The NUFR Philosophy

For NUFR version 0.1

Copyright © 2018, Bernie Woodland

All rights reserved.

Redistribution and use, with or without

modification, are permitted provided that the following conditions are met:

1. Redistributions must retain the above copyright notice, this

list of conditions and the following disclaimer.

THIS DOCUMENTATION IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE

DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;

LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Arm® is a trademark of Arm Holdings

Arm® Cortex® is a trademark of Arm Holdings

# Basic Feature Set

## What is NUFR?

NUFR is Not Your Father's RTOS, and if the reader's familiar with RTOSs but not with *not-your-father's-something*, then he or she will figure out what that means by perusing this manual rather than by googling the phrase.

But we'll back up a step and define what an *RTOS* is. Make that two steps and first define the terms behind *RTOS,* namely *Real Time* and *Operating System.* An Operating System is a software component that manages the resources of a processor, providing services to applications which ride on top of it. *Real Time* is a computing constraint on a processor which mandates that the software complete certain execution objectives by pre-defined time deadlines, in addition to the software fulfilling all the other requirements that software must fulfill. A video game is a real-time computing application: the computer must read the player's inputs and cause the game to respond within a few milliseconds. If the game responds correctly to the player's inputs, but in doing so delays the processing by too long of an interval, the game is consequently in a state of failure.

Real time systems go hand-in-hand with embedded systems. An *embedded system* is a processor which is embedded in a system where the computing complex is one component of the entire system. A car's braking system has a processing complex in it. This processor is an embedded processor, since the overall system is a braking system. The processor is also a real-time processor. A video game, on the other hand, is a real-time computing application but usually doesn't run on an embedded processor. In other words, the game runs on a PC or a gaming console, and a PC is a system unto itself.

Of course, RTOSs are operating systems made specifically to run real-time applications. The constraints put on real-time applications require special operating systems which can support these requirements. Since the requirements of real-time OSs can be onerous to non-real-time OSs, RTOSs are given their own processors to run in, processors which are dedicated to solely run the RTOS and its hosted real-time applications, sharing them with nothing else. And since an RTOS more often than not runs as an embedded processor, it doesn't require devices such as hard drives and the what-not that a computer has. And an embedded processor is smaller, lower power, and costs less than a computer. What this means is that the RTOS runs on a smaller CPU and with less hardware resources than what would be found on a PC. To fit on a smaller processor, an RTOS must be smaller in size and operate more efficiently than a non-real-time OS. On the other hand, many of the features in non-real-time OSs are missing in RTOSs.

## NUFR Design Philosophy

The prominent aspects of NUFR are that it:

* Is a micro-kernel
* Has a layered architecture
* Allows vital portions to be customized by the application developer
* Has a sophisticated set of features

### Micro-Kernel Architecture

A *micro-kernel* is a small kernel, so this means that the bottom-most layer of NUFR, the kernel, is small. In fact, it was designed to be as small as possible: if a service or a feature can be implemented outside of the kernel, then it is. NUFR has as few OS object types and as few APIs as possible. At the other end of the rope, however, there are features which the kernel must support, and some of these features are gnarly. An effort was made to pull out as much functionality as possible, but yet include as much functionality that could ever be needed. As a compromise, a few features are hidden behind compile switches, as many applications won't need them. But NUFR is defined as much by what is *doesn't* have as what it *does* have. It stays out of the way of the developer, and beckons him or her to peer into the kernel innards. NUFR tries to reveal itself rather than hide itself.

### Platform Layer

Another aspect of the micro-kernel architecture is that a select portion of NUFR is defined by the developer. This allows it to scale down to light-weight hardware devices but yet scale up to larger, more demanding environments. Namely, the customization layer is the *NUFR platform layer*. Like any other layer, the platform layer consists of a set of API calls. Most of these APIs are categorized as *mandatory*: NUFR cannot work without them, seeing that the kernel compiles against them and calls them. All platform APIs are written by the developer, howbeit with archetypical examples provide with the OS distribution. For mainstream applications, the archetypical examples will suffice—the developer can use the template APIs out-of-the-box, without modification. There is some functionality which is defined in the platform layer which must be supplied in its entirety by the application developer: he or she must define the number of tasks, the default priority and entry points of the tasks, the size and location in memory of the task stacks, the number of semaphores and their attributes, and the number of messaging blocks. This is expected of any RTOS; NUFR distinguishes itself by the degree of control and ease in which this is accomplished.

Some of the platform layer APIs can and will be called from the services and application layers, and not just from the kernel. Moving what other RTOSs keep in the kernel to a layer which can be tweaked by the application developer makes NUFR portable, flexible, and scalable. Are you tired of RTOSs burying Task Control Blocks (TCBs) in self-made memory heaps, and then having to configure those memory heaps? NUFR rids you of that—but only if you so choose: you could create your own memory heaps, peppered with your platform's particular debug asserts and other diagnostic hooks, and use those instead.

### Guided Application Development

Together, the NUFR kernel layer and NUFR platform layer present a terse set of APIs that guide application frameworks into utilizing solid design patterns. Many an embedded RTOS application codebase suffer from poor design patterns, and in the case of some RTOSs, these deficiencies have their roots in the RTOS's lack of support of crucial components or features. But in the plurality of cases, an RTOS provides numerous primitives—a set of adequate primitives—from which the founding developers must judiciously construct higher-level objects, the macro objects that would anchor application tasks in healthy design patterns. More often than not, application tasks make use of OS primitives in a manner that cause the entire task to veer off into a ditch, architecturally speaking.

By shepherding developers to the greener pastures of event-driven tasking, the applications that they produce will have less bugs, be less prone to spaghetti code, enjoy quicker time to market, and on top of that will consume less resources. The savings gleaned from using NUFR over other RTOSs are not necessarily that the kernel is written much more efficiently than other RTOSs (although NUFR is pretty tight), but that substantial gains are realized at the application layer.

### NUFR on Small Devices

NUFR excels on small devices more so than other RTOSs. It's smaller than other RTOSs and is easier to scale down, as there aren't as many compile switches that have to be wrangled over, and has flexibility through platform layer customizations that allows a developer to squeeze more out of it.

System-on-a-Chip (SoCs) devices have evolved over the years, and a comparison from a couple decades ago to today shows where great advances have been made, and conversely, where the limitations lie. Nowadays, CPU designs are more efficient per device (a *device* being a transistor) than they were in decades past. In other words, modern CPUs are not only across-the-board more powerful than older designs, but modern CPUs have more execution power per transistor than older designs. The peripherals found nowadays in small SoCs are better and more useful than they were before. There is more opportunity to optimize CPU designs and peripheral designs per transistor than to optimize RAM designs per transistor. It's difficult to find clever ways to get more memory cells out of 30,000 transistors, whereas ASIC designers have done just that with CPU and peripheral designs: over the years, they've squeezed more performance out of the same number of transistors. The resource that's most scarce in today's SoC's is RAM first and FLASH second. The bottom line is that an application running on a small footprint SoC will, most likely, be limited in performance and functionality by RAM or by FLASH, and not by CPU cycles. In other words, expect your typical application to run out of RAM before it runs out of computational power.

That RAM is a greater limiting factor to application size and scope than is CPU power is better understood, perhaps, by software engineers than by hardware engineers. An experienced embedded developer knows that there's a tradeoff between RAM utilization and CPU consumption. Given an algorithm to implement, or a functional library to develop, if the amount of available RAM is low, and must be conserved, the solution will be more CPU intensive (and likely will consume more code space also). But CPU power is a renewable resource; it's renewed over time. If CPU cycles aren't put to good use in the present, they cannot be saved up for use in the future. RAM and FLASH are not like that; a savings now will always be available in the future.

An understanding of the critical resource in SoCs has been a key factor in NUFR design decisions. NUFR was designed to save RAM and FLASH. The RAM savings is not just in the OS itself, but more importantly in the application space. How do NUFR applications save RAM? First, by having tasks implemented as event-driven message pumps. Event-driven designs allow for tasks to be combined. Tasks that aren't well-architected have execution paths that seize the task's thread for extended periods of time, or the execution path is delicate and brittle that adding new code to it risks introducing a regression bug. For that reason, firmware developers prefer to create new tasks, rather than use existing ones, when new features are added to a codebase. The problem with adding new tasks is that each new task must be supplied with a stack of its own, and on small RAM footprint devices, task stacks consume a lot of RAM.

