

Concurrency Without Threads for Multicore Microprocessors

Samuel Berkun

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-112

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-112.html>


May 16, 2024



Copyright © 2024, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

The thesis of Samuel Berkun, titled Concurrency Without Threads for Multicore Microprocessors, is approved:

Chair		Date	May 15, 2024
		Date	May 15, 2024
		Date	

University of California, Berkeley

Abstract

Concurrency Without Threads for Multicore Microprocessors

by

Samuel Berkun

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Edward A. Lee

Cyber-physical systems are often deployed in settings where low cost and power consumption are required, traditionally favoring single-core processors. However, as computer processors fall in cost and rise in capabilities, multi-core processors are becoming increasingly prevalent. This necessitates the use of a software system, such as a real-time operating system (RTOS), to help create and manage concurrent tasks. However, the use of an RTOS comes with several disadvantages, including scheduler overhead and inefficient memory usage. We develop LTA (Library for Timing-aware Actors), and adapt Lingua Franca, a polyglot coordination language, as alternative concurrency systems. We demonstrate that these thread-less concurrency systems can offer more precise timing than an RTOS, while also consuming less memory.

Contents

Contents	i
1 Introduction	1
1.1 Organization	1
1.2 Evaluation Context	2
2 Alternative Concurrency Models	4
2.1 Green Threads	4
2.2 Coroutines	5
2.3 Protothreads	7
2.4 Async/Await	8
2.5 Event-driven Programming	9
2.6 Actor Models	10
3 LTA	11
3.1 Terminology and API	11
3.2 Algorithms	13
4 Lingua Franca	17
4.1 Motivation	17
4.2 Concepts	17
4.3 Platform API	19
4.4 Adaption to Embedded Platforms	21
5 Memory	23
5.1 FreeRTOS Memory Usage	23
5.2 LTA Memory Usage	24
5.3 Lingua Franca Memory Usage	25
5.4 Code Size Comparison	25
6 Core Interference	27
6.1 RP2040 Bus Fabric	27
6.2 Instruction Fetch Conflicts	27

6.3	Data Memory Latency	30
6.4	Conclusions	35
7	Timing	36
7.1	Causes of Timing Variation	36
7.2	Experimental Setup	37
7.3	Overhead	40
7.4	Precision at Low Utilization	41
7.5	Precision at High Utilization	42
8	Case Study: Tunnelling Ball Device	44
8.1	Hardware Design	44
8.2	Physical Constraints	45
8.3	Timing Requirements	47
8.4	Software Design	52
8.5	Software Version 1: LTA	53
8.6	Software Version 2: Lingua Franca	55
8.7	Software Version 3: Lingua Franca with Programmable IO	56
8.8	Potential Improvements	57
9	Conclusion and Future Work	59
	Bibliography	61

Acknowledgments

I would first like to thank Shaokai Lin for bringing me into the lab and for mentoring me through my earliest stages as a researcher. I would also like to thank my advisor Professor Edward A. Lee for giving me the opportunity to take part in the 5th Year Masters program, as well as his patience, advice and mentorship throughout it. I want to thank Erling Jellum and Marten Lohstroh for their mentorship over the last several years. Finally, many thanks to everyone in the lab group; it has been an honor to be part of this group, and everyone's work has been inspiring to witness.

Chapter 1

Introduction

Threads are a nearly ubiquitous construct in modern computer science. In general-purpose computing, threads are a popular mechanism for parallelism and concurrency, opening the door to a wide range of tasks that would not be possible otherwise. In the world of embedded systems, RTOSes make threads available to embedded systems programmers, to adapt this familiar model onto low-level systems.

However, threading-based systems suffer from a variety of downsides. To begin with, threading often comes with significant memory overhead, which can be a problem for highly concurrent systems, and systems that have tight memory constraints. Thread schedulers have non-trivial overhead, and context switching may incur more overhead than alternate approaches. When there are more software threads than hardware threads, the execution time of threads may become highly variable, depending on how the scheduler decides to schedule threads. Finally, as noted by Edward Lee in his paper *The Problem with Threads*, threaded computation is highly nondeterministic and opens the door to data races, deadlocks, and other common concurrent pitfalls [7].

Several of these problems can become showstoppers in the context of hard real-time systems. These systems depend on worst case execution time estimates to meet real-time deadlines, and any added variance in timing could be prohibitive. At the same time, these systems could benefit from concurrency, and the additional computing power made available by parallelizing across multiple cores. This presents a need for alternative system that offers concurrency, but avoids the downsides of a threading-based system.

In this thesis, we present two such alternatives. The first is an actor library developed to meet the needs of hard real-time systems. The second, *Lingua Franca*, is a system for deterministic concurrency that we adapt to the RP2040. Each has unique advantages and drawbacks, and represent different points in the design space.

1.1 Organization

The topics covered in this thesis are as follows:

- This chapter covers preliminaries and background.
- Chapter 2 will cover several concurrency models which can be used as alternatives to threading. This is mainly background, but may be interesting for anyone researching software architecture or programming language design.
- Chapter 3 describes LTA (Library for Timing-aware Actors), an actor library we developed for the RP2040.
- Chapter 4 describes Lingua Franca, a polyglot coordination language developed at Berkeley. It also described contributions we made to adapt it to the RP2040.
- Chapters 5, 6, and 7 evaluate the performance of FreeRTOS, LTA, and Lingua Franca on the RP2040. Specifically, Chapter 5 evaluates memory usage, Chapter 6 evaluates timing predictability of code execution, and Chapter 7 evaluates timing predictability with regards to scheduling overhead.
- Chapter 8 describes the tunnelling ball device, a hardware demonstration of the capabilities of LTA and Lingua Franca. The device requires a control loop executing every 62.5 microseconds with microsecond-level precision, demonstrating that LTA and Lingua Franca can meet tight timing requirements and are appropriate for hard real-time systems.

1.2 Evaluation Context

RP2040

The RP2040 is a dual-core ARM Cortex-M0+ microprocessor by Raspberry Pi. It was released in 2021 with development boards available for just \$4, making this a low cost and powerful option for embedded systems. One of the unique features of the RP2040 is its programmable IO (PIO), which is a collection of hardware state machines that can be programmed to read inputs or trigger outputs at cycle accurate times. This can be used to handle a wide range of tasks, including bit-banging communication protocols, driving VGA displays, and emulating a logic analyzer. This feature makes the RP2040 applicable in extremely precise timing applications, where software-driven actuation might not otherwise be possible.

Given its low cost and applicability to hard real-time systems, the RP2040 was chosen as the primary microprocessor for the evaluations in this thesis. Although most of the results should be applicable to other microprocessors, using a single microprocessor allows us focus our evaluation and investigate low-level micro-architectural details that may affect timing (particularly in Chapter 6).

FreeRTOS

A major goal of this thesis is to compare non-threading approaches to RTOS-based approaches, so we choose an efficient and widely-used RTOS to represent the class. FreeRTOS fits this bill perfectly. It is a free, open-source RTOS widely used in embedded systems, and has good performance according to prior research. When compared to alternatives like Zephyr, FreeRTOS is relatively minimal, and focuses on the scheduler and related synchronization primitives.

Goals and Key Results

This thesis focuses on hard real-time systems, so the evaluation metrics focus on constraints that hard real-time systems typically face. Chapter 5 evaluates memory usage, concluding that for highly-concurrent programs with minimal state, FreeRTOS uses nearly 90 times more memory than an equivalent LTA program, and 6 times more memory than an equivalent Lingua Franca program. Chapter 6 evaluates how the microarchitecture of the RP2040 affects program timing, concluding that FreeRTOS may significantly increase the standard deviation of execution times due to using heap-allocated stacks. Chapter 7 evaluates scheduling overhead, concluding that LTA is able to offer higher timing precision and lower scheduling overhead than FreeRTOS.

Chapter 2

Alternative Concurrency Models

The main motivation for the work in this thesis is to investigate alternatives to threading for embedded systems. Although we develop two alternatives (Chapters 3 and 4), there is a diverse space of potential alternative concurrency systems that can be explored. This chapter gives background on several concurrency models, both in general-purpose computing and in the space of embedded systems.

2.1 Green Threads

In a general purpose OS, user threads are scheduled by the kernel, and are typically backed by an associated kernel thread. These two stacks and associated kernel structures have significant overhead, making them unsuitable for tasks like handling millions of concurrent connections.

Green threads are one answer to this problem. Rather than being backed by kernel structures, green threads are threads scheduled by the user process. This gives them lower overhead, and enables creating many more of them than ordinary user threads. The major downside of this approach is that since green threading runtimes map many green threads onto a single operating system thread, if a green thread executes a blocking operation, it blocks many threads at once. This problem can be mitigated by using asynchronous IO, but this adds complexity. One example of a successful green threading runtime is the runtime used by the Go programming language, which calls its green threads “goroutines”.

In systems without virtual memory (including many microprocessors, like the RP2040), there is no separation between kernel space and user space, so there is no distinction between threading and green-threading. The goals of an RTOS and a green threading runtime are very similar: the RTOS need to schedule many RTOS threads onto a few hardware threads, and a green threading runtime needs to schedule many green threads onto a few operating system threads. Some techniques used in green threading systems may be useful for RTOSes, and vice versa. For example, segmented stacks, a feature found in Go, may also be applied to embedded systems [9].

2.2 Coroutines

In a normal program, it is assumed that execution will run sequentially, and consume the resources of the current thread until it is done running. For example, in Listing 2.1, there is an implicit assumption that the processor is “busy” while `reverse_file` is being run. However, in reality, the processor needs to wait for long periods of time while asynchronous operations run. For example, the thread is typically blocked while the file is being read, during which time the OS may decide to run another thread.

```
1 void reverse_file(char* filename) {
2     char contents[100];
3     read_file(contents, 100, filename);
4     reverse(contents, 100);
5     write_file(contents, 100, filename);
6 }
```

Listing 2.1: Example of a subroutine

```
1 void reverse_file(char* filename) {
2     char contents[100];
3     read_file(contents, 100, filename);
4     yield();
5     reverse(contents, 100);
6     write_file(contents, 100, filename);
7     yield();
8 }
```

Listing 2.2: Example of a coroutine

In general, a subroutine is called, works for a bit, then (optionally) returns a value. A coroutine can be thought of as a generalization of a subroutine. Rather than simply executing until it is finished, a coroutine may stop (yield) at any point during its execution. The caller then needs to resume the coroutine at a later point in time, and the coroutine’s execution continues again until the next yield point. For example, in Listing 2.2, the `read_file` and `write_file` functions do not complete a read or write; they simply start an operation, and the coroutine yields until the operation is finished (presumably the IO functions also do some bookkeeping to make sure `reverse_file`) will be resumed at the correct time). Listing 2.2 does not correspond to any real coroutine system; it uses a hypothetical syntax created for pedagogical purposes.

Under the hood, many coroutine systems transform coroutines into finite state machines. For example, Listing 2.3 is a hypothetical state machine that exhibits the same behavior as the coroutine in Listing 2.2.

```
1  char* filename;
2  char contents[100];
3  int state;
4  void reverse_file() {
5      switch (state) {
6          case STATE_1:
7              read_file(contents, 100, filename);
8              state = STATE_2;
9              return;
10         case STATE_2:
11             reverse(contents, 100);
12             write_file(contents, 100, filename);
13             state = STATE_3;
14             return;
15         case STATE_3:
16             state = FINISHED;
17             return;
18     }
19 }
```

Listing 2.3: Coroutine turned into a state machine

There are many different flavors of coroutine, and a myriad of different syntaxes to support them. One split is *symmetric* coroutines vs *asymmetric* coroutines. The example in Listing 2.2 is an example of an asymmetric coroutine, where the coroutine can only yield back to the caller. In contrast, in a symmetric coroutine, the coroutine must specify a yield destination. Another split is *stackless* vs *stackful* coroutines. Stackless coroutines can not call other coroutines, so coroutines may only exist at the top level. In contrast, stackful coroutines may call other coroutines (thus creating a stack of coroutines to resume at any yield point). Implementation-wise, a stackful coroutine’s state needs a structure similar to a thread stack, which often takes more memory than the state of a stackless coroutine.

Coroutines are closely related to generators. Generators are essentially a simple form of asymmetric coroutine, and often have a very similar implementation. In fact, in October 2023, the Rust programming language renamed its **Generator** trait to **Coroutine**, as it “was effectively a coroutine” [13]. Typically, the distinction between generators and coroutines is that generators only produce values, while coroutines both produce and consume values when yielding.

2.3 Protothreads

The Protothreads library implements asymmetric stackless coroutines in C. It is written by Adam Dunkels and aimed at memory constrained systems, including embedded systems [3]. Although C doesn't natively support syntax for coroutines, the library uses a set of macros to turn coroutine code into a sequence of switch statements. For example, Listing 2.4 shows an example of a coroutine written using the protothreads library (adapted from an example on the protothreads website).

```
1     PT_THREAD(example(struct pt *pt)) {
2         PT_BEGIN(pt);
3         while(1) {
4             if(initiate_io()) {
5                 PT_WAIT_UNTIL(pt, io_completed());
6                 read_data();
7             }
8         }
9         PT_END(pt);
10    }
```

Listing 2.4: A protothreads example

```
1     char example(struct pt *pt) {
2         switch ((pt)->lc) {
3             case 0:
4                 while (1) {
5                     if (initiate_io()) {
6                         (pt)->lc = 13;
7                     case 13:
8                         if (!io_completed()) {
9                             return PT_WAITING;
10                        }
11                        read_data();
12                    }
13                }
14            }
15            (pt)->lc = 0;
16            return PT_ENDED;
17        }
```

Listing 2.5: Protothreads example after the macros are expanded

After the macros are expanded, the example in Listing 2.4 turns into Listing 2.5 (some parts omitted for clarity). Note that the fundamental construct that makes the library work is a switch statement; the library relies on the fact that switch statements can be interleaved with other control structures. For example, in Listing 2.5, the switch statement is interleaved with the `while` and `if`. This may seem reminiscent of Duff’s device, which is not just coincidence: protothreads are based off of a coroutine implementation by Simon Tatham, who was inspired by Duff’s device [12].

The protothreads library is extremely memory efficient; it only uses a single integer to represent the state of each coroutine. However, this comes with an important downside: coroutines may not use any stack variables across yield points. This means that not only are the coroutines stackless, they are also “data-less” as well. Stack variables can be emulated using global variables, but this makes the coroutines non-reentrant. Tom Duff (who created Duff’s device) thought this limitation was prohibitive, commenting “I never thought it was an adequate general-purpose coroutine implementation because it’s not easy to have multiple simultaneous activations of a coroutine and it’s not possible using this method to have coroutines give up control anywhere but in their top-level routine” [1].

2.4 Async/Await

Async/Await is a paradigm commonly implemented using asymmetric stackless coroutines. If approached from a coroutine perspective, then async functions are coroutines with yield points specified by the `await` keyword. If approached from a programmer’s perspective, then async functions are similar to normal functions, except they have special magic that lets them do IO without blocking.

```
1  async function reverse_file(filename) {
2      let contents = await read_file(contents, filename);
3      reverse(contents);
4      await write_file(contents, filename);
5  }
```

Listing 2.6: Example of an asynchronous function in Javascript

Many modern languages include syntax for the `async/await`, including F#, C#, Python, Javascript, and Rust. Async/await is popular for the implementation of web servers, allowing them to handle millions of concurrent connections with only a few threads.

