NUFR Manual

For NUFR Version 1.03

Copyright © 2018, 2019, 2020 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 HoldingsTable of Contents

[Feature Set 5](#_Toc31028895)

[NUFR Components and Layers 6](#_Toc31028896)

[Services Offered by the SL 7](#_Toc31028897)

[Remarks about this Manual 8](#_Toc31028898)

[Configuration 9](#_Toc31028899)

[Sanity Checking File 9](#_Toc31028900)

[Compile Switches 10](#_Toc31028901)

[Compile Switch Instructional 11](#_Toc31028902)

[Including the Service Layer 12](#_Toc31028903)

[Hooking Up NUFR Exception Handlers 13](#_Toc31028904)

[OS Tick 13](#_Toc31028905)

[Tick Setting Instructional 14](#_Toc31028906)

[Extended Usage of the OS Tick 14](#_Toc31028907)

[Tick-less System Instructional 14](#_Toc31028908)

[NUFR Interrupt Lock Level Setting 15](#_Toc31028909)

[Message Block Pool Size 15](#_Toc31028910)

[Sizing the Main Stack and the Background Task Stack 16](#_Toc31028911)

[The Import and Export Header Files 17](#_Toc31028912)

[Preparing Main() for the BG Task 18](#_Toc31028913)

[OS Tick 18](#_Toc31028914)

[Tick Setting Instructional 18](#_Toc31028915)

[Configuring Application Settings 21](#_Toc31028916)

[Defining Tasks 21](#_Toc31028917)

[Defining Task Priorities 21](#_Toc31028918)

[Task Entry Points 22](#_Toc31028919)

[Task Definition Instructional: Correct Config 22](#_Toc31028920)

[Task Definition Instructional: Incorrect TID enums 24](#_Toc31028921)

[Task Definition Instructional: Incorrect TID/Descriptor Pairs 25](#_Toc31028922)

[Semaphores, Mutexes, and the SL Semaphore Pool 26](#_Toc31028923)

[Semaphore Definition Instructional 26](#_Toc31028924)

[Semaphore Definition: Incorrect Examples 27](#_Toc31028925)

[Mutex Definition Instructional 27](#_Toc31028926)

[Semaphore Pool Reservation Instructional 27](#_Toc31028927)

[SL Particles 28](#_Toc31028928)

[Particle Configuration Instructional 28](#_Toc31028929)

[SL Pool 29](#_Toc31028930)

[SL Timers 29](#_Toc31028931)

[Tasks 30](#_Toc31028932)

[Definition of Terms 30](#_Toc31028933)

[Task Activation and Termination 32](#_Toc31028934)

[Task Scheduling and Switching Algorithms 32](#_Toc31028935)

[Task Scheduling from Interrupt Handler 33](#_Toc31028936)

[Tasking Instructionals 34](#_Toc31028937)

[Setup 34](#_Toc31028938)

[Main Startup and Background Task Instructional 36](#_Toc31028939)

[Task Switching Instructional 38](#_Toc31028940)

[Multiple Task Instances Instructional 40](#_Toc31028941)

[Bops 42](#_Toc31028942)

[Bop Keys 43](#_Toc31028943)

[Bop Message Abort Feature 44](#_Toc31028944)

[~~Bop Pre-Arrival~~ 44](#_Toc31028945)

[~~Bop Pre-Arrival Instructional~~ 45](#_Toc31028946)

[Local Structs and Bop Locking 45](#_Toc31028947)

[Bop Locking and Local Struct Instructional 46](#_Toc31028948)

[Semaphores, Mutexes, and SL Semaphore Usage 49](#_Toc31028949)

[Semaphore Initialization and Priority Inversion Protection 49](#_Toc31028950)

[Semaphore Algorithms 50](#_Toc31028951)

[Semaphore Ownership 51](#_Toc31028952)

[Mutexes 51](#_Toc31028953)

[The SL Semaphore Pool 51](#_Toc31028954)

[Semaphore API Calls from Interrupt Handlers and BG Task 52](#_Toc31028955)

[Messaging 53](#_Toc31028956)

[Message Prefixes and Message IDs 53](#_Toc31028957)

[Message Priorities 54](#_Toc31028958)

[Message Optional Parameter 54](#_Toc31028959)

[Message Sends from Interrupt Handlers 55](#_Toc31028960)

[Message Send from Interrupt Handler Instructional: No Inline Send 55](#_Toc31028961)

[Message Send from Interrupt Handler Instructional: Inline Message Send 57](#_Toc31028962)

[Message Block Allocation 58](#_Toc31028963)

[Message Block and Message Send Instructional 58](#_Toc31028964)

[Block Allocation in a Real System 60](#_Toc31028965)

[Message Prefixes and IDs in the SL 61](#_Toc31028966)

[SL Lookup Based on Message Prefix 65](#_Toc31028967)

[Message Abort and Task Kill Features 67](#_Toc31028968)

[Message Abort 67](#_Toc31028969)

[Message Abort Instructional 67](#_Toc31028970)

[Task Kill 68](#_Toc31028971)

[The TYWYLH Message 69](#_Toc31028972)

[SL Pool Manager 71](#_Toc31028973)

[Particles 73](#_Toc31028974)

[Configuring and Initializing SL Particles 73](#_Toc31028975)

[Basic Particle API Reads and Writes 73](#_Toc31028976)

[Particles in Exception Handlers 75](#_Toc31028977)

[App Timers 77](#_Toc31028978)

[Clock Source 78](#_Toc31028979)

[Hooking Up Clock Source to App Timer Component 78](#_Toc31028980)

[Synchronous Messaging 79](#_Toc31028981)

[Pipes 83](#_Toc31028982)

[Exported Defines, Macros, and APIs 86](#_Toc31028983)

[Defines and Macros 86](#_Toc31028984)

[API Functions 88](#_Toc31028985)

Feature Set

NUFR is a Real Time Operating System (RTOS). The major features of NUFR are:

* Prioritized tasks with preemptive and round-robin multitasking
* Event-driven messaging
* Prioritized messaging
* Messaging to single or multiple tasks
* Full-featured semaphores
* Priority inversion protection for semaphores
* Lightweight semaphore-like task control objects (*bops*)
* Mechanism for sharing a stack-resident data structure between two tasks
* Timeout option on API calls
* Ability to run without an OS tick
* Ability to make kernel API calls from interrupt level
* Task killing
* Mutexes
* Task timers
* Generic memory pool
* Particle-based memory pool
* Specialized synchronous messaging capabilities
* A wide range of scalability between platforms and applications
* Can be configured to use a tiny amount of resources
* Layered architecture
* Hooks for platform customizations

# NUFR Components and Layers

In order to facilitate scalability, extensibility, and configuration, NUFR is divided into components and layers. The layers and components are:

**NUFR Kernel**

The kernel manages the basic tasking, the OS kernel objects, and the OS tick. It is accessed by API calls, and some or most of these APIs can be called from any other NUFR component or NUFR layer, from the application code, or from an interrupt handler.

**NUFR Platform Component**

The kernel is divided into two pieces, the second piece being the *Platform Component* or alternatively called the *Platform Layer.* The Platform Layer is the customizable portion of the kernel. The intention is that any portion of the kernel that the system developer might need to customize is located in the Platform Layer rather than in the kernel itself. The Platform Layer is not to be confused with the BSP or any driver; it is entirely different.

**Platform Application Component**

The Platform Component has a separate piece called the *Platform Application Component* or *Application Layer*. This layer is where tasks are defined, stacks are created, and semaphores are created. Any OS object is defined in this layer. The application developer will configure the Platform Application Component.

**Services Layer**

The *Services Layer (SL)* contains features and OS objects that most RTOSs include in their kernel. The NUFR kernel is a micro-kernel, and any functionality which can be placed outside the kernel (the kernel includes the platform layer) is moved outside of it. The SL has features and functionality which the typical RTOS will include in the kernel, and is therefore tightly coupled to it. It rounds out the kernel, feature-wise. The SL saves the application developer the time and energy needed to develop these features for himself or herself.

**Services Layer Application Component**

Similar to the Platform Application Layer, the SL App Component (or *SL App Layer*) contains the application-specific configuration of the SL. For example, mutexes are defined in the SL App Layer. Like the Platform Application Component, the application developer will configure the SL Application Component.

**Application Code**

The code which the app developer writes. Application code makes API calls into any and all of the above components and layers. In the context of this manual, also included in the category of “application code” are the Board Support Package (BSP), drivers, and interrupt handlers—anything which isn’t covered by the layers specified in this list of layers.

Kernel

CPU

Platform App

Service Layer

(SL)

SL App

Application Layer

Platform

Figure NUFR Layering

## Services Offered by the SL

The SL makes available these features:

* Dynamic semaphore pool
* SL message block pool manager (*bpool*)
* SL messaging
* Mutexes
* Generic memory pool manager
* App timers
* Particles

# Remarks about this Manual

Throughout this manual, there are references made to two hypothetical software developers, the *application developer* and the *system developer.* Both are software developers, meaning that they write the firmware for whatever target platform NUFR will be used on. The application developer is the one who writes the code which runs in tasks, or which runs in interrupt handlers. The app developer writes device drivers, up to a point, and occasionally is involved in low-level areas, but the app developer is more concerned with implementing features than with the system as a whole. The system developer, on the other hand, deals with any sort of low-level part of the codebase, which would include the BSP (board support package), and addresses the aspects of device drivers which could affect system performance. The system developer handles all occurrences of source code coded in assembly language, and would oversee how critical sections are used. It is the system developer who would modify any of the code in the NUFR Platform Component, which is outside of the platform app section (and which falls in the app developer’s domain).

In reality, both the application developer and system developer are roles on a development team, and not necessarily individuals per se. These roles may be fulfilled by the same developers or the same group of developers. But comments in this manual are addressed to one or the other of these two pseudo-developers.

Another comment: there are places in this manual where existing files are referred to or where hypothetical files are referred to. In most cases, only a fragment of the file is shown. The file fragment is displayed in this way:

>>>>>>>>>>> *some-file.c* begin >>>>>>>>>>>>>>

void some\_function\_call(void)

{

...

}

>>>>>>>>>>> *some-file.c* end >>>>>>>>>>>>>

Also shown is a set of ellipses (the “...”). In the place of the ellipses the reader should imagine that there are one or several lines of code, the content of which is either not interesting or is left to the imagination of the reader.

# Configuration

Configuring NUFR consists of choosing the settings which will adapt NUFR to the size, scale, and demands of the environment it will run in. It is the system developer’s responsibility to determine and implement NUFR configuration settings. Generally speaking, configuration settings won’t need to be changed between comparable platforms that host different applications—only the NUFR application settings will need to be changed. Configuring application settings, the job of the app developer, is covered in the next section.

Let’s suppose that NUFR will be configured with one of the following goals in mind, and use these goals to guide one’s way through the decision-making of selecting among configuration options:

* A small RAM/FLASH footprint system
* A low-powered system
* An average system
* A full-featured system

For argument’s sake, FLASH storage, in this context, is non-volatile code segment memory—not FLASH used for data or parameter storage. As a rule of thumb, I’d estimate that the cutoff for a small RAM/FLASH system is any processing complex which has less than 4kbytes of RAM. Another rule of thumb is that a system typically requires from 4–8 times the amount of FLASH as it does RAM.

Low-powered systems may likely run off a battery. In these applications, CPU cycles are precious; there’s only so much CPU power the system can consume before it exceeds its energy lifetime. A low-powered system is not necessarily a small RAM/FLASH footprint system, but often the two go hand-in-hand.

A full-featured system might need to utilize some of the more sophisticated features that NUFR offers. One such feature is called *task killing*. This is the ability for one task to kill another task, and would be used when a task has locked up or has stopped working in part or in totality, but which hasn’t (at least, at the time of killing it) crashed the CPU. In certain mission-critical applications, it might be necessary or advantageous to restart just the task which has locked up without restarting the entire software system.

With these system categories in mind, the following sections will step through all the configurations and customizations which NUFR offers or requires. This is a comprehensive list of all things which must be configured to run NUFR; anything *not* included in the following sections does not have to be configured in order for NUFR to run.

## Sanity Checking File

An optionally-used file called *nufr-sanity-checks.c/.h* has been created to do sanity checking on compile-time NUFR configurations, the ones mentioned in this section. Unfortunately, many compile-time settings can only be verified at runtime. This file provides an entry point called *nufr\_sane\_init()*, which is a wrapper for the suite of initialization functions used by NUFR and the SL and the runtime sanity checks.

## Compile Switches

The file *nufr-compile-switches.h* contains all the compile switches that should remain constant for a given nufr-platform model and that are used in any of the NUFR layers or components. There are only a handful of switches.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Switch Name*** | ***Tiny*** | ***Avg.*** | ***Full*** | ***Purpose*** |
| NUFR\_CS\_SEMAPHORE | 0 | 1 | 1 | Include kernel semaphores |
| NUFR\_CS\_MESSAGING | 0 | 1 | 1 | Include kernel messaging |
| NUFR\_CS\_MSG\_PRIORITIES | 0 | 2 | 4 | Number of message priority levels |
| NUFR\_CS\_LOCAL\_STRUCT | 0 | 1 | 1 | Local struct passing feature |
| NUFR\_CS\_TASK\_KILL | 0 | 0 | 1 | Task kill feature |
| NUFR\_CS\_OPTIMIZATION\_INLINES | 0 | 1 | 1 | Optimization to save CPU cycles at expense of codespace |

There is also the file *nufr-platform-app-compile-switches.h.* This file is similar to *nufr-compile-switches.h,* but contains compile flag settings that stand a substantial probability of changing on a project-to-project basis. This file will get created one per project and will reside in the same directory as *nufr-platform-app.h.* It also contains compile switches which the NUFR kernel, platform, and SL layers use.

This file can also be used by the system and application developers to house any compile switches that any developer creates in their project. Since the file is (should be) globally included by all .c files in a project, it is a convenient place to locate any project defines which the developer wishes to propagate throughout the codebase and which he or she doesn’t wish to do this using the make facility.

|  |  |
| --- | --- |
| ***Switch Name*** | ***Purpose*** |
| NUFR\_CS\_WHICH\_PLATFORM\_MODEL | Specify which nufr-platform code we’re using |
| NUFR\_CS\_USING\_OS\_TICK\_CALLIN | Using an application-level OS Tick function |
| NUFR\_CS\_EXCLUDING\_OS\_INTERNAL\_TICKS | Disable code to tick NUFR internally. With this set, nufr\_sleep(), etc. won’t work. |
| NUFR\_ASSERT\_LEVEL | Logging level of NUFR asserts |

The above two tables list the switches, briefly describes their purpose, and gives the recommended setting according to the system size, which is the small footprint, the average, and the full-featured system categories previously listed.

Naturally, for any compile switch setting choice, there are ramifications one way or the other. On a tiny system, turning off all the switches may be the first instinct of the system developer. And that might be the prudent thing to do. But that might not lead to the most optimal solution. On tiny systems, the amount of RAM consumed by task stacks and by task control blocks (TCBs) is a large percentage of the RAM budget. It may be worthwhile, in some cases, to keep messaging enabled, and use the messaging facility to create a single event-driven task, one that combines several features into a single task, rather than to distribute those features among multiple tasks. In other words, the system developer might determine that he or she can invest a little extra in the way of RAM expenditure in the NUFR messaging feature so that he or she can then cut corners on one or more app features, recouping the investment in messaging by means of these cut corners.

On a small system, it’s more likely that messaging is needed than semaphore support. NUFR semaphores are intended to allow multiple tasks to contend for limited resources (mutexes). Simple task blocking and unblocking can be done using bops. Since tiny systems are less likely to have tasks contend with one another, the system developer might just be able to turn off semaphore support without noticing much, if any, of a deficiency. Generally speaking, NUFR messaging is more useful than NUFR semaphores, since NUFR is built around messaging. The system developer should think twice before turning messaging support off, even on a small system. Another alternative to disabling messaging entirely is that by reducing the number of message priority levels, one can recoup some RAM consumption and CPU usage, and a bit of FLASH consumption too. Reducing the number of message priority levels could be beneficial on any sized system, not just on small systems. On low-powered systems, the system developer might try to use as few message priority levels as possible, keeping the number of levels to a minimum, for the CPU savings alone.

The SL is dependent on having both kernel semaphore and kernel messaging support enabled. Therefore, when evaluating resource savings, it makes sense to remove the SL first, then see if removing either semaphore support or messaging support gains you much in terms of resource savings. The system developer may want to replace the SL with one of his or her own, or modify it, or discard the parts he or she doesn’t need.

The other switches, for enabling local structs and for enabling task killing, are for features intended for more sophisticated systems. They don’t consume much in the way of RAM and FLASH, but they do consume a few more CPU cycles, and these CPU cycles go up in scale with the scale of the system. On a low-powered system, if you don’t use these features, don’t turn them on.

### Compile Switch Instructional

The following implements the settings for an average system, the one listed in the table above. Note that NUFR binary compile switches are not the define-exists/define-doesn’t-exist variety, but the define-always-exists variety, and the value is set to 0/1.

>>>>>>>>>>> *nufr-compile-switches.h* begin >>>>>>>>>>>>>>

#define NUFR\_CS\_LOCAL\_STRUCT 1  
#define NUFR\_CS\_MESSAGING 1  
#define NUFR\_CS\_MSG\_PRIORITIES 2  
#define NUFR\_CS\_TASK\_KILL 0  
#define NUFR\_CS\_SEMAPHORE 1

#define NUFR\_CS\_OPTIMIZATION\_INLINES 0

>>>>>>>>>>> *nufr-compile-switches.h* end >>>>>>>>>>>>>

>>>>>>>> *nufr-platform-app-compile-switches.h* begin >>>>>>>

#define NUFR\_CS\_WHICH\_PLATFORM\_MODEL NUFR\_SMALL\_MODEL  
#define NUFR\_CS\_USING\_OS\_TICK\_CALLIN 0

#define NUFR\_ASSERT\_LEVEL 9

>>>>>>>> *nufr-platform-app-compile-switches.h* end >>>>>>>

## Including the Service Layer

The SL consists of a collection of services. Each service is compiled into the codebase separately by including in the makefile the source files needed to build that service. On small FLASH footprint systems, it may be prudent to only include the files for the services which are used and skip the others. The matrix of services vs. the source files that need to be included is specified thus:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | *Dynamic sema-phores* | *Bpool* | *Messaging* | *Mutexes* | *Generic Pool* | *App Timers* | *Particles* |
| *nsvc-api.h* | X | X | X | X | X | X | X |
| *nsvc-app.h* |  | X | X | X |  | X | X |
| *nsvc-app.c* |  |  | X |  |  | X |  |
| *nsvc.c* | X | X | X | X | X |  | X |
| *nsvc.h* | X | X | X | X | X |  | X |
| *nsvc-globals.c* | X | X | X | X | X | X | X |
| *nsvc-mutex.c* |  |  |  | X |  |  |  |
| *nsvc-pool.c* |  |  |  |  | X |  | X |
| *nsvc-timer.c* |  |  |  |  |  | X |  |

The SL services also have a dependency on NUFR compile or configuration options. The table below indicates the NUFR compile flag or configuration vs. the SL service. An *X* indicates a complete dependency on the NUFR compile flag/configuration, whereas a *P* indicates a partial dependency (like for APIs with timeouts).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | *Dynamic sema-phores* | *Bpool* | *Messaging* | *Mutexes* | *Generic Pool* | *App Timers* | *Particles* |
| *OS Tick* |  |  | P | P | P | X | P |
| *Semaphores* | X | X | X | X | X |  | X |
| *Messaging* |  |  | X |  |  | X |  |
| *Local Struct* |  |  |  |  |  |  |  |
| *Task Kill* |  |  |  |  |  |  |  |

## Hooking Up NUFR Exception Handlers

Whatever mechanism on whichever platform is used to do the actual context switch, the NUFR macro *NUFR\_INVOKE\_CONTEXT\_SWITCH* must call, invoke, or otherwise point to the code which does the context switch. On Arm® Cortex® M-series based systems, NUFR does all context switches in the PendSV exception handler. NUFR takes exclusive ownership of the PendSV handler, but the system developer is responsible for hooking up NUFR’s PendSV handler into the interrupt vector table, since NUFR, unlike other RTOSs, doesn’t touch the interrupt vector table.