Another area where NUFR saves RAM is that it makes the hardware source which generates the OS tick available to the application developer. On the ARM Cortex series of CPUs, this is the SYSTICK handler. This handler does double-duty: it not only drives NUFR's OS system clock, but it is exposed at the NUFR Platform Layer for use by the application developer. In this way, the SYSTICK handler is like a band-scheduled task which costs no extra RAM, with a few restrictions of course. Furthermore, since most firmware images are launched from a *main()* startup entry point, this thread is put to use as well (NUFR calls this the *Background Task* (BG Task). NUFR allows as many of its API calls as possible to be invoked from not just the task level, but from the BG Task and from ISR handlers, which includes the OS tick handler. And to save FLASH, there is one set of API calls, not two, for use at task level or at ISR level.

### Advanced Features and the Services Layer

In addition to the kernel and the Platform Layer, there is a third layer called the *Services Layer* (SL). Whereas the kernel and the platform layers are mandatory, the SL is optional. The purpose of the SL is to...

* Transform some of NUFR's less refined and tedious APIs to APIs which are more palatable to the application developer
* Add extra features and functionality that are absent in NUFR, being a micro-kernel, but are present in the typical RTOS

Mutexes are a prime example of SL content. Mutexes are nothing but semaphores used in a specific manner and for a specific purpose. Since NUFR builds semaphores into the kernel, adding mutexes to the kernel would contradict NUFR's micro-kernel design philosophy: if a service can be built outside the kernel, then do so. Mutexes are therefore an SL component which presents mutex API calls to the application layers, but internally consists of wrapper calls to NUFR semaphore APIs. This begs the question, "Why provide mutexes at all? Applications can use semaphores instead." A few reasons. First, the application developers who are accustomed to using mutexes will looks for mutex APIs in NUFR, since mutex APIs are normally found in RTOSs. Not finding one will be a nuisance, not and just because the APIs would have to be written, but a mutex wrapper at the application layer has a tendency to get lost. Some application layer APIs get because one only expects to find APIs pertaining to a specific application in that application go looking for it there. Furthermore, a mutex API up in the application code, might not be available for use elsewhere, as it might've been tailor-written for that application, and not made for generic use, or it might be difficult to export one of the application's APIs out to some other remote app. On top of all this, such an exported API is awkward. It just doesn't belong there. In contrast, a mutex API in the SL is expected and is designed to be freely distributed and is named accordingly. A SL mutex has almost the look and feel of a kernel mutex API that another RTOS would present.

NUFR comes with an SL that provides the OS objects found in the typical RTOS, mutexes being just one example; queues and pipes being others. But in these various OS objects is where one RTOS differs from another. Some codebases require that the application code adhere to an RTOS-independent set of APIs, or a codebase use one RTOS's APIs as a generic interface, and change the RTOS under-the-covers in a translation layer. This generic or translation API layer would be part of the SL.

# Case Studies

## A Typical Story

A hypothetical story shows how RTOSs contribute to poor architectures. Acme Inc. sells Halloween props. They decide their product portfolio needs to be enhanced to include some high-end gadgets which can be controlled by a laptop which hosts a haunted house control application. Towards this end, Acme hires a handful of embedded firmware engineers, and they brainstorm a solution which has USB-to-serial lines that connect the PC to any number of small, SoC based controllers, one controller per prop. The controller's micro turns on and off lights, and extends and retracts pneumatic actuators—all the sort of things that Halloween devices do—based on commands from the PC, and reports back interesting events to the PC, such as switches which were closed due to a haunted house guest stepping on a floor mat. The firmware in these props runs an RTOS. The application that runs on the RTOS has a variety of I/O lines that it reads and writes to make the actuator work, but then it needs a serial protocol (in this case HDLC) to communicate to the PC. At launch, the project is already late to market. The firmware engineers have no time for frills, they just need to get something working. They create a couple of tasks to handle the I/O devices and to run the serial interface. The tasks just run through their calculations in a long look, making an RTOS sleep API call so as to not hog the CPU. The product is demonstrated to the sales team, and they're delighted. With spring turning into summer, then quickly rush this into production to get this out before the end of October.

Meanwhile, Acme catches wind that their competitor, a mail-order company called *The Horror House*, plans to release competing products that do what Acme's products do *plus* have a WiFi interface. Acme gets their engineers working on WiFi versions, and to up the ante plan to migrate their serial link to run PPP and will mount an IP stack on top of it. Long term, they want to put a web interface on their chips, so users can configure the WiFi parameters using it. Speaking of WiFi parameters...the settings have to be stored persistently, and that points to a driver which permits storage of parameters in non-volatile memory.

Well, the Acme engineers get to work. They don't have time to re-do their simple sleep/polling loop—and why should they? In their minds, it works just fine. So they add a FLASH driver to preserve the WiFi parameters across power cycles, plus they add a WiFi driver and an IP stack. They decide to integrate these into the existing tasks, as they're running out of RAM. The tasks start to get gnarly. Timing corner-cases pop up...packets get dropped occasionally, causing the web interface seizes up for a few seconds every now and then. A few things that used to work reliably aren't reliable anymore. They tell management that it's going to take longer than anticipated to get the new features working right—and that they announce that they'll have to spend more for a chip that has more RAM, has a faster CPU, or whatever. One of the Acme VP's has a friend at another company who swears by this newest process he just implemented, where all the engineers get rid of their cubicles and arrange their desks in one giant circle. The VP is thinking this is what his firmware engineers need to do, since they take so long to get products out the door.

## A Design Study in NUFR

Acme decides to add a new product to their haunted house portfolio: a shrunken-skull scare trap. This contraption consists of a small plastic “skull” which has red eyes that illuminate. The entire skull can be lit from a key light off to the side. Along with the skull is a pre-recorded soundtrack embedded in an Acme standard module, and is triggered by a short digital pulse. The sound runs for two seconds. The scare is triggered by a contact switch that lies on a small floor mat; it closes when someone steps on it.

Acme decides to use a tiny, inexpensive Arm®-based SoC controller. The controller has discrete inputs and discrete outputs (DI and DO) lines—all the I/O that’s needed for this application.

The scare trap will work like this:

* The “victim” steps on the mat, activating the mat switch
* After a 2 second delay, the soundtrack starts
* ½ second later, the eyes illuminate
* 2 seconds later, the key light illuminates and the eyes flash rapidly 4 times
* 3 seconds later, the key light extinguishes
* After this, there is a 4 second interval before the scare trap can be activated again

Two state machines are formed from these requirements.

## The Master State Machine.

Master State Machine States

Idle-State

Start-Delay-State

Soundtrack-Started-State

Eye-Lit-Started-State

Key-Lit-Started-State

End-Delay-State

Master State Machine Events

Mat-Switch-Event

Master-Timeout-Event

Master-Cancel-Event

State Transition Table

This table lists what state the state machine will transition if/when an event of the specified type is received.

|  |  |  |  |
| --- | --- | --- | --- |
| *State* | *Event* | | |
|  | **Mat-Switch-Event** | **Master-Timeout-Event** | **Master-**  **Cancel-**  **Event** |
| **Idle-State** | Start-Delay-State | [ignored] | [ignored] |
| **Start-Delay-State** | [ignored] | Soundtrack-Started-State | End-Delay-  State |
| **Soundtrack-Started-State** | [ignored] | Eye-Lit-Started-State | End-Delay-  State |
| **Eye-Lit-Started-State** | [ignored] | Key-Lit-Ended-State | End-Delay-  State |
| **Key-Lit-Ended-State** | [ignored] | End-Delay-State | End-Delay-  State |
| **End-Delay-State** | [ignored] | Idle-State | Idle-State |

## The Eye Flash State Machine

This machine runs in conjunction with the Master State machine to control eye illumination only.

