EmbedFX: A Domain-Specific Language with Asynchronous Communication for Embedded Systems

Danel Batyrbek
(Master's Program in Computer Science)

Advised by Yukiyoshi Kameyama

Submitted to the Graduate School
of Science and Technology
Degree Programs in Systems and
Information Engineering
in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering
at the
University of Tsukuba

March 2025

Abstract

Writing code for microcontrollers, especially for tasks and communication between them is notoriously error-prone. Often code is written in C to fit into tight constraints and to have direct control over hardware to exact optimizations. However, C language lacks crucial abstractions that represent the software that is typically written for embedded systems. For example, the task code that is run asynchronously is different from user-defined procedures that are invoked occasionally, however, both are represented using C functions.

This thesis proposes EmbedFX, a new programming language for deeply embedded microcontrollers. We advocate using asynchronous effects by Ahman and Pretnar[3] as the basic building block of our language. Asynchronous effects is a variation of traditional algebraic effects, where the handling of the effect does not block the computation that raised the effect in the first place. Additionally, it defines primitives for process creation and asynchronous communication of those processes. Having theoretical abstractions that abstract processes and communication is useful because many Real-Time Operating Systems (RTOS), such as FreeRTOS, provide API that performs that task manipulation and provide communication mechanisms, however, often come with implicit preconditions on their application which needs to be ensured by the developers. While designing such code in isolation is not impossible, the source of problems is the constant rewriting of the code that introduces bugs into the previously working code. This introduces a cognitive overhead, as developers have to ensure the correctness of the complex communication in an evolving software structure.

To solve this problem, this thesis shows the connection between the calculus of asynchronous effects and FreeRTOS API, namely task creation and communication, under restrictions. While the absence of dynamic memory allocation puts limits on the expressivity of the resulting calculus, the feasibility of a limited asynchronous effects calculus in such an environment has been established. While asynchronous effects can be used to describe some of the programs in deeply embedded software, they are not yet suitable to represent programs that require real-time reaction. As part of a contribution, the memory footprint and CPU load of EmbedFX have been measured, as well as well as various simple, but not trivial examples of software that have been executed and tested on ESP32.

Contents

List of Figures

Chapter 1

Introduction

Embedded software is a key milestone in the engineering progress, bringing intelligence to everyday devices, yet it remains one of the most challenging aspects of engineering due to technical and commercial environments [24].

Programming for embedded software is difficult for a variety of reasons, some of which are inherent to software engineering, while others are unique challenges of the embedded software context. While there are many ongoing attempts to improve the state of development tools, bringing many state-of-the-art techniques and technologies are often impossible due to prohibitive costs and limitations, especially in the context of *deeply embedded systems* [2]. Deeply embedded systems in this context mean the microcontrollers that are embedded in some machinery or other system with the sole purpose of performing a specific task [8] [15]. Those devices are usually low-end microcontrollers, having limited memory of several KB or MB and a processor of a couple of MHz, from 32 to a couple hundred, to save cost [2]. As such, there is still a search for lightweight and accessible abstractions that improve the quality of the embedded software.

To take advantage of the constrained resources and have precise control over the underlying hardware, languages such as C and C++ are often used for embedded software programming. The problem is that C and C++ do not provide abstractions that represent the properties of the embedded software, which means that the correctness of the written programs needs to be ensured by the developers themselves. The challenge is further complicated by the fact that embedded software is often very complicated, many parts interacting with each other in unpredictable ways because of implicitly shared mutable state and debugging failures in such software are usually complicated by the limited visibility into the hardware.

This work attempts to improve the quality of the embedded software with asynchronous effects, a variation of traditional synchronous effect handlers, introduced by Ahman and Pretnar[3]. Asynchronous effects represent effectful computations that interact asynchronously via explicit communication primitives. Their contribution is ensuring the type safety of asynchronous computations. This thesis proposes EmbedFX, a programming language based on the asynchronous effects calculus by Ahman and Pretnar, and targets deeply embedded microcontrollers by compiling code to C with FreeRTOS.

The benefit of EmbedFX is that it defines asynchronous computation according to the original asynchronous effects calculus, which is very similar to how asynchronous tasks are defined in many RTOSes. Additionally, many RTOS provide primitives to establish communication and coordination between the said tasks [23]. Asynchronous effects provide communication primitives as well, which can be used to represent and abstract the communication that occurs on the task level. As a result, EmbedFX helps to ensure the safety of the communication of the embedded software and type safety of the tasks even in complex scenarios. This should help developers focus on designing the software without worrying about implementing and ensuring the safety of task creation and communication themselves.

This thesis from now on will be structured as follows:

- Chapter 2 gives the background of embedded systems and RTOS in general.
- Chapter 3 goes into detail about asynchronous effects as they are defined by Ahman and Pretnar and FreeRTOS API.
- Chapter 4 describes the contribution of this thesis – EmbedFX, the decisions behind the design of the language, and the connection between EmbedFX and the original asynchronous effects calculus.
- Chapter 5 goes into details of implementing EmbedFX and transpiling asynchronous effects to C with FreeRTOS.
- Chapter 6 gives an evaluation of EmbedFX based on the examples that have been executed on the microcontroller ESP32. The heap memory consumption and CPU load by EmbedFX were measured.
- Chapter 7 discusses the problems encountered when designing EmbedFX and highlights future work that needs to be done to improve the state of asynchronous effects on deeply embedded software.
- Chapter 8 concludes the thesis.

Chapter 2

Background

This chapter introduces the background and motivation necessary to understand the goals and purpose of this thesis.

2.1 Developing Embedded Software

The current thesis aims to contribute to the quality of the embedded software by borrowing ideas from recent programming language research and apply them in the context of embedded software. This is because many of the embedded software developers are traditionally electrical engineering background [20]. Practitioners are expected to have a good understanding of the fundamentals of electric circuits, the electricity, as well as good programming skills.

Additionally, not every embedded system is built equal. Sometimes to solve a problem, no coding may be required: by just properly connecting the hardware or ordering the logic gates and connectors is enough.

On the other hand, the system may be extremely complex and specific to require a custom hardware. This means that there are no prior software libraries and toolchains around such hardware and needs to be developed from scratch as well. Secondly, while the hardware is being designed, the means to test the software are limited. Thirdly, when the problem arises, it is not clear where the problem may be: in the software or hardware? Lastly, the hardware requirements and specifications may change, which is inadvertently reflected on the software and incurs additional development costs.

While the challenges associated with such endeavors are high, there are benefits of having tightly coupled software and hardware, which is sometimes a necessity. It happens when developing electronic components for automotive systems [5].

The design of the project is constrained by different factors, such as:

- safety and design requirements
- available resources: developers, money, time, hardware, etc
- limitations of tools: problems associated with the programming language, building tools, etc.
- risks: used libraries, support, security, etc.

- laws: the written software is subject to strict safety and standards certification and other applicable laws

Trying to balance the said factors while developing the software is difficult. As such, different approaches have been undertaken to resolve them.

This thesis focuses primarily on programming languages for embedded systems.

2.2 Programming Languages

As have been mentioned in the Chapter 1, the ecosystem for embedded systems is lagging behind the traditional general purpose programming. Despite that, there have been several attempts at developing programming languages for that. The following section will quickly go over the programming languages used and/or developed for embedded systems.

2.2.1 C and C++

C and C++ are the de facto programming languages when it comes to embedded systems. There are several reasons which contribute to their widespread use in the industry:

1. low-level languages: C and C++ provide almost direct, low-level access to the hardware. This can be important when having to communicate directly with the devices and sensors (e.g. writing device drivers) or trying to optimize the code.

2. simple semantics: C is arguably simple to understand language with defined semantics. While some of the aspects of the language remain undefined, the said semantics are simpler than that of other mainstream languages, such as Java or JavaScript.

3. popularity: C and C++ are among top used programming languages for embedded systems [13]. While the reasons for popularity are outside of the scope of this thesis, being a popular language it is arguably easier to find employees with proficiency in C programming.

4. rich ecosystem and legacy: C and C++ have been used for a long time, with many systems still being developed and actively supported in C (e.g. Linux). As a result, the accumulated rich ecosystem reduces the costs of developing in C and C++. Sometimes this is a necessity, because of the need to write *predictable* software (e.g. rocketry).

While the list is not exhaustive, it provides the main reasons for the dominance of C and C++ in the branch of embedded systems.

Of course, the overly expressive and low-level nature of C is credited for the vast majority of the security vulnerabilities [1]. Stack and buffer overflow problems are common offenders, with manual memory allocation coming after , such as memory leaks and double-free vulnerabilities. Mechanisms to prevent the easy exploitation of the security vulnerabilities exist, but often expensive or outright dangerous, such as Address-Space Layout Randomization (ASLR), stack canaries, etc. As such, attempts have been made to displace the C and C++.

2.2.2 Rust

Rust is a relatively new mainstream systems programming language. The main value proposition of Rust is that it is a *memory safe* programming language, which means that it is difficult (and often impossible) to violate spatial constraints. This is achieved through the *borrow checker*, the verification mechanism which ensures that each acquired resource has a single ownership and that there are no violation of the property throughout the code. The Rust development team has a special focus on embedded systems and has shown that Rust is a viable option for developing safe embedded software [19]. Rust has been also tested by other teams for suitability for writing embedded software [11].

However, the are still problems with the application of Rust in real-life settings. It is noted that the current Rust ecosystem is not mature enough to be used seriously, incompatible with existing code-bases due to specific Rust constructs, and borrow checker is too restrictive, preventing from writing effective low-level software [10] [22].

In the future, the situation is projected to improve, but for now, there is a clear need for a language with strong abstractions, but that is not difficult to use.

2.2.3 Arduino

Arduino is an umbrella term which encompasses different things depending on the context, namely:

- Hardware platform
- Language
- IDE
- Ecosystem

In the context of the current discussion, Arduino refers mainly to the language.

Arduino was developed primarily as a learning tool, to teach the basics of microcontroller programming for students.

Arduino has many aspects of low-level hardware abstracted away with specific library methods. The simplicity and ease of development are why Arduino sometimes used as a prototyping tool, when testing out ideas or new devices and sensors.

Arduino mainly abstracts over a *superloop* abstraction. In superloop design, the program is mainly a single `main` function in which the code is repeatedly executed:

```
function main() {
  while(true){
    f_1();
    f_2();
    ...
    f_n();
  }
}
```

The benefit of such a design is that it is very simple and makes analysis of the problems simpler. The code written in Arduino is structured around this abstraction.

There are two main built-in methods: `setup()` and `loop()`.

Code written in `setup()` is run once, during the start-up of the system. It is primarily used as a *setup* for the consequent code, however can be used instead of traditional `main()`.

Code written in `loop()` is repeated in an infinite loop.

The problem with such an approach is that the functions are heavily *sequential*, which means that `f_1()` always comes before `f_2()`, `f_2()` always comes before `f_3()` and so on. This is important in the case of **jittering** [4]. If the previous run is slower or faster than before, it affects the timings of the subsequent code.

In cases when the superloop design is not enough, Real-Time Operating System (RTOS) is used.

Arduino is a great introduction to the microcontrollers and programming.

2.3 Algebraic Effect Handlers

Another approach on which this thesis is built is the notion of the algebraic effect handlers. Algebraic Effects and Effect Handlers is an approach to structuring and representing effectful programs [18]. The effects are meant to represent the effects that occur during the computation and the handlers provide the interpretation of those effects:

```
with h handle
  {
    print "A";
    print "B";
    print "C";
  }
where h = handler {
  | print (s; k) -> sys_print s; k ()
  }
```

This is a small example of how a `print` effect can be implemented and used in the context of effect handlers. The main computation `print "A"; print "B"; print "C";` does not perform printing by itself, but rather invokes the *effect* `print` which is then evaluated by the handler. The important point here is that the **interpretation** is provided by the outside environment. With this separation of the main logic (the *intent* of printing) from its implementation (the actual *act* of printing). This is useful, as it allows us to provide a clear interpretation of the effects and make modular changes without affecting the computation.

In this example, several parts require attention:

- Main computation: In the main computation, we invoke the effects (in this case `print`)
- Effect handler: The handler provides the interpretation of different effects
- Continuation: along with the effect `print` and its argument `s`, there is another argument: continuation, in this case `k`.

Continuation represents the rest of the computation from the point where the effect was raised. The addition of continuation is what enables the effect handlers with the expressivity.

For example, one could easily change the handlers to perform the printing in the reverse order:

```
with h handle
  {
    print "A";
    print "B";
    print "C";
  }
where h = handler {
  | print (s; k) -> k (); sys_print s
  }
```

If we want to collect the printed strings into one big string, we just have to provide different interpretation of the `print`.

```
with h handle
  {
    print "A";
    print "B";
    print "C";
  }
where h = handler {
  | return x -> return (x, "")
  | print (s; k) ->
    do (x, acc) = k () in
    return (x, join s acc)
  }
```

Effect handlers are a rich and highly expressive abstraction that allows us to encode even non-deterministic behavior:

```
with h handle
  {
    let x = choice () in
    return (f x)
  }
where h = handler {
  | return x -> return x
  | choice ((), k) ->
    let a = k true in
    let b = k false in
    max a b
}
```

This thesis focuses on a variation of algebraic effect handlers, called asynchronous effects [3].

2.4 FreeRTOS

A Real-Time Operating System (RTOS) is a specialized type of operating system designed to ensure that tasks are executed within specific timing constraints, a critical requirement in embedded systems and time-sensitive applications. Unlike General-Purpose Operating Systems (GPOS), which aim to optimize throughput and responsiveness, an RTOS prioritizes predictability and timing precision [14]. RTOSes shine in contexts when it is important to modularize the software or where simple superloop approach is not sufficient, however, implementing a task scheduling, scheduling algorithms and primitives are difficult and time-consuming.

RTOS is different from general-purpose operating systems such as μLinux as it provides minimal abstraction over underlying hardware and where the additional abstractions, such as filesystem, need to be added explicitly. While general purpose operating system provides complete abstraction over hardware and low-level details and security mechanisms, especially in case of program crash and recovery, the technical details of RTOS need to be taken into consideration. Additionally, RTOS intends to be as simple and efficient as possible, to boot up in milliseconds, whereas a μLinux would take several seconds to boot. However, due to this minimalism, close attention to detail is required when using RTOS for deeply embedded systems.

Out of all the available RTOS, the current thesis focuses on FreeRTOS, a popular choice for RTOS when developing embedded software. Several reasons make FreeRTOS an appealing choice for development [4]:

- Open Source: FreeRTOS is licensed under the MIT License, which allows developers to easily use the software without licensing costs and leaves an option to modify it for the custom needs
- Lightweight Design: FreeRTOS has an efficient design that allows it to run on constrained systems
- Portability: FreeRTOS is supported by a wide array of microcontroller families
- Task Management: FreeRTOS provides mechanisms for task creation and communication among those tasks, as well as a preemptive and cooperative scheduling

There are many benefits to using FreeRTOS, moreover, due to its Open Source nature and widespread use, along with the rich available supporting materials, the FreeRTOS was chosen as a focus for this research.

Chapter 3

Existing Work

This thesis seeks to improve the state of embedded systems programming using the recent advancements in programming language research and combine it with existing solutions. One such advancement is the notion of Asynchronous Effects.

3.1 Asynchronous Effects

Asynchronous Effects [3] is the variation of Algebraic Effects and Effect Handlers. The notion of Algebraic Effects and Effect Handlers is explained in the Background chapter. This section will review the asynchronous effects and their types as they are defined by Ahman and Pretnar. In the traditional effect handlers, when the computation raises an effect, essentially requests an effect to be evaluated with specific parameters, the effect is propagated upwards, until it is captured by the nearest suitable handler:

```
with h handle
  { let x = op V in M }
where h = handler {
  | return x -> return x
  | op (x, k) -> let res = M_op in k res
}
```

The evaluated result is usually passed back to the call site and the rest of the computation proceeds.

The main idea is that in the case of traditional effect handlers, the main computation that raised the effect is blocked until the effectful computation is resolved.

$$\begin{tikzcd}
& M_{\text{op}}[V/x] \arrow[r, squiggly, "*"] & \text{return } W \arrow[d] & \\
\ldots \arrow[r, squiggly] & \text{op}(V, y.M) \arrow[u] & M[W/y] \arrow[r, squiggly] & \ldots
\end{tikzcd}$$

Figure 3.1: Flow of a synchronous algebraic effect handler

In asynchronous effects, there is a decoupling of the main computation from the effectful computation of the effects:

$$\begin{array}{ccccccc} & & \uparrow & & \downarrow & & \\ \ldots & \rightsquigarrow & \uparrow \mathrm{op}\ (x, M) & \rightsquigarrow^{*} & \downarrow \mathrm{op}'\ (W, M') & \rightsquigarrow & \ldots \end{array}$$

Figure 3.2: Flow of an asynchronous algebraic effect handler

So from the moment when the effect was raised ($\uparrow \mathrm{op}\ (V, M)$) to the moment when incoming interrupt reached back ($\downarrow \mathrm{op}'\ (W, M')$), the main computation M continued its evaluation. The diagram has also changed its shape. In the traditional effect handlers, there is a handler defined in the outer scope of the current computation, and thus the current computation is bound by that handler. In the case of asynchronous effects, however, there is no dedicated handler that binds the current computation. As such, the moment effect is raised, it may "trigger" different "handlers" (in this calculus *promises*). The diagram does not assume the nature or existence of such promises.

The small change in the algebraic effects calculus, such as decoupling the main computation from the effectful computation resulted in significant change and re-purposing of the calculus. If in the traditional effect handlers effects can be viewed as a "request" to perform some effectful computation, then in the asynchronous effects calculus, the effects have the purpose of acting as signals between different computations.

The original work by Ahman and Pretnar[3] focuses on defining the core calculus of asynchronous effects and proving the type safety of the computations in the asynchronous context. This work focuses on adapting the said core calculus, albeit with some constraints, for the deeply embedded microcontrollers, to provide the missing abstractions in C.

The following sections will describe the asynchronous effects calculus which is defined by Ahman and Pretnarand is used by the EmbedFX.

Figure 3.3 describes the syntax of asynchronous effects calculus [3]. The asynchronous effects calculus is based on lambda calculus and extended with primitives for asynchronous computations and communications. In addition to the standard values there is a notion of "fulfilled promise". It is essentially the `Promise` or `Future` constructs currently found in many mainstream programming languages (such as JavaScript): instead of a value, there is a promise of the value, which becomes available using specific synchronization constructs, such as `await`.

