NUFR’s Tiny Mode and Porting to the ARM Cortex M0

Copyright © 2019, Bernie Woodland

All rights reserved.

Redistribution and use, with or without

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

1. Redistributions must retain the above copyright notice, this

list of conditions and the following disclaimer.

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

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

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

Arm® is a trademark of Arm Holdings

Arm® Cortex® is a trademark of Arm Holdings

Table of Contents

[Introduction 4](#_Toc28873652)

[ARM Cortex M0 vs. M3/M4 4](#_Toc28873653)

[File Differences 4](#_Toc28873654)

[PRIMASK vs. BASEPRI on M0 4](#_Toc28873655)

[General Guidelines for Minimizing FLASH Footprints 5](#_Toc28873656)

[Function Call Inclusions with the Linker 5](#_Toc28873657)

[NUFR Compile Switches 5](#_Toc28873658)

[The SysTick Callin 5](#_Toc28873659)

[A Tickless NUFR That Uses SysTick 6](#_Toc28873660)

[And Don’t Forget… 6](#_Toc28873661)

[Totals 6](#_Toc28873662)

[General Guidelines for Minimizing RAM Consumption 6](#_Toc28873663)

[The RAM Cost of Task Stacks 6](#_Toc28873664)

[Using a Single Event-Driven Task 7](#_Toc28873665)

[Using a Lower-Priority Work Task 7](#_Toc28873666)

[Using the Background Task 8](#_Toc28873667)

[Outgrowing This Dual-Task Model 8](#_Toc28873668)

[Building for the Tiny Model 9](#_Toc28873669)

[Some Other Thoughts 9](#_Toc28873670)

# Introduction

This document explains how to make a minimally sized (“tiny”) installation of NUFR and also describes the in’s-and-out’s of running on the ARM Cortex M0.

# ARM Cortex M0 vs. M3/M4

The Cortex M0 is based on the ARMv6 architecture, whereas the other M-Series/Cortex M3 and M4 are based on the ARMv6 architecture. The M0 has limitations that cause breakages across the M-series family, so that special adaptations have to be made for it.

## File Differences

There are certain files which must be substituted in an M0 build, instead of the one usually used on an M3/M4 build. These files are:

|  |  |
| --- | --- |
| *M3/M4 File* | *M0 Equivalent File* |
| ./platform/ARM\_CMx/gcc/assembly.c | ./platform/ARM\_CMx/gcc/assembly-m0.c |
| ./platform/ARM\_CMx/gcc/nufr-context-switch.s | ./platform/ARM\_CMx/gcc/nufr-context-switch-m0.s |
| ./platform/ARM\_CMx/gcc/armcmx-utils-mem.c | ./sources/raging-utils-mem.c |
|  |  |
|  |  |

## PRIMASK vs. BASEPRI on M0

The M0 has no BASEPRI register. This means that the PRIMASK register must be used instead for interrupt locking. To do this, enable the compile switch *USE\_PRIMASK* in the nufr platform code (in *./small-soc/nufr-platform-import.h* or *./tiny-soc/nufr-platform-import.h*.

Using the PRIMASK instead of the BASEPRI register means that there’ll be a loss of feature or a limitation in the M0 codebase. Using BASEPRI allows the system developer to reserve high interrupt priority levels and to assign interrupt priorities using such high priority levels so that certain interrupts will not be affected by NUFR interrupt locking. This cuts back on worst-case latency for these high priority interrupts. By using BASEPRI, all IRQs will be disabled in a NUFR critical section.

There appears to be a bug or limitation in GCC’s inline assembler. The inline assembler, while passing variables into or out of an assembly section, violates M0’s index range limitation, causing a compile error. For this reason, the PRIMASK register value cannot be saved locally and restored. This causes a limitation in nesting interrupt locking/critical sections—which is not a good programming practice in the first place. Since NUFR relies uses interrupt locks extensively in its kernel, the application developer should not have interrupts locked when making a NUFR API call.