Eye Flash State Machine

Idle-State

Long-Light-State

Flash-State

Eye Flash State Machine Events

Eye-Flash-Start-Event

Eye-Flash-Timeout-Event

Eye-Flash-Cancel-Event

State Transition Table

|  |  |  |  |
| --- | --- | --- | --- |
| *State* | *Event* | | |
|  | **Eye-Flash-**  **Start-Event** | **Eye-Flash-**  **Timeout-Event** | **Eye-Flash-Cancel-**  **Event** |
| **Idle-State** | Long-light-  State | [ignored] | [ignored] |
| **Long-Light-State** | [ignored] | Flash-State | Idle-State |
| **Flash-State** | [ignored] | [flash light on & off for “N” counts, then]  Idle-State | Idle-State |

## Memory Map

We’ll assume a simple I/O layout that provides for Discrete Inputs and Discrete Outputs (DIs and DOs) in directly addressable DI and DO registers. These DIs and DOs are the only I/O needed by the scare trap feature. A bit value of “1” means “active” (switch engaged or light on).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Register* | *Address* | *Bit 2* | *Bit 1* | *Bit 0* |
| Scare Trap DI | 0xD0000000 | [not used] | Mute switch | Mat switch |
| Scare Trap DO | 0xD0000004 | Key light | Sound track | Eye light |

The mute switch is a hidden switch which, when activated, stops the scare in progress.

No hardware interrupt support for any of the DIs—polling of some sort will have to be used in order to detect DI bit transitions.

## Scare Trap Code

[*Editor’s note: review code snippet. There’s certain to be bugs!*]

Let’s take the state machines and the I/O described above and convert it to NUFR application code. Though there are several ways this can be coded, we’ll follow an event-driven design pattern. This will demonstrate the ease by which this scare trap example can be transformed from conception to implementation.

First, we’ll convert the state machines and state machine events into enums:

typedef enum MasterStateMachine

{

MSM\_IdleState,

MSM\_StartDelayState,

MSM\_SoundtrackStartedState,

MSM\_EyeLitStartedState,

MSM\_KeyLitEndedState,

MSM\_EndDelayState

};

typedef enum MasterStateMachineEvent

{

MSME\_MatSwitchEvent,

MSME\_MasterTimeoutEvent,

MSME\_MasterCancelEvent

};

typedef enum EyeFlashStateMachine

{

ESM\_IdleState,

ESM\_LongLightState,

ESM\_FlashState

};

typedef enum EyeFlashStateMachineEvent

{

ESME\_StartEvent,

ESME\_TimeoutEvent,

ESME\_CancelEvent

};

To help with the message routing, a set of message prefixes will be defined. These are defined in the Services Layer (SL), in the file *nsvc\_app.h.*

typedef enum

{

NSVC\_MSG\_PREFIX\_local = 1, //mandatory

NSVC\_PREFIX\_MAIN\_STATE\_MACHINE,

NSVC\_PREFIX\_EYE\_FLASH\_STATE\_MACHINE

} nsvc\_msg\_prefix\_t;

Next is to tackle the DI polling problem. The DIs must be polled, but where and how should this be done? Polling consumes a lot of CPU cycles, and, if implemented incorrectly, ratchets up code complexity. NUFR has an efficient solution to this problem, particularly valuable on resource-constrained systems. The solution is to insert the polling code in the NUFR Platform Component, in nufrplat\_systick\_hander(). Placing the DI polling code in this location saves resources in the following way:

* A dedicated task won’t be required. This saves the RAM that a task TCB and task stack would consume.
* The context switching that a polling task would require is removed, saving CPU time. This context switching consumes more CPU cycles than the polling logic.
* Adding the polling code in *nufrplat\_systick\_handler()* means that no new function calls will be necessary. The poll check will only require a few assembly instructions.

The output of the polling code will be events that are thrown into the Main State Machine. The OS tick timer is set to 50 times/second, which is 20 milliseconds (*NUFR\_TICK\_PERIOD* set to 20). This tick interval is slow enough that no switch debouncing logic is necessary.

uint32\_t previous\_di\_register;

void nufrplat\_systick\_handler(void)  
{

uint32\_t current\_di\_register;

....

current\_di\_register =

\*(volatile uint32\_t \*0xD0000000);

// Check entire register for any changes.

// This minimizes CPU usage, on average.

if (current\_di\_register != previous\_di\_register)

{

bool mat\_switch\_deactivated;

bool mute\_switch\_activated;

nufr\_msg\_t \*msg;

// Trigger on the 1-to-0 transition. This trigger

// is when someone was sanding on the mat then

// leaves it.

mat\_switch\_deactivated =

((current\_di\_register & 1) == 0)

&&

((previous\_di\_register & 1) == 1)

// Mute switch triggers on 0-to-1 transition.

// The moment the switch is pressed, the event

// occurs.

mute\_switch\_activated =

((current\_di\_register & 2) == 1)

&&

((previous\_di\_register & 2) == 0)

if (mat\_switch\_deactivated ||

mute\_switch\_activated)

{

msg =

nufrplat\_msg\_get\_block(nsvc\_bpool\_sema\_block);

if (NULL != msg)

{

// Shortcut to save a bit of code: assume

// both events never happen at same time.

if (mat\_switch\_deactivated)

{

msg->fields = NUFR\_SET\_MSG\_FIELDS(

NSVC\_PREFIX\_MAIN\_STATE\_MACHINE,

MSME\_MatSwitchEvent,

NUFR\_TID\_SCARE\_TRAP,

NUFR\_MSG\_PRI\_MID);

}

else

{

msg->fields = NUFR\_SET\_MSG\_FIELDS(

NSVC\_PREFIX\_MAIN\_STATE\_MACHINE,

MSME\_MasterCancelEvent,

NUFR\_TID\_SCARE\_TRAP,

NUFR\_MSG\_PRI\_MID);

}

}

With the polling code taken care of in the SysTick Handler, there remains the state machines. This will be done in a task with a task ID of *NUFR\_TID\_SCARE\_TRAP* and an entry point of *ScareTrapTaskEntry()*. The task sits idle, having no activity whatsoever, until scare trap activity awakens it.

void ScareTrapTaskEntry(unsigned parmNotUsed)

{

nsvc\_prefix\_t msgPrefix;

uint16\_t msgIdUint16;

while (1)

{

nsvc\_msg\_get\_argsW(NULL,

&msgPrefix,

&msgIdUint16,

NULL,

NULL,

NULL);

switch (msgPrefix)

{

case NSVC\_PREFIX\_MAIN\_STATE\_MACHINE:

MasterStateMachine(

(MasterStateMachineEvent)msgIdUint16);

break;

case NSVC\_PREFIX\_EYE\_FLASH\_STATE\_MACHINE:

EyeFlashStateMachine(

(EyeFlashStateMachineEvent)msgIdUint16);

break;

default:

break;

}

}

}

The scare trap task naturally decomposes into two functions, one for each state machine. The state transition tables are converted into classic state machine implementations:

#define DO\_EYE\_LIGHT 0x00000001

#define DO\_SOUND\_TRACK 0x00000002

#define DO\_KEY\_LIGHT 0x00000004

#define SOUNDTRACK\_PULSE\_MSECS 25

#define DEAD\_TIME\_SECS 4

volatile uint32\_t \*do\_register =

(volatile uint32\_t \*)(0xD0000004);

MasterStateMachine MasterState = MSM\_IdleState;

nsvc\_timer\_t mainTimer;

void MasterStateMachine(MasterStateMachineEvent event)

{

switch (MasterState)

{

case MSM\_IdleState:

if (MSME\_MatSwitchEvent == event)

{

MasterState = MSM\_StartDelayState;

MainTimerStart(2000);

}

break;

case MSM\_StartDelayState:

if (MSME\_MasterTimeoutEvent == event)

{

MasterState = MSM\_SoundtrackStartedState;

// Start the soundtrack: pulse DO quickly

\*do\_register |= DO\_SOUND\_TRACK;

MainTimerStart(SOUNDTRACK\_PULSE\_MSECS);

}

else if (MSME\_MasterCancelEvent == event)

{

CancelMain();

}

break;

case MSM\_SoundtrackStartedState:

if (MSME\_MasterTimeoutEvent == event)

{

// First timeout? This is to turn pulse off

if ((\*do\_register & DO\_SOUND\_TRACK) != 0)

{

// Stop the soundtrack

\*do\_register &= ~DO\_SOUND\_TRACK;

// finish ½ second delay

MainTimerStart(500 –

SOUNDTRACK\_PULSE\_MSECS);

