NTNU
Kunnskap for en bedre verden

Department of Engineering Cybernetics

TTK 4550 - Specialization Project

Embedded Lingua Franca and FreeRTOS

Author:
Silje Susort

December, 2022

Table of Contents

Abstract

Lingua Franca is a coordination language implementing a model of computation called the reactor model. The goal of the model is to specify deterministic and easily parallelizable concurrent program design. This project contributes with porting the single-threaded Lingua Franca runtime environment to two new platforms: bare-metal on the FRDM-K22F development board, and to the popular real-time operating system FreeRTOS. Further, the project investigates different characteristics of running Lingua Franca on the new platforms, and compares them against each other. FreeRTOS introduces extra overhead and worse time precision, but provides major advantages in developing more advanced embedded systems that can utilize Lingua Franca.

1 Introduction

1.1 Problem Background

Lingua Franca is a coordination language implementing a model of computation called the reactor model. The goal of the model is to specify deterministic and easily parallelizable concurrent program design. To do this, the reactor model combines principles from other models of computation to create a formally deterministic model with a semantic notion of time.

Lingua Franca has been ported to several different platforms, and can be run with both centralized or distributed coordination. In addition, the Lingua Franca C runtime is very small. This means that using Lingua Franca in embedded, real-time and Internet of Things applications could be very promising. However, its application in these systems is still in a very early phase.

When developing embedded applications, the designer must face a choice of whether to use an operating system in the application. FreeRTOS is a small, widely used real-time operating system which is well-suited for platforms with constrained resources. This makes it a good candidate for exploring new applications for Lingua Franca.

The Lingua Franca runtime needs to be ported to any new platform it should run on. Each platform will have unique characteristics that will benefit different applications. It is therefore of interest to be able to use a wide selection of platforms.

1.2 Problem Description

The objective of this project is to port the Lingua Franca single-threaded runtime to two new platforms:

- Bare-metal on the FRDM-K22F board
- FreeRTOS on the FRDM-K22F board

First, the runtime needs to be implemented such that Lingua Franca functionality works correctly. Important design and implementation choices should be justified and explained. Second, it is of interest how using an operating system affects the time precision, performance and other aspects compared to a bare-metal platform. The two different ports should be compared through a series of tests.

2 Theory

This section provides the relevant theory for the project, focusing on Lingua Franca and the reactor model, real-time operating system theory and FreeRTOS, and lastly an introduction to the the hardware platform and relevant software to be used.

2.1 Terminology

This project and its relevant theory deals with certain classes of computer systems, which need to be classified and defined. The literature uses different terms which are not clearly defined, so this chapter will define different concepts and explain how they closely relate to each other.

Embedded systems
Embedded systems are computer systems that serve some dedicated role within some larger system with physical parts. The scope of embedded systems range from single-task microcontrollers to large distributed control systems. Embedded systems can be found in everything from children's toys to subsystems in space rockets. Embedded systems will often interact with the physical world, and is often (but not necessarily) subject to real-time constraints. Embedded systems may have heavy constraints on computer resources like processing power, available memory and power consumption constraints.

Distributed Systems
A distributed system is a computer system that have sub-systems spread out on different networked computational entities, and communicate by message-passing. The computational entities may be embedded systems or general-purpose computers. They may be geographically or physically separated, but not necessarily. [14] defines the defining characteristic of a distributed system as:

> "A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process."

Distributed systems can be used in everything from process industry control systems, Internet of Things, parallel computation to telecommunication.

Reactive and Cyber-Physical Systems
Reactive systems, or cyber-physical systems, are a class of computer systems that hold a permanent interaction with the physical environment [7]. Specifically, embedded computers monitor and control physical processes, usually by utilizing sensors, feedback loops and computational elements [15]. The computer systems and physical environment mutually affect each other. These systems must be subjected to higher standards of safety, reliability, responsiveness and predictability, as a range of properties from the physical world must be taken into account. This is a broad class of systems, and examples include modern vehicles, control systems, Internet of Things and robotics systems.

It is worth noting that reactive systems have not benefited from progress in programming technology as much as traditional computing [7], as many of the strategies focus on average efficiency instead of (and often at the cost of) repeatability and predictability [15]. Timing correctness is irrelevant to traditional computing, and have been omitted from all software abstractions [15], but is often an important aspect of reactive systems.

Real-time Systems

Real-time systems are computer systems which have to react to input from the environment within finite time intervals, often decided by the environment [7]. These finite time intervals are called deadlines. All real-time systems are also reactive systems [7]. The importance of meeting a deadline may differ for each system task. If a single system task has a deadline, then the whole system can be called a real-time system. Real-time systems are reactive systems with a concept of timing correctness.

We call these timing constraints for real-time constraints. Real-time constraints may be hard, firm or soft [6]. If the system has a hard real time constraint, then a deadline miss means the system has failed [6]. An example of this is a automatic brake-system on a car; if it fails, then the car potentially crashes. If the system has a firm real-time constraint, then a deadline miss means the result is valueless. An example of this is an automated assembly line; if a component is delivered too late, then it is useless. If the system has a soft real-time constraint, then a deadline miss means the results becomes less valuable. An example of this is a system using a sensor measurement; the older the measurement is, the less useful.

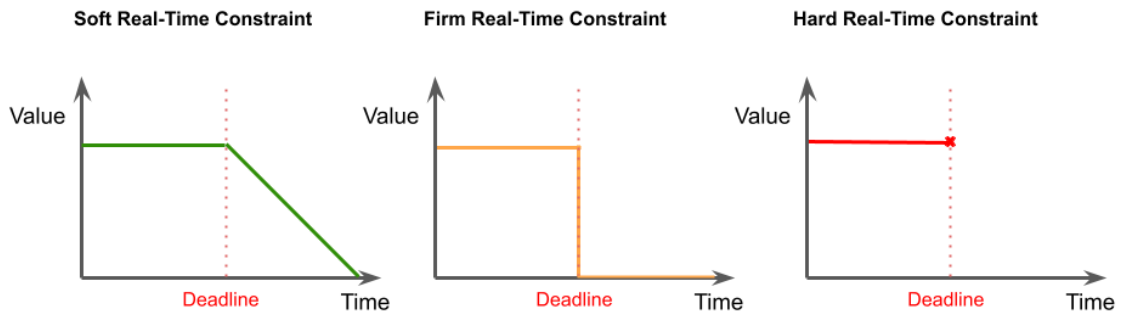


Figure 1: The Figure shows a graphical representation of the value over time for soft, firm and hard real-time constraints.

2.2 The Reactor Model

This chapter is mostly based on Chapters 2, 4 and 5 of the tecnical report "Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems" by Marten Lohstroh, which can be found at [16]. Where further explanations are needed, additional sources will be provided.

2.2.1 Overview

The reactor model is an implementable, deterministic model of concurrent computation for reactive systems, that includes time semantics. An implementation of the reactor model is called Lingua Franca, and will be explained in detail in Chapter 2.3. The model is designed with the philosophy that a model of computation should be as deterministic and reliable as is feasible. The reactor model uses reactors as the elementary structures; each containing blocks of computation, message-passing mechanisms and internal state. A single block of computation is called a *reaction*. A reactor's behavior is decided by it's reactions, and a reaction may trigger and be triggered by discrete *events*. When a reaction executes, it has mutually exclusive access to the reactor's internal state. If several reactions can execute at any one time instant, then a specific ordering is imposed by the model.

Events are time-tagged values, and are the only means of communication between individual reactors. Events must be processed in timestamp order. Reactors observe and produce events through *ports*, forming a hierarchical structure. A single time tag is composed of a time value and a *microstep index*. The sequence of monotonically increasing tags present in the program, represent the *logical* timeline. Logical time is different from *physical* time, which is the traditional notion of time from the physical world.

Actions are a port-like mechanism that may produce events within a reactor. If the origin is from the execution of some reaction, then we call it a logical action, while if the origin is from outside the program, then we call it a physical action. The events produced from actions will be tagged with either a logical or physical time value, depending on if it is a logical or physical action.

2.2.2 Ports and hierarchy

Reactors are connected through input and output *ports*. For example, if reactor A needs to observe an event from reactor B, then reactor A's input port need to be explicitly connected to reactor B's output port. The event is then produced on B's output port, and observed on A's input port. In this way, communication between reactors are indirect. An important implication from this structure is that dependencies between reactors are explicit.

A reactor may contain other reactors, and is then called a *composite* reactor. Composite reactors may contain other composite reactors. Thus, a hierarchical structure is present. Connections between ports may only traverse one hierarchy layer. All

reactor programs are organized inside a top-layer reactor called the main reactor. The main reactor is the only reactor without any ports.

Connections between reactors contained in a composite reactor is defined by the composite reactor. Thus, this explicit dependency information can be used to enforce deterministic ordering of reactions. Having this information available is useful for deterministic ordering, but also for exposing parts of the program that are able to run in parallel.

2.2.3 Time

The reactor model utilizes two different timelines; logical and physical time. Physical time is the traditional notion of time, and will be the actual time of when events are observed in the physical world, ie. measuring from a platform-specific hardware clock. The measured time value will vary nondeterministically based on many real-world factors, like interrupts and physical speed limitations. In physical time, there is no notion of simultaneity, and repeating a run will yield different results each time. When the reactor model needs a physical time measurement, it is treated as an input to the system. In this way, determinism in the system is preserved.

Logical time is a sequence of monotonically increasing tags. A tag consists of a time value and a microstep index. While the time value must have a time unit that corresponds to the physical time unit (ex. nanoseconds), the microstep index is unitless. That is because it is used to express order for simultaneous time values. To explain this, imagine a collision in Newton's pendulum device. Energy is conserved in the collision, as it is transferred from the colliding sphere to the outermost sphere on the other side. Roughly, one can imagine that this transfer happens simultaneously through all the middle-spheres until it reaches the outer one. At the same time, there must be an order to the events, as the energy is transferred from one sphere to the next. The microstep can then be used to infer the order in this simultaneous collision. This is relevant for the reactor model as a single event may trigger several other events simultaneously in logical time, but still with a well-defined order. This model of time is called *superdense*. Two tags are only equal if both the time value and the microstep index are alike.

Logical time is a way of modelling time as in an ideal system, without the nondeterministic qualities of physical time measurements. This concept is borrowed from synchronous-reactive principles, see [7]. The main idea of synchronous-reactive systems are that systems that produce outputs *synchronously* with its inputs, are much easier to describe, analyze and compose. Furthermore, communication and internal actions are instantaneous. If these properties are met, then a qualitatively correct behavior can be defined. The claim from [7] is that the physical time consumption of the implementation is another, conceptually separate issue. Thus, in the reactor model, logical time does not progress during a reaction, meaning that outputs have the same logical time value as the inputs.

Physical time can be placed on the logical timeline by transforming it into a logical time tag and then comparing it to logical time values. The two timelines will be

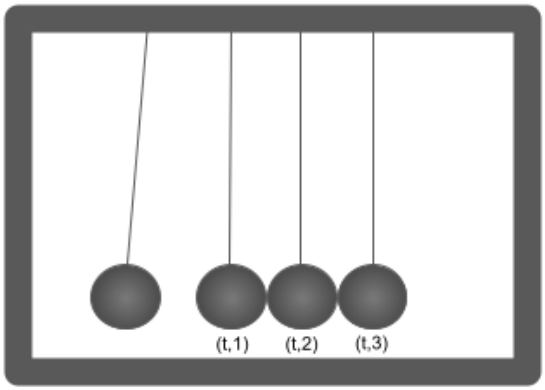


Figure 2: The Figure shows a visualization of the concept of an order in a simultaneous event. When a collision happens in a Newton's cradle, the forces travels through the spheres to the last one which will go upwards. This is observed as instantaneous.

synchronized at certain points, such that logical time will always "chase" physical time.

By using both of these timelines, deterministic computational behavior and non-determinism from the real-world execution of the program are elegantly interleaved such that the model of computation will be deterministic. A program expressed in the reactor model of computation may express timing behaviors, which may be essential to the correctness of the system. For example, deadlines can be set and checked if it is missed.

Timers
Timers are logical actions that are scheduled with periodic intervals.

Deadlines
Deadlines are time intervals in which a reaction must occur within. Deadlines use logical time to formulate a time interval, and physical time to check whether this has been maintained. Setting deadlines enables the designer to also handle deadline misses, which may require fault handling in the user program. This is typically useful for all kinds of real-time systems, for example with sensing and actuation timing.

2.2.4 Actions

Actions are internal ports that may trigger events for a reactor. The tag of the scheduled event from an action is computed based on whether the action origin is physical or logical. If it is a logical action, then the associated event will be assigned a tag from the well-defined logical time instant of the action, plus a minimum delay and an additional, optional delay. Because of this, a logical action can only be scheduled from inside a reaction. The minimum delay for a logical action is one

microstep, and thus, can never give rise to causality loops[1].

If it is a physical action, then the associated event will be assigned a value and tag from outside the program. This means that the time value will be an input to the system and originates from some physical clock, plus a minimum delay and an additional, optional delay, as with logic actions. A physical action can thus be used to capture nondeterministic values as *inputs* (preserving model determinism, see definition at Chapter 2.2.5), like sensor measurements, clock measurements, and other events from the physical environment. Physical actions can be scheduled at any time according to something in the physical world, for example from an ISR. This is useful since we are dealing with cyber-physical systems, and must therefore be able to handle asynchronous events.

2.2.5 Determinism

Determinism is defined by [16] as

> "A system is deterministic if, given an initial state and a set of inputs, it has exactly one possible behavior."

The reactor model is deterministic in the sense of the above definition, with *inputs* and *behavior* defined in a clever manner. As we saw in Chapter 2.2.4, whenever external measurements are included with physical actions, it is in the form of *inputs* to the model.

The reactor model is inherently deterministic in the sense that the semantics itself define valid programs to be programs which only produce one behavior for a given set of initial states and inputs. If this is not true, then the program is not a valid program after the reactor model. Mathematically, this can be proven by formulating all reactor systems as a function with a number of inputs and outputs, and showing that the behavior of the system will always reach a unique least fixed point. If the least fixed point exists, then a behavior for the program exists. If it is unique, then only one behavior for the program is possible. Any reactor program which do not comply with this will not comply with the reactor model semantics. The semantics are explained in detail in [16].

The content of the reactions in a reactor is excluded from the analysis of determinism. The whole model is only deterministic if also the content of the reactions are deterministic. However, reactions provide a way of integrating nondeterminism where it is necessary and/or useful, for example for random number generation.

It is worth noting that determinism is a property of the model, not of a physical realization of the model. Any physical realization will always have nondeterminism. Having a deterministic model to build your system after is a good idea since it clearly defines the correct behavior, and anything that deviates from it can be classified as a fault. This increases the system reliability and testability.

[1] Causality loops are when dependencies loop in such a way that the program has no well-defined order.

2.2.6 Scheduling and Parallelism

As reactors are isolated objects with their own data and statically declared dependencies between them, they provide an easily parallelizable structure. The scheduling of reactions are constrained by the following conditions in order to preserve determinism:

- If two reactions within one reactor are triggered simultaneously, then they must execute in the order that they are defined within the reactor
- If a reaction A declares an antidependency[2] to reaction B in another reactor, then A must execute before B starts executing

The second condition is conservative in the sense that A *may* produce an event that B is dependent on, but, regardless of if it does or not, they still have to execute in this order. This means that, stictly speaking, the program could be more parallel. The benefits of doing it this way is that the potential dependencies are always clear, and that the reaction contents do not need to be analyzed at all.

2.2.7 Distributed Execution

The reactor model may be used on a distributed system of spatially separate hosts, where reactors need to communicate with each other through a network. This is called a federation or federated reactor.

The main challenge of distributed execution is how to coordinate deterministic execution across multiple physical timelines. The multiple timelines originate when execution happens on several platforms with different physical clocks. Each physical clock will vary from another by some amount, since each clock does not keep precise physical/real time. With this setup, it can be impossible to tell whether one event happened before another [14].

A globally correct ordering of timestamped events is possible using logical clocks [14]. The reactor model already have the building blocks to define a global ordering of tags, valid across a distributed system. However, to do this, two key assumptions are needed. First, a bounded clock synchronization error E. This error is an assumed bound on the maximum synchronization error between clocks on different hosts. Second, a bounded message delivery latency L. This latency is an assumed maximum possible latency when delivering messages on the network connection. The benefit of formulating explicit bounds on assumptions is that it makes it possible to detect when they are violated. This makes it possible to create a fast and fault-tolerant decentralized coordination scheme.

[2] An antidependency is simply the opposite relation of a dependency: If B depends on A, then B is an antidependency of A, and B is a dependency of A.

2.3 Lingua Franca

2.3.1 Project Overview

Lingua Franca (LF) is an open-source project with the aim of creating a coordination language with basis in the reactor model. It was started in 2019 and is a collaboration project between groups from several universities. It is still under development, and more features are added continuously [17].

A coordination language is a programming language that only handles the coordination between blocks of computation (blocks of sequential code), instead of the computation itself [15]. Lingua Franca allows for creating a network of reactors, as explained in the previous chapter. The reaction contents, which is the computation part, is written in any target language supported. Currently the languages C, C++, Python, TypeScript and Rust are supported as target languages, with more under development. The advantage of using separate target languages is that the user may utilize existing knowledge and code in a programming language they are more familiar with.

The LF compiler translates the LF code and generates code purely in the target language. The application code is then combined with the LF runtime, which is also written in the target language. The result is a deterministic standalone program. The runtime implementation handles the interactions between the program and the underlying platform or operating system, as well as implementing the functionality of reactors. It needs to be implemented for each supported target language. More on the LF runtime in Section 2.3.5.

Lingua Franca has a command-line interface, its own Eclipse-based IDE called Epoch, and an extension for Visual Stuido Code. If using the last two alternatives, interactive diagrams of the LF program is created automatically. A test suite is available to run thorough testing of functionality as well as compilation.

Figure 3: The Figure shows the Lingua Franca logo. Taken from the LF website at [20].

2.3.2 Features of Lingua Franca

There are a number of benefits to using Lingua Franca and the Reactor model. They can be summarized as follows:

- *Deterministic coordination.* Enables deterministic interactions between model components. A deterministic model will drastically reduce the number of

possible behaviours compared to a nondeterministic model, and is a property that is generally ideal to have. LF still allows for nondeterminism where suitable.

- *Modularity.* Enables a modular model of concurrency that is easy to parallelize, and thus is suitable for distributed computing.
- *Dependencies statically declared.* Dependencies between reactors are part of the definition. This has the advantage that it is impossible to forget to declare a dependency.
- *Clear assumptions for distributed coordination.* Uses static assumptions on the timing behaviour to be able to optimize both communication speed and efficiency. These assumptions provide a basis for fault detection, as it is easy to detect when the assumptions are broken, and thus a fault has occured.
- *Semantic concept of time.* Enables expressive control of timing behaviour by utilizing different physical and logical timelines. Physical time will represent input from some hardware clock, while logical time will represent program timing. Deadlines may be easily represented with combining these features.

