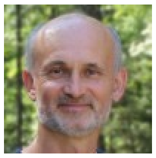

Miro Samek

Founder and CEO, Quantum Leaps, LLC

View full profile

# Is an RTOS really the best way to design embedded systems?

**Miro Samek** posted in **Real-Time Embedded Engineering**

An RTOS 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 "supuerloop". 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 blocking.

Blocking occurs any time you wait explicitly in-line for something to happen. All RTOSes provide an assortment of blocking mechanisms, such as various 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 evil, because it appears to work initially, but almost always degenerates into an 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 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, hairy, unintended side-effects.

For these reasons experienced RTOS users avoid blocking as much as possible. They structure their tasks in a particular way, typically 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 task loop. This is, of course, the event-driven paradigm applied on top of a traditional RTOS.

But there is more, much more to using an RTOS safely. A traditional preemptive RTOS is really quite a *low-level* mechanism that can be used in many ways. Yet, I don't see enough information, online, in the literature, or otherwise, which would explain how to use an RTOS safely, what to avoid, and how to move beyond the RTOS.

I am looking forward to an interesting discussion of these critical questions.

- 👍 21
- 657 comments

## Reactions

Most relevant

Douglas B. 2ndSenior Firmware Engineer at Integrity Insitu

I don't use either RTOS or super loops. (I assume that by super loop you're talking about a loop full of semaphores which are triggered via interrupt.)

Early in my career I realized that super loops resulted in tight coupling between firmware modules, and a relative inflexibility, and inability to deal with different interrupt rates (some are frequent, and some are not), order of event processing was random, etc.

So what I came up with was much more elegant. (Excuse the pseudo code.)
This code lets me completely decouple all layers in my code. Order is preserved, and variable interrupt rates are accommodated.

```
Array *Function(INT32)[100];
Array Parameter[100];
INT8 Read;
INT8 Write;
INT8 Depth;

// Main loop
void MainLoop( void )
{
    while(forever) {
        if(Depth) {
           INTERRUPT BLOCK
           Depth-- ;
           INTERRUPT UNBLOCK
           Function(Parameter[Read])[Read];
           Circular Increment Read Pointer;
        }
        else
           IDLE ;
    }
}

void EventPostinMainContext(*PostedFunction, PosterParamter)
{
    Function[Write]=PostedFunction;
    Parameter[Write]=PostedParameter;
    INTERRUPT BLOCK
    Depth++ ;
    Circular Increment Write Pointer
    INTERRUPT UNBLOCK
}
```

There is of course an interrupt context version for posting which of course doesn't need to block interrupts.

The 'main' on start up simply calls the Init function for each module or driver in the system. This is order dependent of course. I do a lot of dynamic connection of modules to one another via call backs which are always configured with pointers to functions.

So A UART receive interrupt will be something like;

```
Interrupt UARTRX( void )
{
    EventPostinInterruptContext(Callback, *RxRegister)
}
```

How you hook up to it;

```
void UARTSetCallback( *Function )
{
    Callback=Function
}
```

2

Miro SamekAuthorFounder and CEO, Quantum Leaps, LLC

I still have a feeling that most of the participants in this discussion believe that superloop and RTOS are the only games in town, so if not RTOS than I must necessarily be advocating the do-it-all-yourself superloop.

That's completely the opposite what I'm trying to say here.

I'm trying to show a way to move **beyond** the traditional RTOS, not backwards to the superloop.

A better game than a blocking RTOS is coming to town and it is called "active objects" or "actors". I would highly recommend two articles by Herb Sutter, which describe the problems with traditional threads and how to address them with active objects:

1. "Use Threads Correctly = Isolation + Asynchronous Messages" at [http://www.drdobbs.com/parallel/use-threads-correctly-isolation-asynch/215900465| leo://plh/http%3A*3*3www%2Edrdobbs%2Ecom*3parallel*3use-threads-correctly-isolation-asynch*3215900465/nA50?_t=tracking_disc]

2. "Prefer Using Active Objects Instead of Naked Threads" at [http://www.drdobbs.com/parallel/prefer-using-active-objects-instead-of-n/225700095| leo://plh/http%3A*3*3www%2Edrdobbs%2Ecom*3parallel*3prefer-using-active-objects-instead-of-n*3225700095/q0wa?_t=tracking_disc]

Here is what Herb says in the second of these articles:

"Using raw threads directly is trouble for a number of reasons ...
Active objects dramatically improve our ability to reason about our thread's code and operation by giving us higher-level abstractions and idioms that raise the semantic level of our program and let us express our intent more directly. As with all good patterns, we also get better vocabulary to talk about our design. Note that active objects aren't a

novelty: UML and various libraries have provided support for active classes"

So, all this is really not new. But what's perhaps less known, especially in the embedded systems community, is that active objects are not only fully applicable to the embedded systems, but they are actually *lighter* than a traditional RTOS.

