MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Jaculus: Approachable Programming of Embedded Devices via Javascript

Bachelor's Thesis

PETR KUBICA

Brno, Spring 2023

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Jaculus: Approachable Programming of Embedded Devices via Javascript

Bachelor's Thesis

PETR KUBICA

Advisor: RNDr. Jan Mrázek.

Department of Computer Science

Brno, Spring 2023


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Petr Kubica

Advisor: RNDr. Jan Mrázek.

Acknowledgements

I would like to thank everyone who helped me make this thesis possible, especially my advisor Jan Mrázek, for his guidance and advice. I would also like to thank everyone from Robotárna for their support, feedback, and constant encouragement. Not to be forgotten are my friends, who for months had to constantly listen to me talking about Jaculus[1] and, nevertheless, helped me proofread this thesis. Finally, I would like to thank my family for their support and bottomless patience.

1. I am sorry, it is going to happen again.

Abstract

This thesis presents the design and implementation of Jaculus, a platform for programming microcontrollers using JavaScript and TypeScript. The platform consists of a runtime for the microcontroller and a command-line tool for accessing the device. The runtime is built around the QuickJS JavaScript engine and provides a high-level abstraction over JavaScript concepts to make creating new native features as easy as possible. The command-line tool can be used to connect to the device, upload source code, and control the device.

The runtime is then ported to the ESP32 and ESP32-S3 series of microcontrollers with several native modules. The port is then used to demonstrate the usage of Jaculus and is evaluated against other similar solutions.

Keywords

embedded systems, embedded programming, ESP32, JavaScript, TypeScript

Contents

1 Motivation

Microcontrollers in embedded devices are typically programmed in compiled languages like C, C++, or Rust. Although these languages provide high performance and low overhead at runtime, they can be challenging to learn and use.

Another problem of compiled languages, especially in embedded environments, is a long development cycle. Because of the limited resources of a microcontroller, the executable of such applications is usually self-contained to decrease the binary size and to avoid the need for a dynamic linker. Besides the application itself, such executables contain all used libraries, including the standard ones. All libraries must be linked on every build and, in some cases, also compiled regardless of whether they have changed.

The development cycle is often further prolonged by long deployment times. The build-deploy process can take several minutes, compromising development speed, especially in the early development stages when the developer rapidly iterates over the code.

The long development cycle can both hinder development speed and increase the development cost. Another big motivator for reducing the development cycle length is teaching beginners embedded programming. Seeing the results of their work quickly is important for keeping the learners motivated and interested in the subject, especially with children and their shorter attention spans.

As embedded devices often interact with the outside world or communicate with other devices, they often have to wait for external events rather than continuously perform a given computation. Therefore, most embedded applications have reactive and asynchronous elements, which are difficult to express in low-level languages. This induces a lot of boilerplate code, which obfuscates the main application logic and makes it harder to develop and maintain. Even though high-level languages such as C++ and Rust somewhat alleviate this issue, C is often the only language with direct support from the manufacturers, which adds more work for the developer and more code to the project.

2 Proposed solution

The proposed solution is to use a high-level, interpreted language instead. The language needs to:

- have low enough hardware requirements to run on a microcontroller,
- be easy to learn and use, and
- be able to express reactive and asynchronous elements.

The solution should also provide an ecosystem for controlling the device and developing applications. The ecosystem should include:

- firmware for the microcontroller, which gives complete control over the device to the runtime environment and provides an interface for controlling the device, and
- suitable tooling for controlling the device and uploading code to it.

2.1 Language choice

Many interpreted languages are available. Because of their popularity in embedding into other applications, the following languages were considered:

- Python
- Lua
- JavaScript

Python is a general-purpose language with an extensive standard library, which makes it suitable for many applications. However, its high hardware requirements caused by its large standard library and its memory model make it unsuitable for embedded devices.

Lua is a lightweight scripting language with a small standard library. It is suitable for embedded devices, and there are efficient Lua interpreters which are embeddable into C/C++ applications. On the other hand, Lua is not nearly as popular as Python or JavaScript, making it harder to find relevant resources and justify learning a new language.

JavaScript is a popular language commonly used in web browsers. The language specification defines a small standard library, which is extended by the runtime environment, meaning a small base memory footprint. It also maps very well to the event-centered nature of many embedded systems, as JavaScript is inherently event-driven. Multiple embeddable JavaScript engines exist, such as DukTape, MuJS, and QuickJS.

Because of the reasoning above, JavaScript was chosen to be used in the solution implemented in this thesis.

However, since JavaScript is weakly typed, debugging errors caused by type mismatches can be challenging. A possible solution is to use a strongly typed language, such as TypeScript. However, as transpiling[1] or interpreting TypeScript on a microcontroller is not feasible, it must first be transpiled to JavaScript outside of the device.

1. Transpilation refers to the process of source-to-source compilation

3 Overview

The main task is to create a JavaScript runtime environment for microcontrollers and an ecosystem around it for managing the device and developing applications for it.

The implementation primarily focuses on the ESP32 and ESP32-S3 series microcontrollers, as they are popular in the maker community and provide high performance at a reasonably low price. Because of their extensive feature set, they also serve as a good entry point for beginners, who can try out many different things with a single device.

Firmware for microcontrollers, which gives complete control over the device to the JavaScript runtime and provides an interface for programming and controlling the device, should be developed. A device running this firmware will be called a *Jaculus device*.

3.1 Runtime environment

The runtime environment should be able to run JavaScript code and be easily extensible with functionality implemented in C++. The runtime should be usable as the primary interface for programming the device and as a component of a larger application.

An example use case for the latter is in a system of devices used as game elements. Their low-level logic (e.g., communication, user interface) can be implemented in C++, while the high-level logic (e.g., game rules) can be updated independently by the user.

3.2 JavaScript engine

A JavaScript engine is needed to interpret JavaScript code. As implementing a custom JavaScript engine would be a significant undertaking, an existing one is used instead. There are multiple options available, the popular ones being V8[1], DukTape[2], MuJS[3], and QuickJS[4].

The V8 engine is the most popular JavaScript engine and is used in Google Chrome and Node.js. It is a high-performance engine with

a very large memory footprint, making it impossible to run on a microcontroller.

DukTape and MuJS are small, embeddable JavaScript engines. They are suitable for embedded devices, but support old versions of the ECMAScript specification.

QuickJS is a small, embeddable JavaScript engine that supports the ECMAScript 2020 specification. According to benchmarks published by its author[5], on a desktop platform, it is 2-4 times faster than DukTape and MuJS. The performance comes for the price of a larger memory footprint, which is still small enough to fit into a microcontroller.

The created solution uses QuickJS because it provides a good balance regarding its feature set, performance, and memory footprint compared to the other options.

3.3 Communication

There should be a way to communicate with the device — to upload code, control the runtime and monitor its state.

Most microcontrollers feature a serial interface, such as a USB-to-UART bridge or a native USB interface. A serial interface only provides a single duplex byte stream, meaning a protocol must be implemented on top of it to enable communication with multiple services over a single connection.

Using a single stream connection also adds flexibility in the choice of the transport medium. Aside from the serial interface, the protocol can be used over a network socket, web socket, or any other kind of byte stream connection.

3.4 Filesystem

The device should have suitable storage to allow the user to upload and store code on the device.

The ESP32 and ESP32-S3 series microcontrollers feature a flash memory chip, which can be used to store user data. The ESP-IDF also provides an implementation of the FAT filesystem, which can be used as an abstraction to store files in the flash memory.

3.5 Tooling

A suitable tooling should be created to support the development of applications for the device. The tools should allow the user to upload code to the device, control the runtime, and monitor the device's state.

3.6 Implementation

To achieve the goals described above and to allow for possible future reuse of independent components, the project is split into multiple parts:

- Jaculus-machine — standalone, embeddable, C++ centric JavaScript runtime using QuickJS at its core
- Jaculus-link — standalone communication library for multiplexing a number of channels on a single stream connection
- Jaculus-dcore — core library for creating new Jaculus devices
- Jaculus-tools — command-line application for controlling and monitoring Jaculus devices
- Jaculus-esp32 — Jaculus device firmware for the ESP platform (with support for the ESP32 and ESP32-S3)

The libraries are implemented in C++ and are designed to be easily embeddable into other C++ applications. They export all of their functionality in the `jac` namespace and are configured as CMake projects and ESP-IDF components. In code examples, the `jac` namespace is omitted for brevity but should be present in actual code.