Specifically, the file *nufr-context-switch.s* contains an Arm® assembly language function called *nufr\_context\_switch().* This function is the entry point for the PendSV handler; this is the function which the system developer must place in the interrupt vector table, wherever that table exists. NUFR also owns the OS Tick handler (but places it in the platform component for extensibility). NUFR requires that the function *nufrplat\_systick\_handler()* in the file *nufr-platform.c* be used in the SysTick Timer exception entry of the interrupt vector table.

Other exception handlers, such as fault handlers and interrupt handlers, are specified and controlled entirely apart from and outside of the domain of NUFR.

## OS Tick

Like most RTOSs, NUFR has an OS tick mechanism. To give a short answer—the OS tick interval is set with the define *NUFR\_TICK\_PERIOD* in *nufr-platform-export.h*. And, yes, NUFR will work with a tick-less OS, meaning that the OS tick portion of NUFR can be completely disabled and NUFR will continue to run, howbeit with intuitive restrictions.

However, there is much to say beyond the short answer, but the discussion is only relevant to those system developers who aren’t certain how to choose the optimal value for their OS tick interval, or those system developers who are working on a low-powered system, and need to acquire a greater understanding and need to command a greater control over the OS tick mechanism. Concerning the OS tick, NUFR offers the system developer a large amount of flexibility and control, so he or she can adapt the OS to his or her requirements, rather than adapting his or her requirements to the OS.

Before deciding anything, one must understand what NUFR uses the OS tick for. NUFR uses the OS tick to implement all the API calls which use any sort of time delay. The call *nufr\_sleep()* uses the OS tick to count the sleep interval. Without the OS tick, *nufr\_sleep()* cannot work, neither will any of the timeout-type API calls. These timeout calls are denoted by a capital letter *T* suffix in the API name, such as *nufr\_bop\_waitT(), nufr\_msg\_getT(), nufr\_sema\_getT(), nsvc\_mutex\_getT()*. Naturally, since the SL timeout calls are built upon NUFR timeout calls, the restriction applies to all SL timeout calls as well.

NUFR’s contract for OS ticking is that the internal function *nufrkernel\_update\_task\_timers()* must be called once each tick interval. If need arises, it can be called from a task, from an exception handler, or from an interrupt handler, rather than from the SysTick handler, but if called from a task, the calling task must run at a priority higher than any other task (at least, higher than any other task which uses timeout calls). Whatever’s used to call the task timer update function, it should call this function at a fixed, predetermined interval—otherwise, the timers will be affected. There’s latitude for creative solutions for low-powered systems, seeing that NUFR has placed the calling environment for the task timer update function in the platform layer, giving leeway to the system developer to manipulate its functionality in interesting ways.

### Tick Setting Instructional

>>>>>>>>>>> *nufr-platform.h* begin >>>>>>>>>>>>>>>

#define NUFR\_TICK\_PERIOD 10

>>>>>>>>>>> *nufr-platform.h* end >>>>>>>>>>>>>>>

The above code snippet informs NUFR that the OS Tick interval is set to 10 milliseconds. The reader should reference the section on startup code to see how *NUFR\_TICK\_PERIOD* gets integrated with the startup code.

## Extended Usage of the OS Tick

By placing the OS tick calling environment in the platform component, the OS tick interrupt handler (on an Arm® Cortex® M-series-based CPU, the *SysTick Timer*) is made available for uses beyond just stroking the OS tick entry point. For small footprint or low-powered systems, significant resource savings can be gleaned. Essentially, the SysTick handler can be used in place of a high-priority, periodically executing task. Invoking a few, or even many, lines of executable code in SysTick, rather than creating a dedicated task that executes periodically, saves the RAM that an addition task would consume and also saves CPU cycles by cutting out the superfluous context switches that the dedicated task would require. The system developer should consider this alongside the considerations for NUFR configuration settings.

### Tick-less System Instructional

At a minimum, removing the call to *nufrkernel\_update\_task\_timers()*, and if the SL is compiled in, the call to *nsvc\_timer\_os\_tick\_callin()*, will cause NUFR to run as a tick-less OS:

>>>>>>>>>>> *nufr-platform.c* begin >>>>>>>>>>>>>>>

void nufrplat\_systick\_handler(void)

{

// nufrkernel\_update\_task\_timers();  
  
// if (NULL != nufr\_sl\_timer\_callback\_fcn\_ptr)  
// {  
// (\*nufr\_sl\_timer\_callback\_fcn\_ptr)();  
// }

}

>>>>>>>>>>> *nufr-platform.c* end >>>>>>>>>>>>>>>

The above may, of course, be accomplished by simply disabling SysTick.

## NUFR Interrupt Lock Level Setting

Unlike many other RTOSs, NUFR uses interrupt locks (i.e., critical sections) extensively. NUFR uses interrupt locks to guarantee that state transitions of the kernel transpire atomically. Like most RTOSs, NUFR permits kernel API calls from interrupt handler contexts, and this is a driving factor for the pervasiveness of interrupt locks throughout the kernel. But in some system performance corner-cases, time-consuming critical sections might cause timing problems. There is a way to mitigate this problem. As Cortex® M-series-based systems are the standard for small processor functionality, an often-overlooked feature of M-series CPUs is their ability to configure interrupts at various priority levels, which includes the ability to disable interrupts by priority level. The interrupt priority level which NUFR’s critical section masks out is configurable; NUFRs “interrupt lock” is not a hard-and-fast lock. So an option is left on the table for a system developer. The question remains, “How does the system developer determine what to set NUFR’s interrupt lock level to? ”

An understanding of the repercussions of misconfiguring or misusing NUFR interrupt level settings will guide the system developer through the decision-making process of selecting the optimal interrupt lock level mask, and will aid in the overall process of assigning interrupt priority levels to interrupt handlers. There are a collection of NUFR API calls which can be invoked from an interrupt handler context. Since most of these result in a kernel state change, and since tasks cause kernel state changes, changes must be handled atomically, hence the use of critical sections. It is the NUFR API calls which interrupt handlers make which must be guarded against; if an interrupt handler doesn’t make NUFR calls, there’s no reason to guard against it. Therefore, if a handler makes no NUFR calls, it can be safely configured to execute at an interrupt priority which is higher than the priority which NUFR locks (masks) interrupts at. Applying this knowledge, the system developer can raise the interrupt priority level of those interrupt handlers which cannot tolerate the interrupt locking latency caused by NUFR. Should those interrupts need to make NUFR calls, there are creative work-arounds, such as using Cortex® tail-chaining to append an SVC exception to an interrupt, and have the SVC interrupt make the NUFR call in place of the high-priority, low-latency interrupt.

## Message Block Pool Size

The define *NUFR\_MAX\_MSGS* in the platform application layer specifies the system-wide total number of message blocks. Message blocks are statically (this behavior can be tweaked, however, in the platform layer) defined and globally shared among all NUFR API clients. The total number of message blocks should be tuned on each platform, according to the needs of the application code. On a running system, there should always be an adequate reserve of free message blocks so that there is always a free block available whenever one is needed. In fact, the message managing facilities in the SL treat an out-of-blocks condition as a near-fatal error.

Message blocks are small, 12 bytes apiece, so there is a minimal penalty in increasing the number of message blocks. The RAM consumed by the number of message blocks needed for both small and large scale systems should be a small percentage of the total RAM budget.

The application developer should write code which uses message blocks properly, returning blocks which aren’t in use to the block pool (*bpool*). An exception to this rule is with interrupt handlers. To cut down on interrupt latency, it is advisable that those interrupt handlers which send messages should have a task pre-allocate one or more message blocks for them, so no blocks have to be allocated while running in interrupt handler context (although NUFR does provide a means for handlers to allocate message blocks).

The SL messaging APIs have built-in message block handling algorithms that provide safety through abstraction. While using these APIs is safer, the system developer for small footprint and low-powered systems may consider saving CPU cycles by cutting out the SL messaging layers, making direct NUFR kernel messaging API calls instead, and also the app developer may consider saving CPU cycles by holding onto message blocks in a task context, so that a task which receives a message will reuse that message’s message block, rather than expending the CPU cycles of freeing the one block and reallocating another. In this and other similar use-cases, the app developer must be careful that the application code has no memory leaks with respect to message blocks, as this will soft-crash the system.

## Sizing the Main Stack and the Background Task Stack

The main stack is the stack which, on an Arm® Cortex M®-based CPU, exception handlers (which includes interrupt handlers) use for their stack while executing. Since NUFR makes use of the Cortex® SysTick handler and the Cortex® PendSV handler, the sizing of the CPU’s main stack (or equivalent stack on non-Arm® CPUs) should take into consideration NUFR’s usage of these handlers. Unlike other RTOSs, though, NUFR has no dedicated kernel stack.

When the firmware boots, the CPU will run on a stack dedicated to the bootup thread, and this stack is different than the main stack. In NUFR, the bootup thread metamorphosises into the Background Task (*BG)*. The system developer must size the bootup thread/BG Task’s stack as well.

SMore detail on the Main/BG Task stacks are included in the Startup section below.Guidance on Startup/CPU-specific Code

*Startup* or *CPU-specific* code (let’s just refer to both as *startup*) is the extra code needed to run NUFR on a real CPU. This code is CPU-specific, so this section is written with one CPU in mind: the ARM® Cortex® M3. Naturally, this will only apply as a guideline for other CPUs.

As much as possible, NUFR decouples the startup code from itself. This is in keeping with the NUFR philosophy of staying out of the developer’s way. It also makes NUFR easier to port and easier to customize by the system developer. But since this subject is of interest to the system developer, it is covered here. Also, the NUFR distribution includes a complete set of startup code, so that the system developer doesn’t have to provide it himself or herself and because source code is the ultimate form of documentation.

Startup code (which by this manual’s definition includes CPU-specific code) consists of the following:

* The CPU reset vectors and start stack
* The exception and interrupt vector tables
* The assignment of exception and interrupt priorities
* The actual code the CPU runs at reset
* The code needed to prepare the C programming environment, which specifically is the initialization of C static and global variables
* The code needed to configure the CPU for the call to *main()*, and therefore the NUFR BG task
* Any assembly language code
* Any compiler-specific code
* The joining of the hardware timer to NUFR’s OS tick handler/SysTick
* The CPU-specific extensions for locking and unlocking interrupts
* The CPU-specific extensions for doing context switches, including PendSV
* Sizing and locating the main (exception) stack
* *nufr\_launch\_task*’s preparation of each task’s stack, prior to the task being launched. This CPU-specific.
* Hooking up or disabling the NUFR assert macros

## The Import and Export Header Files

To decouple NUFR from the startup code, two platform files are controlled by the system developer: the files *nufr-platform-import.h* and *nufr-platform-export.h.* These are simply referred to as the *import file* and the *export file.* These files are written in such a way that they have minimal header file dependencies, so that they can be easily compiled against in any number of source files.

Both the import and export files are part of the platform layer. The platform layer is CPU-all-inclusive. This means that a single instantiation of the platform layer may/will support multiple CPUs. The platform layer is the fine-tuning of NUFR for a given type or sized platform—it is not the adjustments necessary to NUFR to make it work on a different CPU or a different SoC (except for the import file, of course).

The export file contains all the NUFR settings which are needed by the startup code. NUFR’s OS tick interval must be there. Also in the export file is the BG Task’s stack size and stack variable, since the BG stack has startup code implications.

The import file is a mapping of the NUFR settings to the CPU-specific macros, inline functions, or C or assembler functions that implement these. The import file maps the interrupt lock and unlock macros and the context switch invocation macro to the CPU-specific implementation. Since the import file supports multiple CPU and compiler environments, it will be segregated by CPU type, being demarcated by CPU- and compiler-specific compile flags. The import file also maps the stack preparation code handler, so that *nufr\_launch\_task* can be decoupled from CPU-specific stack code.

## Preparing Main() for the BG Task

In a C program, one normally finds a *main()* call. In NUFR, this becomes the BG Task. There are few things that must be set in order before *main()* can be used as the BG task. The system developer has can accomplish this in the way that he or she pleases, but they must happen somewhere.

* The C global and static variables must have been initialized (.bss and .data segments)
* On Arm® Cortex® M-series based systems, the CPU must be in thread mode, thread mode must be set to privileged, and the default stack must be set to use the Process Stack (PSP)
* Interrupts must be enabled

## OS Tick

The OS tick interval is set with the define *NUFR\_TICK\_PERIOD* in *nufr-platform-export.h*. Setting *NUFR\_TICK\_PERIOD* doesn’t configure the Cortex® peripheral register that determines SysTick’s timer interval, since setting that register is done in the startup code, outside of NUFR’s domain.

### Tick Setting Instructional

>>>>>>>>>>> *nufr-platform.h* begin >>>>>>>>>>>>>>>

#define NUFR\_TICK\_PERIOD 10

>>>>>>>>>>> *nufr-platform.h* end >>>>>>>>>>>>>>>

The above code snippet informs NUFR that the OS Tick interval is set to 10 milliseconds. But for completion’s sake, a hypothetical code snippet showing the setting of the SysTick Reload Register integrated with *NUFR\_TICK\_PERIOD* is illustrated (note that *MILISECS\_PER\_SEC* is defined in *raging-global.h*):

void Initialize\_SystemTick(void)  
{  
 SysTick->CTRL = CTRL\_CLKSRC; //use core clock  
  
// SysTick->LOAD = 8000000; /\* Frequency of 1 Hz \*/  
 // 64 bit math prevents intermediate 32 bit overflow by multiply op  
 SysTick->RELOAD = (uint32\_t) ((uint64\_t)\_IMPORT\_CPU\_CLOCK\_SPEED \* **NUFR\_TICK\_PERIOD**  
 / MILLISECS\_PER\_SEC) - 1;  
  
 SysTick->CTRL |= (CTRL\_ENABLE | CTRL\_TICKINT); // enable counting, interrupts  
}

Assuming that the above settings are in place, the behavior of the sleep API call below is to delay by 1 second:

void one\_second\_delay\_wrapper(void)

{

nufr\_sleep(100);

}

Best-practice when specifying the delay in *nufr\_sleep()* is to avoid the form written above and in its place use one of the absolute time abstraction macros provided in *nufr-api.h*, like this:

void improved\_one\_second\_delay\_wrapper(void)

{

nufr\_sleep(NUFR\_SECS\_TO\_TICKS(1));

}

This form is preferable since it’s first of all more readable, then second of all decouples the application code from changes to *NUFR\_TICK\_PERIOD.*

# Configuring Application Settings

Application settings consist of OS objects which can be created and used by applications. In the kernel, these objects are tasks and semaphores. In the Services Layer, these objects are mutexes. The application developer is free to create SL memory pools and SL timers, but these are not considered “application settings” as defined in this section, and are therefore covered elsewhere.

The files specified by the wildcard file specifier \****app****.\** contain the application settings—and contain only application settings. The files which begin with the prefix *nufr* are files in the Kernel Layer or in the Platform Layer. The files beginning with the prefix *nsvc* are Service Layer files. Both the platform layer and the SL have application setting files, the files named *nufr-platform-app.h, nufr-platform-app-compile-switches.h, nufr-platform-app.c, nsvc-app.c,* and *nsvc-app.h.*

## Defining Tasks

To add a new task , the app developer must make the following changes:

* Add a task ID: an enum for that task in the enum *nufr\_tid\_t*
* Create a function definition for the entry point for the new task
* Create a global variable which will be used as the stack task
* Add a task descriptor entry (variable *nufr\_task\_desc*) for the task, filling in the necessary task attributes

There are rules to follow when modifying *nufr\_tid\_t* and *nufr\_task\_desc:*

(a) *nufr\_tid\_t* must have the null and max values (*NUFR\_TID\_null, NUFR\_TID\_max*) kept as the first and last values in the enum. The null must always be assigned the vale of zero.

(b) Integer values must not be assigned to the *nufr\_tid\_t* identifiers. The identifiers must increment by one, which is the default behavior of enums.

(c) The entries in *nufr\_task\_desc* must correspond 1-to-1 with the *nufr\_tid\_t* entries.

### Defining Task Priorities

All tasks are assigned a default priority, and this priority is entered in the descriptor for that task. Priority values are enumerated in *nufr\_tpr\_t,* to be used wherever necessary. Alternatively, the app developer may choose to specify task priorities the old-fashioned way, with positive integers, instead of using the enumerated values. The task priority values which can be assigned to applications range from 2 to 255 inclusive, where 2 is the highest priority and 255 is the lowest priority.

One thing that should be noted: The task priority named *NUFR\_TPR\_NOMINAL* in *nufr\_tpr\_t* is special. It should be assigned to the task priority which has the most number of tasks using this priority. Usually, there are a few tasks which run at a higher or lower priority, so the nominal priority is somewhere in the middle of the task priority distribution. On a hypothetical system of, let’s say, 20 tasks, 14 will be assigned to the nominal priority, 4 will be assigned at a higher priority, and 2 at a lower priority.

NUFR compiles against *NUFR\_TPR\_NOMINAL*, optimizing the kernel to take advantage of the tendency in codebases for a majority of the tasks to run at the same priority, forming a round-robin quorom. If systems have the majority of tasks assigned to a value other than *NUFR\_TPR\_NOMINAL,* there will be no failure, just a loss in performance.

### Task Entry Points

When a task is launched, its entry point is invoked. This entry point is included in the Task Descriptor line for the task. The entry point function must have a single parameter of type *unsigned:*

void foo\_entry\_point(unsigned foo\_parameter)

{

}

The parameter in *foo\_entry\_point* is called *foo\_parameter.* Any value can be passed to the entry point’s parameter at launch time, however, for convenience, the Task Descriptor provides a means of hard-coding a specific value, to be used as the task’s parameter, should the app developer desire to do so. The entry point’s parameter can be used any way the app developer pleases, but it is intended to be used to distinguish instances of tasks which use the same entry point. For example, let’s say that Task A and Task B are drivers for two different USB ports, and share the function *usb\_task()* as their entry point. Task A’s parameter value is set to 1, and Task B’s value is set to 2, where the *1* and *2* specify different USB devices.