2.3.3 Installation

Lingua Franca can be set up in three different ways [17]. The first alternative is using the included IDE; Epoch IDE, which is based on Eclipse IDE. The second alternative is through a Visual Studio Code extension. Both of these alternatives have the advantage that the automatically generated graphs are configured automatically. The last alternative is as a command line tool. All the alternatives are available for download at the Lingua Franca website [20]. It is possible to use Lingua Franca on Windows, Linux or MacOS. Not all of the download options are available for Windows.

For developers, the project is openly available to clone or fork on github. The command line LF compiler can be modified and built from scratch using a ready-made shell script.

When the code has been compiled (by the LF compiler), the auto-generated code can be found in a folder called *src-gen*. Normally, the built executable will be found in a folder called *bin*.

2.3.4 Syntax and Diagrams

LF uses its own syntax for creating the components of the reactor network. The full syntax can be found at [20], but the following section will give an solid introduction. At the top level, an LF program consists of a target declaration, a main reactor and possibly a number of other reactors.

The target declaration must specify the target language to use, but has several optional parameters as well. Examples of optional parameters are specifying custom

build scripts to run, warning settings, whether to use threaded execution or not, number of threads to use, how long the program should run before it times out, and if it should run as fast as possible. There are many more optional parameters. An example of a C target declaration is provided in Figure 4.

```
target C {
    threading: false,
    build: "custom-build.sh",
    timeout: 10 secs,
};

```

Figure 4: The figure shows an example of a target declaration in LF. The declaration uses the C target language, no threading, a custom build script and exits the program after 10 seconds have elapsed.

The main reactor is the program entry point and is the top level reactor that contains other instances of reactors, if any. It creates the individual instances of each reactor and declares the dependency relations between them. The main reactor is a valid reactor in itself, so it may also have reactions and states. However, the main reactor is the only reactor that cannot have inputs or outputs [17]. Figure 5 provides a simple example of a main reactor. In federated programs, the main reactor is replaced with a federated reactor, and signals that the program should be split up by reactor instances and executed on a distributed system.

The target code contained in reactions are separated from LF code with the special delimiters {= .. =}. The code inside these delimiters are simply pasted directly into the auto-generated code when compiled. This means that the content of reactions are not checked for syntactic or logical correctness at all. Some Application Programming Interface (API) functions are accessible from within the delimiters, as well as access to the reactor data variables.

```
target C;

main reactor {
    reaction(startup) {=
        printf("Hello World.\n");
    =}
}

```

Figure 5: The figure shows an example of a valid main reactor that simply prints "Hello World." at startup.

Another example of a reactor network is shown in Figure 6. Here, the main reactor creates an instance of each reactor, and declares that ExampleReactorA may produce an output that ExampleReactorB is dependent on. ExampleReactorA has a single output, and a reaction at startup which prints "Hello". It then sets the output variable firstWordPrinted to "true". The fact that the reaction could produce an output must also be explicitly declared, using a "->", since it is an *internal*

dependency. If this is not done, then the LF compiler will generate an error. Upon receiving firstWordPrinted, ExampleReactorB prints " world!". The printed output of the whole program becomes "Hello world!".

```
target C;

reactor ExampleReactorA {
    output firstWordPrinted:bool;
    reaction(startup) -> firstWordPrinted {=
        printf("Hello");
        lf_set(firstWordPrinted, true);
    =}
}

reactor ExampleReactorB {
    input firstWordPrinted:bool;
    reaction(firstWordPrinted) {=
        printf(" world!");
    =}
}

main reactor {
    ExampleReactorA = new ExampleReactorA();
    ExampleReactorB = new ExampleReactorB();

    ExampleReactorA -> ExampleReactorB;

}

```

Figure 6: The figure shows an example of a reactor network.

If the program needs code that is outside any specific reactor, then one can create a preamble section. Typically, the preamble will contain necessary C library or file includes, interrupt handlers or general functions. The preamble can be included in any reactor, but will be available for all reactors within the same file. In the C target, the preamble can also be put outside a reactor (it will be totally equivalent). An example of a preamble is given in Figure 7.

A reactor may have physical or logical actions, which will be used to schedule triggers according to physical or logical time instants, respectively. A physical action could be scheduled at any time, for example from an interrupt. An example of this is given in Figure 7. Here, a asynchronous button press from the environment triggers an interrupt, which then uses a physical action to schedule a trigger at that time instance.

Diagrams Auto-generated diagrams is a feature built using KIELER Lightweight Diagrams Framwork [17]. These diagrams are an easy-to-use feature and can help the programmer in a concrete way by visualizing the program structure. The diagrams can directly alert the programmer of some mistakes, like dependency loops. The diagrams are also interactive, such that the programmer can show more or less details, as well as click on elements of diagram and highlights the corresponding code. An example of an auto-generated diagram of a hierarchical reactor network

```
preamble {=
    #include "board.h"

    #define BUTTON_PORT           BOARD_SW2_PORT
    #define BUTTON_GPIO_PIN       BOARD_SW2_GPIO_PIN
    #define BUTTON_IRQ_HANDLER    BOARD_SW2_IRQ_HANDLER

    void* button_press_action;

    void BUTTON_IRQ_HANDLER(void) {
        PORT_ClearPinsInterruptFlags();
        lf_schedule(button_press_action, 0);
    }
=}

```

Figure 7: The figure shows an example of a preamble section which includes a file, creates macros and defines an interrupt handler for a button press.

is shown in Figure 8. The square boxes are the reactors, the arrow-like boxes are reactions and the black triangles are ports. Connections are shown as lines.

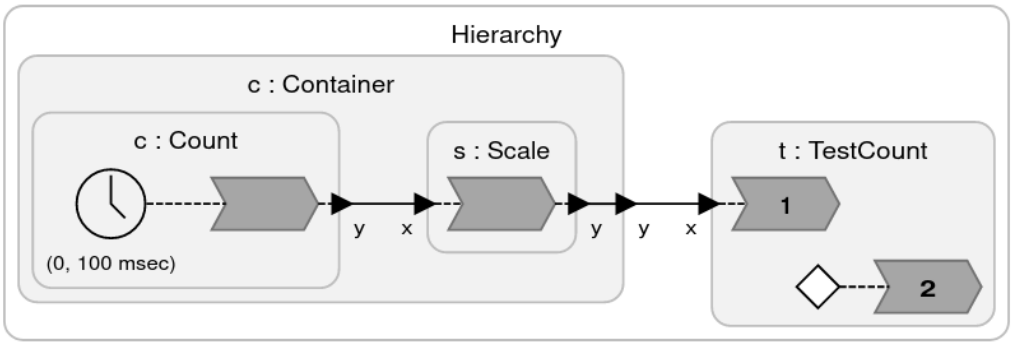


Figure 8: The Figure shows an example of a auto-generated diagram for an LF program. The figure is taken from the LF website at [20].

2.3.5 Runtime Environment

In general, a runtime, runtime system or runtime environment is the collection of necessary code required to implement the features of a language. It is a structure that brings another layer of abstraction, meaning the user of the language will not need to study the implementation, only use the API.

The LF language is built on such a runtime environment, and thus it is this collection of code that actually implements the reactor model. A runtime is written in each of the supported target languages, and must be ported to different platforms. Therefore, a specific runtime consists of a general part, as well as a platform-specifc part. The platform-specific files must implement core functions that use the underlying platform, like setting up and using the LF clock, sleeping functionality and thread handling (if multithreaded). There are 18 platform-specific functions for

multi-threaded LF and 7 for single threaded LF. The general part then utilizes these functions to implement the reactor model logic.

The C runtime is the implementation with the least overhead, and can therefore be used on heavily resource-constrained systems, like embedded applications. It consists of around 2000 lines of code, and is ported to various OSes and some bare-metal platforms [17].

The general structure of the C runtime is shown in Figure 9. The most relevant files are platform.h, reactor.c, reactor.h, reactor_common.c and the contents of the "platform" folder.

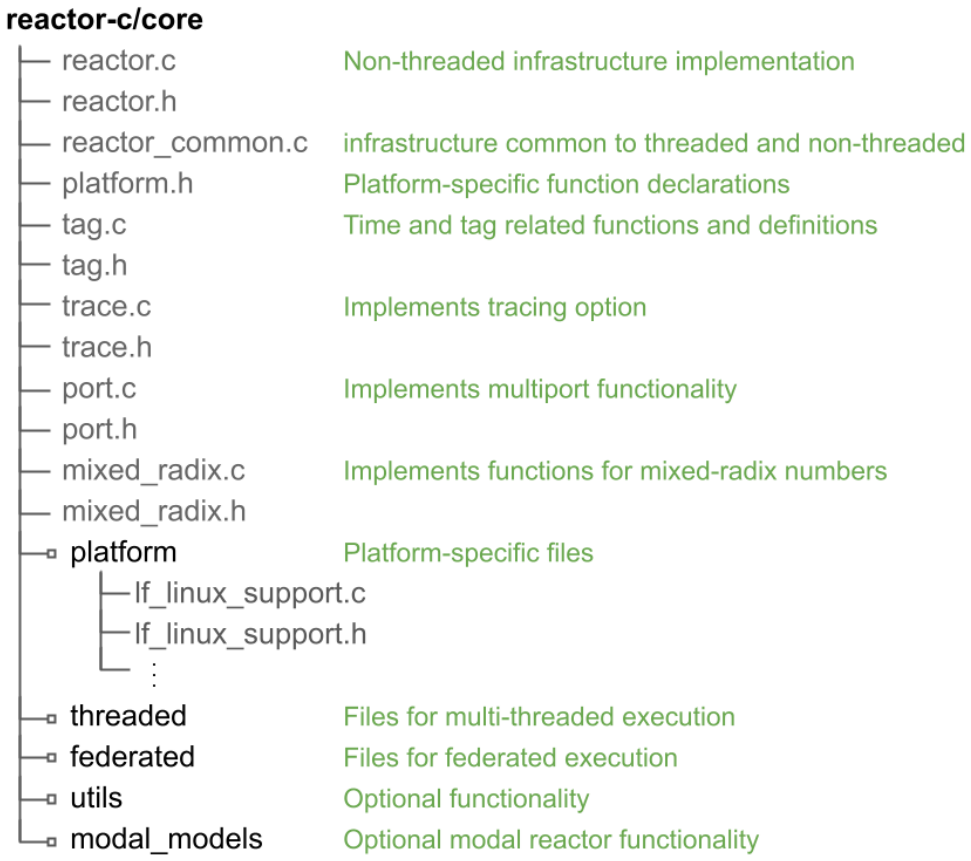


Figure 9: The Figure shows the file and folder structure of the C runtime.

2.3.6 LF Compiler

The Lingua Franca compiler generates standard target code from the LF program. It is written in Java. The LF compiler can validate the LF program and return compilation error messages if not valid. The auto-generated target code includes the necessary LF runtime files, and is then compiled as normal with the target compiler. The full compilation flow can be seen in Figure 10.

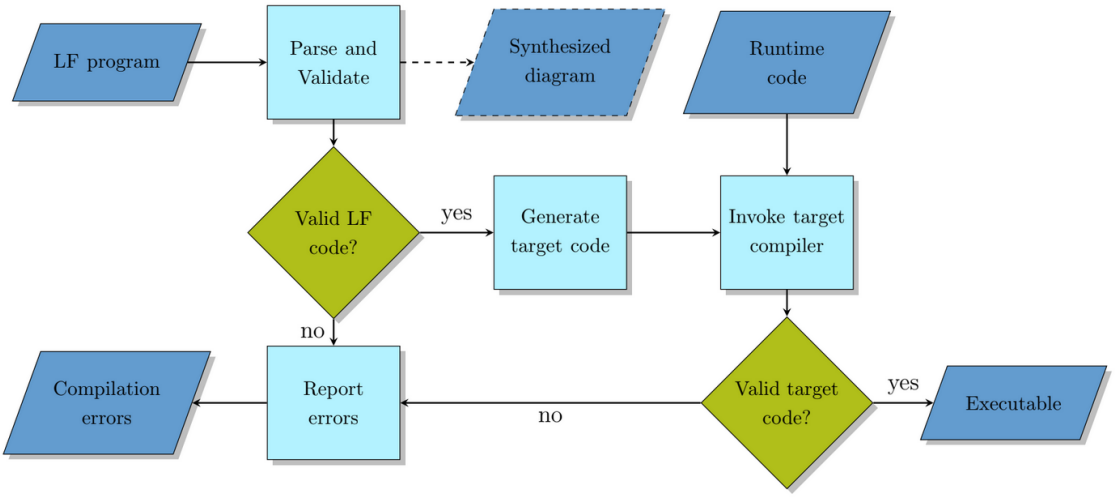


Figure 10: The Figure shows a flow chart of the LF compilation structure. The flow chart figure is taken directly from [16].

2.4 Real-Time Operating Systems

2.4.1 Overview

Real-Time Operating Systems (RTOS) are small (or at least scalable) operating systems [6] which may run on highly resource constrained applications. RTOSs handle time-critical tasks with processing deadlines and may provide predictable timing properties [6] (as opposed to general purpose operating systems). The RTOS kernels are configurable such that the needs of specific applications may be met. The kernel is the core functionality of the operating system [6]. For an RTOS, predictability is much more important than speed/performance.

Using RTOS may be a wise alternative to bare-metal software if the application is complex enough[3][5]. Using an RTOS means that the kernel will provide an interface for concurrency and synchronization, provide timing APIs and easily configure the software design [5] [6]. This could make the software more modular, portable, maintainable, testable and efficient [5].

Some potential downsides to using RTOSs could be that it uses too much system resources for a constrained system and may be complex to edit beyond what is configurable through the RTOS API.

2.4.2 Tasks

A task is also called a process or thread[4], and is the entity that can be executed by a processor. It contains a set of instructions and a block of data, plus a process control block which contains necessary process administration information for the OS [18]. The task can be modeled to exist in different states, signaling if it is available to the processor to run, if it is waiting for some event to happen, if it has been swapped into memory and more. The simplest task model is just two states; ready or not ready [18], while some operating systems can have a much more complicated task model.

2.4.3 The scheduler

Scheduling is the activity of deciding which tasks should run on a processor over time [18]. There are different decisions involved in this, like which tasks which should be handled or not, which should be swapped out and which of the ready tasks should run. We can view this as scheduling over different time frames; long-term, medium-term and short-term [18]. When speaking of scheduling in this project, it will mostly refer to short-term scheduling.

[3]What is too complex or not is mostly a subjective judgement or preference, and may depend on other requirements and knowledge. Writing bare-metal code certainly requires low-level programming knowledge, which may not be a given.

[4]The lack of term consensus is a root of confusion since the terms are also used for conceptually different things [5]. A thread can also be a path of execution inside an individual process.

Short-term scheduling can use different strategies that will optimize different aspects of the systems behavior [18] [6]. This can be, for example, to optimize for processor utilization, keeping task deadlines or task priorities.

A scheduling policy can be preemptive or nonpreemptive [18]. If it is preemptive, then the scheduler can interrupt running tasks and switch them out. This can be when some new task arrives, on an interrupt or system call, or when a certain period of time has passed. Time-slicing is when each task is given a period of time to execute, until it will possibly be interrupted and switched out. If the scheduling policy is nonpreemptive or cooperative, then the task will run until it has completed (run-to-completion), or until it voluntarily lets itself be switched out.

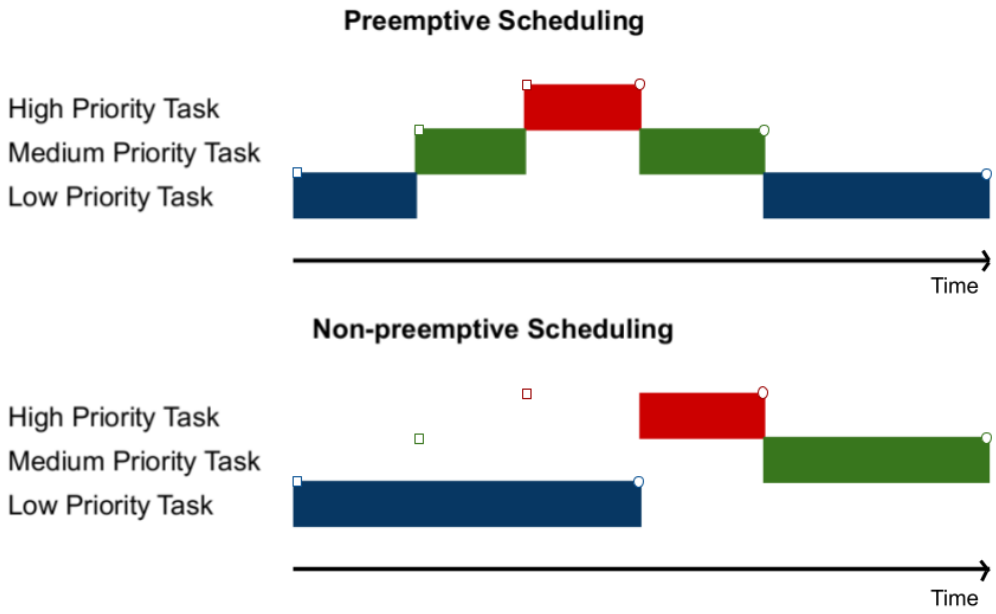


Figure 11: The Figure shows a visualization of two scheduling strategies. Preemptive scheduling interrupts and switches out lower priority tasks when high priority tasks become ready. Nonpreemptive scheduling lets the tasks finish, and then choosing the highest priority task to run next. Squares signify when the task is ready, and circles signify when the task is finished.

2.4.4 Memory Management

Memory management is an important and complex task in an operating system. It concerns how to allocate and share the memory resource for several tasks [18].

If the memory is partitioned into equal size blocks, then it is very fast to place a requested block of memory [18]. However, a phenomena known as internal memory fragmentation may happen [18], as the requested blocks won't be the same size as the fixed block. See *a)* turn into *b)* in Figure 12. Memory allocations will fail to find enough contiguous free memory, even if there exists enough memory in total. To mitigate this, several strategies can be used. An alternative is to use variable size fixed blocks, but then the placement algorithm will have to do more, time-consuming work in finding the block that fits best.

External fragmentation is another way memory fragmentation can happen [18]. After allocating and deallocating memory, several empty blocks of free memory will be spread out, and the same consequences happen as with internal fragmentation. See *c)* turn into *d)* in Figure 12, where some blocks have been freed, but a memory block of size two cannot be allocated. To combat this, blocks need to be merged, which requires further time-consuming administration work.