}

// Second timeout

else

{

MasterState = MSM\_EyeLitStartedState;

\*do\_register |= DO\_EYE\_LIGHT;

MainTimerStart(2000);

// Start the eye flash sequence

nsvc\_msg\_send\_argsW(NULL,

NSVC\_PREFIX\_EYE\_FLASH\_STATE\_MACHINE,

ESME\_StartEvent,

NUFR\_MSG\_PRI\_MID,

nufr\_self\_tid(),

0);

}

}

else if (MSME\_MasterCancelEvent == event)

{

CancelMain();

}

break;

case MSM\_EyeLitStartedState:

if (MSME\_MasterTimeoutEvent == event)

{

MasterState = MSM\_KeyLitEndedState;

\*do\_register |= DO\_KEY\_LIGHT;

MainTimerStart(3000);

}

else if (MSME\_MasterCancelEvent == event)

{

CancelMain();

}

break;

case MSM\_KeyLitEndedState:

if ((MSME\_MasterTimeoutEvent == event)

|| (MSME\_MasterCancelEvent == event))

{

CancelMain();

}

break;

case MSM\_EndDelayState:

if (MSME\_MasterTimeoutEvent == event)

{

MasterState = MSM\_IdleState;

\*do\_register = 0;

}

break;

}

void MainTimerStart(uint32\_t timeoutMillisecs)

{

nsvc\_tm\_divisor\_t divisor;

uint32\_t timeout\_ticks;

// Save some CPU time on long timeouts

if (timeoutMillisecs > 100)

{

divisor = NSVC\_TMDIV\_100MILLISECS;

timeout\_ticks = NSVC\_MILLISECS\_TO\_DIV100M(

timeoutMillisecs);

}

else

{

divisor = NSVC\_TMDIV\_NONE;

timeout\_ticks = NSVC\_MILLISECS\_TO\_DIVNONE(

timeoutMillisecs);

}

nsvc\_timer\_start(&mainTimer,

divisor,

NSVC\_TMODE\_SIMPLE,

timeout\_ticks,

NSVC\_TIMER\_SET\_PREFIX\_ID(

NSVC\_PREFIX\_MAIN\_STATE\_MACHINE,

MSME\_MasterTimeoutEvent));

}

void CancelMain(void)

{

\*do\_register = 0;

nsvc\_timer\_kill(&mainTimer);

delayValue = DEAD\_TIME\_SECS;

MasterState = MSM\_EndDelayState;

// Send cancel to eye flash state machine too

nsvc\_msg\_send\_argsW(NULL,

NSVC\_PREFIX\_EYE\_FLASH\_STATE\_MACHINE,

ESME\_CancelEvent,

NUFR\_MSG\_PRI\_MID,

nufr\_self\_tid(),

0);

}

#define FLASH\_INTERVAL 200

EyeFlashStateMachine EyeFlashState = ESM\_IdleState;

nsvc\_timer\_t eyeFlashTimer;

unsigned flashCount;

void EyeFlashStateMachine(EyeFlashStateMachineEvent event)

{

switch (EyeFlashState)

{

case ESM\_IdleState:

if (ESME\_StartEvent == event)

{

\*do\_register |= DO\_EYE\_LIGHT;

EyeFlashTimerStart(2000);

}

break;

case ESM\_LongLightState:

if (ESME\_TimeoutEvent == event)

{

// start with off cycle

\*do\_register &= ~DO\_EYE\_LIGHT;

flashCount = 4\*2; // x2 for on/off cycle

EyeFlashTimerStart(FLASH\_INTERVAL);

}

else if (ESME\_CancelEvent)

{

CancelEyeFlash();

}

break;

case ESM\_FlashState:

if (ESME\_TimeoutEvent == event)

{

if ((\*do\_register & DO\_EYE\_LIGHT) != 0)

{

\*do\_register |= DO\_EYE\_LIGHT;

}

else

{

\*do\_register &= ~DO\_EYE\_LIGHT;

}

flashCount--;

if (0 == flashCount)

{

CancelEyeFlash();

}

else

{

EyeFlashTimerStart(FLASH\_INTERVAL);

}

}

else if (ESME\_CancelEvent)

{

CancelEyeFlash();

}

break;

}

}

void EyeFlashTimerStart(uint32\_t timeoutMillisecs)

{

nsvc\_tm\_divisor\_t divisor;

uint32\_t timeout\_ticks;

// Save some CPU time on long timeouts

if (timeoutMillisecs > 100)

{

divisor = NSVC\_TMDIV\_100MILLISECS;

timeout\_ticks = NSVC\_MILLISECS\_TO\_DIV100M(

timeoutMillisecs);

}

else

{

divisor = NSVC\_TMDIV\_NONE;

timeout\_ticks = NSVC\_MILLISECS\_TO\_DIVNONE(

timeoutMillisecs);

}

nsvc\_timer\_start(&eyeFlashTimer,

divisor,

NSVC\_TMODE\_SIMPLE,

timeout\_ticks,

NSVC\_TIMER\_SET\_PREFIX\_ID(

NSVC\_PREFIX\_EYE\_FLASH\_STATE\_MACHINE,

ESME\_TimeoutEvent));

}

void CancelEyeFlash(void)

{

EyeFlashState = ESM\_IdleState;

\*do\_register &= ~DO\_EYE\_LIGHT;

flashCount = 0;

nsvc\_kill\_timer(&eyeFlashTimer);

}

## Conclusion

The simple example above takes a problem from conceptualization to implementation, from state diagrams and events to code. The event-driven design patterns which NUFR promotes guides the application developer during the coding phase. The resultant code has no unforeseen corner cases with respect to the state diagrams. Since there is little polling, the code is only exercised when there is actual activity which warrants it. This makes it easy to set breakpoints and stop when needed and to add logging, since there are no wasted paths.

One might be tempted to skip the event driven code and implement the above in a single polling loop. And the result may be smaller than the example code. But as the states become more complicated, it becomes more and more difficult to extend the polling model—whereas the event-driven model extends itself logically and elegantly.

# Where Other RTOSs Go Astray

At a high level, most of the mistakes which RTOS creators commit can be categorized into three groups:

1. The Big-OS Mentality

2. Failure to Thoroughly Think Through the Architecture

3. Inadequate Understanding of the Application Developer’s Needs

## The Big-OS Mentality

For a lack of a better term, a *Big-OS Mentality* is when RTOS creators put features and functionality in an RTOS which is suitable for an OS like Linux but not suitable for an RTOS. Big-OS features are added to paint a veneer of sophistication over the RTOS, or to try to make the RTOS look advanced or more capable and therefore more attractive. Another reason is to add a feature that, on a big OS, provides protection, in an attempt to add the protection to an RTOS-based codebase.

Code that runs in an RTOS environment doesn’t enjoy the same protection that code which runs on a memory-managed OS like Linux does. Those who develop in an environment where they’re used to that protection may have trouble adjusting to the bare-bones environment of an RTOS-based computing system. But there’s no sense in trying to make an RTOS work like a memory-managed OS. Since RTOSs are compiled together with the application code which runs on them, into a single image, there is an intimate relationship between the two.

## Failure to Thoroughly Think Through the Architecture

Since resources are constrained in an RTOS environment, there’s little tolerance for mediocre designs. Every byte must tell, every function must be carefully considered, and the system as a whole must be well thought out.

## Inadequate Understanding of the Application Developer’s Needs

I have to how much experience those who design RTOSs have in writing code to run in an RTOS-like environment. Most know how to write efficient code and can optimize for this or for that, but they don’t fully understand what end-goal they need to optimize to. These designers add features nobody’s going to use, omit features that are really needed, and in the end create something that requires an expert in real-time embedded OSs to create good design patterns. Sure, the competent developer will be able to use their OS to craft a solution. But most competent developers aren’t familiar enough with design patterns to take a give OS’s object primitives and form good designs out of them.