Figure 3.1 shows an overview of the full Jaculus firmware. Jaculus-machine and Jaculus-link implement the runtime environment and communication protocol, respectively. Jaculus-dcore builds on top of them and provides a higher-level interface for controlling the device. Jaculus-esp32 then utilizes Jaculus-dcore and implements the firmware for the ESP platform by implementing the device-specific functionality.

The implementation opts for a multithreaded design because continuously interrupting the runtime to poll for new events would slow down the execution of JavaScript code. The JavaScript runtime runs in a separate thread and allows other threads to schedule tasks to be

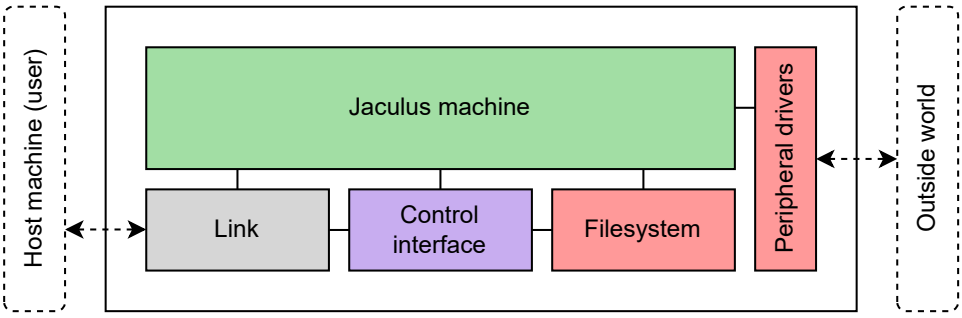

Figure 3.1: Structure of the Jaculus firmware

executed in the runtime thread. The main thread then executes these tasks in the order they were scheduled. Separate threads are also used for other tasks, such as communication with the host device. Therefore, most implemented data structures implement blocking operations, allowing these threads to wait for respective events instead of polling for them.

4 Used technologies

This chapter briefly describes some selected technologies used in the project and some of their specifics.

4.1 ESP32 and ESP32-S3

ESP32[6] and ESP32-S3[7] are microcontrollers series by Espressif Systems.

ESP32 is a dual-core microcontroller based on the Xtensa LX6 CPU architecture with 520 KiB of SRAM. It features a UART interface for programming.

The ESP32-S3 is a newer iteration of the ESP32 series of microcontrollers. It is based on the Xtensa LX7 CPU architecture and has 512 KiB of SRAM. Aside from a UART interface, it also features a native USB interface, which can be used for programming.

Both of these microcontrollers also provide a Wi-Fi and Bluetooth interface and a large number of other peripherals. On many boards based around these microcontrollers, the programming UART interface is connected to a USB-to-UART bridge, which allows the microcontroller to be programmed over USB. The USB ports are also often used for communication with the program running on the microcontroller.

These microcontrollers are also coupled with flash memory, which stores the firmware and can be used to store other data. The flash memory size varies between individual boards but is usually at least 4 MiB.

4.2 ESP-IDF

ESP-IDF is the official development framework for microcontrollers from Espressif Systems. The framework is based on FreeRTOS and provides a set of libraries and tools for developing applications for, among others, ESP32 and ESP32-S3 microcontrollers.

Most of the libraries provided with ESP-IDF have only a C API. The framework also supports C++20 with a large subset of its standard

library. However, some parts of the standard library do not work entirely correctly (e.g., `std::filesystem`).

4.3 JavaScript

JavaScript is a high-level, interpreted, and weakly, dynamically typed programming language. It is standardized in the ECMAScript specification, which is maintained by Ecma International.

Although JavaScript programs are event-driven, the code is executed in a single thread. This is achieved by using an event loop, where asynchronous events are queued and executed in the order they are received. Therefore, JavaScript programs must be written in a non-blocking manner, as blocking the event loop will cause the program to stop responding to events. Events are generated by the JavaScript engine or the host environment both synchronously and asynchronously.

4.4 TypeScript

TypeScript is a strongly typed superset of JavaScript and is developed and maintained by Microsoft. TypeScript is typically not interpreted directly and is instead transpiled into JavaScript, which can be interpreted using any JavaScript runtime that supports the specified ECMAScript version.

4.5 QuickJS

QuickJS is a JavaScript engine implementing the ECMAScript 2020 Language Specification[8] (ES2020). It was developed by Fabrice Bellard and Charlie Gordon and is available under the MIT license. It is written in C and is designed to be embeddable into other applications.

QuickJS uses POSIX to implement atomic operations and system time. Although this slightly limits its portability, ESP-IDF, the primary target platform for Jaculus, does support POSIX.

According to ES2020, JavaScript code is evaluated in a *Realm*, which defines the execution environment (e.g., global object and set of built-

in objects). QuickJS uses a different term for this concept — *Context*, which I have adopted for Jaculus and which will be used throughout the rest of this thesis.

Slight modifications have been made to the QuickJS source code to allow it to be compiled with the ESP-IDF toolchain. These modifications do not concern the engine's logic and only change its platform-specific configuration. It has also been extended to work with the CMake and ESP-IDF build systems.

5 Jaculus-machine

Jaculus-machine is a standalone, embeddable, C++ centric JavaScript runtime. It uses the QuickJS JavaScript engine at its core and provides a C++ interface, with the aim of providing a simple and easy-to-use API for adding features to the runtime.

A large portion of Jaculus-machine is a set of wrapper classes around the QuickJS API. The classes provide RAII semantics and an easy-to-use API for routine use cases.

Jaculus-machine uses two core concepts, which are referred to as *Machine* and *MFeature* in the code and the rest of this thesis. A *Machine* is an instance of the runtime complete with all the selected MFeatures. An *MFeature* is a module that can be used as a part of a Machine and which provides functionality to the runtime or to other MFeatures.

The modular design of Jaculus-machine allows for easy extensibility and customization of the runtime. The user can select which MFeatures should be included in the runtime and also create new MFeatures with fairly low effort. It also allows for easier portability of existing MFeatures to other platforms.

All public functionality of the library is contained in the `jac` namespace. QuickJS exports its symbols in the global namespace, and their names are usually prefixed with `JS`. The user should not interact with QuickJS directly for regular use, as the library provides wrappers for most of its functionality.

5.1 Machine and MFeatures

The main entry point of the library is a Machine instance. After a Machine is created and initialized, it can be used to interact with the JavaScript runtime and execute JavaScript code.

A Machine is defined using templated stack inheritance from the `MachineBase` class and selected MFeature classes. The `MachineBase` class provides the core functionality of the runtime, and the MFeature classes implement additional functionality, such as an event loop and filesystem access.

Stack inheritance is a technique where a class is defined as a template parametrized by another class, which it then inherits from. The

inheritance chain is resolved at compile time, and each class in the stack can access the members of the classes lower in the stack. An example of stack inheritance can be seen in a code snippet below.

```
class A { ... };

template<class Next>
class B : public Next { ... };

template<class Next>
class C : public Next { ... };

using Stack = C<B<A>>;
```

The stack design of the Machine allows interfacing with different MFeatures in C++ directly without any middleware. Lower-level MFeatures are located lower in the stack and implement platform-specific functionality, while higher-level MFeatures are located higher in the stack and can use the abstraction provided by the lower-level MFeatures. This allows for easy portability of higher-level MFeatures to other platforms.

5.2 JavaScript concepts in C++

Because many JavaScript concepts do not have a direct equivalent in C++ or have different semantics, Jaculus-machine provides a set of classes that wrap their representation from the QuickJS API. The classes provide RAII semantics and aim to be as easy to use as possible.