### Task Definition Instructional: Correct Config

The below is a correctly prepared configuration for two tasks, Task A and Task B. The configuration has only one task priority enum defined, the mandatory one, *NUFR\_TPR\_NOMINAL*. The app developer prefers to configure task priorities the old-fashioned way in some cases (the value *5* for Task A), and the standard NUFR way in other cases (*NUFR\_TPR\_NOMINAL* for Task B). The tasks reside entirely in their own files, *app-A.c* and *app-B.c.*

Notice how each task has its own TID enum, its own entry in the descriptor table, and that the TID corresponds to the descriptor table 1-to-1. The tasks require their own entry points, but the code “cuts corners” a bit by listing the entry point function prototypes in *nufr-platform-app.h* instead of placing them in *app-A.h* and *app-B.h*, and including these two files in *nufr-platform-app.h*. The tasks just spin in wait loops infinitely. No optional parameter is used.

There are two distinct global variables defined which’ll serve as Task A’s stack and Task B’s stack. These variables are sized by two distinct stack size defines so that Task A’s and B’s stacks can be independently sized.

>>>>>>>>>>> *nufr-platform-app.h* begin >>>>>>>>>>>>>>>

typedef enum  
{  
 NUFR\_TID\_null = 0, // not a task, do not change  
 NUFR\_TID\_TASK\_A,  
 NUFR\_TID\_TASK\_B,  
 NUFR\_TID\_max // not a task, do not change  
} nufr\_tid\_t;  
  
#define NUFR\_NUM\_TASKS (NUFR\_TID\_max - 1)

typedef enum

{  
 NUFR\_TPR\_null = 0, // Do not change. Do not use  
 NUFR\_TPR\_guaranteed\_highest = 1, // Do not change/use

// Must have this enum (can change value, however).  
 // Default priority, most tasks will use this  
 NUFR\_TPR\_NOMINAL = 10,

} nufr\_tpr\_t;

>>>>>>>>>>> *nufr-platform-app.h* end >>>>>>>>>>>>>>>

>>>>>>>>>>> *nufr-platform-app.c* begin >>>>>>>>>>>>>>>

#define STACK\_SIZE\_TASK\_A 150

#define STACK\_SIZE\_TASK\_B 160

uint32\_t Stack\_A[STACK\_SIZE\_TASK\_A/BYTES\_PER\_WORD32];  
uint32\_t Stack\_B[STACK\_SIZE\_TASK\_B/BYTES\_PER\_WORD32];

void entry\_task\_A(unsigned optional\_parameter);  
void entry\_task\_B(unsigned optional\_parameter);

const nufr\_task\_desc\_t nufr\_task\_desc[NUFR\_NUM\_TASKS] = {

{"task A", entry\_task\_A, Stack\_A, STACK\_SIZE\_TASK\_A,

(nufr\_tpr\_t)5, 0},

{"task B", entry\_task\_B, Stack\_B, STACK\_SIZE\_TASK\_B,

NUFR\_TPR\_NOMINAL, 0},  
};

>>>>>>>>>>> *nufr-platform-app.c* end >>>>>>>>>>>>>>>

>>>>>>>>>>> *app-A.c* begin >>>>>>>>>>>>>>>

void entry\_task\_A(unsigned optional\_parameter)  
{  
 UNUSED(optional\_parameter);

while (1)

{

nufr\_sleep(NUFR\_MILLISECS\_TO\_TICKS(200));

}  
}

>>>>>>>>>>> *app-A.c* end >>>>>>>>>>>>>>>

>>>>>>>>>>> *app-B.c* begin >>>>>>>>>>>>>>>

void entry\_task\_B(unsigned optional\_parameter)  
{  
 UNUSED(optional\_parameter);

while (1)

{

nufr\_sleep(NUFR\_MILLISECS\_TO\_TICKS(500));

}  
}

>>>>>>>>>>> *app-B.c* end >>>>>>>>>>>>>>>

### Task Definition Instructional: Incorrect TID enums

All the below code snippets are erroneous. NUFR may compile, but it will crash.

typedef enum  
{  
 NUFR\_TID\_null = 0, // not a task, do not change  
 **NUFR\_TID\_TASK\_A = 5, //ERROR! CANNOT ASSIGN VALUES!**  
 NUFR\_TID\_TASK\_B,  
 NUFR\_TID\_max // not a task, do not change  
} nufr\_tid\_t;

typedef enum  
{  
 NUFR\_TID\_null = 0, // not a task, do not change  
 NUFR\_TID\_TASK\_A,

**//ERROR! NUFR\_TID\_max must always be last!**

NUFR\_TID\_max, // not a task, do not change  
 NUFR\_TID\_TASK\_B

} nufr\_tid\_t;

typedef enum  
{  
 **NUFR\_TID\_null = 1, //ERROR! Must be == 0**

NUFR\_TID\_TASK\_A,

NUFR\_TID\_TASK\_B,

NUFR\_TID\_max, // not a task, do not change  
} nufr\_tid\_t;

### Task Definition Instructional: Incorrect TID/Descriptor Pairs

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

typedef enum  
{  
 NUFR\_TID\_null = 0, // not a task, do not change  
 NUFR\_TID\_TASK\_A,  
 NUFR\_TID\_TASK\_B,  
 NUFR\_TID\_max // not a task, do not change  
} nufr\_tid\_t;

const nufr\_task\_desc\_t nufr\_task\_desc[NUFR\_NUM\_TASKS] = {

**// ERROR! Every TID MUST have an entry**

// {"task A", entry\_task\_A, Stack\_A, STACK\_SIZE\_TASK\_A,

// (nufr\_tpr\_t)5, 0},

{"task B", entry\_task\_B, Stack\_B, STACK\_SIZE\_TASK\_B,

NUFR\_TPR\_NOMINAL, 0},  
};

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

typedef enum  
{  
 NUFR\_TID\_null = 0, // not a task, do not change

**// ERROR! Entries are out of order w.r.t. descriptor!** NUFR\_TID\_TASK\_B,  
 NUFR\_TID\_TASK\_A,  
 NUFR\_TID\_max // not a task, do not change  
} nufr\_tid\_t;

const nufr\_task\_desc\_t nufr\_task\_desc[NUFR\_NUM\_TASKS] = {

{"task A", entry\_task\_A, Stack\_A, STACK\_SIZE\_TASK\_A,

(nufr\_tpr\_t)5, 0},

{"task B", entry\_task\_B, Stack\_B, STACK\_SIZE\_TASK\_B,

NUFR\_TPR\_NOMINAL, 0},  
};

## Semaphores, Mutexes, and the SL Semaphore Pool

To create a semaphore, add an enum to *nufr\_sema\_t.* The enum naming and numbering rules are similar to the rules for task enums; the enums must be numbered contiguously, and *NUFR\_SEMA\_null* and *NUFR\_SEMA*\_*max* are special values, not to be tampered with. Since semaphores are kernel objects, *nufr\_sema\_t* is defined in *nufr-platform-app.h.* Mutexes, on the other hand, are Service Layer (SL) objects, and are therefore enumerated in *nsvc-app.h,* and not *nufr-platform-app.h.* Mutexes use the enum *nsvc\_mutex\_t*, and like the other object enums, the first mutex starts at 1 and subsequent mutexes increment by 1. *NSVC\_NUM\_MUTEX* must be incremented each time a mutex is added.

There are a couple of additional magic enums in *nufr\_sema\_t* named *NUFR\_SEMA\_POOL\_START* and *NUFR\_SEMA\_POOL\_END.* The SL compiles against these values, using these to create a static pool of semaphores from which SL objects which require a semaphore draw from. The list of SL objects which require semaphores is:

* Mutexes (*nsvc-mutex.c*). One semaphore per mutex.
* The SL pool (*nsvc-pool.c*). One semaphore for each pool implementation.
* The particle pool (*nsvc-pcl.c*). One semaphore for the entire pool (the pool is actually an SL pool instance).

The number of semaphores in the SL semaphore pool can be calculated by this equation:

*pool-size = NUFR\_SEMA\_POOL\_END - NUFR\_SEMA\_POOL\_START + 1*

The current NUFR release (and foreseeable future releases) has only statically allocated SL objects which require semaphores from the SL pool. This means that semaphores are allocated from the pool, but never get freed back to the pool. The allocation is a one-time allocation at during SL initialization.

### Semaphore Definition Instructional

To create a semaphore, add an enum for it in *nufr\_sema\_t*. In this example, a single semaphore named *NUFR\_SEMA\_FOR\_ME* has been added. In this example, there is no SL semaphore pool defined

>>>>>>>>>>> *nufr-platform-app.h* begin >>>>>>>>>>>>>>

typedef enum  
{  
 NUFR\_SEMA\_null = 0, // not a sema, do not change  
 NUFR\_SEMA\_FOR\_ME,

NUFR\_SEMA\_max // not a sema, do not change  
} nufr\_sema\_t;

#define NUFR\_NUM\_SEMAS (NUFR\_SEMA\_max - 1)

>>>>>>>>>>> *nufr-platform-app.h* end >>>>>>>>>>>>>>

### Semaphore Definition: Incorrect Examples

The rules for adding semaphores are similar to the rules for adding task TIDs. Here are some misconfiguration examples. If any of these errors are made, expect a system crash:

>>>>>>>>>>> *nufr-platform-app.h* begin >>>>>>>>>>>>>>

typedef enum  
{  
 NUFR\_SEMA\_null = 1, // **Error!** Must be == 0  
 NUFR\_SEMA\_A = 5, // **Error!** Cannot set to value

// **(..except for SL sema pool)**

NUFR\_SEMA\_max // **Error!** Must be last  
 NUFR\_SEMA\_B, // **Error!**

} nufr\_sema\_t;

#define NUFR\_NUM\_SEMAS (NUFR\_SEMA\_max - 1)

>>>>>>>>>>> *nufr-platform-app.h* end >>>>>>>>>>>>>>

### Mutex Definition Instructional

Creating a mutex follows the same guidelines as creating a semaphore, except the file is the SL app file (*nsvc-app.h*), and, naturally, the difference in naming:

>>>>>>>>>>>>>>>> *nsvc-app.h* begin >>>>>>>>>>>>>>>>>

typedef enum  
{  
 NSVC\_MUTEX = 1,   
 NUFR\_MUTEX\_1,

NUFR\_MUTEX\_max   
} nsvc\_mutex\_t;

#define NUFR\_NUM\_SEMAS (NUFR\_SEMA\_max - 1)

>>>>>>>>>>>>>>>> *nsvc-app.h* end >>>>>>>>>>>>>>>>>

### Semaphore Pool Reservation Instructional

There are three magic enums (*NUFR\_SEMA\_POOL\_START*, *NUFR\_SEMA\_POOL\_END* and *NUFR\_SEMA\_POOL\_SIZE*) compiled against by the SL semaphore pool. These are the static reservations semaphores from which the pool draws.

To create this semaphore pool , assign the “start” magic number the same way any semaphore would be assigned, then assign the “end” magic number after that, assigning it to the start plus one. The “pool size” enum is the difference plus one. In the example below, 11 semaphores are reserved for the pool:

>>>>>>>>>>> *nufr-platform-app.h* begin >>>>>>>>>>>>>>

typedef enum  
{  
 NUFR\_SEMA\_null = 0,   
 NUFR\_SEMA\_X,  
 NUFR\_SEMA\_Y,  
 NUFR\_SEMA\_Z,  
 NUFR\_SEMA\_POOL\_START, // fixed enum name, used by SL

// fixed too  
 NUFR\_SEMA\_POOL\_END = NUFR\_SEMA\_POOL\_START + 10,  
 NUFR\_SEMA\_max

} nufr\_sema\_t;  
  
#define NUFR\_NUM\_SEMAS (NUFR\_SEMA\_max - 1)  
  
#define NUFR\_SEMA\_POOL\_SIZE \

(NUFR\_SEMA\_POOL\_END - NUFR\_SEMA\_POOL\_START + 1)

>>>>>>>>>>> *nufr-platform-app.h* end >>>>>>>>>>>>>>

## SL Particles

The *SL Particle Pool* is configured by the defines *NSVC\_PCL\_SIZE* and *NSVC\_PCL\_NUM\_PCLS* in *nsvc-app.h.* These two enums specify the number of particles in the pool and the size of the memory buffer size which each particle offers.

### Particle Configuration Instructional

This code snippet creates a pool of 10 particles. Each particle can hold 100 bytes of data:

>>>>>>>>>>> *nsvc-app.h* begin >>>>>>>>>>>>>>

#define NSVC\_PCL\_SIZE 100  
  
#define NSVC\_PCL\_NUM\_PCLS 10

>>>>>>>>>>> *nsvc-app.h* end >>>>>>>>>>>>>>

## SL Pool

Since SL pool instances are allocated at runtime, and not at compile time, these are not considered part of application configuration. See SL pool section.

## SL Timers

The configuration settings for SL timers consist of the enum *nsvc\_tm\_divisor\_t* and *NSVC\_TMDIV\_NUMBER,* both in *nsvc-app.h.* These are used to specify *SL Timer Divisors,* or simply *divisors.* A divisor is a fixed ratio of OS ticks that form a secondary tick value upon which SL timers are decremented. Divisors are primarily used to save CPU cycles. The CPU cycles savings occurs in the SysTick Timer handler, which runs at a priority higher than any task. Saving CPU cycles in the SysTick handler also results in savings in task scheduling latency.

The values in the divisor enum *nsvc\_tm\_divisor\_t* must comply with the rules used for other OS object definition enums, such as task IDs, but with a few exceptions. Divisor enums start at zero (other OS object enums don’t start at zero) and increment by one and only by one. The magic enum *NSVC\_TMDIV\_max* must always be the last enum. Each time a divisor is added or removed, the define *NSVC\_TMDIV\_NUMBER* must be readjusted to be equal to the total number of enums specified in *nsvc\_tm\_divisor* but not including *NSVC\_TMDIV\_max.*

Since the allocation and creation of timers occurs in the applications themselves and not in the SL, timer allocation and creation is not covered in this section.

# Tasks

## Definition of Terms

If one defines a thread conceptually as a contiguous execution path, one that may be paused and resumed (some threads, not all), that uses one and only one stack, on an Arm® Cortex® M-series-based CPU, the following constitute threads:

* Each interrupt handler
* The SysTick Timer handler
* The PendSV software exception handler
* Each fault handler
* The SVC software exception handler (not normally configured on NUFR)
* The bootup thread, which becomes the NUFR Background (BG) Task
* Each NUFR task

The principal responsibility of any RTOS is the management of tasks. A *task* is a single executable thread whose stack and static variables reside in a common address space accessible by all tasks. RTOS tasks serve the same role as processes and threads in memory managed OSs like Unix.

In most cases, NUFR tasks conform to the boilerplate set of behavior and functionality used by most every RTOS. There are well-known terms used to describe this functionality, and for the sake of clarity these terms are defined here, having been defined in the way that they’re used throughout this manual and throughout the NUFR source code. Keep in mind, these definitions may diverge from their use elsewhere in the software community:

**Blocked State** (Or simply *Blocked)* State of a task when it is paused in execution. In the blocked state, the task is dormant.

**Blocking** *Blocking* is the transition of a task from the ready state to the blocked state. It is also when an OS object of one sort or another causes that task to transition to the blocking state.

**Context Switch** A task transition from running to blocked, and the replacement of a task which is in the running state to the blocked state. The task *context* is the state of the stack which the task is running on and what the task’s stack pointer is set to, which specifies the task’s function call stack.

**Launched/Not Launched State** A binary task state indicating the obvious—whether a task has been launched or not launched. A task is in one of these two states. Stacks are statically defined and default to Not Launched upon initialization.

**Multitasking** The ability for the multiple tasks of an RTOS to share a single CPU by cooperatively blocking and scheduling in such a way that all tasks are able to fulfill their application contracts. Multi-tasking is also the illusion created from the user’s perspective that multiple tasks are executing concurrently, when in fact they share the CPU, being rapidly switched in and out.

**Preemption** The action in which a task at a higher priority becomes scheduled, replacing the running task, which in this case would have a lower priority. The higher priority task *preempts* the lower priority task.

**Ready List** The list of all ready tasks.

**Ready State** (Or simply *Ready)* State of a task when it desires to become the running task, or is the running task. A launched task at any given time is either ready or is blocked, but not both.

**Reentrancy** *Reentrancy* is when a first thread invokes a function, then that thread is blocked before it has completed the function, and a second thread (or multiple threads) invokes the same function. A function is said to be *reentrant* if it is designed to accommodate reentrancy. Reentrancy is not to be confused with recursion; recursion is when a thread invokes a function and, before completing that function, from within that function invokes that same function a second time (or multiple times).

**Round-Robin** The algorithm the RTOS uses to schedule a single task from among multiple tasks which are ready and which are of the same task priority, which for NUFR is intended to be (but does not have to be) the NUFR nominal priority (*NUFR\_TPR\_NOMINAL*). It also means the cooperative sharing of the CPU among tasks of the same priority.

**Running Task** The task which is the current executing task. There can be only be one running task at any given time.

**Scheduling** The act of changing a task from the blocked or the ready state to become the running task. Also, a task which is s*cheduled* means that that task is the running task.

**Task Context** A task stack’s data values. Also, *context* by itself means the task from which a function was called, or the exception handler or BG Task from which a function was called.

**Task Killing** Given a Task A and a Task B, a task kill occurs when Task A terminates Task B.

**Task Priority** An integer value assigned to each task which the RTOS uses to determine which becomes the running task when multiple tasks are in the ready state. The RTOS uses task priorities to determine where in the ready list to insert a task, thereby maintaining the ready list in a task priority sort order. Note that when one says in says in general “high task priority”, he or she means that the actual task priority number is lower, not higher, since the lower a task’s task priority number is, the higher that task’s priority is.

**Task Switching** When one task ceases to be the running task, having been replaced by another task which becomes the new running task.

**Task Termination** When a task returns to the unlaunched state. Only the task itself can terminate itself.

## Task Activation and Termination

Tasks are statically allocated: they’re created at compile-time and cannot be created at runtime, nor can they be deleted. Tasks, however, are launched and terminated. Before a task is launched, it’s in an inactive state. Any messages sent to the task while in this state or any attempts to send a bop will be ignored.

Tasks are launched by calling *nufr\_task\_launch().* Tasks can be launched from any other task, including the BG Task and can be launched from the OS Tick (SysTick) handler, should the platform component be modified to do so. Tasks could conceivably be launched from an interrupt handler, but this is not advisable due to the length of time interrupts would be locked during the launch interim. A task, upon being launched, is added to the ready list. The launch ready list insertion follows the same rules as any other ready list insertion, so if the task being launched is at a higher priority than the task invoking the launch, a context switch will occur, as the task being launched is scheduled, preempting the launcher.

A task terminates when the task’s entry point function executes a *return* statement. Of course, simply running off the end of a function is the equivalent of a *return*. When a task terminates, the kernel schedules the next ready task.

## Task Scheduling and Switching Algorithms

There are no surprises in the algorithms by which NUFR schedules tasks. To begin with, tasks, whether blocked or ready, have a task priority. This priority is assigned by default to the value specified in the task descriptor table. Afterwards, the task priority can be changed, explicitly or implicitly. The task priority is a positive integer, and the lower the value of this integer, the greater the task priority. So if Task A has a task priority of 5, and Task B has a priority of 6, Task B has a higher priority than Task A. If both tasks are ready, the kernel will schedule Task B before it schedules Task A.