For example, my DDJ article "UML Statecharts at $10.99" ([http://www.drdobbs.com/architecture-and-design/uml-statecharts-at-1099/188101799|leo://plh/http%3A*3*3www%2Edrdobbs%2Ecom*3architecture-and-design*3uml-statecharts-at-1099*3188101799/SD7l?_t=tracking_disc]) describes an implementation of an active object framework (called QP-nano) running a non-trivial application in on the C8051F300 microcontroller (with 256 bytes of RAM, 8KB of Flash, 11 pins). No RTOS can fit there!

👍 5

Don Pieronek 1stOwner at Real Time Objects & Systems, LLC

@Bush: I agree. RTOS is great for big platforms and lots of programmers since it provides a level of convergence and isolates programmers to some extent. In small stuff like we do, no need. Relative to screen updates just finishing porting our display driver from one vendors display to another vendors and we update very fast compared to the time you reference, of course we are updating a 128 x 160 pixel display with 16 bit colors and not a big one. Yes "al" the applications remain the same except for the display driver. And why is it done? Reduce cost signfiicantly. So if I used an OS, gee do they have a driver for the new display? Unlikely. Do they maintain a common interface to the different display they may support? Perhaps. Would they have a driver for the cpu we are using, the pinouts we are using, the memory footprint we require? Unlikely. The product supports a CAN network, keypad, color display and a LOT of screens, variable editing and all that stuff. We have 49K bytes of code space, 3328 bytes of RAM. We convert caracters to color pixels, on the fly, one character at a time, and update "very quickly". But again, we are not using a standard platform where the manufactured cost of this unit is "very low" compared to any user interface I know of, else I would be using one. So I agree with you. On big stuff, use a generic OS, but in my world it would not work out except of course I do use a PC for manufacturing testers, configuration tools, object tester and so on and do have to often "fight the OS", available drivers and so on. The good news is, in my case, the tools are not the product as is the case with many on this dialog. It all comes down to requirements, time, cost and we are too low of the food chain. However, that said, our technology is often considered an OS where the drivers often use the same objects as applications, as we don't differentiate between the two, which is also very memory efficient. So in reality, we sell "an OS" which happens to be included with the applications and we always start a product "porting" the objects and core applications to the new CPU, new pinouts, changing some drivers for the new cpu, while dragging and droping existing "common applications" to test the hardware, then start adding functionality. So I agree with core code reuse, just can't use a generic OS. Hope this helps.

👍 1

Douglas B. 2ndSenior Firmware Engineer at Integrity Insitu

I don't even follow the train of thought in that argument. (And to be clear.. I'm anti RTOS.)

You can't measure an OS just by its real time performance. No one would buy computers then, right?

I feel that the best applications where an RTOS can help are ones where lots of programmers are involved, or you get a lot of useful device drivers and applications.

When you have a gaggle of programmers you can't track CPU loading that well, so an RTOS will make everyone's life easier, except for the device driver programmer who must finely craft his work to operate outside the OS.

When you get useful device drivers and applications an RTOS speeds up your development.

In practice, I prefer to work on smaller microcontrollers, and I use an OS I rolled on my own. (Run to completion, non-context switching.) This runs with a single AVR ATMega128; [http://www.galvanic.com/pdf/h2s-analyzer-903.pdf|leo://plh/http%3A*3*3www%2Egalvanic%2Ecom*3pdf*3h2s-analyzer-903%2Epdf/KWi_?_t=tracking_disc]

Don Pieronek: It was quite hard getting the graphic display (pixel oriented) to work and update in a reasonable time frame. It updates twice a second but restarts the update every a key is pressed so it feels natural.

So... Its say that the answer is that an RTOS makes sense if its required.