Some similarities with the lambda calculus are let-bindings and function application. One of the additions is the let-rec definitions, which is used to define recursive functions in lambda calculus [1]. It is important to note that the recursions themselves are disallowed within EmbedFX.

[1]The reason for the separation into let and let-rec is in the name bindings. If we are binding a variable x, for example, in the definition of the said x we may introduce another let-binding which again re-defines the said x. However, outer-scope x and inner-scope x are different due to lexical scopes: inner-scope x is not used outside of the definition of the outer scope x.

Functions are also defined as a let-binding over an anonymous function, so when we try to recursively refer to the function inside the function definition, the name is not defined due to lexical scoping. As such, let-rec introduces the name into the inner scopes whilst it is being defined.

Values

V	$::=$	x	variables
	$\mid$	$()$	unit
	$\mid$	$\mathrm{fun}(x\ :\ X) \mapsto M$	anonymous function definition
	$\mid$	$\langle V \rangle$	fulfilled promise

Computations

M, N	$::=$	$\mathrm{return}\ V$	return
	$\mid$	$\mathrm{let}\ x = M\ \mathrm{in}\ N$	let-bindings
	$\mid$	$\mathrm{let\,rec}\ f\ x\ =\ M\ \mathrm{in}\ N$	let-rec bindings
	$\mid$	VW	funciton application
	$\mid$	$\uparrow \mathrm{op}\ (V, M)$	signal
	$\mid$	$\downarrow \mathrm{op}\ (V, M)$	interrupt
	$\mid$	$\mathrm{promise}\ (\mathrm{op}\ x \mapsto M)\ \mathrm{as}\ p\ \mathrm{in}\ N$	promise definition
	$\mid$	$\mathrm{await}\ \langle V \rangle\ \mathrm{until}\ \langle x \rangle\ \mathrm{in}\ M$	await

Figure 3.3: Values and computations

Other than that, the lambda calculus is also extended with the asynchronous effects primitives, namely:

- Signals (outgoing effects)
- Interrupts (incoming effects)
- Promises (effect handlers)
- Await (synchronization construct)

It can be noted that effects are split into two parts: signals and interrupts. In original calculus [3], signals refer to outgoing effects that try to propagate outwards, whereas interrupts represent incoming signals that may reach some ongoing computations. One of the key insights is that signals and interrupts are “two sides of the same coin”, i.e. the signal raised from one of the computations is an interrupt when observed from another computation. The effect of such duality can be illustrated with this example, taken from the original paper [3]:

$$\begin{aligned}
& \mathrm{run}\ (\uparrow \mathrm{request}\ (V, M_{\mathrm{feedClient}}))\ \|\ \mathrm{run}\ (M_{\mathrm{feedServer}}) \\
\rightsquigarrow\ & \uparrow \mathrm{request}\ (V, \mathrm{run}\ (M_{\mathrm{feedClient}}))\ \|\ \mathrm{run}\ (M_{\mathrm{feedServer}}) \\
\rightsquigarrow\ & \uparrow \mathrm{request}\ (V, \mathrm{run}\ (M_{\mathrm{feedClient}})\ \|\ \downarrow \mathrm{request}\ (V, \mathrm{run}\ (M_{\mathrm{feedServer}}))) \\
\rightsquigarrow\ & \uparrow \mathrm{request}\ (V, \mathrm{run}\ (M_{\mathrm{feedClient}})\ \|\ \mathrm{run}\ (\downarrow \mathrm{request}\ (V, M_{\mathrm{feedServer}})))
\end{aligned}$$

Figure 3.4: Duality of signals and interrupts

In this example, we have two imaginary asynchronous computations or processes, $M_{\mathrm{feedClient}}$ and $M_{\mathrm{feedServer}}$, running in parallel (via $\|$ operator). The operational semantics of the calculus are provided later (Figure 3.5 and Figure 3.11). For this example, the $M_{\text{feedClient}}$ raises an effect, which is a signal. As can be seen from the example, in the context of two running asynchronous computations, the signal raised from the first computation of $M_{\text{feedClient}}$ becomes an interrupt which propagates into the second computation $M_{\text{feedServer}}$. It is also noted that the original raised signal propagates outwards, thus the two computations are within the $\uparrow$ request.

Additional elements that are provided are *promises* and *await*.

Promise is an alternative to an effect handler in algebraic effect handlers. Just like effect handlers, promises represent a code that is executed when the promise is *triggered*. A promise is triggered when an effect (namely *interrupt*) of matching type reaches the promise.

$$\downarrow \text{op}\ (V, \text{promise}\ (\text{op}\ x \mapsto M)\ \text{as}\ p\ \text{in}\ N) \rightsquigarrow \text{let}\ p = M[V/x]\ \text{in}\ \ \downarrow \text{op}\ (V, N)$$

When the types do not match, the promise rests untriggered, while the interrupt continues propagation further into the computation.

$$\downarrow \text{op}'\ (V, \text{promise}\ (\text{op}\ x \mapsto M)\ \text{as}\ p\ \text{in}\ N) \rightsquigarrow \text{promise}\ (\text{op}\ x \mapsto M)\ \text{as}\ p\ \text{in}\ \ \downarrow \text{op}'\ (V, N)$$

The difference between promises and effect handlers is that a traditional handler can handle several effects whereas a promise can be triggered only by an effect of a single type. Of course, the handlers that handle several effects can be modeled as nested handlers that handle an effect of a single type.

However, a more significant difference is that handlers are triggered by the effects that are raised from "inside" of the handler, in lexical scope terms. Promises, on the other hand, can be triggered only by effects that were raised "outside" of the promise, again, in lexical scope terms. Promises nested within the other promises do not get triggered unless the enclosing promise gets triggered, which is the reason why promises cannot be nested to handle different effects, whereas traditional effect handlers can.

Lastly, promises represent a piece of code that is triggered and runs *asynchronously* to the main computation in which it is defined. What this means, is that the main computation continues the execution without waiting for the promise to trigger or finish the computation.

$$N \rightsquigarrow N' \quad \text{implies} \quad \text{promise}\ (\text{op}\ x \mapsto M)\ \text{as}\ p\ \text{in}\ N \rightsquigarrow \text{promise}\ (\text{op}\ x \mapsto M)\ \text{as}\ p\ \text{in}\ N'$$

Await is a synchronization mechanism that is used when the computation needs to be blocked. Await accepts a promise and blocks the computation until the promise is resolved, the result of which is then bound by a local variable which can be used in subsequent computations.

$$\downarrow \text{op}\ (V, \text{promise}\ (\text{op}\ x \mapsto M)\ \text{as}\ p\ \text{in await}\ \langle p \rangle\ \text{until}\ \langle x \rangle\ \text{in}\ N) \rightsquigarrow \text{let}\ x = M[V/x]\ \text{in}\ N$$

Those building blocks are the basis for the communication in asynchronous effects calculus. The Figure 3.5 describes the operational semantics of asynchronous effects as defined by Ahman and Pretnar[3].

Standard computation rules

$$\begin{aligned}
(\mathrm{fun}(x\ :\ X) \mapsto M)V &\leadsto M[V/x]\\
\mathrm{let}\ x = (\mathrm{return}\ V)\ \mathrm{in}\ N &\leadsto N[V/x]\\
\mathrm{let\,rec}\ f\ x\ =\ M\ \mathrm{in}\ N &\leadsto N[\mathrm{fun}(x\ :\ X) \mapsto \mathrm{let\,rec}\ f\ x\ =\ M\ \mathrm{in}\ (f\ x)/f]
\end{aligned}$$

Signal and promises propagation

$$\begin{aligned}
\mathrm{let}\ x = \uparrow \mathrm{op}\ (V, M)\ \mathrm{in}\ N &\leadsto \uparrow \mathrm{op}\ (V, \mathrm{let}\ x = M\ \mathrm{in}\ N)\\
\mathrm{let}\ x = (\mathrm{promise}\ (\mathrm{op}\ y \mapsto M)\ \mathrm{as}\ p\ \mathrm{in}\ N_1)\ \mathrm{in}\ N_2 &\leadsto \mathrm{promise}\ (\mathrm{op}\ y \mapsto M)\ \mathrm{as}\ p\ \mathrm{in}\ \mathrm{let}\ x = N_1\ \mathrm{in}\ N_2
\end{aligned}$$

Commutativity of signals with promises

$$\mathrm{promise}\ (\mathrm{op}\ x \mapsto M)\ \mathrm{as}\ p\ \mathrm{in}\ \uparrow \mathrm{op}'\ (V, N) \leadsto \uparrow \mathrm{op}'\ (V, \mathrm{promise}\ (\mathrm{op}\ x \mapsto M)\ \mathrm{as}\ p\ \mathrm{in}\ N)$$

Interrupt propagation

$$\begin{aligned}
\downarrow \mathrm{op}\ (V, \mathrm{return}\ W) &\leadsto \mathrm{return}\ W\\
\downarrow \mathrm{op}\ (V, \uparrow \mathrm{op}'\ (W, M)) &\leadsto \uparrow \mathrm{op}'\ (W, \downarrow \mathrm{op}\ (V, M))\\
\downarrow \mathrm{op}\ (V, \mathrm{promise}\ (\mathrm{op}\ x \mapsto M)\ \mathrm{as}\ p\ \mathrm{in}\ N) &\leadsto \mathrm{let}\ p = M[V/x]\ \mathrm{in}\ \downarrow \mathrm{op}\ (V, N)\\
\downarrow \mathrm{op}'\ (V, \mathrm{promise}\ (\mathrm{op}\ x \mapsto M)\ \mathrm{as}\ p\ \mathrm{in}\ N) &\leadsto \mathrm{promise}\ (\mathrm{op}\ x \mapsto M)\ \mathrm{as}\ p\ \mathrm{in}\ \downarrow \mathrm{op}'\ (V, N)\ (\mathrm{op} \neq \mathrm{op}')
\end{aligned}$$

Awaiting promises to be fulfilled

$$\mathrm{await}\ \langle V \rangle\ \mathrm{until}\ \langle x \rangle\ \mathrm{in}\ M \leadsto M[V/x]$$

Figure 3.5: Small-step operational semantics

Standard computational rules come from the traditional lambda calculus which describes the effects of function application and recursive definition.

Signal propagation shows that signals always try to propagate outwards in terms of lexical scope. This is important for the signal broadcast aspect of operational semantics. Promise propagation shows that promises also propagate outwards. What this rule doesn't explain is what would be the result of such propagation in a case of this:

```
if $e$ then
        promise (op1 $x \mapsto M_1$) as $p1$ in $p1$
else
        promise (op2 $x \mapsto M_2$) as $p2$ in $p2$
```

According to Ahman and Pretnar[3], this can be reduced as either case:

promise (op1 $x \mapsto M_1$) as $p1$ in
promise (op2 $x \mapsto M_2$) as $p2$ in
if e then $p1$ else $p2$

promise (op2 $x \mapsto M_2$) as $p1$ in
promise (op1 $x \mapsto M_1$) as $p2$ in
if e then $p1$ else $p2$

However, for the practical purposes, either reduction is fine. According to the rules of interrupt propagation, whether the interrupt triggers the promise or not, it propagates downwards into the computation. If the promises have the same effect type that they trigger upon, both promises would trigger. Otherwise, only those promises that have a matching effect type will do. As such, either case is consistent with the intuition of asynchronous effects.

The commutativity of signals and promises showcases the interaction between signals (outgoing effects) and promises. As has been stated before (and is described in the "Interrupt propagation" section of small-step operational semantics), promises are triggered only upon the interrupts, i.e. incoming effects. As such, signals continue to propagate outwards without triggering or interacting with the promises.

Section "Interrupt propagation" explains the behavior of interrupts in different scenarios. The important ones are:

- Interrupts stop if there is nowhere to propagate.
- Interrupts and signals don't interact with each other. As such, signals will continue to propagate outwards, while interrupts inwards.
- The standard rules of triggering promises.
- The order of interrupts is not consistent.

Figure 3.6 and Figure 3.8 describe the type system and typing rules of asynchronous effects calculus as defined by Ahman and Pretnar[3].

$$\begin{array}{rcl}
\text{Ground type} A, B & ::= & b \mid 1 \\
\text{Signal or interrupt signature:} & & \mathrm{op} : A_{\mathrm{op}} \\
\text{Outgoing signal annotations:} & & o \in O \\
\text{Interrupt handler annotations:} & & \iota \in I \\
\text{Value type:} & ::= & A \mid X \mapsto Y!(o,\iota) \mid \langle X \rangle \\
\text{Computation type:} & & X!(o,\iota)
\end{array}$$

Figure 3.6: Value and computation types

$$\frac{}{\Gamma, x : X, \Gamma' \vdash x : X} \qquad \frac{}{\Gamma \vdash () : 1} \qquad \frac{\Gamma \vdash V : X}{\Gamma \vdash \langle V \rangle : \langle X \rangle} \qquad \frac{\Gamma, x : X \vdash M : Y!(o,\iota)}{\Gamma \vdash \mathrm{fun}(x : X) \mapsto M : X \mapsto Y!(o,\iota)}$$

Figure 3.7: Value typing rules

$$\frac{\Gamma \vdash V : X}{\Gamma \vdash \text{return } V : X!(o, \iota)}$$

$$\frac{\Gamma \vdash M : X!(o, \iota) \qquad \Gamma, x : X \vdash N : Y!(o, \iota)}{\Gamma \vdash \text{let } x = M \text{ in } N : Y!(o, \iota)}$$

$$\frac{\Gamma, f : X \mapsto Y!(o, \iota), x : X \vdash M : Y!(o, \iota) \qquad \Gamma, f : X \mapsto Y!(o, \iota) \vdash N : Z!(o', \iota')}{\Gamma \vdash \text{let rec } f\ x\ =\ M \text{ in } N : Z!(o', \iota')}$$

$$\frac{\Gamma \vdash V : X \mapsto Y!(o, \iota)\ \Gamma \vdash W : X}{\Gamma \vdash VW : Y!(o, \iota)}$$

$$\frac{\text{op} \in o \qquad \Gamma \vdash V : A_{\text{op}} \qquad \Gamma \vdash M : X!(o, \iota)}{\Gamma \vdash \uparrow \text{op } (V, M) : X!(o, \iota)}$$

$$\frac{\Gamma \vdash V : A_{\text{op}} \qquad \Gamma \vdash M : X!(o, \iota)}{\Gamma \vdash \downarrow \text{op } (V, M) : X!\text{op} \downarrow (o, \iota)}$$

$$\frac{\iota(\text{op}) = (o', \iota') \qquad \Gamma, x : A_{\text{op}} \vdash M : \langle X \rangle!(o', \iota') \qquad \Gamma, p : \langle X \rangle \vdash N : Y!(o, \iota)}{\Gamma \vdash \text{promise } (\text{op } x \mapsto M) \text{ as } p \text{ in } N : Y!\text{op} \downarrow (o, \iota)}$$

Figure 3.8: Asynchronous effects typing rules

As can be seen from the typing rules of the computations, the information about the outgoing effects and installed interrupts is associated with each computation. What this means is that for each computation we know the effects that could be raised, as well as the interrupts that can be triggered by some effects. The action of triggered effects is defined by "action of interrupts" [3]: op $\downarrow (o, \iota)$.

$$\text{op} \downarrow (o, \iota) \stackrel{\text{def}}{=} \begin{cases} (o \cup o', \iota[\text{op} \mapsto \bot] \cup \iota') & \text{if } \iota(\text{op}) = (o', \iota') \\ (o, \iota) & \text{otherwise} \end{cases}$$

Figure 3.9: Definition of action of interrupts

This is the description of the action of interrupt on a type level: if the op matches the effect that is listened by the interrupts ($\iota(\text{op}) = (o', \iota')$), then we remove interrupts that have been triggered ($\iota[\text{op} \mapsto \bot]$), and thus they are not triggered anymore, and in their stead, newly installed interrupts (ι') are being associated with the computation. For that reason we take the union of two sets: $\iota[\text{op} \mapsto \bot] \cup \iota'$. The raised effects in both cases are added: $o \cup o'$. If, on the other hand, the interrupt is not triggered by the effect op, then the computation remains unchanged: (o, ι).

The sections before have described the signals, interrupts, and interrupt handlers. However, another core part that is required for the definition is the notion of asynchronous computations and their interaction with signals and interrupts.

The asynchronous part of the asynchronous effects calculus is defined as follows:

$$P, Q \quad ::= \quad \text{run}\,(M) \;\mid\; P \mid\mid Q \;\mid\; \uparrow \text{op}\,(V, P) \;\mid\; \downarrow \text{op}\,(V, P)$$

Figure 3.10: Syntax of parallel processes

Individual computations

$$\frac{M \rightsquigarrow N}{\text{run}\,(M) \rightsquigarrow \text{run}\,(N)}$$

Signal hoisting

$$\text{run}\,(\uparrow \text{op}\,(V, P)) \rightsquigarrow\ \uparrow \text{op}\,(V, \text{run}\,(P))$$

Interrupt propagation

$$\begin{aligned}
\downarrow \text{op}\,(V, \text{run}\,(M)) &\rightsquigarrow \text{run}\,(\downarrow \text{op}\,(V, M)) \\
\downarrow \text{op}\,(V, \text{run}\,(N) \mid\mid \text{run}\,(M)) &\rightsquigarrow\ \downarrow \text{op}\,(V, \text{run}\,(N)) \mid\mid\ \downarrow \text{op}\,(V, \text{run}\,(M)) \\
\downarrow \text{op}\,(V, \uparrow \text{op}'\,(W, P)) &\rightsquigarrow\ \uparrow \text{op}'\,(W, \downarrow \text{op}\,(V, P))
\end{aligned}$$

Broadcasting

$$\begin{aligned}
\uparrow \text{op}\,(V, P) \mid\mid Q &\rightsquigarrow\ \uparrow \text{op}\,(V, P \mid\mid\ \downarrow \text{op}\,(V, Q)) \\
P \mid\mid\ \uparrow \text{op}\,(V, Q) &\rightsquigarrow\ \uparrow \text{op}\,(V, \downarrow \text{op}\,(V, P) \mid\mid Q)
\end{aligned}$$

Figure 3.11: Small-step operational semantics of parallel processes

The small-step operational semantics of the parallel processes (Figure 3.11) describe how asynchronous parts interact with the signals and interrupts in the context of asynchronous computations [3].

The "run" primitive captures a single computation that is performed asynchronously, with respect to other computations.