The kernel maintains a ready list, and the tasks appear listed in the ready list in the order in which the kernel will schedule them. Also, the running task sits at the head of the ready list. If there are no tasks that are ready, the ready list will be empty, and the BG Task is scheduled. This means that, unlike application tasks, the BG Task when scheduled does not show up in the ready list, which also means that the ready list is always empty when the BG task is scheduled. The CPU must always have a thread to execute; without a task to execute it will crash (if not put into a hardware sleep mode, if sleep mode is available). The BG task does—must –fulfill this role. The BG task can make NUFR API calls, but none of these calls can result in the BG task blocking. If it were permissible and if the BG task were to call *nufr\_sleep(),* the system would crash.

There are a number of events which can cause a task to unblock and thereby become ready. When a task is unblocked, this algorithm is applied:

* The ready list is walked looking for the first task which has a lower task priority than the unblocking task
* The task which is being unblocked is placed in the ready list just before the first task in the list which is at a lower priority
* If the ready list was empty, the unblocking task becomes the sole task on the ready list
* If there are other tasks on the ready list that are at the same priority of the one being inserted, the steps above dictate that the unblocking task will be placed in the ready list behind the last task of the same priority
* Given the above algorithm, the unblocking task may become the ready list head by virtue of having a higher priority than the previous head or by virtue of the ready list having been empty
* If the unblocking task becomes the ready list head, the unblocking task is scheduled.
* If the unblocking task is scheduled, the previously scheduled task stays in the ready state, but is pushed back to second place. In this case, the previous running task was preempted by the unblocking task.
* Any action where the unblocking task becomes the running task constitutes a context switch, even if the running task was the BG task

## Task Scheduling from Interrupt Handler

A valuable feature, one which most RTOSs offer, is the means for an interrupt handler to make an RTOS API call that will cause a task to unblock, scheduling the task if the scheduling rules determine so. Once the interrupt handler terminates, the task which the handler unblocked (assuming it’s the ready list head task) is immediately scheduled. NUFR also supports this feature, and the sequence of events happens as such (described for an Arm® Cortex® M-series-based CPU):

* (Let’s assume that the interrupt handler itself is not preempted by a higher-priority interrupt)
* While an interrupt handler is executing (CPU in exception mode), the handler makes a NUFR API call.
* The API call executes in the interrupt handler’s context
* In this use-case, the API call causes the task to unblock. While unblocking, the task is placed on the ready list.
* If the task is to be scheduled, PendSV handler invocation is requested.
* Since the PendSV handler runs as the lowest priority interrupt, nothing immediately happens. The PendSV handler is said to be in a *pending* state, awaiting execution.
* If the SysTick timer expires before the interrupt handler completes, then the SysTick timer is also pending
* The interrupt handler completes
* If the SysTick timer was pending, then it executes. The SysTick timer handler is *tail-chained* (Cortex® terminology) to the end of the interrupt handler.
* The SysTick handler may itself call a NUFR API and cause a task to unblock, and possibly schedule a task
* When the SysTick handler completes (or the interrupt handler, if SysTick wasn’t pending), the PendSV handler runs, being tail-chaining off the end of the SysTick or interrupt handler
* The PendSV handler schedules whatever task is at the head of the ready list, whether it was placed there by the interrupt handler or by SysTick
* The PendSV handler completes. The CPU returns to process mode. The newly scheduled task takes the CPU.

The switching sequence between interrupt handlers, SysTick, and PendSV handlers may be complex, but the important point is that if an interrupt scheduled a task to run, that task will begin running the moment all of the hardware interrupts and software exceptions complete. The latency between interrupt handler termination and the commencement of the scheduled task is as minimal as possible.

The capability that an interrupt handler has to schedule a task and to have that task begin executing immediately (for all intents and purposes) after the interrupt handler completes gives even a resource-constrained computing platform the appearance of having greater computing power than it actually has, relative to non-RTOS OSs. Stated another way, one can achieve respectable packet ping rates on even a small SoC. The system developer should take note of this, and be on the alert for those APIs which can be called from an interrupt handler context. All NUFR APIs are noted as to whether they can be called from interrupt context or not. Unlike some other RTOSs, there is not a special set of APIs which interrupt handlers must use.

## Tasking Instructionals

### Setup

Assume the app settings listed in this section are used throughout the the instructionals which immediately follow this section. The few pieces which are missing are left to the reader’s imagination.

>>>>>>>>>>> *nufr-platform-app.h* begin >>>>>>>>>>>>>>>

typedef enum  
{  
 NUFR\_TID\_null = 0, // not a task, do not change  
 NUFR\_TID\_TASK\_A,  
 NUFR\_TID\_TASK\_B,  
 NUFR\_TID\_TASK\_C,  
 NUFR\_TID\_max // not a task, do not change  
} nufr\_tid\_t;

typedef enum  
{  
 NUFR\_TPR\_null = 0,   
 NUFR\_TPR\_guaranteed\_highest = 1,   
  
 // Add/delete/change per needs  
 NUFR\_TPR\_HIGHEST = 7,  
 NUFR\_TPR\_HIGHER = 8,  
 NUFR\_TPR\_HIGH = 9,  
  
 // Must have this enum (can change value, however).  
 // Default priority, most tasks will use this  
 NUFR\_TPR\_NOMINAL = 10,  
  
 // Add/delete/change per needs  
 NUFR\_TPR\_LOW = 11,  
 NUFR\_TPR\_LOWER = 12,  
 NUFR\_TPR\_LOWEST = 13  
} nufr\_tpr\_t;  
  
>>>>>>>>>>> *nufr-platform-app.h* end >>>>>>>>>>>>>>>

>>>>>>>>>>> *nufr-platform-app.c* begin >>>>>>>>>>>>>>>

const nufr\_task\_desc\_t nufr\_task\_desc[NUFR\_NUM\_TASKS] = {

{"Task A",

entry\_A,

Stack\_A,

STACK\_SIZE\_A,

NUFR\_TPR\_HIGH,

0},

{"Task B",

entry\_B,

Stack\_B,

STACK\_SIZE\_B,

NUFR\_TPR\_NOMINAL,

0},

{"Task C",

entry\_C,

Stack\_C,

STACK\_SIZE\_C,

NUFR\_TPR\_NOMINAL,

0},  
};

>>>>>>>>>>> *nufr-platform-app.c* end >>>>>>>>>>>>>>>

### Main Startup and Background Task Instructional

All processors boot from the reset vector, and all must, at a minimum, prepare the BSS and Data RAM segments so that a high-level language—*C* in this case—will run as compiled. This means that all the global variables must be initialized to zero by default or to the initialized values specified in the source files, then the call to *main()* may begin. Let us assume that on whatever platform NUFR runs, the system has been properly prepared for the call to *main;* NUFR has no participation is this preparation.

Let’s walk through a NUFR *main()* call. The bracketed-letters *[<letter>]* denote an alphabetically-ordered sequence of events.

>>>>>>>>>>> *main.c* begin >>>>>>>>>>>>>>>

void main(void)  
{  
 const nufr\_task\_desc\_t \*desc\_ptr;

uint32\_t \*top\_of\_stack\_ptr;

bool top\_null;

// [A]: Must do BSP initializations, like powering

// on peripherals, configuring NVIC, setting up

// interrupt vector table, etc.

board\_initializations();

// [B]: This call initializes NUFR, but not the SL

// it must be made before any NUFR call is made,

// therefore it must be called from main().

nufr\_init();

// [C]: These are the initializations for the SL.

// They’re shown here commented-out for reference.

// This example has no SL calls, so no SL init is

// needed. One can make these calls from main() or

// from a task.

//nsvc\_init(NULL);

//nsvc\_timer\_init();

//nsvc\_pcl\_init();

// [D]: Launching a task causes main() here to become

// the Background Task (BG). Since any task is a

// higher priority than the BG, a context switch

// occurs as Task A gets scheduled.

nufr\_launch\_task(NUFR\_TID\_TASK\_A, 0);

// [G]: We arrived here because Task A blocked.

// If Task A never blocks, we never get here.

// [H]: Find the top of Task A’s stack

desc\_ptr = nufrplat\_task\_get\_desc(NULL,

NUFR\_TID\_TASK\_A);

top\_of\_stack\_ptr = desc\_ptr->stack\_base\_ptr;

// [I]: Now that we’re the BG Task, we have to obey

// BG Task rules. BG Task must run in an

// infinite loop, so there must be a ‘while(1)’

// loop here.

while (1)

{

// [J]: The BG Task will do something useful:

// It will continuously monitor Task A’s

// stack’s topmost 16 bytes for a stack

// overrun. Since the stack

// is initialized to all zeros, any non-zero

// value is an overrun. If an overrun is

// detetected, call a fcn.

top\_null = memcmp(top\_of\_stack\_ptr, 0, 16) == 0;

if (!top\_null)

stack\_overrun\_error\_handler();

}

}

>>>>>>>>>>> *main.c* end >>>>>>>>>>>>>>>

>>>>>>>>>>> *app-A.c* begin >>>>>>>>>>>>>>>

void entry\_A(unsigned optional\_parameter)  
{  
 UNUSED(optional\_parameter);

// [E]: Task A will do almost nothing, as it is

// Put to sleep for a long time.

while (1)

{

// [F]: ping-pong between this and [J]

nufr\_sleep(NUFR\_SECS\_TO\_TICKS(100000));

}  
}

>>>>>>>>>>> *app-A.c* end >>>>>>>>>>>>>>>

### Task Switching Instructional

Three tasks are launched, and they ping-pong back and forth in context switches, then they terminate. BG Task runs through this launch/terminate sequence a second time.

>>>>>>>>>>> *main.c* begin >>>>>>>>>>>>>>>

void main(void)  
{  
 ....

// Pre-[A]

nufr\_init();

// Schedule Task C

nufr\_launch\_task(NUFR\_TID\_TASK\_C, 0);

// [J]: All tasks ran and terminated

// Run through [A]-[J] one more time.

nufr\_launch\_task(NUFR\_TID\_TASK\_C, 0);

// Final state.

while (1)

{

}

}

>>>>>>>>>>> *main.c* end >>>>>>>>>>>>>>>

>>>>>>>>>>> *app-A.c* begin >>>>>>>>>>>>>>>

void entry\_A(unsigned optional\_parameter)  
{  
 UNUSED(optional\_parameter);

// [C]: Change our own priority to be the

// same as Task B’s and C’s priorities.

nufr\_change\_task\_priority(nufr\_self\_tid(),

NUFR\_TPR\_NOMINAL);

// [D]: We’re still scheduled, because of

// round-robin share rules. Let Task C run

// again.

nufr\_yield();

// [G]: Task C yielded. This will cause us to

// terminate. Termination will cause another

// context switch, to Task B.

}

>>>>>>>>>>> *app-A.c* end >>>>>>>>>>>>>>>

>>>>>>>>>>> *app-B.c* begin >>>>>>>>>>>>>>>

void entry\_task\_B(unsigned optional\_parameter)  
{  
 UNUSED(optional\_parameter);

// [H]: This is an empty task, the moment it’s

// scheduled, it terminates.

}

>>>>>>>>>>> *app-B.c* end >>>>>>>>>>>>>>>

>>>>>>>>>>> *app-C.c* begin >>>>>>>>>>>>>>>

void entry\_task\_C(unsigned optional\_parameter)  
{

static bool do\_once = false;

UNUSED(optional\_parameter);

// [A]: We’re using the entire SL. Initialize

// all components of it. We choose to do

// it here, but can only do it once.

if (!do\_once)

{

do\_once = true;

nsvc\_init(NULL);

nsvc\_timer\_init();

nsvc\_pcl\_init();

}

// [B]: Launching Task A will cause a context switch

// to Task A, since Task A is at priority

// NUFR\_TPR\_HIGH, and Task C is at priority

// NUFR\_TPR\_NOMINAL.

nufr\_launch\_task(NUFR\_TID\_TASK\_A, 0);

// [E]: Got scheduled when Task A yielded.

// Launch Task B.

nufr\_launch\_task(NUFR\_TID\_TASK\_B, 0);

// [F]: We’re still scheduled, because of

// round-robin share rules. Let Task A run

// again, and Task B run after that.

nufr\_yield();

// [I]: Tasks A and B have both terminated, so

// context switch back to us. We’ll terminate

// too. The BG will continue running now.

}

>>>>>>>>>>> *app-C.c* end >>>>>>>>>>>>>>>

### Multiple Task Instances Instructional

Two tasks are launched, and both share the same entry point. Notice how the task descriptor has a different parameter. This is used to distinguish the USB devices.

>>>>>>>>>>> *nufr-platform-app.h* begin >>>>>>>>>>>>>>>

typedef enum  
{  
 NUFR\_TID\_null = 0, // not a task, do not change  
 NUFR\_TID\_USB\_DRIVER\_A,  
 NUFR\_TID\_USB\_DRIVER\_B,  
 NUFR\_TID\_max // not a task, do not change  
} nufr\_tid\_t;  
  
>>>>>>>>>>> *nufr-platform-app.h* end >>>>>>>>>>>>>>>

>>>>>>>>>>> *nufr-platform-app.c* begin >>>>>>>>>>>>>>>

const nufr\_task\_desc\_t nufr\_task\_desc[NUFR\_NUM\_TASKS] = {

{"USB Driver A",

entry\_usb\_driver,

Stack\_usb\_device\_A,

STACK\_SIZE\_USB\_DEVICE\_A,

NUFR\_TPR\_NOMINAL,

**0**},

{"USB Driver B",

entry\_usb\_driver,

// Must use a different stack!

Stack\_usb\_device\_B,

STACK\_SIZE\_USB\_DEVICE\_B,

NUFR\_TPR\_NOMINAL,

**1**},

};

>>>>>>>>>>> *main.c* begin >>>>>>>>>>>>>>>

void main(void)  
{

const nufr\_task\_desc\_t \*desc\_ptr;

....

nufr\_init();

desc\_ptr = nufrplat\_task\_get\_desc(NULL,

NUFR\_TID\_USB\_DRIVER\_A);

nufr\_launch\_task(NUFR\_TID\_USB\_DRIVER\_A,

desc\_ptr->instance);

desc\_ptr = nufrplat\_task\_get\_desc(NULL,

NUFR\_TID\_USB\_DRIVER\_B);

nufr\_launch\_task(NUFR\_TID\_USB\_DRIVER\_B,

desc\_ptr->instance);

while (1)

{

}

}

>>>>>>>>>>> *main.c* end >>>>>>>>>>>>>>>

>>>>>>>>>>> *nufr-platform-app.c* end >>>>>>>>>>>>>>>

>>>>>>>>>>> *usb-driver.c* begin >>>>>>>>>>>>>>>

// Common entry point shared for all USB devices

void entry\_usb\_driver(unsigned optional\_parameter)  
{

initialize\_usb\_peripheral(optional\_parameter);

....

}

>>>>>>>>>>> *usb-driver.c* end >>>>>>>>>>>>

# Bops

The *bop,* a kernel object, is unique to NUFR. Superficially, bops are semaphores which are limited to a count of zero or of one and which can only be used by one task. With an API call, a task will block itself on a bop. Another task, the BG Task included, and any exception handler can send a bop to a task which is blocked on a bop. This will cause the task to unblock.

In a list of bullet-points, bops exhibit these characteristics:

* Bops are the most light-weight OS objects NUFR offers
* There is one bop for each task; one bop is integrated into each Task Control Block (TCB)
* A task’s bop need not be allocated, created, configured, or enabled. It exists regardless of compile switch settings or configuration settings. Bops cannot be deleted either.
* Bops can be sent from the BG Task or from an interrupt handler

The following is a simple example of a bop. This example uses bops in the simplest mode: keyless and with no message abort. Assume that *entry\_A()* is Task A’s entry point. The BG Task launches Task A, causing Task A to be scheduled immediately. Task A then blocks on a bop. The BG Task sends it a bop, and Task A gets scheduled again:

void main()

{

nufr\_init();

// Launching Task A causes us (BG Task) to block

nufr\_launch\_task(NUFR\_TID\_TASK\_A, 0);

// Task A just blocked on a bop, now we’re

// scheduled again. Send it a bop. This will

// cause us to block.

nufr\_bop\_send\_with\_key\_override(NUFR\_TID\_TASK\_A);

...

}

// Task A’s entry

void entry\_A(unsigned optional\_parameter)  
{  
 // Task A runs after being launched, then

// blocks waiting indefinitely on a bop

// first thing.

nufr\_bop\_waitW(NUFR\_NO\_ABORT);

// BG Task sent us a bop; we proceed

...

}

## Bop Keys

Keys are used to ensure that a task which is waiting on a bop can only be released from the wait if the bop sender sends a key along with the bop, and the sent key matches the waiting task’s key. This feature was added to protect against corner-cases with bop waits, so that bop wait instances can be distinguished from each other. So if the timed bop wait API is called, and the timed wait times out, then if a second timed bop wait is initiated, should another task send a bop, if this other task uses the old key, it won’t work with the new wait instance.

This is made clear in an example. This is a code fragment of two tasks—the reader can fill in the missing details—where Task A waits on a bop, twice, and Task B sends a bop once. We’re not quite sure that Task B will be able to send the bop in time for Task A to receive it on the first bop wait without timing out. The key check makes sure that the bop send only works on the first wait instance, thus protecting the bop wait:

uint16\_t global\_bop\_key;

uint16\_t second\_bop\_key;

// Task A

void task\_A(void)

{

nufr\_bop\_wait\_rtn\_t wait\_rv;

....

global\_bop\_key = nufr\_bop\_get\_key();

// Wait 200 milliseconds for a bop from Task B

// and timeout if we don’t get one.

wait\_rv = nufr\_bop\_waitT(NUFR\_NO\_ABORT,

NUFR\_MILLISECS\_TO\_TICKS(200));

// Did the bop wait timeout?

if (NUFR\_BOP\_WAIT\_TIMEOUT == wait\_rv)

{

// Take some action

....

// Re-key. This invalidates ‘global\_bop\_key’,

// so Task B’s send will be ignored.

// Only the second key will be recognized now.

second\_bop\_key = nufr\_bop\_get\_key();

nufr\_bop\_waitW(NUFR\_NO\_ABORT);

}

....

}

// Task B

void task\_B(void)

{

....

// This will only work for Task A’s first bop wait.  
 nufr\_bop\_send(NUFR\_TID\_TASK\_A, global\_bop\_key);

....

}

## Bop Message Abort Feature

A bop wait API includes a parameter which specifies the priority level of a message sent to that task waiting on a bop which will cancel that task’s bop wait instance. The message abort feature is a component in the task kill feature. The intention is for a TYWYLH Message to cancel a bop wait instance (or any other wait, for that matter, not just a bop) that the destination task might be blocked on.