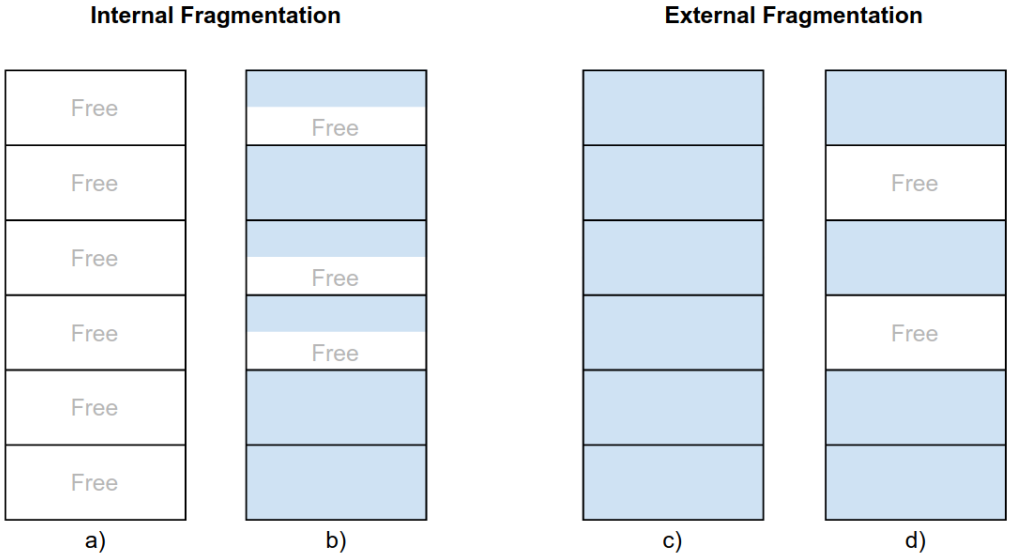


Figure 12: The Figure shows a visualization of internal and external memory fragmentation. *a)* shows memory which is partitioned into equal size blocks. In *b)* memory of different sizes have been allocated. In *c)* all blocks have been allocated. In *d)*, some blocks have been freed.

2.5 FreeRTOS

Note: The following chapter was written in conjunction with an assignment in TTK8, which was to present and compare two real-time operating systems; FreeRTOS and Zephyr RTOS. The presentation of FreeRTOS has been extracted and modified for this report. The section is mostly based on the book "Mastering the FreeRTOS Real Time Kernel", at [5], but also FreeRTOS' official online documentation.

FreeRTOS was launched back in 2003, and is today the market leading RTOS [4]. FreeRTOS is described on their website [3] as:

> "Developed in partnership with the world's leading chip companies over an 18-year period, and now downloaded every 170 seconds, FreeRTOS is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of IoT libraries suitable for use across all industry sectors. FreeRTOS is built with an emphasis on reliability and ease of use."

It was originally developed by Real Time Engineers Ltd., but later bought up by Amazon Web Services [2]. It is Open Source with a MIT license. This is a permissive software license which places few restrictions on use and reuse of code.

FreeRTOS supports a wide range of combinations of processor families and compilers, see [3].

While FreeRTOS has a free open source license, there exist both a commercially licensed version, called OpenRTOS, and a commerical version satisfying various international safety standards, called SafeRTOS.

We will dive deeper into several aspects of the RTOS, mostly focused on the kernel functionality. Most of the information in the following sections are taken from the book "Mastering the FreeRTOS Real Time Kernel" [5].

Figure 13: The figure shows the FreeRTOS logo. Found via Wikimedia Commons.

2.5.1 Structure and footprint

The FreeRTOS distribution can be downloaded as a single .zip-file. The distribution consists of port-specific demos and FreeRTOS kernel source files. Which source files are needed depend on the application. Only two files are common to all uses; tasks.c and list.c. Further than this queue.c is nearly always required. The other source files are optional. The structure can be summarized in Figure 14.

Specific ports can be found in the folder *portable*. The ports are specific to both the compiler and the architecture. The folder is sorted after compiler first, then architecture. FreeRTOS is ported to many different combinations.

The footprint of the OS will depend on the application-based configuration and optimization. From the FreeRTOS webpage [1] they describe a possibly minimal configuration, which achieves a RAM usage of around 236 B for the scheduler, and then 64 B for each task. It is noted that the scheduler RAM usage may be further optimized by using smaller data types.

When it comes to flash/ROM usage, with the same configuration as above, the kernel used 5-10 kB. This is not necessarily the minimal usage, as it depends on the compiler, architecture and further optimization of the configuration.

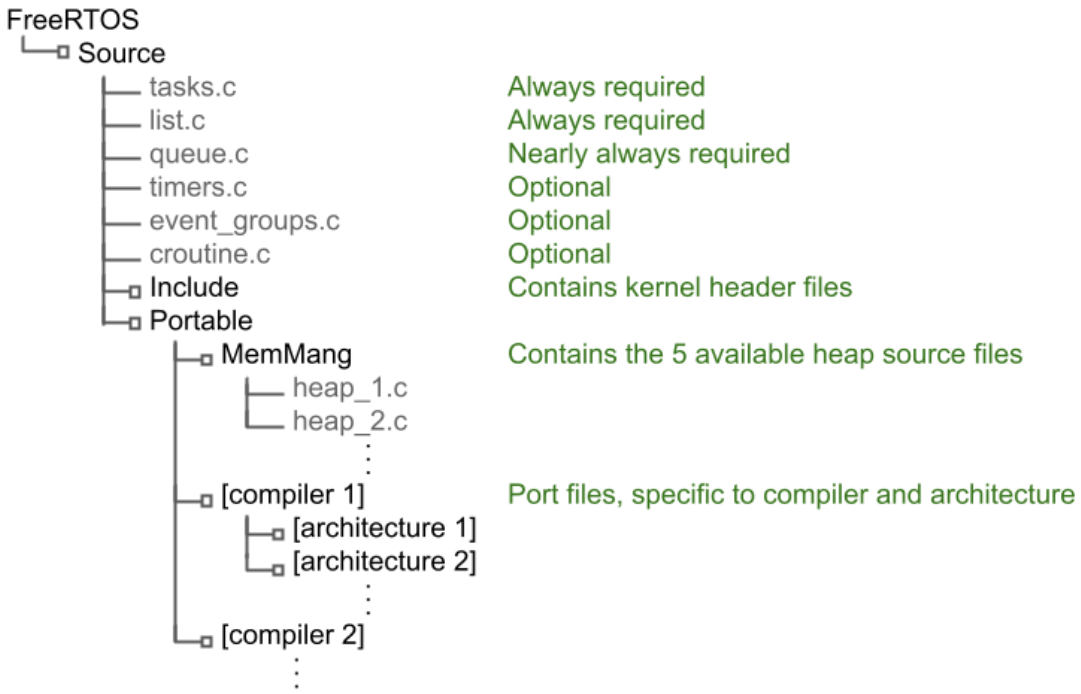


Figure 14: The figure shows the structure of the FreeRTOS kernel.

2.5.2 Installment and configuration

FreeRTOS can be easily installed by downloading a small zip file. This makes it easy to integrate into other SDKs. Such is the case for the NXP FRDM-K22F. Part of the NXP SDK are the FreeRTOS kernel, but also ready-to-run examples. Thus installation and use is simple if it is already included in the SDK.

All RTOS/kernel configurations are done in the FreeRTOSConfig.h file. The file is about 150 lines of code (including newlines). With some previous knowledge (ie. reading documentation) it is possible to understand what each option is doing. All mentions of configurations in the following chapters will thus refer to values in this file. A full example of a FreeRTOS configuration file is provided in Appendix B.

2.5.3 Task management

Tasks
Tasks in FreeRTOS are independently executable C programs running infinitely, or rather that will never return. Typically, this will be implemented as an infinite loop. Instead of returning they will be deleted by the scheduler if they are not needed.

Tasks are managed by the scheduler, which decides how long tasks should run and when.

A task can be in four different states: running, ready, blocked or suspended. When

a task is in the running state, it is being executed by the processor.

The last three states all describe states a task can be in if it is not running. A task is in the blocked state if it has to wait for some event to occur in the program. An event can be either related to time or synchronization. A task is in the suspended state if it is not available to the scheduler, typically being switched in to a slower storage type. Many applications do not need this state. A task in the ready state are neither of these things, and can be put in the running state when the scheduler decides to.

Figure 15 illustrates the task state structure and valid transitions, taken from [3].

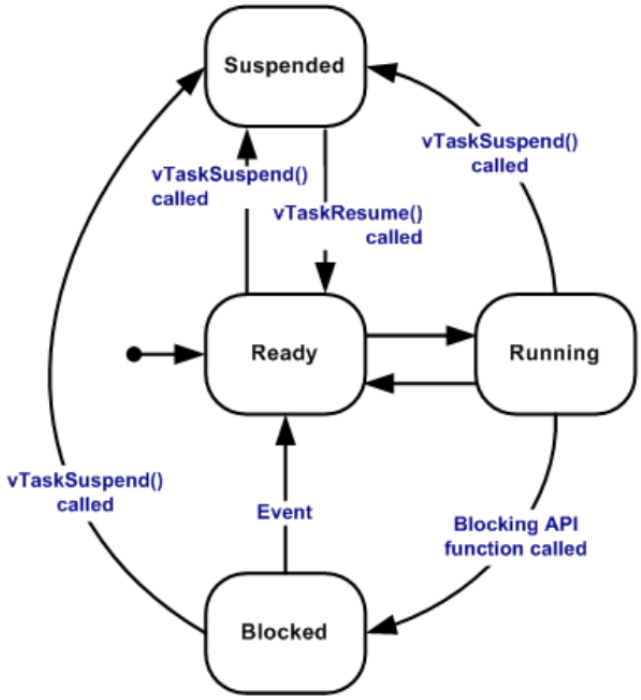


Figure 15: The Figure shows the FreeRTOS task states and valid transitions.

In FreeRTOS there are automatically created system tasks. The most important one is called the idle task [5]. This task runs when no other task can run, ie. it has the lowest possible priority. The idle task cleans up deleted tasks, and can be modified to do additional low-priority jobs in the system. It is also possible to make the idle task yield to another task of the same priority if needed. However, it needs to eventually be able to run if a task has been deleted.

Another system task is the timer daemon task, also called timer service task. This task is optional, and handles commands from the software timers.

A task has a given priority, which can be set when creating the task, but also changed during execution. The lowest priority is 0, and the highest priority can be configured according to application needs. Interrupt Servcie Routines (ISR) will always interrupt tasks, no matter the task priority.

Scheduling

The scheduler is a kernel software routine that decides which of the task will be in the Running state at any given time. Since only one task may run at a time in a single core processor, it is important that the scheduler ensures no starvation of tasks, and that tasks with the highest priority are chosen first. No starvation means that all tasks may eventually run.

The FreeRTOS scheduler can be configured to use several different scheduling algorithms, by changing the parameters configUSE_PREEMPTION and configUSE_TIME_SLICING. The scheduler may be configured to use prioritized pre-emptive scheduling with time slicing, prioritized pre-emptive scheduling without time slicing or co-operative scheduling.

2.5.4 Memory management

FreeRTOS applications can be configured to be either completely statically allocated, or also dynamically allocated [5]. Dynamic memory allocation is when the amount of needed memory is not known at compile-time, and therefore has to be allocated at run-time. This could be a problem for embedded systems, since they often have limited resources, meaning failing to allocate memory is much more likely.

The best option for embedded systems will often be to not use dynamic allocation at all, but if it is necessary, then a suitable heap manager scheme should be used. That way, specific strategies to avoid memory fragmentation can be used. Another aspect is that since dynamic allocation can fail (if not enough memory), it introduces nondeterminism in the application, and a source of failure. This might not be acceptable in some systems.

In FreeRTOS there are five different memory allocation schemes ready-made. Which one to choose will vary based on your application. Table 1 briefly summarizes the options and their characteristics [5].

Name	Deterministic?	Heap size	Allocation Algorithm	Fragmentation
heap_1	Yes	Single Statically declared	First fit	Not possible, since cannot free memory
heap_2	No	Single Statically declared	Best fit	Possible if memory blocks are of different sizes
heap_3	No	Defined by linker configuration	implemented C std library	implemented C std library
heap_4	No	Single statically declared	First fit	Combines adjacent blocks
heap_5	No	Multiple statically declared	First fit	Combines adjacent blocks

Table 1: The table shows available heap alternatives in the FreeRTOS kernel, and their characteristics. The allocation algorithms have different properties, but will not be discussed further.

2.5.5 Interrupt Management

FreeRTOS have separate *interrupt-safe* API functions for use in Interrupt Service Routines (ISR). ISRs are not called from tasks, and therefore some of the standard API logic will not work. By using two versions of each API functions where this may be a problem, each function is simpler and easier to test [5]. The execution context does also not need to be checked. In general, using this implementation is more efficient. Interrupt-safe API functions have "_FROM_ISR" or "FromISR" appended to its name.

If some non-ISR-safe API *needs* to be executed anyway, then it is possible to use *deferred* interrupt handling. This can happen in two ways; either deferring the function to the FreeRTOS Daemon task (timer task), or to a custom application task. The task handling the deferred function needs to have a higher priority than normal application tasks, so that it executes first.

2.5.6 Inter-process communication methods

Queues
FreeRTOS queues are special types of communication objects that provide task-task and task-interrupt communication, as well as ordered data storage. Data in queues are stored by copy, not by reference [5]. A queue must be explicitly created and can be accessed through it's handle by any task or ISR. Thus mutual exclusion must be ensured.

When a task performs a queue read or write, it may optionally set a block time. The block time specifies a maximum time to block waiting for the queue to contain an element or free an element, respectively.

Event Groups
Event groups are special types of communication objects that provide extended functionality with regard to events. Firstly, a state may wait for one or more events occur before it can be unblocked. Secondly, a single event may unblock several waiting tasks. These functionalities are not present in queues or semaphores.

As for queues, a wait time may be set to limit the maximum time waiting in the blocked state.

In an event group, event flags are represented as bits in a variable of type EventBits_t. This data type may have either 8 or 24 usable bits, depending on your configuration. APIs are provided to wait on or set certain bits in this data structure.

Semaphores
FreeRTOS implements binary and counting semaphores. The underlying implementation is using a queue structure.

Task Notifications
FreeRTOS also offers a method of direct communication between tasks; namely task notifications. Task notifications are faster and has smaller RAM footprint than

other communication objects, and can often be used instead of these. However, it is not suitable in some scenarios, like buffering data elements, task-to-interrupt communication or broadcasting to more than one task.

With task notification functionality configured, every task will have a notification state and a notification value. When a notification is recieved, the notification state will be set to pending, and the notification value will be meaningful. When this value is read, the notification state will be cleared (set to not pending). A task can be configured to block while waiting for a notification.

2.5.7 Time Utilities

Time is measured in multiples of tick periods, generated from a periodic tick interrupt. The tick interrupt increments the tick variable, and is run with some configurable frequency based on the underlying processor clock. The maximum tick frequency that can be set is 1 kHz. The reason for this maximum value is that the interrupt handling will take too much of the processors time compared to useful application tasks. See the configuration options in Appendix B.

Time values are given in multiples of tick periods, often called just ticks, by the FreeRTOS API, and is the only time unit it handles. However, it offers macros for converting between milliseconds and ticks, if the tick frequency is 1 kHz or less. Time consistency is handled by FreeRTOS, so the user application won't have to consider counter overflows.

Tickless idle is a power-saving configuration that shuts down the processor for a certain time during idle task execution. The missed number of ticks are then calculated and added to the count after wake-up. The systick uses a 24 bit counter, so the maximum time span possible to sleep for is limited by the frequency of the input clock.

Timers
FreeRTOS provides an optional timer mechanism called software timers. Software timers are independent of hardware timers, and are controlled by the kernel. They do not use processing time unless they execute their callback function. Software timers may execute their callback function periodically, called auto-reload timers, or some time in the future, called one-shot timers. The time between callback function executions is called the timer period.

All software timer callback functions execute within the context of the RTOS deamon task, or timer service task. This tasks' priority and stack size may be configured, but runs as any other task.

2.6 Development Platform: NXP FRDM-K22f

The NXP FRDM-K22F Development Board is a low-cost development board. It uses the microcontroller NXP K22F-120 MHz (datasheet at [12]), which uses the Arm Cortex-M4 core. It has 512 kB flash memory and 128 kB RAM. The board has a programmable OpenSDAv2.1 debug circuit. The board has some simple user components like an RGB LED and push buttons.

Figure 16: The figure shows the NXP FRDM-K22F Development Board.

2.6.1 Low-power modes

The MCU has several power modes with different functionality activated. The following power modes, taken from [12], are relevant for this project:

- *Normal run* is the default mode out of reset. The maximum core clock frequency is 80 MHz. The maximum bus clock frequency is 50 MHz.
- *High Speed Run* is the mode with the maximum chip performance. The maximum core clock frequency is 120 MHz. The maximum bus clock frequency is 60 MHz.
- *Normal Wait* is a low-power mode that sets the core in sleep mode, while allowing the peripherals to function as normal. Wakeup from sleep is fast and easy since any interrupt will wake up the CPU.
- Other low power modes allow further power saving, but wakeup functionality is more limited and they need more time to wakeup from sleep.

2.6.2 Timer modules

The K22 MCU has several timer modules. Some select characteristics from [12] are summarized in Table 2. Notice the PIT timer especially.

Timer Modules in K22F	
Name	**Select Characteristics**
Programmable Delay Block (PDB)	- 16-bit counter - Uses bus clock as source clock
Flexible Timer Modules (FTM)	- 16-bit counter - Counting up or down - Programmable source clock
Periodic Interrupt Timers (PIT)	- 32-bit counter - Counts down - Possible to chain several counters - Uses bus clock as source clock
Low-power Timer (LPTimer)	- 16-bit counter - Selectable source clock
Real-time Clock (RTC)	- 32-bit counter in seconds - 1 kHz or 32 kHz source clock

Table 2: The figure shows some select characteristics of timer modules on the K22 MCU.

2.6.3 NXP FRDM-K22F SDK

NXP provides Software Development Kits (SDK) that simplify interaction with the microcontroller. The SDK may be configured according to what platform it will be used with, and what toolchain is used. For this project, ARM GNU toolchain is used instead of NXPs specialized IDE or any other. More about ARM GNU toolchain in Section 2.7.

The NXP FRDM-K22F SDK will be used, as it contains the drivers, the FreeRTOS kernel and example programs. The SDK can be customized and dowloaded from NXP's SDK builder. To access this, you will need to create a user. Choose Linux as your host OS and GCC Arm Embedded as your toolchain. Include FreeRTOS in your SDK build as a minimum.

2.7 ARM GNU Toolchain

GCC Arm Embedded is the GNU toolchain for ARM embedded products, containing ready-to-use tools like arm-gcc compiler and arm-gdb debugger. The compiler implements the C standard libraries, optimized for embedded applications. To download it, go the Download section of their webpages. Choose x86_64 linux host, AArch32 bare-metal target and version 10.3-2021-10, as this is the latest version which is supported by the NXP SDK. Unpack with

```
$ tar -xvf gcc-arm-none-eabi-10.3-2021.10-x86_64-
linux.tar.bz2
```

and move the folder to /opt (for example). Add to environment variables in /etc/environment:

```
ARMGCC_DIR='/opt/gcc-arm-none-eabi-10.3-2021.10'
```

This is so that the NXP SDK will find the arm-gcc compiler.

2.7.1 GNU Debugger (GDB)

Debugging functionality is invaluable to any form of software development. Setting up debugging for embedded systems is not necessarily trivial, as the target code runs on another platform than the debugger on your host computer. This is solved by using special hardware and software.

