Beyond the RTOS: A Better Way to Design Real-Time Embedded Software

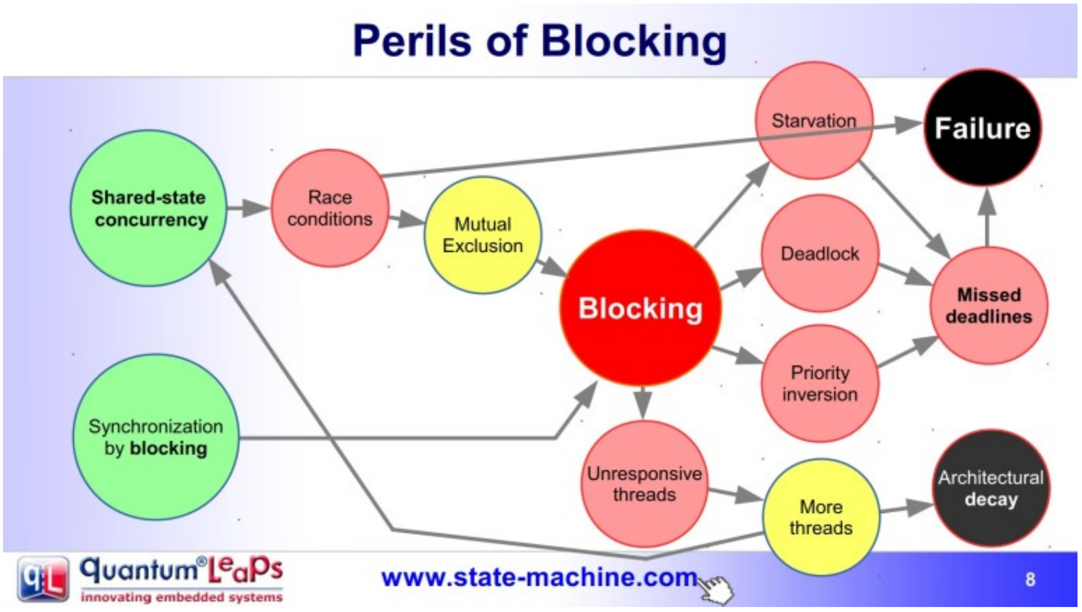
Wednesday, April 27th, 2016 by Miro Samek

An RTOS (Real-Time Operating System) is the most universally accepted way of designing and implementing embedded software. It is the most sought after component of any system that outgrows the venerable "superloop". But it is also the design strategy that implies a certain programming paradigm, which leads to particularly brittle designs that often work only by chance. I'm talking about *sequential programming* based on **blocking**.

Blocking occurs any time you wait explicitly in-line for something to happen. All RTOSes provide an assortment of blocking mechanisms, such as time-delays, semaphores, event-flags, mailboxes, message queues, and so on. Every RTOS task, structured as an endless loop, must use at least one such blocking mechanism, or else it will take all the CPU cycles. Typically, however, tasks block not in just one place in the endless loop, but in many places scattered throughout various functions called from the task routine. For example, in one part of the loop a task can block and wait for a semaphore that indicates the end of an ADC conversion. In other part of the loop, the same task might wait for an event flag indicating a button press, and so on.

This excessive blocking is insidious, because it appears to work initially, but almost always degenerates into a unmanageable mess. The problem is that while a task is blocked, the task is not doing any other work and is not responsive to other events. Such a task cannot be easily extended to handle new events, not just because the system is unresponsive, but mostly due to the fact that the whole structure of the code past the blocking call is designed to handle only the event that it was explicitly waiting for.

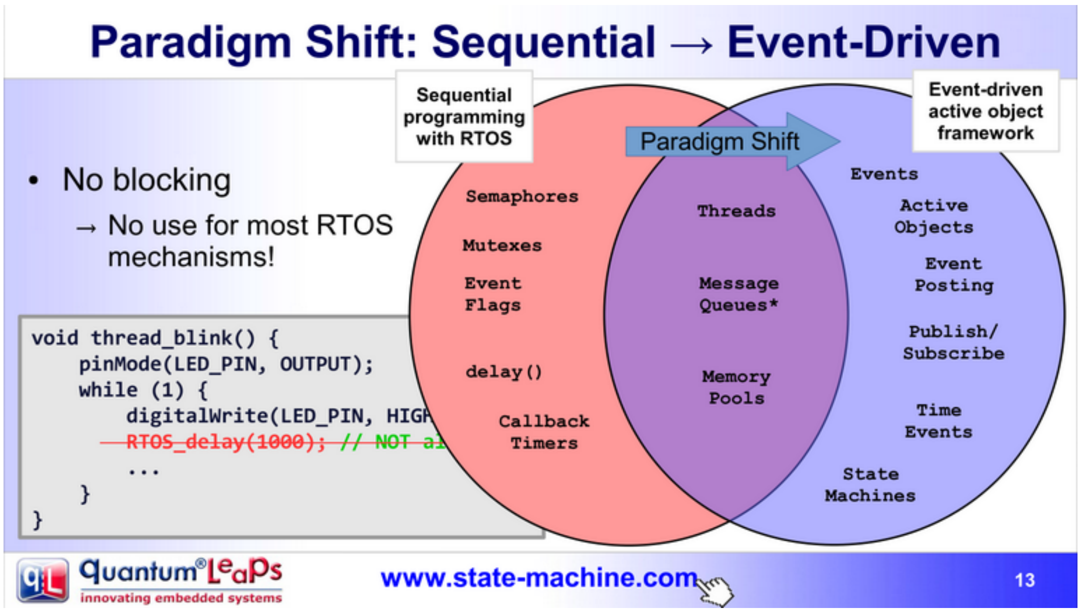
You might think that difficulty of adding new features (events and behaviors) to such designs is only important later, when the original software is maintained or reused for the next similar project. I disagree. Flexibility is vital from day one. Any application of nontrivial complexity is developed over time by gradually adding new events and behaviors. The inflexibility makes it exponentially harder to grow and elaborate an application, so the design quickly degenerates in the process known as architectural decay.



