Document Title	Explanation of ARA Applications in Rust
Document Category	Explanation (EXP)
Document UID	1079
Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R26-11

Document Change History			
Date	**Release**	**Changed by**	**Description**
2026-11-26	R26-11	AUTOSAR Release Management	• No content changes
2025-11-27	R25-11	AUTOSAR Release Management	• No content changes
2024-11-27	R24-11	AUTOSAR Release Management	• No content changes
2023-11-23	R23-11	AUTOSAR Release Management	• Initial release

Disclaimer

This work (specification and/or software implementation) and the material contained in it, as released by AUTOSAR, is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the work.

The material contained in this work is protected by copyright and other types of intellectual property rights. The commercial exploitation of the material contained in this work requires a license to such intellectual property rights.

This work may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only. For any other purpose, no part of the work may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The work has been developed for automotive applications only. It has neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Table of Contents

1 Introduction 5
1.1 Motivation . . . 5
1.2 Rust Binding Advantages . . . 5
1.3 Binding Methodology . . . 6
1.3.1 Layered architecture . . . 6
2 Definition of terms and acronyms 8
2.1 Acronyms and abbreviations . . . 8
2.2 Definition of terms . . . 8
3 Related Documentation 9
3.1 Input documents & related standards and norms . . . 9
4 Rust language tutorials 10
4.1 Tool installation . . . 10
4.2 Which Advanced Rust concepts to know . . . 10
4.2.1 Asynchronous Rust: futures, streams, tasks . . . 10
4.2.1.1 `Async` operations and executors . . . 11
4.2.1.2 The `Future` trait . . . 12
4.2.1.3 Tasks . . . 13
4.2.1.4 Main function usage . . . 13
4.2.1.5 I/O operations . . . 14
4.2.1.6 Streams . . . 14
4.2.1.7 Synchronization . . . 15
5 Tour of AUTOSAR with Rust 17
5.1 AUTOSAR Rust API overview . . . 17
5.2 Finding the example code . . . 17
5.2.1 Generating the latest documentation . . . 18
5.3 Minimal example . . . 18
5.4 Writing a simple ara::com client . . . 20
5.5 Implementing an ara::com service . . . 21
5.6 Creating a new ErrorDomain . . . 23
6 In depth discussion 25
6.1 Application linking . . . 25
6.2 Foreign Function Interface types . . . 25
6.3 Rust to C++ . . . 26
6.4 ara::com binding generation . . . 26
6.4.1 Proxy code . . . 27
6.4.1.1 C++ side . . . 27
6.4.1.2 Rust side . . . 27
6.4.2 Skeleton code . . . 28
6.4.2.1 C++ side . . . 28

6.4.2.2 Rust side . . . 28
6.5 Application lifecycle . . . 29
6.6 UML and Rust . . . 29
6.6.1 Module . . . 30
6.6.2 Struct . . . 30
6.6.3 Enumeration . . . 31
6.6.4 Type alias . . . 31
6.6.5 Traits and impl blocks . . . 32
6.6.6 Generics . . . 33
6.7 Advanced concepts . . . 35
6.7.1 Ownership . . . 35
6.7.2 Structured concurrency . . . 35
6.7.3 Detailed event handling . . . 36
7 Recommend further readings 39
7.1 Recommended . . . 39
7.2 Outlook . . . 39
7.2.1 WebAssembly Interface Types . . . 40
A Appendix 41

1 Introduction

This document introduces the current proposal for a programmer's interface to write AUTOSAR Adaptive applications in the Rust programming language. This API was created within the AUTOSAR Rust working group.

The target audience of this document is application developers who want to write Adaptive applications in the Rust programming language. Some level of familiarity with Adaptive AUTOSAR concepts and its C++ application programmer interface is assumed.

1.1 Motivation

Challenges of C++ usage for automotive like memory management complexity, multithreading complexity, default copying semantics, absence of out-of-range detection motivate us to introduce AUTOSAR Adaptive Rust binding. Its purpose is to utilize Rust advantages for AUTOSAR Adaptive platform usage and potential future structural changes.

Rust is a relatively recent programming language with modern tooling. It is convenient, fast-growing and understandable for C++ developers. It brings improvements in many areas that have traditionally been hard to manage in C++. Rust also has advanced compile time checking which can prevent whole categories of common but non-trivial errors in code.

1.2 Rust Binding Advantages

The introduction of Rust aims to address the following challenges with C++:

Use cases	C++ approach	Rust approach
Language design focus	Speed and maximum flexibility	Correctness with good enough speed
Memory management	Speed-optimized manual memory management	Formalized ownership based memory management, checked at compile time
Data protection for multithreaded application	Association between critical sections and data by documentation, checked by runtime instrumentation	Send and Sync restrict thread local and concurrent access, checked at compile time
Object semantics	Objects are copied by default, aliasing between pointers	Objects are moved by default, read-write interference is prevented across modules
Out of range detection for array-like data type	No boundary check by default. Checking bounds is the more verbose variant.	Always checked at compile or runtime, unchecked access is verbose and requires careful review
Object initialization	Initialization checked by static analyzers	Object initialization required by compiler
Modules	Introduced in C++20	Modules are a core language feature. Multiple versions of the same dependency in a single binary are possible
Stackless co-routines	`co_await` since C++20	`async` since 2018

△

Macros	Macros bypass syntax checks and have non-local effects	Declarative and procedural types of macros. Enable introspection and variable number of arguments
Type casting	Implicit casting prevented by MISRA checks	Only explicit type casting[1]
Code checking and review	Static code checking with local reasoning, manual review of global correctness	Compile time checks global correctness, unsafe code requires manual module-local reviews
Ecosystem	Variety of code checkers, dependency management and build systems, code styles and test harnesses	Single widely accepted solution: Clippy, cargo, rustfmt

1.3 Binding Methodology

Despite availability of several Rust binding generators, we introduce manually implemented Rust bindings for AUTOSAR Adaptive APIs to provide community agreed and reviewed bindings, and also avoid inconsistency of automated data types conversion between languages. This approach may change in the future as Rust binding generators become more feature rich and production ready. In our Rust binding methodology, we would like to utilize the best of both worlds, top-down (package design drives implementation) design and bottom-up (interface implementation leads to usable packages structure) to introduce Rust bindings that will be easy to understand, fast to adapt to new projects and minimize incorrect usage cases.