5.2.1 Values

JavaScript values can contain either primitive types (e.g., numbers) or objects (e.g., arrays). Primitive values are stored in stack memory, while objects are stored in the heap. QuickJS uses reference counting for memory management, and the user is responsible for decreasing the reference count of JavaScript values that are no longer needed. Jaculus-machine provides a set of classes that wrap JavaScript values and provide RAII semantics for them. The classes also provide a simple API for converting to and from C++ types.

The base class for JavaScript value wrapper is `ValueWrapper` and provides a general API for working with JavaScript values. More specific wrapper classes are derived from `ValueWrapper` and provide additional functionality, such as `ObjectWrapper` and `FunctionWrapper`.

`ValueWrapper` has a template parameter `managed` that specifies whether the wrapper takes ownership of the underlying JavaScript value. This pattern is assumed from QuickJS, which sometimes does not give ownership of the value to the user to reduce the number of changes in its reference count. If the value is a reference, depending on the value of `managed`, the wrapper will either be a strong or a weak reference to the value. For convenience, the library provides two type aliases for all built-in wrappers, which are usually used instead and respect the following pattern:

```
// value/strong reference
using Value = ValueWrapper<true>;
// value/weak reference
using ValueWeak = ValueWrapper<false>;
```

A `ValueWrapper` instance can be obtained either as a result of the execution of JavaScript code or by using static methods of `ValueWrapper` and its subclasses. The following code shows some examples:

```
ContextRef ctx = ...;

Value undefined = Value::undefined(ctx);
Value number = Value::from<int>(ctx, 42);
Value string = Value::from<std::string>(ctx, "Hello, world!"
    );
Value object = Object::create(ctx);
Value array = Array::create(ctx);
```

The `ValueWrapper` class provides methods for converting the value to a C++ value or a subclass of `ValueWrapper` to access additional functionality. The following code shows some examples:

```
Value value = ...;

int number = value.to<int>();
std::string string = value.to<std::string>();
Object object = value.to<Object>();
```

If the wrapped value cannot be converted to the requested type, a `jac::Exception` will be thrown.

5.2.2 Type conversions

In many cases, the library can perform automatic type conversions between JavaScript values and C++ types. Among others, automatic conversions happen in getters, setters, and function calls. This allows for wrapping existing functions without having to write conversion code manually.

Default conversions for several built-in types are provided, such as `int`, `double`, `std::string`, and `std::vector`. The library also provides a mechanism for defining custom conversions for user-defined (and not-yet-supported built-in) types. To define a custom conversion, the user must implement a template specialization of the `ConvTraits` structure for the type. The following code shows a specialization for the `bool` type, which is included with the library:

```
template<>
struct ConvTraits<bool> {
  static bool from(ContextRef ctx, ValueWeak val) {
    return JS_ToBool(ctx, val.getVal());
  }

  static Value to(ContextRef ctx, bool val) {
    return Value(ctx, JS_NewBool(ctx, val));
  }
};
```

5.2.3 Exceptions

A portion of the Jaculus-machine library consists of wrappers that allow calling C++ functions from JavaScript code. When an exception is thrown in the wrapped C++ code, it is caught by the wrapper, converted to a JavaScript exception, and propagated to the runtime. The library allows the user to specify the JavaScript `Error` type or any other JavaScript value that should be thrown.

Similarly, when a JavaScript function is called from C++ and an exception is thrown in the runtime, it is caught by the wrapper, converted to a C++ exception of type `jac::Exception`, and propagated to the C++ code.

Different types of C++ exceptions are converted to JavaScript exceptions as follows:

- `jac::Exception` — a specified `Error` type, or the wrapped value if `type == Type::Any`
- `std::exception` — an `InternalError` object with the message `e.what()`
- any other exception — an `InternalError` object with the message "unknown exception"

To create an `Exception` that will be converted to JavaScript as the specified `Error` type, the `Exception::create` method can be used:

```
Exception::create(Exception::Type::TypeError, "invalid
    argument");
```

To create an `Exception` that will be converted to JavaScript as the specified value, the value should be created independently and then converted to an `Exception` using the `ValueWrapper::to` method:

```
Value value = ...;
Exception exception = value.to<Exception>();
```

5.2.4 Functions

The library provides an interface for defining JavaScript functions by wrapping most existing callable C++ objects. This interface is presented in the form of `FunctionFactory` class.

To wrap a function, the user can use the `newFunction` and `newFunctionThis`. All arguments that are passed to the function call will be converted to the types of the function parameters. If the number of arguments does not match or if the values cannot be converted to the required types, a `TypeError` will be thrown.

Because variadic functions in C++ are processed at build time, they can not be universally wrapped to create their JavaScript counterparts. For this reason, `FunctionFactory` lets the user define a function with an argument of type `std::vector<ValueWeak>`. When this function is called from JavaScript, the vector will contain all arguments passed to the function. These functions can be created using the `newFunctionVariadic` and `newFunctionThisVariadic` methods.

The methods `newFunctionThis` and `newFunctionThisVariadic` additionally give access to the `this` value of the function (for example, when the function is called as a method of an object).

The following code shows some examples:

```
ContextRef ctx = ...;
FunctionFactory ff(ctx);

Function add = ff.newFunction([](int a, int b) { return a +
    b; });

Function sum = ff.newFunctionVariadic([](std::vector<
    ValueWeak> args) {
  int sum = 0;
  for (auto& arg : args) {
    sum += arg.to<int>();
  }
  return sum;
});
```

JavaScript functions can be called from C++ using the methods `Function::call` and `Function::callThis`. The latter additionally gives access to the `this` value of the function. Both methods have a template argument that specifies the type the return value should be converted to; arguments are converted to JavaScript values automatically. Object constructors in JavaScript are also represented as functions but have different call semantics — in JavaScript, they are called with the `new` keyword. To call a constructor from C++, the `Function::callConstructor` method should be used. The following code shows some examples using the functions defined above:

```
int added = add.call<int>(1, 2);  // returns 3
int summed = sum.call<int>(1, 2, 3, 4);  // returns 10

Function ctor = machine.eval("class { constructor(a) { this.
    a = a; } }");
Value res = ctor.callConstructor(42);  // returns { a: 42 }
```

5.3 Features

Jaculus-machine mainly consists of wrappers around QuickJS API and provides only a small set of MFeatures that are required for the runtime to function correctly or are helpful for development and debugging. Instead, the library aims to provide functionality, which makes the creation of new MFeatures as easy as possible.

5.3.1 Built-in MFeatures

The following MFeatures are included with the library (their class names are suffixed with "Feature" in the library):

- EventQueue — an event queue that can be used to schedule events to be executed in the event loop
- EventLoop — an event loop that executes events from an event queue in the main thread
- Filesystem — an abstraction over the filesystem that provides access to files and directories
- ModuleLoader — an implementation of module loader for importing modules from the filesystem (using the `import` statement in JavaScript) and evaluating JavaScript files (using the `evalFile` method in C++)
- BasicStream — an implementation of simple Readable and Writable stream types
- Stdio — a feature adding `stdin`, `stdout`, and `stderr` streams to the Machine and `console` interface to the JavaScript side
- Timers — a feature for configuring timers from JavaScript and C++

5.3.2 Default event loop

Default event loop implementation is split into two separate MFeatures. One implements an event queue, and the other implements the event loop. This allows for easier portability, as the event loop itself can be reused on different platforms, while the event queue can be extended to, for example, support scheduling events from interrupts.

5.3.3 Watchdog

The `MachineBase` class implements a watchdog timer that can be used to detect infinite loops in the runtime. The watchdog can be configured using the `setWatchdogTimeout` method. By setting the timeout to 0, the watchdog can be disabled, which is the default state.

By default, the watchdog will interrupt the runtime on timeout. This behavior can be changed by setting the watchdog handler using the `setWatchdogHandler` method. Instead of interrupting, the handler

will be called, and depending on its return value, the runtime will either be interrupted or continue running.

5.3.4 Class wrapping

Sometimes, it is desirable to create a JavaScript object backed by a C++ object. This can be done using the class `Class`. The class is templated by a `ProtoBuilder` structure, which defines how the JavaScript object prototype should be constructed and how the optional opaque data should be managed. The class `Class` can then be used to initialize the class in a given Context, construct the JavaScript object or obtain its constructor.