Often, application developers use critical sections unnecessarily or use them when other alternatives are available. Using APIs like *nufr\_prioritize()* instead of interrupt locks will solve many of these problems.

# General Guidelines for Minimizing FLASH Footprints

## Function Call Inclusions with the Linker

The linker, with any sort of optimizations applied, will only include in an image those functions which actually get called somewhere else in the codebase. If a function never gets used, it’s a no-brainer to exclude it, in order to save on text size (FLASH space). Now, since most other ARM-based RTOS’s wrap their kernel internals inside of an SVC call, this wrapping tells the linker that that functionality is always used—when in fact it often is not used. For this reasons, most RTOS’s have multiple compile switches; otherwise, their kernels would be too big.

Since most of NUFR’s kernel code is processed at a top-level API, and not through an SVC call, NUFR doesn’t need any compile switches to cut down on its size under any sort of optimization. The linker will simply omit a NUFR API which isn’t used. An advantage of this architecture is that the myriad of compile switches (and their code generators, etc.) are eliminated. The application developer needs to be mindful that whenever he or she calls a new API—any API—it could pull in a pyramid of supporting function calls, and this could cause the text space to balloon. For this reason, there are a few NUFR compile switches provided. These provide safeguards to prevent inadvertent use of APIs which shouldn’t be used.

## NUFR Compile Switches

There are a handful of compile switches that NUFR has. These are found in *nufr-compile-switches.h.* As stated in the previous section, turning certain compile switches off is optional—assuming that the app developer is cognizant that there’s a price to be paid for the use of each API call—but, for the sake of completeness, they are there. For the tiny model, it’s recommended that semaphore support be disabled, as semaphores in a NUFR codebase are used to instantiate mutexes and to service other inter-task arbitrations. A Tiny Model seeks to minimize the number of tasks, and as a result there’s little need for semaphores.

Other code-saving measures use accomplished by selectively removing or selectively replacing files, rather than by turning on or off compile switches. The SL is an example. To not include the SL, simply do not include the SL files in a build. There is no compile switch to remove the SL.

## The SysTick Callin

The SysTick exception handler is a powerful entity in a resource-constrained environment. Of course, this assumes that a tickless OS is not required. But as far as SysTick is concerned, the usefulness of a periodic callback can be leveraged to save code. So, with the exclusion of SL App Timers, a cheap substitute can be obtained by manually coding all timers in the SysTick NUFR Platform callin. See file *example-tiny-model-systick-callin.c* and how it ties into *./tiny-soc/nufr-platform.c*.

Not only can the SysTick callin be used to count timers, but it can be used to debounce switches. But in either case, the SysTick callin takes a short piece of polled code, like for a timer, and converts it to NUFR message. As long as the code is short, it should be straightforward enough to not cause problems. And once the code sends a message, the logic path goes from being polled to being event-driven. This architectural variant consume little RAM and little text space, is simple to implement, and therefore simple to prototype, test, debug, and harden.

Naturally, at some point the amount of functionality chained in through the SysTick callin can exceed the savings of using SL App Timers, etc., but it’s assumed that a project that uses the Tiny Model won’t have that many timers, etc. in the first place. So if you’ve outgrown this model, you’ve probably outgrown your SoC too.

## A Tickless NUFR That Uses SysTick

Eliminating the OS tick part of NUFR partially—not entirely—will save a good percentage of text space in a Tiny Model footprint. If we were to not call the function *nufrkernel\_update\_task\_timers()*, and assuming that the linker won’t include *nufrkernel\_add\_to\_timer\_list()* and *nufrkernel\_purge\_from\_timer\_list()*, then the NUFR text size would shrink by about 18%. The downside is that functions like *nufr\_sleep()* can’t be used. See manual for more details. But those sort of functions aren’t used much anyways.