The Rust programming language is unique in that it targets system level software, and has strong support for `async/await`. In particular, Embassy is a Rust library aimed at embedded applications, that supports tasks written using `async/await`. Embassy contains hardware abstraction layer (HAL) implementations for several popular microcontrollers, including the RP2040 [4].

2.5 Event-driven Programming

In an event-driven architecture, rather than having execution units (threads / coroutines / etc) continuously execute tasks, the system responds to incoming events as they happen. For example, consider a hypothetical program that lights up an LED every time a button is pressed. One could dedicate a thread to the button and the light, as in Listing 2.7. However, the system could also be handled in an event-driven manner, as in Listing 2.8. This has the advantage of not needing a thread stack, and being able to wait for the button press without polling.

```
1  void button_task() {
2      while (1) {
3          // poll the button
4          while (!button_is_pressed()) {
5              sleep_ms(10);
6          }
7          turn_on_led();
8          sleep_ms(1000);
9          turn_off_led();
10     }
11 }
```

Listing 2.7: Example of a button system using threads

```
1  void button_interrupt_handler() {
2      turn_on_led();
3      add_timer_ms(1000);
4  }
5
6  void timer_interrupt_handler() {
7      turn_off_led();
8  }
```

Listing 2.8: Example of a button system using events

Event driven systems work best when the events are independent and share no state (as in Listing 2.8). However, they get trickier to design when events need to be associated with state. For example, suppose we wanted to scale the above example to multiple buttons and associated LEDs. With threading, this would simply be a matter of initializing a matching number of threads, and initializing each thread with the appropriate GPIO pin numbers. With the event driven system, there needs to be some way of telling the

`timer_interrupt_handler` which LED to turn off. In this particular example, it can be solved with a global queue of LEDs to turn off, but more complex scenarios may require more complex solutions.

2.6 Actor Models

An actor model is built around isolated units call **actors**. Actors run concurrently, and interact only by sending messages to each other. Each individual actor may be viewed as an event-driven system, where events are messages from other actors. This naturally solves some of the state-tracking issues of event driven systems, as each actor can manage its own state.

Because actors interact only by sending messages, actor models avoid several common problems present in other models of concurrent computation. For example, threading models must synchronize memory using mutexes to prevent data races. Improper use of mutexes may lead to further problems, such as deadlock, livelock, and starvation. In contrast, actors typically don't share state, and coordinate via messages instead. This sidesteps most of these common problems.

Actors are a major part of the Erlang programming language, which is famous for its use in extremely reliable systems. Joe Armstrong (co-creator of Erlang), in his 2003 PhD dissertation, described the main focus of Erlang as large, distributed, fault-tolerant systems [2]. Actors lend themselves well to this domain: independent actors are a good fit for a distributed system, and fault-tolerance can be achieved by restarting failed actors.

Actor models may be particularly viable for embedded systems. In the paper *Actor-oriented design of embedded hardware and software systems* [6], the authors note that actors lend themselves well to components with well-defined interfaces, and several models of computation (synchronous/reactive, dataflow, discrete-event, etc) can be described using actors.

Chapter 3

LTA

In Chapter 2, we noted that actor models have unique advantages that can be used for embedded systems. Actor frameworks exist in many languages, including C. However, most frameworks have a focus on soft-realtime systems, and give very little regard to the timing of messages. For example, Actix (a prominent actor framework for Rust) focuses primarily on web servers.

In order to test the viability of actor models for embedded systems, an actor framework is needed that can meet the requirements of hard-realtime systems. Specifically, the framework should be able to:

- Run functions at precise times
- Efficiently handle high-frequency events
- Have low overhead (both in computation and memory)

These requirements may not be necessary for all hard real-time systems, but are aimed at satisfying the systems with the most demanding timing requirements. In particular, the benchmarks in Chapter 7 and the system in Chapter 8 require all of these.

We designed LTA (Library for Timing-aware Actors) to be a lightweight library that meets these requirements, and to provide an efficient actor library to compare against FreeRTOS. It is designed for the RP2040, but can be ported to any platform that can implement semaphores and critical sections.

3.1 Terminology and API

High-level concepts

Most actor models are built around *actors* and *messages*. LTA deviates slightly by requiring that each message be accompanied by a time at which to send the message. This makes

the messages closely resemble the events in an event-driven system, so they are called *events* rather than messages.

Each actor needs one or more functions to “receive” events. These functions are called *actions*, and have a specific signature. Each actor may only have one action running at a time, which ensures that actions may freely modify an actor’s state without the need for locks. Actions should not block execution by sleeping or waiting on synchronization variables: sleeping or waiting can be accomplished by scheduling events instead.

All events are stored in a global event queue. An actor “sends” an event to another actor (or to itself) by adding an event to the global event queue.

Unlike many actor frameworks, actors cannot be created dynamically. All actors should be known at the start of the program, to prevent the need to dynamically allocate memory.

Terminology

- Actor: An entity with a unique ID and (optionally) some associated data.
- Action: A function that is executed when an event is invoked.
- Invoke: To start the execution of an event.
- Release time: The earliest time an event may be invoked. Ideally events should be invoked close to their release times.
- Start time: The time that the event is actually invoked.
- Busy/Free: An actor is busy if it is currently executing or about to invoke an event. Otherwise, the actor is free.
- Blocked: An event is blocked if its corresponding actor is busy.

API

Actions must have the following signature:

```
1 typedef void (*action)(uint64_t release_time_us, size_t
   actor_id, void* arg);
```

Notably, when an action is being run, it will know the release time of the event, which actor the event is for, and an optional argument that may carry data. The release time is useful if the action needs to check for deadlines, or schedule something periodically.

There are only four functions in LTA. They are:

```
1 int event_queue_init(event_queue_t* queue, size_t max_size,
   size_t num_actors);
2
```

```

3 void event_queue_deinit(event_queue_t* queue);
4
5 void work(event_queue_t* queue);
6
7 int schedule_event(event_queue_t* queue, uint64_t
   release_time_us, size_t actor_id, action fn, void* arg);

```

The first function, `event_queue_init`, initializes the global event queue. This allocates all the memory needed for LTA to work; none of the other functions allocate memory. The `max_size` argument corresponds to the maximum number of events expected to be in the queue at any given time. The `num_actors` argument corresponds to the total number of actors in the system.

The second function, `event_queue_deinit`, is rarely used but may be useful for a program that needs to gracefully shut down. It is possible to use multiple event queues (i.e. one on each core), but this is atypical.

The third function, `work`, is essentially an infinite loop that waits for events, then executes them. Since the rp2040 has two cores, a typical pattern is to do some setup, then run `work` on both cores.

The fourth function, `schedule_event`, adds an event to the global event queue. The parameters are:

- `queue`: The global event queue.
- `release_time_us`: The release time of the action, in microseconds since the rp2040's boot time.
- `actor_id`: Which actor this event is for. This should be in the range $[0, \text{num_actors} - 1]$ inclusive.
- `fn`: The action to invoke.
- `arg`: Optional argument to the action.

3.2 Algorithms

The event queue keeps two large arrays. The first is a large boolean array, representing whether each actor is busy. The second is a large queue of events. The event array is a linear, circular array, which allows $O(1)$ operations in the best case (when events are scheduled in order). The events in the event array are sorted by release time, although they don't necessarily run in order of release time. Figure 3.1 shows what an event queue with 5 actors and 8 events might look like at one point in time.

It might seem odd to use a linear array, which has $O(N)$ operations in the worst case, rather than a heap, which has sublinear operations. The reasons are twofold: First, most

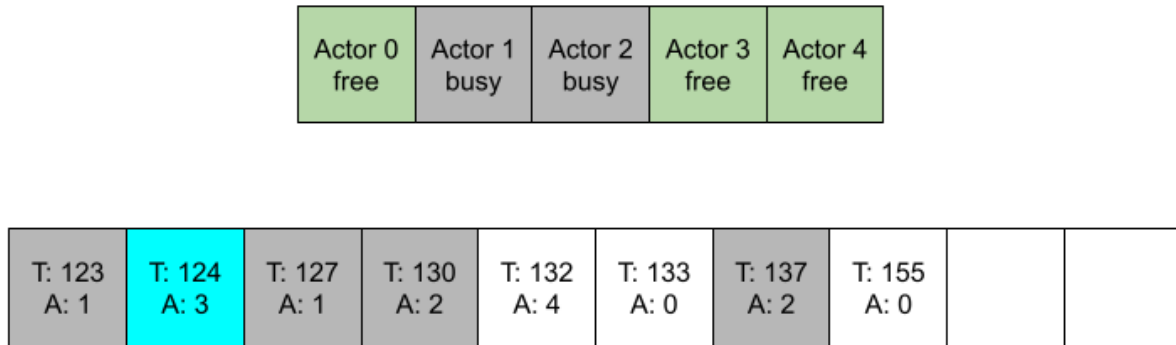


Figure 3.1: Example state of an event queue at $t = 130$. Actors 1 and 2 are busy, so all events corresponding to them are blocked. The second event is ready, since its release time has passed, and it is non-blocked. The other non-blocked events are not ready yet.

systems tend to have a small number of events outstanding at any given time. For example, actors that perform polling loops will simply send an event to themselves once for every poll, so the number of events is always equal to the number of active polling loops (a few dozen, at most. Not very many systems have more polling loops than GPIO pins). Secondly, a linear array can more gracefully handle blocked events. For example, if the release time of an event passes but its actor is busy, the event is blocked until the actor has finished processing its previous event. In a linear array, the blocked event may simply be left at the head of the array and skipped over. In a heap, a separate structure would be needed to store blocked events.

Scheduling an Event

Scheduling an event is relatively straightforward. All the program needs to do is enter a critical section, insert the event at the end of the queue, bubble it down through the queue if needed (to keep the queue sorted by release time), exit the critical section, and notify any waiting threads that the queue has changed. Figure 3.2 shows the result if an event for actor 3 at time 134 is inserted into the queue shown in figure 3.1.

Work

A worker thread's goal is simple: always execute the earliest event that isn't blocked. The only thing that complicates this goal is some minor bookkeeping to make sure that an actor always only has one running event. A quick outline of the algorithm is as follows:

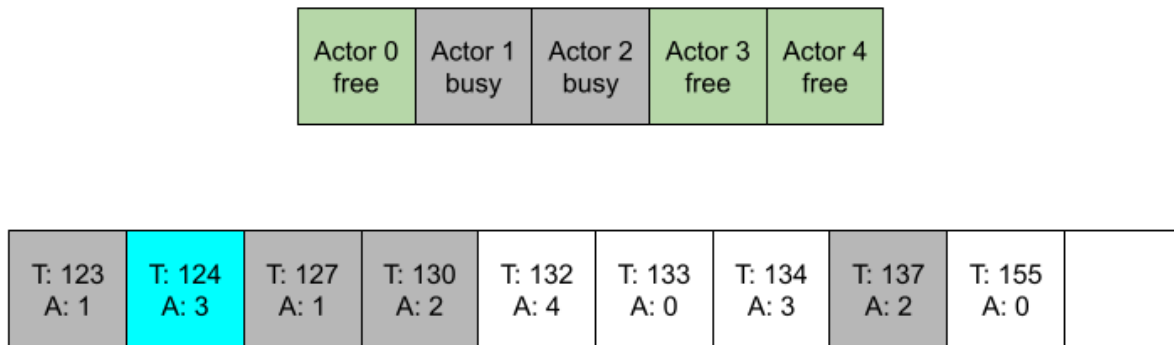


Figure 3.2: Example state of an event queue at $t = 130$ after a new event ($T=134, A:3$) is inserted.

1. First, the worker needs to “claim” the earliest event. It checks the queue for the earliest unblocked event. If it finds one, it removes it from the queue, and marks the actor as busy. Otherwise, it waits for a notification from another thread scheduling an event.
2. The worker waits until the release time of the event. Earlier events may appear during this time, either from other threads, or from interrupts. If so, the worker will be notified, in which case it checks the queue for the earliest unblocked event. If it finds one earlier than the one it has claimed, it puts its claimed event back on the queue, marks the actor as free, claims the new earliest event, and marks the new event’s actor as busy. It can then go back to waiting.
3. Once the release time of the event occurs, the worker invokes the event (executes the action), then marks the actor as free. The worker then goes back to step 1 (claiming an event).

Note that event claiming is an optimization; an earlier version of the algorithm had workers only remove events from the queue once they became ready to execute. However, this introduces a small delay between the release time of the event and event invocation, especially if multiple events release at the same time. Event claiming was introduced as a way of reducing this issue and making start times more precisely align with release times.

A valid question to ask is whether the worker needs to notify the other threads when marking an actor as free. After all, marking an actor as free unblocks events, which potentially could result in another worker grabbing an earlier event. However, the notifying worker itself is about to grab an event (either in step 1, or in step 2). If it is possible to grab an earlier event for the newly free actor, the notifying worker itself is going to do so,

mark the same actor as busy again, and thus undermine its own effort in notifying the other threads. Therefore, it's simpler to just not notify at all, except when new events appear.

Chapter 4

Lingua Franca

4.1 Motivation

Pure actor models like LTA avoid several pitfalls with threading models, but can suffer from concurrency problems of their own. For example, consider the program in Listing 4.1. If `actor_1_loop` and `actor_2_loop` are both started at the same time, the sequence of states it prints often looks like 0,1,3,3,5,5 rather than 0,1,2,3,4,5 like one might expect. This is because depending on which `schedule_event` call occurs first, either of `actor_0_print` or `actor_0_increment` could be called first. Unfortunately, the program exhibits nondeterministic behavior.

Bugs related to nondeterministic behavior are significantly harder to fix compared to their deterministic counterparts. The number of states a program can be in rises exponentially with the number of threads, further exacerbating this issue.

Going back to Listing 4.1, we can view the 4 actions as happening at the same “logical time”. In reality, the start times of the actions are separated by a few microseconds, but on an infinitely fast microprocessor, all 4 could execute before the clock ticks over to the next microsecond. The race condition occurs because although the actions all occur at the same logical time, the order they execute depends on the physical time of their execution (which is nondeterministic). Therefore, nondeterminism can be avoided by structuring the program flow based on logical time rather than physical time, and deterministically ordering messages that occur at the same logical time.