An example use case could be a class that represents a device peripheral. The C++ object could contain a handle to the peripheral, implement functionality for using the peripheral, and use the JavaScript object as a proxy to access these functions from JavaScript code.

5.4 Usage

The central part of the library is the `Machine` type, which represents a JavaScript runtime with a set of MFeatures. An example definition of a Machine is shown below:

```
using Machine = ComposeMachine<
  ModuleLoaderFeature,
  FilesystemFeature,
  StdioFeature,
  BasicStreamFeature,
  MachineBase
>;
```

The `ComposeMachine` class is a helper class that builds the Machine inheritance chain. The MFeatures at the beginning of the argument list will be at the top of the stack, and the MFeatures at the end will be at the bottom. The `MachineBase` class is a base class for all Machines and must be at the end of the list.

To use the machine, an instance of the `Machine` type must be created. Then the machine can be configured and after calling the `initialize` method, it can be used. The following code shows an example of creating and initializing a Machine:

```
Machine machine;

// Configure the machine
machine.setWatchdogTimeout(1000);
machine.setWatchdogHandler([](Machine& machine) {
  std::cerr << "Watchdog timeout!" << std::endl;
  return true;
});
machine.initialize();

// Use the machine
machine.evalFile("main.js");
```

5.4.1 JavaScript classes

As described in Section 5.3.4, the library provides a mechanism for creating JavaScript classes, which can be backed by C++ objects.

To create a class, the user must first define a `ProtoBuilder` structure. The `ProtoBuilder` describes the class's behavior and properties through its static interface. Its features are specified by inheriting from the structures in the `ProtoBuilder` namespace and overriding their static interfaces. These structures contain the default implementation of their interfaces and some convenience functions for describing the class. The following code shows an example of a `ProtoBuilder`:

```
struct MyBuilder : public ProtoBuilder::Opaque<MyClass>,
    public ProtoBuilder::Properties {
  static MyClass* constructOpaque(ContextRef ctx, std::
      vector<ValueWeak> args) {
    return new MyClass();
  }

  // The default destructOpaque implementation uses delete.
     By overriding it, a custom deleter can be used

  static void addProperties(ContextRef ctx, Object proto) {
    addPropMember<int, &MyClass::foo>(ctx, proto, "foo");
  }
};
```

The `Class` template can be instantiated with the `ProtoBuilder` structure to create a class definition, and a name can be assigned using the `init` method. The `init` method can be called repeatedly without

any effect if called with the same name; otherwise, an exception will be thrown. To use the class in a given Context, the class must be initialized in the Context by calling the `initContext` method:

```
// Initialize the class
using MyClassJs = Class<MyBuilder>;
MyClassJs::init("MyClass");

// Initialize the class in a context
ContextRef ctx = ...;
MyClassJs::initContext(ctx);
```

After performing the above steps, the class constructor and prototype can be retrieved, and the class can be instantiated:

```
Value constructor = MyClassJs::getConstructor(ctx);
Value prototype = MyClassJs::getPrototype(ctx);

Value obj = constructor.to<Function>().callConstructor();
```

5.4.2 Creating new Features

An MFeature is defined as a template class. To allow building the inheritance chain of a Machine, the class must set the class `Next` presented in its last template parameter as its base class. To add functionality to the machine, the class may:

- present a public C++ interface to MFeatures higher in the inheritance chain or to the user of the Machine, and
- use the `initialize` method to add functionality to the JavaScript runtime

In the MFeature constructor, the user must not interact with the JavaScript runtime in any way, as it is not initialized yet. Initialization of the MFeature involving the JavaScript runtime should be done in the `initialize` method.

The following code shows an example of a Feature that adds a `foo` property to the global object:

```
template<typename Next>
class MyFeature : public Next {
  void initialize() {
    ContextRef ctx = this->context();
```

```
    Object global = ctx.getGlobalObject();
    global.defineProperty("foo", Value::from(ctx, 42));
  }
};
```

The user might also want to define a JavaScript module. This can be done using the `MachineBase::newModule` method. The following code creates a module named `myModule`, which exports a value named `foo`:

```
template<typename Next>
class MyFeature : public Next {
  void initialize() {
    ContextRef ctx = this->context();

    Module& module = this->newModule("myModule");
    module.addExport("foo", Value::from(ctx, 42));
  }
};
```

6 Jaculus-link

As described in the introduction, the Jaculus device and the host communicate through a single byte stream connection.

Because the services running on the device work mostly independently, it is desirable to have independent communication channels for each of them. This can be achieved by multiplexing a number of channels on a stream connection.

Sometimes, the device may have multiple communication interfaces, which can all be used for communication with the host. In such cases, it is desirable to be able to route the communication to the appropriate interface.

This functionality is implemented in the Jaculus-link library, which provides a way to multiplex 256 channels on a single byte stream connection and to route the communication between the appropriate interface and its consumer.

The library is implemented strictly using only the C++ 20 standard library for easy portability. For this reason, it does not provide an implementation for communication interfaces, and the user must provide one themselves.

6.1 Architecture

The model of Jaculus-link is split into three layers:

1. Data link layer
2. Routing layer
3. Communicator layer

6.1.1 Data link

The data link is responsible for transmitting data along with channel identifiers. The data link provided in this library is implemented in the `Mux` class and multiplexes 256 channels on a stream connection.

It is possible to use other data link implementations as long as they implement the `DataLinkTx` interface for transmission and provide a way to connect them to a `DataLinkRx` for processing received data.

6.1.2 Routing layer

The routing layer is responsible for routing the received data to the channel consumer. The routing layer is implemented in the `Router` class.

A `Router` instance can be connected to multiple data links and will route data from all of them to the appropriate consumer with the information about the link it was received from. It also allows sending data to a specific channel and link.

6.1.3 Communicator layer

The communicator layer is used as an abstraction layer for communicating through channels. Typically, the communicator is associated with a single channel and provides either an interface for sending or receiving data.

Communicators used for receiving data from a `Router` must implement the `Consumer` interface, which allows them to be subscribed to a specific channel on a `Router` instance. They must process the received data without blocking, preferably only by storing it in a buffer and processing it later.

Communicators that send data do not have a unified binding interface. Instead, they access the `Router` instance directly and send data to a specific channel on a specific link (or links).

6.1.4 Full Jaculus-link pipeline

An example configuration of the entire pipeline provided by the library is shown in the diagram in Figure 6.1.

6.2 Multiplexer protocol

The library provides one protocol for the multiplexer in the class `CobsEncoder`. The protocol is based on a modified version of the COBS[9] algorithm for data framing and was originally proposed[1] by Jaroslav Malec for the use in the predecessor version of the Jaculus project.