Miro SamekAuthorFounder and CEO, Quantum Leaps, LLC

@Frank Dever: I don't really follow all of your rants, but most of this sounds either backwards or completely beside the point. But your first rant really begs clarification.

Object oriented design does not encapsulate anything in terms of concurrency hazards. As long as objects are accessed by synchronous function calls, potentially from different threads, you have the same race conditions caused by data sharing (the objects in this case).

To achieve real encapsulation for concurrency, you need to take object orientation to the next level called "active objects" or "actors". Here I highly recommend reading the article "Prefer Using Active Objects Instead of Naked Threads" by Herb Sutter ([http://www.drdobbs.com/parallel/prefer-using-active-objects-instead-of-n/225700095|leo://plh/http%3A*3*3www%2Edrdobbs%2Ecom*3parallel*3prefer-using-active-objects-instead-of-n*3225700095/q0wa?_t=tracking_disc]).

👍 3

Frank Dever 2ndSoftware Engineering Contractor at FirstPass Engineering

@Miro Samek and @Don Pieronek: I really can't agree with dismissing an RTOS in favor of a superloop. At the overhead of less than 4% of the CPU in typical applications, an RTOS allows us to *decouple* tasks in the *time domain*, which is a huge benefit well worth 4% of the CPU.
----------
"decoupling" is just another word for object oriented design. Modular design in which the objects that you create are domains unto themselves are "decoupled" - the concepts of data hiding, abstraction through interface contracts, and so on, provide efficient access to all of the elements you need to control. Try to think of an object as a "task" if that helps. No need for the 4% CPU nor the cost of an OS... people have mentioned deadlocks, priority inversion and lack of deterministic behavior.. why? Because the same people that find object oriented execution hurts their head, aren't good at design to begin with, and are looking for an easy way out of cvomplications. Object oriented design is that "easy"' way out.

This means that higher priority tasks are practically insensitive to changes in the tasks of lower priority.
----------
in a system using interrupts to meet hard real-time requirements, priority and multitasking is an artificial construct, created for those who have trouble with the concept or those whose hardware just barely has the cycles to accomplish the task at hand. Sending messages around, so that people can wait to get them synchronizesd tasks, but its still an interrupt. The desiire to interleave instructions in order to allow multiple tasks to complete at the same time is fretting over a fish without a gun.

The main problem is something takes too long. Not that something happens too fast. No one wants to write a state machine when you can just lollygag along with context switches controlled by a pre-emptgive timer - who cares how long it takes to print a line, to a console, or calculate the fourier transform of data collected over 24 hours?. No states needed. Draw it out as long as you want, and you never have to worry about where you were last time - you just keep on keeping on. Breaking up one of these procedural monsters is hard!

Work smarter, not harder!

For a GPRS stack, ok, mutliple layers, shared resources, mutliple interfaces one processor... 70 engineers who sit down and start writing code in their own private box to match their interface, knowing the details of the design was worked out by ivory tower architects may justify using an RTOS.

I mean, who could imagine a team that large working together to design an object per layer with interrupts, an interface that reports, can be queried, manages its own private data and presents that interface to a scheduler?

Not me.I can't drive to the store without a semaphore and a priority scheme designed by a PHD..

In contrast, every change to a superloop has immediate impact on the timing of all activities. This is fragile and it is not the way of building robust systems.
---------
who told you that? Mitt Romney? LOL!

Look, I know people like to drive sports cars on roads posted 35mph. At least they look faster than everyone else.

I'm not dismissing an RTOS in favor of a superloop. I'm just poking fun at the investment bankers who used middle America's retirement money to pay out 4 million dollar bonuses and secured tax breaks to compensate the "best and the brightest" for bankrupting America, and to further incentivise them to create jobs. And all of this so that I won't be forced to stop praying and have an abortion against my will.

It's all right there in the fine print.