## The Disliked-Feature List

The RTOS Source Code is Missing

Any party shopping for a prospective RTOS on their next project should find one that includes the source code and should compile the source code together with the application code. There are two reasons they do this. First, a pay-for-license RTOS developer is in the business for profit and therefore wants to protect their Intellectual Property (IP). So they try to sell their RTOS without the source code. Nobody should purchase an RTOS without the source code, and without the source code supplied in a way in which it will be compiled on their system. The only reason preventing system developers from only choosing free RTOSs is that in many industries, like the medical, aerospace, or automotive industries, RTOSs have to be certified, and certification costs money. So a free RTOS—while just as good, probably better than a paid RTOS—would do just as well as a certified one, it can’t be used. If for no other reason than the fact that a purchased RTOS costs money, either one-time or royalties. But if an RTOS developer is withholding the source code for their RTOS, then they’re not looking out for the best interests of their customers. You must have the source code.

The Big OS mentality prevents some from delivering RTOSs without the source code. Having kernel source code compile with application source code feels like there’s a violation of the separation of kernel and application code. So they feel the need to deliver a library instead.

Using only the RTOS’s binary without the source forces the system developer to live with the compile switch settings that the vendor compiled their library with. It prevents developers from stepping through the kernel source code, which is one of the most valuable ways of learning the kernel, and from setting break points in the kernel. It also prevents the system developer from making custom changes to the RTOS, to add features unique to his or her environment, from fixing bugs that have an impact on his or her system, from adding logging hooks, crash dump hooks, or any number of system-wide utilities at the end-user’s disposal.

The RTOS Source Code Is Not Commented Sufficiently

The Big-OS mentality is that the app developer should keep away from the kernel, that he or she need only read the manual and make the API calls and the OS will take care of them. Embedded developers should have access to the source code in their project, so they can browse through it or step through it in a debugger, to understand how it works. Documentation is no substitute for source code. Source code is the ultimate documentation. There should be copious amounts of comments in RTOS source files so that the system or app developer can read through them. Function definitions should include commented descriptions, should list all the inputs and outputs to the functions, should list the range of valid inputs and outputs for function parameters, should list the corner-cases of misuse of parameter values. Source code in an RTOS should be held to the same standards as the application source code—at a minimum. The RTOS should follow best-practices in style and coding and document reasons (kernels have to break the rules because they have special requirements) why they deviate from them.

The RTOS Rely on Inline Functions

A lot of developers love inline functions. I don’t. One can’t set breakpoints on an inline function. Code having inline functions is hard to read. Inline functions are used to put common functionality in the same location, for the same reason that functions are used. Most of the time, they’re made inline to eliminate the overhead of a function call. But heavy use of inlines in an RTOS often times masks a greater problem—the problem that the RTOS has too much unnecessary junk in it.

A mixture of inline functions, macros, and compile switch will most likely be challenging to read. It’s best to limit their use and limit their interaction. This is challenging in an RTOS kernel, requiring a lot of thought.

The RTOS Has Too Many Compile Switches

Many RTOS creators think they’re doing the end-user a favor by having a lot of compile switches. They are not doing them a favor. RTOS creators have so many switches that they have to make a PC applet to manage the switches. Many a system developer just turns on all the switches so he or she won’t ever have to deal with them. They just turn on all the switches and take the hit on the increased FLASH size, etc. With a multitude of compile switches, you enter the compile-switch-dependency-jungle. The only thing worse than trying to get the customized load to compile is missing a switch and not finding out about it until something doesn’t work.

An RTOS with a lot of compile switches is probably bloated, and the RTOS maker is trying to mitigate the damage—at the end-user’s expense. A component that has a lot of compile switches is difficult to navigate. Often it become problematic just to tell if a compile switch is on or not. For this reason, a lot of large codebases have restrictions in place which forbid the use of compile switches. RTOSs should follow suit.

No Kernel-Level Asserts or Equivalent Checks

Asserts, if used properly, are an asset to application code. They’re also an asset to kernel code. Why not use them? Even if they’re turned off, they provide a supplemental means of documenting the kernel code.

Bloat

Many RTOSs are bloated with features that nobody will ever use. Why include them? RTOS creators think they’re doing app developers a favor by including a lot of junk, but in reality, it just gets in the app developer’s way. It’s not just bloat which is a problem, but it is kernel bloat. RTOS kernels get filled with too much stuff which can, and should, get moved outside of the kernel.

Any piece of functionality added to the kernel is subject to the difficulties of that kernel. If a kernel is difficult to read, if the kernel is crammed with features, then all those features are difficult to read. Not only that, but code which runs inside of a kernel is subject to constraints that code outside of a kernel isn’t. There must be a compelling reason to put anything in the kernel. If there is no compelling reason, that functionality shouldn’t be there. Given the choice between placing an object or a service inside the kernel or outside, with all things being equal, a developer should choose putting it outside. Services outside the kernel are easier to read, easy to compile in or out, not subject to the myriad of compile switches that the kernel is. These services can share logging facilities, or things of that nature, with the codebase in which they live. They are more easily customizable. They are more easily tested in a unit test environment. Furthermore, there’s a reduced risk and fear of something going wrong with these. Kernels are tricky business.

Kernel Heaps

The Big-OS mentality says that an RTOS should have a kernel heap. All objects in the RTOS should allocate their objects, like TCBs, semaphores, etc., from this heap at startup. But why? What advantage does a kernel heap give you?

Generally speaking, embedded code (well...any code) should prefer statically allocated objects over dynamically allocated objects. All things being equal between a static allocation and a dynamic allocation, choose the static. Only choose the dynamic if the static won’t work, and that after careful consideration.

Kernel heaps inherit the same problems as any dynamically allocated object. First, the translation from kernel enum to actual kernel object is more time consuming and requires some sort of a RAM translation table (which gets embedded in an inline function). Static objects can be translated from their enum identifier to the object in a handful of assembly language instructions.

Second, the kernel objects are unlocatable in a linker map file. This means that the average debugger has no easy way to access them and to use their parsing features to decompose the kernel structure members. So kernel developers add some sort of utility or hook or something to allow you to view the dynamic objects. Why not just use the debugger as-is and skip all that? What happens when that special hook or tool, etc. isn’t supported on a platform one wishes to port the RTOS to?

Another advantage to having kernel objects statically defined, and thereby appearing as linker symbols, is that if there’s a memory corruption due to a stack overrun, an array bounds overrun, or some other memory overrun, then one can look up adjacent variables in the linker list file. If an RTOS developer wants more separation between kernel and application variables, that can be handled by using compiler-specific directives to mark the kernel objects so that they are placed in special RAM regions designated by the linker.

Third, having a kernel heap means that you have to do heap management for the kernel objects. This means tuning the heap size to fit the consumption. Sometimes RTOS makers add a PC applet that helps with this. Why manager a heap when the linker will do the management for you? That’s what linkers do.

Code Generators and App Tools

Over the years I’ve used a few code generators. I have yet to like any of them. Most code generators won’t let you modify the code which has been generated. Then code generators change from release to release. I could go on. App tools for RTOSs are hit & miss too; some are good, others are not. The problem isn’t the app itself necessarily; the problem is that the need for an app hides a greater problem: the code’s too difficult to use. Eliminate the complexity and the code generator and applet go away.

Non-Standard Scheduling Algorithms

There’s a well-known algorithm for task scheduling used by the prominent RTOSs. It covers multi-tasking, task prioritization, preemption—all the things tasks can and should do. Learning the nuances of this algorithm requires thought and experience. Once learned, this investment in understanding can only be carried over to another RTOS if that RTOS adheres to the same scheduling algorithm. Otherwise, the app developer must learn an new algorithm.

That in and of itself may not be so bad, but if application code is written to a non-standard scheduling algorithm, there are more difficulties when porting parts or all of the codebase an RTOS with a standard scheduler. The problems get magnified as the codebase grows in size and complexity, to the point where the risk in porting the codebase becomes the deciding factor in not porting it. One becomes locked into an RTOS in this way. This applies to things like non-standard semaphores, mutexes, etc.

APIs which Require Users to Construct OS Services from Primitives

There are well-known and commonly used RTOSs that don’t provide singular or dedicated APIs for the services that the app developer needs. For example, many RTOSs don’t provide a messaging API, so the app developer has to construct on using the OS objects that the RTOS supplies. More often than not, this expertise is beyond the realm of even good embedded developers. Many times a developer isn’t aware that he or she should seek a better solution, or that there’s a way to build interesting features out of a collection of RTOS objects.