The *signal hoisting* part of operational semantics shows that the signals continue to propagate outwards, even outside of the context of asynchronous computations.

Interrupt propagation shows how interrupts interact with asynchronous computations. Contrary to signals, interrupts try to propagate *inwards*, into the computation. The interesting rule here is how to interrupt when encountering parallel computations propagates into *both* of the computations. As seen before, two computations are parallel if they are connected by a parallel operator (||).

The purpose of the parallel operator is best explained by the *Broadcasting* section of the operational semantics. The rules state that if one of the computations raises a signal, and the two computations are connected via the parallel operator, then the raised signal will transform into an interrupt for another computation, whilst the signal continues propagating upwards, similar to the signal hoisting rule. The transformation is explained by the duality of the signals and interrupts, i.e. a raised signal from one computation is the same signal which is perceived as an interrupt from the outside.

3.2 FreeRTOS API for Embedded Systems

This section reviews the FreeRTOS API of interest, which is intended to be used to implement asynchronous effects. The main mechanism for communication in FreeRTOS is a queue. If the communication is known to be one-to-one with boolean values of $\top$ and $\bot$, then task notifications can be used, but the queues are more general, and task notifications are outside of the scope of this thesis.

Queues are uniformly typed containers that store objects and are synchronized with the FreeRTOS's scheduler.

To create a queue, one uses a `xQueueCreate` method:

```
QueueHandle_t queue_handle = xQueueCreate(sizeof(my_message), num_of_elements);
```

The method `xQueueCreate` takes two arguments:

- Size of the single element
- Capacity of the queue

If the queue creation is successful, then the `xQueueCreate` returns a pointer to the queue which must be stored in the queue handle.

Communication between tasks occurs over the queues. To put an element onto the queue (and send a message), the following method is used:

```
xQueueSend(queue_handle, &(value), ticks_to_wait);
```

The method `xQueueSend` takes three arguments:

- Queue handle to which queue to send value to
- Value that needs to be sent
- Number of ticks to wait for the operation to succeed

This method can fail in case the queue doesn't become available within the specified time period, so in practice, the return value of the `xQueueSend` must be checked like this:

```
if(xQueueSend(queue_handle), &(value), ticks_to_wait) == pdPASS){
  // successfully sent the value to the queue
} else {
  // sending to the queue has failed...
}
```

There are several problems with `xQueueSend`, which will be explained down below.

First of all, the need to check for a successful operation complicates the logic of the program. Because of the branching logic, now two separate cases need to be closely considered.

Secondly, the need for verifying the success of the operation is not automatically enforced. The semantics of C allow us to invoke the `xQueueCreate` without checking the return type. While in some cases this might be desired, it is still a source of subtle bugs. In theory, this can be enforced by the C compiler to check for the ignored return values of the functions. However, doing so would complicate the code even more, because there are many more functions whose return values are not traditionally checked for, for example `printf`.

Thirdly, `xQueueSend` assumes that the `queue_handle` is a valid handle and not `NULL`. So unless there is a guarantee that the queue will be valid at the moment of sending, the queue handle must be checked before attempting to send the value onto the queue:

```
if(queue_handle != NULL){
  if(xQueueSend(queue_handle), &(value), ticks_to_wait) == pdPASS){
    // successfully sent the value to the queue
  } else {
    // sending to the queue has failed...
  }
} else {
  // queue handle is not valid...
}
```

Fourth, type safety of `value` is not guaranteed. The type signature of `xQueueSend` is the following:

```
 BaseType_t xQueueSend(
                       QueueHandle_t xQueue,
                       const void * pvItemToQueue,
                       TickType_t xTicksToWait
                     );
```

It is important to notice that the type of `pvItemToQueue` is of `const void *`. The type signature tells that the value that is being put onto the queue can be essentially anything. What this means for developers is that it is their responsibility to ensure the type correctness of their operations. While it is easy in simple cases, during development it is often the case that the problem becomes more well-defined, and this is reflected in the code, where the changes to the code eventually applied, however, done so partially.

Fifth, while not exactly the problem with the `xQueueSend`, but is still a source of possible bugs: it is important to ensure that the `queue_handle` is the intended handle. In an application where there are several queues, it is possible to choose the wrong queue to send values to. Ensuring the correctness of the communication in this setting is an additional mental overhead during development.

Those are some problems with the `xQueueSend` method. However, a queue is a unidirectional communication channel, and if the `xQueueSend` is a way to *send* a message, then there is a way to *receive* a message.

To receive a message from the queue, the following method is used:

```
xQueueReceive(queue_handle, &(value), ticks_to_wait);
```

As with the `xQueueSend`, the `xQueueReceive` is a mirror opposite operation and takes three arguments:

- Queue handle from which queue to receive value
- Variable which will hold the resulting value
- Number of ticks to wait for the operation to succeed

Same as `xQueueSend`, the `xQueueReceive` can also fail, so the return value of the operation must also be checked:

```
if(xQueueReceive(queue_handle, &(value), ticks_to_wait) == pdPASS){
  // received a value
} else {
  // didn't receive a value
}
```

If the queue is still empty for the duration of `tick_to_wait`, then the operation will fail.

Fundamental problems with `xQueueSend` apply to `xQueueReceive`'s case as well. In case the operation fails, the `value` variable will hold the previous value, which could be incorrect. As such, checking for the return value of `xQueueReceive` is arguably even more important.

With regards to queues, the last element that needs to be mentioned is the deletion of the said queues. The queues can be deleted using this method:

```
vQueueDelete(queue_handle);
```

The only condition necessary for the following method to work is a valid queue handle. This is arguably a simple method, but it still can cause subtle bugs. In case the queue has been already freed, this will cause a double-free, which is a bug. Depending on the version of the FreeRTOS and other SDKs used, the effects are always different. What needs to happen after the queue is freed is the setting of the queue handle to `NULL`:

```
vQueueDelete(queue_handle);
queue_handle = NULL;
```

While it is not difficult to do, it requires discipline or special wrapper methods that do this automatically:

```
void mySafeQueueDelete(QueueHandle_t *queue_handle){
  if( *queue_handle != NULL ){
    vQueueDelete( *queue_handle );
    *queue_handle = NULL;
  }
}
```

Another important aspect of the FreeRTOS is the *tasks*. Tasks are the main mechanism through which the concurrency and asynchronicity in FreeRTOS are achieved.

To create a task in FreeRTOS, the following method can be used:

```
xTaskCreate( (TaskFunction_t)(&(my_task_body))
           , "my_task_name"
           , stack_size
           , pointer_to_arguments
           , tskIDLE_PRIORITY + 10
           , &my_task_handle
           );
```

So far, the function signature of the `xTaskCreate` is the most involved one. It accepts the following arguments:

- Pointer to the task entry
- Task name (for debugging purposes)
- Stack depth (size)
- Pointer that is used as an argument to the task
- Task priority
- Task handle pointer by which the task can later be referenced

The first argument is essentially a function pointer to the code written in C. The task should be implemented as a function that never returns. It could be a task that gets deleted as soon as it finishes its tasks, but it is typically implemented as a continuous loop. As soon as the task is created, it is scheduled to start and starts to run as soon as the resources become available.

The method is not that difficult to use, but as with `xQueueDelete`, it needs to be controlled properly. For example, the code may be failing, because the is not created when it is assumed to be running. Secondly, `xTaskCreate` requires 6 arguments to be created. It can be easily abstracted with custom functions:

```
void create_my_task(){
  int my_task_stack_size = 1000;
  int my_task_priority = tskIDLE_PRIORITY + 1;
  xTaskCreate( (TaskFunction_t)(&(my_task))
             , "my_task"
             , my_task_stack_size
             , NULL
             , my_task_priority
             , &(my_task_handle)
             );
}
```

The main goal in this case is to make the creation of tasks uniform and consistent throughout the code. Trying to create a task with different parameters in different locations is another source of subtle bugs. Thirdly, trying to dynamically create tasks during runtime is a dangerous task due to limited resources. To create a task, a virtual stack of the task is allocated on the heap. As such, if the allocation fails, the task is not created. There are four ways to go about it:

1. Check for the successful creation of the task
2. Create tasks at the initial start-up only
3. Use `xTaskCreateStatic`
4. Use a microcontroller with more available memory

There are no easy ways to deal with the failure of memory allocation, as such, most of the best practices are focused on making the task creation predictable or trying to optimize the code to get the most efficiency from the available resources.

However, there's yet another important consideration that needs to be made when working with the FreeRTOS tasks. The tasks use a cooperative multithreading model. This means that if the task doesn't voluntarily relinquish control over the CPU, then it will effectively starve out the rest of the tasks. As such, the primitives such as `vTaskDelay` need to be used. They effectively block the currently running task and give way to other scheduled tasks.

While the FreeRTOS provides many primitives and abstractions which can be abstracted over by higher-level languages, the main focus of this thesis is mainly on the tasks and queues.

Chapter 4

EmbedFX

This chapter introduces EmbedFX: a programming language based on asynchronous effects for deeply embedded microcontrollers. The current thesis aims to answer the following questions:

1. Are asynchronous effects feasible in deeply embedded microcontrollers with no dynamic memory allocation?
2. Can EmbedFX be used to express software written in embedded systems?

While the explanation of the answers will be given throughout the chapter, the short answers to the stated questions are as follows:

1. Yes, asynchronous effects are feasible in deeply embedded microcontrollers, but with some limitations to the expressivity of the original calculus
2. Yes, it can be used to express software that is based on asynchronous communication, but otherwise require falling back to idioms and semantics of C, such as having global variables, etc.

In Chapter 3 the asynchronous effects calculus and FreeRTOS API of interest has been described.

4.1 Asynchronous Effects and Deeply Embedded Micronconcontrollers

The reason asynchronous effects were chosen for the embedded systems is because the calculus by Ahman and Pretnar conceptualizes the communications between the asynchronous tasks. Embedded systems are often asynchronous by nature, because:

1. Hardware interrupts: an outside event can trigger an interrupt, which preempts the ongoing computation, performs the dedicated code associated with the interrupt and resumes the computation.
2. Asynchronous tasks: programs for embedded systems are different in nature from those written for general purpose computers: the systems are expected to run indefinitely and respond to outside stimuli. To prevent the blocking of the computation and improve performance, the Real-Time Operating Systems are utilized and the code is modularized into independent tasks.

Embedded systems have interrupts and asynchronous tasks, while asynchronous effects conceptualize interrupts (incoming signals) and asynchronous tasks, as well as their communication mechanism.

Of course, what we are interested in when writing software for embedded systems is to have some guarantee of reliability, i.e. the program does not accidentally crash or halt. However, when we work with deeply embedded systems, we have many constraints to keep in mind, but mainly the limited memory.

The problem with the original asynchronous effects calculus is that it is highly expressive for deeply embedded microcontrollers. It is based on lambda calculus and in a similar fashion, while it does not have loop constructs, it utilizes recursion to achieve that. While the recursion is a theoretically simple concept of self-reference in the definition of the mathematical object, in practice it complicates the development. To achieve the recursion in the mathematical sense, we utilize the stack: saving the current state and then shifting the stack pointers to refer to a new stack frame. When the function returns, the stack is deallocated, and the previous stack frame is returned. Currently, the standard for a general-purpose computer is to have 8GBi of Random-Access Memory, (however, for the best experience, 16GBi and even 32GBi are often used), programs with the unbounded recursion is usually not a problem: the bugs are usually found and dealt with quickly, and even the crash of a single poorly designed program doesn't affect the stability of the system: Operating System recovers all the resources and continues to work. However, in deeply embedded systems the memory constraints are much harsher. Additionally, there is often no Virtual Memory (VM), no Memory Management Unit (MMU), and no Memory Protection Unit (MPU), which are often considered as given in the current general-purpose computers [2]. What this means in practice is that the misuse of memory can have disastrous effects on the stability of the program. While doing the unlimited recursion, it is possible to run on onto other task's stack and corrupt the state. If the recursion uncontrollably allocates memory, at some point the memory allocation will fail. If the error for memory allocation is not checked for, then using and dereferencing a null pointer becomes an undefined behavior. Overall there are many problems with unrestricted recursion, and as such, the recursion needs to be restricted.

Needless to say, the asynchronous effects calculus is a theoretical concept that requires the small details and assumptions of operational semantics carefully implemented, such as the propagation of signals or the capturing of the closures.

In practice, this assumes dynamic memory allocation, which needs to be heavily restricted in deeply embedded microcontrollers. As such, the asynchronous effects need to be "interpreted" to be implemented in such a context.

4.2 Restricting the Expressivity of Asynchronous Effects

The problem with the asynchronous effects that needs to be resolved is that the order of effects affects the computation.

Assuming we have two effects op_1 and op_2, and we have a computation $N : X!(o, \iota)$, we have two situations:

$$\downarrow \mathrm{op}_1\ (V,\ \downarrow \mathrm{op}_2\ (W, N))$$

and

$$\downarrow \mathrm{op}_2\ (W,\ \downarrow \mathrm{op}_1\ (V, N))$$

According to the definition of actions of interrupts ($\mathrm{op} \downarrow (o, \iota)$) we will have the following types:

$$\mathrm{op}_1 \downarrow (o, \iota) = (o \cup o', \iota[\mathrm{op}_1 \mapsto \bot] \cup \iota')$$

and

$$\mathrm{op}_2 \downarrow (o, \iota) = (o \cup o'', \iota[\mathrm{op}_2 \mapsto \bot] \cup \iota'')$$

In general case:

$$(o \cup o', \iota[\mathrm{op}_1 \mapsto \bot] \cup \iota') \neq (o \cup o'', \iota[\mathrm{op}_2 \mapsto \bot] \cup \iota'')$$

What this means, is that the different order of interrupts may affect which of the subsequent interrupt handlers will be triggered. According to the operational semantics, as soon as the handler is triggered, it is removed from the active listening handlers. However, there is no easy way to describe this behavior: the main communication mechanism observed in this thesis is queues. The queues have specific endpoints: start and end.

What may happen in the original asynchronous effects calculus, is the following: when op_1 triggers some interrupt handler, it may install new interrupt handlers which may get trigger on subsequent op_2, but was not present at the moment of raising the said op_2. Such behavior is difficult to represent with queues because queues need to have specific sources and destinations. We may try to solve the following equation:

$$\mathrm{op}_1 \downarrow (\mathrm{op}_2 \downarrow (o, \iota)) = \mathrm{op}_2 \downarrow (\mathrm{op}_1 \downarrow (o, \iota))$$

$$\mathrm{op}_1 \downarrow (o \cup o'', \iota[\mathrm{op}_2 \mapsto \bot] \cup \iota'') = \mathrm{op}_2 \downarrow (o \cup o', \iota[\mathrm{op}_1 \mapsto \bot] \cup \iota')$$

$$\iota[\mathrm{op}_2 \mapsto \bot](\mathrm{op}_1) = (o', \iota')$$

$$\iota[\mathrm{op}_1 \mapsto \bot](\mathrm{op}_2) = (o'', \iota'')$$

$$((o \cup o'') \cup o' \cup o''') = ((o \cup o') \cup o'' \cup o'''')$$

$$o''' = o''''$$

$$(\iota[\mathrm{op}_1 \mapsto \bot, \mathrm{op}_2 \mapsto \bot] \cup \iota''[\mathrm{op}_1 \mapsto \bot] \cup \iota' \cup \iota''') = (\iota[\mathrm{op}_1 \mapsto \bot, \mathrm{op}_2 \mapsto \bot] \cup \iota'[\mathrm{op}_2 \mapsto \bot] \cup \iota'' \cup \iota'''')$$

$$(\iota''[\mathrm{op}_1 \mapsto \bot] \cup \iota' \cup \iota''') = (\iota'[\mathrm{op}_2 \mapsto \bot] \cup \iota'' \cup \iota'''')$$

$$(\iota' \cup \iota''[\mathrm{op}_1 \mapsto \bot] \cup \iota''') = (\iota'[\mathrm{op}_2 \mapsto \bot] \cup \iota'' \cup \iota'''')$$

Again, in the general case, the equations are not unifiable, unless:

$$(\emptyset \cup \emptyset \cup \emptyset) = (\emptyset \cup \emptyset \cup \emptyset)$$

As such, this equation:

$$((o \cup o'') \cup o' \cup o''') = ((o \cup o') \cup o'' \cup o'''')$$

Becomes this:

$$((o \cup o'') \cup o' \cup \emptyset) = ((o \cup o') \cup o'' \cup \emptyset)$$

This is possible if:

$$\iota(\mathrm{op}_1) = (o', \emptyset) \text{ and } \iota(\mathrm{op}_2) = (o'', \emptyset)$$

When a triggered interrupt handler *does not* install new interrupt handlers, they can be represented with queues. In such a case, the order of effects has no meaning on the propagation of the effects. To achieve the following effect, the promises have to be defined *non-recursively*: in the definition of the computation of the promise, if we use another promise, that promise must be defined beforehand, i.e.:

$$\frac{\Gamma \vdash M_{\mathrm{op}} : X!(o, \emptyset) \qquad \Gamma, p : \langle X \rangle, x : A_{\mathrm{op}'} \vdash M_{\mathrm{op}'} : Y!(o', \emptyset) \qquad \Gamma, p : \langle X \rangle, p' : \langle Y \rangle \vdash N : Z!(o'', \emptyset)}{\Gamma, p : \langle X \rangle \vdash \text{promise } (\mathrm{op}'\ x \mapsto M_{\mathrm{op}'}) \text{ as } p' \text{ in } N : Z!(o'', \emptyset)}$$

Figure 4.1: Non-recursive definition of the promises

If promises are defined in this way, they are defined non-recursively. For each promise definition that refers to another promise, the referred promise must be defined beforehand. The definition of non-recursive promises is the core idea of EmbedFX.

With this, it is now possible to explain the logic of EmbedFX. While EmbedFX is based on the concepts and ideas of asynchronous effects, to make the execution on deeply embedded microcontrollers possible, it applies limitations on how the primitives of original asynchronous effects calculus may be used. This chapter will show how the primitives of EmbedFX can be described using the original asynchronous effects calculus.