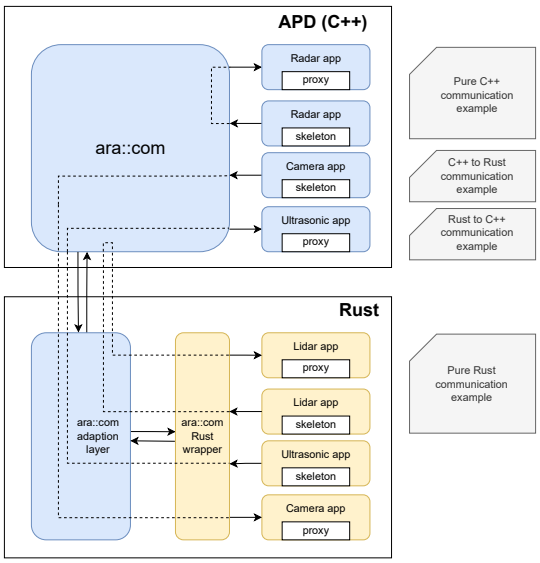


Figure 1.1: Current AUTOSAR-Rust communication diagram

1.3.1 Layered architecture

Starting with a simple application, omitting ARXML generated code for now, we see that Rust application support needs numerous layers. The Rust API adapter can't directly

[1] Error propagation will automatically convert the error type.

talk to the AUTOSAR C++ API, because there is no common binary interoperability between Rust and C++ objects. A language independent binary interface, an ABI, is needed.

One option[2] is to define this interface with C compatible data types and calling conventions, because the standardized binary interface for the language C is accessible from both C++ and Rust. The current implementation defines this interface on the Rust side and uses `cbindgen` to generate C(++) headers used by the C++ adapter.

Please note that there exists no C code for this purpose, it merely uses the C compatible subset of Rust and C++ to interoperate.

<table>
<tr><th colspan="2">Rust Application</th></tr>
<tr><td>AUTOSAR Rust API
AUTOSAR Rust adapter</td><td rowspan="3">Rust standard library</td></tr>
<tr><td>Language neutral binary interface (ABI)
Language neutral to C++ adapter</td></tr>
<tr><td>AUTOSAR C++ API
AUTOSAR C++ stack</td></tr>
<tr><td colspan="2">POSIX PSE51 API
Operating system</td></tr>
</table>

Table 1.1: Layers of the Rust binding

Zooming in on the AUTOSAR specific part, it consists of application independent code and ARXML generated proxy and skeleton code. The middle columns contain three separate files per skeleton or proxy which are generated from ARXML:

<table>
<tr><th colspan="4">Rust Application</th></tr>
<tr><td colspan="4">AUTOSAR Rust API</td></tr>
<tr><td>App independent Rust adapter</td><td>Proxy Rust adapter</td><td>Skeleton Rust adapter</td><td>...</td></tr>
<tr><td colspan="4">Language independent binary interface (ABI)</td></tr>
<tr><td>App independent ABI to C++ adapter</td><td>Proxy ABI to C++ adapter</td><td>Skeleton ABI to C++ adapter</td><td>...</td></tr>
<tr><td colspan="4">AUTOSAR C++ API</td></tr>
<tr><td>AUTOSAR C++ stack</td><td>Stack specific proxy code</td><td>Stack specific skeleton code</td><td>...</td></tr>
</table>

Table 1.2: Rust binding with ara::com

[2]See the outlook section 7.2.1 for a future option to define this interface for multiple languages at once.

2 Definition of terms and acronyms

Acronyms and abbreviations which have a local scope and therefore are not contained in the AUTOSAR Glossary [1]:

2.1 Acronyms and abbreviations

Abbreviation / Acronym:	Description:
ABI	Application Binary Interface
ANSSI	Agence nationale de la sécurité des systèmes d'information
APD	Adaptive Platform Demonstrator
API	Application Programming Interface
ARA	AUTOSAR Adaptive (Platform)
FFI	Foreign Function Interface
I/O	Input/Output
MDG	Model Driven Generation
SDK	Software Development Kit
TDD	Test Driven Development
UML	Unified Modeling Language

Table 2.1: Acronyms and abbreviations used in the scope of this Document

Note: `ABI`[1] and `API`[2] are easily confused.

2.2 Definition of terms

Definition of terms which are not self-explaining and are needed to understand the explanations in this document.

Terms:	Description:
async	asynchronous
impl	implementation

Table 2.2: Definition of terms in the scope of this Document

[1] An ABI is a low level representation defined for a specific platform to interact between binary modules. These modules can be created from different programming languages.

[2] An API is a high level abstraction created for programmers.
If this API also maps to an ABI, an API definition can be sufficient to combine multiple languages.
To communicate over a network a common binary representation of data beyond a specific ABI is needed.

3 Related Documentation

3.1 Input documents & related standards and norms

Programming rules for secure applications by ANSSI [2]

High Assurance Rust standard [3]

SAE International good practices [4]

Qualified toolchain for Functional Safety and matching specification [5]

4 Rust language tutorials

This chapter first introduces the Rust programming language and then the more recently standardized asynchronous Rust used for the AUTOSAR binding. Asynchronous Rust is equivalent to `co_await` in modern C++.

Rust is a complex multi-paradigm programming language. While it might take a while to learn it compared to Python or Swift, it pays off when designing large-scale projects with high stability and security requirements. There is a number of free books and tutorials available on the Internet:

- The official documentation [6]
- A good free course [7]
- C and C++ developers might find a lot of familiar concepts like manual memory management, `RAII`, destructors. They can read [8] and [9] to speed up the learning process.
- The Rust cheat sheet [10]

4.1 Tool installation

We recommend using the version of the toolchain (rustc/cargo) included in the Rust-enabled ARA SDK. See 5 for the details.