GDB is the a full-feature debugger in the GNU ecosystem [19], and is used by many IDEs and other applications as the underlying software enabling debugging. However, GDB can also be used standalone as a command line tool. When referencing GDB in this project, it will refer to the ARM GNU toolchain version, arm-gdb.

GDB on your host computer runs as a client communicating over TCP/IP or UDP with a GDB server [19]. The server can run on a remote machine, but may also run on your host computer [9]. The server then interfaces with a debugger probe on the target board through USB [9]. The hardware probe interfaces with the microcontroller using the JTAG or SWD protocols. This can be done by using an external hardware probe or, if the necessary hardware is already integrated on the board, firmware only.

The NXP FRDM-K22F (as well as other NXP evaluation boards) has the necessary debugger probe and communication circuitry integrated by default. This debug adapter is called OpenSDA, and functions as a bridge between your computer and the target processor, as explained above. Other firmwares can be used on this circuitry to use other providers of GDB servers.

2.8 Docker

2.8.1 Installing Docker

To install Docker, your system must fulfill the requirements listed in the installation part of the documentation pages [11]. The easiest and safest way to fulfill these is to use an updated version of Ubuntu. This project used Ubuntu 22.04. The installation process is documented in the same website.

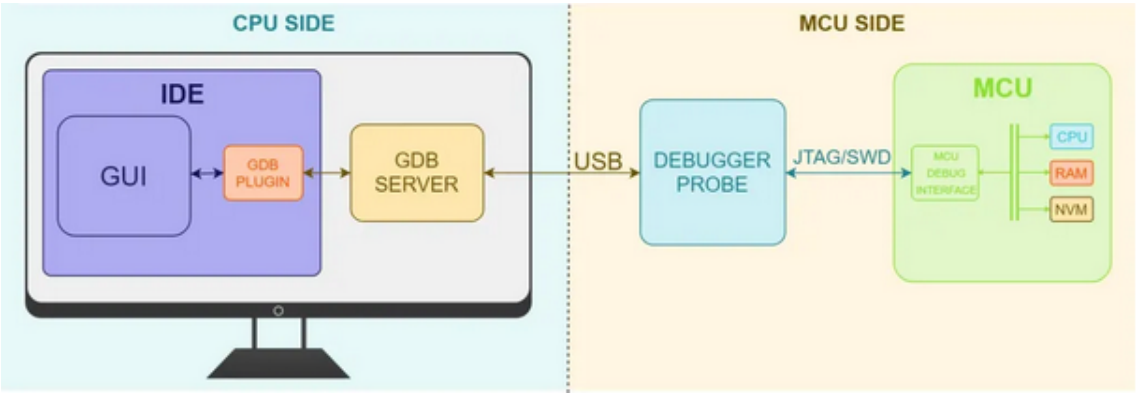


Figure 17: The figure illustrates the debugging setup that allows a host computer to debug code running on another target platform. Taken from [9].

2.8.2 Using Docker

Docker is a tool for containerization; isolated virtualized OS environments [11]. The purpose of using it for this project is to have a documented and easily recreatable build environment.

The main process is to

1. *Write a Dockerfile.* The Dockerfile can use some other image as a base [11]; like a Docker image of Ubuntu 22.04LTS. Then the Dockerfile may specify terminal commands to run, environment variables, or copy folder contents from the host machine to the virtual environment. The Dockerfile language uses simple keywords like "FROM", "RUN", "ENV" and so on.

2. *Build the Docker image.* The Docker image is created from building the Dockerfile. It is a template for the creation of a Docker container, and will appear in the Docker Desktop application. Any number of containers may be created from a certain image. The image may be uploaded to DockerHub (Docker "GitHub") and shared. Build the Docker image by navigating to the folder of the Dockerfile and running

   ```
   $ docker build -t image-name .
   ```

3. *Run a Docker container.* The container is the isolated, (possibly) interactive virtual environment [11]. If a container is deleted, all changes which are done within one will be lost. The container may be uninteractive if excluding the parameter -i, and may run automatic commands specified in the Dockerfile and/or from the command line instead. A container may also mount a directory with -v from the host system, ie. share a directory. In the project, this was used to place the built binary files. Run an interactive container, mounting a volume directory as specified with

   ```
   $ docker run -d -i --name container-name -v
   /host/directory:/container/directory image-name:latest
   ```

4. *Develop, experiment or build files care-free inside the container!* In this project, the container environment was used to build the example programs.

It can be useful to run

```
$ docker system prune
```

once in a while to clean up the cached and outdated ("dangling") images and containers [11].

2.9 CMake

CMake is a powerful, cross-platform build environment that is highly configurable and can manage complex hierarchies. For UNIX-systems, CMake automatically generates Makefiles. Both Lingua Franca and NXPs SDK use CMake.

3 Specifications

The objective of this project is to port an existing software framework (Lingua Franca's single-threaded C runtime) to two new platforms. As explained in Chapter 2.3.5, only the platform-specific part of the runtime needs to be ported. The platform specific part of the runtime consists of seven functions that need to implement specific functionality using the underlying platform. The specifications of these functions, classified by their pre- and postconditions, are given in Table 3. Preconditions should be true before calling the function and postconditions should be true after the function has been called.

The interactions between an LF application and the LF runtime can be shown in the use-case tables 4-8. In the tables, the actor is the LF application, and the system is the runtime. Because the runtime is implemented as general and platform-specific parts, the port will only implement *a portion of* the full runtime functionality. Therefore, the steps which are implemented or modified as part of the port, will be emphasized in the use-case tables. Note that some of the general functionality of the runtime will need to be modified to adapt it for embedded platforms.

Several use-cases are presented. Tables 4 and 5 show use-cases of start and exit of an LF program with the C runtime, for the bare-metal platform. Tables 6 and 7 show use-cases of start and exit of an LF program with the C runtime, for the FreeRTOS platform. The specific order of actions in the start use-case is a result of requirements from both the LF runtime and a general FreeRTOS application. At least one FreeRTOS task must be created before starting the scheduler, and the scheduler should never return. It is therefore followed by an eternal loop. Any task should also never return, but rather be deleted by the scheduler. After the LF main function returns, the C program therefore goes into an eternal loop at the end of the task. This can be seen in the exit use-case in Table 7.

Platform-Specific Function Specifications		
Function Name	**Condition Type**	**Conditions**
lf_critical_section_enter()	Pre	critical sections have been initialized;
	Post	Logical time and event queue can not change unless changed by the calling function OR it failed;
lf_critical_section_exit()	Pre	critical sections have been initialized; lf_critical_section_enter() have been called;
	Post	Logical time and event queue can be changed at any time OR it failed;
lf_notify_of_event()	Pre	An event has been added to the event queue; lf_critical_section_enter() have been called;
	Post	A signaling mechanism has been set OR it failed;
lf_initialize_clock()	Pre	-
	Post	Hardware has been initialized; A platform-specific time counting mechanism has been initialized and started (LF clock);
lf_clock_gettime()	Pre	lf_initialize_clock() has been called;
	Post	Time from LF clock has been read and stored OR it failed; Time value is in nanoseconds;
lf_sleep()	Pre	lf_initialize_clock() has been called; critical sections have been initialized; A sleep duration has been specified;
	Post	Program execution paused for a specified time period OR execution pause was interrupted by new event;
lf_sleep_until()	Pre	lf_initialize_clock() has been called; critical sections have been initialized; A time instance to sleep until has been specified;
	Post	Program execution paused until specified time instance OR execution pause was interrupted by new event;

Table 3: The table shows the functions that need to be ported and their specifications in terms of pre- and postconditions.

Table 8 show a use-case of the LF application scheduling a physical action from an interrupt handler. This use-case is the conceptually the same for both platforms.

Start LF application, Bare-metal	
Description	Start an LF application
Preconditions	Valid LF program; enough heap memory
Postconditions	Application started; startup event triggered
Actor Actions	*System Actions*
1. Run binary	
	2. Initialize hardware
	3. Initialize LF clock
	4. Set LF start time
	5. Display start message
	6. Trigger startup event
7. Execute startup reactions and other reactions at the same tag	
	8. Enter main loop

Table 4: The table shows the use-case between application (actor) and runtime (system) at application start for the bare-metal port. The points marked in bold and red font are the points that is affected by the port.

Exit LF application by stop request, Bare-metal	
Description	Exit LF application by calling lf_request_stop() in a reaction
Preconditions	Executing reaction
Postconditions	Shutdown events triggered; Termination function called
Actor Actions	*System Actions*
1. Call lf_request_stop()	
	2. Set stop tag
	3. Complete current tag
	4. Trigger shutdown event
5. Execute shutdown reactions and other reactions at same tag	
	5. Free allocated memory
	6. Display exit message

Table 5: The table shows the use-case between application (actor) and runtime (system) at application exit for the bare-metal port. The points marked in bold and red font are the points that is affected by the port.

Start LF application, FreeRTOS	
Description	Start an LF application
Preconditions	Valid LF program; enough heap memory
Postconditions	Application started; startup event triggered
Actor Actions	*System Actions*
1. Run binary	
	2. Initialize hardware
	3. Create LF task
	4. Start Scheduler
	5. Initialize LF clock
	6. Set LF start time
	7. Display start message
	8. Trigger Startup event
9. Execute startup reactions	
	10. Enter main loop

Table 6: The table shows the use-case between application (actor) and runtime (system) at application start for FreeRTOS. The points marked in bold and red font are the points that is affected by the port.

Exit LF application by stop request, FreeRTOS	
Description	Exit LF application by calling lf_request_stop() in a reaction
Preconditions	Executing reaction
Postconditions	Shutdown event triggered; Termination function called; FreeRTOS in empty loop
Actor Actions	*System Actions*
1. Call lf_request_stop()	
	2. Set stop tag
	3. Complete current tag
	4. Trigger shutdown event
5.Execute shutdown reactions and other reactions at same tag	
	6. Free allocated memory
	7. Display exit message
	8. Empty loop

Table 7: The table shows the use-case between application (actor) and runtime (system) at application exit for FreeRTOS. The points marked in bold and red font are the points that is affected by the port.

Schedule physical action from interrupt	
Description	Schedule a physical action at any time, typically during sleep
Preconditions	Interrupt handler in preamble; interrupt triggered
Postconditions	Execute reaction triggered by physical action
Actor Actions	*System Actions*
	1. Sleep; system in low power
2. Call to lf_schedule() inside interrupt handler	
	3. Wake up from sleep
	4. Schedule event at physical time as specified
5. Execute reactions triggered by action event	

Table 8: The table shows the use-case between application (actor) and runtime (system) at physical action scheduled in an interrupt handler. The points marked in bold and red font are the points that is affected by the ports.

4 Design

This section will present important design considerations and justify the design choices. The two platforms give different possibilities, and need to be examined separately.

4.1 Bare-metal Runtime

The most important design choices concern the LF clock source and how to handle sleeping. These subjects are discussed in the next sections.

4.1.1 Choice of LF Clock Source

The C runtime for Lingua Franca uses a 64-bit clock, which must be implemented using platform-specific counters or clocks. The K22F have a number of timer modules (Summarized in Table 2 in Chapter 2.6.2), but none of them have 64-bit counters. A way to handle this is to either use two chained 32-bit counters or four chained 16-bit counters. Chaining means that one counter may only count when another has expired. This can be done either in software, keeping track of volatile variables holding the upper counters, or in the timer module itself, if it is supported.

Periodic Interrupt Timers (PIT) supports chaining 32-bit counters. Another advantage to this module is that is uses the bus clock as a source clock. The bus clock will run at either 50 MHz or 60 MHz, meaning the time granularity in the LF clock

would be either 16 or 20 nanoseconds. A disadvantage of using the PIT module is that it countsdownwards to zero, instead of up, which would be more intuitive. Since we want to utilize the maximum value of the counters, they both need to be initialized as the maximum value of an unsigned 32-bit integer.

4.1.2 LF Sleep and Physical Actions

The implementation of LF sleeping on a bare-metal platform can be done in two ways. Either letting the program poll the LF clock value and physical action event status (busy-sleeping), or letting the CPU go into some form of low-power mode, and be awakened by interrupts when the sleep should finish or a physical action has occurred. Since no other code need to execute on this platform, both implementations would be fine. However, the interrupt approach has the benefit of being more power-efficient. Especially if the sleep duration is long. However, it does create some additional overhead with going-to-sleep and waking-up times. These together can not be longer than the desired sleep duration (obviously), so that also requires some additional logic.

A compromise between these two choices is to use the *Normal Wait* power mode. This is a low-power mode which only shuts down the CPU clock, thus provides fast wake-up times, and can be woken up by any interrupt. We can then start a sleep countdown timer before going to sleep, and then be awoken when it, or any other interrupt, triggers. The advantages of using this mode is that

- The LF clock will not stop running, as the PIT timers use the bus clock, and not the CPU clock
- An interrupt handler scheduling a physical action event will interrupt the sleep, thus the function lf_notify_of_event() is not necessary
- Any interrupt will make the CPU exit the power mode, so no additional configuration is needed
- Quick transition times between power modes

A downside to this choice is that the *Normal Wait* mode is not so power saving as other low-power modes. What the best choice of power mode is will ultimately depend on the user application priorities, and will be hard to optimize for a general implementation.

4.2 FreeRTOS Runtime

The most important design choices concern how to fit LF into the FreeRTOS task and scheduler structure, LF clock source, how to handle sleeping, interrupt handling in FreeRTOS and dynamic memory allocation. These subjects are discussed in the next sections.

4.2.1 Task Structure in FreeRTOS

It is not obvious how to structure Lingua Franca in FreeRTOS's task structure, nor how to integrate this structure into the LF auto-generated C code. For single-threaded LF it is reasonable to think that LF could run as a single task in the system. The task would have to be created before starting the scheduler and would have to contain the whole LF program (starting from the lf_reactor_c_main, which is the LF main function). In addition, the relevant hardware should be initialized *before* starting the FreeRTOS scheduler. As for the multi-threaded runtime version, this structure could also be used, as additional tasks may be created by the LF task at a later point.

4.2.2 Choice of LF Clock Source

In FreeRTOS the only available clock API is the tick count, as described in Chapter 2.5.7. The maximum frequency the systick can have is 1 kHz, meaning the minimal time resolution achievable with this is 1 millisecond. The number of ticks since FreeRTOS scheduler start can be read out simply using the API function xTaskGetTickCount(). Thus a LF time source can be made using this current tick count and then converting ticks to equivalent time in nanoseconds, based on the systick frequency.

A disadvantage of using systick is that the frequency is limited to 1 kHz, since the systick interrupts the program for every tick. If the frequency was set higher, then the program would not be able to do much other than interrupt handling. This has implications for both time precision and time overhead in using FreeRTOS. Sections 6.2 and 6.3 investigate these implications.

The K22F board has several external hardware timers available, which would provide higher achievable time resolution. The downside of using one of these is that the runtime becomes less portable. It is worth noting that even if the precision in reported time would be higher, FreeRTOS would still run at 1 kHz maximum. Therefore it is arguably not very useful, and was therefore not done for this project.

4.2.3 LF Sleep and Physical Actions

Since the single-threaded LF program runs on a single FreeRTOS task, it must (or should) share the CPU resource with other tasks in the system. Busy-sleeping would therefore be highly undesirable. Instead, we want the LF task to go into the Blocked task state (See Chapter 2.5.3), so that the scheduler will run other tasks, or at the least, the idle task. Furthermore, the task should go into the blocked state for a certain (sleep) duration, and be possible to wake up when a physical action occurs.

To achieve these properties, the FreeRTOS construct Event Groups may be used (see Chapter 2.5.6). The LF task may then wait on an event group bit flag, and be put in the Blocked state. A max wait duration is specified when waiting for a bit flag. Thus sleep duration can be implemented by using this duration argument,

since it will go out of the blocked state after this duration. If a physical action occurs, then the bit flag may be set in the ISR context, thus setting the LF task out of the Blocked mode. As the function may not finish the sleep, this can be tested using the return values of the event group wait function.

While the LF task is blocked, idle will enter a low-power mode if tickless idle is configured. However, because of limitations in tickless idle mode (mentioned in Section 2.5.7), the maximum sleeping time span will be only about 130 ms at a time (if core clock at 120 MHz).

4.2.4 Dynamic Memory Allocation

In the bare-metal port, we use the compiler's implementation of functions like malloc(), calloc() and free(). FreeRTOS has its own implementation of dynamic memory allocation, as described in 2.5.4. Using the heap_3 option will simply wrap the compiler's implementation, like before. Using heap_4 is generally recommended and provides tools to investigate the heap usage, and will therefore be used in this project. All calls to malloc(), calloc() and free() must be mapped to the FreeRTOS versions. FreeRTOS only implements malloc() and free(). None of these functions can be used in an interrupt context, and there are no interrupt-safe versions.

4.2.5 Interrupt Handling

Several of the single-threaded LF runtime functions may be called from both interrupt and (normal) task contexts. Since FreeRTOS requires that non-interrupt safe APIs are never called from interrupt contexts, this requires additional logic to determine. Specifically, the LF functions that may be called from both contexts must first check its context, and then choose correct API accordingly. By doing this, the FreeRTOS API functions lose the benefits of having two separate implementations, see Chapter 2.5.5.

An alternative could be to also use two separate functions in the LF runtime, however, this would require to change other parts of the runtime code that should be as platform-independent as possible, and possibly provide little gain.

In the normal usage of scheduling physical actions from interrupts in LF, the lf_schedule function is called from within the interrupt handler. This function uses dynamic allocation. Since the dynamic allocation functions are not interrupt safe, this can not be done in FreeRTOS. Besides, it goes against the general advice that interrupts should be as short as possible. The solution is to use *deferred interrupt handling*. If functions that use dynamic allocation, or other non-interrupt safe functions, need to be used within an interrupt context, then they must be deferred to the FreeRTOS Daemon task. This will not affect the runtime itself, but must be taken into account in the user applications.

5 Implementation

This section will give an overview of the project structure and some selected code, as well as tips for debugging. The complete project can be found on github here, and will be provided in a .zip-file together with this report.

5.1 Project Structure

The projected is structured with four main folders: a forked lingua franca repository, a folder for install files, a folder for the bare-metal port and a folder for the FreeRTOS port. The ports are not yet integrated completely with the lingua franca repository. The top-most structure is shown in Figure 18. Two Dockerfiles are provided as well. They will build images that can generate build-containers where the LF file to build is given as an argument. A binary will be generated inside the project folder.

The Lingua Franca repository is largely unmodified, with the exception of a modified LF compiler that will generate a C main function that is specialized for FreeRTOS. To use FreeRTOS as a platform, simply set "target: "FreeRTOS"" in the LF program target declaration.

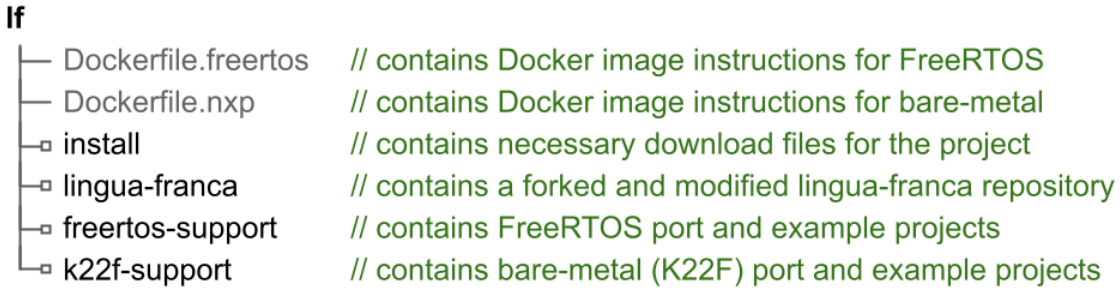


Figure 18: The Figure shows the top-most structure of the project.