4.2 Concepts

Lingua Franca is a polygot coordination language that provides deterministic concurrency. Rather than being a general-purpose programming language by itself, it “wraps” other programming languages such as C, Python, and Rust. The overall structure of an LF program uses LF syntax to define the reactors, reactions, and other timing features, and another


```

1   int actor_0_state;
2
3   void actor_0_print(...) {
4       printf("%d\n", actor_0_state);
5   }
6
7   void actor_0_increment(...) {
8       actor_0_state += 1;
9   }
10
11  void actor_1_loop(uint64_t t, ...) {
12      schedule_event(&q, t, 0, actor_0_print, NULL);
13      schedule_event(&q, t + 1000000, 1, actor_1_loop, NULL);
14  }
15
16  void actor_2_loop(uint64_t t, ...) {
17      schedule_event(&q, t, 0, actor_0_increment, NULL);
18      schedule_event(&q, t + 1000000, 2, actor_2_loop, NULL);
19  }

```

Listing 4.1: Example of an actor program with a race condition. Some arguments are omitted for clarity.

```

1   reactor Printer {
2       input print: bool
3       input increment: int
4       state x: int
5
6       reaction(print) {=
7           printf("%d\n", self->x);
8       =}
9
10      reaction(increment) {=
11          self->x += increment->value;
12      =}
13  }

```

Listing 4.2: Example of a Lingua Franca reactor, implementing the same logic as actor 0 from Listing 4.1. Lingua Franca guarantees that reactions with the same logical timestamp run in order, so this reactor will always print before it increments.

programming language to write the bodies of each reaction. An example is shown in Listing 4.2.

Reactions and Reactors

Lingua Franca reactors are similar to actors in other actor frameworks, and reactions are similar to the actions described in Chapter 3. However, reactors have some slightly different semantics from other actor models in order to provide concurrency guarantees:

- A reactor has explicit inputs and outputs. Rather than sending messages directly to other actors, a reactor sets its outputs, which are connected to the inputs of other reactors.
- Inputs and outputs are only set once per timestamp. If an output is set multiple times at a given timestamp, only the last one takes effect.
- Reactions within a reactor are run one at a time. If multiple reactions are run at a given timestamp, they will always run top to bottom.

Logical and Physical Time

On an infinitely fast computer, all events scheduled for a given timestamp would happen instantly at that timestamp. Lingua Franca uses this as its model of logical time; this is essentially equivalent to release time (as described in 3).

The logical time of the system always increments in lockstep across the whole system. Naturally, this means it always lags slightly behind the physical clock of the system.

Reactions can be scheduled either with timers, or with actions (actions in Lingua Franca are similar to events from Chapter 3). Timers have a constant period, and always trigger reactions once per period. By contrast, actions can be scheduled to trigger a reaction at any arbitrary point in the future.

4.3 Platform API

Lingua Franca, as the name implies, aims to support a wide variety of languages and platforms. Within the C language specifically, Lingua Franca aims to support OS platforms like Windows, Linux, and MacOS, RTOS platforms like Zephyr, and bare-metal embedded platforms like Arduino and the RP2040.

Lingua Franca has two runtimes for the C language: a single-threaded runtime and a multi-threaded runtime. The single-threaded runtime is more efficient for single-core platforms or programs that do not require parallelism, while the multi-threaded runtime can take advantage of multiple cores at the cost of some overhead. To make the runtimes as

portable as possible, they are implemented on top of an internal platform API, which is then implemented for every platform that Lingua Franca supports.

A simplified view of the platform API is presented in Listings 4.3 and 4.4. The single-threaded runtime presents facilities to get the time, sleep, and wake up the scheduler from an interrupt. In contrast, the multi-threaded runtime emulates a subset of the pthreads interface, with `lf_cond_signal` and `_lf_cond_timedwait` taking on the roles of `lf_interrupt` and `_lf_interruptable_sleep_until_locked`. In theory, one could port Lingua Franca to an entirely new platform by simply implementing these functions.

```

1  void _lf_initialize_clock(void);
2  int  _lf_clock_gettime(instant_t* t);
3  int  _lf_interruptable_sleep_until_locked(environment_t* env,
      instant_t wakeup_time);
4  int  lf_disable_interrupts_nested();
5  int  lf_enable_interrupts_nested();
6  int  _lf_single_threaded_notify_of_event();

```

Listing 4.3: Lingua Franca platform API for the single-threaded runtime.

```

1  void _lf_initialize_clock(void);
2  int  _lf_clock_gettime(instant_t* t);
3  int  lf_available_cores();
4  int  lf_thread_create(lf_thread_t* thread, void *(*lf_thread)
      (void *), void* arguments);
5  int  lf_thread_join(lf_thread_t thread, void** thread_return);
6  int  lf_mutex_init(lf_mutex_t* mutex);
7  int  lf_mutex_lock(lf_mutex_t* mutex);
8  int  lf_mutex_unlock(lf_mutex_t* mutex);
9  int  lf_cond_init(lf_cond_t* cond, lf_mutex_t* mutex);
10 int  lf_cond_broadcast(lf_cond_t* cond);
11 int  lf_cond_signal(lf_cond_t* cond);
12 int  lf_cond_wait(lf_cond_t* cond);
13 int  _lf_cond_timedwait(lf_cond_t* cond, instant_t wakeup_time
      );

```

Listing 4.4: Lingua Franca platform API for the multi-threaded runtime.

4.4 Adaption to Embedded Platforms

When Lingua Franca was first developed, it only targeted OS platforms such as Windows, Linux, and MacOS. This simplified early development, but also constrained the platform API. However, as the project grew, work was put into making the project more portable, which exposed a series of unique challenges.

Single-core platforms

The earliest efforts were to port the single-threaded runtime to embedded platforms. The single-threaded runtime was extremely simple at this time, with the only required functions being `_lf_initialize_clock`, `_lf_clock_gettime`, and `lf_sleep`.

The two earliest platforms to be ported were the NRF52 and Arduino family of processors¹. Each presented unique challenges. In particular, the NRF52 presented a need to schedule actions from interrupts, which led to some reorganization of the runtime and the introduction of interrupt-related functions to the platform API. The Arduino support presented a need to optimize for extremely resource constrained chips, which is still in progress today.

Zephyr

The next platform to join the embedded systems list was an implementation of the runtime on top of the Zephyr RTOS. This, in theory, allows Lingua Franca to run on any platform that Zephyr can run on. However, the addition of the Zephyr kernel adds overhead to the runtime, so this solution is not ideal.

One of the hurdles that needed to be solved with the port to Zephyr was the handling of atomics. The multi-threaded runtime uses atomic primitives (i.e. atomic add) to improve scheduler performance, but atomic primitives are not available on many embedded platforms. Therefore, a polyfill was created to replace atomic instructions with a mutex-based implementation on platforms that do not support them.

Bare metal multi-core platforms

After the port to Zephyr, efforts were started by Abhi Gundrala to port both runtimes to the RP2040. This represents the first platform that supports the multi-threaded runtime, but without an OS or RTOS.

This is where our contributions to Lingua Franca begin. In order to finish the port to the RP2040, we implemented the following:

¹One of the first research projects I ever worked on was LF runtime support for the NRF52. Back then, I barely knew how to write C, and most of the tooling was a complete mystery to me. Now I am older and wiser and still barely know how to write C.

- Implementations of underlying synchronization primitives: the multi-threaded runtime relies on mutexes and condition variables. The mutexes could be directly implemented using mutexes from the RP2040's standard library. However, condition variables are not present in the RP2040's standard library, so those were implemented on top of semaphores, relying on the fact that the RP2040 only has two cores.
- Implementation of `lf_thread_create`: On other multi-threaded platforms (OSes and RTOSes), the platform supports an arbitrary number of threads, and `lf_thread_create` can simply create one. However, in a bare-metal environment running on an multicore processor, `lf_thread_create` needs to assign the thread function to one of the available cores. Generally, the n th invocation of `lf_thread_create` should assign the function to core $n + 1$. The RP2040's standard library contains `multicore_launch_core1` for doing this, so `lf_thread_create` simply needs to check that it isn't being called twice and call `multicore_launch_core1`.
- Modifications to runtime thread creation: before, to create N workers, the multi-threaded runtime would create N threads, then join them all from the main thread. This is fine for OS platforms, as the idle main thread is relatively inexpensive. However, for the RP2040, which only has two hardware threads, this strategy would only allow one worker (defeating the purpose of the multi-threaded runtime). To remedy this, we modified the runtime to also run a worker on the main thread, thus allowing two workers on the RP2040, and generally allowing N workers for an N -core embedded system.
- Critical sections for interrupts: in the multi-threaded runtime, the global data structures (specifically, the event queue) are protected via mutexes. However, on an embedded platform with interrupts, this can deadlock if an interrupt handler tries to schedule an action. To remedy this, access to the event queue needs to be protected with a critical section (which disables interrupts) rather than a mutex. The implementation of this change is in progress as of the time of writing.

The end result of these changes is that the multi-threaded runtime is now equipped to efficiently run Lingua Franca programs in bare-metal environments. Generally, the runtime creates N workers that get assigned directly to the N cores of the processor; these workers can then run indefinitely with no context switches (apart from those triggered by interrupts). This allows the Lingua Franca scheduler to take full advantage of different processors, simply by scaling the number of workers to the number of cores.

Chapter 5

Memory

One common concern with threading-based systems is the memory overhead. In this chapter, we analyze that concern and compare theoretical memory usages between FreeRTOS, LTA, and Lingua Franca.

5.1 FreeRTOS Memory Usage

The main reason why threads tend to have a high memory overhead is that you need a separate stack for each thread. The first problem is determining how much stack a given thread will use. It is sometimes possible to statically analyze how much stack a given function will use, but this is a nontrivial task and adds extra complexity to an engineer’s workflow. In particular, analyzing the stack usage of a given thread involves expanding the call tree of its entry point function (i.e. with the `dump-rtl-expand` compilation flag) and adding up the stack usage of each function (using information from the `stack-usage` compilation flag).

The easier thing to do is initialize each thread stack to a “safe” amount, then increase it if irregularities are noticed. Generally, “irregularity” means a program crash (i.e. a segmentation fault on a general-purpose OS), or worse. Thankfully, several RTOSes (including Zephyr and FreeRTOS) have capabilities to detect stack overflows, although FreeRTOS notes that this introduces context switch overhead.

What is a “safe” amount for a thread stack? On Linux, the default stack size is 8 MB. Given that most microcontrollers have RAM measuring in the kilobytes, this may not be a wise choice for a microcontroller program. On the RP2040, which has 264kB of total SRAM, each core’s main thread is given 2 kB of SRAM by default (see `.ld` file). Anecdotally, it takes approximately 1kB of stack memory to use `printf` (with USB communication to the host), so 2kB is a reasonable default.

The maximum number of possible threads is dependant on the heap size and the size of each thread stack. On the RP2040, the `.data` section, `.bss` section, and heap all share the 4 large SRAM banks which contain a total 256 kB of RAM. When building FreeRTOS, the

Stack size (bytes)	Maximum thread count
128	892
256	607
512	370
1024	207
2048	110
4096	57

Table 5.1: Maximum number of threads in FreeRTOS with different stack sizes

maximum size the heap can be configured to (for a minimal test program) is 245 kB¹. With this heap size configured, we can verify the maximum number of threads by running a test program that, given a stack size S and number of threads N , creates the threads (plus one verification thread with a stack size of 1kB), verifies that they have started correctly, and prints a verification message. The maximum value of N for each stack size is given in Table 5.1.

If we assume that FreeRTOS has a constant memory overhead and a constant overhead per thread, Table 5.1 allows us to deduce that FreeRTOS has a constant overhead of around 7 kB (leaving 238 kB available for threads), and an overhead of 144 bytes per thread. This is negligible for threads with large stacks (like the 2kB default discussed earlier), but may be prohibitive on systems with less memory than the RP2040.

5.2 LTA Memory Usage

LTA has the following memory overheads:

- 48 bytes for the event queue
- 24 bytes for each event
- 1 byte for each actor

Although actors aren't directly comparable to threads, the memory overhead for an actor is still demonstrably lower than a FreeRTOS thread with even a tiny stack. For example, a minimal actor that has a maximum of 1 event at a time uses 25 bytes of memory. In contrast, a minimal thread with a tiny stack size of only 128 bytes still uses 272 bytes of memory, which is an order of magnitude more than the actor.

¹Configuring it higher causes the linker to complain that the `.bss` section overflows the RAM. Interestingly, the FreeRTOS heap is actually a large static array (allocated in the `.bss` section), which is separate from the heap section allocated in the linker script.

What happens if we try to create as many actors as possible? As an extreme example, one could imagine a program with N actors that each just print a message periodically. Each actor has no state, and only creates one event a time, so the amount of memory this program uses is roughly $48 + 25N$. On the RP2040, the maximum value of N for this program is 10068 (determined via testing). This is an order of magnitude more than the number of threads that FreeRTOS could create with 128-byte stacks, and two orders of magnitude greater than the number of threads that FreeRTOS could create with 2 kB stacks.

5.3 Lingua Franca Memory Usage

Similar to LTA, the closest analogue that Lingua Franca has to threads are reactors. Reactors vary in complexity, so the memory usage of a reactor depends on the number of reactions, timers, actions, inputs, outputs, and more. Like LTA, the minimum useful reactor is one that executes a bit of code periodically. In Lingua Franca, this requires just a timer and single reaction.

An an experiment, we can create a bank of these minimal reactors, then use `malloc` to estimate the heap usage (assuming that the next block to be allocated will lie after every currently allocated block). The results are summarized in Figure 5.1. Unsurprisingly, the graph is almost perfectly linear. Using a best-fit line, we can extrapolate that this program uses approximately 360 bytes per reactor, plus a constant 13.3 kB of heap memory. Note that a minimal bare-metal program with IO uses 9.6 kB of heap memory, so Lingua Franca only adds 3.7 kB of constant overhead.

With these minimal reactors, the largest possible number of reactors on the RP2040 is 681. This is more than six times greater than the number of 2 kB threads possible under FreeRTOS on the same hardware. Although such a large number of reactors is unlikely for typical programs, this demonstrates the greater degree of concurrency that Lingua Franca programs can attain under memory constraints.

5.4 Code Size Comparison

The RP2040 has access to 2 megabytes of flash memory, so it may seem like code size is a non-issue. However, the RP2040 uses a 16 kB cache in front of the flash memory for instruction fetches. Keeping a smaller code size allows better cache utilization and therefore faster average execution. This is similarly relevant for other microprocessors that have instruction caches. For example, the ESP32 series uses a 32 kB unified cache per core [11].

To estimate code size between different frameworks (FreeRTOS, LTA, and Lingua Franca), we create a minimal blink program that uses the framework (i.e. in LTA an actor is used and in FreeRTOS a thread is used). The results are as follows:

- On bare metal (no framework used), the binary file is 8.7 kB

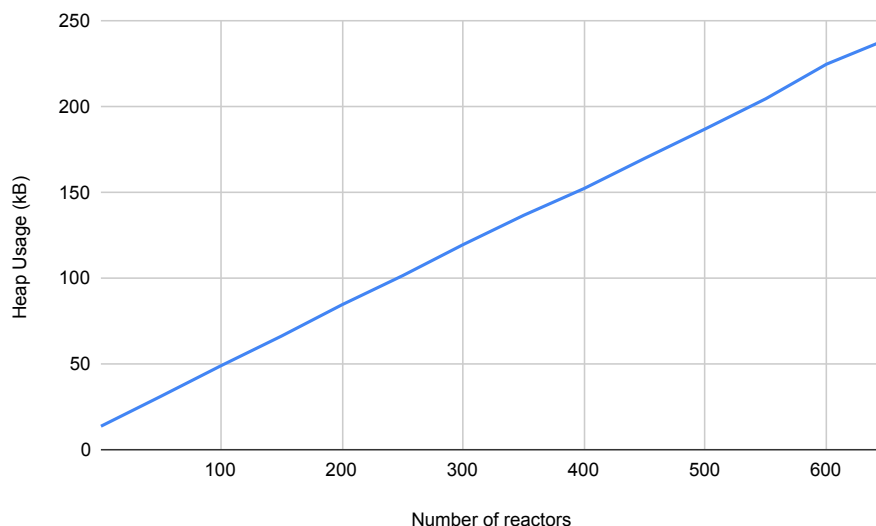


Figure 5.1: Heap usage of a Lingua Franca program with a large bank of reactors.