4.2 Which Advanced Rust concepts to know

The Rust binding to AUTOSAR Adaptive uses asynchronous programming to minimize the number of necessary threads and context switches. So Asynchronous Rust is an advanced topic end users should be familiar with.

4.2.1 Asynchronous Rust: futures, streams, tasks

The main source of information about asynchronous Rust is [11]. Here is a short overview of its features.

While the identically named C++ feature `std::async` is typically based on a thread pool, Rust's `async` is the equivalent to C++-20's `co_await`.

Using the AUTOSAR specific `ara::core::Future::then` extension to the C++ `std::future` already reduces context switches and is more efficient than a blocking method invocation, but also reduces readability by introducing nested callbacks. Coroutines, like `co_await` and `async` Rust functions, can re-linearize this callback code again.

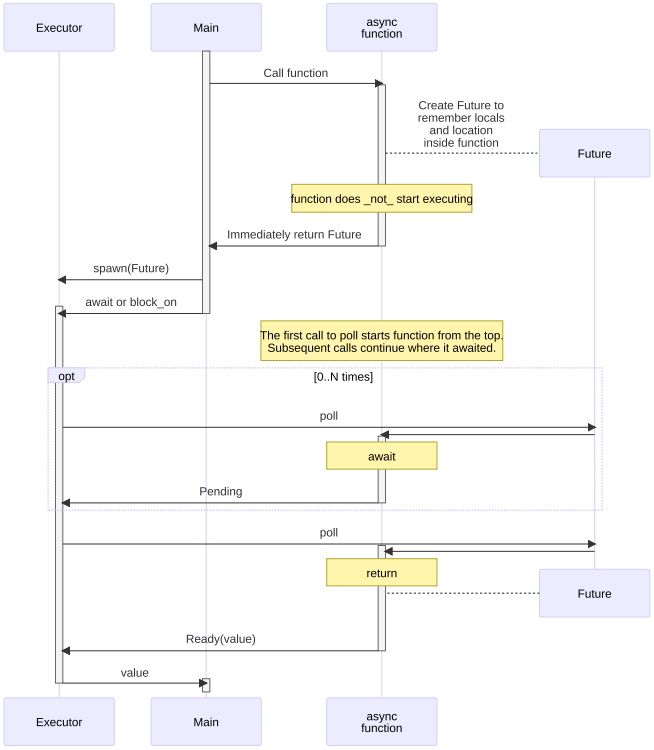


Figure 4.1: `async` function execution

Diagram 4.1 shows how the state machine represented in the `Future` data structure enables `async` functions remember their state. So in a way which maintains source-code readability, `async` automatically resumes execution at the last wait point across invocations of `poll` until the function finally returns a value.

Since its introduction to Rust in 2018 `async` became a commonly taught and used language feature. It also showed unique performance advantages in microcontroller environments (see [12]), which looks promising in context of future AUTOSAR Classic Rust bindings.

4.2.1.1 `Async` operations and executors

Rust supports asynchronous operations to allow time-consuming operations like I/O interactions (network, files) or procedures dependent on waiting for external events to perform without blocking program execution. `Async` operations rely on an executor, which polls asynchronous operation result to check if progress could be made or if the whole operation has finished.

There are four main Rust packages implementing `async` executors and features for `async` operations:

- ***tokio***
- ***async-std***

- ***smol***
- ***embassy***

tokio and ***async-std*** are the most popular, general-purpose runtimes providing well developed tools for asynchronous and concurrent operations like `Future` bookkeeping, task scheduling, I/O operations and inter-task synchronization.
tokio seems to be more mature and stable. It also covers more features like `select` for concurrency, however ***async-std*** appears to be faster, so a choice between the two depends on importance of high performance.

smol is set of basic `async` packages made in a way to improve speed and space efficiency. It also supports essential `async` runtime features like task spawning, executor implementation and I/O handler, but its overall possibilities are more limited than e.g. ***tokio***.

embassy is a bare metal abstraction layer providing features to perform asynchronous operations on bare metal embedded systems (typically lacking an operating system). Because embassy is very low-level, asynchronous I/O processes could be done in cooperation with hardware abstractions directly. For example, it has API for timers usage for queuing and sleep in `async` operations. It also could use interrupts as awakening mechanisms. Besides that embassy implements similar features as the rest of the runtimes: executor, task spawner etc.
Since embassy is specified for bare metal systems, and ***AUTOSAR Adaptive*** is based on POSIX API and abstraction, it will not be discussed further in this document.

4.2.1.2 The `Future` trait

The Executor's polling entry point and indicator for the state of the asynchronous operation is a concept called a `Future`.
A `Future` represents an operation that may not have produced its value yet. This kind of "asynchronous value" makes it possible for a thread to continue doing useful work while it waits for the value to become available. The core method of `Future`, `poll`, attempts to resolve the `Future` into a final value. This method does not block if the value is not ready. Instead, the current operation is scheduled to be woken up when it's possible to make further progress by polling again. The context passed to the `poll` method can provide a `Waker`, which is a handle for waking up the current task.

```
1 fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output>
```

This function returns:

`Poll::Pending` if the `Future` is not ready yet
`Poll::Ready(val)` with the result val of this `Future` if it has finished successfully.

`Future`s alone are inert; they must be actively polled to make progress, meaning that each time the current operation is woken up, it should actively re-poll pending `Future`s that it still has an interest in.

Example:

```
// `foo()` returns a type that implements `Future<Output = u8>`.
// `foo().await` will result in a value of type `u8`.
async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {
    // This `async` block results in a type that implements
    // `Future<Output = u8>`.
    async {
        let x: u8 = foo().await;
        x + 5
    }
}
```

Inside the `async` block, the `Future` for function `foo` will be polled (with the `await` keyword) until it finishes and returns a result.

4.2.1.3 Tasks

Inside an `async` executor we can spawn tasks, which are concurrent units of execution with their own context. Unlike threads, tasks are handled inside executor thread, where they are polled and managed (without creating/switching threads).
Spawning tasks is available for each listed packages, in similar manner:

```
handler = tokio::spawn(async{
//async closure
});
```

or

```
handler = async_std::task::spawn(async{
//async closure
});
```

Tasks have to be spawned in the context of an executor, and produce `Future`s which then get polled by the executor.