The structures of the port folders are shown in Figures 19 and 20. Notably, the new and modified runtime files are places in the "platform" folder. All application source files will be in individual folders under "src", with their respective hardware configuration files and, for FreeRTOS, the FreeRTOS configuration file. These are application specific, but a full example is provided in Appendix B.

In the hardware configuration files, the debug interface, clocks and pins are initialized. Some hardware configuration can also be done in the LF programs themselves, for example in the startup reactions. This would typically be GPIO and interrupt configuration.

The "armgcc" folder inside each application folder contains the build files. Specifically this is a custom build script for LF programs, CMake files (flags, compiler setup, main CMake file) and is also where the compiled binary will be placed.

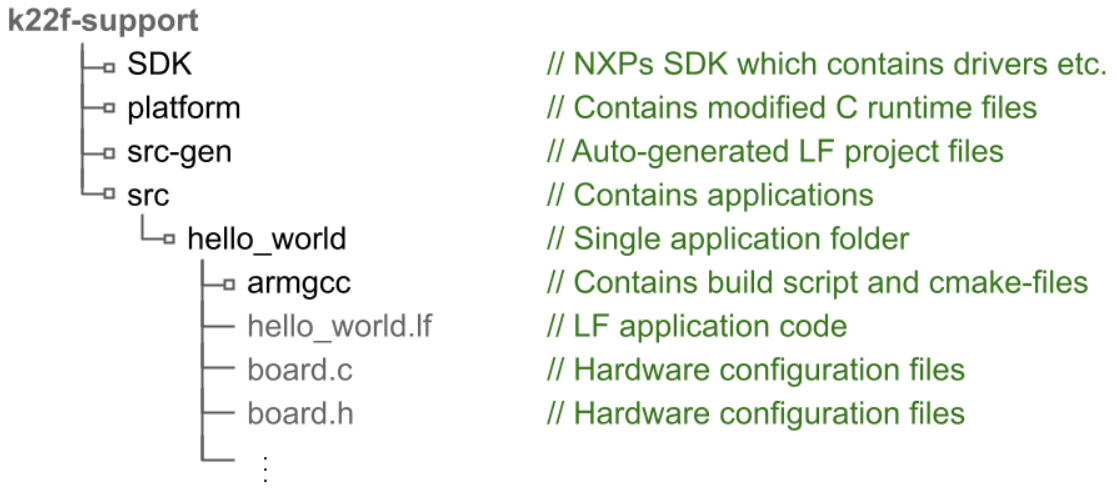


Figure 19: The Figure shows the structure of the bare-metal port folder, k22f-support.

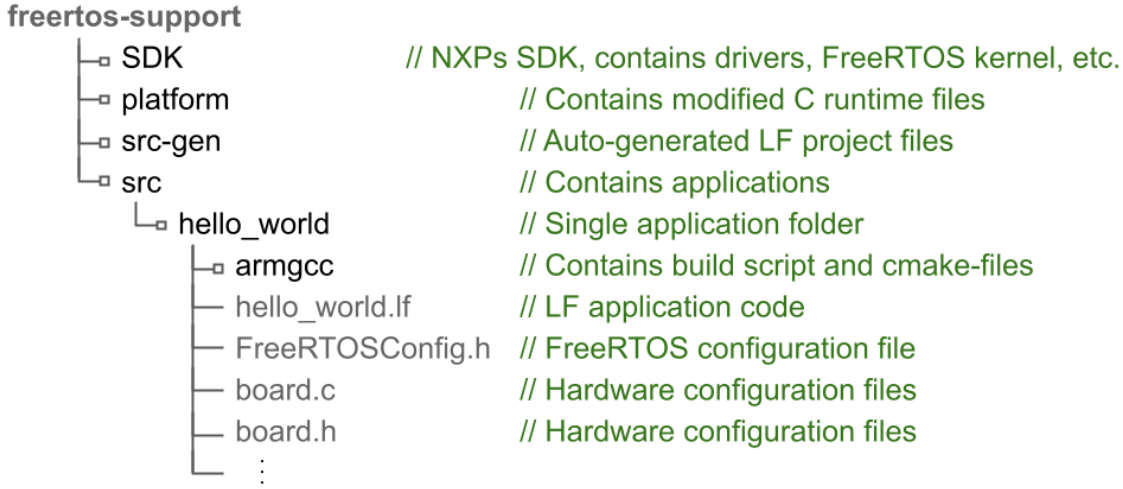


Figure 20: The Figure shows the structure of the FreeRTOS port folder, freertos-support.

5.2 Bare-metal Runtime

An overview of the implementation of the bare-metal runtime port is shown in Table 9.

Bare-metal Port Function Design	
Function Name	**Design**
lf_critical_section_enter()	- Disable interrupts globally
lf_critical_section_exit()	- Enable interrupts globally
lf_notify_of_event()	- Do nothing (see Sec 4.1.2)
lf_initialize_clock()	- Initialize hardware - Initialize PIT counter as LF clock - Initialize PIT counter as sleep timer - Start LF clock
lf_clock_gettime()	- Read PIT counter values - Calculate nanoseconds from count value and counter frequency
lf_sleep()	- Exit critical section - Start sleep timer - Go into Wait mode - Exit Wait mode on interrupt, either from sleep finish or event creation - Enter Run mode - Enter critical section
lf_sleep_until()	- Calculate sleep duration - Call lf_sleep()

Table 9: The Table shows a summary of the implementation of the platform-specific runtime functions for the bare-metal port.

5.2.1 Reading the LF Clock

To implement the LF clock, which should have a 64-bit value, two 32-bit counters are chained together (see Chapter 4.1.1). The chaining itself is done when initializing the clock. When reading the values, some precautions need to be implemented. Since the counters are free-running[5] they can go from zero to the maximum value at inconvenient time instances, for example when reading the clock value. To account for this, the high 32-bit counter is read before and after reading the low 32-bit value. If the low 32-bit counter overflows, then the before and after value will have changed, and the low 32-bit value then needs to be reread. This case is shown graphically in Figure 21.

The current timer count values need to be subtracted from the maximum value since the PIT counters count downwards.

[5]free-running means no external intervention is needed for it to run.

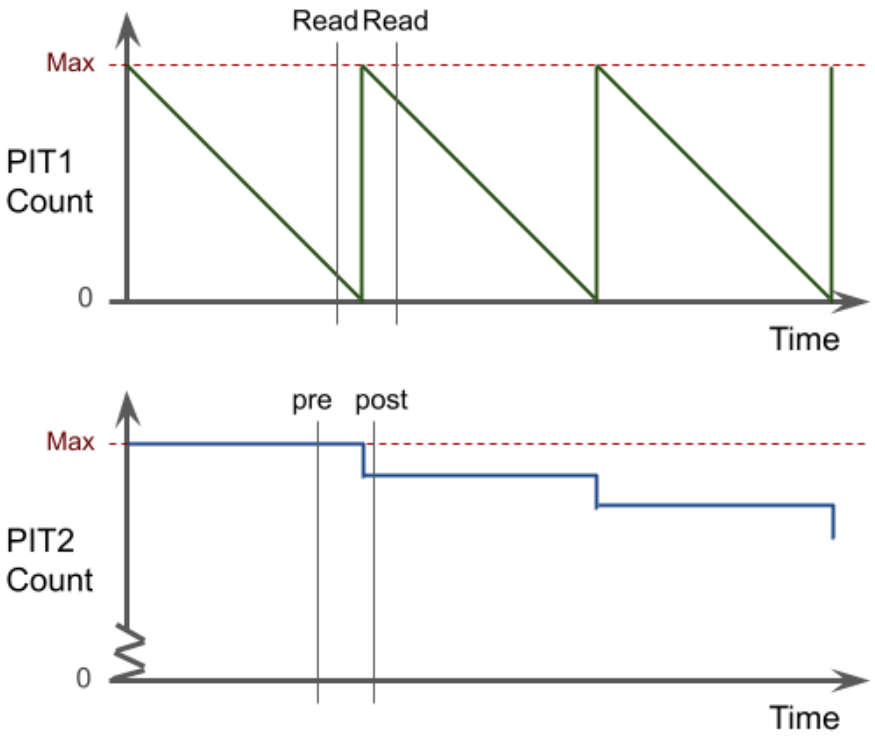


Figure 21: The Figure shows how the PIT counter values are read if the PIT1 counter reaches zero and goes to maximum, to prevent incorrect readings.

5.3 FreeRTOS Runtime

The implementation of the FreeRTOS runtime port is summarized in Table 10.

5.3.1 Task structure in FreeRTOS

In order to let LF's main function run as a FreeRTOS task, a function with the correct task prototype[6] must be created. The LF main function is then called inside this function. The task function must also have an infinite loop so that it never returns. The task function can be seen in Figure 22 as *vLFTask*. Note that this is pseudocode and some details are therefore excluded.

The main function itself must initialize board hardware, create the task, and then start the scheduler. See the rough structure of the auto-generated main function in Figure 22.

[6] A function prototype specifies the return type, argument types and number of arguments.

FreeRTOS Port Function Design	
Function Name	**Design**
lf_critical_section_enter()	- Check if called from interrupt or task context - Disable interrupts with FreeRTOS APIs
lf_critical_section_exit()	- Check if called from interrupt or task context - Enable interrupts with FreeRTOS APIs
lf_notify_of_event()	- Check if called from interrupt or task context - Set Event Group bit
lf_initialize_clock()	- Initialize Event Group for sleep functions - FreeRTOS clock is initialized by scheduler start; this is used as LF clock
lf_clock_gettime()	- Check if called from interrupt or task context - Read tick count from FreeRTOS clock - Calculate nanoseconds from ticks and FreeRTOS clock frequency
lf_sleep()	- Exit critical section - Use Event Group to block LF task until: a. Wait bit is set by event creation (sleep was interrupted) b. Maximum wait duration reached (sleep was finished) - Enter critical section
lf_sleep_until()	- Calculate sleep duration - Call lf_sleep()

Table 10: The Table shows an overview of the implementation of the platform-specific runtime functions for the FreeRTOS port.

```
void vLFTask(arguments) {

    // Start LF C runtime main function
    lf_reactor_c_main(arguments);

    // Infinite loop so that the task never returns
    for( ;; );
}

int main(argc, argv) {

    // Initialize board hardware
    BOARD_init();

    // Create FreeRTOS task running vLFTask function defined above
    xTaskCreate( vLFTask, "LF Task", stack size, arguments,
priority, NULL);

    // Start the scheduler, which should never return
    vTaskStartScheduler();

    // Infinite loop
    for( ;; );
}

```

Figure 22: The code shows how the main function must be modified to run LF in FreeRTOS.

To modify the auto-generated main function, we must modify the LF compiler that generates it. The code in Figure 23 hints at how to do this. The code must be added to where the main function is generated in the file *CMainGenerator.java*.

```
if (platform == FREERTOS) {
    // Necessary code lines as strings separated by '\n'
    return String.join( "\n",
                        "#include \"FreeRTOS.h\"",
                        ...  // more includes
                        "void vLFTask(arguments) {",
                        "lf_reactor_c_main(arguments);",
                        "for( ;; );",
                        "}",
                        "int main(argc, argv) {",
                        ...  // rest of main function above
                        );
}

```

Figure 23: The code shows how the file CMainGenerator.java in the LF compiler needs to be modified.

5.3.2 Scheduling Physical Actions from ISR

As discussed in Chapters 4.2.4 and 4.2.5, lf_schedule can not be called from an ISR without any further treatment. The code snippet in Figure 24 shows an example of a correct handling of scheduling a physical action from an ISR. A wrapper function must be used to match the correct prototype, and then let lf_schedule be called from within it. The wrapper function can then be passed to xTimerPendFunctionCallFromISR() which will defer the execution of the function to the Daemon task.

The purpose of the xHigherPriorityTaskWoken variable is to let the scheduler know if it should switch running task. It will be set by the function xTimerPendFunctionCallFromISR() if this is true.

```
1   /* The function that will execute in the context of the Daemon task
    . It must have this prototype (return type and parameter types). */
2   void vSchedule( void *pvParameter1, uint32_t ulParameter2 )
3   {
4       lf_schedule(pvParameter1, ulParameter2);
5   }
6
7   void INTERRUPT_HANDLER(void) {
8
9       BaseType_t xHigherPriorityTaskWoken;
10
11      // clear interrupt flags
12
13      /* The actual processing is to be deferred to a task. */
14      xHigherPriorityTaskWoken = pdFALSE;
15      xTimerPendFunctionCallFromISR(  vSchedule,
16                                      button_press_action,
17                                      time_instance,
18                                     &xHigherPriorityTaskWoken );
19
20      /* If the Daemon task has higher priority than the LF task,
    then xHigherPriorityTaskWoken will be set to pdTrue by the above
    function. Then a context switch should be requested. */
21      portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
22  }
23
```

Figure 24: The code shows how to handle calls to lf_schedule or other functions that use non-interrupt safe APIs in the user application.

5.3.3 Checking Interrupt Context

Checking the context can be done by calling the function xPortIsInsideInterrupt() which is part of the FreeRTOS API. The way this is handled in the implementation is shown in Figure 25. Some ISR-safe functions need more logic than simply calling the FromISR-version.

```
    if (xPortIsInsideInterrupt()) {
        uxTicks = xTaskGetTickCountFromISR();
    } else {
        uxTicks = xTaskGetTickCount();
    }

```

Figure 25: The code shows how to structure code that needs to be called from both interrupt and task contexts.

5.3.4 Event Group Sleeping and Wakeup

Sleeping is implemented using the Event Group structure. When waiting for ACTION_EVENT_BIT to be set, the task will be blocked. If the idle task then runs, and tickless idle is enabled, the CPU goes into low-power until interrupted or finished waiting. The sleep_duration is the maximum time to sleep.

When a physical action event is created, the function lf_notify_of_event() is called. The function has to set the ACTION_EVENT_BIT with xEventGroupSetBitsFromISR() so that the LF task is unblocked.

The return value from xEventGroupWaitBits() can be used to determine if it was interrupted or waited for the full sleep_duration. See Figure 26 for a pseudocode version of the sleep logic using xEventGroupWaitBits() and checking the return value.

```
    lf_critical_section_exit();

    /* Let current task wait for full sleep_duration, or be interrupted
    by action event */
    uxResultBits = xEventGroupWaitBits(xEventGroupHandle,
   ACTION_EVENT_BIT, sleep_duration);

    lf_critical_section_enter();

    if ((uxResultBits & ACTION_EVENT_BIT) == 0) {
        /* Task waited for full sleep_duration */
    } else {
        /* Task wait was interrupted */
    }

```

Figure 26: The code shows how Event Groups are used for sleep functionality in FreeRTOS.

5.4 Setup

From experience, debugging the applications are tricky since many software frameworks are involved. Using GDB or another debugger tool is absolutely necessary.

This section will explain how to set up GDB in this project, and highlight some interesting bugs.

5.4.1 Debugger Setup

The OpenSDA hardware consists of a circuit featuring it's own microcontroller with USB support. OpenSDA software implements a bootloader, providing an easy interface to load applications or change firmware. However, the out-of-the-box firmware did not manage to connect to a GDB server. Therefore, to use the Jlink GDB server, a new firmware must be flashed to the board. The firmware was downloaded from Jlink OpenSDA firmwares as a binary file, called OpenSDA V2.1 firmware. This file is also provided in the *install* folder (Figure 18).

To flash a new firmware to the board, do the following:

1. Hold in the reset button while plugging the board into the computer. The board will mount on the computer as "MAINTANENCE".
2. Drag and drop the downloaded binary file (Jlink OpenSDA V2.1) into "MAINTANENCE".
3. Wait a little, then unplug the board. Plug in the board again and the board has new firmware for a Jlink GDB server.

The Jlink GDB server is downloaded from here. It will also be provided in the *install* folder. To install, run

```
$ sudo apt install ./JLink_Linux_V782_x86_64.deb
```

To run the server, connecting to the FRDM-K22F debugging circuitry, run the following command

```
$ /opt/SEGGER/JLink/JLinkGDBServerCLExe -localhostonly
-ir -if SWD -speed 1000 -s -device MK22FN512VLH12
```

Now, the server should have successfully established a connection to the board. The next step is to run the GDB client, which is just the normal GDB application. In the case of arm-gdb, and the specific version used in this paper:

```
$ /opt/gcc-arm-none-eabi-10.3-2021.10/bin/arm-none-eabi-gdb
```

Lastly, connect to the server and load the program inside GDB with the following commands:

```
$ target remote localhost:2331
$ monitor reset
$ file /path/to/your/file.elf
$ load /path/to/your/file.elf
```

Putty is used to print the serial interface output. Install by running

```
$ sudo apt install -y putty
```

Launch Putty to listen on /dev/ttyACM0, with the settings, to get debugging output from the board:

```
$ sudo putty /dev/ttyACM0 -serial -sercfg 115200,8,n,1,N
```

5.4.2 Interesting bugs

Heap Size
Dynamic memory allocation is used several places in the platform independent code of the runtime. This might pose a problem for LF in embedded applications, as no operating system layer may handle or mitigate risks of failed allocations or memory fragmentation. Also the available heap memory in embedded systems may be a constraining factor for the amount of reactors and events an LF program may have.

Bugs will arise if the heap is initialized to be too small for the application. Usually the program will just stop if this happens, and by using GDB one can see that the program returns an "Out of memory" error code. If using the FreeRTOS port, then FreeRTOS will also use heap space, meaning the heap size requirement is even larger.

For the bare-metal port, the heap size can be changed by passing a linker flag specifying a size. The flags will result in linker symbols being overwritten, but without needing to modify the linker file itself. The necessary flags are "-Xlinker –defsym=__heap_size__=0xSIZE", where SIZE must be replaced with the wanted heap size in hexadecimal. For FreeRTOS, the heap size is set in the config file (See an example FreeRTOS config file in Appendix B).

Context-specific APIs FreeRTOS
If restrictions on function contexts are violated, for example a non-interrupt-safe function are called from an ISR, then program execution will go into an infinite loop. This is, by design, an optional debug option, as an assert() function finds the violation and sends the program into a loop. This can be verified using GDB. If examining the assert() statement in the file it is called from, the condition it checks is explained in the comments above the statement. Enabling this debug option is highly recommended for development.

The context restrictions can be somewhat bypassed using the strategy shown in Chapter 5.3.2.

6 Testing and Results

This section presents tests and test results. Full functionality testing or benchmarking would be too comprehensive for this project. Some select functionality has been tested, and this is listed in the first subsection. Three different aspects are invest-

igated in the following sections, overhead of using FreeRTOS, the lowest response times and power consumption.