1. **Promises are defined on a toplevel**. If in the original asynchronous effects calculus promises are a part of an expression and could be defined in the middle of the computation, in EmbedFX they are defined at the toplevel, similar to how functions are defined in C. This brings several benefits which are important for the proper conversion to C:

   - promises have unique identifiers
   - promises do not capture closures
   - promises are not a first-class-citizen

   In EmbedFX promises are referred to by their identifiers, rather than the variables they are bound to. Additionally, since promises are evaluated asynchronously to the computation in which they are referred to, they are implemented using tasks. Having a unique name and referring to that name helps ensure that if the names are the same, they are referring to the same promise definition, which helps effectively re-use the promise definitions and tasks. Lastly, since the promises are not first-class citizen, it means they cannot be stored in a variable, which is necessary for the calculation of the propagation of the effects.

2. **Promises are not recursive**. Promises do not have recursion. What this means is, that if we are defining a promise p, then we cannot refer to p from inside the definition. However, We can refer to other promises if they eventually don't lead to p somewhere in their definition.

3. **There is no "run" primitive, only tasks**. EmbedFX introduces the concept of a task, a way to define asynchronous computations. Code defined with the "task" primitive is known to run asynchronously. The difference between tasks and "run" primitive is that if the computation captured by "run" is run only once, then the computation defined with the task runs infinitely. If the computation finishes, then the computation is then restarted.

4. **Tasks are defined on a toplevel**. Similarly to promises, tasks are defined on a toplevel and we have two specific benefits of importance:

   - Tasks are referred to by a unique identifier
   - Tasks do not capture closures

   Since tasks are eventually implemented using FreeRTOS tasks, the structure is similar to those used in C. Having a unique identifier allows us to define tasks on toplevel and still refer to them by name, which removes the need to awkwardly structure code when we want to connect several of them via a parallel operator. Just like promises, since tasks are toplevel, they don't capture closures, which means no inherent state associated with tasks.

5. **Parallels is a list of tasks, rather than binary operator**. In original asynchronous effects calculus, parallel asynchronous computations are defined with a parallel binary operator (||). In EmbedFX, however, parallels is a toplevel construct that connects tasks. Tasks connected via parallels can directly send effects to each other. However, the transitivity on parallels doesn't apply. For effect to traverse to other tasks, it must become an outgoing signal. However, according to operational semantics, signals become interrupts when they enter other tasks.

$$\frac{x : A_{\text{op}} \qquad \Gamma, x : A_{\text{op}} \vdash M_{\text{op}} : X!(o, \emptyset) \qquad (p', \langle X \rangle) \notin \Gamma \qquad \Gamma, p' : \langle X \rangle \vdash N : Y!(o', \iota')}{\text{promise}(p,\ \text{op},\ x)\{M_{\text{op}}\} \text{ represented as promise (op } x \mapsto M_{\text{op}}) \text{ as } p' \text{ in } N}$$

$$\frac{\Gamma \vdash N : X!(o, \iota)}{\text{task}(t,\ N) \text{ represented as run (let rec } t\ () \ = \ (N; t()) \text{ in } t())}$$

$$\frac{\Gamma \vdash p_i = \text{task}(t_i,\ N_i)}{\text{parallel}[p_1, \ldots, p_n] \text{ represented as } p_1 \ || \ \ldots \ || \ p_n}$$

Figure 4.2: Representation of EmbedFX constructs in asynchronous effects calculus

Figure 4.2 shows how the modified elements of EmbedFX can be represented in the core asynchronous effects calculus.

As has been mentioned before, non-recursive promises are necessary for the eventual translation of EmbedFX to C. So the first deduction rule shows that the promises defined in the EmbedFX are apriori non-recursive, i.e. when interrupts trigger the promise, the triggered promise doesn't install new interrupt handlers. Since each promise is defined in a toplevel, they can be used in the definition of the tasks or other promises, given they do not introduce recursion. When the promises are used within a task, the promise is associated with that task, and thus the computation has some interrupt handlers installed (ι). If the promise definition in EmbedFX refers to some other promise name, then the referred promise is bound by the task from which the parent promise is invoked. So what happens is essentially topological sorting of the promises. Something, that is not clear from the rules, is that unique identifiers are given to promises defined within EmbedFX to distinguish between different occurrences of those promises. So if for example in the definition of promise p_1 in EmbedFX we refer to another promise p_2 two or more times, what would happen is that we install *two* new interrupt handlers, each having a unique name, by which it can be referred to from within the **core asynchronous effects calculus**, not EmbedFX. While we cannot tell what the bound names are, we can say for sure that their computations are essentially the same, because they are referred to by the same identifier in EmbedFX. This is necessary because if the names from within the asynchronous effects calculus are the same, then for all occurrences of p_2, the interrupt handler would trigger only once, whereas we expect the interrupt handlers to trigger for *each* occurence of p_2.

The second rule describes how tasks correspond to the "run" primitives from asynchronous effects calculus. As has been stated before, tasks run infinitely, whereas the simple asynchronous computations run only once. To bridge the gap between them, the computation of tasks is wrapped in a recursive function that evaluates infinitely. It is important to note that in EmbedFX projection, this is the only case when a recursive function definitions are used.

The last rule describes the relationship between the "parallel" toplevel definition and a parallel operator ($||$). Essentially all the tasks that are being referred to in a "parallel" construction are all connected via a parallel operator. From this we can derive the fact that:

$$\text{parallel}[p_1, \ldots, \uparrow \text{op } (V, p_i), \ldots, p_n] \rightsquigarrow \text{parallel}[\downarrow \text{op } (V, p_1), \ldots, p_i, \ldots, \downarrow \text{op } (V, p_n)]$$

This comes straight from the definitions:

$$\begin{array}{rlrcccccccc}
 & \text{parallel} & [p_1, & & \ldots, & & \uparrow \text{op } (V, p_i), & & \ldots, & & p_n] \\
= & & p_1 & || & \ldots & || & \uparrow \text{op } (V, p_i) & || & \ldots & || & p_n \\
\rightsquigarrow & & \uparrow \text{op } (V, \downarrow \text{op } (V, p_1) & || & \ldots & || & p_i & || & \ldots & || & \downarrow \text{op } (V, p_n)) \\
= & & \downarrow \text{op } (V, p_1) & || & \ldots & || & p_i & || & \ldots & || & \downarrow \text{op } (V, p_n) \\
= & \text{parallel} & [\downarrow \text{op } (V, p_1), & & \ldots, & & p_i, & & \ldots, & & \downarrow \text{op } (V, p_n)]
\end{array}$$

First, we expand the "parallel" according to the definition. After that apply the small-step evaluation rule from Figure 3.1, namely the rules from the *Broadcasting* section. In the next step, we use the fact that the "parallel" is a toplevel construct, which means that there is no lexical scope outside of it, and thus nowhere for the signal propagate to. As such, we remove the outgoing signal. Lastly, we reuse the definition of "parallel" in reverse.

With this, we have shown that the EmbedFX primitives can be defined with no alterations to the original asynchronous effects calculus, and for that reason, we can benefit from the type safety that has been established by [3]. In simple terms, EmbedFX can be seen as a specialization of asynchronous effects. Additional specializations that haven't been mentioned, as they are not necessary for establishing correspondence between asynchronous effects and EmbedFX, they are still included in the design of the EmbedFX:

1. **No recursion**. EmbedFX targets deeply embedded microcontrollers. As such, those devices often have limited resources. Unbounded recursion would result in unbounded memory consumption and stack overflow. For that reason, recursion is disallowed in EmbedFX.

2. **No higher-order functions**. The resulting EmbedFX code is eventually transpiled to C. While the core asynchronous effects calculus permits higher-order functions, introducing them in deeply embedded microcontrollers introduces problems that require additional remediation of asynchronous effects calculus and C. The main source of the problems is the prohibition of dynamic memory allocation. The problems with the higher-order functions are described in the Discussion chapter.

This section covers the theoretical underpinnings of EmbedFX, and the next will describe how the EmbedFX is intended to be used.

4.3 EmbedFX in practice

EmbedFX specializes asynchronous effects calculus in a specific way, but the usage of asynchronous effects is consistent with the rules of the core calculus.

It would be easier to explain the structure and behavior of EmbedFX with some examples. EmbedFX is implemented as a library in an OCaml language, as such, it uses the constructs of OCaml to construct an AST that represents an EmbedFX program. Each program starts with the `toplevel` clause.

```
open EmbedFX_DSL

let my_code =
  toplevel [
    (* EmbedFX code goes here *)
  ]
```

While internally `toplevel` is implemented as an identity function, which means that EmbedFX is just a list of toplevel elements, it helps to improve readability and provide type information to make type errors readable. Since currently EmbedFX is implemented as a DSL library, it means that we can perform any AST manipulations, as long as the resulting code is an EmbedFX program.

There are several toplevel elements in EmbedFX. One of them is functions:

```
  _func void "app_main" [] [
    (* initial code goes here *)
  ];
```

Functions in EmbedFX are declared using the `_func` construct. As in C, it takes several parameters, such as:

- return type

- function name
- input arguments
- function code

It is important to note that each EmbedFX must have an `app_main` function. Since the resulting code is converted to C which utilizes FreeRTOS, the function `app_main` must be present, even if it is empty. It can be used as the initial code to execute, similar to Arduino's `setup()`.

Currently, however, EmbedFX is far from being useful in serious settings, since not all of the ergonomic features are implemented. As such, it may be important to fall back on a C or external functions. For that reason, EmbedFX provides a way to define external functions so the references to them from within EmbedFX is valid:

```
  _func_type "printf" [str] void;
  _func_type "rand" [] _int;
```

With this code, we define two external functions `printf()` and `rand()`. While their actual type signatures may be different, the function signatures defined need to be consistent *within* EmbedFX. As such, for example, `printf()` can actually accept variable number of arguments and returns an integer, however, in EmbedFX it has a different type signature that is consistent with its use case.

The defined functions can now be called with a special primitive:

```
let printf x =
  e (_func_call "printf" [x])
```

The `_func_call` primitive performs a function call with specified arguments. The primitive is quite verbose, so it is preferable to wrap the function calls in another function and utilize the wrapper function in the code instead. The `_func_call` is an expression, while the function body expects a statement, so the function call is wrapped into an expression constructor: `e`. If the specific function call is expected to be used only as a statement (such as `vTaskDelay` or `printf`), then the `_func_call'` can be used instead (note the `'`). It automatically converts `_func_call` into a statement.

Since EmbedFX is based on asynchronous effects calculus, it is important to define effects:

```
  eff "E" []; (* accepts list of types *)
```

The `eff` primitive defines effects. It also allows to define the list of types, the values of which are passed along with the effects.

To raise an effect, a `raise` primitive is used:

```
  raise "E" [];
```

`raise` accepts a string and a list of expressions, which are used to construct an effect that needs to be raised.

To define a promise, a `promise` primitive is used:

```
promise "p" ("E", []) [
  return (_b true); (* promise code goes here *)
];
```

`promise` primitive accepts several values, such as:

- promise name
- listening effect and bound variables
- interrupt handler code

It is important to note that `promise` is a toplevel element, whereas it accepts a list of statements. When defining a promise, it is necessary to give it a unique name and specify the effect on which it is triggered, along with the variable names, to which the passed values are bound. To pass by the values that have been computed within promises, they need to be "returned" explicitly. As a result of that "return", if the promise is being awaited, the promise is fulfilled with that value. Promises can also refer to other promises' identifiers, as long as it doesn't create a recursion.

With the created promises it is also important to be able to await them:

```
_let "x" await (id "p");
```

The `await` primitive blocks the current task until the promise it refers to becomes fulfilled. Additionally, `await` primitive can only accept strings, which must be a valid promise identifier.

It is necessary to be able to define tasks:

```
task "my_task" [
  (* task code goes here *)
];
```

Code defined by tasks runs asynchronously and infinitely: when the computation is evaluated, it restarts again. Each task has a unique name, by which it can be referred. This is necessary when defining parallels. Tasks are also a toplevel construct, like functions, promises, and effects.

Tasks host code that installs interrupt handlers and raises effects. To allow the propagation of effects between tasks, it is necessary to define parallels:

```
parallel [
  "task_1";
  "task_2";
  "task_3";
]
```

Parallel is a toplevel construct that accepts a list of strings: names of tasks, which can send and receive signals and interrupts from each other.

Lastly, we may have signals that are coming from the outside of our system. To define them, we can use `interrupt` primitive:

```
interrupt "interrupt_name" "E";
```

From EmbedFX's type system perspective, `interrupt` is just another task with the name of `"interrupt_name"` which simply raises an effect. Since `interrupt` mimics tasks, it is also a toplevel element, and the interrupt name can be used with `parallel` primitive. After the code is transpiled, the user must provide code that raises necessary effects and propagates them to the correct destinations.

With the basic building blocks covered, we can now present a simple example in EmbedFX:

```
let random_sleep_example =
  let delay x = _func_call' "vTaskDelay" [x] in
  let printf s' = _func_call' "printf" [_s (s' ^ "\n")] in
  let rand_sleep =
    let rand = _func_call "rand" [] in
    [ _let "sleep_time" ((rand |% (_i 30)) |+ (_i 10));
      delay ((id "sleep_time") |* (_i 20));
    ]
  in
  toplevel [
    eff "E" [];

    _func_type "rand" [] _int;
    _func_type "vTaskDelay" [_int] void;
    _func_type "printf" [str] void;

    promise "receive" ("E", []) [
      printf "MSG receiver: received";
    ];

    task "sender" ([
      printf "MSG sender: raising effect...";
      raise' "E" [];
      printf "MSG sender: sleeping for random time...";] @
      rand_sleep);

    task "receiver" [
      e @@ id "receive";
    ];

    parallel [
      "sender";
      "receiver";
    ];

    _func void "app_main" [] [
    ]
]
```

The code seems a bit long but can be divided into several parts. In the OCaml code, we provide helper code that simplifies writing in EmbedFX, namely, invoke `printf()`, `vTaskDelay()` and `rand()`.

```
let delay x = _func_call' "vTaskDelay" [x] in
let printf s' = _func_call' "printf" [_s (s' ^ "\n")] in
let rand_sleep =
  let rand = _func_call "rand" [] in
  [ _let "sleep_time" ((rand |% (_i 30)) |+ (_i 10));
    delay ((id "sleep_time") |* (_i 20));
  ]
in
```

Next begins the toplevel section of the EmbedFX code, and we start by defining effect "`E`" and external functions:

```
toplevel [
  eff "E" [];

  _func_type "rand" [] _int;
  _func_type "vTaskDelay" [_int] void;
  _func_type "printf" [str] void;
```

After that, we define a promise that performs some operations when it is triggered by an effect "E", in this case printing:

```
  promise "receive" ("E", []) [
    printf "MSG receiver: received";
  ];
```

The finishing touch is the definition of tasks and allowing communication between them:

```
  task "sender" ([
    printf "MSG sender: raising effect...";
    raise' "E" [];
    printf "MSG sender: sleeping for random time...";] @
    rand_sleep);

  task "receiver" [
    e @@ id "receive";
  ];

  parallel [
    "sender";
    "receiver";
  ];
```

In this case, we have two tasks: `sender` and `receiver`. As can be seen, the sender sends a signal "`E`" and then goes to sleep for a random time. Receiver on the other hand has only one identifier: `receive`, which is the name of the promise. Lastly, the tasks are connected via `parallel`. When the effect "`E`" is raised from `sender`, it propagates to `receiver` task, where the promise `receive` is waiting. Because the promise is mentioned in the body of the task, it gets triggered, which then executes the code defined in the promise `receive`.

So in practice, if the `raise` is mentioned in the body of the task, it raises an effect, and if a matching promise is mentioned in the body of the task the effect can propagate to, then the effect is triggered. Those are the simple mechanics behind the EmbedFX.

We can even have promises inside other promises:

```
let promise_in_promise_2_example =
  let printf s = _func_call "printf" [_s s] in
  let printf' s = e @@ printf s in
  let _task x =
    task x [
      e @@ raise "E" [];
      e @@ id "p1";
      printf' @@ Printf.sprintf "From task %s" x;
    ]
  in
  toplevel [
    _func_type "printf" [str] void;
    eff "E" [];

    promise "p2" ("E", []) [
      printf' "Printing from promise p2";
    ];

    promise "p1" ("E", []) [
      e @@ id "p2";
      printf' "Printing from promise p1";
      e @@ id "p2";
    ];

    _task "A"; (* simply raises an effect *)
    _task "B"; (* and mentions p1 *)

    parallel [ "A"; "B" ];

    _func void "app_main" [] [
    ];
  ]
```

This example illustrates that we can mention promises inside other promises. As mentioned before, the promises are implemented *non-recursively*, which in this case means that a single raised effect “E” triggers both p1 and p2, and that p2 gets triggered *twice*, because it is mentioned twice in the body of p1.

Another example illustrates the blocking behavior of await primitive:

```
let await_2_example =
  let printfn s = e @@ _func_call "printf" [_s (s ^ "\n")] in
  let delay i = e @@ _func_call "vTaskDelay" [_i i] in
  toplevel [
    _func_type "printf" [str] void;
    _func_type "vTaskDelay" [_int] void;

    eff "E" [];
    eff "F" [];

    promise "p_E" ("E", []) [
      printfn "Triggered on E";
      return (_b true);
    ];

    promise "p_F" ("F", []) [
      printfn "Triggered on F";
      return (_b true);
    ];

    task "sender" [
      printfn "Sending E";
      raise' "E" [];
      delay 50;
    ];

    task "await_E" [
      e @@ await "p_E";
      printfn "p_E resolved, sending F";
      raise' "F" [];
    ];

    task "await_F" [
      e @@ await "p_F";
      printfn "p_F resolved";
    ];

    parallel [
      "sender";
      "await_E";
      "await_F";
    ];

    _func void "app_main" [] [
    ];
  ]
```

In this example we have three tasks, one "`sender`" and two await tasks: "`await_E`" and "`await_F`". What happens in this case is that the following:

1. Initial task `sender` sends an initial signal "`E`" which triggers `p_E`.
2. Task `await_E` awaits until `p_E` is resolved, after which it raises "`F`", which triggers `p_F`.
3. Task `await_F` awaits until `p_F` is resolved.
4. Steps 1 to 3 repeat again.

The result of the code is the following message repeated over and over:

```
Sending E
Triggered on E
p_E resolved, sending F
Triggered on F
p_F resolved
```

More examples are provided in the Appendix of this thesis.

Chapter 5

Implementation of EmbedFX

As has been stated before, one of the main primitives supplied by the FreeRTOS are tasks and queues. Queues act mainly as a communication mechanism between tasks. Depending on the design and the architecture of the software, communication between tasks can quickly become complicated.