1. The discussion can be found at `https://github.com/yaqwsx/Jaculus/pull/15`.

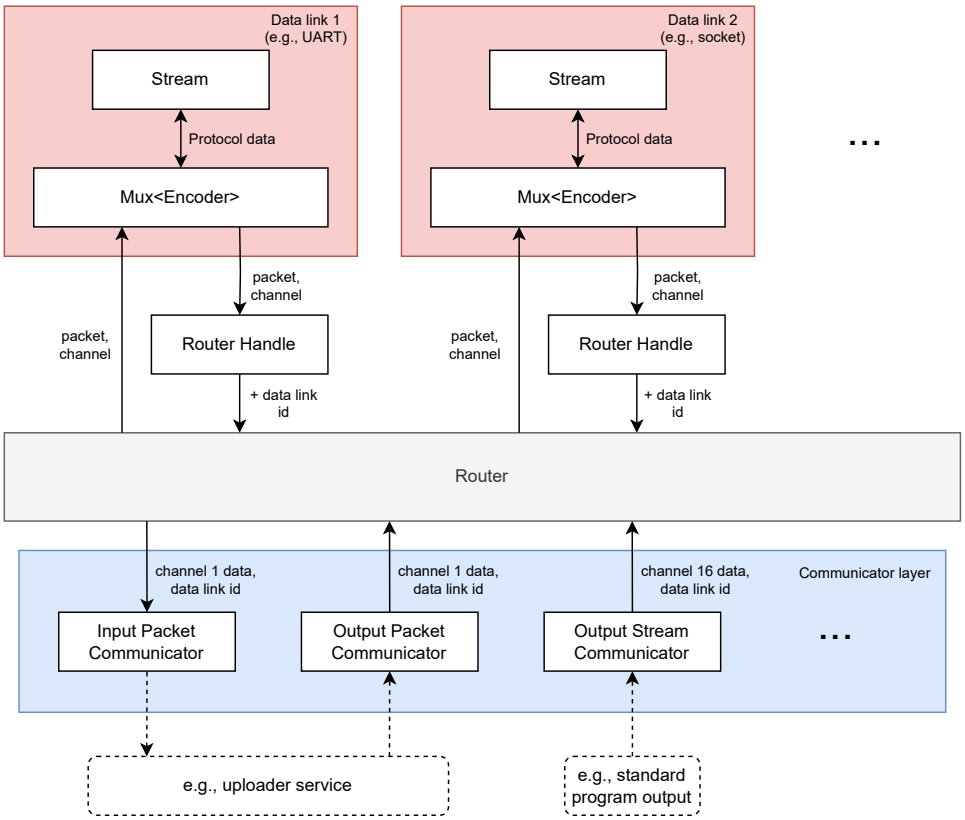

Figure 6.1: An example configuration of the full Jaculus-link pipeline

In the original COBS algorithm, a delimiter byte — typically zero — is inserted at the end of the data frame. To encode the transmitted data, every occurrence of the delimiter byte in the data is replaced by a value representing the number of bytes to the next delimiter byte. This allows for data framing with a fixed two-byte overhead while limiting the maximum data size to 254 bytes.

The modified version of the algorithm moves the delimiter byte to the start of the data frame and adds length information to the second byte of the data frame. The rest of the data frame is encoded in the same way as in the original algorithm, except for the missing delimiter byte at the end of the data frame, which is only implied by the data frame length. Every such data frame has a three-byte overhead and can contain up to 254 bytes of data.

Moving the delimiter byte to the start of the data frame allows for resetting the packetization state in case a previous data frame is lost, malformed, or other corrupted data is received on the stream. This is important for the use on microcontrollers, where the used serial port is also often used for logging errors by the built-in libraries. These errors would otherwise cause the packetizer to lose synchronization and cause data frames to be lost.

Figure 6.2 shows a data frame structure diagram. In the data section of the data frame, the first byte contains the channel identifier, followed by the transmitted data. The last two bytes of the data frame contain the CRC16 checksum of the data section for error detection.

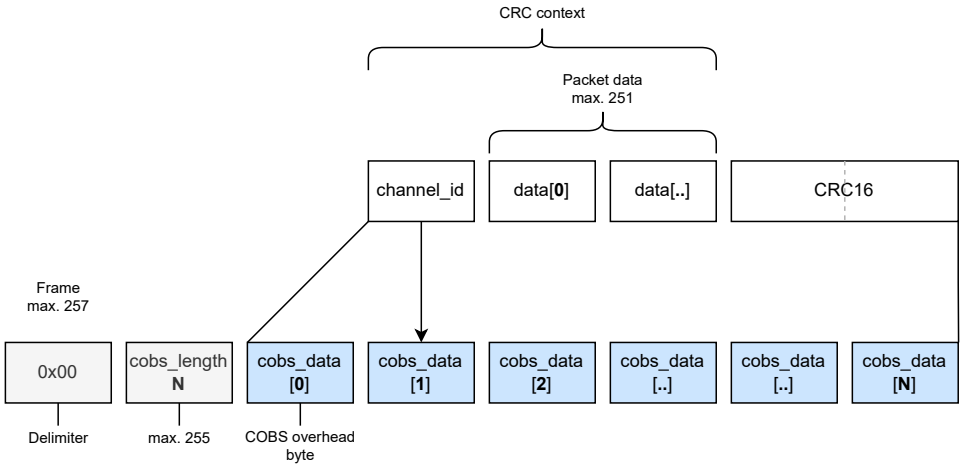

Figure 6.2: Diagram of the COBS-based multiplexer protocol data frame[a]

a. Adapted from: MALEC, Jaroslav. *Protocol diagram*. In: GitHub [online]. 2021 [visited on 2023-05-10]. Available from: `https://github.com/yaqwsx/Jaculus/pull/15`.

6.3 Usage

This section describes the usage of the Jaculus-link library.

6.3.1 Multiplexer

The class `Mux` is a data link implemented as a multiplexer and is templated by the class implementing the underlying multiplexer protocol. The only protocol provided with the library is in the `CobsEncoder` class. To implement another protocol, the user can create a class with the same interface as `CobsEncoder`.

The `Mux` constructor takes a `Duplex` instance, which defines the stream connection used for transmitting and receiving data. The `Duplex` interface serves as an abstraction layer for the stream connection and must be implemented by the user.

6.3.2 Router

The `Router` class implements the routing layer. A `Router` instance can be connected to multiple data links, as shown in the following example:

```
// Create a router
Router router;

// Create a stream connection
auto stream = std::make_unique<MyStream>();

// Configure a data link
Mux<CobsEncoder> mux(std::move(stream));

// Connect the data link to the router
auto handle = router.subscribeTx(1, mux);
mux.bindRx(std::make_unique<decltype(handle)>(std::move(
    handle)));

// Connect another data link to the router
auto link2 = ...;
auto handle2 = router.subscribeTx(2, link2);
link2.bindRx(std::make_unique<decltype(handle2)>(std::move(
    handle2)));
```

The `handle` object is used to receive data from the data link and must be bound to the same data link instance as the one used to subscribe to the router, as it adds the information about the data link to the received data.

6.3.3 Communicators

The communicators are used as an abstraction layer for communicating through channels. Communicators provide an interface for sending or receiving data through a channel. The provided communicator types are:

- `OutputStreamCommunicator` — sends data as a stream of bytes
- `InputStreamCommunicator` — receives data as a stream of bytes
- `OutputPacketCommunicator` — sends data while exposing the underlying data framing
- `InputPacketCommunicator` — receives data while exposing the underlying data framing

These communicator types are only interfaces, and their implementations for `Router` are provided in classes with the same names prefixed with `Router`.

The following example shows how to create a pair of stream communicators:

```
Router router;

// Create an input stream communicator
RouterInputStreamCommunicator input({});

// Subscribe the communicator to the router
router.subscribeChannel(1, input);

// Create an output stream communicator and connect it to
    the router
RouterOutputStreamCommunicator output(router, 1, {});
```

7 Jaculus-dcore

To make porting Jaculus to different platforms easier, the core functionality of a Jaculus device (e.g., communication, control protocol, JavaScript runtime) is implemented in the Jaculus-dcore library in a platform-independent way. The library requires C++20 and POSIX support to be available on the target platform.

Jaculus-dcore uses the Jaculus-machine library for the JavaScript runtime and the Jaculus-link library for communication.

7.1 Architecture

The Jaculus-dcore library is centered around the `Device` class, which is used to define a Jaculus device and which bundles all of the core functionality of the library.

7.1.1 Device class

The `Device` class is the entry point for defining a Jaculus device. It is a template parametrized by the Machine type used for the JavaScript runtime. The Machine type must implement the `evalFile` method, which is used to run the code uploaded to the device.

The `Device` class exposes a `Router` object from Jaculus-link, which is used to connect the device to the communication interface. `Device` also exposes an interface for controlling the internal Machine instance from C++ code.

Three services are also part of the `Device` class and expose functionality over the communication channel (or channels):

- Controller — service for controlling and monitoring the device
- Uploader — service for uploading code/data to the device
- Logger — service for logging messages from the device

The services use separate channels provided by the `Router` object. Other channels are also reserved for the standard input and output of the JavaScript instance.

The `Device` implements a locking mechanism to prevent multiple clients from accessing the device simultaneously. The lock is paired

with a timeout, which is continuously reset while the client communicates with the devices. If the timeout expires, the lock is released, and other clients can access the device. The lock is exposed to the client via the Controller service.

7.2 Implementation

7.2.1 Controller service

The Controller service is implemented in the `Controller` class. It uses a *PacketCommunicator* interface to communicate with the client.