6.1 Functionality Testing

Some basic functionality has been tested, and verified to work correctly, using simple LF programs. The functionalities are:

- Application startup
- Application Exit
- Printing time values
- Scheduling trigger from logical action in reaction
- Scheduling trigger from physical action in ISR
- Specifying timeout of application in preamble
- Specifying FreeRTOS as platform in preamble
- Timers
- Inputs and outputs
- Reactor states
- Reactor instantiating and connecting in main reactor

6.2 FreeRTOS Overhead

An interesting aspect to test is the overhead that using an RTOS introduces to the application. The test should find the basic RTOS overhead by eliminating all other factors as much as possible. To achieve this, an LF program can be run in "fast mode", meaning that the program should execute as fast as possible. Practically, this means that no sleep functions will ever be called. This eliminates all other timing factors of the port implementation, and allows an insight into the overhead of using an RTOS.

The test itself is part of the Savina Actor Benchmark Suite [10] that have been ported to Lingua Franca. These benchmarks are possible to run on embedded platforms as well, as they simply produce printed output.

6.2.1 Test Description: PingPong Benchmark

Reactor Ping sends a ping to reactor Pong which answers Reactor Ping as soon as possible. Then this is repeated 10 000 times in each iteration. The test will then be run for 10 iterations.

The test is a ready-made LF program with a structure as shown in Figure 27.

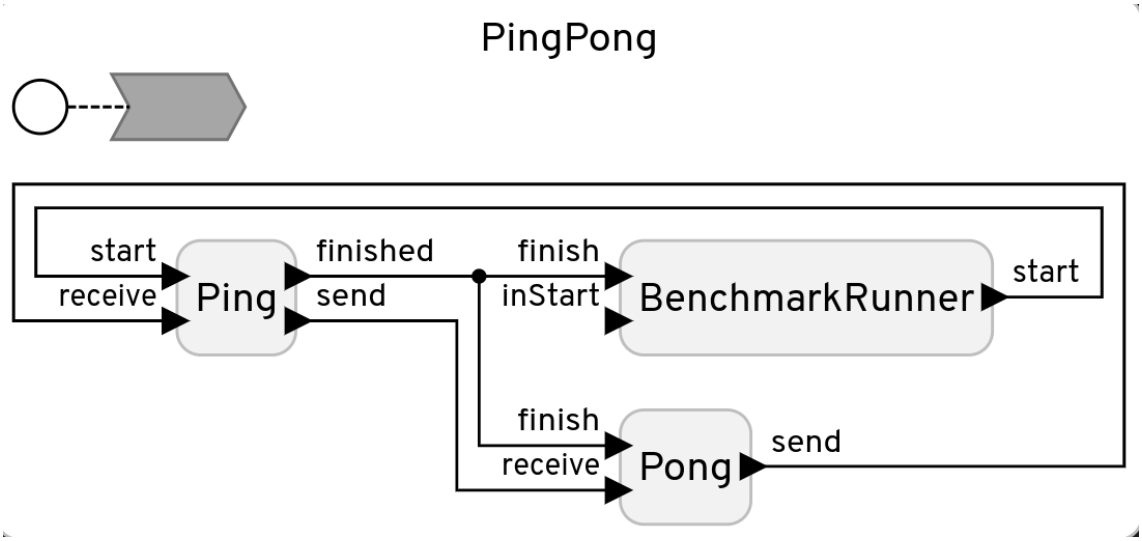


Figure 27: The figure shows the LF diagram of the PingPong benchmark.

The bare-metal runtime is expected to perform significantly better on this benchmark, as it is the only thing going on in the system. For FreeRTOS, it will be interrupted every 1 ms by the systick interrupt, and will have a longer startup procedure.

6.2.2 Experimental Results

The test is done with 10 000 ping-pongs, and the result is shown in Table 11. In addition to comparing the two platforms, we also compare two core clock frequencies, as this will affect the speed substantially.

The first observation is that the FreeRTOS implementation takes roughly 20% more time compared to the bare-metal implementation for each core clock frequency. This tendency is as we would expect, since FreeRTOS is continuously interrupted. This difference is larger at a higher core clock frequency.

The second observation is that the variation in the result for each iteration is extremely low (single microseconds) for the bare-metal implementation. For FreeRTOS the variation is in units of 1 millisecond. This also makes sense since it runs at 1 kHz and therefore has a granularity of 1 millisecond.

To investigate the granularity further, an additional test is run for FreeRTOS. The test result is shown in Table 12. This time, the FreeRTOS frequency will vary, instead of the core clock. Now, one instance has a worst performance that is 10

Iteration Number	Result for 10 000 ping pongs [ms]			
	Bare-metal on K22F (80 MHz)	Bare-metal on K22F (120 MHz)	FreeRTOS on K22F (80 MHz)	FreeRTOS on K22F (120 MHz)
1	818.992	610.578	992.000	762.000
2	818.990	610.576	992.000	762.000
3	818.990	610.576	992.000	761.000
4	818.990	610.575	992.000	762.000
5	818.990	610.576	991.000	762.000
6	818.990	610.576	992.000	762.000
7	818.990	610.576	992.000	762.000
8	818.990	610.577	992.000	762.000
9	818.990	610.576	992.000	762.000
10	818.990	610.576	991.000	761.000

Table 11: The table shows the result of running the single-threaded PingPong benchmark with 10 000 ping-pongs on two platforms and with two different core frequencies.

milliseconds worse than the best performance. This is a big variation compared to the 1 kHz case.

The observed overhead does not significantly decrease with a lower interrupt frequency. This is surprising, as it could imply that the largest part of the FreeRTOS overhead is from another aspect that the systick interrupts. Perhaps it is from the startup procedures for FreeRTOS or the memory allocation scheme performance.

Iteration Number	Result for 10 000 ping pongs [ms]	
	FreeRTOS at 100 Hz (120 MHz)	FreeRTOS at 1000 Hz (120 MHz)
1	760.000	762.000
2	760.000	762.000
3	760.000	761.000
4	760.000	762.000
5	760.000	762.000
6	760.000	762.000
7	760.000	762.000
8	770.000	762.000
9	760.000	762.000
10	760.000	761.000

Table 12: The table shows PingPong test results for different FreeRTOS tick frequencies.

6.3 Time Precision: Shortest Timer Test

Another interesting aspect is how the runtime implementations affect the time precision, or the shortest response time. The test should push the implementations to see at what periods it no longer produces reliable or useful output. To do this, a test utilizing an LF timer to toggle a GPIO could be used, and then see what the shortest period achievable is.

6.3.1 Test Description

For each trigger of the timer, it should simply toggle a GPIO pin on the board. This will generate a square waveform that can be measured by an oscilloscope. The hardware setup is shown in Figures 28 and 29.

The following code is a sketch of the LF test program. The timer periods that will be tested include 1 ms, 500 us, 100 us, 50 us and 1 us.

```
target C {
    threading: false
};

main reactor {
    preamble {=
        //includes
    =}
    timer t(1 sec, 50 usec)        // (start time, timer period)
    reaction(startup) {=
        // Initialize GPIO for output
    =}
    reaction(t) {=
        // Toggle GPIO
    =}
}
```

Both implementations are configured to run with the K22 core clock at 80 MHz. The shortest timer period achievable for the bare-metal implementation will possibly be limited by the LF runtime's own operations or by transitions from normal power mode and to low power mode. The sleep function goes into low-power mode no matter what the sleep duration is. The shortest timer period achievable for the FreeRTOS implementation will be limited by the systick clock. The expected time precision would then be 1 millisecond.

6.3.2 Experimental Results

Figure 30 shows the GPIO voltage output for a LF timer with a period of 1 ms. Both implementations produce a correct output at this period, as expected. Figure 31 shows the output if the period is set to 500 microseconds. Here we can see that the bare-metal implementation produces a correct output; the voltage level is toggled every 500 µs. The FreeRTOS implementation produces a clearly incorrect output.

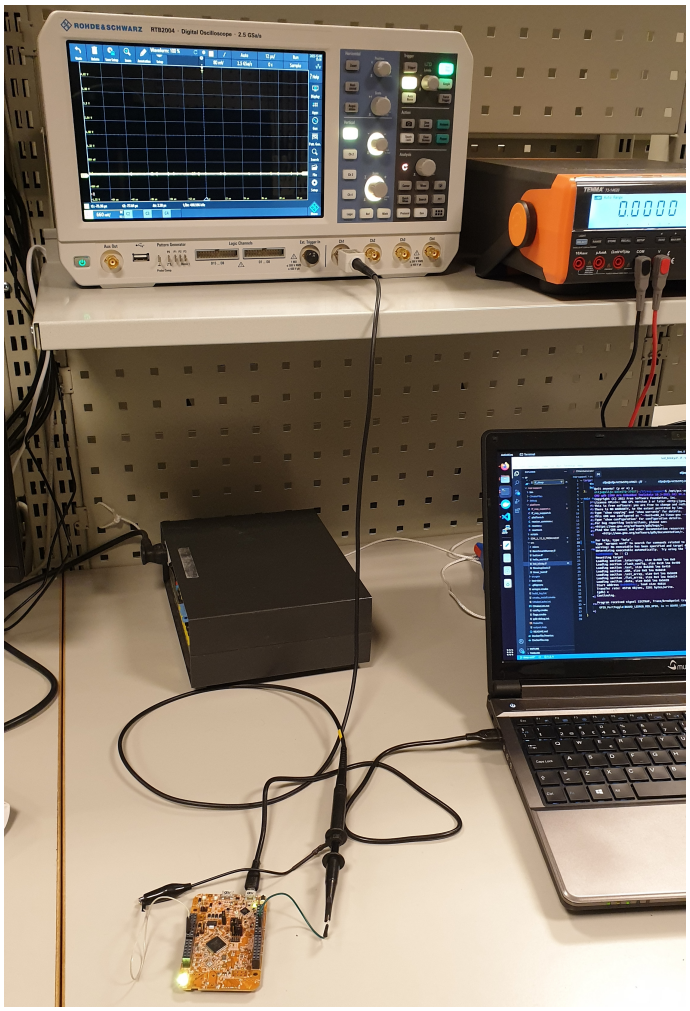


Figure 28: The figure shows the hardware setup for the test.

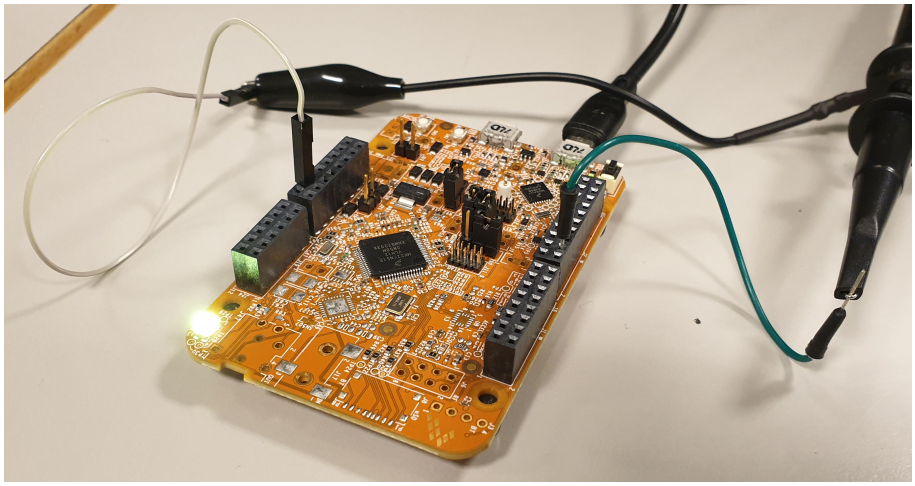


Figure 29: The figure shows a close up of the board test setup.

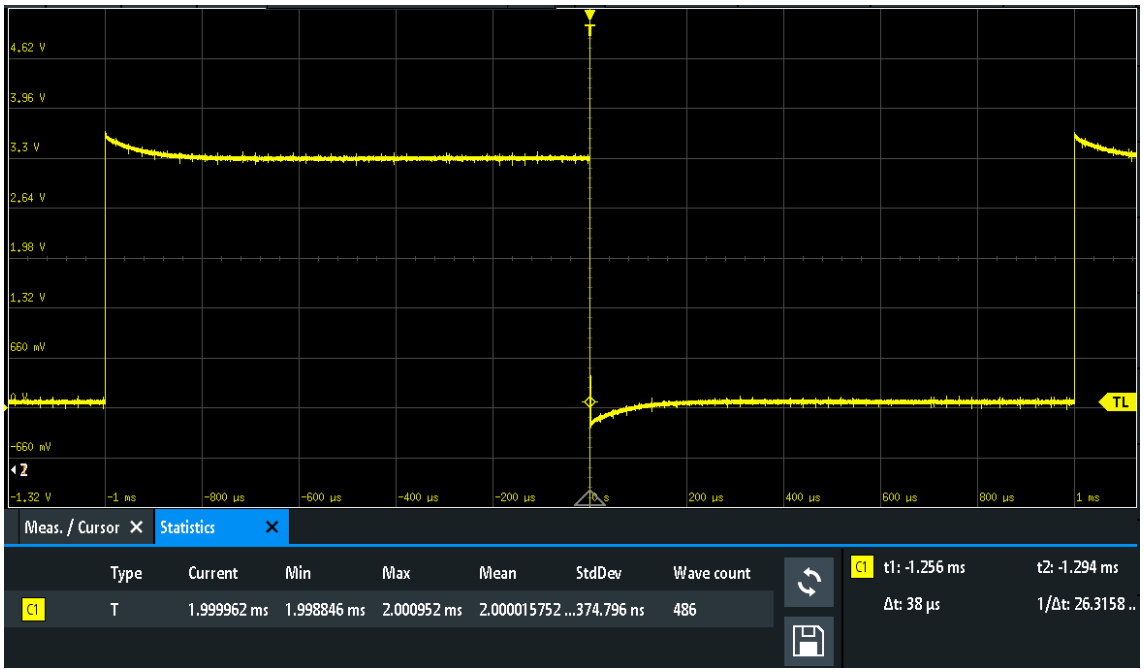
Instead of toggling every 500 µs, the GPIO is toggled after around 920 µs, and then again after just around 80 µs. Interestingly, the total square wave period is correct (two toggles in 1 ms), with a low standard deviation of only 5 ns (bottom part of oscilloscope image).

Figure 32 shows the output of setting the timer to have 100 µs period and 50 µs, respecitvely. Notice these timer values are only tested for the bare-metal implementation, since the FreeRTOS implementation already produced incorrect output at 500 µs. The result of 100 µs period is correct, while the one for 50 µs is incorrect. Somewhere in between here is the lowest possible timer period. By closer inspection, it is between 50 µs and 60 µs, but a figure of this is not included.

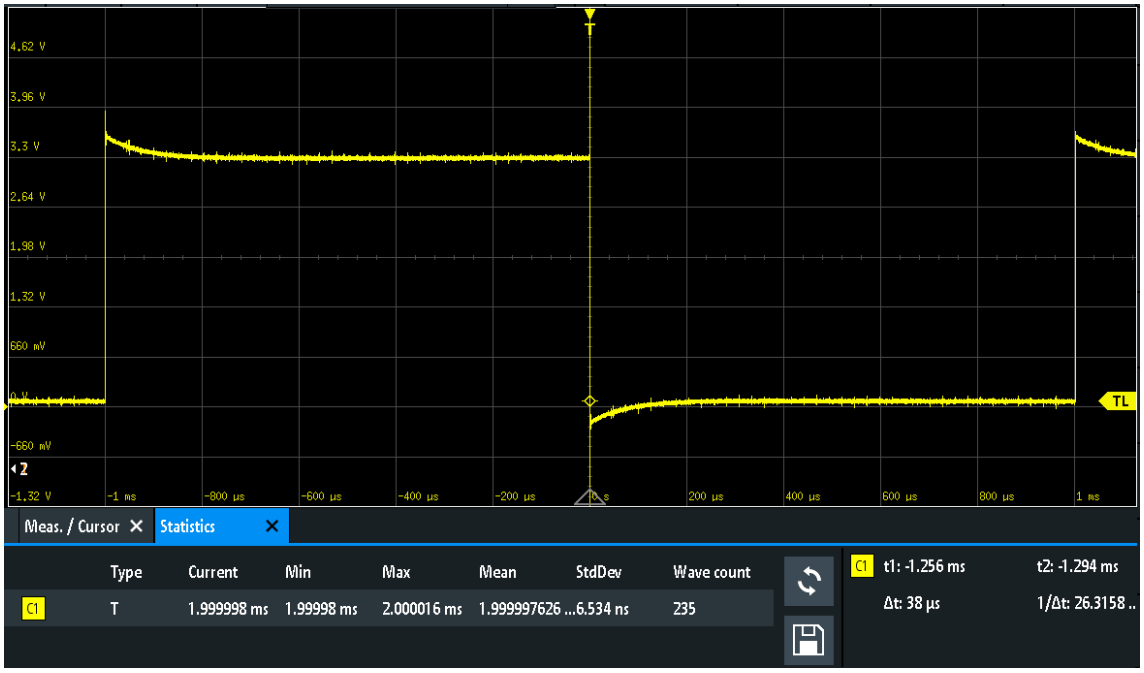
Transitions in power modes that are invovled in the bare-metal implementation, do not use as much as 50 µs, based on power mode transition times in [13]. Therefore, it seems that the limiting factor is the LF program operations themselves. This is consistent with the results for 1 µs in Figure 33, since at this short period, the runtime avoids calling sleep functions[7] entirely. Both implementations then produce incorrect results that toggle the GPIO every 45 µs, for the bare-metal port, and 60 µs, for the FreeRTOS port. This tells us that the LF program cannot run faster than 45 us. Any additional low-power/sleeping functionality will limit this minimum period further. Additionally, we see that the FreeRTOS overhead is present here as well, causing the FreeRTOS minimum time to be longer.

If we compare the bare-metal implementation's toggle periods when it does not call the sleep function (Figure 33(a)) and when it does (Figure 32(b)), we can provide a rough estimate of the sleeping overhead that comes from power mode transitions. Since the minimum toggle period with sleep is around 57 µs, and without is around 45 µs, then the sleeping overhead is around 12 µs.