**Don Pieronek 1stOwner at Real Time Objects & Systems, LLC**

@Miro: I have no problem with people using an OS when they can, when they have enough memory and so on. I have used many an OS in my time and understand when I use one, and why I used one. My issues are never % of cpu related, mine in response time to an interrupt and the rate the interrupts occur, where often many different interrupts are occuring at the same time and taking actions from data acquired in an interrupt in the required time is obviously critical. Add up the numbers for various scenarios and I must "keep up". If I had something like the OS kernel that could run to do things like looking for the next highest priority task ready to run, where it may disable interrupts for a "short" period of time, my system would not keep up. If you look into the details of whatever OS you are using, you will likely be able to find the various timing values. In many devices I can not miss an event. I did a pretty thorough investigation in the real time OS's available when I was at RA to see if we could "standardize" on one, or maybe two. At that time, the duration the kernels disabled interrupts to do "house keeping" varied with the various OS's, where one OS was OK for some types of products, but not others. Bottom line is, we could not standardize at that time, and obviously standarizatrion could only be considered for devices with sufficient code space and hardware resources to support the OS. We had some products at that time that had 4k of code space whether you needed it or not! Today obviously most products I work on are in the 32K range, and when luck, maybe 144K. I could go on, but do what you need to do to "ship it on time and under budget".

1

Miro SamekAuthorFounder and CEO, Quantum Leaps, LLC

@Frank Dever and @Don Pieronek: I really can't agree with dismissing an RTOS in favor of a superloop. At the overhead of less than 4% of the CPU in typical applications, an RTOS allows us to *decouple* tasks in the *time domain*, which is a huge benefit well worth 4% of the CPU. This means that higher priority tasks are practically insensitive to changes in the tasks of lower priority. In contrast, every change to a superloop has immediate impact on the timing of all activities. This is fragile and it is not the way of building robust systems.

The problem I try to address in this thread is different, and is related to *how* we use the RTOS. Perhaps the best explanation of the perils of using "naked threads" is the article "Prefer Using Active Objects Instead of Naked Threads" by Herb Sutter ([http://www.drdobbs.com/parallel/prefer-using-active-objects-instead-of-n/225700095|leo://plh/http%3A*3*3www%2Edrdobbs%2Ecom*3parallel*3prefer-using-active-objects-instead-of-n*3225700095/q0wa?_t=tracking_disc]). This is exactly what I've been talking about here for many posts.

Harlan Rosenthal 2ndSenior Design Engineer at ASCO Power Technologies

I'm confused. My last RTOS-based project was, as far as I understand it, event-driven. The object is a task, events are announced to the task through messages, and all of this happens using the classic message-queue and event-flag paradigm.

I submit that sequential problems are *typical* if you partition the functionality properly. I further submit that the less-skilled problem I see most is unnecessary coupling of data and/or decisions in ways that interfere with such decomposition. The comm process should not stop after each packet to parse it; that's some other process's job, because the comm process should be doing nothing but accumulating data and transmitting data, which is a purely sequential task (and if full duplex two separate sequential tasks for the receive and transmit sides). If you break the problem down far enough that each part is a nice little gear, they fit together and make clockwork; if one of your gears is a Mobius strip painted by Escher, you missed something.

Dan George 1stFlight Software Engineer at Anduril Industries

Miro's opinion that the RTOS api (blocking constructs, in specific) is insidious or even evil does not mean he thinks people who use the api are insidious or evil (right, Miro?). I think the issue is that implementing an object-oriented, event-driven design isn't as easy as it should be if all you have to work with is the typical RTOS api. It is no wonder because the RTOS api pre-dates OO, event-driven (can't use the acronym for that one). I hope we can get some objective discussion about improvements and get away from beating the dead horse of whether improvements are needed. If no improvements are needed then the discussion will soon die a natural death.

Dan George 1stFlight Software Engineer at Anduril Industries

@Julio and Harlan, you are distracted, like many others, with attention-getting aspect of the original post. In the context of the overall discussion, you argument is that there are no pitfalls with the RTOS api but it is qualified with a whopping assumption: "as long as you use it properly." I'd wager most people on this thread can use the api properly but some find it takes more effort than they'd like to expend. It is not "evil" to properly use the API but it can seem evil when people you depend upon don't use it properly.

@All,
General pattern for RT:
1. Partition the events into small enough subsets so that it is possible to make sure those events are always handled
2. Minimize resource sharing between the activities involved in the partitions so that it possible to predict how the response to events of one partition will be affected by other partitions
3. Make resource sharing very visible so that it is easier to analyse and validate the timing
4. Organize the activities such that it is possible to validate the sequence of actions is correct.

Activities = one or more actions.
Actions = blocks of exclusive access to shared resource (cpu, memory, ...).