The mechanisms of architectural decay of RTOS-based applications are manifold, but perhaps the worst is the unnecessary proliferation of tasks. Designers, unable to add new events to unresponsive tasks are forced to create new tasks, regardless of coupling and cohesion. Often the new feature uses the same data and resources as an already existing feature (such features are called cohesive). But unresponsiveness forces you to add the new feature in a new task, which requires caution with sharing the common data. So mutexes and other such **blocking** mechanisms must be applied and the vicious cycle tightens. The designer ends up spending most of the time not on the feature at hand, but on managing subtle, intermittent, unintended side-effects.

For these reasons experienced software developers avoid blocking as much as possible. Instead, they use the **Active Object design pattern**. They structure their tasks in a particular way, as “message pumps”, with just **one** blocking call at the top of the task loop, which waits *generically* for all events that can flow to this particular task. Then, after this blocking call the code checks which event actually arrived, and based on the type of the event the appropriate event handler is called. The pivotal point is that these event handlers are **not allowed** to block, but must quickly return to the “message pump”. This is, of course, the event-driven paradigm applied on top of a traditional RTOS.

While you can implement Active Objects manually on top of a conventional RTOS, an even better way is to implement this pattern as a software **framework**, because a framework is the best known method to capture and reuse a software architecture. In fact, you can already see how such a framework already starts to emerge, because the “message pump” structure is identical for all tasks, so it can become part of the framework rather than being repeated in every application.


Paradigm Shift: Sequential → Event-Driven

- No blocking
  - → No use for most RTOS mechanisms!

```
void thread_blink() {
    pinMode(LED_PIN, OUTPUT);
    while (1) {
        digitalWrite(LED_PIN, HIG
        ~~RTOS_delay(1000); // NOT al~~
        ...
    }
}
```

This also illustrates the most important characteristics of a framework called **inversion of control**. When you use an RTOS, you write the main body of each task and you call the code from the RTOS, such as delay(). In contrast, when you use a framework, you reuse the architecture, such as the "message pump" here, and write the code that **it** calls. The inversion of control is very characteristic to all event-driven systems. It is the main reason for the architectural-reuse and enforcement of the best practices, as opposed to re-inventing them for each project at hand.

But there is more, much more to the Active Object framework. For example, a framework like this can also provide support for **state machines** (or better yet, **hierarchical state machines**), with which to implement the internal behavior of active objects. In fact, this is exactly how you are supposed to model the behavior in the UML (Unified Modeling Language).

As it turns out, active objects provide the sufficiently high-level of abstraction and the right level of abstraction to effectively apply **modeling**. This is in contrast to a traditional RTOS, which does not provide the right abstractions. You will not find threads, semaphores, or time delays in the standard UML. But you will find active objects, events, and hierarchical state machines.

An AO framework and a modeling tool beautifully complement each other. The framework benefits from a modeling tool to take full advantage of the very expressive graphical notation of state machines, which are the most constructive part of the UML.

In summary, RTOS and superloop aren't the only game in town. Actor frameworks, such as Akka, are becoming all the rage in enterprise computing, but active object frameworks are an even better fit for deeply embedded programming. After working with such frameworks for over 15 years , I believe that they represent a similar quantum leap of improvement over the RTOS, as the RTOS represents with respect to the "superloop".

If you'd like to learn more about active objects, I recently posted a presentation on SlideShare: Beyond the RTOS: A Better Way to Design Real-Time Embedded Software

Also, I recently ran into another good presentation about the same ideas. This time a NASA JPL veteran describes the best practices of "Managing Concurrency in Complex Embedded Systems". I would say, this is exactly active object model. So, it seems that it really is true that experts independently arrive at the same conclusions…

This entry was posted on Wednesday, April 27th, 2016 at 2:28 pm and is filed under event-driven programming, MCUs, modeling tool, RTOS Multithreading, state machines. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response, or trackback from your own site.