- For FreeRTOS, the binary file is 23.1 kB
- For LTA, the binary file is 15.1 kB
- For Lingua Franca, the binary file is 96.1 kB

The Lingua Franca result, unlike the others, includes code for communication over USB (since the Lingua Franca runtime prints messages on startup and shutdown). However, the Lingua Franca runtime is rather large, even when accounting for this. Much of the added overhead is due to the inherent complexity of Lingua Franca’s model: it must keep track of events, reactions, and scheduling with respect to a reaction graph.

FreeRTOS also incurs significant overhead, especially when compared to the bare metal and LTA results. However, this is tolerable, especially for larger programs. For reference, the LTA version of the code in Chapter 8 has a 39.8 kB binary.

Chapter 6

Core Interference

For CPU-bound workloads, ideal scaling should give double the performance when using both cores. Unfortunately, this is not the case, even if the cores are working on completely independent tasks. On the RP2040 in particular, interference between the cores is common and has severe implications for performance.

Core interference effects are particularly distressing because they complicate worst case execution time (WCET) measurements. In the presence of core interference effects, a function's running time depends not only its input, but also on the activity on the other core. Given that estimating WCET is difficult even on a single core, these effects make it even harder.

6.1 RP2040 Bus Fabric

The RP2040 contains two cores with no data or instruction cache. Each core accesses the AHB-Lite Crossbar for every memory access, including instruction fetches and data accesses. The Crossbar allows single-cycle access from either core to any of the 10 downstream ports, so both cores may run at full speed as long as they are accessing separate memory devices. However, if both cores try to access the same memory device, one core must stall for a cycle. By default, this is done in a round-robin fashion, but may be configured to favor one core.

6.2 Instruction Fetch Conflicts

Bare Metal

The most common accesses to memory are instruction fetches. Most code is stored in flash memory, which takes several cycles to fetch. However, the RP2040 uses a 16kB XIP (eXecute In Place) cache, so most instruction fetches end up being single-cycle. When only one core is running, this allows almost full speed execution.

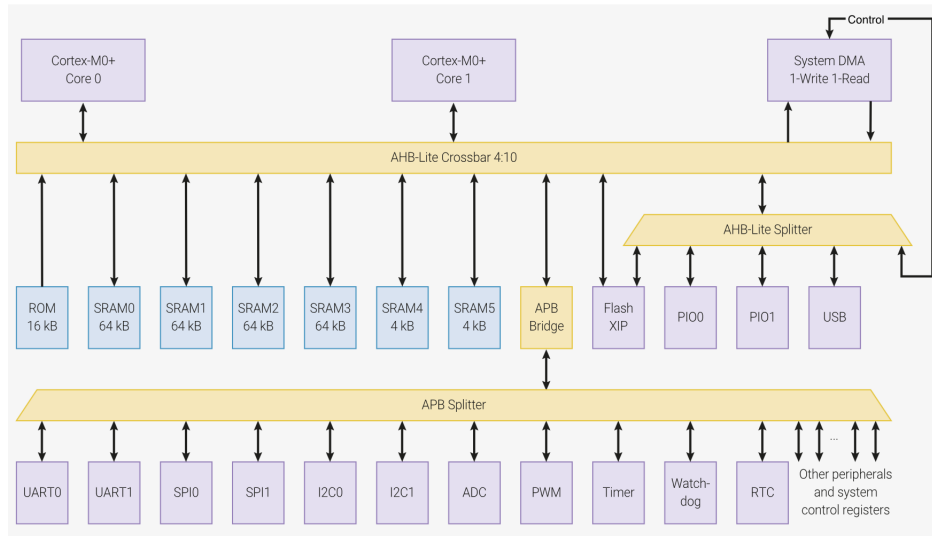


Figure 6.1: RP2040 Bus Fabric (from RP2040 Datasheet [8])

However, when both cores are running, both cores tend to make many accesses to the XIP cache, causing the cores to stall frequently. We can demonstrate this by executing a benchmark on Core 0, while Core 1 runs different workloads. The results of this experiment are in Figure 6.2. Observe that the benchmark (on Core 0) runs quickly when Core 1 is idle, and is significantly slower when Core 1 is also executing the benchmark. Surprisingly, it performs even worse (20% slower) when Core 1 is running an infinite loop consisting of a single unconditional branch.

The infinite loop on Core 1 causes Core 0 to perform especially poorly because the infinite loop causes an instruction fetch on every cycle. The program uses the Arm Thumb instruction set, which is composed mainly of 16-bit instructions, so most programs need an instruction fetch every other cycle. Since the singular branch causes an instruction fetch every single cycle, this causes a conflict every time Core 0 has an instruction fetch. Since Core 0 has an instruction fetch for close to 50% of its cycles, it needs to stall for close to 25% of its cycles. This is consistent with the 20% decrease in performance that we see in the benchmark. The difference can be explained by the presence of multi-cycle instructions, such as PUSH and POP.

FreeRTOS

If we create a single-task program in FreeRTOS, it seems to perform very poorly. In fact, if we run the benchmark from the previous section on a FreeRTOS task, it performs even worse than when both cores run the benchmark on bare metal. We can infer that this is because while the benchmark is running on Core 0, Core 1 is running some non-trivial workload.

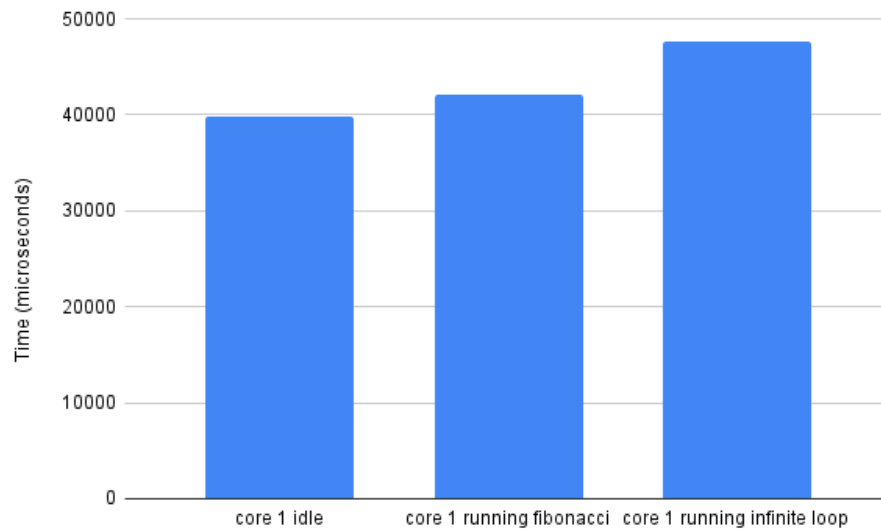


Figure 6.2: Time taken to calculate $\text{fib}(25)$ on Core 0 under different conditions. In the first bar, Core 1 is idle. In the second bar, Core 1 is also calculating $\text{fib}(25)$. In the third bar, Core 1 is running an infinite loop.

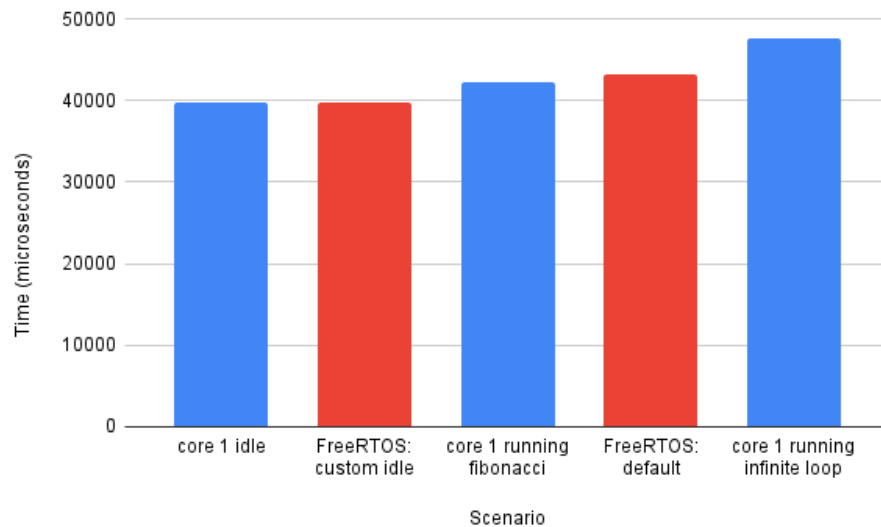


Figure 6.3: Time taken to calculate $\text{fib}(25)$ under different conditions. The blue bars are as they are in Figure 6.2. The two red bars are from running the same benchmark in a FreeRTOS task.

Since there is only a single task (the benchmark), Core 0 is running the idle task, which happens to be a busy waiting loop.

Luckily, this is a fixable problem. FreeRTOS allows applications to configure their own idle task, so we can configure the idle task to execute the WFE instruction in a loop. The WFE instruction puts the core into a low-power state until an event is raised. Since events are raised infrequently, this causes the core to be idle almost all the time. With this change, the benchmark matches the performance of the best bare-metal result, as shown in Figure 6.3.

Given that applications in FreeRTOS get significantly lower performance by default, to be fair to FreeRTOS the rest of the benchmarks in this thesis will use this modification.

6.3 Data Memory Latency

Instruction fetches are not the only activity on the main bus; all loads and stores must use it as well. Similar to how one core must stall if both are performing an instruction fetch from the XIP cache, one core must also stall if both are performing a load or store on the same SRAM bank. While not as frequent as instruction fetches, loads and stores still account for a large portion of most programs, so this is a significant source of stalling.

It turns out that the way RTOS's manage thread memory exacerbates this problem. In fact, under the right conditions, the Fibonacci benchmark runs 19% slower on FreeRTOS as a result of data memory stalling. Ordinarily, the effect is not this severe, but this still reveals a fundamental detriment to using an RTOS.

RP2040 Memory Layout

The rp2040 contains a 16kB ROM, 264kB of total SRAM, and access to external flash memory. The SRAM is partitioned into six banks: four large 64kB banks (which we will refer to as banks 0-3), and two small 4kB banks (which we will refer to as “scratch X” and “scratch Y”).

The SRAM banks are mapped to the addresses shown in figure 6.4. Notably, there are two ways to access the memory in the large SRAM banks: either individually in the upper region, or via the word-stripped memory in the lower region.

These SRAM banks are, for the most part, single cycle. However, as mentioned before, if both cores try to access the same bank in the same cycle, one core must stall for a cycle. The word-stripped memory serves to alleviate this problem: if both cores are doing sequential accesses, there is a much lower probability of a conflict if they access the word-stripped memory.

When running bare-metal programs the default linker script puts the core 0 stack in Scratch X, the core 1 stack in Scratch Y, and the heap in the striped region. This guarantees single-cycle accesses to the stack. However, this strategy only works when there are exactly two stacks (one per core). This assumption holds for bare-metal programs, but an RTOS

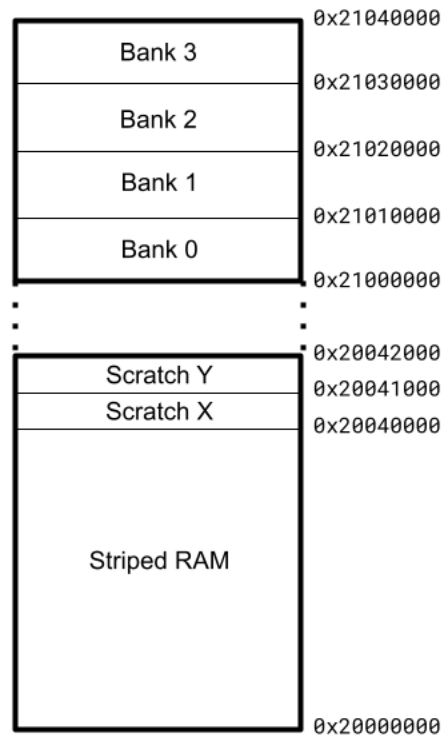


Figure 6.4: Memory map for rp2040.

must schedule many concurrent threads, each with their own stack. This means that in an RTOS, thread stacks must be allocated from the heap. As a result, stack accesses are no longer guaranteed to be single-cycle, leading to slightly lower performance.

Putting the heap in the striped SRAM region tends to minimize conflicts for heap accesses, and therefore minimize the performance loss from using heap-allocated stacks. However, it may not always be desirable to use striped SRAM. For example, if the application uses a very large data buffer, it may be desirable to dedicate a large SRAM bank to it to improve DMA performance. In this case, the heap (and consequently, the thread stacks) must reside in the non-striped region, presumably occupying the other 3 large banks.

Since using the non-striped region has very different performance characteristics from using the striped region, we test both scenarios.

Experimental Setup

To investigate the effects of data memory stalling, we compare the average execution time of each benchmark using the following methods:

- FreeRTOS: the benchmark is run in two parallel threads. To reduce the scheduling overhead, the system tick rate is set as low as possible (10Hz).
- LTA: the benchmark is run in two parallel actors.
- Lingua Franca: the benchmark is run in two parallel reactors.
- Bare metal: the benchmark is run directly on both cores of the RP2040.

For each method, we also test it using the striped region for the heap, and using the blocked (non-striped) region for the heap. To eliminate as many variables as possible, the benchmarks are written in assembly, and not inlined. This prevents GCC optimizations from affecting timing results. In addition, the code is compiled in release mode in all scenarios, to make the scenarios as realistic as possible¹. All benchmarks are run on the same hardware. During each benchmark, the benchmarking function is run periodically 100 times, and the results averaged.

Fibonacci Benchmark

The Fibonacci benchmark is a simple, recursive Fibonacci function. The majority of its execution time is dominated by recursive function calls, making it a good candidate for revealing the effects of stack access conflicts.

Figure 6.5 shows the average execution times of the Fibonacci benchmark. Almost all of the results are within 1% of each other, with the only exception being FreeRTOS using the blocked heap. That result is 19% worse than the others, clearly showing the detrimental effects of frequent stack access conflicts.