## And Don’t Forget…

* Turn off asserts (*CONTRACT\_ENFORCEMENT\_LEVEL* = 0)
* Don’t use *nufr\_sane\_init()* when codebase is stable
* Turn off *NUFR\_CS\_OPTIMIZATIONS\_INLINE*
* Don’t use assembler versions of *rutils\_memset()* and *rutils\_memcpy()*
* In the project-specific makefile, set the cpu type to an M0 or M0+ *(-mcpu=cortex-m0*)

## Totals

Take a look at *./docs/performance-numbers.xls*, the tab *M3 Tiny Model FLASH*. This has a tally of current consumption of a tiny model on the M3. I would expect this to be a bit bigger for the M0, since the M0 lacks the full instructions set of the M3.

# General Guidelines for Minimizing RAM Consumption

Low-end SoC’s which would be candidates for NUFR and its Tiny Model may feature on the order of 4k or 8k of RAM. In addition to low-end SoC’s, RAM consumes power, so low-power devices minimize RAM as well—not just low-cost devices. NUFR and NUFR’s tiny model has been designed to be used on small RAM sized SoC’s. The following are tips for saving RAM.

## The RAM Cost of Task Stacks

In any multi-threaded system, each thread will require its own task stack. Thread stacks (task stacks) consume a large percentage of RAM on a small footprint system. This is perhaps the primary reason why older embedded CPUs were single-threaded, which means they used a scheduler instead of an RTOS. RTOSs assume multiple threads, multiple threads consume RAM—lots of it. On a 4k SoC, it’s not difficult to design a codebase where half of the RAM is dedicated to stacks.

The fact that each thread requires its own stack cannot be altered. What can be changed, however, is the number of threads in the system and the size of each task stack (to a certain degree). The NUFR Tiny Model seeks to get its largest RAM savings by minimizing the number of tasks that a system needs, and by minimizing the number of tasks stacks, the RAM consumption will go down accordingly.

To solve the RAM problem, we must address the task proliferation problem. Without understanding why tasks are unnecessary created, we cannot minimize the number of tasks in a system, and without minimizing the number of tasks, we’ll waste precious RAM in a RAM-scarce environment. So I’ll list the reasons for unnecessary task proliferation:

* App developers are accustomed to having tasks form the architectural dividing line between different subsystems in a project
* There are instances of “thread hogging” which cause intolerable delays in a shared thread, prompting app developers to spin off functionality in stand-alone tasks
* App developers simply do not understand multitasking
* App developers don’t understand real-time computing. In other words, they don’t have a feel for timing, for how long a specific computing task takes to execute, and how this timing has an impact on the real-time requirements of all parts of the system.
* Code is ported from other platforms where thread usage, task allocation, etc. is already established and refactoring the code would be too time consuming, too error prone, or cause other problems

## Using a Single Event-Driven Task

Continuing the thought started in the previous section, it is desirable to create a single event-driven, message-based task (a “message pump” task), and to share this among several state machines. In fact, this single task is the primary task in a Tiny Model architecture. It relies on using several NUFR features in harmony with each other, and having done that, the code will be modular, easily managed, easily tested, easily expanded, easily ported, and use the CPU efficiently.

There are some examples of this alluded to in the NUFR User Manual. Suffice to say, the principle is to divide state machines and other components by message prefix, and direct messages to their final destination by message prefix. Another important element of this design is to apply this rule: message path in this task should take more than a few milliseconds to complete. This rule becomes a contract whereby several components can share this message pump task and all components can meet their timing constraints. In fact, as an app developer becomes more experience, he or she will gain an appreciation of the vast extent of the real-time computations that can be handled by a modern CPU.

## Using a Lower-Priority Work Task

In addition to the message pump task described above, a second task may be useful or necessary to handle some of the things which require a long time to process, and thereby hog the message pump task’s thread. Some examples of these would be:

* Transmission of large log messages out of a 9600 baud serial output
* Certain FLASH erases or writes
* Cryptographic calculations done in firmware, such as cryptographic hashes, signature verification, symmetric key encryption, public key encryption

Of course, one may argue that at least some of these long computations can be broken up into several messages instances in a message pump task. And so it can be—but at a price of code complexity. It is generally simpler not to have to break up a long calculation into several message events.