## ~~Bop Pre-Arrival~~

~~Bop~~ *~~Pre-Arrival~~* ~~is a features which allows a sent bop to arrive early, before the waiting task has a chance to wait on the bop, and still be taken by the waiter. In other words, a task can receive a bop before it actually attempts to block on the bop. This closes a corner-case with bops, allowing them to work more like semaphores. The sequence of a pre-arrival is as follows:~~

* ~~Task A prepares for a bop wait by getting a bop key (~~*~~nufr\_bop\_get\_key()~~*~~)~~
* ~~Task B preempts Task A before Task A can make the bop wait API call (~~*~~nufr\_bop\_waitW()~~*~~).~~
* ~~Task B gets Task’s A key and sends a bop to Task A using that key. The bop is now pre-arrived.~~
* ~~Task A gets scheduled again~~
* ~~Task A makes a call to wait on a bop (~~*~~nufr\_bop\_waitW()~~*~~). Since the bop is pre-arrived, the wait call returns immediately; task A never blocks.~~

~~Note that Pre-Arrival only applies to keyed bop sends. A key-override bop send (~~*~~nufr\_bop\_send\_with\_key\_override()~~*~~) will not pre-arrive.~~

### ~~Bop Pre-Arrival Instructional~~

~~Assume Task A and Task B run at the same priority. Task A is scheduled at the start time of this code fragment:~~

~~uint16\_t global\_bop\_key;~~

~~// Task A~~

~~void task\_A(void)~~

~~{~~

~~nufr\_bop\_wait\_rtn\_t wait\_rv;~~

~~....~~

~~global\_bop\_key = nufr\_bop\_get\_key();~~

~~// This causes Task B to run~~

~~nufr\_yield();~~

~~// Task B yielded~~

~~// This call will return immediately~~

~~wait\_rv = nufr\_bop\_waitW(NUFR\_NO\_ABORT);~~

~~// This task continues~~

~~....~~

~~}~~

~~// Task B~~

~~void task\_B(void)~~

~~{~~

~~....~~

~~// Task A hasn’t called ‘nufr\_bop\_waitW()’ yet.~~

~~// This bop will pre-arrive however.  
 nufr\_bop\_send(NUFR\_TID\_TASK\_A, global\_bop\_key);~~

~~// Allow Task A to run again~~

~~nufr\_yield();~~

~~....~~

~~}~~

## Local Structs and Bop Locking

*Bop Locking* is a bop feature that allows for a task waiting on a bop to be locked on that bop. The locking is actuated by another task (and can be actuated by the BG task), not the task waiting on the bop. In order for a bop lock to happen, the intended target task must already be waiting on a bop. The task which does the locking must pass to the lock API call the matching key in order for the lock to engage. The lock prevents a task waiting on a bop from cancelling (i.e. unblocking from) its bop wait due to a bop send, a timeout, or an abort message send, which are the only three actions (short of a task kill) which can cause a task to unblock from that bop block instance. A Bop Lock is followed by a Bop Unlock much like a critical section is entered then exited, or like a mutex which is taken and then given: the locking typically occurs for a short duration and for an intended purpose.

One of the use-cases for a Bop Lock is another feature called *Local Struct*. The Local Struct feature is not bound to bops, but the APIs can be used with any of the other OS objects and features—so long as the contract is honored. If the contract is violated, the Local Structs misusage can cause a crash. The purpose of Local Struct is for one task to pass a pointer to a variable or structure which exists on its task stack (in C, an auto storage class variable) to another task, as a means of sharing data between tasks. It’s possible for one task to access and share a local, stack-resident variable or structure from another task so long as the task which has offered up the variable/structure to be shared is not scheduled for the duration of the share. This is the contract, and this is the main purpose of the Bop Lock feature—to ensure that a task which is sharing out a variable remains blocked on a bop. Otherwise, if the task accessing the shared variable writes to that address, and if the task isn’t blocked for the duration, the stack might’ve changed. If the stack changed, the write is basically doing the equivalent of writing to a random location on the stack, which would be a data corruption.

The below list outlines a use-case and the list specifies the sequence of events for a bop lock used in conjunction with a local struct, where Task A offers up a local struct to be shared and Task B accesses the local struct:

* Task A gets a bop key and publishes it to Task B
* Task A then sets its local struct. This gets stored in Task A’s TCB.
* Task A then blocks on a bop get
* Task B gets scheduled
* Task B takes Task A’s key and uses it to lock Task A to its bop wait
* Task B then accesses Task A’s local struct. Task B writes to it. Task B is finished with using the data.
* Task B unlocks Task A’s bop lock
* Task B (presumably) sends Task A a bop, making Task A ready

An additional piece of logic with bop locks is that a bop lock can only be applied if the target is blocked on a bop. Pre-arrivals don’t work with bop locks—the target task must already be blocked on the bop for the lock to stick.

### Bop Locking and Local Struct Instructional

At the beginning of this instructional, Task A is scheduled and Task B is blocked. We’ll assume that when Task A blocks on its bop, that some other event not documented here causes Task B to unblock and invoke the function *task\_B\_access().*

uint16\_t global\_bop\_key;

struct data\_tp

{

unsigned x;

unsigned y;

};

// Task A

void task\_A(void)

{

nufr\_bop\_wait\_rtn\_t wait\_rv;

struct data\_tp my\_data;

....

global\_bop\_key = nufr\_bop\_get\_key();

my\_data.x = 0;

my\_data.y = 0;

// Save my\_data’s address to Task A’s TCB

nufr\_local\_struct\_set(&my\_data);

// Wait for a long time. Assume Task B calls

// ‘task\_B\_access()’ in the meantime.

wait\_rv = nufr\_bop\_waitT(NUFR\_NO\_ABORT,

NUFR\_SECS\_TO\_TICKS(5));

// This task continues with new values:

// my\_data.x == 1 and my\_data.y == 2

....

}

// Assume Task B doesn’t execute here until Task A

// is waiting on the bop

void task\_B\_access(void)

{

nufr\_bop\_rtn\_type lock\_rv;

struct data\_tp \*my\_data\_ptr;

// Attempt to apply lock

lock\_rv = nufr\_bop\_lock\_waiter(NUFR\_TID\_TASK\_A,

global\_bop\_key);

if (NUFR\_BOP\_RTN\_TAKEN == lock\_rv)

{

// We have the lock. It’s safe to retrieve

// Task A’s local struct. Task A isn’t

// going anywhere.

my\_data\_ptr =

nufr\_local\_struct\_get(NUFR\_TID\_TASK\_A);

// This will write to Task A’s ‘data\_ptr’

// local variable.

my\_data\_ptr->x = 1;

my\_data\_ptr->y = 2;

// We’re done accessing data. Release the lock.

nufr\_bop\_unlock\_waiter(NUFR\_TID\_TASK\_A);

// Do send inside the ‘if’: reasoning is that

// if lock wasn’t applied, then no sense sending

// bop—task timed out of bop weit

nufr\_bop\_send(NUFR\_TID\_TASK\_A, global\_bop\_key);

}

....

}

# Semaphores, Mutexes, and SL Semaphore Usage

NUFR semaphores work similar to most other RTOS semaphores. There’s an additional feature to NUFR semaphores—message abort on a semaphore wait—but apart from that, NUFR semaphores are boilerplate. Semaphores exhibit these characteristics:

* They have counts (unsigned integers—no negative values)
* Multiple tasks can block on the same semaphore
* Tasks blocking on the same semaphore are ordered first by priority and second by order of arrival
* Priority inversion protection (optional)
* Timeouts on wait
* Message abort capability
* Ability to compile in or compile out semaphore support

Unlike other RTOSs, NUFR semaphores don’t take center stage. For simple task control, it’s preferable to use bops instead of semaphores. (For that matter, it’s better to use messaging for task control rather than bops or semaphores.) Excluding semaphores from basic task control, the left-over semaphore use-cases place semaphore usage behind the scenes, their primary purpose being used in the construction of SL objects and features.

## Semaphore Initialization and Priority Inversion Protection

For the reason recently alluded to, the semaphore reset API is partially obscured from the end-user. Semaphores must be reset before being used. Using the default-provided platform code that’s provided with the OS distribution, all semaphores get a default initialization. If the app developer needs to use a semaphore directly, and if he or she needs to change the default initialization settings, the API is available, and works in this way. Note the necessity of including the file *nufr-kernel-semaphore.h*, which isn’t part of the API set. This file contains the macro *NUFR\_SEMA\_ID\_TO\_BLOCK*(), which isn’t normally exported to the end-user.

>>>>>>>>>>> *nufr-platform-app.h* start >>>>>>>>>>>>>>>

typedef enum  
{  
 NUFR\_SEMA\_null = 0, // not a sema, do not change  
 NUFR\_SEMA\_X,  
 NUFR\_SEMA\_Y,  
 NUFR\_SEMA\_Z,  
 NUFR\_SEMA\_POOL\_START,   
 NUFR\_SEMA\_POOL\_END = NUFR\_SEMA\_POOL\_START + 10,   
 NUFR\_SEMA\_max // not a sema, do not change  
} nufr\_sema\_t;

>>>>>>>>>>> *nufr-platform-app.c*  end >>>>>>>>>>>>>>>

>>>>>>>>>>> *foo-app.c* begin >>>>>>>>>>>>>>>

#include “nufr-kernel-semaphore.h”

void foo(void)

{

// We need to reinitialize NUFR\_SEMA\_X to an

// initial count of 0 instead of 1.

nufrkernel\_sema\_reset(

NUFR\_SEMA\_ID\_TO\_BLOCK(NUFR\_SEMA\_X),

0,

// enable priority inversion protection

true);

}

>>>>>>>>>>> *foo-app.c* end >>>>>>>>>>>>>>>

The example above looks innocuous but in fact can cause a system crash if not implemented properly, and for this reason *nufrkernel\_sema\_reset*() is designated a kernel call, rather than being exported as an app API. The side-effect of this reset API is that the semaphore’s task wait list is cleared in the course of resetting the semaphore, and consequently any tasks which were waiting on the semaphore at the time this reset function is called will become stranded, and this will cause a system lockup.

Nevertheless, there are cases where *nufrkernel\_sema\_reset()* needs a direct call from the application layer. A reset provides the means of forcing a semaphore count to a specific value and the means of either turning on or off the priority inversion protection feature in real-time. Priority inversion protection shouldn’t be used when the semaphore count will exceed one, as protection is senseless and could potentially cause a crash if a semaphore is used in this way. Priority inversion assumes that a single task “owns” a semaphore when a second, higher-priority task needs to take it. The first task is temporarily raised in priority to that of the second task’s priority, in the hope that the owning task will return the semaphore so that the higher-priority task can take it. If a semaphore is initialized to a count greater than zero, then there can be no single semaphore owner, so if priority inversion were enabled, which task must be raised in priority? It’s the opinion of the creators of NUFR that the perceived need in the software community for priority inversion protection attempts to mask over a misuse of an RTOS by application code, implying that a properly architected application won’t ever need priority inversion protection. Nevertheless, for the sake of compatibility with other RTOSs (and hence portability), NUFR supports priority inversion, and applies it by default to SL mutexes, its most likely usefulness.

## Semaphore Algorithms

Semaphore actions consist of “gets” and “releases”. A get action takes a semaphore, decrementing the semaphore count, and a release action returns the semaphore, incrementing the count. If a semaphore count is zero and a get is attempted, the count is kept at zero, and the task attempting to get the semaphore will block on the semaphore. If a release happens when no tasks are blocking on that semaphore, the semaphore count is incremented. If a release action happens when the semaphore count is zero, and if any tasks are blocking on the semaphore, the count will be kept at zero, but instead the task at the head of the semaphore’s wait list will be unblocked and made ready. Naturally, any changes made to the ready list, whether a task is added to the list or the scheduled task becomes blocked, may or will necessitate a context switch.

A semaphore’s task wait list is an ordered list, and the ordering is identical to the ordering algorithm used by the ready list. When a release action looks to take a task off the semaphore wait list, it always takes the list head.

## Semaphore Ownership

The kernel specifies that a semaphore can have an “owner”. This concept of ownership is obscured from the application developer, and only comes into play with priority inversion protection and with task killing. To summarize, however, the semaphore “owner” is the last task to have done a semaphore get on a given semaphore. The semaphore keeps a record of the owner so that it can do a check for priority inversion when applicable.

Semaphore owner information comes into use when a task is terminated or is killed. Inside the kernel, when a task terminates or when a task kill is executed, the kernel attempts to do as much clean-up as possible on the terminating task, to avoid a system crash. Task kills are risky and invite system crashes. The termination or task kill logic checks all semaphores to see if they’re owned by the task being killed, and if any one is owned, the termination/kill logic releases that semaphore.

The task killing feature works in conjunction with the message abort feature. A semaphore get API specifies the priority level of a message sent to that task which will cancel a task’s blocking on a semaphore. Message abort on a semaphore works like message aborts on bops.

## Mutexes

NUFR provides mutexes in the SL rather than in the kernel. Mutexes are simple wrappers for semaphores. The mutex wrapper ensures that the mutex’s underlying semaphore is initialized to a count of one and that priority inversion is enabled.

Since mutexes consume semaphores at the SL, they draw semaphores from the semaphore pool dedicated for the SL use. If this pool is too small, meaning that the pool is less than the clients needing semaphores, in this case the clients being mutexes, then a system failure will occur.

## The SL Semaphore Pool

Some of the Service Layer components are implemented using semaphores. As a convenience to the application developer, the platform app code (*nufr-platform-app.c/h*) need not be reconfigured to accommodate the insertion or removal of a component or object at the SL which utilizes semaphores. Instead, a static pool of semaphores is created at the platform app level, and the SL draws from this pool. Currently, this pool works as a one-time allocation, so there is no pool freeing capability.

For those using the SL, it’s expected that the pool size be set to an adequate level. If resources aren’t terribly scarce, its convenient to burn a few bytes of RAM and size the pool with a few extra semaphores, to ensure greater decoupling between the SL and the platform app layer. The app developer need be mindful of pool consumption at the SL level. He or she is responsible for ensuring an adequate pool size.

## Semaphore API Calls from Interrupt Handlers and BG Task

Interrupt handlers and the BG Task are prevented from making semaphore get and release calls. The reason is that this interferes with the task ownership logic, which is necessary to support task killing and priority inversion. Future NUFR releases may revisit this functionality, but there is no pressing use-case to support this.

Having stated this, there are some back-doors that the SL uses to increment and decrement semaphores from an interrupt handler, so that the interrupt handler can make SL pool allocation or free calls (which is not necessarily recommended). In this case, the semaphores assigned to a SL memory pool will have counts greater than zero, allowing the interrupt handler to innocuously increment or decrement a count. The back-door route is also much faster than the kernel API call. The app developer wanting to manipulate semaphores from a non-task thread should first study the SL back-door then should consider the corner-cases that may make this problematic.

# Messaging

*Messaging* is the term used to describe the means of sending messages to tasks and for tasks to receive messages. Each task has a message queue dedicated to that task. This queue is built into the TCB. Since there are no other message receive queues, messages can only be received by tasks. A message consists of a 12 byte *message block*. Message blocks are dynamically allocated from a message block allocation service in the platform component.

Messaging steers tasks into event-driven models, where a task blocks on an API call awaiting a message to be sent to it. If a task is blocking on a message to be sent to it, and if a message is sent to this task, the kernel will make this task ready, adding it to the ready list. When the task gets scheduled, it will retrieve that message and consume it. If a task receives multiple messages before it can process them, those messages are enqueued first by message priority and second by the order in which they were sent, and tasks dequeue messages accordingly. If a task attempts to block on a receive message, and if a message was already queued up, the task will dequeue the message and proceed without blocking.

The specific steps in a message send and receive (from a kernel API perspective) are the following, where Task A is the message sender and Task B is the message receiver. Task B is blocked on a message receive (*nufr\_msg\_getW*() call).

* Task A fills in the message fields using macro *NUFR\_SET\_MSG\_FIELDS()* or equivalent. This gets packed into a *uint32\_t* or can be expanded directly in the *nufr\_msg\_send()* call.
* Task A calls *nufr\_msg\_send*() to send the message, specifying Task B as the destination task
* Inside *nufr\_msg\_send()*, the kernel internally allocates a message block from the message block pool, populates it, and enqueues it onto Task B’s message queue. Task B’s message queue must have been empty, since Task B was blocked on a message receive.
* Task B is unblocked from its message block state and is added to the ready list
* Assuming Task B has the same task priority as Task A, Task B is made ready, but it’s not scheduled. Task A continues to run.
* For reasons unspecified, Task A will eventually block. Task B gets scheduled.
* Task B dequeues the message, copies the contents, and then returns the message block to the message block pool. This happens inside the kernel.
* Task B returns from the *nufr\_msg\_getW*() call that caused it to block. The call returns the message fields that the sender packed, putting them into the parameter variables specified by Task B in *nufr\_msg\_getW()*.
* Task B proceeds and presumably consumes the message (does some unspecified processing based on the message)

## Message Prefixes and Message IDs

Two message fields are used as the means for applications to distinguish one message from another: the message prefix and the message ID. When a task consumes a received messaged, the task will use message prefix and ID to select among various code paths to take. How these are used—if one or the other or either is used—is determined at the application level. The kernel has no interest in these fields and has no logic which uses them.

Applications can do creative things with message prefixes and IDs. The SL uses prefixes and IDs according to a design methodology, and if a codebase includes the messaging services that the SL provides, they will inherit the benefits of this design. The SL messaging prefixes and IDs are designed to work in this manner:

* Prefixes are used as an abstraction for specifying a task as the message destination. Messages are sent to a prefix directly, which determines which task it gets sent to indirectly. Addressing messages to prefixes decouples the messages sent from the receiving tasks.
* A prefix can be used to specify a set of message IDs, or a message ID space. In this way, a prefix is a category of messages, and the ID is the specific message. Message ID numbers can be reused from prefix to prefix. Applications can define message ID enums, and these enums will only be applicable for one specific prefix.
* Prefixes can be reserved for the purpose of specifying message sends to multiple tasks. A message sent to a prefix gets replicated and sent to a list of tasks. In a medium to large system, this enables the publication of power-up, shutdown, restart, configuration received, etc. events.
* In smaller codebases, to simply matters, the prefix can be ignored (see *NSVC\_MSG\_PREFIX\_local*), and the message can be sent directly to a task/message ID pair.

## Message Priorities

The kernel maintains an ordered list of all messages that are awaiting processing by a task. This list is ordered by message priority first and by order of arrival second. When a task is awakened from a blocking state due to a call to *nufr\_mesage\_getW*()or *nufr\_msg\_getT(),* the messages are presented to the receiving task in order. Prioritization of messages gives applications the ability to assign a priority when sending a message and, if a high priority is assigned, be guaranteed that the receiving task will take action immediately, as a high priority message preempts messages at a lower priority.

## Message Optional Parameter

Four bytes of each message block are reserved for an optional parameter, to be used by the application layer any way the app developer pleases. This 32-bit integer can be used to attach a data block pointer to the message. When a buffer pointer is attached to the message, the attached buffer can serve as an extension of the message, filling in additional fields that the app developer needs. Or, an attached buffer can be used in a networking stack, as a means to pass buffers from one task to another task or within a task from one state machine to another. The interpretation of how the parameter is used can vary from message type to message type; in one message, it might be an IPv4 packet, in another message type it might be data to be written to a file. Regardless of how the parameter value is used, the kernel takes no interest in it.