4.2.1.4 Main function usage

The life-cycle of an asynchronous Rust application starts by first creating the `async` executor instance and having it run a top-level task to completion in a blocking manner. This top-level task can only complete when all of its arbitrarily nested sub-tasks have also completed. The common way is to do this in the main function.

Tokio example:

```
fn main() {
let runtime = tokio::runtime::Runtime::new();
runtime.block(async {
    //asynchronous operations
    });
```

```
}
```

For convenience, we can use a macro to create an equivalent structure with simplified code:

```
#[tokio::main]
async fn main() {
    //asynchronous operations
}
```

Similar declarations could be done for the other packages.

4.2.1.5 I/O operations

Async allows for the continuation of program execution during time-consuming external operations like network communication or file read/write.
All `async` runtime packages implement functions to interact with I/O in a non-blocking way.

For example `async` accessing a file:

```
#[tokio::main]
async fn main() -> io::Result<()> {
    let mut f = File::open("foo.txt").await?;
}
```

These packages also include features to simplify performing `async` read/write operations, especially when we have multiple connections assigned to multiple tasks, and all of them could perform reads and/or writes in parallel.

```
let socket = TcpStream::connect("127.0.0.1:6142").await?;
let (mut rd, mut wr) = io::split(socket);

// Write data in the background
tokio::spawn(async move {
    wr.write_all(b"hello\r\n").await?;
    wr.write_all(b"world\r\n").await?;
})?;
```

and somewhere else:

```
loop {
    let n = rd.read(&mut buf).await?
}
```

4.2.1.6 Streams

A stream is an asynchronous series of values. It is the asynchronous equivalent to Rust's `std::iter::Iterator` and is represented by the Stream trait. Streams can be iterated in `async` functions. They can also be transformed using adapters.

Currently, the Rust programming language does not support `async` for loops. Instead, iterating streams is done using a while let loop paired with `StreamExt::next()`.

```
use tokio_stream::StreamExt;

#[tokio::main]
async fn main() {
    let mut stream = tokio_stream::iter(&[1, 2, 3]);

    while let Some(v) = stream.next().await {
        println!("GOT = {:?}", v);
    }
}
```

4.2.1.7 Synchronization

All `async` runtime packages provide methods for asynchronous resource sharing and inter-task communication similar to multithreading features (`async` mutex guards and channels).

Channel communication creates transmitter and receiver (producer and consumer) to exchange data between tasks. Channels can have a single producer and multiple consumers, multiple producers and a single consumer, or be broadcast. Receiver tasks wait asynchronously for messages to pass.
Example:

```
use tokio::sync::mpsc;

async fn some_computation(input: u32) -> String {
    format!("the result of computation {}", input)
}

#[tokio::main]
async fn main() {
    let (tx, mut rx) = mpsc::channel(100);

    tokio::spawn(async move {
        for i in 0..10 {
            let res = some_computation(i).await;
            tx.send(res).await.unwrap();
        }
    });

    while let Some(res) = rx.recv().await {
        println!("got = {}", res);
    }
}
```

Mutexes could also be used for task synchronization. However, a locked `async mutex` will yield context of the task and not block the thread.
Example:

```rust
async fn main() {
    let data1 = Arc::new(Mutex::new(0));
    let data2 = Arc::clone(&data1);

    tokio::spawn(async move {
        let mut lock = data2.lock().await;
        *lock += 1;
    });

    let mut lock = data1.lock().await;
    *lock += 1;
}
```

5 Tour of AUTOSAR with Rust

This chapter explains the Adaptive Platform Rust binding for applications.

Section 5.1 describes some higher level concepts of the API, section 5.2 points to the existing examples and documentation, sections 5.3 to 5.6 guide through adaptive Rust application code, starting at the basic APIs, then introducing `ara::com` and finally creating a new `ErrorDomain`.

5.1 AUTOSAR Rust API overview

The Adaptive Platform Rust binding is an example of integrating Rust-based applications into Adaptive AUTOSAR ecosystem (APD in this particular case). It can be used to develop AUTOSAR applications in Rust, which could bring enhanced memory safety combined with high performance and modern high-level abstractions. It may prove especially useful when building data processing, network protocols, command line interfaces, security applications or virtually any long-running services with high requirements towards robustness, memory footprint, and availability. Also, it is possible to port existing ARA-based applications to Rust, as it was done with `radar` and `fusion` sample applications.

The Rust binding grants access to the most important parts of ARA API, such as:

- Core Types: String, Result, etc.
- Execution Management
- Logging
- Communication

Other clusters could also be added in the future upon request.

The Rust API mostly resembles the existing the C++ counterpart while following Rust best practices. The resulting applications can communicate with other AUTOSAR services by means of ARA Communication API. In 5.2.1, API documentation generation is explained in details.

In the next sections, we will try to show how to build ARA Rust applications from scratch.

5.2 Finding the example code

The AUTOSAR Adaptive Demonstrator examples have a `rust` subfolder. Within this folder there is the `ara` folder containing the Rust adapter for AUTOSAR Adaptive APIs, an `examples` folder containing several example applications and a `UML` folder contain-

ing styles for `Enterprise Architect`. Each of these Rust folders have C++ and Rust sources built with CMake and cargo respectively. (`cpp` and `src` folders.)

5.2.1 Generating the latest documentation

Rust libraries are typically documented within the source code similarly to `doxygen` tool for C++. To generate documentation for the `ara` package navigate to the ara subfolder and run

```
cargo doc --no-deps --open
```

Here creating the documentation for dependencies is skipped and the generated documentation is opened in a web browser.