From the data in Figure 6.5, it may appear that using the striped region for the heap effectively eliminates the timing consequences of heap-allocated stacks. While it is true that FreeRTOS and the other frameworks have a similar average execution time when using a striped heap, the timing results for FreeRTOS are much less consistent than the others. Figure 6.6 shows the standard deviation of 100 trials for each scenario. As a result of stack access conflicts, the standard deviations of FreeRTOS execution times are an order of magnitude greater than the others.

Interestingly, the standard deviations of Lingua Franca's results were much lower than any of the others. In both tests, Lingua Franca's standard deviations were under a microsecond, which is not even visible in Figure 6.6. It is unknown why they are so low, but one plausible explanation could be slight differences in execution time starts reducing inter-core conflicts.

Matrix Multiplication Benchmark

Matrix multiplication is a computationally heavy task, especially for large matrices. Normally such matrices would be heap allocated or statically allocated, but might be stack

¹As an extra precaution, the benchmarks were also repeated at different optimization levels. The results are within margin of error of each other, so only the release mode results will be reported.

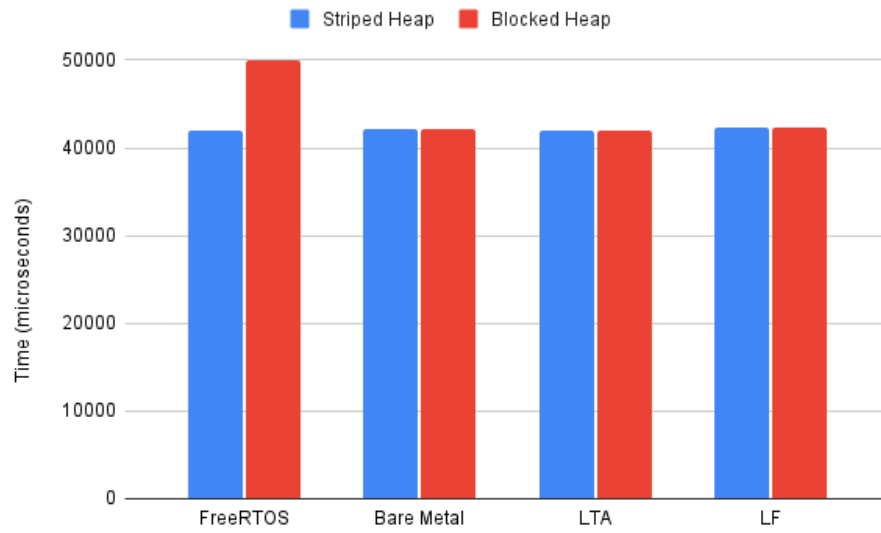


Figure 6.5: Average execution time of Fibonacci benchmark.

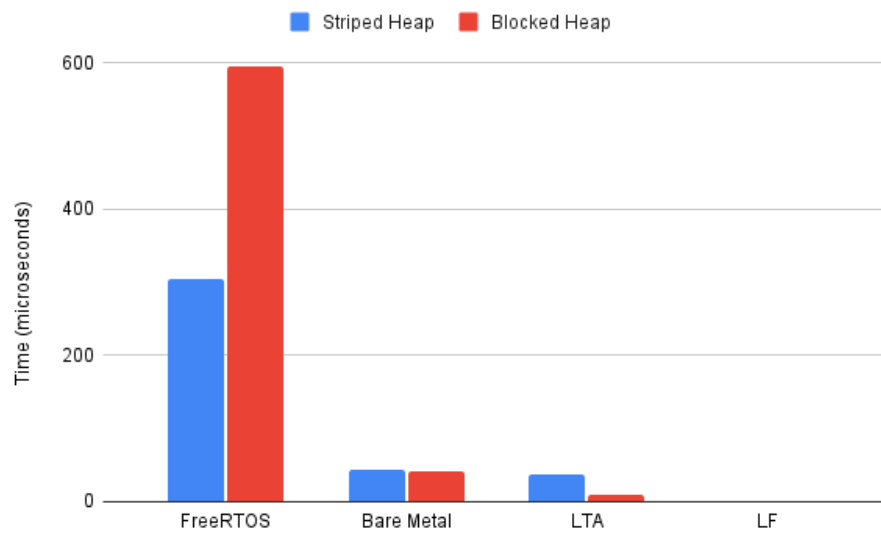


Figure 6.6: Standard deviation of execution time of Fibonacci benchmark.

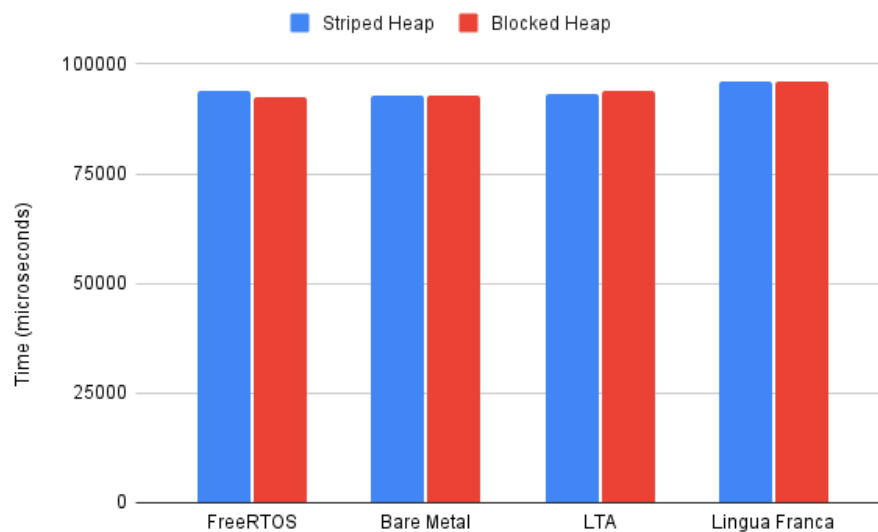


Figure 6.7: Average execution time of the matrix multiplication benchmark.

allocated in niche situations. The matrix multiplication benchmark repeatedly multiplies two small stack-allocated 10 by 10 matrices.

However, despite using a significant amount of stack memory, the matrix multiplication benchmark actually does not spend much of its time doing stack accesses. The inner loop consists mostly of `mov`, `mul` and `add` instructions (to calculate indices), so the effects of loads and stores are minimal.

Despite the benchmarking having a low number of stack accesses, the timing results are still interesting. Figure 6.7 shows the average execution times of the matrix multiplication benchmark. Most of the averages are within 1% of the bare metal result, with the exception of Lingua Franca, which was around 3% slower. The cause of the slowdown is unknown, but could be due to a lower hit rate in the XIP instruction cache. Figure 6.8 shows the standard deviations of the execution times. Interestingly, the biggest outlier here is FreeRTOS when using the striped heap. The execution times for this case range between 91 milliseconds and 97 milliseconds, which is a much larger range than the other cases. This variation could be due to stack access conflicts, but given the relatively small proportion of stack accesses in the benchmark, there may be another factor at play. Regardless, this reveals that using FreeRTOS adds significant variability to execution timings.

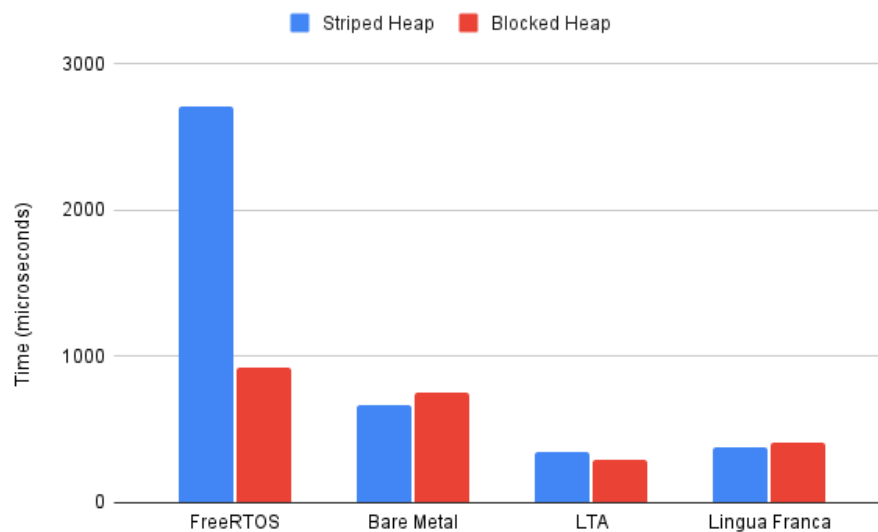


Figure 6.8: Standard deviation of execution time of the matrix multiplication benchmark.

6.4 Conclusions

This chapter analyzed three different interference affects: instruction fetch conflicts, general data access conflicts, and stack access conflicts. Both bare-metal approaches and RTOSes suffer from the first two, but the third only occurs on RTOSes. As a result, applications may suffer a performance penalty from using an RTOS, irrespective of the scheduling overhead. In most real world programs, the detriment to average execution time may be negligible, but there is a significant detriment to timing variability.

Although the performance results in this section are specific to the RP2040, this conclusion applies to any system that has dedicated memory units available for instruction stacks. For example, this applies to any system that has per-core data scratchpad memory.

In systems where stack memory is not treated differently from heap memory, this conclusion does not apply. However, the use of small dedicated SRAMs for program stacks has evidently improved performance on the RP2040, so this technique may be used more commonly in future microprocessors.

Chapter 7

Timing

When building timing critical systems, there are two main concerns: how precisely we can control the time at which code will run, and how much code we can run without missing deadlines. Obviously, the execution throughput is crucial, as higher throughput opens the door to more complex algorithms or running algorithms more frequently. The timing precision has an equally important role to the overall responsiveness of the system. For example, imagine a flight controller on a quad-rotor drone. The flight controller might have a central control algorithm, which uses accelerometer data to adjust motor current to keep the drone upright. If the control algorithm runs at irregular times, this will negatively affect the worst-case time until the control algorithm responds to a perturbation. On the flip side, if the control algorithm is run at extremely regular times, then a discrete-time algorithm can be used instead of a continuous-time algorithm, potentially simplifying the computation.

Given the importance of both timing precision and throughput, this chapter analyzes and contrasts the timing performance of FreeRTOS, LTA, and Lingua Franca.

7.1 Causes of Timing Variation

In an ideal world, the execution time of any algorithm would depend solely on its input and be perfectly predictable. Unfortunately, in all but the simplest processors, micro-architectural innovations improve performance, but also add variation to timings. Modern high-performance processors are full of these innovations: branch prediction, memory caches, frequency scaling, and out-of-order execution are just a few.

Thankfully, the RP2040 and similar microcontrollers have almost none of these innovations. However, many have at least a few sources of timing variation, such as caches and arbitration to common data buses (in Chapter 6, we demonstrated that arbitration to SRAM banks was a particularly impactful source of timing imprecision on the RP2040). However, the innovations in high-performance processors may eventually trickle down into microcontrollers, meaning that execution timings may become even less predictable than they are today.

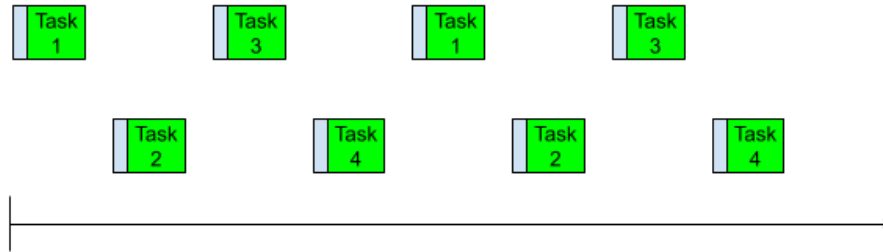


Figure 7.1: Periodic, interleaved tasks.

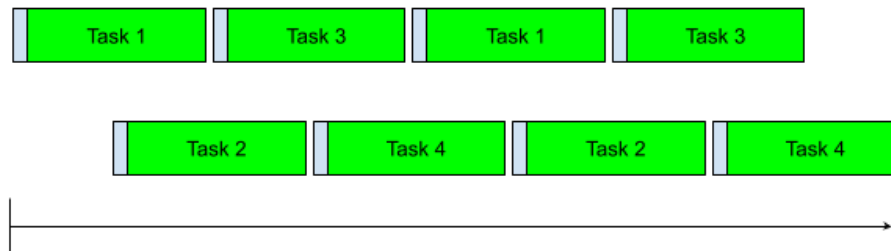


Figure 7.2: Periodic, interleaved tasks.

7.2 Experimental Setup

Regardless of which concurrency framework a program uses (an RTOS, LTA, Lingua Franca, or something else entirely), all frameworks add some execution time overhead to the program. In FreeRTOS, this mainly comes from the scheduler during system ticks; in LTA and Lingua Franca, this mainly comes from the manipulation of data structures when actions/reactions are scheduled or released. This slightly reduces execution throughput, and may affect precision as well. To characterize these effects, we design a simple system to measure both execution time and precision.

The system

Consider a system with several periodic, interleaved tasks. At low processor utilization, the system may be scheduled onto the RP2040's two cores as in Figure 7.1. As the tasks require

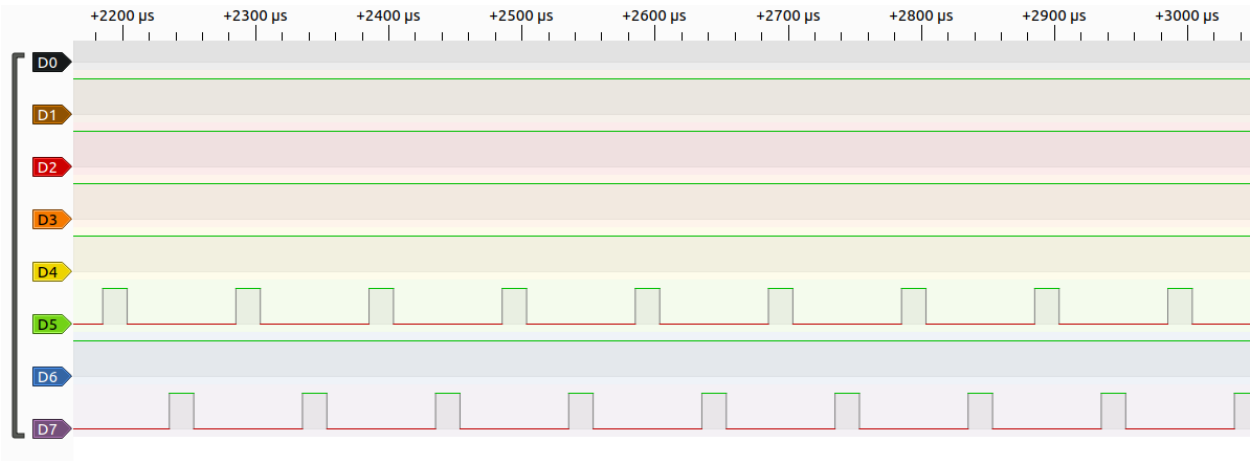


Figure 7.3: Periodic, interleaved tasks, as observed with a logic analyzer.