## Message Sends from Interrupt Handlers

Messages can be sent from an exception handler (interrupt or Systick) or from the BG Task. There are some strategies that an application developer can apply to minimize the CPU cycles of a message send, and thereby reduce the interrupt latency when messages are sent from an interrupt handler:

* Use the raw NUFR API call (*nufr\_msg\_send()*) instead of SL calls to send messages
* Pre-allocate message blocks: a task should allocate a message block and copy its pointer to a global variable, so that when the interrupt handler needs to send a message, it block is already allocated.
* Pack the message fields at compile time, or have the fields filled in by the task which allocates the message block. Either way, make it so that the interrupt handler uses pre-formatted messages which have pre-packed bit fields.
* Turn off the compile-features for Task Kill and for Local Structs. These consume a few CPU cycles in *nufr\_msg\_send()*.

An even faster means than the above of doing a message send is to use of these two macros: *NUFR\_MSG\_SEND\_INLINE()* or *NUFR\_MSG\_SEND\_INLINE\_NO\_LOCKING().* These are in the file *nufr-kernel-message-send-inline.h.* (Currently not supported on MSP430.) Of course, using these inline macros will consume more text space (FLASH), but they will be faster. *NUFR\_MSG\_SEND\_INLINE\_NO\_LOCKING* is faster than *NUFR\_MSG\_SEND\_INLINE*, but can only be used where it’s possible to skip the interrupt locks. The system engineer must be vigilant when skipping the interrupt locks to ensure that the codebase doesn’t have any instances where *NUFR\_MSG\_SEND\_INLINE\_NO\_LOCKING* is used in an ISR that is not the highest priority ISR (on the ARM Cortex M, for example, this is the interrupt vector priority), and if one or more of the other higher priority ISR(s) makes any NUFR API calls, then a kernel corruption can occur.

In addition to using less total CPU cycles, the inline message send macros have another advantage: the message block allocation and send are combined in one operation. This streamlines the code, as there’s less things for the application engineer to manage. You can see the difference by comparing the two instruction code examples below. The second instructional—the one that uses *NUFR\_MSG\_SEND\_INLINE* is simpler from a code complexity perspective than the first.

### Message Send from Interrupt Handler Instructional: No Inline Send

>>>>>>>>>>> *foo-handler.c* start >>>>>>>>>>>>>>>

#include “nufr-global.h”

#include “nufr-platform.h”

#include “nufr-platform-app.h”

#include “nufr-api.h”

#define FOO\_MSG\_PREFIX 1

#define FOO\_MSG\_ID 5

const uint32\_t prepacked\_fields = NUFR\_SET\_MSG\_FIELDS(

FOO\_MSG\_PREFIX,

FOO\_MSG\_ID,

NUFR\_TID\_null,

NUFR\_MSG\_PRI\_MID);

const uint32\_t prepacked\_parameter = 1;

void foo\_interrupt\_handler(void)

{

....

// Send pre-allocated+pre-formatted msg to task foo

(void)nufr\_msg\_send(prepacked\_fields,

prepacked\_parameter,

NUFR\_TID\_FOO);

}

>>>>>>>>>>> *foo-handler.c*  end >>>>>>>>>>>>>>>

>>>>>>>>>>> *foo-app.c* begin >>>>>>>>>>>>>>>

// task ‘foo’ entry point

void foo\_task(void)

{

uint32\_t fields;

uint32\_t parameter;

....

while (1)

{

nufr\_msg\_getW(&fields, &parameter);

if (FOO\_MSG\_ID == NUFR\_GET\_MSG\_ID(fields))

{

// Some unspecified msg usage

....

}

// Handle other message

else

{

....

}

}

}

>>>>>>>>>>> *foo-app.c* end >>>>>>>>>>>>>>>

### Message Send from Interrupt Handler Instructional: Inline Message Send

>>>>>>>>>>> *foo-handler.c* start >>>>>>>>>>>>>>>

#include “nufr-global.h”

#include “nufr-platform.h”

#include “nufr-platform-app.h”

#include “nufr-api.h”

#include “nufr-kernel-message-send-inline.h”

#define FOO\_MSG\_PREFIX 1

#define FOO\_MSG\_ID 5

void foo\_interrupt\_handler(void)

{

....

NUFR\_MSG\_SEND\_INLINE(NUFR\_TID\_FOO, // Destination task

FOO\_MSG\_PREFIX, // Message prefix

FOO\_MSG\_ID, // Message ID

NUFR\_MSG\_PRI\_MID, // Msg priority

0); // Optional parameter

}

>>>>>>>>>>> *foo-handler.c*  end >>>>>>>>>>>>>>>

>>>>>>>>>>> *foo-app.c* begin >>>>>>>>>>>>>>>

// task ‘foo’ entry point

void foo\_task(void)

{

uint32\_t fields;

uint32\_t parameter;

....

while (1)

{

nufr\_msg\_getW(&fields, &parameter);

if (FOO\_MSG\_ID == NUFR\_GET\_MSG\_ID(fields))

{

// Some unspecified msg usage

....

}

// Handle other message

else

{

....

}

}

}

>>>>>>>>>>> *foo-app.c* end >>>>>>>>>>>>>>>

## Message Block Allocation

NUFR messaging uses something called *message blocks* when doing messaging. Each message uses a single message block. The kernel uses the blocks to enqueue messages onto a task’s message queue. The allocation and freeing of message blocks is done internally in the kernel when using APIs such as *nufr\_msg\_send()* and *nufr\_msg\_getW(),* and the SL messaging routines manage message blocks internally also. The application developer need not concern himself or herself with message blocks at a superficial level. In advanced debugging scenarios, the application and system developers will need to delve into the details of message blocks and the *message block pool* (*bpool* for short).

The *bpool* is initialized internally by *nufr\_init(),* and pool accessor functions are kept in *nufr-kernel-message-blocks.c*. In advanced scenarios, these routines can be called, but typically they will not be. Care must be taken to make sure no corner cases arise where the message pool gets depleted. Properly sizing the message pool (setting define *NUFR\_MAX\_MSGS*) is the responsibility of the system developer.

There is a more advanced message send API called *nufr\_msg\_send\_by\_block(),* and this requires that the caller allocate the block rather than the kernel allocate the block. This is only needed for some advanced corner cases. An example of this is given here.

### Message Block and Message Send Instructional

Here is a complete message allocation, message send, message get, then message free example. Comments inline:

>>>>>>>>>>> *nufr-platform-app.h* begin >>>>>>>>>>>>>>>

typedef enum  
{  
 NUFR\_TID\_null = 0, // not a task, do not change  
 NUFR\_TID\_SENDER,  
 NUFR\_TID\_RECEIVER,

NUFR\_TID\_SOME\_OTHER\_TASK,

NUFR\_TID\_max

} nufr\_tid\_t;

>>>>>>>>>>> *nufr-platform-app.h* end >>>>>>>>>>>>>>>

>>>>>>>>>>> *sender-task.c* begin >>>>>>>>>>>>>>>

void some\_sender\_function(void)

{

nufr\_msg\_t \*msg;

nufr\_msg\_send\_rtn\_t send\_result;

msg = nufrplat\_msg\_get\_block();

// ‘msg’ == NULL if block pool is depleted

// Under ordinary conditions, pool should not be empty.

// If you ever get a NULL return value, you either have

// a bpool leak or you need to increase the number

// of blocks. Whatever reason, it is advisable to

// fix all NULL return values.

if (NULL != msg)

{

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

0, // prefix

1, // ID

NUFR\_TID\_SENDER, // sending task

NUFR\_MSG\_PRIORITY\_MID);

msg->parameter = 0;

send\_result = nufr\_msg\_send\_by\_block(msg,

NUFR\_TID\_RECEIVER);

// You can skip the ‘send\_result’ check if you’re

// confident NUFR\_TID\_RECEIVER task has been

// launched and that all the msg parameter are

// correct. Otherwise, you must free block or

// else this’ll be the cause of a leak

if (NUFR\_MSG\_SEND\_ERROR == send\_result)

{

nufrplat\_msg\_free\_block(msg);

}

}

...

}

>>>>>>>>>>> *sender-task.c* end >>>>>>>>>>>>>>>

>>>>>>>>>>> *receiver-task.c* begin >>>>>>>>>>>>>>>

void receiver\_function(void)

{

uint32\_t fields;

uint32\_t value;

// Block indefinitely for a message.

// ‘fields’ always gets filled in

(void)nufr\_msg\_getW(&fields, NULL);

// Is this the message sent by NUFR\_TID\_SENDER?

if (1 == NUFR\_GET\_MSG\_ID(fields))

{

....

}

}

>>>>>>>>>>> *receiver-task.c* end >>>>>>>>>>>>>>>

### Block Allocation in a Real System

The intent of NUFR messaging is that a runtime system never runs out of message blocks. The system developer must create a large enough block pool to handler all corner-cases. The app developers must not have any message block leaks and must not allocate messages and enqueing them faster than the system can consume them. The first, message block leaking, is obvious. The second, allocating and enqueueing blocks at too fast of a rate, is not as obvious. The following steps illustrate:

1. A discrete input (DI) triggers an interrupt
2. The interrupt handler for that discrete input sends a message to a task notifying it of the DI toggle event
3. The message for the toggle event is enqueued in some applications task
4. Before the application task can process that message, another DI interrupt occurs; another message is sent to that task.
5. Sequence continues until the message block pool is depleted

But what should a system do when it runs out of blocks while a task requests a block? Here are the options:

1. Wait for a block to become available
2. Force-crash the system
3. Drop the message send
4. A combination of the above

## Message Prefixes and IDs in the SL

Each message block reserves 10 bits (ref. *NUFR\_MSG\_MAX\_PREFIX*, *NUFR\_MSG\_MAX\_ID* in *nufr-api.h*) for a message prefix and 10 bits for a message ID. In fact, the app developer may choose to concatenate the prefix and ID bits into a single 20 bit value (ref. *NUFR\_GET\_MSG\_PREFIX\_ID\_PAIR*, *NUFR\_SET\_MSG\_PREFIX\_ID\_PAIR*). But there are recommended design patterns for using prefixes and IDs, and these patterns are in place in the SL messaging component, which is the file *nsvc-messaging.c*.

The SL has an enum called *nsvc\_msg\_prefix\_t* in the SL app configuration file, *nsvc-app.h.* This enum should be filled in by the app developer. Here’s an example:

typedef enum  
{  
 NSVC\_MSG\_PREFIX\_local = 1, //mandatory  
 NSVC\_MSG\_PREFIX\_UI\_STATE\_MACHINE,  
 NSVC\_MSG\_PREFIX\_DRIVER\_DIO,  
 NSVC\_MSG\_PREFIX\_DRIVER\_UART  
} nsvc\_msg\_prefix\_t;

The entry *NSVC\_MSG\_PREFIX\_local* should always appear, and should always appear as the first entry and should always be set equal to one. The SL code compiles against this value. *NSVC\_MSG\_PREFIX\_local* is used as a null prefix, a don’t-care prefix, or to indicate that the prefix field isn’t being used. The app developer should add other prefix entries to his or her contentment. Since the prefix maps to a 10-bit value, there is a maximum of 1024 enum entries which *nsvc\_msg\_prefix\_t* can hold (the same 1024 limit applies to IDs).

It’s recommended that that the app developer created enums for message IDs, grouping message IDs into a single enum typedef. These enums will exist in app files, not in any NUFR or SL file. The ability to define message IDs in app files rather than creating them in the SL app file, like the message prefix is, provides flexibility to the app developer, putting message IDs under his or her thumb. Continuing the previous example, here’s a few message ID enums created in a couple of hypothetical app files:

>>>>>>>>>>> *ui-app.h*  begin >>>>>>>>>>>>>>>

// Belong to prefix ‘NSVC\_MSG\_PREFIX\_UI\_STATE\_MACHINE’

typedef enum

{

UI\_ID\_INIT,

UI\_ID\_OK\_BUTTON\_PRESSED,

UI\_ID\_POWER\_BUTTON\_PRESSED,

} ui\_id\_t;

>>>>>>>>>>> *ui-app.h* end >>>>>>>>>>>>>>>

>>>>>>>>>>> *driver-app.h*  begin >>>>>>>>>>>>>>>

// Belong to prefix ‘NSVC\_MSG\_PREFIX\_DRIVER\_DIO’

typedef enum

{

DIO\_ID\_INTERRUPT\_DI\_INPUT,

DIO\_ID\_CHIP\_INIT,

} dio\_id\_t;

// Belong to prefix ‘NSVC\_MSG\_PREFIX\_DRIVER\_UART’

typedef enum

{

UART\_ID\_INIT,

UART\_ID\_INTERRUPT\_RX,

UART\_ID\_RECONFIGURE

} uart\_id\_t;

>>>>>>>>>>> *driver-app.h* end >>>>>>>>>>>>>>>

This example is a use-case of message prefixes and message IDs used together. In the code snippet above, there is a prefix entry for each ID enum; for any given message block, the prefix in that block will indicate which enum is used in that block’s ID field. Continuing with the code for that example, this is how it will be used. As a side note, the SL messaging APIs are used below instead of the NUFR APIs.

>>>>>>>>>>> *ui-app.c*  begin >>>>>>>>>>>>>>>

void ui\_task(unsigned parm)

{

nsvc\_msg\_prefix\_t prefix;

uint16\_t id\_parm;

ui\_id\_t id;

UNUSED(parm);

while (1)

{

// We’re only interested in prefix+id in msg

nsvc\_msg\_get\_argsW(NULL,

&prefix,

&id\_parm,

NULL,

NULL,

NULL);

if (NSVC\_MSG\_PREFIX\_UI\_STATE\_MACHINE == prefix)

{

id = (ui\_id\_t)id\_parm;

switch (id)

{

case UI\_ID\_INIT:

....

break;

case UI\_ID\_OK\_BUTTON\_PRESSED:

....

break;

case UI\_ID\_POWER\_BUTTON\_PRESSED:

....

break;

default:

// Error, this shouldn’t happen

....

break;

}

}

else

{

// Unexpected message. Quietly discard

// Notice that we don’t have to free msg

// block; nsvc\_msg\_get\_argsW() does that

// for us.

}

}

}

>>>>>>>>>>> *ui-app.c* end >>>>>>>>>>>>>>>

>>>>>>>>>>> *driver-app.c*  begin >>>>>>>>>>>>>>>

void driver\_task(unsigned parm)

{

nsvc\_msg\_prefix\_t prefix;

uint16\_t id\_parm;

dio\_id\_t dio\_id;

uart\_id\_t uart\_id;

UNUSED(parm);

while (1)

{

// We’re only interested in prefix+id in msg

nsvc\_msg\_get\_argsW(&prefix,

&id\_parm,

NULL,

NULL,

NULL);

if (NSVC\_MSG\_PREFIX\_DRIVER\_DIO == prefix)

{

id = (ui\_id\_t)id\_parm;

switch (id)

{

case DIO\_ID\_INTERRUPT\_DI\_INPUT:

// Send a message to the UI task above.

// Discard return value.

(void)nsvc\_msg\_send\_argsW(NULL,

NSVC\_MSG\_PREFIX\_UI\_STATE\_MACHINE,

UI\_ID\_OK\_BUTTON\_PRESSED,

NUFR\_MSG\_PRIORITY\_MID,

NUFR\_TID\_UI, //assume this is UI task

0);

....

break;

case DIO\_ID\_CHIP\_INIT:

....

break;

default:

// Error, this shouldn’t happen

....

break;

}

}

else (NSVC\_MSG\_PREFIX\_DRIVER\_UART == prefix)

{

id = (ui\_id\_t)id\_parm;

switch (id)

{

case UART\_ID\_INIT:

....

break;

case UART\_ID\_INTERRUPT\_RX:

....

break;

case UI UART\_ID\_RECONFIGURE:

....

break;

default:

// Error, this shouldn’t happen

....

break;

}

}

else

{

//Unexpected message. Quietly discard

}

}

}

>>>>>>>>>>> *driver-app.c* end >>>>>>>>>>>>>>>

### SL Lookup Based on Message Prefix

The SL app layer has a facility that is filled in by the app developer. This facility maps message prefixes to tasks. An SL message send API can specify a message prefix, and the SL layer, using the app-defined prefix mapping, will look up the destination task for that prefix. Here’s an example, one which continues the flow of examples from above:

>>>>>>>>>>> *driver-app.c*  begin >>>>>>>>>>>>>>>

(void)nsvc\_msg\_send\_argsW(NULL,

NSVC\_MSG\_PREFIX\_UI\_STATE\_MACHINE,

UI\_ID\_OK\_BUTTON\_PRESSED,

NUFR\_MSG\_PRIORITY\_MID,

NUFR\_TID\_null,

0);

>>>>>>>>>>> *driver-app.c*  end >>>>>>>>>>>>>>>

Notice how the fifth parameter is *NUFR\_TID\_null* instead of *NUFR\_TID\_UI.* The selection of a null task ID tells the SL to look up the destination task internally. The internal call to look up the task uses the mapping which was created by the app developer in *nsvc-app.c*. Assume that *NUFR\_TID\_UI* and *NUFR\_TID\_DRIVER* are the enums in *nufr\_tid\_t* for the two tasks in the example snippets:

>>>>>>>>>>> *nsvc-app.c*  begin >>>>>>>>>>>>>>>

bool nsvc\_msg\_prefix\_id\_lookup(nsvc\_msg\_prefix\_t prefix,  
 nsvc\_msg\_lookup\_t \*out\_ptr)  
{

....

switch (prefix)  
 {  
 case NSVC\_MSG\_PREFIX\_UI:

tid = NUFR\_TID\_UI;

break;

case NSVC\_MSG\_PREFIX\_DRIVER\_DIO:

case NSVC\_MSG\_PREFIX\_DRIVER\_UART:

tid = NUFR\_TID\_DRIVER;

break;

....

}

....

}

>>>>>>>>>>> *nsvc-app.c*  end >>>>>>>>>>>>>>>

The app developer has the option of addressing a message to a message prefix or addressing a message directly to a task. There are advantages and disadvantages of both approaches. The advantage of sending to a message prefix instead of directly to a task is that the application code can choose prefixes and IDs along feature or application boundaries, independent of tasks. This makes a large codebase more readable and more maintainable. Of course, sending a message directly to a task instead of a prefix takes less CPU cycles, so low-powered systems will likely prefer to direct message sends to tasks instead of to prefixes, to save power.

Another reason to direct messages to prefixes instead of directly to destination tasks is the multi-message case. A *multi-message* is a message send which gets sent to multiple destination tasks. Multiple destination tasks can be configured in *nsvc\_msg\_prefix\_id\_lookup*(); multiple destination tasks are bound to a message prefix. This allows the app developer to send global messages such as restart notifications, shutdown notifications, etc. to all tasks at once, for example. Naturally, the system must have a large enough message block pool to accommodate the block allocations required by a multi-message send. On a medium to large system, multi-message capability is a handy feature in the hands of the app developer.

# Message Abort and Task Kill Features

*Message Aborting* is a feature integrated into all blocking kernel API calls, except for the message-get calls. When a high-priority message is sent to a task, the Message Abort feature causes the task to unblock. *Task Killing* is when a task terminates another task, when the other task has become unresponsive or is not working properly. A task kill is a means of resetting a single task as an alternative to warm-starting the entire firmware image.