5.3 Minimal example

First, let's install a Rust-enabled version AUTOSAR Adaptive SDK a.k.a. ARA SDK. Here we'll assume it's installed to the directory `/opt/sdk`. Next, we need to initialize the SDK by running:

```
source /opt/sdk/environment*
```

Also, there is an optional dependency on the `clang-format` that formats the C++ source code. If it is not installed, the generated C++ code will be left unformatted. It can be set up by means of the packet manager of the development machine. For instance, Ubuntu users could use the following command:

```
sudo apt install clang-format
```

Executing `cargo new project_name` will start a new Rust project. Let's name it `fusion_test`. Then, let's add the `ara` and `log` dependencies:

```
[dependencies]
log = "0.4.17"

[dependencies.ara]
version = "0.1.0"
path = "/opt/sdk/sysroots/core2-32-poky-linux/usr/local/src/ara_rust"
```

Here, `core2-32-poky-linux` is the target system root directory of a 32-bit Intel i686 version of the SDK. It will be different for other targets. You can identify it by looking up the `SDKTARGETSYSROOT` environment variable, e.g., `echo $SDKTARGETSYSROOT`.

Within `main` `ara::core::initialized` gives you lifetime constrained access to logging (via the `create\_logger` method) and takes care of automatic initialization and deinitialization of the AUTOSAR Adaptive stack:

```
fn main() -> ara::core::MainResult<()> {
    ara::core::initialized(|env| {
        // ...
```

```
        Ok(())
    })?;
    Ok(())
}
```

The `ara` package provides logging macros similar to the standard Rust logging facade [1] with an additional logger `target:`. This gives access to switching between multiple loggers in the program.

```
ara::info!(target: self.logger, "Initializing Component...");
```

The method `set_default` connects a logger to the normal Rust logging macros which don't require a target:

```
env.set_default(env.create_logger("Ttrl",
            "Demonstrator Tutorial", LogLevel::Verbose))?;
log::info!("Starting DemonstratorTutorialRust...");
```

Reporting the state to the execution manager is supported via `ara::core::Environment::report_execution_state_running` call:

```
env.report_execution_state_running();
```

The async equivalent to `initialized` is `async_main` which enables `await` from within this main closure:

```
fn main() -> ara::core::MainResult<()> {
    match ara::core::async_main(|env| async move {
        // ...
        Ok(())
    }) {
        Err(ara::core::rust::MainError::Terminated) => Ok(()), // not an error
        r => r,
    }
}
```

Within blocking code, `std::thread::spawn` and `std::thread::sleep` work as usual:

```
let thread = std::thread::spawn(|| { // ...
    std::thread::sleep(Duration::from_millis(1000));
    // ...
});
```

while async programs should use `env.spawn` and `ara::core::sleep` instead

```
let task = env.spawn(async move { // ...
    ara::core::rust::sleep(Duration::from_millis(400)).await;
    // ...
});
```

`cargo build` will compile and link the program.

[1] For Rust `Logging facade` see the `log` crate at [13]

5.4 Writing a simple ara::com client

In order to communicate with other APD programs, an application is supposed to have ARXML configuration and be integrated into the APD machine configuration. In our case, the easiest option is to simply reuse the configuration of one of existing example applications. The SDK includes a tool called `aragen-rs` generating Rust projects out of ARA COM wrappers for a particular APD application. Let's generate one for the `fusion` application:

```
cd path/to/fusion_test
aragen-rs --app fusion -m RadarFusionMachine
```

This will create a new project `fusion_gen` containing a Rust package and a C++/CMake shared library project wrapping the `fusion` service API. Let's build it:

```
cd fusion_gen
./build.sh
```

After that, we will be able to link it to our example project. First, `fusion_gen` and `futures` dependencies should be added to `Cargo.toml`. Also, `[dependencies.ara]` path should now point the symlink `ara` inside the `fusion_gen` folder:

```
[dependencies]
futures = "0.3.21"
fusion_gen = { version = "0.1.0", path = "fusion_gen" }
...

[dependencies.ara]
version = "0.1.0"
path = "fusion_gen/ara"
```

Now we can implement the client code. To call methods and receive events from a service offered via the Communications framework `ara::com`, the first step is to connect to the service by means of the proxy code generated from ARXML, and then subscribe to events and call methods.

The first step is to look up an instance specifier:

```
let port_specifier = ara::core::InstanceSpecifier::new
        ("fusion/fusion/radar_RPort")?;
```

Note the error forwarding to the calling function by the question mark operator.

Then we can search for the service:

```
// add this line at the top:
use fusion_gen::radar_proxy::Proxy;

// and this is the service search code
let m_proxy = Proxy::find_service(
            env,
            ara::com::FindServiceParameter::InstanceSpecifier(
    port_specifier),
        ).await?;
```

Here, we use an enum (variant) to select the type of search (here `InstanceSpecifier`). As finding the service might take a while, we pass control back to the executor by `await`ing the result. The executor resumes the current task (there is no context switch: `async` tasks are implemented as state machines, purely in user space) once the service is found.

Calling a method looks similar (second `let`):

```
// add this line at the top:
use fusion_gen::types::{radar::AdjustOutput, Position};


// this code invokes the proxy method Adjust
let pos = Position{ x: 1, y: 2, z: 3};
let result = m_proxy.Adjust(pos).await;
match result {
    Ok(AdjustOutput{success, effective_position}) => {
        log::info!("Adjust result {success} {effective_position:?}");
    }
    Err(e) => {
        log::error!("Adjust failed {e:?}");
    }
}
```

Here we choose a position, call the method, await the result and handle potential errors.

Events are best modeled as a stream (a.k.a. generator or `async` iterator):

```
// add this line at the top:
use futures::stream::StreamExt;

// add this code receiving a stream of parkingBrake events from Proxy
let mut brake_subscription = m_proxy.parkingBrakeEvent().subscribe(3)?;
while let Some(item) = brake_subscription.stream.next().await {
    log::info!("ParkingBrakeEvent received: active={}", item.active);
}
```

Here we subscribe to a buffer of up to three samples, then loop over the events as they arrive.

If you want to process multiple events you can either create separate tasks per event:

```
let brake_task = env.spawn(async move { /* task code */ });
```

or `select` on multiple `next` calls.

5.5 Implementing an ara::com service

Services react to method calls and send events.