[7]The limit is set to 10 µs in the runtime version this project has used

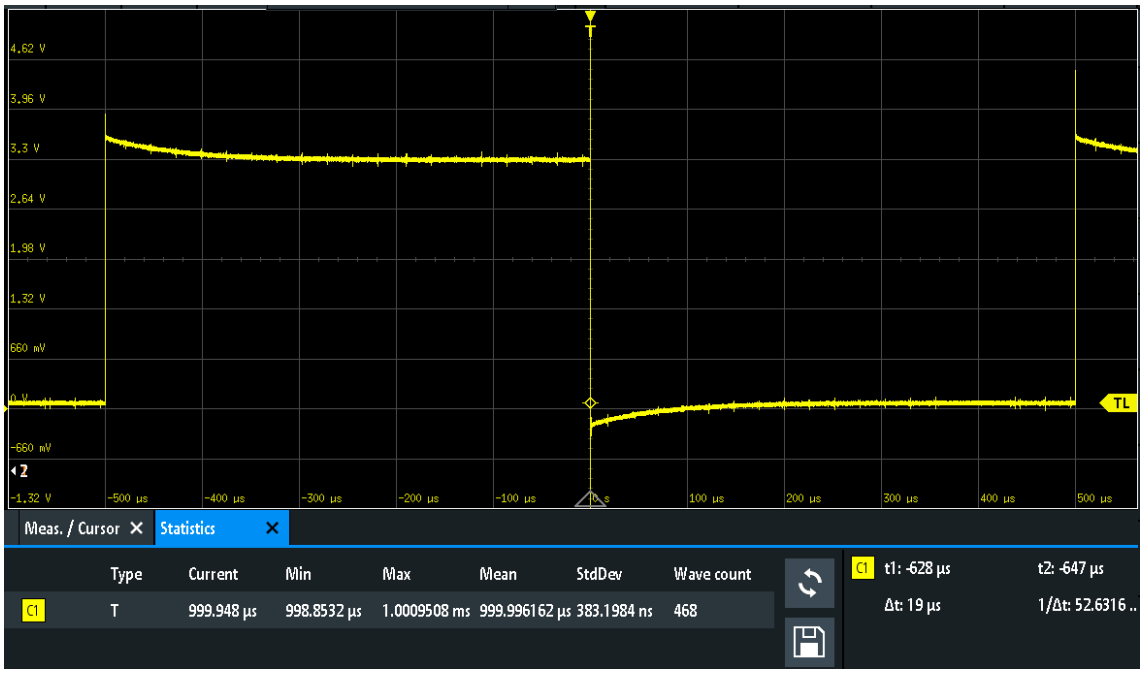


(a) Bare-metal, 1 ms between each timer trigger. Toggles the GPIO pin every 1 ms.

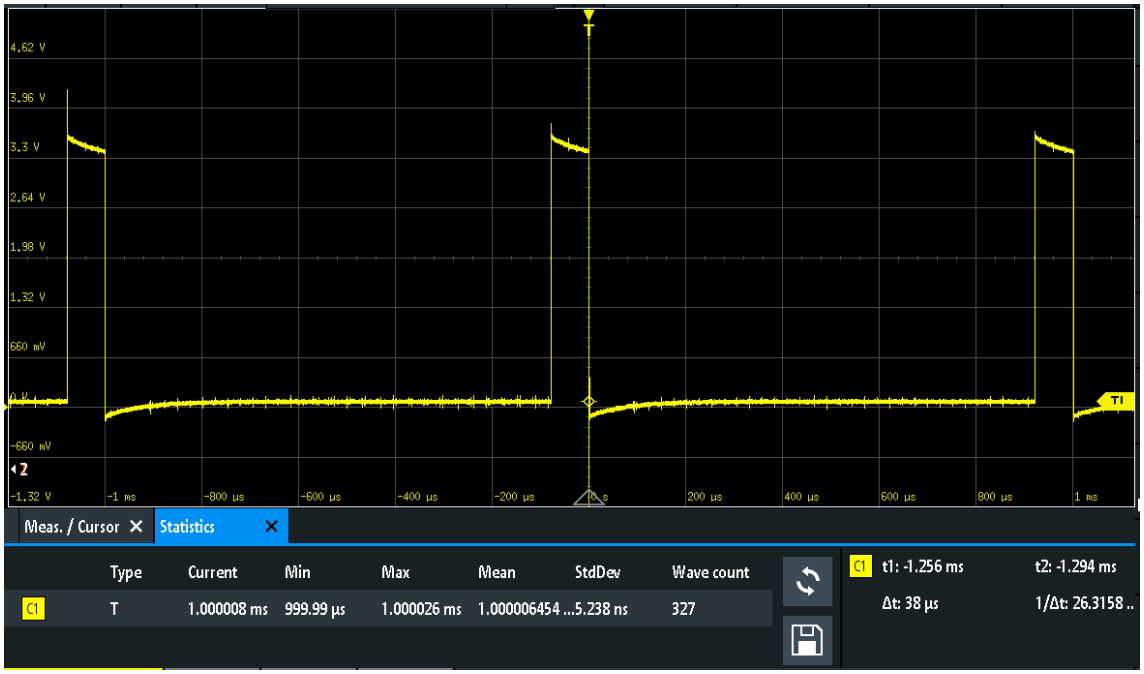


(b) FreeRTOS, 1 ms between each timer trigger. Toggles the GPIO pin every 1 ms.

Figure 30: The figure shows experimental test results for a timer period of 1 millisecond, photographed from the oscilloscope screen. Here we expect both to work fine.

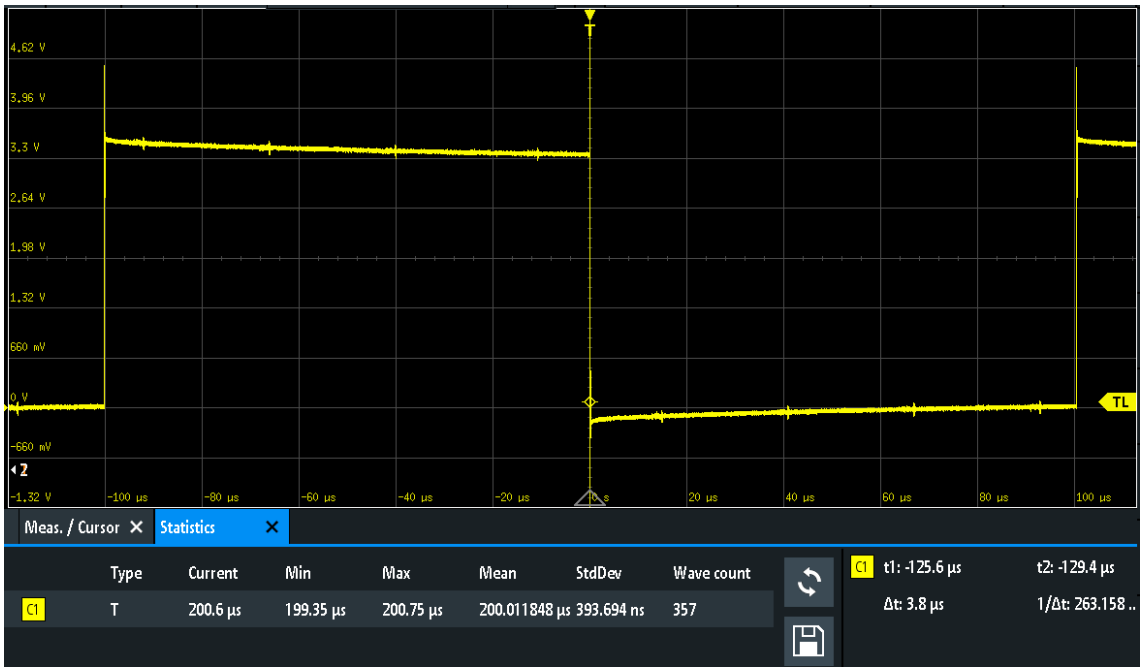


(a) Bare-metal, 500 µs between each timer trigger. Toggles the GPIO pin every 500 µs.

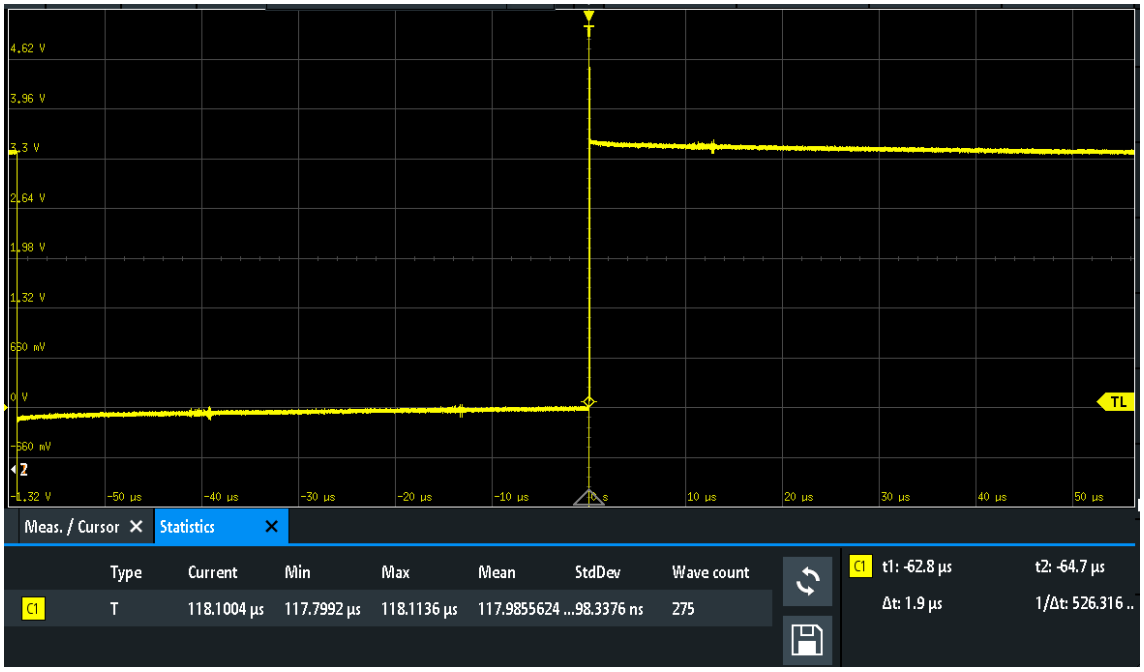


(b) FreeRTOS, 500 µs between each timer trigger. Toggles the GPIO pin around every 920 and 80 µs.

Figure 31: The figure shows experimental test results for a timer period of 500 microseconds, photographed from the oscilloscope screen. Here we expect only bare-metal to work.

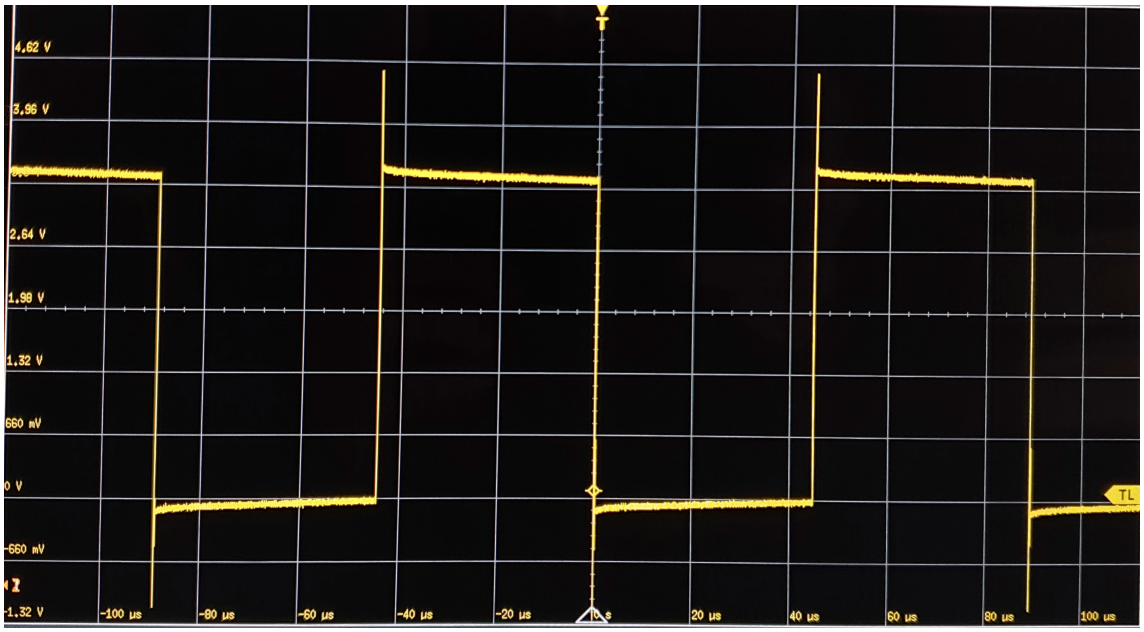


(a) Bare-metal, 100 µs between each timer trigger. Toggles the GPIO pin every 100 µs.

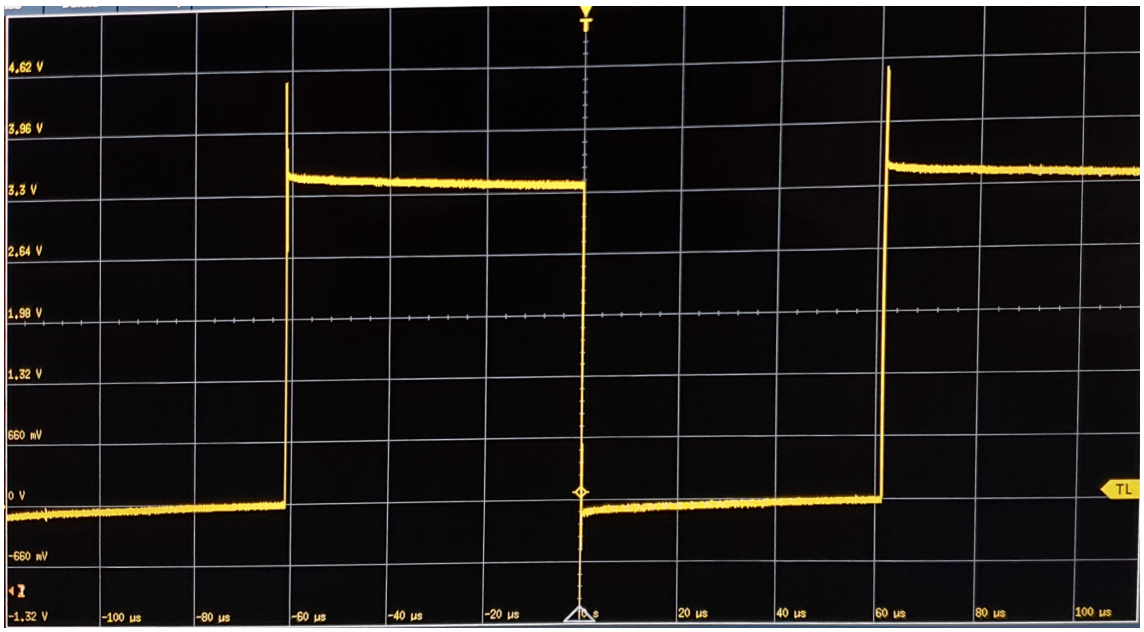


(b) Bare-metal, 50 µs between each timer trigger. Toggles the GPIO pin around every 57 µs.

Figure 32: The figure shows experimental test results for bare-metal with a timer period of 100 microseconds and 50 microseconds, photographed from the oscilloscope screen. At 50 us, the output is incorrect.



(a) Bare-metal, 1 µs between each timer trigger. Toggles the GPIO pin around every 45 µs.



(b) FreeRTOS, 1 µs between each timer trigger. Toggles the GPIO pin around every 60 µs.

Figure 33: The figure shows experimental test results for a timer period of 1 microsecond, photographed from the oscilloscope screen. Sleep function are skipped entirely in both implementations. We see that both produce wildly incorrect outputs.

6.4 Power Consumption

Measuring current to the CPU has two important functions. First, it makes it possible to verify that the LF program sends the CPU into low power mode during sleep. Second, it gives insight into the power consumption characteristics, which will impact how long the program can run on the MCU while being powered by a battery.

6.4.1 Test Description

To measure normal current consumption, the PingPong test from Chapter 6.2 is used. The reason for this is that it never calls sleep functions, and therefore never goes into low-power mode.

To measure low-power current consumption, a simple button push application is used. The application waits for a button push on the board, and sleeps while waiting. So, by never pressing the button, the application is always in low-power. For the bare-metal port, the low power mode "Wait" is used, while for FreeRTOS, "Tickless Idle" is used.

It is possible to measure power consumption of the MCU by removing a jumper [8]. A current probe can then be placed on the two pins and be in series with the circuit. Voltage can also be measured by placing one probe to GND and the other to one of the pins. A schematic of the power supply circuit is included in Appendix A.

6.4.2 Test Results

The result from the bare-metal port is shown in Table 13, for two core clock frequencies. The first observation is that the MCU goes into low-power mode, and it measures for 80 MHz at 0.4 µA. Compared to 1.15 µA at normal mode this is a definite decrease. The second observation is that for the high-speed run, all currents are higher.

The result from the FreeRTOS port is shown in Table 14, for two core clock frequencies. The first observation is that the low-power mode measures for 80 MHz at 0.44 µA. This is also a decrease from normal mode. Again, all currents are higher for he high-speed run. Another observation is that the tickless idle mode in FreeRTOS produces very similar results to the low-power mode to "Wait". The FreeRTOS implementation of tickless idle is decided by the specific FreeRTOS port to NXP FRDM-K22F.

Lastly, all current measurements are unexpectedly low. We would expect numbers in the mA area, not µA. Perhaps there is some error in the measurement technique used. Because of this, not much can be said about the MCU power consumption from this test. However, it was able to verify entering the low-power modes.

Bare-metal Current Measurements [µA], 3.3V		
	Normal Mode	Low Power Mode
Normal run (80MHz)	1.15	0.4
High speed run (120MHz)	1.74	0.8

Table 13: The table shows current measurements to the CPU for different power modes and clock speeds for the bare-metal port.

FreeRTOS Current Measurements [µA], 3.3V		
	Normal Mode	Low Power Mode
Normal run (80MHz)	1.2	0.44
High speed run (120MHz)	1.8	0.9

Table 14: The table shows current measurements to the CPU for different power modes and clock speeds for the FreeRTOS port.

7 Discussion

This section will discuss the project and its results in a larger context.

7.1 Project Outcomes

Embedded systems give new design considerations, requirements and priorities, compared to general-purpose computing. This is also true for embedded Lingua Franca, as this project has explored through implementing a port for two different embedded systems.

Interacting directly with the K22F MCU gives great design flexibility and optimization for the bare-metal runtime. As we saw in section 6.3, the shortest timer possible was limited by the programmatic operations of the LF runtime, not by the LF clock granularity achieved bythe PIT counters. Therefore, if the platform-independent part of the LF runtime is optimized for embedded applications, then a much higher shortest timer could be achieved. Replacing dynamic memory allocation with static allocation could be one such optimization. Having shorter possible response times makes the system more suitable for applications with faster dynamics.

This is, perhaps, the biggest weakness of the FreeRTOS runtime. Since the shortest response time is 1 ms, and no optimizations are possible to improve it (as far as this project has found), then there is a limit to which physical systems it can interact with. One could argue that most systems do not require response times faster than 1 ms, and that this therefore is not a big disadvantage. In addition, it is not the point of an RTOS to run fast, since the main idea is for reliable and consistent executions. This idea should be transferred to Lingua Franca programs as well, if there should be a point of using it with real-time systems.

In section 6.4 we saw that the low power modes of both the bare-metal implementation and the FreeRTOS implementation were similar. However, for the bare-metal

application, the chosen power mode was a compromise for this project. A number of power modes are available for the K22F MCU, which could optimize the power management further. A deep sleep power mode could be used, thus reducing power consumption substantially more than FreeRTOS with tickless idle. This would be of great value for the up-time of battery powered systems, for example in IoT or industry applications.