Message aborting and task killing are related features and are therefore controlled by the single compile switch *NUFR\_CS\_TASK\_KILL*. The NUFR distribution’s default compile switch settings has *NUFR\_CS\_TASK\_KILL* turned off, as the features it controls are more relevant on larger codebases, and carry a small CPU execution cycle penalty when enabled.

## Message Abort

The following API functions support message aborting:

*nufr\_sleep()*

*nufr\_bop\_waitW()*

*nufr\_bop\_waitT()*

*nufr\_sema\_getW()*

*nufr\_sema\_getT()*

*nsvc\_mutex\_getW()*

*nsvc\_mutex\_getT()*

The message abort works the same on all these APIs: an extra parameter called *abort\_priority\_of\_rx\_msg* is included in the argument list for the aforementioned APIs. The value specified in the parameter indicates a message priority, the highest message priority which will *not* cause a message abort. Any message of a priority higher than the *abort\_priority\_of\_rx\_msg* will cause the call to abort when that priority message is received.

### Message Abort Instructional

// Assume Task 1 and Task 2 are same task priorities.

// Assume Task 1 gets launched from somewhere, and

// Task 2 must be launched from Task 1

//

// Assume NUFR\_TID\_TASK1 and NUFR\_TID\_TASK2

// Assume NUFR\_MSG\_PRI\_HIGH == 1 and

// NUFR\_MSG\_PRI\_CONTROL == 0

void task1\_entry(unsigned parm)

{

nufr\_launch\_task(NUFR\_TID\_TASK2, 0);

// Task 1 is still scheduled; Task 2 is

// ready, but won’t get scheduled until

// nufr\_sleep call causes Task 1 to block

// Sleep for an arbitrarily long time

nufr\_sleep(NUFR\_MSG\_PRI\_HIGH, 1000000000);

// Task 1 was made ready by message abort;

// the nufr\_sleep was cancelled by the 2nd

// message send.

// Task 1 arrives here because of Task 2’s

// call to nufr\_yield(). Task 1 terminates

// before Task 2.

}

//

void task2\_entry(unsigned parm)

{

// This message send does not cause a

// message abort of Task 1 of its nufr\_sleep

nufr\_msg\_send\_argsW(NULL,

0, 1, // dummy data, for brevity

NUFR\_MSG\_PRI\_HIGH,

NUFR\_TID\_TASK1,

0);

// This message send does cause a

// message abort of Task 1 of its nufr\_sleep

nufr\_msg\_send\_argsW(NULL,

0, 1, // dummy data, for brevity

NUFR\_MSG\_PRI\_CONTROL,

NUFR\_TID\_TASK1,

0);

// Allow Task 1 to continue

nufr\_yield();

// Finally, Task 2 terminates after Task 1 terminates

}

## Task Kill

A task kill is accomplished by calling the API *nufr\_kill\_task*(). Since this API requires more processing time than other NUFR API calls, it should not be called from an interrupt handler. Apart from the CPU time it takes to complete this call, there is no functional reason why it couldn’t be called from an interrupt handler. It could be called from the SysTick Timer handler, but only if the system developer has deemed that the CPU time required to complete a task kill won’t cause an overrun of the OS tick period, or whether a non-repeating overrun of the OS tick period would be detrimental enough to cause a system failure. One fixed restriction, however, is that a task cannot kill itself. It makes no sense to do so anyways.

Simply calling *nufr\_kill\_task*() will kill a task, but there is much to consider, as killing a task is not as simple as merely calling *nufr\_kill\_task.* In order for a system to continue working smoothly after a task is killed and restarted, all aspects of the task must’ve been cleanly cleared and reset. Specifically:

* All semaphores and mutexes that the task has taken must be given back
* The task’s message queue must be cleared, and all message blocks on the queue must be freed
* All other message blocks which the task owns must be freed
* Any memory blocks—specifically SL particles and SL pool objects—must be returned
* SL app timers must be killed
* Any tasks which are waiting on the task for some sort of ack, such as waiting on a bop, must be ack’ed
* Many of the global variables that the task uses must be reset
* If the task manages a hardware peripheral or talks to an off-chip peer of some sort, some resets or notifications may be appropriate

As part of *nufr\_kill\_task,* the kernel does as much as cleaning as possible. In addition to cleaning up the kernel objects which are hidden to the app developer, the kernel does these things:

* Searches any semaphore whose owner is the task being killed, and releases those semaphore, removing the task’s ownership. Note that only semaphores operating as binary semaphores can be cleaned correctly.

The list is short, due to the kernel’s difficulty in determining what needs to be cleaned; only the application itself knows what needs cleaning, and, in fact, most times this requires specialized logic embedded in the task being killed to do the missing cleanup.

## The TYWYLH Message

A *Tell-Your-Wife-You-Love-Her* (TYWYLH) Message is a message sent by the killing task to the task being killed informing it that it is about to be killed, that it needs to prepare itself for that by doing the necessary cleanup, in the hope that the task will have cleaned up the missing items that the kernel will not clean, so that the task alone can be restarted rather than the entire image. In many instances, this is the only way a task kill can achieve a clean kill, one that won’t necessitate a system restart.

Of course, the problem with sending a TYWYLH message is that the task which is being killed might be blocked on a kernel call of some sort. Therefore, the task being killed should have been written so that every blocking call it makes sets the abort priority parameter in the API call to allow the highest priority message (pre-named *NUFR\_MSG\_PRI\_CONTROL*) to cause a message abort, so that the task being killed will get unblocked from the offending blocking API call, and (if coded correctly) will be able to retrieve the TYWYLH message and perform the cleanup. Also, it may be prudent for the task being killed to monitor return values of abortable API calls, so that the task can immediately curtail its processing and commence with the task shutdown logic.

## SL Pool Manager

The SL Pool Manager feature allows for like-sized data structures to be centrally organized in a pool, from where they can be allocated and freed. The SL Pool Manager is a compromise between having the *malloc*()standard library call and having no memory allocation facilities whatsoever. It’s not unusual to have no *malloc* or to forbid the use of *malloc*  in an RTOS-based codebase. But often the need arises for some sort of memory allocation service.

The SL Pool Manager has the following features:

* Like-sized data structures are made globally available to any task and to interrupt/exception handlers for allocation
* The allocation and freeing of data objects is faster and more debug-able than what *malloc* could do
* If a task attempts to allocate an object, and the pool is empty, the task has the option of blocking until an object becomes available. In this way, a task can make allocation calls which are guaranteed to return an object.

There can be multiple SL Pool Management objects, each managing a single SL Pool. An SL Pool requires the below:

* A static structure of type *nsvc\_pool\_t* in the application code. This is the pool manager.
* An array of objects which will comprise the objects in the pool. This can be a statically defined array or an array from a single malloc call.
* Each data object must have 4 bytes reserved in a fixed offset in the structure for a linked-list pointer (*flink*).
* Each SL Pool Manager requires one semaphore from the fixed SL semaphore pool (see *NUFR\_SEMA\_POOL\_START, NUFR\_SEMA\_POOL\_END* in *nufr\_sema\_t).*

An example pool manager instantiation with its initialization:

#define POOL\_OBJECT\_SIZE 100

#define NUMBER\_POOL\_ELEMENTS 10

typedef struct pool\_object\_t\_

{

struct pool\_object\_t\_ \*flink;

buffer[POOL\_OBJECT\_SIZE];

} pool\_object\_t;

pool\_object\_t object\_array[NUMBER\_POOL\_ELEMENTS];

nsvc\_pool\_t pool\_manager;

void pool\_init(void)

{

pool\_manager.base\_ptr = object\_array;

pool\_manager.pool\_size = NUMBER\_POOL\_ELEMENTS;

pool\_manager.element\_size = sizeof(pool\_object\_t);

pool\_manager.element\_index\_size = (unsigned)

((uint8\_t \*)&object\_array[1] –

(uint8\_t \*)&object\_array[0]);

pool\_manager.flink\_offset =

OFFSETOF(pool\_object\_t, flink);

nsvc\_pool\_init(&pool\_manager);

}

This creates a pool manager object and the objects being managed. Allocations from this pool in general use the API *nsvc**\_pool\_allocateW*() or *nsvc\_pool\_allocateT*(), if from task level. If an allocation needs to be made from an interrupt/exception handler, a call to *nsvc\_pool\_allocate*() should be used, as *nsvc\_pool\_allocateW/T* cannot be called from an exception handler.

# Particles

*Particles* (abbreviated *pcls*) are individual memory buffers that are designed to be chained together in a linked list. A chain of particles is presented to the app layer as a virtual contiguous memory buffer. The app developer accesses a particle chain through the particle APIs, doing reads and writes to the chain. As a side benefit, particles have the capability of dynamically increasing in size as data writes append bytes to the chain. The nature of particles allows for an efficient use of RAM in a system that requires large blocks of memory which are non-uniform in length. A specific application for particles is for packet data storage in a network stack. In a system which is tight on RAM, the use of particles can allow a greater number of packet buffers to be in circulation, where that number could not exist apart from the use of particles. So the typical use-case for particles is on small RAM footprint systems that require network connectivity. For low-powered systems, it may not be advisable to use particles, as using particle chains require extra CPU cycles than single buffer equivalents. Whether to use particles or to use a memory pool full of worst-case sized buffers is a textbook tradeoff, one which the system developer must decide when matching features and applications with platforms.

## Configuring and Initializing SL Particles

Particle support is an SL feature, being located in the file *nsvc-pcl.c.* There are a couple of settings in the SL app settings (*nsvc-app.c/h*) files which configure SL particle support. Particle support can be included or excluded at compile time by including or excluding *nsvc-pcl.c.* There can only be one particle manager instance, and this instance is configured with the variables *NSVC\_PCL\_SIZE* and *NSVC\_PCL\_NUM\_PCLS,* which are both found in *nsvc-app.h.* The particle manager is initialized by calling *nsvc\_pcl\_init*(). Under the covers, the particle manager uses a single instance of a SL Pool Manager to manage the allocation and freeing of the individual particles. Therefore, when particle support is compiled in, the particle subsystem consumes one semaphore from the SL semaphore pool (noted in the SL Pool Manager).

## Basic Particle API Reads and Writes

The simplest way to use particles is to use the top-level APIs and to not attempt to use the lower-level APIs until you’ve mastered the top-levels. Particle APIs work something like file APIs. In place of a file handle is a particle seek structure (type *nsvc\_pcl\_chain\_seek\_t*). This structure keeps track of where the last read or write ended in a chain, so that multiple reads or writes to the same chain using the same seek pointer will step through the chain linearly and contiguously. It’s permissible to have multiple seek structures operating on the same chain at the same time—as long as one understands what they’re both doing.

To create a particle chain, reserve memory for a seek structure and simply write to the chain like this:

// Seek struct memory reserved here by variable

// definition, rather than creation of a ptr.

nsvc\_pcl\_chain\_seek\_t seek\_struct;

nsvc\_pcl\_t \*chain\_ptr;

void start\_chain\_with\_3bytes(void)

{

uint8\_t write\_data[] = {1, 2, 3};

nsvc\_pcl\_write\_dataWT(&chain\_ptr,

&seek\_struct,

write\_data,

sizeof(write\_data),

NSVC\_PCL\_NO\_TIMEOUT);

}

Calling *nsvc\_pcl\_write\_dataWT*() with a null chain pointer will cause a new chain to be created and at the same time will initialize the seek struct (in the example, the variable *seek\_struct*) to point to the end of the first write. In the above example, the chain is created. The new chain will only consist of one particle, since only three bytes are written. In the example, these three bytes are the values *1, 2,* and *3*.

For this first write, the particle subsystem allocates one particle. At the time of allocation, if there are no free particles available in the particle pool, then the task currently running will block indefinitely, due to the use of the parameter *NSVC\_PCL\_NO\_TIMEOUT,* until one becomes available. Since *nsvc\_pcl\_write\_dataWT*() has this blocking feature, the call can only be made from a task context, since only tasks can block. If a particle write operation needs to be made from an exception handler context or from the BG Task, a lower-level particle API function must be used, one that’s callable from an exception handler. The system developer will determine if it is prudent to do so.

The chain, now that it has been created, can have a string of bytes (a single byte in the example below) appended to it by calling the same API again and using the same seek structure:

void append\_1byte(void)

{

uint8\_t write\_data[] = {4};

nsvc\_pcl\_write\_dataWT(&chain\_ptr,

&seek\_struct,

write\_data,

sizeof(write\_data),

NSVC\_PCL\_NO\_TIMEOUT);

}

Data can be appended in this way by making successive calls to this API. The limitation to the amount of data that can be appended is the size of the particle pool. Under the covers, the SL will allocate and append particles to the existing chain as needed, and the seek structure will get updated to point to the current particle in the chain and the offset in that particle.

Note that the first particle in a chain has bytes reserved in it as a master chain header (type *nsvc\_pcl\_header\_t).* This is automatically reserved when the first particle is created.

Continuing with the example, one may read from this new particle by creating a second seek structure, initializing it to an offset in the chain’s virtual data array, and reading:

nsvc\_pcl\_chain\_seek\_t read\_seek;

uint8\_t single\_byte\_read(unsigned offset)

{

bool seek\_success;

unsigned bytes\_read;

uint8\_t read\_byte = 0;

seek\_success =

nsvc\_pcl\_set\_seek\_to\_absolute\_offset(

chain\_ptr,

&read\_seek,

offset);

if (seek\_success)

{

bytes\_read =

nsvc\_pcl\_read(&read\_seek,

&read\_byte,

1);

// ignoring ‘bytes\_read’ for brevity’s sake

}

return read\_byte;

}

Note that the call to *nsvc\_pcl\_set\_seek\_to\_absolute\_offset*() skips over the chain head particle’s header reserved space and does a seek as though this reserved space doesn’t exist, meaning that a seek to the offset of zero positions the seek struct to the first data byte, not to the chain header. The lower-level APIs require the caller to manually adjust the offset to skip over the chain header.

## Particles in Exception Handlers

The use of particles is attractive on small RAM footprint systems. When working with a tiny amount of RAM, RAM savings must be gleaned from whatever means possible. Normally, a driver which receives packets over a serial line has a few bytes of hardware storage to cache the bits received off the receive (rx) end of the serial line in a FIFO. In a worst-case scenario, in terms of hardware capabilities, each byte received will generate an interrupt, and the interrupt handler will have to take that byte out of the hardware peripheral register and store it somewhere. Typically, that “somewhere” is a circular buffer, a RAM FIFO dedicated to the receive line, a buffer large enough to hold an entire packet and some extra, in case the CPU is tied up with other computations when a packet is being received and cannot service the interrupt in a timely manner. On a small RAM system, this circular buffer consumes a significant percentage of the total system RAM budget, and therefore the system developer may consider eliminating the circular buffer to save RAM. To do this in a particle-based NUFR system, it may be practicable to have the interrupt handler write each rx byte directly to a particle, eliminating the circular buffer. Therefore, an example in the file *./examples/examples-pcl-irq-handler.c* has been provided.

# App Timers

App timers are one of the most important NUFR features. An app timer is a timer which is managed by the SL app timer component. When the app timer expires, a pre-formed message is sent to an application.

App Timers are available in the SL as a compiled feature. The feature allows the app developer to create his or her own timers. The timers send the task which created them a message when they expire.

This example creates a task which wakes up periodically due to the timer going off:

#include “nsvc-api.h”

#define FOO\_ID\_TIMER 1 // better to use an enum

nsvc\_timer\_t \*foo\_timer;

// Some task called ‘foo’

void foo\_task\_entry(unsigned parm)

{

uint16\_t id\_uint16;

foo\_timer = nsvc\_timer\_alloc();

// Send the first message after 2 secs

foo\_timer->mode = NSVC\_MODE\_SIMPLE;

foo\_timer->duration = 2 \* MILLISECS\_PER\_SEC;

foo\_timer->msg\_fields =

NSVC\_TIMER\_SET\_ID(FOO\_ID\_TIMER);

foo\_timer->msg\_parameter = 0;

foo\_timer->dest\_task\_id = NUFR\_TID\_null;

nsvc\_timer\_start(foo\_timer);

while (1)

{

// For brevity, ignoring message prefix

nsvc\_msg\_get\_argsW(NULL, NULL,

&id\_uint16, NULL, NULL, NULL);

// Was this message the timer expiration message?

if (FOO\_ID\_TIMER == id\_uint16)

{

// Restart timer, using 10 sec interval

foo\_timer->mode = NSVC\_MODE\_SIMPLE;

foo\_timer->duration = 10 \* MILLISECS\_PER\_SEC;

nsvc\_timer\_start(foo\_timer);

}

}

}

A few assumptions made in the example. First, the one-time call to initialize the timer subsystem, namely the API *nsvc\_timer\_init*(), isn’t shown. Note that *nsvc\_init()* does not have to be called to initialize app timers—only *nsvc\_timer\_init()* is necessary.

App timers are created and used by the app developer. No kernel or SL object, or anything in the kernel or SL app components, has to be changed to create a timer. In a large codebase, this is reassuring, as there is a reluctance to modify the OS app files.

## Clock Source

The app timer component must be attached to either a periodic clock, such as the OS tick handler, or to a quantumtimer. A *quantum timer* is a hardware timer and IRQ handler associated with that timer which work in conjunction with the app timer component to handle app timer timeouts. Since app timers work by queuing active timers in the order in which they’ll expire, the quantum timer is set to the timeout of the next timer to expire. When each timer expires, the quantum timer is reconfigured to the next timer’s timeout interval. In this way, app timers can be used on systems without an OS clock. This is a valuable feature on battery-powered devices, as app timers send messages to tasks when they timeout, thereby satisfying most task timing requirements for systems which have no OS tick (or even those which do). In an event-driven system, when there are no events to handle, the system goes to sleep; the quantum timer wakes the system up when the next app timer expires. No polling is needed.

Another advantage of app timers driven by a quantum timer source is that each app timer can be configured down to a 1 millisecond resolution. Even if an OS tick is used in a system, the OS tick rate can be lowered, saving CPU cycles. For example, if the OS tick rate can be lowered to 50 milliseconds per tick (*NUFR\_TICK\_PERIOD*), but tasks can receive timing events resolved to a single millisecond of resolution.

## Hooking Up Clock Source to App Timer Component

The system developer will hook up the app timer to its timer source. Each time this API is called, the expired timers are processed:

nsvc\_timer\_expire\_callin()

It is the responsibility of the system developer to provide BSP-level or equivalent functions which the app timer module can use. These functions are:

* A 32-bit revolving time reference, scaled to milliseconds
* If a quantum timer is used, the quantum timer reconfiguration callback.