Having stated these justifications for a second task, this second task we’ll call a “work task.” It will handle the items that take a long time to complete, the items listed above. The work task will run at a lower priority than the message pump task, so that the message pump task will not be impacted by the work task. The work task will receive the spare CPU cycles that the other higher priority threads must have. The constraint is that whatever the work tasks does should have more relaxed timing constraints.

By prudently diving the algorithmic logic and computations between these two tasks, many timing challenges can be tackled. And, of course, this model can be extended to have multiple lower priority tasks, and not just a single one. At a price, of course.

## Using the Background Task

While the work task was explained in the last section, what was not considered is a further RAM-saving alternative: use the Background Task (BG) instead of a work task. Skip the work task entirely and have the BG do what’s described above instead. This saves a single task stack. The instincts of a season developer would be to use a dedicated work task, and not the BG task, but when the boat of RAM consumption is sinking in a storm on the ocean of application bloat, you cannot take such a RAM savings measures off the table. There are cases where, to save RAM and eliminate an additional task, on a RAM-constrained SoC (and I’m thinking of a 4k or less system), this measure will be seriously considered.

## Outgrowing This Dual-Task Model

All of the measures described above are practicable, but as any codebase grows, and as new requirements for new features come in, the need to add more tasks begins pressing upon you. After all, this is the reason to put an RTOS on a tiny SoC: when your application grows, you can grow into the next sized SoC and add those extra tasks.

The Tiny Model becomes outgrown when the codebase exceeds—and I’m totally guessing here—16k of RAM or so. This doesn’t mean that you shouldn’t continue to use some or all of the Tiny Model architectural points, but other considerations become manifest.

First, applications begin needing large chunks of RAM. These large chunks are more efficiently shared between memory pools. So the SL memory pools and SL particles become necessary to save RAM.

Second, when you begin to need things like memory pools and particles, this will require more FLASH space. There is a tradeoff between FLASH consumption, RAM consumption, and CPU cycle usage. So to save RAM, one must use more FLASH. “More FLASH” in this case means including the NUFR SL, so that services such as memory pools and particles can be applied to realize further RAM savings.

In addition to using the SL to save RAM, the SL App Timers are a means to implement timers on tickless OS’s/low –power systems. App Timers require FLASH, but they save CPU cycles by letting you “go tickless.”

# Building for the Tiny Model

The *Tiny Model* is NUFR running in a minimal RAM and FLASH footprint. Tips for this are covered in the user manual. Note that the Tiny Model is not necessarily a tickless-OS model, which is usually the most CPU efficient model. A tickles OS, while minimizing CPU usage, requires a bit more FLASH usage than the Tiny Model envisions.

What I say here is that *./tiny-soc/nufr-platform-import.h* is specifically tuned for minimal installations, and specifically set for use with the M0. Here’s the strategy for the Tiny model:

* Don’t use the Service Layer (SL). Any file named *nsvc-\*.\** is part of the service layer.
* Don’t compile in semaphore support unless you need it
* NUFR, unlike other RTOS’s, doesn’t have a lot of compile switches to contend with. But remember that the compiler will compiler all functions but the linker will only include functions in the build if they get called. Most NUFR API calls won’t get included unless somebody calls them. When you do choose to call a function, all functions under it get pulled into the build, so a single function call be expensive as far as FLASH consumption.
* This applies to the raging utilities also. Be mindful when using them of this “cascading affect” of the linker pulling in function calls.
* With the SL removed, you’ll have to create timers the “old-fashioned” way. See file *example-tiny-model-systick-callin.c* and how it ties into *./tiny-soc/nufr-platform.c*.
* A big savings in the tiny model RAM consumption is by restricting the number of tasks that a project uses.

# Some Other Thoughts

Modern SoC’s—especially ARM Cortex M-based CPUs—have a relative abundance of CPU cycles over RAM and FLASH resources for mains-powered (non-battery-powered) devices.

Once a system gets enough RAM to allow the app developer to add the extra tasks he or she desires, he or she will find that the limitation of FLASH space will be more pressing.