The power mode in tickless idle can be modified as well. However, the tickless idle has a hard limit on the sleep duration. Therefore the gain would probably be little. A way to remedy this weakness is using an external timer source for the systick. That way, even if the processor goes into low power, the peripheral timer would still be running. If implemented in this way, then there will be no need of recalculating tick count before and after sleep. This is likely a good power optimization strategy for the FreeRTOS LF runtime, as it should be able to sleep for however long is necessary.

In section 6.2 we saw that using FreeRTOS introduces an overhead of around 20%, compared to the bare-metal implementation. As we saw when comparing the systick frequencies, it is not only because of the continuous interrupts generated by the systick. Additional overhead means that the performance decreases a bit, in general. Therefore, FreeRTOS should not necessarily be used if there is no need for its functionality in the user applications.

Using an RTOS opens up for the development of significantly more complex systems than a bare-metal platform. The reason for this is that the task and scheduler structures makes it easier to do many tasks in a system, such as communication. It is more portable, scalable and easy to use than working with the MCU directly. FreeRTOS is already ported to a range of platforms. Communication stacks are already implemented and ready-to-use. The weaknesses in using FreeRTOS is possibly greatly outweighted by these aspects, depending on the user application. While the bare-metal runtime outperforms the FreeRTOS runtime in all tests in this project, it would likely be another story if this project tested with more advanced applications.

The frequent use of dynamic memory allocation in the Lingua Franca runtime is problematic on embedded platforms. If using only static allocation at compile-time, then there would be a hard limit on the number of reactors, events etc. However, it is reasonable to set these limits for embedded platforms, as they reflect the constrain on resources. There is no set limit to the amount of memory that can be allocated dynamically in a LF program, and thus it will fail eventually, if the program is large enough, has memory leakages or creates a fragmented memory. To do replace dynamic allocation in the runtime, macro logic could be set up for every instance of malloc, calloc, realloc and the corresponding free's such that they could be replaced with some alternative.

Even if it was not a realistic goal for this project to implement the multi-threaded runtime for FreeRTOS, a few observations that could be useful are noted here. To run LF inside the FreeRTOS framework, it has to be run inside a task. A task can create other tasks. To then create more LF tasks with lf_thread_create(), and let them run on their own FreeRTOS task, the two task notions need to be unified such

that lf_thread_create() creates a FreeRTOS task. Further, FreeRTOS implements mutexes, semaphores and condition variables (in the form of Event Groups) which can be implemented as the corresponding LF types. The Event Group implementation of sleeping is valid for multi-threaded as well, as Event Groups essentially set flags that can signal several tasks. FreeRTOS does not support multicore systems, so lf_available_cores() should simply return 1. Preemptive scheduling is not yet supported by the LF scheduler. This means that multi-threaded LF may yet have limited usage for FreeRTOS on one core. However, running Federated LF programs on different, networked MCUs running FreeRTOS could be a possibility in the future. This is an example of extended potential that using an operating system provides.

7.2 Areas of Improvement

The project has several areas of improvement. A lot of time went into learning the different software frameworks, how to combine them and debugging. A consequence of this is that the ports are not as thoroughly or systematically tested as they should be before integrating them into the main Lingua Franca project repository. The results show some unexpected behaviors, especially with power management, which should be examined further.

The hardware portability should be examined and improved. FreeRTOS is ported to many boards, using the same APIs. The Cortex M4 port has some peculiarities, meaning the FreeRTOS LF runtime implementation cannot be used unmodified on other processors. This should be the only restriction, since the LF runtime port has exclusively used FreeRTOS APIs, even for the LF clock. The bare-metal runtime implementation should be able to use on other boards with the same K22 MCU.

A challenge to the hardware portability is the build structure of the project. The CMake files is heavily based on NXP's structure and this might be problematic for other boards.

Sleep functionality could be improved. For example setting a minimum sleep limit for the bare-metal port, and possibly using more advanced low-power modes. Another example is examining low-power modes further in FreeRTOS. For example using a peripheral timer for systick, so that the tickless idle maximum sleep time is not limited by having to calculate number of missed ticks.

8 Conclusion

Lingua Franca is possible to run on both bare-metal applications and real-time operating systems. There are some limitations with both these platform that will impact the overall performance of the Lingua Franca programs.

Time precision will be limited by the possible choices of underlying counters. This creates a bound on the fastest dynamics possible to interact with. The bare-metal

implementation has the possibility of interacting with relatively fast dynamics. In addition, there are definitively possibilities of optimizations in this implementation. The FreeRTOS implementation can not run faster than with a precision of 1 millisecond, meaning it potentially is not possible to use for some applications. Even if using a hardware counter for the LF clock, the operating system itself cannot run faster than at 1 kHz. It is not the point of an RTOS to run fast, since the main idea is for reliable and consistent executions. In addition, using an RTOS gives other benefits, especially if dealing with more complex systems, for example running network/communication stacks concurrently with the LF program. It also makes the code easier to port to other hardware platforms.

Using FreeRTOS introduces additional overhead in timing and memory usage. Even with only one application task (the LF task) running, using FreeRTOS introduced a significant decrease in performance, compared to the bare-metal implementation.

The limited memory resources may be a challenge to both implementations, as the Lingua Franca runtime uses (possibly unbounded) dynamic memory allocation. Certainly, the dynamic memory allocation scheme impacts the speed of embedded LF programs, as well as a relatively large chance of allocation failure, given possibilities of memory fragmentation, memory leakage and limited space.

It was a challenge to deal with and combine many different software systems in a good way; the Lingua Franca project itself, NXPs SDK setup and FreeRTOS. The structure of the project could have been better, the hardware portability could be improved, and possibly many optimizations in terms of performance and power usage could be implemented, however this is of course a series of trade-offs which will depend on the priorities of the user application. All in all this project provides a solid baseline to improve further on.

Bibliography

[1] Inc. Amazon Web Services and its affiliates. *FreeRTOS FAQ - Memory Usage, Boot Times Context Switch Times*. URL: https://www.freertos.org/FAQMem.html (visited on 10th Nov. 2022).

[2] Inc. Amazon Web Services and its affiliates. *FreeRTOS History*. URL: http://www.openrtos.net/RTOS.html (visited on 10th Nov. 2022).

[3] Inc. Amazon Web Services and its affiliates. *FreeRTOS Web Page*. URL: https://www.freertos.org/index.html (visited on 10th Nov. 2022).

[4] AspenCore. *2019 embedded markets study*. URL: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf (visited on 10th Nov. 2022).

[5] Richard Barry. *Mastering the freeRTOS Real Time Kernel: A Hands-On Tutorial Guide*. Real Time Engineers Ltd., 2016.

[6] S. Baskiyar and N. Meghanathan. 'A Survey of Contemporary Real-Time Operating Systems'. In: *Informatica* 29.2 (2005), pp. 233–240.

[7] Albert Benveniste and Gérard Berry. 'The Synchronous Approach to Reactive and Real-Time Systems'. In: *Proceedings of the IEEE* 79.9 (1991), pp. 1270–1282.

[8] *Freedom Board for Kinetis K22F Hardware (FRDM-K22F)*. FRDMK22FUG. Rev. 0. NXP Semiconductors. July 2014.

[9] Rocco Marco Guglielmi. *Debugging on STM32 with Chibistudio: THe Ultimate Guide*. URL: https://www.playembedded.org/blog/debugging-stm32-chibistudio/ (visited on 2nd Aug. 2019).

[10] Shams Imam and Vivek Sarkar. 'Savina - An Actor Benchmark Suite'. In: *Proceedings of the 4th International Workshop on Programming based on Actors Agents Decentralized Control*. Oregon, USA, 2014, pp. 67–80.

[11] Docker Inc. *Docker Docs*. URL: https://docs.docker.com/ (visited on 19th Dec. 2022).

[12] *K22F Sub-Family Reference Manual*. K22P121M120SF7RM. Rev. 4. NXP Semiconductors. Aug. 2016.

[13] *Kinetis K22F 512KB Flash*. K22P121M120SF7. Rev. 7.1. NXP Semiconductors. Aug. 2016.

[14] Leslie Lamport. 'Time, Clocks, and the Ordering of Events in a Distributed System'. In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[15] Edward A. Lee. *Cyber Physical Systems: Design Challenges*. Technical Report UCB/EECS-2008-8. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html: Electrical Engineering and Computer Sciences, University of California at Berkeley, Jan. 2008.

[16] Marten Lohstroh. *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*. Technical Report UCB/EECS-2020-235. http://www2.eecs.berkeley.e
2020-235.html: Electrical Engineering and Computer Sciences, University of California at Berkeley, Dec. 2020.

[17] Soroush Bateni Marten Lohstroh Christian Menard and Edward A. Lee. 'Toward a Lingua Franca for Deterministic Concurrent Systems'. In: *ACM Transactions on Embedded Computing Systems* 20.4 (2021), 36:1–36:27.

[18] William Stallings. *Operating Systems: Internals and Design Principles, 9th edition, global edition*. Pearson Education Limited, 2018.

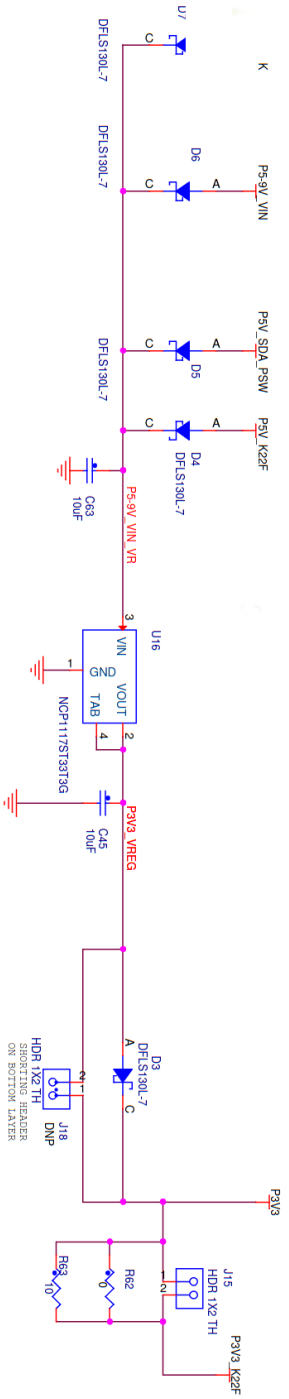
[19] Richard Stallman, Roland Pesch and et al. Stan Shebs. *Debugging with GDB, tenth edition*. Free Software Foundation, 2018.

[20] The Lingua Franca Team. *Lingua Franca Website*. URL: https://www.lf-lang.org/ (visited on 12th Dec. 2022).

Appendix

A NXP FRDM-K22F Power Supply Schematic

The power supply schematic is shown below. The relevant pins to measure power consumption is to the right in the figure. P3V3_K22F is the MCU voltage supply. Header J15 has a jumper that needs to be removed for power consumption measurements.



B Example of FreeRTOS Configuration File

Below is a complete example of a FreeRTOSConfig.h file for K22F, which specifies the application-specific FreeRTOS configuration. It is included in its entirety to be a reference. The value to notice especially are highlighted with in-line comments on the format "//!".

The configuration options to notice especially are:

- *configUSE_TICKLESS_IDLE* must be used to go into a low-power during idle. The exact power mode is decided by the specific FreeRTOS port for the board.
- *configTICK_RATE_HZ* decides the tick rate and is important for this report.
- *configFRTOS_MEMORY_SCHEME* decides the heap memory scheme.
- *configTOTAL_HEAP_SIZE* decides the total heap size; it may need to be adjusted depending on the application heap usage.
- *configUSE_TIMERS* must be set to 1 if using deferred interrupt handling. Daemon task (=timer task) is enabled here.

```
1  /*
2   * FreeRTOS Kernel V10.4.3
3   * Copyright (C) 2020 Amazon.com, Inc. or its affiliates.  All Rights
      Reserved.
4   *
5   * Permission is hereby granted, free of charge, to any person
      obtaining a copy of
6   * this software and associated documentation files (the "Software"),
      to deal in
7   * the Software without restriction, including without limitation the
      rights to
8   * use, copy, modify, merge, publish, distribute, sublicense, and/or
      sell copies of
9   * the Software, and to permit persons to whom the Software is
      furnished to do so,
10  * subject to the following conditions:
11  *
12  * The above copyright notice and this permission notice shall be
      included in all
13  * copies or substantial portions of the Software.
14  *
15  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
      EXPRESS OR
16  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
      MERCHANTABILITY, FITNESS
17  * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
      AUTHORS OR
18  * COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
      LIABILITY, WHETHER
19  * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR
       IN
20  * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
      SOFTWARE.
```

```
 *
 * https://www.FreeRTOS.org
 * https://github.com/FreeRTOS
 *
 */

#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

/*-----------------------------------------------------------
 * Application specific definitions.
 *
 * These definitions should be adjusted for your particular hardware and
 * application requirements.
 *
 * THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE
 * FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.
 *
 * See http://www.freertos.org/a00110.html.
 *----------------------------------------------------------*/

#define configUSE_PREEMPTION                    1
#define configUSE_TICKLESS_IDLE                 1 //!
#define configCPU_CLOCK_HZ                      (SystemCoreClock)
#define configTICK_RATE_HZ                      ((TickType_t)1000) //!
#define configMAX_PRIORITIES                    5
#define configMINIMAL_STACK_SIZE                ((unsigned short)90)
#define configMAX_TASK_NAME_LEN                 20
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_TASK_NOTIFICATIONS            1
#define configUSE_MUTEXES                       1
#define configUSE_RECURSIVE_MUTEXES             1
#define configUSE_COUNTING_SEMAPHORES           1
#define configUSE_ALTERNATIVE_API               0 /* Deprecated! */
#define configQUEUE_REGISTRY_SIZE               8
#define configUSE_QUEUE_SETS                    0
#define configUSE_TIME_SLICING                  0
#define configUSE_NEWLIB_REENTRANT              0
#define configENABLE_BACKWARD_COMPATIBILITY     0
#define configNUM_THREAD_LOCAL_STORAGE_POINTERS 5

/* Used memory allocation (heap_x.c) */
#define configFRTOS_MEMORY_SCHEME               4 //!
/* Tasks.c additions (e.g. Thread Aware Debug capability) */
#define configINCLUDE_FREERTOS_TASK_C_ADDITIONS_H 1

/* Memory allocation related definitions. */
#define configSUPPORT_STATIC_ALLOCATION         0
#define configSUPPORT_DYNAMIC_ALLOCATION        1
#define configTOTAL_HEAP_SIZE                   ((size_t)(10 * 1024)) //!
#define configAPPLICATION_ALLOCATED_HEAP        0

/* Hook function related definitions. */
#define configUSE_IDLE_HOOK                     0
```

```
76 #define configUSE_TICK_HOOK                     0
77 #define configCHECK_FOR_STACK_OVERFLOW          0
78 #define configUSE_MALLOC_FAILED_HOOK            0
79 #define configUSE_DAEMON_TASK_STARTUP_HOOK      0
80
81 /* Run time and task stats gathering related definitions. */
82 #define configGENERATE_RUN_TIME_STATS           0
83 #define configUSE_TRACE_FACILITY                1
84 #define configUSE_STATS_FORMATTING_FUNCTIONS    0
85
86 /* Task aware debugging. */
87 #define configRECORD_STACK_HIGH_ADDRESS         1
88
89 /* Co-routine related definitions. */
90 #define configUSE_CO_ROUTINES                   0
91 #define configMAX_CO_ROUTINE_PRIORITIES         2
92
93 /* Software timer related definitions. */
94 #define configUSE_TIMERS                        1 //!
95 #define configTIMER_TASK_PRIORITY               (configMAX_PRIORITIES -
       1)
96 #define configTIMER_QUEUE_LENGTH                10
97 #define configTIMER_TASK_STACK_DEPTH            (
      configMINIMAL_STACK_SIZE * 2)
98
99 /* Define to trap errors during development. */
100 #define configASSERT(x) if((x) == 0) {taskDISABLE_INTERRUPTS(); for
      (;;);}
101
102 /* Optional functions - most linkers will remove unused functions
      anyway. */
103 #define INCLUDE_vTaskPrioritySet                1
104 #define INCLUDE_uxTaskPriorityGet               1
105 #define INCLUDE_vTaskDelete                     1
106 #define INCLUDE_vTaskSuspend                    1
107 #define INCLUDE_vTaskDelayUntil                 1
108 #define INCLUDE_vTaskDelay                      1
109 #define INCLUDE_xTaskGetSchedulerState          1
110 #define INCLUDE_xTaskGetCurrentTaskHandle       1
111 #define INCLUDE_uxTaskGetStackHighWaterMark     0
112 #define INCLUDE_xTaskGetIdleTaskHandle          0
113 #define INCLUDE_eTaskGetState                   0
114 #define INCLUDE_xTimerPendFunctionCall          1
115 #define INCLUDE_xTaskAbortDelay                 0
116 #define INCLUDE_xTaskGetHandle                  0
117 #define INCLUDE_xTaskResumeFromISR              1
118
119
120
121 #if defined(__ICCARM__)||defined(__CC_ARM)||defined(__GNUC__)
122     /* Clock manager provides in this variable system core clock
      frequency */
123     #include <stdint.h>
124     extern uint32_t SystemCoreClock;
125 #endif
126
127 /* Interrupt nesting behaviour configuration. Cortex-M specific. */
128 #ifdef __NVIC_PRIO_BITS
```

```
129 /* __BVIC_PRIO_BITS will be specified when CMSIS is being used. */
130 #define configPRIO_BITS __NVIC_PRIO_BITS
131 #else
132 #define configPRIO_BITS 4 /* 15 priority levels */
133 #endif
134
135 /* The lowest interrupt priority that can be used in a call to a "set
    priority"
136 function. */
137 #define configLIBRARY_LOWEST_INTERRUPT_PRIORITY ((1U << (
    configPRIO_BITS)) - 1)
138
139 /* The highest interrupt priority that can be used by any interrupt
    service
140 routine that makes calls to interrupt safe FreeRTOS API functions. DO
    NOT CALL
141 INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A
    HIGHER
142 PRIORITY THAN THIS! (higher priorities are lower numeric values. */
143 #define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 2
144
145 /* Interrupt priorities used by the kernel port layer itself. These
    are generic
146 to all Cortex-M ports, and do not rely on any particular library
    functions. */
147 #define configKERNEL_INTERRUPT_PRIORITY (
    configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 - configPRIO_BITS))
148 /* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero
    !!!!
149 See http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. */
150 #define configMAX_SYSCALL_INTERRUPT_PRIORITY (
    configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8 -
    configPRIO_BITS))
151
152 /* Definitions that map the FreeRTOS port interrupt handlers to their
    CMSIS
153 standard names. */
154 #define vPortSVCHandler SVC_Handler
155 #define xPortPendSVHandler PendSV_Handler
156 #define xPortSysTickHandler SysTick_Handler
157
158 #endif /* FREERTOS_CONFIG_H */
```