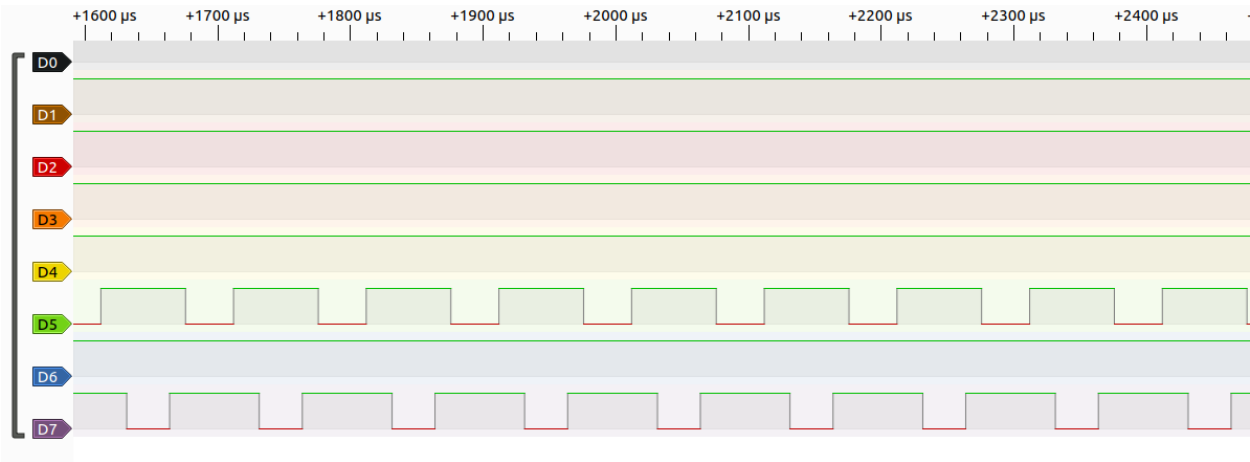


Figure 7.4: Periodic, interleaved tasks, as observed with a logic analyzer.

more computation, the processor utilization may increase, until the system looks like Figure 7.2. In order to observe the timings of these tasks, we can drive GPIO pins at the beginning and end of each task, and observe them using a logic analyzer. Figures 7.3 and 7.4 are screenshots of this.

To keep the system as simple as possible, a constant frequency of 10kHz is used for all tasks throughout the experiment, and only two tasks are used (one per core). FreeRTOS can only schedule tasks to multiples of the system tick rate, so this requires setting the system tick rate to 20kHz, and driving both tasks once every two ticks. Since each framework only needs to handle scheduling one task at a time, this should present a best case for the timing overhead of all three frameworks.

The workload the tasks will execute is an function called `sumsq`, which computes $0^2 + 1^2 + 2^2 + 3^2 + 4^2 + 5^2$. This takes roughly 0.4 microseconds on the RP2040, so we can control the execution time of the task by simply executing this function a set number of times. For example, the first experiment is designed with 100 microsecond tasks, which is equivalent to 250 loop iterations.

This system can be modelled idiomatically in all 3 frameworks: FreeRTOS uses two looping threads that call `xTaskDelayUntil` to wait for the start of the next task, LTA uses two actors with actions that re-schedule themselves, and Lingua Franca uses two timers driving two reactions. However, Lingua Franca has a limitation that slightly interferes with this assessment: since logical time must advance globally and the two reactions have offset times, there is no way to create staggered reactions that run in parallel¹. Therefore, to assess Lingua Franca at high utilization, we only run one task, and leave the other core idle. Unfortunately, this means that the results for Lingua Franca are not directly comparable to the other two frameworks, but they still give insight into Lingua Franca's performance.

The measurements

There are a few metrics which will be relevant for all experiments:

- **Loop Iterations:** The number of times each task computes `sumsq`.
- **Task execution time:** The amount of time spent inside each task.
- **Total execution time:** The time between the start time of a task and the start time of the next task. Ideally, this should be 100 microseconds.
- **Start time deviation:** The standard deviation of the total execution times. If each task is started exactly on time, each total execution time will always be exactly 100 microseconds, so the start time deviation will be 0. If there is some imprecision in the start times of the tasks, the start time deviation will be positive.

¹Actually, Lingua Franca recently introduced *scheduling enclaves*, which can have decoupled logical times. However, at the time of writing, scheduling enclaves have not been tested on the RP2040.

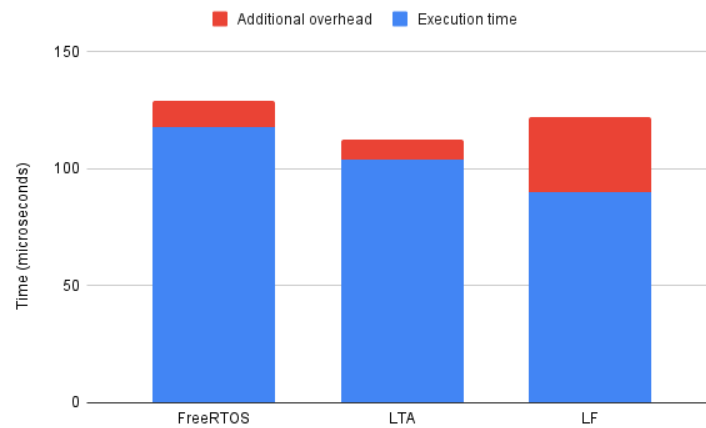


Figure 7.5: Task execution times and total execution times with 250 loop iterations.



Figure 7.6: Task execution times for FreeRTOS and LTA.

7.3 Overhead

In order to measure the overhead of each framework, we set the loop iterations to 250, guaranteeing that the system will not be able to keep up with running the tasks at 10kHz. In this situation, the system will not sleep between each task, so the difference between the total execution time and the task execution time should give the overhead. Figure 7.5 shows the results of this experiment.

In Figure 7.5, LTA has an average task execution time of 103.9 microseconds and an average total execution time of 112.5 microseconds, giving it an overhead of 8.6 microseconds.

	Overhead (microseconds)
FreeRTOS	24.7
LTA	8.6
Lingua Franca	32.0

Table 7.1: Timing overheads of each framework (lower is better).

Similarly, Lingua Franca has an average task execution time of 90.0 and an average total execution time of 122.0, giving it an overhead of 32.0. Lingua Franca’s lower task execution times can be explained by the fact that it’s only running one task, which improves efficiency for reasons explained in Chapter 6. By the same calculation, FreeRTOS seems to have an overhead of 10.3, but this ignores FreeRTOS’s higher than expected task execution time.

FreeRTOS does most of its scheduling work during system ticks, which trigger the scheduler via an interrupt. In this system, the system tick rate is 20kHz, so a system tick usually occurs in the middle of each task. This means that on average, the task execution time is increased by the length of the system tick handler. By comparing the task execution times between LTA and FreeRTOS, we can deduce that the system tick handler takes around 10 microseconds. We can verify this by comparing task execution times for a range of loop iterations. Figure 7.6 shows that regardless of utilization level, the task execution times tend to differ by around 10 microseconds, which is consistent with it being due to system tick overhead. Therefore, to calculate FreeRTOS’s overhead, we’ll subtract FreeRTOS’s average total execution time (128.6) and LTA’s task execution time (103.9), to obtain an overhead of 24.7 microseconds.

The overheads of these three frameworks are summarised in Table 7.1.

7.4 Precision at Low Utilization

In order to determine the best case for the precision of each framework, we set the loop iterations very low (10 iterations), then measure the start time deviations. Since each task will spend very little time computing and most of its time sleeping, this essentially tests how precisely each framework can wake from sleep.

The results of this experiment are summarized in Figure 7.7. FreeRTOS and LTA have almost 0 start time deviation, while Lingua Franca has a relatively high start time deviation of 4.3 microseconds. This might be a result of Lingua Franca’s relatively high overhead. In this experiment, the Lingua Franca runtime needs to service the global event queue every 50 microseconds. If this takes 30 microseconds each time, then Lingua Franca suffers from 60% utilization on Core 0 from overhead alone.

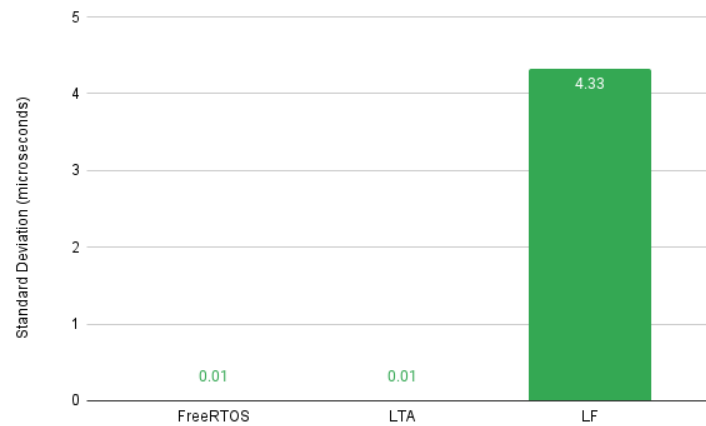


Figure 7.7: Start time deviations at 10 loop iterations.

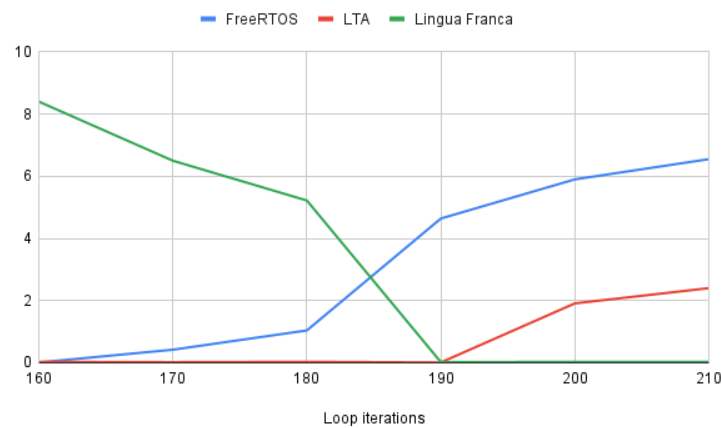


Figure 7.8: Start time deviations with high loop iterations.

7.5 Precision at High Utilization

In order to stress test each framework, we set the loop iterations high (160-210 iterations), then measure the start time deviations. The loop iterations are high enough that in some data points, the system is barely able to keep up, and in other data points, the system is not able to keep up a 10kHz task frequency.

The start time deviations are shown in Figure 7.8. The results can be interpreted as follows:

- FreeRTOS: at 160 loop iterations, the system has almost 0 start time deviation. How-

ever, this slowly rises until 190 iterations, where the system stops being able to keep up. At 190 iterations and above, the start time deviation rises sharply and stays high.

- **LTA:** the system has almost 0 start time deviation until 200 iterations, where it starts to rise. The system is able to keep a 10KHz task frequency until 220 iterations (outside the range of Figure 7.8). Interestingly, at 220 iterations and beyond, the start time deviation falls to 0 again as the system stops sleeping.
- **Lingua Franca:** The system has a high start time deviation, but this drops until 190 iterations, where the system stops being able to keep up. Interestingly, at 190 iterations and beyond, the start time deviation falls to 0 as the system stops sleeping.

The interesting behavior of LTA and Lingua Franca can be explained by the fact that they both use the same underlying sleeping mechanism: waiting on a semaphore. It appears that although semaphore timeouts are precise for long sleep intervals, they become very imprecise for short sleep intervals. Therefore, both systems can potentially be improved by using a different sleeping mechanism, such as the RP2040's hardware alarms.

Chapter 8

Case Study: Tunnelling Ball Device

So far, the focus has been on performance. However, a large part to the success of threading has been because it provides a simple and understandable model for processing concurrent tasks. Many projects choose to use an RTOS despite performance drawbacks because of this. Therefore, for an alternate model to be successful, it needs to provide a usable and ergonomic way to manage concurrency.

To demonstrate that LTA and Lingua Franca can meet this goal, we build a control system using 3 strategies to build the software: LTA, Lingua Franca, and Lingua Franca with programmable IO. Since each version of the software has the same requirements and expected behavior, this serves as a useful comparison to see what a non-trivial system looks like from a programming perspective.

The system in question will be a tunnelling ball device, similar to the one built by Jeffrey C. Jenson in his thesis *Elements of Model-Based Design* [5]. The device drops a steel ball bearing towards a target below a rapidly spinning disk. The disk has two small holes in it, through which the bearing may pass, but the timing of the ball and the disk must be precisely aligned in order for the ball to not collide with the disk. Done successfully, the system gives the impression that the ball has magically teleported through a solid disk.

A video of the device can be found at https://www.youtube.com/shorts/ppC6_Cjvkjw.

8.1 Hardware Design

The main components of this system are:

- A wooden disk with two small holes bored in it.
- A stepper motor, spinning the disk. This stepper motor is driven by an A4988 stepper motor driver.
- A 35cm acrylic tube for dropping the ball bearing through.

- An electromagnet at the top of the tube, for releasing the ball. The electromagnet is driven by a MOSFET module.
- A 12 volt DC power supply. A large 100 μF capacitor is connected across the inputs of the power supply to protect the circuit from voltage spikes. This power supply provides power to the stepper motor and the electromagnet.
- Two photogates, located 5mm and 60mm from the bottom of the tube. Each photogate consists of an infrared LED and a phototransistor, spaced 14mm apart. These photogates track the timing of the ball so that the motor may speed up or slow down as needed.
- A large button, used to start the system.
- Three status LEDs. Two green LEDs are connected in parallel with the photogate's infrared LEDs. Because the infrared LEDs have a forward voltage drop of 1.2V, and the green LEDs have a forward voltage drop of 2V, these should stay off when the system is working correctly. If the infrared LEDs burn out or become disconnected, the green LEDs turn on to indicate a problem. The third LED is a red LED that indicates when the system has power.

Figure 8.1 shows how the electrical components are wired together. Figure 8.2 shows photographs of the device. The mechanical construction of the device is generally of “prototype” quality, as evidenced by several key details:

- The body of the device is mainly plywood scraps that have been hot-glued together.
- The tube is affixed to the device using several zip-ties.
- The breadboard is a structural element.
- The stepper motor is clamped to the body of the device. An attempt was made to hot-glue this instead, but the glue was not strong enough.
- The “landing zone” is made of cardboard. This may actually be optimal, as the material needs to be soft enough to absorb some of the energy of the ball bearing.