Asynchronous effects, on the other hand, provide low-level primitives for task creation and communication. As such, in theory, Asynchronous Effects can be used to provide meaningful abstractions over the task creation and communication. However, the original work of Ahman and Pretnar is a theoretical concept that has not been yet implemented for deeply embedded systems. The original calculus makes several assumptions such as unlimited resources, success of low-level operations and step-by-step evaluation of the semantics. To adapt the asynchronous effects calculus to the deeply embedded systems' environment, several concessions need to be made.

5.1 EmbedFX Tasks to C

In the original asynchronous effects calculus, the asynchronous computations can be created from anywhere and in any quantity. Trying to dynamically create tasks in a microcontroller setting will quickly exhaust the available memory and crash the system. As such, the creation of tasks needs to be limited.

In EmbedFX, the tasks are a toplevel construct:

```
toplevel [
  task "task_name" [
    (* task statements go here *)
  ]
]
```

Secondly, in the asynchronous effects calculus, the asynchronous tasks run only once. However, the tasks in embedded systems are typically designed as continuous loops which do not return.

Thirdly, in the original asynchronous effects, since the tasks can be created anywhere, they also effectively capture the closures: the surrounding environment, the variables, definitions, etc.

To have proper closures, one must perform dynamic memory allocation to create the closures and store them on heap, to make them available throughout the program. Additionally, the resulting closures need to be deallocated when the capturing task or function goes out of scope, which requires a garbage collector. Trying to create the closures on stacks may be possible, but it is difficult and must be handled carefully, as well as different copies of the closures need to be remediated.

The code from above produces an analogous code in C:

```
TaskHandle_t task_my_task;

void task_my_task(void);

void task_my_task(void) {
  while(1){
    {
      /* task statements go here */
    }
    vTaskDelay((TickType_t)(10));
  }
  vTaskDelete(NULL);
}

// ...

void setup(){
  // ...
  // formatted for readability
  xTaskCreate( (TaskHandle_t)(&(task_my_task))
             , "my_task"
             , TASK_my_task_STACK_SIZE
             , NULL
             , (tskIDLE_PRIORITY) + (1)
             , &(task_my_task_handle)
             );
}
```

When transforming the tasks from EmbedFX to FreeRTOS, several considerations need to be made.

First of all, the function signature of `my_task` is created:

```
void task_my_task(void);
```

This makes referencing the function throughout the code easier.

Later, the task is converted to the C function:

```
void task_my_task(void){
  while(1){
    {
      /* task statements go here */
    }
    vTaskDelay(10);
  }
  vTaskDelete(NULL);
}
```

The body of the task is converted into analogous C statements which are then wrapped into an infinite while loop. At the end of the statements body, the call to `vTaskDelay` is made, to allow the scheduling of other tasks, since FreeRTOS supports cooperative scheduling and cooperative scheduling is assumed. At the end of the function, the single call to `vTaskDelete` is made. Since the pointer that is passed is `NULL`, the task that made the call is the one that gets deleted. While in theory there should be no way to escape from the `while` loop, since many unpredictable things may happen, as a precaution, the single call to the task deletion is made. In reality, it could be substituted with something else, like an infinite while loop which will block the whole application. Since the `task` primitive is a toplevel construct, no closure is being captured. As such, this code mimics the asynchronous computations from the asynchronous effects calculus.

Lastly, to properly ensure the task creation, the last bits of code are generated:

```
TaskHandle_t task_my_task;

// ...

void setup(void){
  // ...
  // formatted for readability
  xTaskCreate( (TaskHandle_t)(&(task_my_task))
             , "my_task"
             , TASK_my_task_STACK_SIZE
             , NULL
             , (tskIDLE_PRIORITY) + (1)
             , &(task_my_task_handle)
             );
}
```

When the body of the task is created, the corresponding call to the `xTaskCreate` is also made in the `setup` method.

`setup` is a function that is run once during the start-up with the sole purpose of setting up the environment for the running of the program: creating queues and initiating tasks. The `TASK_my_task_STACK_SIZE` is a macro defined at the beginning of the file and has a default set by the language. The values can be easily changed by the user of the language. Additionally, the `task_my_task_handle` is defined at the global level and is guaranteed to always refer to the `task_my_task` if such task is created and has not been deleted yet.

5.2 EmbedFX Effects to C

Effects are the "messages" of the asynchronous effects calculus. They are used in conjunction with `raise op(V)` (when raising effects) and `promise(op(x) -> M)` (when triggering promises). So the example effect definition in EmbedFX:

```
toplevel [
  eff "E" [_char, _bool];
]
```

Will be converted to C is as follows:

```
typedef struct {
  TASKS_ENUM_TYPE pos_0;
  char pos_1;
  bool pos_2;
} E;
```

Essentially C structs are used to represent the effects. In the definition of the effect in EmbedFX, there are two requirements:

1. Effect name
2. Effect values

When converted to C, we have the same information:

1. Struct name
2. "Origin" of the effect (`pos0`)
3. Effect values (`pos1`, ...)

Since in the previous example, the effect `E` has two values of types `char` and `bool`, the arguments `pos1` and `pos2` have the corresponding types of `char` and `bool` as well.

There is an additional field that is present in the struct definition: `pos0`. This field has a specific purpose, to annotate the "origin" of the effect. If we have the following example in EmbedFX:

```
toplevel [
  task "sender_A" [
    raise "E" [];
  ];
  task "sender_B" [
    raise "E" [];
  ];
  task "receiver" [
    e @@ id "p"; (* triggers on E *)
  ];
]
```

The same effect "E" is raised from two different locations. As such, in the resulting C code the structs will have two different values:

```
void task_sender_A(void){
  while(true){
    {
      E new_local_3 = { .pos_0 = TASK_ENUM_VALUE_sender_A };
      // code omitted...
}

void task_sender_B(void){
  while(true){
    {
      E new_local_1 = { .pos_0 = TASK_ENUM_VALUE_sender_B };
      // code omitted...
}
```

As can be seen from the examples, `pos_0` holds the origin task of the raised effect. `pos_0` has a type `TASK_ENUM_TYPE` which is an enumerator which contains a value for each task defined. Since all tasks are statically defined, the enum is known to be complete. While the information about the origin of the effect is already encoded in the code, as such, there is no need to store the origin of the effect in the effect definition. However, in such cases, in case there are no types stored in the effect, the struct size becomes 0. Creating a queue for such a struct would result in a queue size of 0, which is incidentally used in the implementation of FreeRTOS semaphores. As such, the queue may behave in unexpected ways due to the implementation details changing across different versions and variations of FreeRTOS. So to keep structures of a size > 0, as an artifact, the origin task of the effect is stored in the effect definition.

5.3 EmbedFX "raise" to C

In Asynchronous Effects calculus, the "raise" primitive is used to send messages. In the context of FreeRTOS, the queues are used for message passing. As such, the raising of effects is converted to the underlying message passing with queues.

For example, we have the following code:

```
raise' "E" [];
```

This is transformed into the following code:

```
E new_local_1 = { .pos_0 = TASK_ENUM_VALUE_A };
uint8_t new_local_2 = (((uint8_t)(1)) << (1)) - (1);
```

```
while((new_local_2) != (0)){
  if((((((uint8_t)(new_local_2)) >> (0)) & (1)) == (1)) && (((queue_p_B) == (NULL)) ||
  ((xQueueSend(queue_p_B, &(new_local_1), (TickType_t)(QUEUE_queue_p_B_SEND_WAIT_TICKS)))
  ==
  (pdPASS))))
  {
    new_local_2 = ((uint8_t)(new_local_2)) - (((uint8_t)(1)) << (0));
  }
  vTaskDelay((TickType_t)(10));
}
```

The raising of the message consists of several steps:

1. Creating the effect
2. Initiating the "flag" bitmask
3. Attempting to send message to all recipients

When trying to send the message, we have to have a list of the recipients. At the code generation stage, we assume that the list of recipients is known. Following the semantics of the asynchronous effects, we can move on with the computation only after we have raised the effect. The propagation of the effects is baked into the semantics of the asynchronous effects calculus. However, in the case of deeply embedded systems, we only have queues. As such, we must attempt to send the message to ALL of the recipients, and only then continue with the computation.

Since `xQueueSend` can fail, we must keep track of whether we have sent the message or not. One way to do that is to have an array of boolean values. However, if we have 126 queues to send messages to, it would require 126 bytes to store true/false values. If we utilize bitmasks, we can reduce the occupied space to 16 bytes (if we use two `uint64_t`) or 17 bytes (four `uint32_t` and one `uint8_t` variables, or 20 bytes if we take the memory alignment into account). As such, bitmasks are used to keep track of the sent messages.

The used bitmask is dependent on the number of queues, and when the bitmask is equal to 0, we have sent messages to all of the queues. As such, the outer loop is a while loop that runs until the bitmask becomes 0. In the body, we check if we haven't yet sent to the queue (the flag is set to 1) and that we can send to the queue (`queue_p == NULL`) and the sending is successful (`xQueueSend(...) == pdPASS`). In such case we set the bit flag to 0, otherwise we move on. In the end, there is one call to `vTaskDelay` to move to allow other tasks to run, and possibly free up the space on queues, to which we haven't sent messages yet.

This conversion of raising effects to C shows what kind of considerations have to be made when writing low-level code.

5.4 EmbedFX Promises to C

A promise is the separation of interrupt handlers from their computation. As such, effectively any computation can trigger the "handler", a.k.a. promise. In that case, promises act as a code that is triggered upon a specific signal.

To model promises effectively in deeply embedded systems, the expressive power of the promises have been decreased. In the original asynchronous effects calculus, the promises are first-class-citizen, which means that the functions can accept, create, and return promises. Additionally, the original asynchronous effects calculus does not have loop constructs, and as such, to achieve the looping behavior, the recursion is heavily utilized. However, in the deeply embedded context having an uncontrolled recursion is dangerous. As such, the promises of EmbedFX do not allow recursion. Since there is no recursion, the computation associated with the promise is assumed to terminate eventually.

Promises in the EmbedFX are a toplevel primitive. Additionally, promises are not first-class-citizen, which means that the promises cannot be stored in the variable and must always be referred to by their name. This is necessary to statically ensure the propagation of the signals.

This is the example definition of the promise in EmbedFX:

```
toplevel [
  promise "p" ("E", []) [
    (* code goes here *)
  ]
]
```

Which is then converted to the following C code:

```
TaskHandle_t promise_p_B_handle;
QueueHandle_t queue_p_B;
void promise_p_B(void);

void promise_p_B(void){
  while(true){
    E new_local_0;
    while(true){
      if((xQueueReceive(queue_p_B,
                        &(new_local_0),
                        (TickType_t)(PROMISE_p_RECEIVE_WAIT)))
          ==
          (pdTRUE))
      {
        break;
      }
    }
    TASKS_ENUM_TYPE origin;
    origin = new_local_0.pos_0;
    {
      /* interrupt handler code here */
    }
    vTaskDelay((TickType_t)(10));
  }
}

void setup(void){
  // ...
  // formatted for readability
  xTaskCreate( (TaskFunction_t)(&(promise_p_B))
             , "promise_p_B"
             , TASK_p_STACK_SIZE
             , NULL
             , (tskIDLE_PRIORITY) + (1)
             , &(promise_p_B_handle)
             );
}
```

Several things are happening at this stage. First of all, the promises are represented in the same way as tasks, i.e. using FreeRTOS tasks. As such, many points regarding the tasks are relevant to promises, such as automatic handler and task creation.

The difference between the task and promise is that a promise is a computation that is triggered by an effect, whereas tasks run constantly and independently. Since promises are on the receiving end of `raise`, they need to await the signals coming from the queues.

The first while loop is a continuous body loop for tasks. The second loop is responsible for awaiting the signal from the queue. If the queue doesn't become available in the given period (`PROMISE_p_RECEIVE_WAIT`), then the code doesn't escape the loop and awaits the signal again until it becomes available.

The received effect is in the local variable `new_local_N`, which is then unpacked if the values of the effect are bound by some variables. Those bound variables are then used in the subsequent computations. There is an additional variable `origin`. It binds the origin of the effect, however, for practical purposes, it is not used. Location of the definition of `origin` variable is where the values associated with the effect are bound to local variables.

As can be noticed, the name of the task associated with the promise "`p`" is `promise_p_B`, where "`B`" refers to the task in which the promise "`p`" is mentioned. What this means is that for each promise there are several copies, each associated with the task from which it is invoked. This is important because of the following case:

```
promise "p" ("E", []) [
  _if' (rand_50_50) [
    printfn "No await";
  ][
    printfn "Have to do await...";
    e @@ await "p2";
  ]
];
```

```
task "A" [
  e @@ id "p";
];

task "B" [
  e @@ id "p";
];

(* rest of the code omitted *)
```

If the promise definition is shared among the different instances of promise "p", then in a 50/50 chance that a promise becomes blocked, because it is awaiting the fulfillment of a promise "p2", then it would block other instances of "p" that are not actually being blocked.

Additionally, a new copy of a promise is being generated for each instance of "p" that is being `await`-ed for. In the following example:

```
promise "p" ("E", []) [
  (* Some interrupt handler code *)
  return (_b true);
];

task "A" [
  raise' "E" [];
];

task "B" [
  e @@ await "p";
  e @@ await "p";
  e @@ await "p";
  printfn "Done!";
];

parallel ["A"; "B"];
```

In this example one instance of raised effect "E" triggers three promises of "p". According to the previous explanation, the `raise` would translate to code that puts effects on three of the queues associated with promise `p`. However, what we know is that when we are returning a value to fulfill the promise, we only know the task from which the promise is invoked. As such we know only the promises that we need to fulfill. If we fulfill all three promises right away, by putting values onto the associated queues, then we would have other two effects that would fulfill the 3 promises two more times. So the number of the raised effects and fulfilled promises does not match. If we just send the effect only once, as a special case. Then, while properly fulfilling promises only once, the code associated with the promise would also execute only once, instead of the expected three. For that reason, EmbedFX generates a new instance of a task with a queue for each awaited promise. That way the code associated with the promise and the awaited promises are fulfilled exactly right number of times.

5.5 EmbedFX "await" to C

Another primitive that is provided by asynchronous effects calculus is a synchronization primitive `await`. It accepts a promise and blocks the computation until the result becomes available. The following code in EmbedFX:

```
toplevel [
  task "B" [
    e @@ await "p";
  ]
]
```

Is converted to this code in C:

```
int new_local_2;
while(true){
  if((xQueueReceive( queue_p_B_0
                   , &(new_local_2)
                   , (TickType_t)(QUEUE_p_0_WAIT_TICKS)))
      ==
      (pdTRUE))
  {
    break;
  }
}
```

The code is very similar to the awaiting of effects when generating promises. The difference is that we have different queues. Each `await` is annotated with a unique identifier, which helps to distinguish different `await` locations.

5.6 EmbedFX "parallel" to C

In asynchronous effects calculus, the propagation of effects is the crucial aspect. The communication of effects is dictated by the parallel computation (||) primitive.

|| is a binary operator which combines two parallel or asynchronous computations. It plays a crucial role in dictating the propagation of effects and communication. In EmbedFX instead of || there is a `parallel` toplevel primitive, which tells the compiler the tasks between which the communication may occur:

```
toplevel [
  parallel [
    "A"; "B"; "C"; "D"
  ]
  parallel [
    "A"; "E"
  ]
]
```

Such list tells that A, B, C and D all can send each other effects. However, only A can send messages to E.

parallel primitive is used only during the type checking and static analysis and does not generate any specific C code.

5.7 EmbedFX "interrupt" to C

Just like parallel, interrupt is solely an EmbedFX construct which is necessary for the type-checking and effect propagation purposes and as such, does not produce any C code. interrupt is an EmbedFX primitive that represents a single task that simply raises an effect. It can be used to introduce signals from outside of EmbedFX. For example, the following code:

```
interrupt "I" "E";
```

Generates the following comment:

```
/*
    Need to send the following effect: E
    pos_0 TASKS_ENUM_TYPE with TASK_ENUM_VALUE_I
  Need to trigger the following queues:
  queue_p_A_0_r
*/
```

The comment provides the necessary information:

- Effect that needs to be constructed
- Data types of each of the struct fields. In case of pos_0 it also provides the the value that it needs to have
- The queues on which the data must be put

5.8 Calculating effect propagation

In the previous subsections, the process behind the translation of EmbedFX to C was explained. However, some parts of the translation process hinge on one crucial assumption: we know the sources and destinations of raised effects. As explained before, while the asynchronous effects calculus' effect propagation can be simulated in an unrestricted environment, we have to make concessions to achieve a similar effect in C without dynamic memory allocation.

This subsection describes the algorithm for calculating the origin of effects and their subsequent destinations, which is enough to achieve the propagation of effects throughout the code.

To calculate the propagation of the effects, for each task we need to know what the raised effects in the task are and what are the promises in other tasks that may be listening to it. The algorithm is:

1. Repeat for each task:
   (a) For the current task obtain the list of other tasks
   (b) For the current task obtain the list of raised effects (including inside nested promises)
   (c) For each other task obtain the list of promises (including nested promises)
   (d) For each raised effect in the *current* task find a list of promises in other tasks that are:
      i. Reachable
      ii. Triggered by the current raised effect
      iii. Additionally store the if is being awaited
   (e) Each such triggered promise is collected in the list
2. Collect the information for each task in a structure

This information is enough to calculate the other necessary information: for each promise, the list of effects, along with their origins and subsequent destinations.

```
Data: all tasks: list of all tasks with promises and raised effects
Result: How to calculate the propagation of effects
for cur_task in all tasks do
    other_tasks ← all tasks − cur_task;
    effects ← effects from cur_task;
    promises ← promises from other_tasks;
    for eff in effects do
        for p in promises do
            if triggers(eff, p) and reachable(eff, p) then
                save eff and p;
            end
        end
    end
end
```

With this information, it is possible to implement the propagation of effects using queues. The above-mentioned algorithms work under several assumptions.

1. Promises are toplevel constructs
2. Defined tasks are static
3. Functions `promises_in_task` and `effects_in_task` return ALL promises and raised effects, even the nested ones
4. There is no recursion among promises

5. `reachable` is a function that checks whether two tasks can communicate
6. `triggers` is a function that tells whether a promise listens to this specific effect

Chapter 6

Evaluation

This chapter evaluates EmbedFX from different perspectives, such as usability, correctness, expressivity and consumed resources. While the previous chapters have explained the design of EmbedFX and the implementation, showing that the asynchronous effects are feasible to some extent on embedded microcontrollers and without a dynamic memory allocation, this chapter investigates the suitability of asynchronous effects for writing software for deeply embedded microcontrollers.