The first byte of each packet is used to specify the command to be executed. The rest of the packet is used to transmit command-specific data. The packet structure of the protocol is shown in a diagram in Figure 7.1.

The service exposes the following functionality:

- accessing the device lock,
- controlling the internal Machine instance,
- using the Machine instance's standard input and output, and
- monitoring the device status.

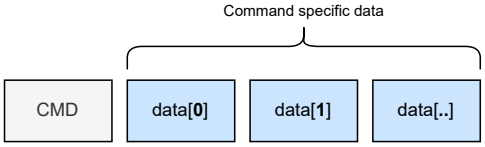

Figure 7.1: Controller protocol packet structure

7.2.2 Uploader service

The Uploader service is implemented in the `Uploader` class. Similarly to the Controller service, it uses a *PacketCommunicator* interface to communicate with the client. The first byte of each packet is used to specify the command to be executed. The rest of the packet is used to transmit command-specific data. The packet structure of the protocol is shown in a diagram in Figure 7.2.

The service provides the following functionality:

- listing files and directories,
- creating and deleting directories, and
- writing, reading, and deleting files.

Most commands are processed in a single packet, but writing files requires the data to be sent in multiple packets. When writing a file, an internal state is set to specify what operation should be performed when data is received and when the transmission is finished. To prevent overflow of the receiving buffer, the command implements, admittedly relatively inefficient, flow control — the client must acknowledge each packet before the next one is sent.

Commands for reading a file and listing a directory might also split the data into multiple packets. For simplicity, no flow control is implemented when transmitting data to the client, as the client is expected to be a much more powerful device that can handle the transmission speed.

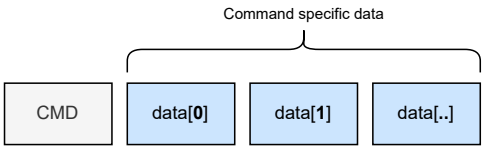

Figure 7.2: Uploader protocol packet structure

7.2.3 Filesystem access

The Uploader service provides access to the filesystem of the device. Unfortunately, because `std::filesystem` is not yet fully implemented in ESP-IDF, the implementation has to, in some cases, rely on the POSIX filesystem API to run on the ESP platform.

7.3 Usage

7.3.1 Creating a new device

To create a new device, an instance of the `Device` class must be created. The class is templated by the Machine type used for the JavaScript runtime. The definition of a Machine type is described in Section 5.4.

The `Device` must be connected to a communication interface. The object exposes a `Router` object from Jaculus-link, which should be used to connect the device to a data link. Definition and binding of a data link are described in Section 6.3.

After the device is initialized, the `Device::start` method must be called to start the provided services.

An example of a device definition can be seen in Jaculus-esp32 in the `main.cpp` file.

7.3.2 Controlling the device

To control the device, the user can use the Jaculus-tools application, which provides a command-line interface for managing the device. The application is described in the following chapter in more detail.

8 Jaculus-tools

To allow the user to interact with Jaculus devices, a command-line application called Jaculus-tools was created. The application is implemented in TypeScript and is available as a package in the npm registry[1] under the name `jaculus-tools`.

The package can also be used as a library for building custom applications, but this functionality is currently not documented. The potential user might want to look at the source code of the command-line application part of the package for inspiration on how to use the library.

8.1 Features

The application provides commands to check the status of the device (`status`, `version`), install Jaculus firmware to the device (`install`), control the JavaScript runtime (`start`, `stop`, `monitor`), and access the device's filesystem (`ls`, `read`, `write`, `rm`, `mkdir`, `rmdir`, `upload`, `pull`). The application also provides commands for compiling and uploading code to the device (`build`, `flash`).

Other utility commands are also available:

- `help` – prints help for the specified command
- `list-ports` – lists available serial ports
- `serial-socket` – tunnel a serial port over a TCP socket

The commands are run by specifying them after the `jac` command.

8.2 Implementation

8.2.1 Communication with the device

To communicate with the device, the application implements the same protocol as the Jaculus-link library described in Section 6.2.

---

1. npm registry contains packages for the Node.js runtime environment. The registry can be accessed through a website at `https://www.npmjs.com/` or through the `npm` command-line tool.

The application implements a similar pipeline to the one used in the library, except for the Routing layer, which is omitted, as connecting to multiple devices would be pointless. Aside from that and the language choice resulting in some different programming paradigms, the architecture is mostly similar.

8.2.2 Device access

The application provides access to the device via the `Device` class. The class exposes the following functionality to the user:

- Controller and Uploader services,
- standard input and output of the running program, and
- output of the device logger.

8.2.3 Command-line argument parser

The application uses a custom parser for command-line arguments, allowing for chaining compatible commands.

The commands can also access and modify a global state object passed to each command, allowing them to share data — for example, once the device is connected, the `Device` instance is saved to the state object and can be accessed by other commands.

For example, the following command:

```
jac --port /dev/ttyUSB0 build flash monitor
```

Will sequentially run the three specified commands (`build`, `flash`, `monitor`). The `build` command compiles the code and saves the compiled code to `build` directory. The `flash` command connects to the device, saves the device to the global state, and uploads the code. The `monitor` command uses the saved device to access its standard input and output without reconnecting.

Command-line options are divided into two types — global and command-specific. Options can be specified in any place of the command, as the parsing is done in multiple passes for each specified command. The global options are parsed first, then options of the first command are extracted, then the second, and so on. Each command then receives only its and the global options. For example, the following two commands would be parsed the same way:

```
jac --port /dev/ttyUSB0 build flash monitor
jac build flash monitor --port /dev/ttyUSB0
```

The commands can also specify positional arguments, which are parsed after the options:

```
jac --port /dev/ttyUSB0 read ./code/index.js
```

8.2.4 TypeScript code compilation

To allow the user to write their code in TypeScript, the command-line tool uses the TypeScript compiler to compile the code. Microsoft provides the compiler as an npm package called `typescript`, which can be used programmatically. The compiler is used to compile the code to JavaScript, which is then uploaded to the device.

The structure of a TypeScript project is described in the `tsconfig.json` file, which is loaded by the compiler. The file specifies, among other things, which files should be compiled, where the compiled files should be saved, and which ECMAScript version should be used for the output code.

Because the target runtime is known to the compiler, some options in the `tsconfig.json` file are constrained in preprocessing, such as the target ECMAScript version. If the user specifies an invalid value of these options, the compiler will throw an error.

8.3 Usage

The application is available as an npm package, which can be installed using the `npm` command-line tool:

```
npm install -g jaculus-tools
```

The `-g` option installs the package globally, which makes the `jac` command available in the terminal[2].

The commands listed in Section 8.1 can be run using the `jac` command.

2. The path to the global `npm` directory must be added to the `PATH` environment variable for the `jac` command to be available. Otherwise, it can be run using `npx jac`.

```
jac <command> [options] [arguments]
```

The `help` command can be used to get help for a specific command:

```
jac help <command>
```

9 Jaculus-esp32

By adding hardware bindings to Jaculus-dcore, it is possible to create firmware for a specific platform. The version created as a part of this thesis is Jaculus-esp32.

Jaculus-esp32 is a Jaculus device firmware for the ESP32 and ESP32-S3 microcontrollers. It uses the ESP-IDF framework and supports connection to a computer via a serial port.

9.1 Features

Aside from the features provided by Jaculus-dcore, Jaculus-esp32 only implements control over the most basic peripherals, which are:

- GPIO — general-purpose input/output pins
- ADC — analog-to-digital converter
- LEDC — generator of PWM signals
- Neopixel — WS2812B smart LED strip

Jaculus-esp32 also implements a specialized event queue based on a FreeRTOS queue and supports scheduling events from an interrupt context. This is used in the GPIO feature, which generates events when the state of a pin changes through an interrupt.

Implementation of the Neopixel MFeature uses the SmartLeds[10] library. The library also allows the control of other types of smart LED strips, but for simplicity, only the WS2812B type is supported by the Neopixel MFeature. The MFeature implementation also nicely demonstrates the use of Jaculus-machine for wrapping existing C++ libraries.