First, we'll need to create another Cargo project (see the previous sections) named `radar_test`. Next, we need to generate a communication project (see the section above for details):

```
cd path/to/radar_test
aragen-rs --app radar -m RadarFusionMachine
```

This will create a new project `radar_gen` containig a Rust package and a C++ shared library interfacing the `radar` service API. Let's build it:

```
cd radar_gen
./build.sh
```

Having done that, we can link it to our example project. First, `radar_gen` should be added to `Cargo.toml`. Also, the path to `ara` package must be updated:

```
[dependencies]
log = "0.4.17"
async-trait = "0.1.56"
radar_gen = { version = "0.1.0", path = "radar_gen" }
ara = { version = "0.1.0", path = "radar_gen/ara" }
...
```

Now we can start writing the service code. To implement methods, let's create an object implementing the skeleton trait:

```
use radar_gen::radar_skeleton;
use async_trait::async_trait;
use radar_gen::types::{
    radar::{AdjustOutput, CalibrateOutput},
    FusionVariant, Position,
};

struct RadarImp { /* member variables */ }

impl RadarImp {
    fn new() -> Self {
        Self { /* member variables */ }
    }
}

#[async_trait]
impl radar_skeleton::RadarSkeleton for RadarImp {
    async fn Adjust(&self, target_position: Position)
        -> ara::core::Result<AdjustOutput>
    { todo!() }

    async fn Calibrate(
        &self,
        configuration: ara::core::String,
        variant: FusionVariant,
    ) -> ara::core::Result<CalibrateOutput> { todo!() }

    async fn Echo(&self, text: ara::core::String) { todo!() }
}
```

Offer the service:

```
    let service = radar_skeleton::create_service(
        env,
```

```
        RadarImp::new,
        InstanceSpecifier::new("radar/radar/radar_PPort")?,
    )?
    .offer_service(radar_skeleton::RadarSkeletonDefaultValues::default())?;
```

To send events, simply use the skeleton's matching methods:

```
let mut event = service.parkingBrakeEvent_allocate().unwrap();
event.active = true;
event
    .objectVector
    .append(&mut vec![0x10, 0x20, 0x30, 42]);
event.send().unwrap();
```

5.6 Creating a new ErrorDomain

An `ErrorDomain` is a collection of error codes (usually in the form of an enum) offering conversion into human-readable text. To create an `ErrorDomain`, we require a unique domain identifier, which is typically centrally assigned during the vehicle design stage.

To turn an enum into an `ErrorDomain`, we simply implement the `ErrorDomain` trait for it. This makes it suitable for generating `ErrorCode` objects, which is the error type used all across AUTOSAR Adaptive.

```
use ara::c_string_ptr;
use std::sync::Mutex;
use ara::core::ErrorCode;
use once_cell::sync::Lazy;

#[repr(i32)]
#[derive(strum_macros::FromRepr, Copy, Clone)]
pub(crate) enum MyErrorDomain {
    NoInstanceFound,
}

impl ara::core::ErrorDomain for MyErrorDomain {
    type CodeType = MyErrorDomain;
    extern "C" fn name() -> *const u8 {
        c_string_ptr!("MyErrorDomain")
    }
    extern "C" fn message(elem: i32) -> *const u8 {
        match MyErrorDomain::from_repr(elem) {
            Some(MyErrorDomain::NoInstanceFound) => {
                c_string_ptr!("No Radar instance found")
            }
            None => c_string_ptr!("unknown"),
        }
    }
    fn id() -> u64 {
        // This number is assigned on a vehicle level
        1234
    }
}
```

```
30
31 impl From<MyErrorDomain> for i32 {
32     fn from(e: MyErrorDomain) -> Self {
33         e as i32
34     }
35 }
36
37 static INSTANCE: Lazy<Mutex<ara::core::rust::ErrorDomain<MyErrorDomain>>> =
38     Lazy::new(|| Mutex::new(ara::core::rust::ErrorDomain::<MyErrorDomain>::
    register()));
39
40 impl MyErrorDomain {
41     pub(crate) fn make_error_code(&self) -> ErrorCode {
42         INSTANCE.lock().unwrap().make_error_code(self.clone())
43     }
44 }
```

Please note that to convert from an integer to enum, we rely on the `strum-macros` package's `from_repr` method. `Clone` and `Copy` are natural properties of an integer and thus added as well.

As the Interface expects a static C string, we rely on the `c_string_ptr` macro to create the return values of name and message at compile time.

The conversion from enum to integer is required by the default trait implementation, so we use `once_cell::sync::Lazy` to easily register the static object once with C++.

Then using this set of error codes in AUTOSAR Adaptive becomes as easy as

```
1 return Err(MyErrorDomain::NoInstanceFound.make_error_code());
```

6 In depth discussion

This chapter briefly describes the internal adapter layers between the Rust application and the C++ stack.

Chapter 6.1 describes why a C interface is needed, chapter 6.2 describes the basic C interface types, chapter 6.3 describes the logic of the Rust to C layer, chapter 6.4 describes the `ARMXL` generated C interface, chapter 6.5 discusses program start and termination, chapter 6.6 discusses documenting Rust code with UML and chapter 6.7 explains ownership, lifetime and event timing.

6.1 Application linking

Currently, Rust projects that interact with C++ can utilize the `cxx` package. However, in our case, it would be really complicated since we have 2 different C++ runtimes involved: Rust is `LLVM` based while ARA API is build with `gcc`. This creates an ABI incompatibility issue, which was sorted out by introducing a C API isolation layer in between. Consequently, Rust applications link to the C++ shared libraries: `libara_rust.so` for general ARA API wrappers and application-specific `*_gen.so` ARA COM wrappers generated by the `aragen-rs` tool, see 6.4. The C++ code building is managed by `CMake` and is similar to C++ APD applications. Linking is done by the SDK-provided linker.

6.2 Foreign Function Interface types

This section presents the C structures which are used by both Rust and C++ for interoperability. They are defined in Rust code and then exported to C headers by `cbindgen`.

Wrapping a C++ type requires understanding its lifetime. If it is a read-only reference to a long living object, just passing the const pointer to a Rust object will do.

If a complex result of a function needs to be passed to Rust, a constant size object could be wrapped in a C `struct` and variable size objects need to be allocated on the heap requiring a destructor to be called on Drop.