There are many thousands of systems that demonstrate proper use of the RTOS api is a solution to the above. No need to debate that. Anyone with experience working in a team of more than five knows that not everyone has the skills to use the RTOS properly. Most people accept that some systems will have to be developed by teams of more than five. Some of those people see the pitfalls of RTOS api as an inhibitor, if not flat out "evil" (whatever that means). Is there a better way?

1

Miro SamekAuthorFounder and CEO, Quantum Leaps, LLC

I'm really glad that in the last couple of posts this discussion really touches the very core of the problem.

Both @Harlan Rosenthal and @Julio Diez Ruiz provide examples of specific event *sequences* that have to happen in a logical order. For instance, in Harlan's example dumping data to a log file is followed by an event signaling the end of the file operation. Such a pre-determined cause-effect event sequence is easy to code sequentially. And I absolutely agree. The sequential solution to a sequential problem is always the most natural and the simplest possible. Any event-driven solution with or without state machines won't be simpler in this case. So, we don't need to debate this.

But my whole point is that sequential problems are rare. I just observe that most our problems require the ability of handling events that arrive in arbitrary sequence and timing. In other words, it is relatively rare that we can control which events will arrive into our system (as in @Harlan's example, a write operation to a file causes the file-operation event). Most of the time events are just thrown at us and we need to deal with whatever sequence they happen to be.

And for this type of problems, sequential code with blocking to wait on events is very quickly becoming unmanageable. This is a very well know fact, and that's why all GUI systems have *not* been programmed sequentially, but instead have been *event-driven* for over three decades now.

So, now the big question for me has always been: Why event-driven programming has not caught on in embedded systems (which are event-driven by nature) to the same degree as it has in GUI programming?

I think that the reason is that sequential and event-driven paradigms really don't mix. They don't mix because sequential programming is all about *blocking*, while event-driven programming is specifically about avoiding any blocking.

## Miro SamekAuthorFounder and CEO, Quantum Leaps, LLC

Incidentally, a system of cooperating state machines is also very easy to execute under a cooperative scheduler, which can be remarkably simple (20 lines of C, literally). The simplicity is due to implicit yielding of state machines to each other after processing of every event. The task-level response of such a system is the longest event processing time of any state machine (the longest RTC step in the system), but because event processing is essentially linear code without blocking or polling, the task-level response is quite often adequate. If not, it is often possible to break up long event processing into shorter pieces (e.g., iteration can be done one step at a time). So even here, event-driven systems are superior to work with than sequential code.

## Miro SamekAuthorFounder and CEO, Quantum Leaps, LLC

Just to clarify, my original post was never intended to criticize preemptive multitasking, which the term "RTOS" is almost synonymous with, at least in the mind of many embedded developers.

In fact, I don't agree with many comments, in which the authors downplay the importance of preemptive multitasking. I think that this concept is invaluable and is only going to be even more important in the future, because it *decouples* tasks in the time domain. Preemptive multitasking makes algorithms like RMA possible, and this is absolutely remarkable that we can mathematically prove schedulability of our systems.

However, and this is critical point of this discussion, we need to clearly distinguish task preemption from task *blocking*. These two concepts are really completely different. Tasks can preempt each other, just like ISRs in a prioritized interrupt controller can preempt each other. (For example, consider interrupts on ARM Cortex-M with the NVIC prioritized interrupt controller). But, I hope everyone here knows that ISRs cannot block. So here you have an example of preemption without blocking.

Unfortunately, a task in a conventional RTOS can only wait for events by blocking. And this, as I've been arguing here for a year now, is almost never a good idea, because a blocked task is unresponsive.

In contrast, an event-driven sytsem with multiple, cooperating state machines can execute the state machines under a fully preemptive scheduler. So, such a system is fully compatible with the RMA assumptions. However, state machines don't need to block (they really *can't* block), so they remain responsive. Such a system is incomparably more extensible than a conventional RTOS.

👍 1

### Dan George 1stFlight Software Engineer at Anduril Industries

@Miro, can you say more about the alternatives to an event detector thread? I understand the down sides to of the approach but I'm curious about solutions besides polling.

### Miro SamekAuthorFounder and CEO, Quantum Leaps, LLC

I'm absolutely not convinced to the conforming opinion that "both sequential blocking code and event-driven code have their place".

I really fail to see where the sequential blocking code would be any better than an event-driven solution or even a situation where blocking would not cause big structural problems down the road.