9.2 Usage

The firmware can be flashed to the device manually using ESP-IDF or the Jaculus-tools application described in the previous chapter.

The firmware uses the Jaculus-machine runtime at its core, meaning that adding new features to the runtime is done the same way described in Chapter 5.

9.2.1 JavaScript API

The runtime exposes the following modules to the JavaScript code:

- `stdio` — standard input and output
- `fs` — file system
- `path` — lexical path manipulation
- `gpio` — GPIO pins
- `adc` — analog-to-digital converter
- `ledc` — PWM generator
- `neopixel` — WS2812B smart LED strip

Aside from these modules, several global constants and functions are also available:

- `PlatformInfo` — information about the device
- `sleep`, `setTimeout`, `clearTimeout`, `setInterval`, `clearInterval` — functions for configuring timers
- `console` — console object for logging messages
- `exit` — function for stopping the program

While there is no browsable documentation, the API is documented in more detail in the type definitions files in the TypeScript examples directory attached to the thesis. The examples also demonstrate the usage of some of the modules.

10 Evaluation

Several requirements were outlined in the introduction. This section evaluates the solution based on those requirements.

10.1 Comparison with native programs

Compared to native programs, the solution significantly reduces the development cycle length. While the build-deploy-test process of native programs can, in worse cases, take several minutes, with Jaculus, it is reduced to several seconds. The build time shows the most considerable improvement, as no build process is needed for JavaScript programs, and the build time of TypeScript programs is mostly negligible compared to native programs. Deployment is also faster, as only the application code needs to be uploaded to the device, unlike native programs, where the entire firmware partition is often overwritten.

The shortened development cycle naturally comes at a price, that being the performance of JavaScript as an interpreted language. However, the performance is still sufficient for running most application logic. If a performance-sensitive part of the application is identified, it can be reimplemented in C++ and compiled into a native module.

10.2 Comparison with other interpreted solutions

There are already existing solutions to programming microcontrollers using JavaScript. While it would be nice to have a detailed comparison of at least some of them, it would stretch the scope of this thesis too far. However, a brief comparison of Jaculus with other solutions is provided in this section. These solutions were chosen based on their popularity:

- CircuitPython[11] (fork of MicroPython[12])
- NodeMCU[13] (Lua)
- Espruino[14] (JavaScript)
- Moddable SDK[15] (JavaScript)

A common characteristic of these solutions is their much larger feature set compared to Jaculus. They have not only much larger hardware and library support but also larger ecosystems and communities. For example, Espruino and CircuitPython provide very convenient development environments.

What sets Jaculus apart from the other solutions is its much higher-level abstraction around different JavaScript concepts. This allows for much easier wrapping of existing libraries and other code to JavaScript modules, meaning that while the hardware support of Jaculus is currently low, adding more features to the runtime is a quick and straightforward task.

10.2.1 Non-JavaScript solutions

Comparing different languages, especially their performance, is a notoriously difficult task, as each language has its own strengths and weaknesses. Therefore, the comparison focuses on the usability of the solutions.

MicroPython and CircuitPython are implementations of Python 3. They are very popular in the maker community and have a large ecosystem of libraries and tools. However, they only support a limited subset of the language, meaning that some Python code cannot be run on them, which could be seen as a disadvantage.

NodeMCU is a Lua interpreter for ESP8266 and ESP32 microcontrollers. On the one hand, it is a well-established project with an existing community. On the other hand, as mentioned in the introduction, Lua is not nearly as popular as the other languages, making it less attractive to new users.

10.2.2 JavaScript solutions

Comparing Jaculus with the other JavaScript solutions is more straightforward, as they all share the same language. The comparison focuses on their performance, implementation, and tooling.

Espruino is an open-source JavaScript interpreter for microcontrollers. The biggest problem of Espruino is its low performance, which can be seen in the performance comparison below. Espruino also implements only a limited subset of an older version of the ECMAScript

standard, meaning some JavaScript code cannot be run on it. Espruino also provides a convenient development environment through its Web IDE, which allows for live code editing and debugging and does not require any additional tools to be installed.

Moddable SDK is another JavaScript runtime for microcontrollers. It is a commercial product but is open-source and free for non-commercial use. Instead of interpreting the JavaScript code on the device, Moddable SDK compiles it into bytecode on the host machine, packs necessary native libraries, and builds a complete firmware image, which is then uploaded to the device. This approach allows for a smaller runtime footprint but causes a slightly longer development cycle and requires much more complex development tools.

The performance of these solutions was compared by running the same benchmarks on the ESP32 microcontroller. Aside from the GPIO benchmark, the benchmarks were taken from the Computer Language Benchmarks Game[16] and slightly modified for each platform to fit the provided API. The GPIO benchmark was created for this comparison and measured the time it takes to toggle a GPIO pin n times. The code of the GPIO benchmark for Espruino is shown below. The results of the comparison are shown in Table 10.1.

```
var pin = 5;
var start = Date.now();
for (var i = 0; i < n; i++) {
  digitalWrite(pin, 1);
  digitalWrite(pin, 0);
}
var end = Date.now();
console.log("Time: " + (end - start));
```

The benchmarks show that Jaculus is significantly faster in computational tasks than Espruino and Moddable SDK. Moddable SDK is faster in the binary-trees benchmark, which is targeted at the creation of new objects and memory allocation. Moddable SDK is also faster in the GPIO benchmark, which may be caused by poorly optimized abstraction used to implement the native module in Jaculus.

The low performance of Espruino can likely be attributed to the fact that internally, Espruino parses the JavaScript code into an abstract syntax tree, which is then directly interpreted. By contrast, the approach used by Jaculus and Moddable SDK is first to compile the

source code into bytecode and then interpret it. This approach allows for more optimizations to be performed on the bytecode and results in better performance.

Table 10.1: Performance comparison of Jaculus with other JavaScript solutions. The results are run times in milliseconds, lower is better.

Test	Jaculus	Espruino	Moddable SDK
n-body (n=50)	36.2	4333.1	54.4
fannkuch-redux (n=6)	35.6	12335.3	75.0
spectral-norm (n=10)	48.9	10749.9	91.4
binary-trees (n=3)	279.3	19930.4	187.6
GPIO (n=5000)	756.1	11720.4	54.7

11 Limitations

This chapter describes the limitations of the created components and the Jaculus-esp32 firmware. While the problems described here are not critical, they cause some inconvenience and should be addressed in the future.

11.1 Only one Context per Machine instance

From the beginning, the design of Jaculus-machine was to have only one Context per Machine instance. It seemed not to be a limiting factor and simplified implementation. However, later in development, it started to show it was a wrong decision and proves to be a limiting factor in some cases.

For example, in REPL, all exceptions should be caught and reported to standard output. When starting REPL from a JavaScript program, the main program should crash on unhandled exceptions, whereas the REPL should not. Implementation of this would require REPL and the main program to be executed in separate contexts to distinguish between their behavior regarding exception handling.

11.2 Unhandled promise rejections not being reported

This is a limitation of QuickJS. Although QuickJS does have a mechanism for reporting unhandled promise rejections, it reports some false positives. Consider the following example:

```
new Promise((resolve, reject) => {
  console.log("promise");
  reject(null);
}).then(() => {
  console.log("ok");
}).catch(() => {
  console.log("error");
});

console.log("after");
```

The promise is created and immediately rejected. At that moment, the promise does not have a rejection handler, and thus QuickJS reports an unhandled promise rejection. However, the handler is added before the promise goes out of scope and handles the rejection.

Because of the false positives, the mechanism for reporting unhandled promise rejections is disabled in Jaculus-machine.

Fortunately, this is not a problem for well-written code, which correctly handles all possible promise rejections. However, when an unhandled promise rejection occurs, it is not reported, which may lead to errors that are difficult to debug.

To fix this, modifying QuickJS internals would be required, which is outside the scope of this theis, but it is an essential consideration for future work.

11.3 Filesystem API

As the Filesystem MFeature is implemented using C++ `std::filesystem` API, which is not yet fully supported by the ESP-IDF, some of its functionality does not work, and some may even block the runtime indefinitely.

The broken functionality involves listing directories — `readdir` and `rmdir`. Fortunately, working with files works without a problem, which is an essential functionality for loading JavaScript files.