The evaluation was performed with ESP32-WROOM-32D and ESP-IDF, version 5.3 (commit `a0f798cfc4bbd624aab52b2c194d219e242d80c1`). In the setting of the project (to build and flash to ESP32), the target for ESP-IDF was set as “`esp32`”, the interrupt watchdogs and task watchdog timers were turned off for all of the builds. To get the runtime stats:

- “`configGENERATE_RUN_TIME_STATS`” was set
- “`configRUN_TIME_COUNTER_TYPE`” was set to “`uint32_t`”
- Tick timer sources was set as `Timer 0 (int 6, level 1)`
- Clock source for run time stats was set as “`ESP TIMER`”

EmbedFX was compiled with OCaml 5.2.0. C code was compiled with GCC version 14.2.1 20240912 (Red Hat 14.2.1-3) (GCC). The bootloader was optimized for size (-Os), whereas the code was optimized to provide debug information (-Og). While different microcontrollers and different RTOSes may provide different experiences, EmbedFX evaluation was performed on ESP32 with FreeRTOS SMP which is a part of ESP-IDF. The reason ESP32 was chosen because it is a fairly cheap, but also reliable microcontroller that is often used in the industry. While different RTOS can be used with ESP32, FreeRTOS is most compatible with it, since ESP32 has two cores and ESP-IDF provides a FreeRTOS SMP, which was modified specifically to take advantage of two cores mentioned. FreeRTOS additionally is a widely used RTOS that is efficient and tested in many situations.

6.1 EmbedFX provides reliable asynchronous effects on microcontrollers

When running the examples of EmbedFX-generated code on ESP32, it ran according to the intuition of asynchronous effects. EmbedFX analyzed the propagation of effects and created necessary queues, ensuring the proper communication between tasks and automating task creation. When changing the code, EmbedFX automatically updates the structure of the software and adds or removes tasks and queues on demand, ensuring consistency. When editing C code by hand, those details must be taken care of by the users themselves and are sources of problems.

6.2 EmbedFX decreases the number of lines

Throughout the generated code examples, EmbedFX code was consistently less than that of C, with the minimum ratio achieved at 2.18 and the highest achieved at 10.95. There are several reasons, such as EmbedFX being implemented as an OCaml library, which allows to use OCaml functions to generate code as a macro, which, if applied properly, decreases the code written in EmbedFX. On the other hand, code written in C is low-level and requires many things to work properly: include directives, long function names to avoid name collision, etc. On top of that, to represent the asynchronous effects, EmbedFX generates many intermediate variables and data structures, structs, as well as some macros for usability. While the code written by hand could be about the same length, or even shorter than EmbedFX, to represent the semantics of asynchronous effects in C, it requires many lines of code similar to that generated by EmbedFX. To some extent, EmbedFX is more concise than C. The ratio of C LoC over EmbedFX LoC:

	Example name	C code LoC	EmbedFX code LoC	Ratio
1	eff	21	5	4.20
2	two_tasks	45	12	3.75
3	two_effects_and_a_promise	89	23	3.87
4	await	99	18	5.50
5	await_2	173	37	4.68
6	interrupt	84	14	6.00
7	parallels	219	20	10.95
8	simple_promise	117	21	5.57
9	promise_in_promise	175	25	7.00
10	promise_in_promise_2	183	26	7.04
11	pseudo_coroutines	475	119	3.99
12	simple_aircon	278	78	3.56
13	random_sleep	74	34	2.18
14	receive_from_many	416	92	4.52
15	voting	827	128	6.46

Table 6.1: Lines of code data

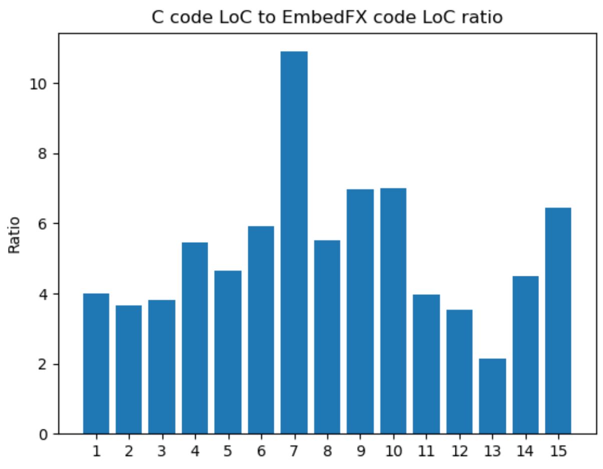

Figure 6.1: C code LoC to EmbedFX code LoC ratio graph

6.3 Heap consumption of EmbedFX

The heap consumption of EmbedFX was calculated. ESP-IDF does not provide methods to get the memory consumed, only `esp_get_free_heap_size()` function which returns the number of bytes available. As such, to calculate the memory consumed, the available heap size from baseline code that does no heap allocation was taken as a total heap size, and for the remaining examples their free heap The data is in the table below.

Example	Occupied Heap (in bytes)
await_2	23312
await	13992
eff	0
parallels	28320
promise_in_promise_2	27976
promise_in_promise	27976
pseudo_coroutines	42764
random_sleep	9300
receive_from_many	47688
simple_aircon	51256
simple_promise	18536
two_effects_and_a_promise	13816
two_task	9096
voting	76464

Table 6.2: Memory consumption of heap by EmbedFX

6.4 EmbedFX heavily underutilizes the CPU

Another useful metric of a language targeting deeply embedded microcontrollers is the CPU load. While it is difficult to measure the performance, as it is heavily dependent on many variables, the significant reason is that it is just not clear how the performance should be measured experimentally, FreeRTOS and ESP-IDF provide some utilities for debugging purposes that can provide some insight. One such utility is `vTaskGetRunTimeStats()`. This function takes the data recorded by the software running starting from the beginning and provides the accumulated statistics, such as tasks, the number of timer ticks they ran, and the percentage of the total timeshare.

As a result, running examples with allocated tasks showed that EmbedFX consistently underutilized the CPU. What the data showed is that both CPUs of ESP32 spend almost entirely in "IDLE" states. For the other tasks running throughout the experiments, the percentage was "$< 1\%$".

This is a result of how EmbedFX generates code. In all of the tasks, EmbedFX generates "`vTaskDelay(10)`", which forces the task to go to sleep for FreeRTOS 10 ticks. This means that code can be tuned to better utilize CPU, which may be better for programs with intensive calculations, or it can be optimized for "slower" programs that do not try to run aggressively and only "wake up" on interrupts or other triggers.

6.5 EmbedFX not suitable for real-time communication

While asynchronous effects can be used to describe programs that rely on asynchronous communication, they may not be suitable for writing programs that need to rely on real-time data. For example the simple air conditioner example, the initial code was written to receive the outside temperature using asynchronous effects. What this resulted in, however, is that the code was making decisions based on outdated information. For example, when the target temperature is set to 17, the air conditioner would go below that, down to almost 1 degree. The reason for that is that the code didn't take into account the changes to the temperature that were happening *not* because of airconditioner. So when the temperature was clearly below the target, the decision to lower the temperature was being made based on the effect when the temperature was *above* the target. When the code was changed to use global variables to store the current outside temperature, the code started to work as expected.

Chapter 7

Discussion

7.1 Challenges of Asynchronous Effects for Deeply Embedded Systems

7.1.1 EmbedFX creates many tasks and queues

To accurately implement asynchronous effects using tasks and queues, EmbedFX generates many tasks and queues.

Their total consumed memory can be roughly calculated in this way:

$$\text{Total memory} = T_T + T_P$$

where, T_T is a total memory consumed by tasks and T_P is a total memory consumed by T_P.

$$T_T = \sum_{i=1}^{N_T} S_{t_i}$$

where N_T is the total number of tasks and S_{t_i} is the stack size of task t_i.

$$T_P = \sum_{i=1}^{N_P}((\#\text{ of tasks } p_i \text{ occurs in})*(S_{p_i}+Q_{\text{receive}_{p_i}}))+((\#\text{ of awaits of } p_i)*(S_{p_i}+Q_{\text{receive}_{p_i}}+Q_{\text{send}_{p_i}}))$$

where N_P is a total number of promises, S_{p_i} is a stack size of a promise p_i, $Q_{\text{receive}_{p_i}}$ is size of a queue that receives effects, and $Q_{\text{send}_{p_i}}$ is the size of a queue that fulfills the promise p_i.

What this tells us is that actions such as adding tasks, using promises in different tasks, awaiting on promises and re-using promises in other promises quickly increases memory consumption.

While the memory consumption of EmbedFX should be further optimized, it is important to note that:

- Heap consumption is constant
- Heap consumption can be reasoned about

When running examples and observing the consumed memory, the heap consumption remained constant. Additionally, the examples that had more tasks, promises and queues consumed more memory. EmbedFX avoids dynamic memory allocation, so the consumed memory is directly related to the amount of stack size of tasks and queue sizes.

7.1.2 EmbedFX and Real-time Communication

EmbedFX employs asynchronous effects for *asynchronous communication*, based on FreeRTOS. However RTOS are often used for scheduling of periodic tasks. Asynchronous effects are not equipped with reasoning about timing constraints, and for that reason, EmbedFX needs to be extended to support wider range of software with timing requirements. This remains a future work.

7.1.3 Uninstalling promises

The semantics of asynchronous effects have proven to effectively abstract over the FreeRTOS API, such as `xTaskCreate`, `xQueueSend` and `xQueueReceive`. However, to achieve such results, several important restrictions have to be put in place:

- Tasks and promises are a toplevel construct, instead of first-class-citizen
- Tasks cannot be dynamically created
- Functions and promises cannot have recursion

This results in limited expressivity, which makes writing some programs more difficult.

One of such challenges is the difficulty of ignoring some effects. Since each promise is represented as a task, and since no tasks are being deleted, it means that each promise ALWAYS listens for a specific signal and we cannot easily represent programs that may require the ignoring of specific signals.

Let's imagine a scenario where we have a microcontroller that reads from two devices: a card reader and door sensor. The card reader is a simple key mechanism that opens the door if the key is correct. The door sensor on the other hand tells whether the door has been closed or opened.

In the software, the events such as the keycard and opening/closing doors can be modeled as effects. One example program that can be written in this case is this:

```
toplevel [
  effect "DoorOpen"
]
```

The problem would result in case if we want to have a following property:

- while the door is opened, ignore signals from the keycard.

This is not a problem in the original asynchronous effects calculus, because it is possible to write a code that effectively ignores the signals.

```
let rec f () = promise (ClosedDoor -> g ()) as p in p
and g = promise (CardReader -> f ()) as p in p
```

What this code effectively does is switch between two different "modes" using recursion. As such, when the effect `CardReader` reaches a promise that is currently listening for `ClosedDoor` effect, it disappears without triggering the promise and thus, the signal is effectively ignored.

One can introduce a notion of “modes”: a list of tasks that are known to run together at specific moment in time. When the “mode switch” occurs, one can generate a code that deletes certain tasks and reinstalls other tasks with other promises. This is one such approach where the higher expressivity of the asynchronous effects on embedded systems can be achieved.

The problem with this approach, however, is that it is difficult to nail down the proper semantics of such mechanism. It does not naturally lend itself to one or the other specific behavior, as the asynchronous effects with the FreeRTOS API. As such, future work can improve the expressivity of the asynchronous effects for embedded systems.

7.1.4 Closures and State

Another important aspect of embedded software is having a state. In the current implementation, the tasks are modeled after asynchronous computations, and designed to mimic the tasks often found in FreeRTOS. One of the concessions made was the disallowing of the state within tasks. While in C one can easily have local variables defined within the tasks and/or pass specific arguments to the task via one of the arguments to `xTaskCreate`, creation of a “hidden” state within the task was deemed to unnecessarily complicate the semantics and obscure the intention of the written code. However, the addition of the innate state to the tasks that transgresses through the iterations is an available option for further expansion of the EmbedFX. Currently, the only way one can achieve stateful computations is through the global variables.

To reduce the dependence on global variables, there is a need to provide flexible, yet efficient closures. This addition in turn complicates the design of EmbedFX. To achieve closures, there are two ways:

1. Allocate closure structure on the heap
2. Create closure structure on the stack

Allocating closure on the heap has the benefit of making it globally accessible and consistent from different parts of the code. The drawback is the allocation on the heap: in deeply embedded systems dynamic memory allocation is undesirable. But even in the contexts when such dynamic allocation is possible, i.e. more powerful microcontroller, the dynamically allocated closures need to be eventually freed. One of the possible solutions is having a garbage collector, but once again, in deeply embedded contexts it may be highly prohibitive in terms of resource consumption and other constraints. Allocating closures on stacks may be more plausible and desirable, but introduces other problems. The closures need to be properly propagated along with the associated computations. Secondly is the synchronization of closures: if two tasks were to receive a computation with a closure and perform different actions, there would be two copies of the closures. In this case, the closures need to be reconciled, or the semantics of closures would need to be clearly defined, possibly with some restrictions. However, out of the two methods, the allocation on the stack may be more promising. Implementation of closures will unlock higher-order functions, which will help to express programs that utilize function pointers in C, such as callback functions.

7.1.5 Practical Limitations of EmbedFX

EmbedFX targets the FreeRTOS API. While during the development, the assumptions tried to be kept to the minimum, the current implementation has only been tested on ESP32-WROOM-32D with the latest stable ESP-IDF development toolset version 5.3. As such, EmbedFX currently may encode more assumptions about the ESP32 and its FreeRTOS SMP rather than FreeRTOS in general.

While developing EmbedFX, the focus was solely on FreeRTOS API and its correctness. In real-world settings, however, one must closely work with other aspects of microcontrollers, such as interrupts and watchdogs. Watchdogs are specific timers that in case of overflow trigger specific interrupts, which in turn trigger interrupt request handlers. That code usually has to do with the possible error recovery, or dumping the state of the registers, or some other code that may be relevant to the context. However, because of them, the code that is a valid use-case of asynchronous effects may inadvertently trigger watchdogs. As such, the examples that have been developed were run with interrupts and task watchdogs turned off. Extending the EmbedFX to take watchdogs into account and evaluate their effect on programming with asynchronous effects will prove useful for making EmbedFX highly practical.

Since EmbedFX compiles to C with FreeRTOS, in theory it should be possible to use EmbedFX to represent the software in C with FreeRTOS, e.g. Internet-of-Things, etc. However, there are several obstacles, described in the other sections of this chapter, which go into research problems that need to be solved for EmbedFX to fully represent embedded software.

7.1.6 Stack Overflows is a serious problem

When developing EmbedFX, the most consistent problem was the small task stack size. If inside a task with a small stack size a function from LibC, such as `printf` is called, then it will easily overflow the stack, causing the software to inexplicably crash. One way in which EmbedFX can be significantly improved is to provide utilities that will automatically calculate the necessary stack size depending on the procedures and other code invoked from within the task.

7.1.7 Data Consistency Problem

Last, but not the least, to make the asynchronous effects easily accessible and useful, the data consistency problem needs to be resolved. This is not a problem in the general sense of asynchronous effects, because such structures can be allocated on heap and pointers to that data can be effectively passed around. In the current implementation of EmbedFX, the implementation of arrays is difficult. Namely, trying to raise effects that contain arrays. This is because in C, data with unknown size is passed as pointers. As such, it is quite easy to pass a pointer instead of a copy of the data itself. The danger of this approach is due to asynchronous context: by the time the pointer is dereferenced, the data may have been changed, or completely dereferenced, which would make the pointer inconsistent. One of the solutions is to have a unique struct for each (Data-type $\times$ Size) pair, i.e. `ArrayInt8` or `ArrayChar7`. However, it would mean that the arrays with different sizes are different types. This would necessitate to have a more flexible type system with subtyping (e.g. `ArrayInt7` is subtype of `ArrayInt8`). A simpler approach would have a consistent data structure with the maximum size available, but this would quickly result in wasted memory, which may be unacceptable for some resource constrained microcontrollers. But even in that case, there is a need to have an unsized array, or arrays, for which the size is unknown on the type level. Those can only be implemented as pointers to arrays, which would resurface the problem of inconsistent data and invalid pointers. The implementation of arrays in EmbedFX is currently stubbed at the code generation part. For the moment, the only meaningful way to work with arrays would be to have a global array, access to which is highly coordinated. Passing arrays via effects is currently impossible.

7.2 Other programming languages for Embedded Systems

7.2.1 Ivory

Ivory is a deep Domain-Specific Language (DSL) embedded in Haskell's type system [6].

Ivory was designed specifically for deeply embedded systems, and attempts to improve upon C, namely, address C's memory-related issues. Ivory redefines C's memory model and defines the proper semantics of using that memory with Haskell's type system. The code written in Ivory is type-checked for correctness with Haskell's type-checker. With Ivory's semantics, it is impossible to dereference a null pointer or go outside of the array bounds.

As a Quality-of-Life improvement, Ivory additionally provides support of contract-oriented-programming (similar to Ada), an ability to define pre-conditions and post-conditions on procedures. Those pre-and-post-conditions can be later exported as an algebraic formula and solved by SMT-solver.

Ivory is a significant contribution to the embedded systems' ecosystem. Unfortunately, it is no longer actively developed. Another problem is that while Ivory is a great tool, it requires getting used to it. There are no learning materials available, which means that to learn to use Ivory it is necessary to read a few examples that are available and read Ivory source code, which is written in Haskell. To use and understand Ivory, it is necessary to have some understanding of Haskell, and by extension monadic computations, upon which the Ivory is built, which is a hard-sell for an ordinary embedded systems engineer.

Nevertheless, Ivory is a step in the right direction. The current thesis was inspired by the ideas and philosophy of Ivory.

7.2.2 Emfrp

Emfrp is a Functional Reactive Programming (FRP) language for deeply embedded systems [21]. It attempts to bring the heavy FRP abstractions to low-end microcontrollers while preserving the causality (current events cannot depend on future events) and preventing space leaks.

With each new iteration, the requirements on Emfrp are slowly relaxed, such as independent nodes having different frequencies [26] or introducing limited recursion [25]. Emfrp is currently being actively developed.

FRP is a powerful, and if used properly, concise, yet expressive abstraction. The current thesis was inspired by Emfrp but tries to take a slightly different approach to its foundational blocks.