In addition to this, if a developer decides to construct interesting services out of RTOS primitives, he or she must anticipate the corner-cases that occur frequently enough to be flagged as a problem but infrequently enough to make the troubleshooting of such problems easy. Having had bad experiences in the past, the developer views the construction of these services as too risk-prone, and shies away from them.

Lack of Essential Features

While building real-time systems on resource-constrained devices or on ones that require large scales, the developer runs across a host of problems which must be solved. The RTOS must provide the means to solve them and solve them efficiently. A nightmare is created when a developer has to work around the limitations of an RTOS.

Unnecessary Code

While quoting William Strunk Jr., I’ll leave it to the reader to substitute *writing* and its equivalent with *coding*:

“Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts. This requires not that the writer make all his sentences short, or that he avoid all detail and treat his subjects only in outline, but that every word tell.”

Like any other piece of codes, RTOSs are infested with unnecessary layers, unneeded abstractions, superfluous object pools, indirections of every sort, defines and macros that could be streamlined. Just a lot of junk.

# NUFR Architecture Explained

## Raging and the NUFR Component

*Raging* is the overarching codebase within which lie several components. NUFR is one such component. Raging has a suite of utilities (*nufr-utils.c* for one) which NUFR uses, so that NUFR makes no standard library calls. Removing any library support from NUFR makes it easier to build and the result should consume less code text (FLASH). While there are a handful of library calls, such as *memset()*, which when written in assembler execute more quickly than the C implementation (and raging utils may one day be optimized with inline assembler to eliminate this advantage), the benefits of being disconnected from the standard library outweigh the corner-case losses to CPU cycles.

Raging provides a near-*sprintf()* equivalent function, which should save FLASH, since its simplicity and reduced functionality doesn’t necessitate pulling in all the under-the-cover functions which the typical full *sprintf* implementation would pull in. This savings in FLASH could make or break a small-footprint project.

And the utilities which NUFR use are available to the app developer in perpetuity. The Raging utilities are given away free, so the app developer can write code which use these knowing that there are no restrictions when ported to other platforms. Even if NUFR is no longer used, the Raging utilities can continue to be used. A problem with many RTOSs—especially the ones with restrictive licenses and which are not free—is that they don’t give away useful code like utilities. Applications must either write their own utilities are try to find some free ones on the Internet.

## A Straightforward API Call Stack

A difference between NUFR and the typical other-RTOS is that NUFR doesn’t use the ARM® Cortex®-supported SVC kernel API calling convention. ARM® Cortex® M Series CPUs provide two software exceptions (interrupts) called *SVC* and *PendSV.* These two exceptions are specifically intended for use by the OS. Most RTOS ports to Cortex M-series CPUs use these facilities; NUFR uses PendSV but does not use SVC.

What this means is that a NUFR API call executes in the caller’s context, instead of invoking an SVC exception and vectoring off into the SVC handler. Using straightforward API calls instead of the SVC-model has advantages. First, a NUFR API call is easier to follow, since it’s a single-level API call which only has a few places where kernel functions are called from within the API. This simple structure allows the app developer to more easily read the lines of code which the kernel exercises during an API and more easily understand what they do. This is in keeping with the NUFR philosophy: keep the app developer close to the OS, not far from it. Many embedded app developers lack an intricate understanding of RTOSs. NUFR remedies this by making itself more transparent and more easily understood, with an eye towards educating the app user. The best way to learn the RTOS that one uses it is to go through it a line at a time. Line-by-line code inspection is commonplace for embedded developers. Stepping through the code in some sort of debugger helps a lot too. NUFR is designed with line-by-line inspection in mind. It prefers simple over complex, straightforward over indirect. It’s commented, has a minimum of compile switches, refrains from using inline functions (many debuggers won’t allow you to step through an inline function). And it doesn’t use the SVC call model, since this adds complexity. Simply run NUFR through any code complexity tool and compare the results with another RTOS—an RTOS having a comparable feature set—and most likely NUFR will score better than the other RTOS.

While user manuals and online documentation are instructive, too many software projects rely on these alone. Manuals should be used in conjunction with the comments in the source code; together, they form the documentation package that the app developer needs. Open source code suffers from poor commenting. NUFR provides comments on all API calls, and the comments describe how the API works from a high level, lists any restrictions or side-effects, then documents all parameters and the return value from an input/output perspective. Function header comment blocks are encoded in Doxygen markup, so that the most up-to-date API documentation can be produced in Doxygen. In fact, the NUFR manual has no details on the API calls. One must resort to the source code or the Doxygen output to ascertain how the APIs work in their totality. Comments embedded in source code are more likely to be up to date than standalone documentation.

Granted, that there are features in NUFR which are tricky or difficult to understand. But decent code browsing tools like Eclipse, SlickEdit, and cscope will allow the end-user to efficiently browse through the source. In addition, the app developer can run the pthread-simulation and step through various kernel scenarios in a source-level debugger, like Visual Studio (including the free edition) or gcc. The app developer, viewing variables populated with the correct values, can either write his or her own code and step through it in the simulator, to see what it would do in a corner-case, or can step through one of the test cases included in the Raging distro. In additon to the pthread simulator, the distro also comes equipped with QEMU simulation environment. In fact, it is recommended that the system or app developer verify the correct functionality of the OS-intensive portions of his or her code first in one of the simulation environments, before putting the code on real hardware.

A primary reason why NUFR is an attractive alternative to other RTOSs is that it promotes a healthy development model. The NUFR development model is to test as much offline in a unit test (UT) or a simulation environment, before testing code on the actual hardware. Too many developers go straight from code completion to hardware, skipping the UT and simulation steps. They end up wasting more time in the long run by skipping the intermediate steps of UT and simulation. In addition, many corner-cases can only be exercised with difficulty on an actual hardware environment, but can be tested with relative ease in a UT or a simulation environment. By skipping the SVC-model, NUFR adapts itself more readily to any offline environment.

Since NUFR allows for better integration with any offline or simulation environment, the imaginative app developer can construct a Linux-based simulator that runs NUFR on a pthread emulation layer. The system integration testing that can be done with such a tool is a powerful means of catching bugs and of accelerating development.

Imagine the following hypothetical system: a small SoC which is used to detect seismic vibrations, gathering data for a central server which runs software which tries to anticipate earthquakes in real time. There are many vibration-detection SoCs connected in a network to the back-end server. Each SoC has an array of built-in A/D converters which are attached to a multitude of vibration detection transducers (accelerometers, possibly). The SoC’s network connectivity consists of a 2-wire serial connection. IPv6 over PPP is run over this serial interface. The SoC does some local signal processing in real-time, then periodically sends an IPv6 packet over the serial connection. This gets switched or routed back to the server, where the data is aggregated, and analyzed.

Many naive developers, pressured by project managers who have little insight into embedded system developers, would encode the algorithms, develop the drivers, and implement the networking stacks, then would immediately, without any testing, conglomerate all these into a single image, so they could rush to deploy SoCs in a “real” environment, making management happy. While this appears to the ignorant as the fast path, in reality it is the slow path. More time will be wasted developing in this fashion than if the same system were developed a piece at a time, and if the firmware was tested on various types of simulators before it was loaded onto any SoCs. For the example here, the NUFR-hosted code could be hosted on a pthread-based Linux PC. Using Linux calls, the transmit and receive over 2-wire serial could be translated to a Linux USB serial output, and this could plug directly into the network. A pthread PC simulation allows for copious logs to be collected, root-causing problems quickly and efficiently. The simulator could be adapted to use simulated A/D readings, even playing back canned data captures from real tremors. The system integration between the SoCs and the server could be achieved much more efficiently. Many—most—of the problems that would occur on real SoCs would occur in the simulation, and would be solved in the simulator. At a minimum, any problem popping up in the simulation environment would surely have appeared on the real chips. The use of simulators builds an understanding of the system amongst the entire team, and hardens the actual software and firmware that will be used on the actual hardware.

Simulators offer many possibilities. The problem is that the average RTOS isn’t constructed with simulation in mind. NUFR is. Naturally, there are some embedded system architects who will criticize this aspect of the NUFR architecture, namely they will criticize NUFR for not using the SVC kernel model. They will say that the proper way for an RTOS to work is to invoke an SVC call and have the kernel execute on the main (interrupt) stack, not on the process stack, not in process mode. This boils down to two specific criticisms: that running an OS in an SVC exception rather than in a task’s context is “safer”. The second criticism is that the NUFR kernel disable interrupts for lengthy periods of time, causing interrupt high interrupt latency.