Please note that the sequential and event-driven programming paradigms don't mix easily, so you need to pick one or the other. (Actually, the only way I know to mix the two, is to use the simple "event detector" tasks" described by @Dan George in his earlier post. But that's an expensive and not very elegant solution compared to purely event-driven system.)

Having tried both traditional RTOSes and event-driven frameworks for many years, I choose event-driven programming paradigm every time. I would never want to go back to a raw RTOS.

**Julio Diez Ruiz 3rd+Strategic Cloud Engineer at Google**

@Miro.
Please, don't make it seems as if I were defending something I haven't said.

When answering @Dan George I stated "...to play fair and be useful to this discussion I precisely stick to protocol examples because I think they [...] are well suited to be implemented as state machines... ", so I think my opinion is quite clear. I'm not hanging on sequential model nor saying in any way to convert a protocol specification into sequential code, but giving a specific example of possible sequential code inside an event-driven framework, as you asked for. I also stated " @Dan. "the frequency at which I can take advantage of the natural sequential style is just too low". I'm not claiming the opposite ... blocking is useful in some occasions, not in a lot of but...".

I also said that the system as a whole has to manage external events and there are different ways to do it, with more or less support from the protocol implementation. Indeed if you need an 'urgent' abort, even a 'pure' event-driven implementation could be not enough responsive at times because it is still processing previous event.

So, because that's my example, I will take your words "If in your protocol you can clearly identify the only valid sequence of events A, B, C... then the sequential solution [...] is the simplest and most natural." as a timid "Well, I agree on that case".

In following posts I will continue looking for specific examples of blocking calls sneaking into event-driven frameworks ;)

Miro SamekAuthorFounder and CEO, Quantum Leaps, LLC

@Dan George: Yes, I think you side-stepped the issue completely. Apparently, you already use an event-driven framework and you are very careful to encapsulate any blocking call in a simple "event detector" task, which converts blocking into posting events to the framework. So, even though it is an inefficient solution, you don't experience the "evil blocking" that I'm talking about.

But this is absolutely *not* the conventional way of using a conventional blocking RTOS. I think that most RTOS users think that they are supposed to, well..., use the RTOS "as-is" rather than carefully encapsulate and contain every blocking call. And this is not only the users. Any documentation, manuals, app-notes, and books provided by the RTOS vendors also recommend using semaphores, event-flags, message mailboxes, message queues, etc. *directly* in the application code, without converting such calls into event objects and definitely without using any event-driven framework.

I think that comments, like the last comment by @Dan George, make this discussion especially hard and confusing to follow for people less familiar with event-driven programming (which I assume is the most of embedded engineers.) Dan does not see any problems, because he already uses the event-driven paradigm.

But most other people don't see any problem with blocking, because they exactly *don't* know any better way. This is obviously a completely different reason.

@Julio Diez: in your earlier post you described a communication protocol, in which at some point "you have to wait for a response or fail by timeout and no other events have to be considered". In my experience, there are always some "other events that have to be considered". Sooner or later, for million of reasons, you have to be able to add some "other event". What do you do then? Your blocking implementation is an example of violating the "Open-Closed" principle of the S-O-L-I-D design that I've mentioned in my earlier post. The system based on blocking can't be extended (by adding events) without re-designing, re-implementing, and re-testing most of it. Such system is brittle.
2

Dan George 1stFlight Software Engineer at Anduril Industries

Regarding the "O" article, static linkage between components makes unit testing harder (unit testing makes TDD more maintainable). But, that's not the point. I'm having a hard time imagining the "evil" blocking case. I keep relegating the blocking code to a simple event detector. When the event detector unblocks, it generates an event to the framework. This approach demonstrates the issue of eating up resources with event detectors but I'm not seeing a significant impact to the open-ness of the event processors. I suspect I've side-stepped the issue. What is the structure of code that is less open because of blocking?

@Julio, I applaud your approach to discussion!

Julio Diez Ruiz 3rd+Strategic Cloud Engineer at Google

@Dan. "the frequency at which I can take advantage of the natural sequential style is just too low". I'm not claiming the opposite and agree on your comments about protocols. Miro draws a situation in which an application has to deal with processing events, and to play fair and be useful to this discussion I precisely stick to protocol examples because I think they are representative in this case, are well suited to be implemented as state machines and can be complex enough. And even so, I intend to show that blocking is useful in some occasions, not in a lot of but in more than accepted by the original post.