It would be possible to separately reimplement the MFeature to use the POSIX API, but it would make more sense to implement a more complex abstraction layer around the filesystem API, which could be used by other Jaculus components as well. However, that would require a significant amount of work, which was not strictly necessary for the functionality of Jaculus-esp32.

11.4 Compatibility with other platforms

While implementing all of the created components, I have focused on only using the standard C++20 library. This worked well for development, as all of the functionality could be tested locally on a desktop PC, and later everything could be easily integrated with the ESP-IDF. Unfortunately, after briefly exploring development options of plat-

forms other than ESP32 (STM32, RP2040), I have discovered that the C++20 standard is often not fully supported. Some platforms do not even fully support older standards, such as C++17.

This dealt a significant blow to the portability of these components, which would have to be partially rewritten to support other platforms. Mainly, an abstraction layer around the filesystem API and asynchronous elements (threads, synchronization primitives) would have to be implemented.

12 Conclusion

The goal of this thesis was to create an ecosystem for programming embedded devices using JavaScript.

The presented solution consists of Jaculus-dcore library and Jaculus-tools command-line application. The library provides the core functionality of a Jaculus device, and the application provides a way to interact with the device. The Jaculus-dcore library is also integrated into the Jaculus-esp32 firmware, which ports the solution to the ESP platform.

Two standalone libraries were also created as part of the solution: Jaculus-link and Jaculus-machine. The former is a communication library, and the latter is an implementation of the JavaScript runtime with easy extensibility. Both libraries are well documented and tested, and can be used independently from the rest of the solution.

Although the solution is usable and, in terms of performance, can compete with other existing solutions, many features are still missing, and some bugs are inevitably present. Most importantly, more MFeatures must be implemented for Jaculus to be a viable alternative to other existing solutions. Further improvements can also be made to the upload protocol, which is presently very simple and could have better performance and reliability. The limitations described in the previous chapter should also be addressed.

A Attachments

All source code created as a result of this thesis is available in the attached ZIP archive. The archive contains the following directories, each containing a snapshot of a git repository of the respective part of the implementation:

- `Jaculus-dcore`
- `Jaculus-machine`
- `Jaculus-link`
- `Jaculus-esp32`
- `Jaculus-tools`
- `QuickJS`

The `Jaculus-esp32/ts-examples` directory contains examples of TypeScript programs that can be run on the Jaculus-esp32 firmware.

Documentation for `Jaculus-machine` and `Jaculus-link` is available in the `docs` subdirectory of their respective directories. The project's homepage, with Getting Started and Troubleshooting guides, is available in the `docs` subdirectory of the `Jaculus-dcore` directory.

The homepage and the documentation are also hosted online at:

- `https://jaculus.org`
- `https://machine.jaculus.org`
- `https://link.jaculus.org`

An upstream version of the respective repositories is available on GitHub:

- `https://github.com/cubicap/Jaculus-machine`
- `https://github.com/cubicap/Jaculus-link`
- `https://github.com/cubicap/Jaculus-tools`
- `https://github.com/cubicap/Jaculus-dcore`
- `https://github.com/cubicap/Jaculus-esp32`
- `https://github.com/cubicap/quickjs`

B Building and flashing Jaculus-esp32

The Jaculus-esp32 firmware can be built using the ESP-IDF version 5.0.1 or newer. For the setup of the ESP-IDF, please refer to the official documentation[1].

The ESP-IDF can be used from the command line. To set up the environment variables for the ESP-IDF, run the corresponding export script in the ESP-IDF directory:

- Linux: `source export.sh`
- Windows: `export.bat` or `export.ps1`

Select the target platform by renaming the corresponding configuration file in the `Jaculus-esp32` directory of the project to `sdkconfig`:

- ESP32: `sdkconfig-esp32`
- ESP32-S3: `sdkconfig-esp32s3`

To build the firmware, run `idf.py build` in the `Jaculus-esp32` directory. To flash the firmware to the device, run `idf.py flash`.

Note that the build process needs to fetch the Jaculus-dcore, Jaculus-machine, Jaculus-link, and QuickJS dependencies from GitHub and requires an internet connection. To install the dependencies manually, copy their source directories to the `Jaculus-esp32/components` directory:

- `Jaculus-dcore/src` →`components/jac-device`
- `Jaculus-machine/src` →`components/jac-machine`
- `Jaculus-link/src` →`components/jac-link`
- `quickjs` →`components/quickjs`

1. In: ESP-IDF Programming Guide [online]. [visited on 2023-05-16]. Available from: `https://docs.espressif.com/projects/esp-idf/en/v5.0.1/esp32/get-started/`

C Building Jaculus-tools

Jaculus-tools is developed in TypeScript and requires Node.js version 18 or newer. To build the application, first install its development and runtime dependencies by running `npm ci` in the `Jaculus-tools` directory. Then, build the application by running `npm run build`.

The application can be run using `npx jac` in the `Jaculus-tools` directory. To install the application globally, run `npm link` in the `Jaculus-tools` directory. This will create a symbolic link to the application in the global `node_modules` directory, allowing it to be run from anywhere using `npx jac` (or `jac` if the global `node_modules` directory is in the `PATH` environment variable).

Bibliography

1. THE CHROMIUM PROJECT. *V8 JavaScript Engine* [online]. 2023. [visited on 2023-09-13]. Available from: `https://v8.dev/`.
2. CONTRIBUTORS TO THE DUKTAPE PROJECT. *Duktape* [online]. 2022. [visited on 2023-09-13]. Available from: `https://duktape.org/`.
3. ARTIFEX SOFTWARE, INC. *muJS* [online]. 2020. [visited on 2023-09-13]. Available from: `https://mujs.com/`.
4. BELLARD, Fabrice; GORDON, Charlie. *QuickJS Javascript Engine* [online]. 2021. [visited on 2023-09-05]. Available from: `https://bellard.org/quickjs/`.
5. BELLARD, Fabrice. *QuickJS Benchmark* [online]. 2019. [visited on 2023-09-14]. Available from: `https://bellard.org/quickjs/bench.html`.
6. ESPRESSIF SYSTEMS. *ESP32 Series Datasheet* [online]. 2023. [visited on 2023-09-16]. Available from: `https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf`.
7. ESPRESSIF SYSTEMS. *ESP32-S3 Series Datasheet* [online]. 2023. [visited on 2023-09-16]. Available from: `https://www.espressif.com/sites/default/files/documentation/esp32-s3_datasheet_en.pdf`.
8. ECMA INTERNATIONAL. *ECMA-262, 11th edition, June 2020 ECMAScript® 2020 Language Specification* [online]. 2020. [visited on 2023-09-05]. Available from: `https://262.ecma-international.org/11.0/`.
9. CHESHIRE, S.; BAKER, M. Consistent overhead byte stuffing. *IEEE/ACM Transactions on Networking*. 1999, vol. 7, no. 2, pp. 159–172. Available from DOI: `10.1109/90.769765`.
10. ROBOTICSBRNO. *SmartLeds* [online]. 2023. [visited on 2023-09-13]. Available from: `https://github.com/RoboticsBrno/SmartLeds`.

11. ADAFRUIT INDUSTRIES. *CircuitPython* [online]. 2023. [visited on 2023-09-13]. Available from: https://circuitpython.org/.

12. GEORGE ROBOTICS LIMITED. *MicroPython* [online]. 2023. [visited on 2023-09-13]. Available from: https://micropython.org/.

13. NODEMCU TEAM. *NodeMCU* [online]. 2018. [visited on 2023-09-13]. Available from: https://www.nodemcu.com/index_en.html.

14. PUR3 LTD. *Espruino* [online]. 2023. [visited on 2023-09-13]. Available from: https://www.espruino.com/.

15. MODDABLE TECH, INC. *Moddable* [online]. 2023. [visited on 2023-09-13]. Available from: https://www.moddable.com/.

16. *The Benchmarks Game Debian Alioth Archive* [online]. 2018. [visited on 2023-09-13]. Available from: https://salsa.debian.org/benchmarksgame-team/archive-alioth-benchmarksgame.