8.2 Physical Constraints

The system looks most impressive at high speeds, so the primary goal is to have the ball fall through the hole when the motor is spinning as fast as possible. In initial testing, the particular stepper motor used for this project can handle up to 10 rotations per second in 1/16th stepping mode, with no load. However, with a disk attached, the maximum speed

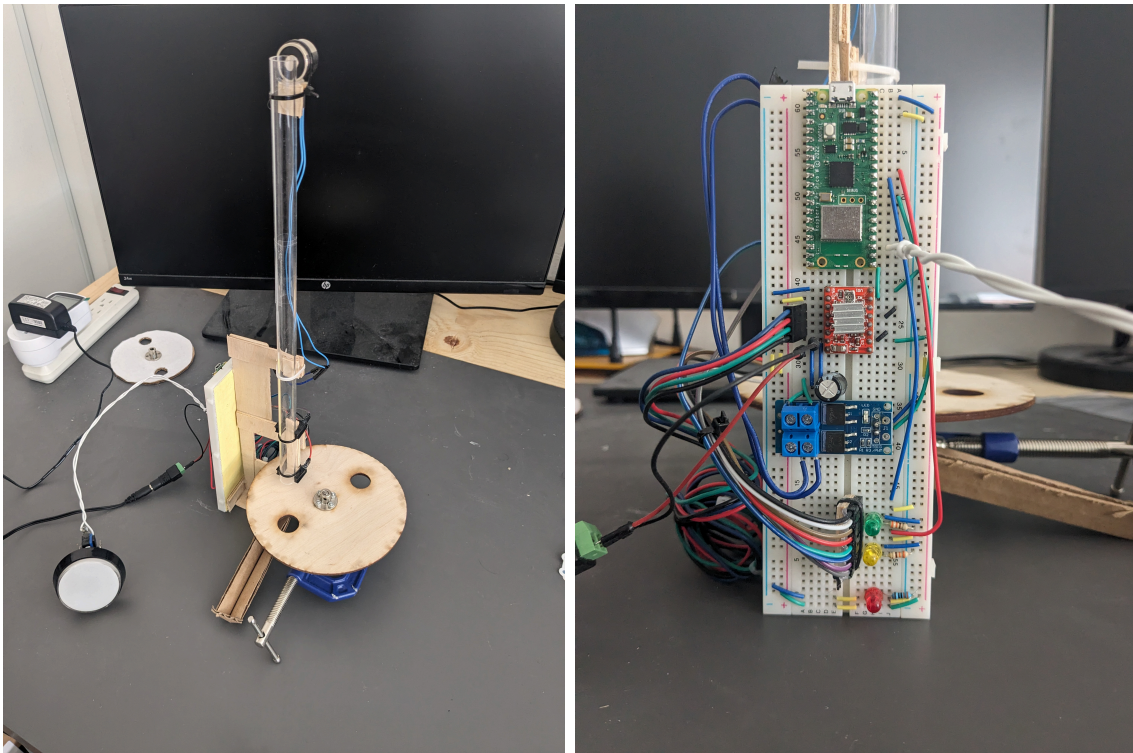


Figure 8.2: Photographs of tunnelling ball device.

the motor can reach is only 5 rotations per second. This sets our “maximum target speed”, which the rest of the system is designed around.

With the speed set, the next goal is to minimize the size of the hole needed (ideally, the size of the hole should approximate the size of the ball to a casual observer). Based on some calculations (which will be outlined in section 8.3) and a drop height of roughly 1/3rd of a meter, this led to a minimum hole width of 16mm or 17mm, depending on how optimistic we feel about mechanical tolerances. The hole width was therefore set to 18mm.

8.3 Timing Requirements

Ball Arrival Timing

From the perspective of a stationary observer, the system looks like Figure 8.3. Both the ball and the disk are in motion: the ball has some vertical velocity, and the hole moves laterally at the same time.

We can simplify our view of the system by imagining ourselves as an ant sitting on the edge of the disk. From the ant’s perspective, the hole in the disk stays still, and the ball has

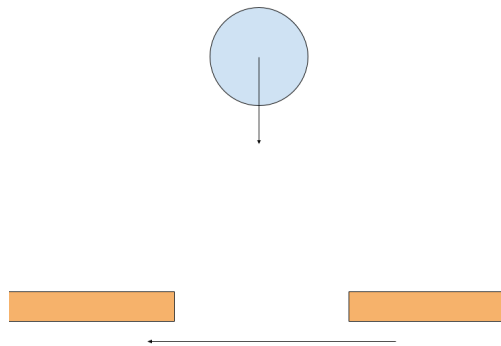


Figure 8.3: Stationary view of ball and disk

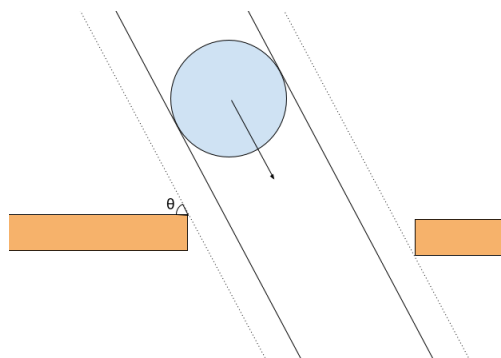


Figure 8.4: Ball and disk, from a moving perspective on the edge of the disk

both vertical and horizontal velocity. The system now looks like Figure 8.4. Figure 8.4 also shows the exact path that the ball sweeps out from the perspective of the disk. Ideally, the path should pass through the center of the hole, and leave some “wobble room” on each side. That “wobble room” tells us how much lateral error the ball can have without colliding with the disk.

In order to solve for the amount of wobble room, we need to solve for the following:

- **Vertical velocity:** The ball falls a distance of 0.36 meters through the acrylic tube, so with a gravitational constant of 9.8 and no air resistance, the time taken should be $\sqrt{2(0.36)/9.8} \approx 0.271$ seconds. With a gravitational constant of 9.8, this yields a vertical velocity of $(0.271)(9.8) \approx 2.66$ m/s at the end of the tube. However, actual measurements of the time taken (using the photogates) yield a time of 0.294 seconds, which is around 8% slower. This is likely because of air resistance in the tube, as the ball must displace most of the air in the tube as it falls. Rather than try to model the

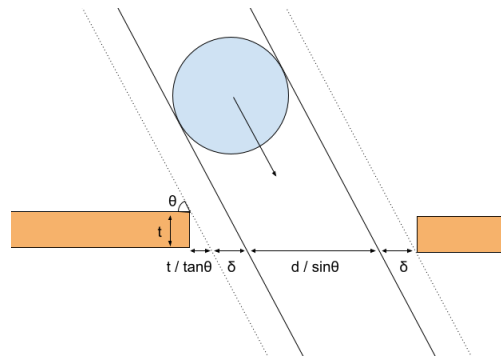


Figure 8.5: Ball and disk, with measurements. Solving for δ gives the amount of lateral error the ball can have without colliding with the disk.

fluid dynamics of the tube, we simply estimate that the ending velocity is around 8% slower, yielding an estimate of 2.45 m/s.

- **Horizontal velocity:** The center of the hole is exactly 40mm from the center of the disk. At the maximum target speed of 5 rotations per second, this leads to a horizontal velocity of $(2\pi)(0.040)(5) \approx 1.26$ m/s.
- **Angle:** The angle θ in Figures 8.4 and 8.4 is simply the arctangent of the ratio of velocities. $\tan^{-1}(2.45/1.26) \approx 1.097$ radians.

With θ solved for, we can now use Figure 8.5 to solve for δ , the amount of “wobble room”. We have the following values:

- θ , the velocity angle, is 1.097 radians.
- t , the thickness of the disk, is 3.5mm.
- d , the diameter of the ball, is 11mm.
- w , the width of the hole, is 18mm.

So we can solve

$$\frac{t}{\tan \theta} + \delta + \frac{d}{\sin \theta} + \delta = w$$

Giving us $\delta = 1.92$ mm. This must account for both mechanical tolerances and timing tolerances. There are two main sources of mechanical imprecision:

- The inner diameter of the drop tube is 12mm, which is 1mm larger than the diameter of the ball. Therefore, 0.5mm needs to be budgeted for worst-case error in the ball position.

- The initial position of the drop tube and the hole are aligned by eye. This is rather inaccurate, so 1mm needs to be budgeted for “eyeball” tolerances.

This leaves us with 0.42mm for timing tolerances¹. With a horizontal velocity of 1.26m/s, means the time that the disk arrives can be up to $0.00042/1.26 \approx 0.000335$ seconds late, or around 335 microseconds late. This magic value of 335 microseconds will be referred to as the “ball arrival precision”.

Photogate Timing

The main job of the photogates is to set the predicted arrival time of the ball. This corrects any imprecision in the drop time from the electromagnet, and imprecision caused by air resistance. Since the precision of photogate detection directly translates to the timing precision of the hole, it is necessary that the photogates operate within the ball arrival precision of 335 microseconds.

There are two possible strategies: interrupts, or polling. Since the photogates detecting a ball happens very infrequently, interrupts may seem like the more natural choice. However, a single ball detection may trigger 4-10 interrupts (all within a few microseconds of each other), due to noise on the wire. Therefore, if interrupts are used, they need to be debounced.

The simpler option is polling. Since debouncing only needs to happen in the microsecond range, if polling is used with a period of over 50 microseconds, it is very unlikely that the signal will need debouncing. Furthermore, polling has the advantage that can be set as a lower-priority task than the stepper motor, providing a guarantee that interrupt polling will never cause the stepper motor to miss its timing. In contrast, if interrupts are used, an interrupt may occur in the middle of a “step” task, causing it to miss its target timing. This issue could be resolved by using different levels of interrupts, but that adds more complexity to the system.

When polling with a period under 200 microseconds, the system can tolerate an imprecision of at most 135 microseconds in order to guarantee that a ball is always detected to within 335 microseconds of it passing through the photogate.

Stepper Motor Timing

The stepper motor used for this project has 200 steps per rotation, and the highest target speed is 5 rotations per second. However, the motor can not reach this speed using full steps; instead, the stepper motor is microstepped using 1/16th steps. This means the microprocessor must drive

$$(16 \text{ microsteps/step})(200 \text{ steps/rotation})(5 \text{ rotations/sec}) = 16000 \text{ microsteps/sec}$$

¹A proper physicist would point out that technically, 2 errors of 0.5mm and 1mm combine to create an error of $\sqrt{0.5^2 + 1^2} \approx 1.12\text{mm}$, leaving us with $\sqrt{1.92^2 - 1.12^2} = 1.56\text{mm}$ to work with for timing error. Hopefully nobody reading this is a proper physicist.

Step size	Absolute Error	Total Error
1/16th	0.049	0.0032
1/8th	0.098	0.013
1/4th	0.19	0.052
1/2	0.37	0.21
1	0.57	0.57

Table 8.1: Step force error for different stepping modes

or a period of 62.5 microseconds between each microstep. It is unknown exactly how much precision is needed. However, we can estimate the precision needed by noting that the goal of microstepping a stepper motor is to allow the force applied to approximate a sin wave. For example, 1/16 stepping approximates a sin wave with the function

$$s(t) = \sin\left(\frac{\lfloor 16t/\pi + 1/2 \rfloor}{16/\pi}\right)$$

We can estimate the absolute error between this equation and the sin wave:

$$\epsilon_a = \int_0^{\pi/2} |s(t) - \sin(t)| dt \approx 0.049$$

Of course, small variations tend to cancel out when the motor is spinning quickly. Another metric of importance² is how much total error there is over a quarter-period:

$$\epsilon_t = \left| \int_0^{\pi/2} s(t) - \sin(t) dt \right| \approx 0.0032$$

If we repeat the same calculation for lower stepping modes, we get the results in table 8.1.

One useful goal to have is to keep the steps precise enough that imprecise 1/16th stepping is better than perfect 1/8th stepping. How should we formalize this? We can imagine that in the worse case, all errors are in the same direction. Therefore, the equation of a curve where every step is δ steps late is

$$s(t) = \sin\left(\frac{\lfloor 16t/\pi + 1/2 - \delta \rfloor}{16/\pi}\right)$$

It turns out that absolute error is a very forgiving metric. All steps can be almost half a step late ($\delta = 0.48$), and we still get an absolute error less than that of 1/8th stepping. Keeping the total error low is a much more stringent requirement, since when every step is

²You might be wondering where these equations came from. I made them up. They do lead to reasonable results, so I think this is justified.

late, the errors don't cancel out. The largest δ value that keeps the total error under 0.01288 (the total error for 1/8th stepping) is $\delta = 0.049$. The smallest delta value that keeps the total error under 0.01288 is -0.082, which is less stringent. Therefore, setting a requirement that $|\delta| < 0.049$ allows us to guarantee that the total error will be better than what 1/8th stepping can achieve.

So with a period of 62.5 microseconds and a delta of at most 4.9%, we require that the step pin is always driven within 3.06 microseconds of its target time.

Button and Electromagnet Timing

A short time after a user pushes the button, the electromagnet releases the ball bearing. Ideally, this should appear instantaneous to a casual observer. A widely quoted figure for this is 100ms[10]. On average, the ball will drop after a quarter turn of the disk (since the electromagnet waits for a hole to line up before dropping), which takes 50ms at 5 rotations per second. This leaves 50ms of leeway for the detection time of the button. We could set the polling rate of the button to 50ms, but any time down to 1ms does not put much stress on the system.

The electromagnet has a similarly loose constraint. The electromagnet is responsible for timing the ball drop to line up with a hole; however, imprecision in the timing of the drop can be corrected by the first photogate. The ball passes through the first photogate 27.1ms before it passes through the disk. If we want to keep the motor speed within 5% of its target speed, this means that the system can tolerate the ball arriving at the first photogate up to 1.35ms early or 1.35ms late. There is some variance in timing due to air resistance, but from testing, that variance tends to be at most 200 microseconds. This means the electromagnet actuation can be up to 1ms early or late and still not cause the system to fail.

8.4 Software Design

A simple way to organize the software for this system is to create software components corresponding to the hardware components. If done this way, the software subsystems are:

- The stepper motor subsystem is responsible for driving the stepper motor at a target speed. At the fastest speed, this system will actuate every 62.5 microseconds.
- The photogate subsystem is responsible for polling the phototransistors to detect when the ball is at that location in the tube. This system can be polled at any rate with a period under 200 microseconds.
- The button subsystem is responsible for polling the button to detect when the system should be started. This is very similar to the photogate subsystem.

- The electromagnet subsystem is responsible for turning the electromagnet on when the system is started, and off when it is time to drop the ball. This actuates very infrequently.
- The main subsystem is responsible for coordinating the other subsystems. For example, it should keep track of the overall state of the system (idle/started/dropping), and calculate the new target speed for the stepper motor when inputs from the phototransistors change.

Of course, there is a lot of room for re-organization here, depending on the needs of the system. For example, the photogate subsystem may be split into two subsystems, one for each photogate. In the other direction, the electromagnet subsystem may be swallowed into the main subsystem, since it has very simple functionality and tends to change only when the system state changes.

Each version of the software should take into account relative priorities or deadlines. The stepper motor subsystem is the most stringent, and must actuate within a few microseconds of its target time. Every other subsystem has timing requirements on the order of 100s of microseconds.

8.5 Software Version 1: LTA

Design Overview

The LTA version of the software uses 4 actors: a “motor” actor, a “main” actor (encompassing a state machine and the electromagnet), a “button” actor, and a “sensors” actor (controlling the photogates). Figure 8.6 shows these actors and the messages between them.

One possible control scheme for these actors could be to give each of them their own control loop. For example, the motor could have an action that gets invoked every 62 microseconds, the button could have an action that gets invoked every 500 microseconds, and the sensors could have an action that gets invoked every 200 microseconds. This would lead to a very symmetrical design.