The first point of contention, that the kernel should only be entered using SVC exceptions, has its roots in non-embedded OS computers, where the OS and the applications are separate entities. In that environment, invoking the kernel vi software exception provides for a means whereby the application code can find the kernel API entry points. Since the software interrupt is the means of invoking the kernel, the kernel API functions never get mapped to real or virtual addresses. On a memory-managed OS, processes can be mapped into their own memory partitions, so that one process cannot directly access the memory that another process lives in. While mapping to physical addresses could be accomplished, the software exception is a cleaner interface between the process running in user-land and the kernel running in kernel-land. When the software exception occurs, levels of protection are traversed as execution passes from user-land to kernel-land. This is significant on a memory-managed OS like linux.

But for small, embedded computing platforms, no memory management exists. The kernel and the application code is compiled together into one image, solving the problem of the app code finding the kernel entry points. There only remains the difference of context and stack, what context the kernel executes in. Putting aside the feeling that having kernel code execute in a task’s context is unclean, there’s a perception that executing kernel calls on a separate kernel stack (interrupt stack) is more resilient to system crashes; that a process overrunning its stack while making a kernel call is the worst of catastrophes that could happen on an embedded processor. First, any stack overrun causes a crash, and any crash is bad, very bad, unacceptably bad. But is a stack overrun while executing kernel code worse? No, it’s no worse. First, the stack overrun itself won’t destroy any kernel objects, as these will be (should be) situated in a RAM partition far from the stack partition. If all the stacks are placed in the same partition, then the stack overrun will, most likely, eat into the next stack. The kernel objects will be unharmed.

Not having the kernel run on its own (actually, not its own, shared with interrupts) stack, may, in some corner-cases cause a net increase in RAM consumption, as all the net increase in the process stacks may be greater than the singular increase of the interrupt stack. This is a minor consideration, however.

The second point of contention against using SVC exceptions/software exceptions is that NUFR must use longer critical sections (interrupt locks) than the SVC method, and that this lengthens interrupt latencies. This is not a problem because NUFR API calls cut out the overhead of an SVC and take a more straightforward path, resulting in a more efficient overall call. The NUFR philosophy is to lock interrupts, quickly do what needs to be done, and unlock interrupts. Number of assembler instructions while interrupts are locked range from 50-150, typical. If one assumes an average of 1 instruction each 2 clock cycles, then for an 16MHz clock, this is a worst-case latency of 40 microseconds. The same folks who are quick to condemn interrupt latency often times cannot tell you what an acceptable interrupt latency is. As latency is a function of CPU power, battery powered devices are prone to have less CPU power than non-battery powered devices. Therefore, latency is more pronounced on battery powered devices. But on battery powered devices, one is more concerned with total CPU cycles per computing task than latency.

## The Rich Feature-Set Micro-Kernel

NUFR is a micro-kernel architecture. This means that an attempt has been made to limit the size, content, and features which exist in the kernel. Anything which can be removed from the kernel has been removed; anything which doesn’t need to be there isn’t there. This is the reason why there’s an Services Layer (SL). The SL contains the extra OS objects which the typical RTOS includes, to fill out the capabilities of NUFR. But these features need not be part of the kernel.

There is, however, a set of features contained in the kernel which cannot exist outside the kernel. Some of these features are essential for any decent RTOS. One such example is the capability of making OS API calls from interrupt context, which includes API calls which result in a task scheduling, so that when the interrupt handler ends, the task which it scheduled (assuming no other is running) is scheduled without delay. Task scheduling from interrupt handlers is a minimal requirement of an RTOS; any RTOS which cannot do this should be regarded as a “toy RTOS”, one to be used for pet home projects, or for students to learn on, but should not be used for any real application.

What would be the problem of using an RTOS which doesn’t support task scheduling from interrupt context? In such an RTOS, if one implemented a networking stack, for example, the bytes received on the networking interface would trigger an interrupt at some point. You might get several interrupts in the course of handling the packet; you might have hardware support which allows DMA movement of the bytes in the packet and end-of-packet detection. In any case, an interrupt handler would detect and end-of-packet condition. At that point, the networking task needs to be notified that has a packet to process and awakened if it is blocked. The delay between end-of-packet detection and when the networking task is awakened can consume a large amount of the total processing latency for the packet. In other words, the CPU sits idle. The sooner it is put to work, the better.

Now, if an RTOS doesn’t support task scheduling from interrupt context, the app developer must solve the task scheduling problem some other way, and this other way will have some undesirable side-effect. One way to solve the task scheduling problem is to have the interrupt set an end-of-packet flag and require that the app code poll this flag. We’ll assume that the flag is polled at the same period as the OS clock tick. If the OS clock is configured at a 10 millisecond interval, then the add-on latency due to the polling will be a random value from 0 to 10 millisecs. This interval could be shortened by decreasing the OS tick interval. But the point is that the app developer inherits this problem if he or she must implement a networking stack on an RTOS which doesn’t support task scheduling from interrupts.

But making NUFR support task scheduling from interrupt context increases NUFR code complexity by a quantum magnitude. NUFR must handle all the corner cases, all the ramifications of allowing interrupts to schedule tasks. This is the main reason why NUFR does the bulk of its kernel work in critical sections. The critical sections must be kept to a minimum, so that NUFR maintains “deterministic” (a favorite word used by other RTOS vendors) lock intervals. Deterministic in this context means that there cannot be any corner-cases where a critical section consumes an enormous amount of CPU time. To achieve this, NUFR takes advantage of the fact that a codebase having multiple tasks will invariably choose one task priority as the nominal priority and most tasks will be assigned to this priority. NUFR enlists the support of the app developer to assign this priority value, so that the kernel can optimize ready list inserts accordingly.

And towards the goal of deterministic interrupt locks, each task maintains separate message queues, one for each priority. If there were a single queue, each message send would have to walk the message queue in order to maintain the priority sort order when doing an insert. There could be a dozen, two dozen—any number of messages on a queue. A queue walk on a list like that is indeterminist. NUFR’s solution is to have one queue per message priority, and to limit the number of message priorities to four. In addition, the system developer is encouraged to limit the number of message priorities on small systems.

There is one indeterministic area which NUFR hasn’t addressed (but might in future releases): a semaphore’s task wait list requires a brute-force walk, unlike the ready list. But that’s all.

One can see the ripple-effect that NUFR’s support of task scheduling from interrupt handlers causes throughout the entirety of the NUFR architecture. What’s described here doesn’t include the corner-cases which must be addressed. Every corner-case must be covered—missing a tiny window can cause an infrequent system crash or lockup. This is the most feared problem of any OS, for it’s the lockups and crashes which happen infrequently which are the hardest to root-cause. Stability in an RTOS is the most sought-after attribute the RTOS can offer. The more sophisticated the features that an RTOS has to support, the more complex the kernel code becomes, and with this complexity comes an increasing risk of introducing a bug. But a lack of the essential features a kernel must support pushes the complexity out to the application layer. The app developer calls on the RTOS to solve certain problems for himself or herself. If the RTOS cannot do this, than the app developer himself/herself must solve these problems, and app layer may not be the best layer to solve the problem. Even if it were, the solution is one which must be developed tested and proven. When folks shop for an RTOS, they’re shopping for a piece of code which makes their application development easier, which offers services that they would have to develop themselves.

The assumption is that any service or feature offered by an RTOS has been vetted, so the developer can be assured that he/she is building on a solid foundation. This is understood. In practice, however, problems arise when an app developer misunderstands how an RTOS feature works, or doesn’t know which among several features is the most prudent to use to solve a known problem. Developers even suffer from a lack of understanding to the degree that they sometimes don’t even know a problem exists that has to be solved. They write app code and open up a timing hole, a memory leak, or something of that nature. These problems in the application layer can take a system down just as easily as a problem in the RTOS.