For me, the point is that the drawn situation is prepared so that final arguments seem correct, but if you play a little changing some assumptions then you get to a different situation and hence conclusions. I think the same about the blog post "RTOS, TDD and..." It explains how you can misuse a feature to get to the desired conclusion that yes, it's right under those circumstances, but...

I intend to elaborate, but to focus and not digress I prefer to go step by step and wait for comments about my previous post. I think it's significant that @Miro, the OP, has taken part very few times so I will be probably in focus if he finds my post interesting enough as to comment.

Julio Diez Ruiz 3rd+Strategic Cloud Engineer at Google

@Dan and @Miro. I think you have summarized quite well very important points and opinions that I almost totally share... Almost because, @Miro, I don't agree on blocking being 'evil'. I think your arguments about proliferation of problems because of excessive use of blocking are right, but no more than any other kind of overuse or misuse of a programming tool or resource. I mean, being right and very good advice doesn't mean you have to almost completely dispense with it.

It's now a bit late hours for me but let me try with a simple example I think some related comments have already appeared.

Imagine you're implementing a well defined protocol with several states. In some of these states at some point you have to wait for a response or fail by timeout and no other events have to be considered. Sure you can implement this as a timer event, but:
1. I don't see the need of doing it so. Indeed I wouldn't like to see mixed events for external states with those for internal states (I suppose you know what I mean).
2. To write something like recv(..., tout), i.e. synchronous programming is much more natural and so simple in this case that doing other way is doing it difficult.
3. Blocking in this case is not 'evil'. It's natural, it's what (my) protocol specification says I have to do: keep in that state waiting for an answer before going to next state or fail.

**William Macre 2ndSr. Embedded Software Engineer at ?**

The subject of RTOSs can be so wide and in-depth that I only want to address one of the writer's statements. "The problem is that while a task is blocked, the task is not doing any other work and is *not responsive* to other events". The ideal number of functions a single task should support is one. Too often a requirement is broken down to a set of functions that the designer tries to cram into a single thread (task). I have found that simplifying the task and increasing the number them reduces the complexity of the problem. For example, a thread that requires five resources (mailboxes, semaphore, event signals, etc) is better served to be deconstructed into five smaller threads under a single parent task that monitors/controls the children task(s). Each child task is only waiting on a single event, not multiple events. If sequence of events is important, as most the time it is, the job is the responsibility of the parent task to preserve order (a.la state machine).

Austin Morgan 3rd+Senior Software Engineer at Garmin International

For those whom like state machines but want an nice event framework I suggest looking at QP [http://www.state-machine.com/qp/index.php|leo://plh/http%3A*3*3www%2Estate-machine%2Ecom*3qp*3index%2Ephp/uAWU?_t=tracking_disc].

1

Emmanuel Gaudin 2ndCEO, PragmaDev

@Miro: That is the problem when discussing UML, this language is so generic and open that you can state it does it all. You can use UML for anything, you can model hardware, software, our current discussion, the weather forecast or my building. Isn't that right ?

But as you say, since it can describe anything, it describes nothing but bubbles and arrows. And people spend their time discussing the meaning of that bubble or that arrow. On the other hand a DSL like SDL has an action language and a semantic of execution so it is not only bubbles and arrows. In practice SDL users do not spend their time discussing the language itself.

Now, as its name states a DSL is dedicated to a domain, and I believe SDL is very well suited to software running on top of a RTOS.

Emmanuel Gaudin 2ndCEO, PragmaDev

@Miro: I find pretty amusing, in the same post, to pretend to avoid a religious war between UML and SDL and to throw in another UML diagram again. In the first place I was replying to Mykola about the fact that a state/event transition could have several output states illustrated by an SDL diagram and you are the one who threw in a UML diagram.

@Miro again, your state machine is still not equivalent to mine.
1) As far as I know there is no concept of timer in UML, you just named the input message timeout_con_req.
2) A timeout is a synchronous operation and is not equivalent to a timer going off.
3) The counter is not incremented in your model.
4) Your counter has not data type.
Most of UML diagrams are quite informal without any semantic so it is not surprising there are comments on what the diagram really means. My piece of advice: if you spend more time talking about the modeling language than about your system, move to

another language.

@Mykola: I guess the level of details you want to put in your model is up to you. In my protocol definition the number of retries is part of the specification, it has an impact on the next state, so it is not an implementation detail.