Both of the these APIs are inserted with *nsvc\_timer\_init().*

# Synchronous Messaging

Synchronous message sending is a design pattern whereby a Task A makes a query to a Task B. Task A blocks waiting for the query to complete, then proceeds. While Task A is blocking, Task B receives the request, processes it, then unblocks Task A. The synchronous message constitutes a transaction where Task A is the client and Task B is the server. Here’s an example of a synchronous message call:

typedef struct

{

unsigned x;

unsigned outcome;

} sync\_parms\_t;

// This runs in Task A’s context

void task\_A\_function(unsigned value)

{

unsigned result;

// ‘synchronous\_api\_call()’ declaration put in

// Task B’s .h file, not Task A’s .h file

// The entire synchronous message send appears looks

// like a simple function call from Task A’s

// point of view.

result = synchronous\_api\_call(value);

// (take action based on ‘result’)

}

// This call runs in Task A’s context, but is

// located in Task B’s file, not Task A’s file

// This API can be called from any task, not just

// Task A. It is a reentrant API call.

unsigned synchronous\_api\_call(unsigned parameter)

{

// ‘sync\_parms’ exist on Task A’s stack, and

// will be made available to Task B.

sync\_parms\_t sync\_parms;

uint16\_t key;

sync\_parms.x = parameter;

sync\_parms.outcome = 0;

// Store ‘sync\_parms’ in Task A’s TCB, so Task B

// can access it.

nufr\_local\_struct\_set(&sync\_parms);

// Mutex used here. This is only necessary for a

// handful of corner cases, but shown here to

// make this example complete.

nsvc\_mutex\_getW(SYNC\_API\_CALL\_MUTEX, NUFR\_NO\_ABORT);

// The key is necessary in case the nufr\_bop\_waitT()

// call below times out before Task B can process the

// request, and Task A calls this API again.

// In this case, the key will cause Task B to ignore

// the first request message when it finally gets

// the chance to process it.

key = nufr\_bop\_get\_key();

// Task B must run at same priority or lower than

// Task A in order for this to work.

// If Task B were at a higher priority, then Task

// A would block on msg send, not bop

//

// If we wanted to have client calling tasks at

// lower priority levels than Task B making this call,

// We would have to eliminate the use of the local

// struct feature.

nsvc\_msg\_send\_argsW(NULL,

0,

ID\_SYNC\_API\_CALL,

NUFR\_MSG\_PRI\_MID,

bop\_key);

// Task A blocks on the bop wait,

// until Task B releases it or a timeout occurs.

nufr\_bop\_waitT(NUFR\_NO\_ABORT, 100);

nsvc\_mutex\_release(SYNC\_API\_CALL\_MUTEX);

// Task B wrote into ‘sync\_parms.outcome’,

// and this gets passed back to Task A

return sync\_parms.outcome;

}

void task\_B\_entry(unsigned parm)

{

uint16\_t id\_uint16;

nufr\_tid\_t source\_task;

sync\_parms\_t \*sync\_parms\_ptr;

uint32\_t bop\_key;

while (1)

{

// We can process many messages here, but

// we’re only concerned about the one message.

//

// We must retrieve ‘source\_task’ here,

// and not assume request came from Task A.

// In this example, ‘source\_task’ will be

// Task A’s TID

nsvc\_msg\_get\_argsW(NULL,

NULL,

&id\_uint16,

NULL,

&source\_task,

bop\_key);

if (ID\_SYNC\_API\_CALL == id\_uint16)

{

// Must lock Task A to prevent Task A’s

// call to ‘nufr\_bop\_waitT()’ from allowing

// Task A to unblock due to a timeout.

// If Task A had already timed out of call

// at this point, Task B won’t enter ‘if’

// construct.

if (NUFR\_BOP\_RTN\_TAKEN ==

nufr\_bop\_lock\_waiter(source\_task,

(uint16\_t)bop\_key))

{

sync\_parms\_ptr = nufr\_local\_struct\_get(

source\_task);

//

// Process request

// (use sync\_parms\_ptr->x here)

...

sync\_parms\_ptr->outcome = 1;

// Task A is free to timeout now.

// It might be a better idea to do

// two bop locks and unlocks, if Task B’s

// processing takes a long time

nufr\_bop\_unlock\_waiter(source\_task);

// Ack Task A: we’re complete

nufr\_bop\_send(source\_task,

(uint16\_t)bop\_key);

}

else

{

// Assume that if waiter lock failed,

// Task A has timed out. Won’t need to

// send bop in this case.

}

}

}

This example demonstrates the challenges of doing a synchronous message send with a timeout for the API caller/client (Task A). A synchronous message call with a timeout (*nufr\_bop\_waitT()*) for the caller is a difficult use-case, especially when the NUFR local struct feature is used. Note the restriction on task priorities in the example’s comments. An alternative to having the timeout in the calling environment (using *nufr\_bop\_waitW*() instead of *nufr\_bop\_waitT*()) may be to manage a timeout in Task B’s context. This has its limitations too.

# Pipes

NUFR—at least this release—has no pipes. Should pipe support be added, it’ll be added to the SL. For those who need pipes, there’s an implementation for them in *example\_pipe.c*.List of Files

List of file names in NUFR distribution, along with their categories. Concerning the categories:

* The category *Export File* specifies files which are designated as include files for the application source files to compile against
* *Plat App* and *SL App* specifies files that are to be/must be modified by the application developer
* *Plat* specifies files which are/optionally can be modified by the system developer

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ***File name*** | ***OS Layer*** | | | ***Service Layer*** | | ***Export File*** |
|  | ***Kernel*** | ***Plat*** | ***Plat App*** |  | ***SL App*** |  |
| raging-global.h |  |  |  |  |  | X |
| nufr-compile-switches.h |  | X |  |  |  | X |
| nufr-context-switch.s | X |  |  |  |  |  |
| nufr-kernel-base-task.h | X |  |  |  |  |  |
| nufr-kernel-task.h | X |  |  |  |  |  |
| nufr-kernel-task.c | X |  |  |  |  |  |
| nufr-kernel-base-messaging.h | X |  |  |  |  |  |
| nufr-kernel-message-blocks.h | X |  |  |  |  | X |
| nufr-kernel-messaging.c | X |  |  |  |  |  |
| nufr-kernel-message-blocks.c | X |  |  |  |  |  |
| nufr-kernel-message-send-inline.h |  |  | X |  |  |  |
| nufr-kernel-semaphore.h | X |  |  |  |  |  |
| nufr-kernel-semaphore.c | X |  |  |  |  |  |
| nufr-platform.h |  | X |  |  |  | X |
| nufr-platform.c |  | X |  |  |  | X |
| nufr-platform-export.h |  | X |  |  |  | X |
| nufr-platform-import.h |  | X |  |  |  |  |
| nufr-platform-app.h |  |  | X |  |  | X |
| nufr-platform-app-compile-switches.h |  |  | X |  |  | X |
| nufr-platform-app.c |  |  | X |  |  |  |
| nufr-api.h | X |  |  |  |  | X |
| nufr-sanity-checks.c |  |  | X |  |  |  |
| nsvc.h |  |  |  | X |  |  |
| nsvc.c |  |  |  | X |  |  |
| nsvc-globals.c |  |  |  | X |  |  |
| nsvc-messaging-bpool.c |  |  |  | X |  |  |
| nsvc-messaging.c |  |  |  | X |  |  |
| nsvc-mutex.c |  |  |  | X |  |  |
| nsvc-pool.c |  |  |  | X |  |  |
| nsvc-pcl.c |  |  |  | X |  |  |
| nsvc-timer.c |  |  |  | X |  |  |
| nsvc-api.h |  |  |  | X |  | X |
| nsvc-app.h |  |  |  |  | X | X |
| nsvc-app.c |  |  |  |  | X |  |

# Exported Defines, Macros, and APIs

This is a list of all the defines, macros, inline functions, and API functions which are intended to be exported to and thereby made available to the application code.

### Defines and Macros

Macros have parameters listed in parentheses. Note this list only includes the NUFR and SL files—it doesn’t include contents of non-NUFR files, such as *raging-global.h*.

|  |  |
| --- | --- |
| ***Define*** | ***Description*** |
| NUFR\_CS\_LOCAL\_STRUCT | Compile switch to enable local structs |
| NUFR\_CS\_MESSAGING | Compile switch to enable kernel messaging |
| NUFR\_CS\_MESSAGE\_PRIORITIES | Number of message priority levels |
| NUFR\_CS\_TASK\_KILL | Compile switch to enable the task kill feature |
| NUFR\_CS\_SEMAPHORE | Compile switch to enable kernel semaphores |
|  |  |
|  |  |
| NUFR\_MILLISECS\_TO\_TICKS( millisecs) | Convert milliseconds value to OS clock tick closest equivalent |
| NUFR\_SECS\_TO\_TICKS( seconds) | Convert seconds value to OS clock tick closest equivalent |
| NUFR\_TICK\_PERIOD | Interval in milliseconds of each OS clock tick |
|  |  |
| NUFR\_LOCK\_INTERRUPTS | Lock macro used by kernel, made available to app code also |
| NUFR\_UNLOCK\_INTERRUPTS(register) | Unlock |
|  |  |
|  |  |
| NUFR\_NO\_ABORT | Used where API parameter abort\_priority\_of\_rx\_msg' appears, to specify no message aborting on this API call |
| NUFR\_MSG\_MAX\_PRIORITY | Max integer value of message priority |
| NUFR\_MSG\_MAX\_PREFIX | Max integer value of message prefix |
| NUFR\_MSG\_MAX\_ID | Max integer value of message ID |
| NUFR\_MSG\_MAX\_TASK\_ID | Max integer value of sending task ID (TID) |
| NUFR\_GET\_MSG\_PRIORITY(fields) | Extract message priority bits from nufr\_msg\_t->fields value |
| NUFR\_SET\_MSG\_PRIORITY(fields, value) | Insert message priority bits into nufr\_msg\_t->fields value |
| NUFR\_GET\_MSG\_SENDING\_TASK(fields) | Extract message sending task TID bits from nufr\_msg\_t->fields value |
| NUFR\_SET\_MSG\_SENDING\_TASK(fields, value) | Insert message sending task TID bits into nufr\_msg\_t->fields value |
| NUFR\_SET\_MSG\_FIELDS( prefix, id, sending\_task, priority) | Insert all bits into nufr\_msg\_t->fields value |
| NUFR\_GET\_MSG\_ID(fields) | Extract message ID bits from nufr\_msg\_t->fields value |
| NUFR\_SET\_MSG\_ID(fields, value) | Insert message ID bits into nufr\_msg\_t->fields value |
| NUFR\_GET\_MSG\_PREFIX(fields) | Extract message prefix bits from nufr\_msg\_t->fields value |
| NUFR\_SET\_MSG\_PREFIX(fields, value) | Insert message prefix bits into nufr\_msg\_t->fields value |
| NUFR\_GET\_MSG\_PREFIX\_ID\_PAIR( fields) | Extract contiguous message prefix and ID bits from nufr\_msg\_t->fields value |
| NUFR\_SET\_MSG\_PREFIX\_ID\_PAIR( prefix, id) | Insert contiguous message prefix and ID bits into nufr\_msg\_t->fields value |
| NUFR\_KERNEL\_MESSAGE\_SEND\_INLINE( task\_id, msg\_prefix, msg\_id, msg\_priority, fixed\_parameter) | Fast message block allocate and message send. Intended to be used from IRQ handler |
| xxx\_NO\_LOCK(xxx) | Same as above but without interrupt locking |
| NUFR\_NUM\_TASKS | Number of tasks |
| NUFR\_MAX\_MSGS | System-wide number of message blocks |
| NUFR\_NUM\_SEMAS | Number of semaphores |
| NUFR\_SEMA\_POOL\_SIZE | Number of semaphores made available to SL, for use in SL objects which require semaphores |
| NSVC\_NUM\_MUTEX | Number of mutexes |
| NSVC\_TMDIV\_NUMBER | Number of SL timer divisors |
|  |  |
|  |  |
| NSVC\_PCL\_SIZE | Data storage bytes available in particle which is not chain head |
| NSVC\_PCL\_SIZE\_AT\_HEAD | Data storage bytes available in particle which is chain head |
| NSVC\_PLC\_NUM\_PLCS | System-wide number of particles |
| NSVC\_PCL\_NO\_TIMEOUT | When used for certain particle API calls, means timeout mode not selected |
| NSVC\_PCL\_HEADER(head\_pcl) | Pointer to chain header from chain head particle |
| NSVC\_PCL\_OFFSET\_PAST\_HEADER(offset) | Translate packetized offset to absolute offset |
| NSVC\_PCL\_SEEK\_DATA\_PTR(seek\_ptr) | “uint8\_t \*” pointer to data from a seek ptr |
|  |  |
| NSVC\_TIMER\_SET\_ID\_PRIORITY | Used in SL timer API calls |
| NSVC\_TIMER\_GET\_ID | Used in SL timer API calls |
| NSVC\_TIMER\_GET\_PRIORITY | Used in SL timer API calls |

## API Functions

List of NUFR kernel, NUFR platform, NUFR platform app, SL, and SL app API functions. These are the ones that can/should be used by the app developer, not ones intended to be used by the OS itself.

|  |  |
| --- | --- |
| ***API Name*** | ***Description*** |
| nufr\_init | NUFR kernel and platform layers init |
| nufrplat\_task\_get\_desc | Get task descriptor |
| nufr\_msg\_get\_block | Allocate a message block, lowest level |
| nufr\_msg\_free\_block | Free a message block |
| nufr\_msg\_free\_count | Get NUFR platform message block free count |
|  |  |
| nufr\_launch\_task |  |
| nufr\_kill\_task | One task kills another task |
| nufr\_self\_tid | Get current running task’s Task ID |
| nufr\_task\_running\_state | Get running state of a task |
| nufr\_sleep |  |
| nufr\_yield | Schedule next task of same priority |
| nufr\_prioritize | Set running task to highest task priority |
| nufr\_unprioritize |  |
| nufr\_change\_task\_priority |  |
|  |  |
| nufr\_bop\_get\_key | Get a new bop key for this task |
| nufr\_bop\_waitW | Wait on a bop |
| nufr\_bop\_waitT | Wait on a bop with timeout |
| nufr\_bop\_send | Release another task waiting on a bop |
| nufr\_bop\_send\_with\_key\_override | Same as *nufr\_bop\_send,* but disregard key |
| nufr\_bop\_lock\_waiter | Lock a task waiting on a bop |
| nufr\_bop\_unlock\_waiter |  |
|  |  |
| nufr\_local\_struct\_set | Store a pointer in this task’s TCB. Pointer points to a stack-resident structure |
| nufr\_local\_struct\_get | Retrieve another task’s pointer stored by *nufr\_local\_struct\_set* |
|  |  |
| nufr\_tick\_count\_get | Get current number of elapsed OS ticks |
| nufr\_tick\_count\_delta | Find delta between previously saved elapsed OS tick count and current OS tick count |
|  |  |
| nufr\_msg\_drain | Purges queues of message(s) from a task’s message queue |
| nufr\_msg\_purge | Purges a single matching message from a message queue |
| nufr\_msg\_send | Send a message to a task |
| nufr\_msg\_send\_by\_block | Send a message to a task. Requires that caller allocate message block himself/herself. |
| nufr\_msg\_getW | Current running task: get next message from message queue |
| nufr\_msg\_getT | ...same, but with timeout |
| nufr\_msg\_peek | Get pointer to next message on queue, but don’t dequeue the message |
| nsvc\_msg\_prefix\_id\_lookup | Look up task(s) bound to message prefix |
| nsvc\_msg\_bpool\_getW | Get message block from SL message block pool |
| nsvc\_msg\_bpool\_free |  |
| nsvc\_msg\_bpool\_init | Initialize SL message block pool |
| nsvc\_msg\_struct\_to\_fields\_inline | Convert msg parm struct to nufr\_msg\_t->fields value. Inline version. |
| nsvc\_msg\_struct\_to\_fields | ....non-inline version |
| nsvc\_msg\_args\_to\_fields\_inline | Convert msg parm list to nufr\_msg\_t->fields value |
| nsvc\_msg\_args\_to\_fields |  |
| nsvc\_msg\_fields\_to\_struct\_inline | (nverse of above functions) |
| nsvc\_msg\_fields\_to\_struct |  |
| nsvc\_msg\_fields\_to\_args\_inline |  |
| nsvc\_msg\_fields\_to\_args |  |
| nsvc\_msg\_send\_structW | SL message send. Uses parm struct. |
| nsvc\_msg\_send\_argsW | SL message send. Uses parm list. |
| nsvc\_msg\_send\_recirculate | Task sends received message back to itself |
| nsvc\_msg\_send\_multi | Replicate-send a single message to multple task destinations |
| nsvc\_msg\_get\_structW | SL message get. Uses parm struct. |
| nsvc\_msg\_get\_structT | ...same but with timeout |
| nsvc\_msg\_get\_argsW | SL message get. Uses parm list. |
| nsvc\_msg\_get\_argsT | ...same but with timeout |
|  |  |
| nufr\_sema\_count\_get | Get a semaphore’s count value |
| nufr\_sema\_getW | Take a semaphore, decrementing count. |
| nufr\_sema\_getT | ...same but with timeout |
| nufr\_sema\_release | Return a semaphore, incrementing count. |
| nsvc\_mutex\_init | Initialize SL mutex component |
| nsvc\_mutex\_getW | Take a mutex |
| nsvc\_mutex\_getT | ...same but with timeout |
| nsvc\_mutex\_release | Return a mutex |
|  |  |
| nsvc\_pool\_init | Initialize SL generic pool management service |
| nsvc\_pool\_is\_element | Verify if pointer points to an object for a given pool |
| nsvc\_pool\_free | Free object to the pool |
| nsvc\_pool\_allocate | Allocate an object, low-level |
| nsvc\_pool\_allocateW | Main SL API to allocate an object |
| nsvc\_pool\_allocateT | ...same but with timeout |
|  |  |
| nsvc\_pcl\_init | Initialize SL particle service |
| nsvc\_pcl\_is | Checks if a pointer points to a particle |
| nsvc\_pcl\_free\_chain | (low-level particle API) |
| nsvc\_pcl\_alloc\_chainWT | Allocate a particle chain |
| nsvc\_pcl\_lengthen\_chainWT | Append more particles to an existing chain |
| nsvc\_pcl\_chain\_capacity | Get chain storage capacity |
| nsvc\_pcl\_count\_pcls\_in\_chain | (low-level particle API) |
| nsvc\_pcl\_write\_data\_no\_continue | (low-level particle API) |
| nsvc\_pcl\_write\_data\_continue | (low-level particle API) |
| nsvc\_pcl\_write\_dataWT | SL particle chain write |
| nsvc\_pcl\_contiguous\_count | (low-level particle API) |
| nsvc\_pcl\_seek\_ffwd | Advance a seek struct by so many bytes |
| nsvc\_pcl\_set\_seek\_to\_packet\_offset | Set a seek struct to an offset relative to packet start in chain |
| nsvc\_pcl\_set\_seek\_to\_headerless\_offset | Set a seek struct to a native offset |
| nsvc\_pcl\_read | SL particle chain read |
|  |  |
| nsvc\_timer\_get\_divisor\_table\_entry | Get attributes of a SL timer divisor |
| nsvc\_timer\_init | Initialize SL app timer component |
| nsvc\_timer\_alloc | Allocate an app timer |
| nsvc\_timer\_free | Free an app timer |
| nsvc\_timer\_start | Start an app timer |
| nsvc\_timer\_kill | Kill an app timer before it expires |
| nsvc\_timer\_expire\_callin | Handle app timer timeouts. Usually called by OS tick handler or H/W timer ISR. |