messages were sent through a "Message Transport" object. The receiver was identified using a 32-bit object ID. The Message Transport had knowledge all objects in its address space, and would pass the message directly to the receiver a virtual "receive"-method. The receiver could even reside in a different address space, in which case the message would be passed on to the Message Transport object in this address space using UDP/IP. Publish/subscribe was implemented inside the superclass for state machine objects. Subclasses could publish "info" messages containing the publicly visible information about the object state. You could subscribe to this message by sending a "subscribe" message to the object. This would put the ID of the sender on the list of subscribers, and all subsequent info message would be passed to this ID using the Message Transport. It would have been easy to allow publish/subscribe for any type of message, but we never found this to be necessary. I found this superior to the publish/subscribe of the QF for the following reasons:

A) An active object that controls two active "sub-objects" will typically need to send command events to a specific sub-object. The QF publish/subscribe system will cause the message to be sent to all objects of the same type in the entire system, and would require that the message contains some sort of receive ID that the receivers can use to determine if the message is for them or not (like the Philosopher ID in your example). This is not very practical and will lead to an enormous amount of unnecessary message traffic in large systems. You can always post the message directly into the receiver's message queue, but that leads to more coupling between the objects.

B) We can subscribe to messages from a specific object, instead of a specific message ID from all objects that publish this message. This is a critical difference when you have a system with a large number of objects of the same class, often contained inside different classes for higher level components (a motor controller class can be used inside many different types of machines). The publish/subscribe used in QF (based only in message ID) would not have worked at all for our system. To use your dining philosopher implementation as an example: What happens when you have a restaurant with more than





one table of Dining Philosophers? Every table would receive events from all the philosophers in the restaurant...

C) The subscription setup did not require access to the publisher object, only knowledge of the object ID (even looser coupling than the QF).

D) Distribution of objects across address spaces and computers: This is seldom necessary, though).

However, I think neither the QF nor our approach to message passing is optimal. If I were to do a similar system in the future, I would probably opt for a simplified version of "ports" as used in the new real-time extensions for UML, but with "type-less" events like the QF. This could be used both for one-to-one and one-to-many (publish/subscribe) type communications by implementing different port types, and could probably also be extended to allow communication across address spaces. This also removes the need for a central data structure containing subscriber lists: Each one-to-many port can contain its own subscriber list. Don't get me wrong: I think a good publish/subscribe system is essential.

It should be noted that our system was fundamentally event driven, which makes it very suitable for an architecture based on state machines. Many other projects that I have worked on have been data driven. This can call for very different architectures, although not necessarily (data are also events). Choosing the architectural principles for any embedded application is the most critical design decision of all, and QF is very helpful since it provides a coherent and almost complete set of design principles for an architecture based on the state machine principle.

So the bottom line is that I am very much impressed by the HSM implementation. The QF is very interesting, but the event passing system and the scheduling system both have scalability problems and seem to need more work in order to be useful for large systems. It has the potential to be brilliant, though.

— **Helge Penne**
Senior Development Engineer
Data Respons AS, Norway