- `ErrorCode` is of a constant size and typically part of Result
- `ResultWrapper` wraps the `Result` generic. It may contain either an `ErrorCode` or a Value, which has to be referenced from heap as the size is instantiation specific. If `obj` is `nullptr`, then the object contains an Error, i.e. `HasValue()` is false. There is a static `dummy_object` used for `ara::core::Result`. If this object is created on the Rust side, the obj pointer will be invalid if nonzero and require correction before use (as the object is directly stored in the ResultWrapper). This works around the missing stable interface to the Pin objects on the stack.

- `VectorView` gives read-only access to a vector reference, containing `.data()` and `.size()`
- `StringView` is an equivalent for character arrays
- `VectorHandle` (variable size) owns the elements contained; the C++ side uses `std::move` to pass ownership to Rust.
- `OwnedSamplePtr` is a family of types for sample pointer ownership. (Allocatee is the skeleton variant)
- `Promise<>` and `PromiseBase` wrap an `ara::core::Promise` object whose ownership is controlled by Rust
- `std::vector`: due to the C++ binary interface incompatibility described in 6.1, the `ara_core_Vector_` functions provide access to it from Rust using the AUTOSAR compiler

6.3 Rust to C++

Callbacks offered towards C++ need to be qualified as `extern "C"`. If you pass objects to the callback, `Box::into_raw` is the most elegant way to send an object via a C callback.

If the value and callback need to be used multiple times (e.g. event reception), the receiver/callback has to access the Box via pointer. Also, the surrounding infrastructure has to destroy the Box pointer after the last callback was made. For asynchronous waiting, `mpsc` channels and `StreamExt::next()` work well.

If the callback is known to happen exactly once (e.g. `find_service` or method invocation), the callback can use `Box::from_raw` to access it. It would have to free the allocated Box. For asynchronous waiting, oneshot channels provide a good solution.

6.4 ara::com binding generation

Each AUTOSAR Adaptive application is described in the ARXML format being part of an AUTOSAR Machine. The Communication wrappers (a C++ library and a Rust package) are generated by the `aragen-rs` command line tool, which is part of the SDK. It reads the ARXML configuration of a specified machine and generates all necessary glue code for a particular application. The primary source of ARXML configuration is the current SDK, which must be initialized beforehand, see 5.3. It also supports external ARXML sources (i.e. not a part of the SDK) including the whole machine configurations.

It is recommended to place the generated project inside the folder of the application using it. An example of how to link it is provided in 5.4.

6.4.1 Proxy code

To get the function name, replace SVC with the name of a service, MTD with the name of a method, EVT with the name of an event, and FLD with a field name.

6.4.1.1 C++ side

The interface between the Rust and C++ side uses the following functions:

- `SVC_proxy_StartFindService_` starts the search for a service; its arguments consist of the search parameters, a callback on changes to the list, and opaque user data.
  Generated for each proxy type.
- `SVC_proxy_StopFindService` stops a running service search (for `FindService`)
- `call_MTD` One function per method; arguments consist of proxy, the arguments to the method, plus a callback on completion and `user_data`
- `subscribe_EVT` registers a callback passing ownership of the sample pointer to Rust
- `unsubscribe_EVT` unregisters a registered event callback
- `get_FLD` requests the value of FLD
- `set_FLD` modifies the value of FLD

6.4.1.2 Rust side

The service discovery function creates an oneshot channel, calls `StartFindService` with proper parameters, and waits on the channel. The callback will call `StopFindService` and send the discovered service to the waiting task via the channel.

A Proxy offers member functions for event access, field access/manipulation, and method invocation.

An event is referenced via an Event structure which provides a subscribe method. Subscribing for an event creates an mpsc (MultiPublisherSingleConsumer) channel, which provides a Stream to read from. A callback is registered to forward received `OwnedSamplePtr`s to the channel. A subscription is kept alive by the handle object returned by a `subscribe_` call: dropping the object would automatically unsubscribe the client code from the event.

A field is accessed via the accessor methods of the Field structure. Depending on the field specification, it can support setting, getting, and subscription to field updates.

Similarly to events, field subscription produces a subscription handle unsubscribing the client from the field updates upon dropping it.

A method is simply invoked by calling the matching method of the Proxy.

6.4.2 Skeleton code

For each skeleton, a class is generated on the C++ side. It calls the connected Rust callbacks, which then call into the trait.

6.4.2.1 C++ side

The interface between the Rust and C++ side uses the following functions:

- `ProcessRequests` runs the dispatch loop for the incoming method calls
- `OfferService` starts offering the service
- `StopOfferService` stops offering the service
- `DestroyService` destroys the service instance
- `make_SVCImp` creates the service instance from function pointers
- `allocate_EVT` allocates an `OwnedSamplePtr` for data filling
- `send_EVT` sends the prepared data.
- `register_FLD_getter` registers a custom field getter.
- `update_FLD` updates the field value and broadcasts it to clients.
- `Promise_set_result` return a value from a method call (fulfills the Promise)
- `Promise_retain` moves the ownership of the return value Promise to Rust (the method `await`ed to complete)

6.4.2.2 Rust side

Until Rust supports `async` functions on traits (the most wanted feature for `async` support) we need to use the `async-trait` package which leverages heap allocation to work around this limitation.

The skeleton provides the callbacks to call the matching trait methods and a method to allocate sample pointers for an event.

The method callback will poll the `async` method once from the method handler context and if necessary (Pending) register the future with the executor. Please note that the self reference passed to the methods is constant, so if a method needs to modify state, it is bound to use internal mutability (e.g. via `Mutex` or `RwLock`). Otherwise, caching,

aliasing, and `async` make a proof of non-interference very difficult even for a single threaded case.

Since Rust futures are lazily evaluated, there is no easier way to postpone it to but require the copying of the arguments at the first await, thus the arguments are cloned and passed by value. Passing a reference would lead to unsoundness if the method implementation does not clone its input before the first await. Documenting this as a non-automatically checkable requirement on the function, which would void the benefits of Rust over C++.

Sample pointer allocation returns a handle offering access (`Deref`) to the elements and a send method consuming the handle.