7 Responses to "Beyond the RTOS: A Better Way to Design Real-Time Embedded Software"

1. *Hugo Rodde* says:
   April 28, 2016 at 6:53 am

   Hi,
   Thank you for this interesting article.
   While reading through I noticed there is a tiny typo here: […] the fact the (that?) the whole structure of the code past the blocking call […].
   Cheers,
   Hugo

   Reply

   - *Miro Samek* says:
     May 12, 2016 at 9:28 am

     The typo has been corrected. Thanks.
     Miro

     Reply

2. *Leandro* says:
   May 10, 2016 at 9:31 am

   Thanks Miro for your dedication and commitment to put light on reactive programming. I share another article related with Active Object design pattern: http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf

   Regards,
   Leandro

   Reply
3. *Phil Matthews* says:
   May 24, 2016 at 12:24 am

   This hits the nail on the head.
   In my experience, using an RTOS always degenerates into an unmanageable mess.

   Reply
   - *Miro Samek* says:
     May 25, 2016 at 3:40 pm

     Unfortunately, given sufficient level of complexity, any method can degenerate to a Big Ball of Mud. And this includes the Active Object pattern as well. However, the traditional shared-state-concurrency with blocking threads is particularly difficult to comprehend by humans. We are just not good at juggling many things at once. Here is a good article that explains the problems with threads: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf . In a quick summary, this article argues that:

     "…Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism…"

     Reply
4. *Matt Chernosky* says:
   July 13, 2016 at 11:02 pm

   This is great!

   It's certainly easy to get yourself into trouble with an RTOS, especially when you try to get too fancy with it. Concurrency issues are always the most difficult to fix. By implementing active objects you greatly reduce the complexity of your application, and make it easier for you (and everybody) to comprehend.

   Another benefit of these "message pumps" is that they're easier to unit test. Typical embedded applications are very tightly coupled and it's difficult to find "seams" to test. With a bunch of loosely-coupled event processors, you can test each individually by feeding in events and verifying the expected behavior.

   Reply
5. *Produse Naturiste* says:
   November 9, 2016 at 8:14 am

   This is great!

   It's certainly easy to get yourself into trouble with an RTOS, especially when you try to get too fancy with it. Concurrency issues are always the most difficult to fix.

   Reply

Leave a Reply

Name (required)

Mail (will not be published) (required)

Website

Submit Comment

- ## State Space

  ### Miro Samek

  Miro Samek is the author of the book *Practical UML Statecharts in C and C++* and a regular speaker at the Embedded Systems Conferences. (full bio)

- ## Pages
  - Contact Miro

- ## Links
  - Embedded C Coding Standard
  - Embedded C Quiz
  - Embedded C++ Quiz
  - Embedded Software Training in a Box
  - Embedded Systems Articles
  - My Book
  - My Company

- ## Recent Posts
  - Beyond the RTOS: A Better Way to Design Real-Time Embedded Software
  - Fast, Deterministic, and Portable Counting Leading Zeros
  - Are We Shooting Ourselves in the Foot with Stack Overflow?

- ## Recent Comments
  - ExploringJS links | Epitomation on Fast, Deterministic, and Portable Counting Leading Zeros
  - Jim Moore on A nail for a fuse
  - Miro Samek on Object-based programming in C

- ## Tags

ARM Cortex-M ARM Cortex-M0 automatic code generation C CLZ code generation embedded books Embedded C embedded software development encapsulation event-driven. event-driven programming graphical tool hardware race condition inheritance modeling multitasking object-oriented programming predictions Productivity Programming techniques protothreads RTOS Software design state machines TDD Teaching UML Video YouTube

- ## Categories
  - agile software development (1)
  - Coding Standards (1)
  - Efficient C/C++ (8)
  - event-driven programming (7)
  - Firmware Bugs (4)
  - MCUs (7)
  - modeling tool (7)
  - productivity (2)
  - prototyping (1)
  - RTOS Multithreading (8)
  - software licensing (1)
  - state machines (9)
  - TDD (4)
  - UML (6)
  - Uncategorized (9)
- ## Archives
  - April 2016 (1)
  - September 2014 (1)
  - February 2014 (2)
  - April 2013 (1)
  - January 2013 (1)
  - December 2012 (1)
  - June 2012 (1)
  - May 2012 (1)
  - April 2012 (2)
  - February 2012 (2)
  - September 2011 (1)
  - August 2011 (1)
  - June 2011 (2)
  - February 2011 (1)
  - November 2010 (2)
  - April 2010 (2)
  - January 2010 (2)
  - November 2009 (1)
  - April 2009 (1)
  - March 2009 (1)
  - January 2009 (1)
  - June 2008 (1)
  - January 2008 (1)
  - September 2007 (1)
  - June 2007 (1)
  - September 2006 (2)