So NUFR offers a full feature set so that the app developer has the features and services that he/she needs to solve the problems that he/she will encounter. NUFR goes a step further and restricts the number of features so that app developers aren’t overwhelmed by the variety and complexity of stuff that’s packed in the OS. For example, NUFR strongly encourages the app developer to have a task come to idle by blocking on a message receive event. The message queue is the default foundation around which a task is built. In this way, NUFR steers the developer away from the morass of semaphore manipulation and towards the greener pastures of event-driven messaging. Naturally, messaging is not a catch-all to fix every embedded tasking problem. But it’s a good starting point.

Thought and planning was put into NUFR’s messaging architecture, and specifically how it can be implemented efficiently on a small, resource-limited platform. Offering advanced features and services and designing them so that they’ll scale down to a small platform (and scale up too) makes NUFR compare favorably against other RTOSs. Other RTOSs can do many or more features than NUFR, but they can’t do them on a resource-limited platform. Other RTOSs work on resource-limited platforms, but omit the features which app developers need.

There are a few more features that NUFR offers which were difficult to implement. One feature is the timeout on API calls. Timeouts are essential. Handling them in an RTOS is a pain in the neck (grep for the term *zombie timer* in the NUFR kernel, and one will see why). The problem with timeouts is more extensive than what first appearances would indicate. Timeouts of bop waits necessitated the use of bop keys and pre-arrived bops. I’ll leave that for another discussion.

Semaphore priority inversion was another difficult feature to add. Message aborting was another. Message abort is part of the task kill feature. Task killing is akin to linux process catching a signal. Catching signals in a linux environment is tricky. It’s tricky in NUFR also.

Whether anyone will every use the task killing feature or the local struct feature is unknown. One might criticize NUFR for having these unnecessary features, and the criticism may be warranted. But since the programming community at large is slow to critique for too many features, I don’t expect the criticism to be forthcoming.

## Bops

Bops are unique to NUFR, so they’re given a unique name so as not to confuse them with anything else (and because it’s a short, three-letter unique name...practical at the expense of appearing a bit silly). Bops will probably be used most in one of two cases: first, in very resource-constrained systems, like on systems which have less than 16k FLASH and/or less than 4k RAM, or on 8- or 16-bit CPUs. Second, as building blocks for services. For example, synchronous message sends or pipes would likely use bops in their implementation. While bops may be replaced with semaphores in many cases, semaphores are not always the best solution.

The bop keying pre-arrival features are there to cover corner cases in bop operations. One might have to think long and hard to appreciate the necessity of these features, but without them, there would be holes in certain corner-cases that bops couldn’t cover. Semaphores, since they have counting capability, have inherent pre-arrival built into them. But semaphores don’t have any keying. Keying is needed if a bop wait times out and another wait is started. The key distinguishes the two wait instances, to the task sending the bop is assured it’s not sending a bop to a wait instance which expired. It’s a tricky corner-case, but one which must be covered.

Bop keys, bop locking, and local structs are intended to be used in conjunction with each other. Refer to the example for synchronous message sends. Synchronous messaging is fraught with difficult corner-cases. But synchronous messaging is of such importance at the app level that the NUFR features specified here were added specifically for this use-case, so that the app developer could build solid synchronous messaging.

## Semaphores

Semaphores are so commonplaces in OSs and RTOSs that NUFR must implement them, and implement them according to common practice. But NUFR only intends for them to be used in a couple of corner cases—though there’s nothing preventing a developer from using them like in his or her previous experience with another RTOS. The use-cases that NUFR semaphores target are two-fold: first, semaphores are, in essence, mutexes. Mutexes are necessary, and had not NUFR included semaphore support, there would be no clean way (spin loops aren’t “clean”) to implement them in NUFR. Semaphores are the only OS objects in NUFR which allow multiple tasks to be blocked on them.

The second use-case which semaphores address is the counting semaphore. The SL uses a counting semaphore in its pool management service. Again, had not NUFR had kernel-level semaphore support, there would’ve been no other clean way to solve the pool management problem. And pool management is important.

## Messaging

The messaging design went through multiple mental iterations of rewrites. Out of that were birthed the key elements to the design.

First, message blocks were made to be small. Really small—12 bytes. Purposely word-aligned. The message fields word packs a lot of information. One reason why message priorities are limited to four levels is that this that there was little room left in the message fields word. The expectation is that four message priority levels, with one reserved for an abort message, leaving three, is enough on high-end applications. Nobody in the field of computer science likes limiting anything, but message priorities are limited. The limitation wasn’t solely because of the bit limitation, but also CPU cycles required on a message get operation. Rather than add in the complexity needed to keep track of the highest priority queue from which to draw the next message, the *nufr\_message\_get()* does a brute-force walk of all queues to find the first available message. Without limiting the number of message priority levels, this walk could become time consuming. Another reason for doing a brute-force walk of all queues when receiving message is that it’s faster on average, especially if there are fewer priority levels configured.

The 12 bytes that a message block consumes means that a resource-limited system can have a message block pool size as large as—say—20 blocks, and the blocks would only consume 240 bytes. The ability to have relatively large block pools on small systems is insurance against the dilemma of running out of blocks in some app corner-case. The easiest way to prevent an out-of-block condition is to supply more blocks than can ever possibly be needed, in any corner-case. A compromise is struck in block size by providing the ability to attach a memory block to any message. So if an app developer wants to include app-defined fields in each message, he/she can create a service of his/her own which adds extension buffers to each message. And, incidentally, this is one use-case for the SL pool manager.

For the sake of interrupt handlers, message block fields bit fields can be assigned static values, rather than having to do bit-packing in an interrupt handler. This optimization cuts down on the interrupt processing latency, so interrupt handlers that need to send a task a message, thereby scheduling the task, can do this with an economy of CPU cycles.

Second key design attribute of the messaging feature is that message block allocations are fast. This goes hand-in-hand with having small message block sizes. If necessary, even on slow CPUs, message blocks can be allocated by interrupt handlers, though this isn’t recommended practice. The message block pool is put over into the NUFR platform component so the system developer can have some control over it. If for no other reason, the freed block list contains valuable information about the recent event history of the system, so the system developer may choose to extract this information during a system dump. The freed blocks have intact the message IDs. The message blocks free list forms a queue whereby freed blocks are enqueued to the tail, while blocks are allocated from the head. There is a wealth of information retained in this pool, as the messages are queued in time sequence. Another reason for making message blocks tiny is that the larger the block pool size is, the more debug information is retained, to be used during a crash. The freed queue acts as a sort of log.

Having a central message block pool, rather than dividing it up per task necessitates the use of critical sections, so that that the block allocation and freeing is reentrant: any task and even interrupt/exception handlers can allocate message blocks without fear of colliding with one another. But having a central block pool is the most efficient use of block RAM, as central pools use RAM more efficiently than dedicated pools. Also, the central pooling makes for a better message block free queue debug-thing.

The third key design attribute is that the kernel message send and get APIs have a curtailed set of features. Messaging scenarios get complicated, and it’s best to deal with this complication in the SL rather than in the kernel. The kernel therefore supplies the bare minimum messaging support. It supplies it efficiently. Message sends and receives don’t require a lot of CPU cycles, and messaging on the whole doesn’t consume a large amount of RAM. On weak CPU systems—battery-powered systems in particular—this is essential. One cannot afford to have messaging consume much CPU time. The solution therefore was to push the complicated messaging scenarios to the SL, so the app developer can decide himself/herself over the tradeoff between efficiency and features. Even the SL has a curtailed feature set, as messaging can get even more complicated than that. In fact, it would come as no surprise if a system or app developer decided to create his/her own SL messaging-equivalent, and use that in parallel or in place of the NUFR SL messaging component. It’s better that he/she decides this rather than if he/she is confined by a NUFR design. And this is one reason for having a SL, placing key RTOS functionality outside the kernel: it makes it easier for the system or app developer to modify or replace that functionality with something of their liking and their choosing. NUFR is quite flexible and goes to lengths to stay out of the developer’s way.

The fourth key design attribute is that each message makes available both a message prefix and a message ID (or the possibility of concatenating these into one). Having two ID-type fields opens more possibilities to the app developer; so does the fact that the NUFR kernel takes no interest in either prefix or ID. It’s crucial in an OS—or most any software component for that matter—to have a solid layer architecture in place. A solid layered architecture has the right layers, and each layer does what it needs to do, no more, no less. The OSI model is a good example of well-designed layered software architecture. RTOSs need be judiciously layered as well. But how many RTOSs out there are?