7.2.3 Giotto

Giotto is another programming language for embedded systems. The main specialty of Giotto is that it is a "time-triggered" programming language [7]. As said before [9], programming languages for deeply embedded systems need to provide special abstractions which reflect the challenges of embedded systems as well as timing constraints. Giotto attempts to address precisely that.

The code written in Giotto is separated into two parts:

1. Platform-independent code and timing constraints: at this level, the developers specify the functionality of their code, as well as the timing requirements, which can be later verified.

2. Low-level code: at this level, the developers provide all the platform-specific code and deal with the communications aspects of the program.

Giotto is an important contribution to the programming languages for embedded systems, especially for writing the safety-critical code where the hard deadlines need to be respected.

7.3 Other implementations of Asynchronous Effects

At the moment of writing, there are currently two other implementations of asynchronous effects as were defined by Ahman and Pretnar[3]: Æff[3] and Leo Poulson's asynchronous effects in Frank[17].

7.3.1 Æff

Æff is the original implementation of asynchronous effects provided by Ahman and Pretnar. It supports the the original operational semantics of asynchronous effects by implementing them in an interpreter which performs step-by-step evaluation. The non-deterministic behavior is achieved through random reduction to one of the terms when several choices are available. The original intent of Æff is to illustrate the operational semantics of asynchronous effects. As such, their system is considered closed and it does not accept signals from the outside.

7.3.2 Asynchronous Effects in Frank

In his Master's Thesis, Leo Poulson implemented asynchronous effects in a Frank programming language. Frank[12] is a research language that supports effect handlers. It was designed to implement algebraic effect handlers similar to their definition by Plotkin and Pretnar [16].

In his work, Leo Poulson implements asynchronous effects in Frank's type system as a library. The main principle is to use effect handlers to represent asynchronous effects. However, since the effect handlers are synchronous by nature, Leo Poulson modifies the operational semantics of Frank, to introduce a mandatory context switch that gives CPU processing time to other threads. Since he is trying to fit asynchronous effects into Frank's type system, in action they behave slightly different, compared to original Æff.

7.3.3 Comparsion with EmbedFX

While all of the languages have been developed as a result of research effort, there are differences:

- Æff is the only implementation that fully implements the operational semantics in their original form. Other languages modify them during implementation.
- Æff interprets effect propagation using step-by-step evaluation, in Leo Poulson's work a signal triggers all promises that listen to that signal. EmbedFX calculates the propagation at compile time and uses queues for communication.
- EmbedFX is heavily restricted to simulate asynchronous effects on deeply embedded microcontrollers. As one of the effects, recursion is prohibited in EmbedFX. Æff and Frank both allow recursion, but cannot run in the contexts without dynamic memory allocation.
- EmbedFX allows to define incoming signals from outside of the system (using `interrupt` primitive), whereas Æff and Frank do not.

7.4 Comparison of EmbedFX with other languages

EmbedFX achieves implementing asynchronous effects on deeply embedded microcontrollers. As such, it was compared with other existing implementations of asynchronous effects. However, there are other languages that support asynchronicity, such as Go and Rust. No such evaluation has been performed, and as such, comparing EmbedFX with those languages for deeply embedded microcontrollers is another possible future work.

Chapter 8

Conclusion

This thesis introduced EmbedFX, a programming language for deeply embedded microcontrollers that abstracts the FreeRTOS API using asynchronous effects. This is the first consideration of asynchronous effects for deeply embedded microcontrollers to our knowledge. The thesis showed the feasibility of asynchronous effects in conditions without dynamic memory allocation, albeit with constraints on the expressivity of the original asynchronous effects calculus. The code produced by EmbedFX was successfully executed on ESP32 and was consistent with the operational semantics and the intuition of the asynchronous communication. The experiments have shown that while there is an opportunity for further optimization of runtime representation of asynchronous effects for better resource utilization, memory consumption can be to some extent reasoned about. The experiments have also shown the current implementation of EmbedFX underutilizes the CPUs of ESP32, which means that there is room for more efficient scheduling of tasks or that asynchronous effects can be used to represent programs that "sleep" most of the lifecycle. Lastly, the current thesis showed that while asynchronous effects can be used to represent some programs in embedded systems, are unsuitable for situations in which the programs need to operate on real-time data.

Acknowledgment

This Master's Thesis would not have been possible without the invaluable guidance and profound wisdom of Professor Yukiyoshi Kameyama. I am deeply indebted to him for his mentorship and unwavering support throughout this journey.

I would also like to express my sincere appreciation to Ahman and Pretnar for their foundational contributions to the asynchronous effects theory, which served as a cornerstone for this research.

I feel grateful to Professors Ohyama Yoshihiro and Abe Hirotake for their invaluable comments of this research, and to my parents who encouraged me on every step of my life which eventually lead to this.

Finally, I extend my heartfelt gratitude to the Ministry of Education, Culture, Sports, Science and Technology for enabling me to pursue this research through their generous support and resources.

Appendix A

EmbedFX Examples

Await example:

```
let await_example =
  toplevel [
    eff "E" [];
    promise "p" ("E", []) [
      return (_i 2);
    ];
    task "A" [
      e @@ raise "E" [];
    ];
    task "B" [
      _let "x" (await "p");
      e @@ unit;
    ];

    parallel [
      "A"; "B"
    ];

    _func void "app_main" [] [
    ];
  ]
```

Await example (2):

```
let await_2_example =
  let printfn s = e @@ _func_call "printf" [_s (s ^ "\n")] in
  let delay i = e @@ _func_call "vTaskDelay" [_i i] in
  toplevel [
    _func_type "printf" [str] void;
    _func_type "vTaskDelay" [_int] void;

    eff "E" [];
    eff "F" [];

    promise "p_E" ("E", []) [
      printfn "Triggered on E";
      return (_b true);
    ];

    promise "p_F" ("F", []) [
      printfn "Triggered on F";
      return (_b true);
    ];

    task "sender" [
      printfn "Sending E";
      raise' "E" [];
      delay 50;
    ];

    task "await_E" [
      e @@ await "p_E";
      printfn "p_E resolved, sending F";
      raise' "F" [];
    ];

    task "await_F" [
      e @@ await "p_F";
      printfn "p_F resolved";
    ];

    parallel [
      "sender";
      "await_E";
      "await_F";
    ];
    _func void "app_main" [] [
    ];
  ]
```

Effect:

```
let eff_example =
  toplevel [
    eff "E" [_char; _bool];

    _func void "app_main" [] [
    ];
  ]
```

Interrupt example:

```
let interrupt_example =
  toplevel [
    eff "E" [];

    interrupt "I" "E";
    promise "p" ("E", []) [
      return (_i 2);
    ];

    task "A" [
      _let "x" (await "p");
      e @@ unit;
    ];

    parallel ["A"; "I"];

    _func void "app_main" [] [
    ];
  ]
```

Parallels example:

```
let parallels_example =
  let body =
    [
      e @@ raise "E" [];
      e @@ await "p";
      e @@ unit;
    ]
  in
  toplevel [
    eff "E" [];

    promise "p" ("E", []) [
      return (_i 2);
    ];

    task "A" body;
    task "B" body;
    task "C" body;

    parallel ["A"; "B"];
    parallel ["A"; "C"];

    _func void "app_main" [] [
    ];
  ]
```

Promise in proimse example:

```
let promise_in_promise_example =
  let printf s = _func_call "printf" [_s s] in
  let printf’ s = e @@ printf s in
  let _task x =
    task x [
      e @@ raise "E" [];
      e @@ id "p1";
      printf’ @@ Printf.sprintf "From task %s" x;
    ]
  in
  toplevel [
    _func_type "printf" [str] void;
    eff "E" [];

    promise "p2" ("E", []) [
      printf’ "Printing from promise p2";
    ];

    promise "p1" ("E", []) [
      e @@ id "p2";
      printf’ "Printing from promise p1";
    ];

    _task "A";
    _task "B";

    parallel [ "A"; "B" ];

    _func void "app_main" [] [
    ];
  ]
```

Promise in promise example (2):

```
let promise_in_promise_2_example =
  let printf s = _func_call "printf" [_s s] in
  let printf' s = e @@ printf s in
  let _task x =
    task x [
      e @@ raise "E" [];
      e @@ id "p1";
      printf' @@ Printf.sprintf "From task %s" x;
    ]
  in
  toplevel [
    _func_type "printf" [str] void;
    eff "E" [];

    promise "p2" ("E", []) [
      printf' "Printing from promise p2";
    ];

    promise "p1" ("E", []) [
      e @@ id "p2";
      printf' "Printing from promise p1";
      e @@ id "p2";
    ];

    _task "A";
    _task "B";

    parallel [ "A"; "B" ];

    _func void "app_main" [] [
    ];
  ]
```

Pseudo-coroutines example:

```
let pseudo_coroutines_example =
  let tree_A = _func_call "pass_tree_A" [] in
  let tree_B = _func_call "pass_tree_B" [] in
  let sleep x = e @@ _func_call "vTaskDelay" [CastTo(CustomType("TickType_t"), _i x)] in
  let gen_task c f =
    task (Printf.sprintf "traverse_tree_%c" c) [
      global "should_search";
      _if' ((id "should_search") === (_b true)) [
        e @@ raise (Printf.sprintf "tree_%c" c) ([f]);
      ] [
        _while' (_b true) [
        ]
      ];
      sleep 10;
    ];
  in
  let puts s = e @@ _func_call "puts" [_s s] in
  let printfcc s a b = e @@ _func_call "printf" [_s (s ^ "\n"); (id a); (id b)] in
  let halt = _while' (_b true) [] in
  let _lxor a b =
    ((a === (_b true )) |><| (b === (_b false))) |<>|
    ((a === (_b false)) |><| (b === (_b true )))
  in
  toplevel [
    eff "tree_A" [_char];
    eff "tree_B" [_char];
    eff "read_A" [_bool];
    eff "read_B" [_bool];
    eff "should_search_eff" [_bool];

    interrupt "read_A_i" "read_A";
    interrupt "read_B_i" "read_B";

    _func_type "pass_tree_A" [] _char;
    _func_type "pass_tree_B" [] _char;
    _func_type "vTaskDelay" [CustomType("TickType_t")] void;
    _func_type "puts" [str] void;
    _func_type "printf" [str; _char; _char] void;

    global_var _bool "should_search";
    global_var _bool "should_search_A";
    global_var _bool "should_search_B";
    global_var _bool "trees_match";

    promise "sig_A" ("tree_A", ["x"]) [
      return (id "x");
    ];

    promise "sig_B" ("tree_B", ["x"]) [
      return (id "x");
    ];

    promise "p_should_read_A" ("read_A", ["v"]) [
      return (id "v");
    ];

    promise "p_should_read_B" ("read_B", ["v"]) [
      return (id "v");
    ];

    promise "p_should_search" ("should_search_eff", ["v"]) [
      return (id "v");
    ];

    task "wait_for_read_signal" [
      global "should_search";
      global "trees_match";
      _if' ((id "should_search") === (_b true)) [
        _let "a" (await "p_should_read_A");
        _let "b" (await "p_should_read_B");
        _if' (_lxor (id "a") (id "b")) [
          (id "trees_match") |= (_b false);
        ][
        ];
        _if' (((id "a") |><| (id "b")) === (_b false)) [
          e @@ raise "should_search_eff" [(_b false)];
          halt
        ][
          e @@ raise "should_search_eff" [(_b true)];
        ]
      ] [
        halt
      ]
    ];
```

```
gen_task 'A' tree_A;
gen_task 'B' tree_B;

task "main_task" [
  global "should_search";
  global "trees_match";
  _if' ((id "should_search") === (_b false)) [
    puts "Trees are equal!";
    halt
  ] [
    _let "a" (await "sig_A");
    _let "b" (await "sig_B");
    _if' ((id "a") === (id "b")) [
      _if' ((id "trees_match") === (_b false)) [
        puts "Trees are not equal! One is shorter than the other!";
        halt
      ][
      ];
      _let "v" (await "p_should_search");
      (id "should_search") |= (id "v");
    ] [
      printfcc "Trees are not the same! %c != %c" ("a") ("b");
      (id "should_search") |= (_b false);
      halt
    ]
  ]
];

parallel [ "main_task"
         ; "traverse_tree_A"
         ; "traverse_tree_B"
         ];
parallel [ "read_A_i"
         ; "read_B_i"
         ; "wait_for_read_signal"
         ];
parallel [ "main_task"; "wait_for_read_signal" ];

_func void "app_main" [] [
  global "should_search";
  global "should_search_A";
  global "should_search_B";
  global "trees_match";
```

```
      (id "should_search")   |= (_b true);
      (id "should_search_A") |= (_b true);
      (id "should_search_B") |= (_b true);
      (id "trees_match")     |= (_b true);
    ]
  ]
```

Helper C code:

```
// good_tree.h

#ifndef TREE_H
#define TREE_H

extern bool should_search_A;
extern bool should_search_B;

/*
    Need to send the following effect: read_A
    pos_0 TASKS_ENUM_TYPE with TASK_ENUM_VALUE_read_A_i
    pos_1 Bool
  Need to trigger the following queues:
  queue_p_should_read_A_wait_for_read_signal_0_r
*/

int tree_a_count = 0;
char pass_tree_A(void){
  char a[] = { 'A', 'B', 'C', 'D', 'E' };
  if( should_search_A && tree_a_count < 5 ){
    tree_a_count ++;
    if( tree_a_count == 5 ){
      should_search_A = false;
    }
    read_A eff = { .pos_0 = TASK_ENUM_VALUE_read_A_i, .pos_1 = should_search_A };
    xQueueSend(queue_p_should_read_A_wait_for_read_signal_0_r, &(eff), (TickType_t)(10));
    return a[tree_a_count - 1];
  }
  return '\0';
}

/*
    Need to send the following effect: read_B
    pos_0 TASKS_ENUM_TYPE with TASK_ENUM_VALUE_read_B_i
    pos_1 Bool
  Need to trigger the following queues:
  queue_p_should_read_B_wait_for_read_signal_1_r
*/

int tree_b_count = 0;
char pass_tree_B(void){
  char b[] = { 'A', 'B', 'C', 'D', 'E' };
  if( should_search_B && tree_b_count < 5 ){
    tree_b_count ++;
    if( tree_b_count == 5 ){
      should_search_B = false;
    }
    read_B eff = { .pos_0 = TASK_ENUM_VALUE_read_B_i, .pos_1 = should_search_B };
    xQueueSend(queue_p_should_read_B_wait_for_read_signal_1_r, &(eff), (TickType_t)(10));
    return b[tree_b_count - 1];
  }
  return '\0';
}

#endif // TREE_H

// bad_tree_nodes.h
#ifndef TREE_H
#define TREE_H

extern bool should_search_A;
extern bool should_search_B;

/*
    Need to send the following effect: read_A
    pos_0 TASKS_ENUM_TYPE with TASK_ENUM_VALUE_read_A_i
    pos_1 Bool
  Need to trigger the following queues:
  queue_p_should_read_A_wait_for_read_signal_0_r
*/

int tree_a_count = 0;
char pass_tree_A(void){
  char a[] = { 'A', 'B', 'C', 'D', 'E' };
  if( should_search_A && tree_a_count < 5 ){
    tree_a_count ++;
    if( tree_a_count == 5 ){
      should_search_A = false;
    }
    read_A eff = { .pos_0 = TASK_ENUM_VALUE_read_A_i, .pos_1 = should_search_A };
    xQueueSend(queue_p_should_read_A_wait_for_read_signal_0_r, &(eff), (TickType_t)(10));
    return a[tree_a_count - 1];
  }
  return '\0';
}

/*
    Need to send the following effect: read_B
    pos_0 TASKS_ENUM_TYPE with TASK_ENUM_VALUE_read_B_i
    pos_1 Bool
  Need to trigger the following queues:
  queue_p_should_read_B_wait_for_read_signal_1_r
*/

int tree_b_count = 0;
char pass_tree_B(void){
  char b[] = { 'A', 'B', 'C', 'D', 'F' };
  if( should_search_B && tree_b_count < 5 ){
    tree_b_count ++;
    if( tree_b_count == 5 ){
      should_search_B = false;
    }
    read_B eff = { .pos_0 = TASK_ENUM_VALUE_read_B_i, .pos_1 = should_search_B };
    xQueueSend(queue_p_should_read_B_wait_for_read_signal_1_r, &(eff), (TickType_t)(10));
    return b[tree_b_count - 1];
  }
  return '\0';
}

#endif // TREE_H

// bad_tree_length.h

#ifndef TREE_H
#define TREE_H

extern bool should_search_A;
extern bool should_search_B;

/*
    Need to send the following effect: read_A
    pos_0 TASKS_ENUM_TYPE with TASK_ENUM_VALUE_read_A_i
    pos_1 Bool
  Need to trigger the following queues:
  queue_p_should_read_A_wait_for_read_signal_0_r
*/

int tree_a_count = 0;
char pass_tree_A(void){
  char a[] = { 'A', 'B', 'C', 'D', 'E' };
  if( should_search_A && tree_a_count < 5 ){
    tree_a_count ++;
    if( tree_a_count == 5 ){
      should_search_A = false;
    }
    read_A eff = { .pos_0 = TASK_ENUM_VALUE_read_A_i, .pos_1 = should_search_A };
    xQueueSend(queue_p_should_read_A_wait_for_read_signal_0_r, &(eff), (TickType_t)(10));
    return a[tree_a_count - 1];
  }
  return '\0';
}

/*
    Need to send the following effect: read_B
    pos_0 TASKS_ENUM_TYPE with TASK_ENUM_VALUE_read_B_i
    pos_1 Bool
  Need to trigger the following queues:
  queue_p_should_read_B_wait_for_read_signal_1_r
*/

int tree_b_count = 0;
char pass_tree_B(void){
  char b[] = { 'A', 'B', 'C', 'D', 'E', 'F' };
  if( should_search_B && tree_b_count < 6 ){
    tree_b_count ++;
    if( tree_b_count == 6 ){
      should_search_B = false;
    }
    read_B eff = { .pos_0 = TASK_ENUM_VALUE_read_B_i, .pos_1 = should_search_B };
    xQueueSend(queue_p_should_read_B_wait_for_read_signal_1_r, &(eff), (TickType_t)(10));
    return b[tree_b_count - 1];
  }
  return '\0';
}
```

```
#endif // TREE_H
```