However, there is a crucial detail that makes this scheme undesirable: the motor subsystem needs to be invoked very precisely. If the system were designed with separate control loops, it would be possible for the button and sensor subsystems to have their actions invoked directly before the motor’s action. In the worst case, this could cause the motor to miss its desired timing, and start skipping steps. This would greatly disrupt the system.

Instead, we can note that the other systems have much looser timing requirements, and therefore can be tied to the motor’s control loop. This means that if the motor is being driven with a period of 62 microseconds, the button and sensors will be polled with the same period. This ensures that every action happens after the motor is driven, so unlucky timings have no chance of causing the motor to be driven late.

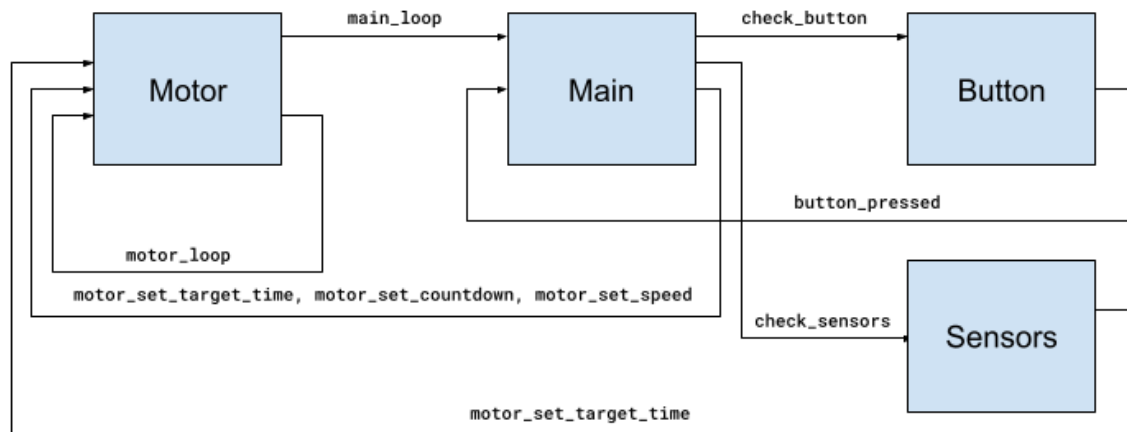


Figure 8.6: Diagram of LTA actors and messages between them.

Motor Actor

The motor actor is responsible for driving the motor, and for adjusting the speed of the motor based on its target speed or target time. Specifically, the `motor_loop` action toggle's the motor driver pin once, calculates a new speed for the motor, then schedules `main_loop` to be called immediately and `motor_loop` to be called after one period (62 microseconds at fastest, or up to 3000 microseconds at slowest). When it schedules `main_loop`, it also passes the current motor position as an argument.

Main Actor

The main actor is responsible for maintaining a finite state machine, and turning the electromagnet on or off. The state machine has the following states:

- Idle: The disk is spinning slowly, and the system is waiting for a button press to advance to the next state.
- Armed: the electromagnet is on and the disk is spinning quickly. At this point the ball bearing should be placed on the electromagnet, then the button should be pushed to advance to the next state.
- Aiming: the electromagnet is on, holding the ball bearing, and the disk is spinning quickly. The system is waiting for the disk to be in the correct position to drop the ball.

- Dropping: the electromagnet is off, and the ball is falling through the tube. The motor is adjusting its speed to make sure that a hole in the disk lines up with the tube at the same time that the ball bearing arrives at the bottom.

Each time `main_loop` is called, it checks the current state, advancing as needed (for example, if the motor position is correct, the state advances from aiming to dropping). It then schedules `check_button` or `check_sensors` (depending on the state) to run immediately.

Button Actor

The button actor is responsible for checking when the button has been pressed, and debouncing the input. When `check_button` is called, the actor checks if the button state has changed, and if the time since the last change is more than 1 second (enforcing an assumption that the button will not be pressed twice in 1 second). If so, it schedules `button_pressed` to be called immediately, which the main actor uses to update its state machine.

Sensors Actor

The sensors actor is responsible for checking the photogates, and for updating the motor's target time during the ball bearing's fall. Whenever `check_sensors` is called, the actor checks if the ball bearing has passed through one of the photogates: if so, it creates a new estimate of the time that the ball will reach the bottom of the tube, and schedules `motor_set_target_time` to notify the motor of this new estimate.

Performance

With the LTA version of the software, the system is able to reliably perform drops at full speed (5 rotations per second). The use of actors allows clean separation of functionality into modular units, and the low time overhead of LTA means that there was plenty of time to spare between action invocations. With a larger motor, it is likely that this version of the software would be able to handle higher speeds.

8.6 Software Version 2: Lingua Franca

Design Overview

Given the similarities between LTA's actors and Lingua Franca's reactors, it should be no surprise that the actors in the LTA version translate almost directly into Lingua Franca reactors. Figure 8.7 is a reactor diagram generated by Lingua Franca; it shows almost exactly the same structure as Figure 8.6.

There are a few minor differences: `Main` has been renamed to `Central`, and the output from the sensors goes through `Central` to `Motor` instead of directly to `Motor`.

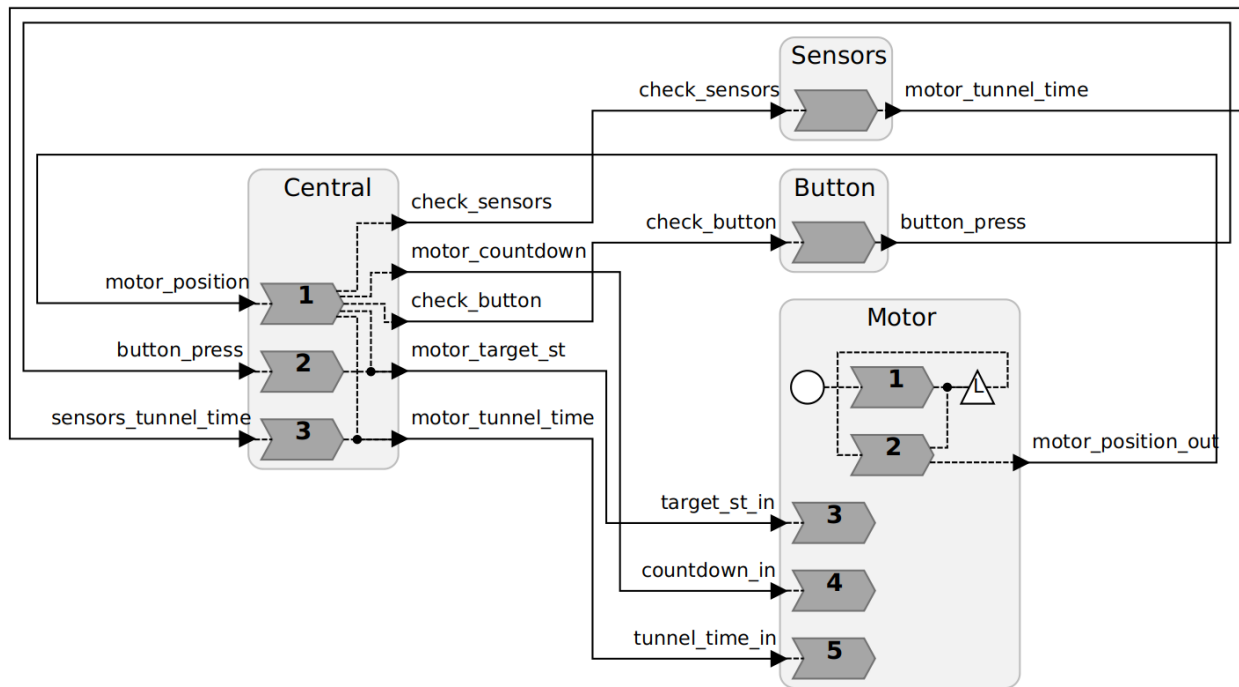


Figure 8.7: Lingua Franca reactor diagram.

Performance

Unfortunately, Lingua Franca has a high overhead per reaction. It is high enough that the system is not able to run at full speed; it can only drive the motor with a period of 200 microseconds, which is around a third of the desired top speed.

However, this is fixable. The motor subsystem is the only system with a desired period of 62 microseconds; every other subsystem can tolerate a period of 200 microseconds or more. Therefore, if an external solution is used to drive the stepper motor, Lingua Franca can be used to drive the rest of the system. This solution is outlined in the following section.

8.7 Software Version 3: Lingua Franca with Programmable IO

Design Overview

The RP2040 contains a Programmable IO system, which are essentially tiny state machines that can be used to precisely drive GPIO pins. Each state machine can take inputs from a queue, which is very useful for our use case. We can divide the task of driving the stepper

motor as follows:

- The state machine pulls integers off the queue, which represent the time to wait until toggling the stepper motor pin. It will toggle on and then off for each integer on the queue.
- The CPU calculates the desired period, and pushes it onto the queue. This can be done infrequently as long as the queue is always updated before it becomes empty. The easiest way to handle this is to simply run the motor reaction at 1/4th the frequency that the stepper motor needs to be driven at, and push 2 items at a time onto the queue (each item representing 2 toggles).

With the scheme, the motor's reaction only needs to be invoked every 240 microseconds at the fastest.

Performance

Using Lingua Franca and programmable IO, the system is able to run at full speed. It is slightly less reliable than the system with the LTA version of the software, which is due to its lower polling rate for the sensors: the LTA version polls them every 62 microseconds and this version only polls them every 240 microseconds. However, the system is still reliable enough to be considered a success.

8.8 Potential Improvements

It's worth noting that in its current form, this system is limited more by mechanical imprecision more than timing imprecision. For example, in section 8.3, more "budget" is given to mechanical tolerances than timing tolerances. This is necessary because the initial position of the disk and drop tube are aligned by eye, which is a relatively imprecise process.

With some very small (1-5mm) holes cut into edges of the disk, and an additional photogate affixed to the structure, the system could calibrate its radial position much more accurately (ideally to within 1 microstep, which equates to 79 micrometers of lateral movement). Then, the only radial inaccuracy would come from mechanical tolerances between the photogate and the drop tube. This could be minimized by laser cutting a structure to hold both parts, or using some other process with tight mechanical tolerances.

In section 8.3, a significant portion of the "budget" is taken up by the thickness of the disk. The disk is also relatively heavy, and the stepper motor is able to spin much faster than the disk. A thinner disk would solve both problems, and potentially allow the disk to spin at 5 rotations/second with a lower stepping mode (such as 1/8th or 1/4th). However, when a thinner wooden disk was used, it cracked after a failed drop. The thicker disk was able to stay intact after frequent failed drops when the system was being tested. This issue

could be resolved by using a different material, such as aluminum, but fabricating aluminum parts is more difficult than wood.

Finally, a more powerful stepper motor may be able to reach higher speeds than the one used in this device. However, higher speeds decrease the effective width of the holes in the disk. Therefore, a higher-speed system will require either a taller drop tube, or larger holes in the disk in order to work successfully. A taller drop tube increases the kinetic energy of the ball bearing, and may break things during failed attempts. On the other hand, larger holes may look less impressive to a casual observer.

Chapter 9

Conclusion and Future Work

LTA

LTA's biggest strengths are being lightweight and efficient; this allows it to outmatch the performance of FreeRTOS in almost every benchmark in this thesis. Furthermore, it retains the ergonomics needed to program a large system, as demonstrated in Chapter 8. There is also a lot of room to improve its performance; in particular, LTA may benefit from directly using the RP2040's hardware timers rather than the timer pool shared by semaphores.

Given that LTA and Lingua Franca have several common underlying concepts, work on LTA may motivate future performance improvements for Lingua Franca. In particular, the circular algorithm for the event queue might be applicable towards Lingua Franca's internal event queue.

Lingua Franca

Lingua Franca is a promising framework; it is already viable for many systems, and although it was not a top performer in this thesis's benchmarks, this is justified by its more complex model and much tighter concurrency guarantees. The performance suffers on the RP2040 partly because Lingua Franca is optimized for 64-bit systems, and several aspects of its runtime are suboptimal for microcontrollers (i.e. 64-bit time, and heavy use of atomics). More work is underway to make Lingua Franca more efficient, and there is a lot of room to improve Lingua Franca's runtime for microcontrollers. One particularly exciting area of research is static scheduling, which has the potential to dramatically improve performance on all platforms.

General Conclusions

We have demonstrated that bare-metal concurrency frameworks are viable for embedded systems, and have the potential to outmatch the performance of threading-based approaches. This thesis explored two actor-based approaches, but there is a wide design space of con-

currency models, some of which may be even more efficient than LTA. If so, we hope to see those options explored in future research.

Bibliography

- [1] URL: <https://brainwagon.org/2005/03/05/coroutines-in-c/#comment-1878> (visited on 05/01/2024).
- [2] Joe Armstrong. “Making Reliable Distributed Systems in the Presence of Software Errors”. PhD thesis. Stockholm, Sweden: Royal Institute of Technology (KTH), 2003.
- [3] Adam Dunkels. *Protothreads*. URL: <https://dunkels.com/adam/pt/> (visited on 05/01/2024).
- [4] *Embassy: The next-generation framework for embedded applications*. URL: <https://embassy.dev/> (visited on 05/01/2024).
- [5] Jeff C. Jensen. “Elements of Model-Based Design”. MA thesis. EECS Department, University of California, Berkeley, Feb. 2010. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-19.html>.
- [6] Edward Lee, Stephen Neuendorffer, and Michael Wirthlint. “Actor-Oriented Design Of Embedded Hardware And Software Systems”. In: *Journal of Circuits, Systems and Computers* 12 (Aug. 2002). DOI: 10.1142/S0218126603000751.
- [7] Edward A. Lee. *The Problem with Threads*. Tech. rep. UCB/EECS-2006-1. The published version of this paper is in IEEE Computer 39(5):33-42, May 2006. Jan. 2006. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html>.
- [8] Raspberry Pi Ltd. *RP2040 Datasheet*. Accessed: April 12, 2024. URL: <https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf>.
- [9] Zhiyao Ma and Lin Zhong. “Bringing Segmented Stacks to Embedded Systems”. In: *Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications*. HotMobile '23. , Newport Beach, California, Association for Computing Machinery, 2023, pp. 117–123. ISBN: 9798400700170. DOI: 10.1145/3572864.3580344. URL: <https://doi.org/10.1145/3572864.3580344>.
- [10] Robert B. Miller. “Response time in man-computer conversational transactions”. In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. AFIPS '68 (Fall, part I). San Francisco, California: Association for Computing Machinery, 1968, pp. 267–277. ISBN: 9781450378994. DOI: 10.1145/1476589.1476628. URL: <https://doi.org/10.1145/1476589.1476628>.

- [11] Espressif Systems. *ESP32 Series Datasheet*. Version 4.5. Accessed: April 12, 2024. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.
- [12] Simon Tatham. *Coroutines in C*. URL: <https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html> (visited on 05/01/2024).
- [13] The Rust Programming Language. *Generators are dead, long live coroutines, generators are back*. Oct. 2023. URL: <https://blog.rust-lang.org/inside-rust/2023/10/23/coroutines.html> (visited on 05/01/2024).