Random sleep example:

```
let random_sleep_example =
  let delay x =
    _func_call' "vTaskDelay" [x]
  in
  let printf s' =
    _func_call' "printf" [_s (s' ^ "\n")]
  in
  let rand_sleep =
    let rand = _func_call "rand" [] in
    [ _let "sleep_time" ((rand |% (_i 30)) |+ (_i 10));
      delay ((id "sleep_time") |* (_i 20));
    ]
  in
  toplevel [
    eff "E" [];

    _func_type "rand" [] _int;
    _func_type "vTaskDelay" [_int] void;
    _func_type "printf" [str] void;

    promise "receive" ("E", []) [
      printf "MSG receiver: received";
    ];

    task "sender" ([
      printf "MSG sender: raising effect...";
      raise' "E" [];
      printf "MSG sender: sleeping for random time...";] @
      rand_sleep);

    task "receiver" [
      e @@ id "receive";
    ];

    parallel [
      "sender";
      "receiver";
    ];

    _func void "app_main" [] [
    ]
  ]
```

Receive from many:

```
let random_sleep_example =
  let delay x =
    _func_call' "vTaskDelay" [x]
  in
  let printf s' =
    _func_call' "printf" [_s (s' ^ "\n")]
  in
  let rand_sleep =
    let rand = _func_call "rand" [] in
    [ _let "sleep_time" ((rand |% (_i 30)) |+ (_i 10));
      delay ((id "sleep_time") |* (_i 20));
    ]
  in
  toplevel [
    eff "E" [];

    _func_type "rand" [] _int;
    _func_type "vTaskDelay" [_int] void;
    _func_type "printf" [str] void;

    promise "receive" ("E", []) [
      printf "MSG receiver: received";
    ];

    task "sender" ([
      printf "MSG sender: raising effect...";
      raise' "E" [];
      printf "MSG sender: sleeping for random time...";] @
      rand_sleep);

    task "receiver" [
      e @@ id "receive";
    ];

    parallel [
      "sender";
      "receiver";
    ];

    _func void "app_main" [] [
    ]
  ]
```

Simple aircon:

```
let simple_aircon_example =
  let inc_temp = e @@ _func_call "inc_temp" [] in
  let dec_temp = e @@ _func_call "dec_temp" [] in
  let printf s = e @@ _func_call "printf" [_s (s ^ "\n")] in
  toplevel [
    eff "cur_temp" [_int];
    eff "set_temp" [_int];
    eff "turn_on_or_off" [];

    _func_type "inc_temp" [] void;
    _func_type "dec_temp" [] void;
    _func_type "printf" [str] void;

    global_var _int "target_temp";
    global_var _bool "switched_on";
    global_var _int "current_temp";

    interrupt "outside_temp_sensor" "cur_temp";
    interrupt "outside_temp_control" "set_temp";
    interrupt "outside_active_switch" "turn_on_or_off";

    promise "p_outside_temp" ("cur_temp", ["v"]) [
      return (id "v");
    ];

    task "fresh_temp_data" [
      global "current_temp";
      _let "x" (await "p_outside_temp");
      (id "current_temp") |= (id "x");
    ];

    promise "p_target_temp" ("set_temp", ["v"]) [
      return (id "v");
    ];

    task "set_target_temp" [
      global "target_temp";
      _let "x" (await "p_target_temp");
      (id "target_temp") |= (id "x");
      printf "Changed the temperature";
    ];
```

```
promise "p_switch" ("turn_on_or_off", []) [
  return (_b true);
];

task "cur_mode_switch" [
  global "switched_on";
  await' "p_switch";
  _if' ((id "switched_on") === (_b true)) [
    (id "switched_on") |= (_b false);
    printf "Switched off";
  ][
    (id "switched_on") |= (_b true);
    printf "Switched on";
  ]
];

task "main_task" [
  global "switched_on";
  global "target_temp";
  global "current_temp";
  _if' ((id "switched_on") === (_b true)) [
    _if' ((id "current_temp") |< (id "target_temp")) [
      inc_temp;
    ][
      _if' ((id "current_temp") |> (id "target_temp")) [
        dec_temp;
      ][
        (* do nothing *)
      ]
    ]
  ][
    (* do nothing *)
  ]
];

parallel [ "outside_temp_sensor"
         ; "fresh_temp_data"
         ];

parallel [ "cur_mode_switch"
         ; "outside_active_switch"
         ];
    parallel [ "outside_temp_control"
             ; "set_target_temp"
             ];

    _func void "app_main" [] [
      global "current_temp";
      (id "current_temp") |= (_i 25);
    ]
  ]
```

Helper code:

```
void inc_temp(void){
  current_temp ++;
  printf("Increased temp by 1\n");
  printf("  Current temp: %d\n", current_temp);
  printf("  Target temp: %d\n", target_temp);
  vTaskDelay((TickType_t)(5));
}

void dec_temp(void){
  current_temp --;
  printf("Decreased temp by 1\n");
  printf("  Current temp: %d\n", current_temp);
  printf("  Target temp: %d\n", target_temp);
  vTaskDelay((TickType_t)(5));
}

/*
    Need to send the following effect: cur_temp
    pos_0 TASKS_ENUM_TYPE with TASK_ENUM_VALUE_outside_temp_sensor
    pos_1 Int
  Need to trigger the following queues:
  queue_p_outside_temp_fresh_temp_data_0_r
*/

TaskHandle_t change_outside_temperature_task_handle;
void change_outside_temperature_task(void) {
  while(1){
    bool should_increase = (rand() % 2 == 1) ? true : false;
    int delta = 0;
    if(should_increase){
      delta = +3;
    } else {
      delta = -3;
    }
    current_temp += delta;
    vTaskDelay((TickType_t)(5000));
  }
}

TaskHandle_t send_outside_temperature_task_handle;
void send_outside_temperature_task(void) {
  while(1){
    cur_temp e = { .pos_0 = TASK_ENUM_VALUE_outside_temp_sensor, .pos_1 = current_temp };
    xQueueSend(queue_p_outside_temp_fresh_temp_data_12_r, &(e), (TickType_t)(10));
    vTaskDelay((TickType_t)(5));
  }
}

/*
    Need to send the following effect: set_temp
    pos_0 TASKS_ENUM_TYPE with TASK_ENUM_VALUE_outside_temp_control
    pos_1 Int
  Need to trigger the following queues:
  queue_p_target_temp_set_target_temp_1_r
*/

TaskHandle_t change_target_temp_task_handle;
void change_target_temp_task(void) {
  while(1){
    int new_temp = (rand() % 10) + 17; // [17, 26]
    target_temp = new_temp;
    printf("Changed target temperature to %d\n", new_temp);
    set_temp e = { .pos_0 = TASK_ENUM_VALUE_outside_temp_control, .pos_1 = new_temp };
    xQueueSend(queue_p_target_temp_set_target_temp_13_r, &(e), (TickType_t)(10));
    vTaskDelay((TickType_t)(500));
  }
}

/*
    Need to send the following effect: turn_on_or_off
    pos_0 TASKS_ENUM_TYPE with TASK_ENUM_VALUE_outside_active_switch
  Need to trigger the following queues:
  queue_p_switch_cur_mode_switch_2_r
*/
```

```
TaskHandle_t turn_on_or_off_task_handle;
void turn_on_or_off_task(void) {
  while(1){
    bool should_switch = (rand() % 2 == 1) ? true : false;
    if(should_switch){
      turn_on_or_off e = { .pos_0 = TASK_ENUM_VALUE_outside_active_switch };
      printf("Toggled the active switch\n");
      xQueueSend(queue_p_switch_cur_mode_switch_14_r, &(e), (TickType_t)(10));
    }
    vTaskDelay((TickType_t)(5000));
  }
}

void setup(void){

  /* EmbedFX genereated code */

  // custom tasks

  xTaskCreate( (TaskFunction_t)(&(change_outside_temperature_task))
             , "change_outside_temperature"
             , 4196
             , NULL
             , (tskIDLE_PRIORITY + 1)
             , &(change_outside_temperature_task_handle)
             );
  xTaskCreate( (TaskFunction_t)(&(send_outside_temperature_task))
             , "send_outside_temperature"
             , 4196
             , NULL
             , (tskIDLE_PRIORITY + 1)
             , &(send_outside_temperature_task_handle)
             );
  xTaskCreate( (TaskFunction_t)(&(change_target_temp_task))
             , "change_target_temp"
             , 4196
             , NULL
             , (tskIDLE_PRIORITY + 1)
             , &(change_target_temp_task_handle)
             );
  xTaskCreate( (TaskFunction_t)(&(turn_on_or_off_task))
             , "turn_on_or_off"
             , 4196
```

```
                , NULL
                , (tskIDLE_PRIORITY + 1)
                , &(turn_on_or_off_task_handle)
                );
}
```

Simple promise example:

```
let simple_promise_example =
  let printf s = _func_call "printf" [_s s] in
  let printf’ s = e @@ printf s in
  let _task x =
    task x [
      e @@ raise "E" [];
      e @@ id "p";
      printf’ @@ Printf.sprintf "From task %s" x;
    ]
  in
  toplevel [
    _func_type "printf" [str] void;
    eff "E" [];

    promise "p" ("E", []) [
      printf’ "Printing from promise p1";
    ];

    _task "A";
    _task "B";

    parallel [ "A"; "B" ];

    _func void "app_main" [] [
    ];
  ]
```

Two effects and a promise:

```
let two_effects_and_a_promise_example =
  toplevel [
    eff "E" [];
    promise "p" ("E", []) [
    ];

    task "sender_A" [
      raise' "E" [];
    ];
    task "sender_B" [
      raise' "E" [];
    ];
    task "receiver" [
      e @@ id "p";
    ];

    parallel [
      "sender_A";
      "sender_B";
      "receiver";
    ];

    _func void "app_main" [] [
    ];
  ]
```

Two tasks:

```
let two_tasks_example =
  let printfn s = e @@ _func_call "printf" [_s (s ^ "\n")] in
  toplevel [
    _func_type "printf" [str] void;
    task "A" [
      printfn "Hello from task A";
    ];
    task "B" [
      printfn "Hello from task B";
    ];
    _func void "app_main" [] [
    ];
  ]
```

Voting example:

```
let voting_example =
  let printfn s' = e @@ _func_call "printf" [_s (s' ^ "\n")] in
  let rand = (_func_call "rand" []) in
  let rand_vote = ((rand |% (_i 3)) |+ (_i 1)) in
  let delay_n n = e @@ _func_call "vTaskDelay" [_i n] in
  let rand_delay =
    [ _let "delay" (((rand |% (_i 5)) |+ (_i 1)) |* (_i 20));
      e @@ _func_call "vTaskDelay" [id "delay"];
    ]
  in
  let voter_task i =
    task (Printf.sprintf "voter_%d" i) ([
      e @@ await "p_start";
      _let "vote" rand_vote;
      ] @ rand_delay @ [
      printfn (Printf.sprintf "Voter %d voted!\n" i);
      raise' (Printf.sprintf "vote_%d" i) [(id "vote")];
    ])
  in
  let voter_promise i =
    promise
      (Printf.sprintf "p_voter_%d" i)
      (Printf.sprintf "vote_%d" i, ["x"]) [
      return (id "x");
    ]
  in
  toplevel [
    _func_type "printf" [str] void;
    _func_type "rand" [] _int;
    _func_type "vTaskDelay" [_int] void;

    global_var _int "choice_1";
    global_var _int "choice_2";
    global_var _int "choice_3";

    eff "vote_1" [_int];
    eff "vote_2" [_int];
    eff "vote_3" [_int];
    eff "vote_4" [_int];
    eff "vote_5" [_int];
    eff "start" [];
```

```
promise "p_start" ("start", []) [
  return (_b true);
];

voter_task 1;
voter_task 2;
voter_task 3;
voter_task 4;
voter_task 5;

voter_promise 1;
voter_promise 2;
voter_promise 3;
voter_promise 4;
voter_promise 5;

task "judge" ([
  global "choice_1";
  global "choice_2";
  global "choice_3";

  (id "choice_1") |= (_i 0);
  (id "choice_2") |= (_i 0);
  (id "choice_3") |= (_i 0);

  printfn "Voting start!";
  raise' "start" [];

  _let "voted_1" (await "p_voter_1");
  _let "voted_2" (await "p_voter_2");
  _let "voted_3" (await "p_voter_3");
  _let "voted_4" (await "p_voter_4");
  _let "voted_5" (await "p_voter_5");
] @ (
  let resolve_vote i =
    let cond j =
      _if' ((id @@ Printf.sprintf "voted_%d" i) === (_i j)) [
        ((id @@ Printf.sprintf "choice_%d" j)
          |=
        ((id @@ Printf.sprintf "choice_%d" j) |+ (_i 1)));
      ] [
      ]
    in
```

```
      List.map cond [ 1; 2; 3 ]
    in
    List.flatten [
      resolve_vote 1;
      resolve_vote 2;
      resolve_vote 3;
      resolve_vote 4;
      resolve_vote 5;
    ]
  ) @ [
    _if' (((id "choice_1") === (id "choice_2")) |><|
          ((id "choice_1") |>  (id "choice_3")) |><|
          ((id "choice_2") |>  (id "choice_3"))
    )[
      printfn "Options 1 and 2 tied!";
    ][];
    _if' (((id "choice_1") === (id "choice_3")) |><|
          ((id "choice_1") |>  (id "choice_2")) |><|
          ((id "choice_3") |>  (id "choice_2"))
    )[
      printfn "Options 1 and 3 tied!";
    ][];
    _if' (((id "choice_2") === (id "choice_3")) |><|
          ((id "choice_2") |>  (id "choice_1")) |><|
          ((id "choice_3") |>  (id "choice_1"))
    )[
      printfn "Options 2 and 3 tied!";
    ][];
    _if' (((id "choice_1") |> (id "choice_2")) |><|
          ((id "choice_1") |> (id "choice_3")))
    [
      printfn "Choice 1 gets the majority!";
    ][];
    _if' (((id "choice_2") |> (id "choice_1")) |><|
          ((id "choice_2") |> (id "choice_3")))
    [
      printfn "Choice 2 gets the majority!";
    ][];
    _if' (((id "choice_3") |> (id "choice_1")) |><|
          ((id "choice_3") |> (id "choice_2")))
    [
      printfn "Choice 3 gets the majority!";
    ][];
    printfn "Voting ended!";
    delay_n 50;
  ]);

  parallel [
    "judge";
    "voter_1";
    "voter_2";
    "voter_3";
    "voter_4";
    "voter_5";
  ];

  _func void "app_main" [] [
  ];
]
```

/ 1mu

Bibliography

[1] Chromium Security - memory safety. `https://www.chromium.org/Home/chromium-security/memory-safety/`. Accessed: 2025-01-02.

[2] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. Challenges in designing exploit mitigations for deeply embedded systems. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 31–46. IEEE, 2019.

[3] Danel Ahman and Matija Pretnar. Asynchronous effects. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, 2021.

[4] Brian Amos. *Hands-On RTOS with Microcontrollers: Building real-time embedded systems using FreeRTOS, STM32 MCUs, and SEGGER debug tools.* Packt Publishing Ltd, 2020.

[5] Robert N Charette. This car runs on code. *IEEE spectrum*, 46(3):3, 2009.

[6] Trevor Elliott, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Launchbury. Guilt free ivory. *ACM SIGPLAN Notices*, 50(12):189–200, 2015.

[7] Thomas A Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In *Embedded Software: First International Workshop, EMSOFT 2001 Tahoe City, CA, USA, October 8–10, 2001 Proceedings 1*, pages 166–184. Springer, 2001.

[8] Philip Koopman, Jennifer Black, and Theresa Maxino. Position paper: Deeply embedded survivability. 2007.

[9] Edward A Lee. What are the key challenges in embedded software. *System Design Frontier*, 2(1):13, 2005.

[10] Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft: experiences building an embedded OS in Rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, PLOS '15, page 21–26, New York, NY, USA, 2015. Association for Computing Machinery.

[11] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The case for writing a kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–7, 2017.

[12] Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 500–514, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Mouaaz Nahas and Adi Maaita. Choosing appropriate programming language to implement software for real-time resource-constrained embedded systems. *Embedded Systems-Theory and Design Methodology*, pages 323–338, 2012.

[14] Y. Neuhard. A comparison of real-time operating systems for embedded computing, 2022. Seminar on Embedded Systems, Embedded Systems Group, Department of Computer Science, University of Kaiserslautern, Germany.

[15] Anni Peng, Dongliang Fang, Le Guan, Erik van der Kouwe, Yin Li, Wenwen Wang, Limin Sun, and Yuqing Zhang. Bitmap-based security monitoring for deeply embedded systems. *ACM Trans. Softw. Eng. Methodol.*, 33(7), September 2024.

[16] Gordon D Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, Volume 9, Issue 4, Dec 2013.

[17] Leo Poulson. Asynchronous effect handling. *Master's thesis. School of Informatics, University of Edinburgh*, 2020.

[18] Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electron. Notes Theor. Comput. Sci.*, 319(C):19–35, December 2015.

[19] Rust on Embedded Devices Working Group et al. The Embedded Rust Book. `https://docs.rust-embedded.org/book/`.

[20] Dan Saks. Writing better embedded software. `https://www.youtube.com/watch?v=3VtGCPIoBfs`, March 2019.

[21] Kensuke Sawada and Takuo Watanabe. Emfrp: a functional reactive programming language for small-scale embedded systems. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 36–44, 2016.

[22] Ayushi Sharma, Shashank Sharma, Sai Ritvik Tanksalkar, Santiago Torres-Arias, and Aravind Machiry. Rust for Embedded Systems: Current State and Open Problems. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 2296–2310, New York, NY, USA, 2024. Association for Computing Machinery.

[23] Su-Lim Tan and Tran Nguyen Bao Anh. Real-time operating system (rtos) for small (16-bit) microcontroller. In *2009 IEEE 13th International Symposium on Consumer Electronics*, pages 1007–1011. IEEE, 2009.

[24] Michael V Woodward and Pieter J Mosterman. Challenges for embedded software development. In *2007 50th Midwest Symposium on Circuits and Systems*, pages 630–633. IEEE, 2007.

[25] Akihiko Yokoyama, Sosuke Moriguchi, and Takuo Watanabe. A functional reactive programming language for small-scale embedded systems with recursive data types. *Journal of Information Processing*, 29:685–706, 10 2021.

[26] Akihiko Yokoyama, Sosuke Moriguchi, and Takuo Watanabe. Switching Mechanism for Update Timing of Time-Varying Values in an FRP Language for Small-Scale Embedded Systems. In *Proceedings of the 2024 13th International Conference on Software and Computer Applications*, pages 45–54, 2024.