6.5 Application lifecycle

The APD application lifecycle is the same for Rust and C++ applications9: the program is launched and shut down by the `Execution Management` service.

The APD execution environment is created by either `ara::core::initialized` (synchronous code) or `ara::core::async_main` (`async` code running the Rust ARA COM bindings). Once the application initialization is done, it should report with `env.report_execution_state_running` call, see 5.3. Then an endless main loop is usually entered.

The implementation uses `ctrl_c` package to support a graceful shutdown triggered by either SIGTERM or SIGINT signal. User code should `.await` on something so that the runtime could break the main loop on shutdown, e.g. `ara::core::rust::sleep`. Upon termination, `ara::core::async_main` returns a special error type `ara::core::rust::MainError::Terminated`, which is a marker of normal program termination. Also, it gracefully stops all the async tasks launched by `env.spawn`, `Proxy::spawn` or `OfferingService::spawn`. They should also contain some `.await` to be interruptible.

6.6 UML and Rust

UML stems from the object-oriented programming. While Rust supports some concepts of UML (e.g. trait is a UML interface; generics is a UML template), it also introduces new concepts such as the reference lifetime.

In this document, a subset of UML elements are used to describe the architecture. This appendix is a summary of elements mentioned in this document with their interpretation in Rust context.

To support the development of architecture pictures, an MDG technology of Enterprise Architect has been developed. It's included in the UML directory of the Rust example code.

6.6.1 Module

Rust modules are represented by means of the class stereotype ≪Module≫.

Functions and static variables can be represented as class operations and class attributes.

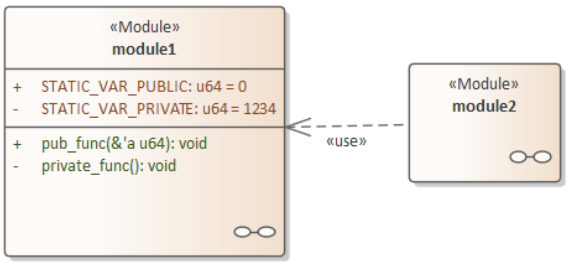


Figure 6.1: Module

```
// src/module1.rs

pub static STATIC_VAR_PUBLIC:u64=0;
static STATIC_VAR_PRIVATE:u64=1234;

pub fn pub_func<'a>(p:&'a u64)  {}
fn private_fun() {}
```

```
// src/module2.rs
use crate::module1;
```

6.6.2 Struct

Struct can be modeled using standard class element plus same stereotypes to express Rust specific semantic.

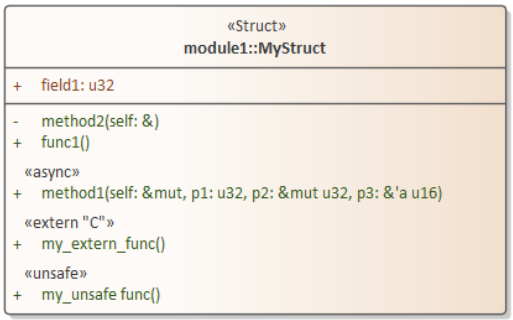


Figure 6.2: Struct and inherent implementation

Operations declared in the ≪struct≫ stereotype shall be interpreted as defined in an inherent implementation block.

```
struct MyStruct {
    pub field : u32
```

```
}

impl MyStruct {
    fn  method2 (&self) {}
    pub fn func1() {}
    pub async fn method1<'a> (&mut self, p1:u32,p2:&mut u32,p3: &'a u16)
   {}
    #[no_mangle]
    extern "C" fn my_extern_func() {}
    pub unsafe fn my_unsafe_func() {}
}
```

6.6.3 Enumeration

Enumeration can be represented the same way as in C language, although in Rust, variants can contain data. In this case, the UML operations can be used.

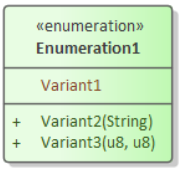


Figure 6.3: Enumeration

```
enum Enumeration1 {
   Variant1,
   Variant2(String),
   Variant3(u8,u8)
}
```

6.6.4 Type alias

Similarly to C `typedef`s, Rust type aliases are represented with `dependency` relationship.

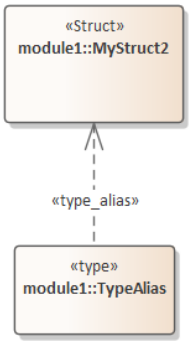


Figure 6.4: Type alias

```
type TypeAlias = MyStruct2;
```

6.6.5 Traits and impl blocks

Trait can be considered an interface from the UML point of view. Rust doesn't have the concept of implementing types: there is an implementation block that associates `implementing type` with `associated items`.

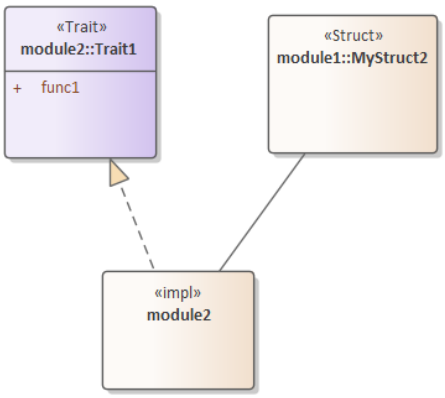


Figure 6.5: Trait and implementation

```
use crate::module1::MyStruct2;

trait Trait1 {
    fn func1();
}

impl Trait1 for MyStruct2 {
    fn func1() {}
}
```

The previous picture could appear cumbersome just to indicate the location of `impl` blocks. Frequently, showing the location of `impl` block is not necessary, so a simpler representation is possible as shown in the below picture. The great advantages of previous representation will become clear in 6.6.6 chapter. The stereotyped `realization` connector indicates that the semantic it is not exactly as in UML.

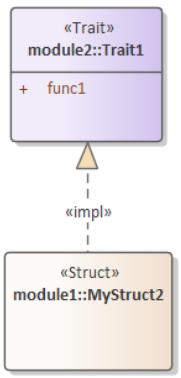


Figure 6.6: Simplified representation of trait and implementation

6.6.6 Generics

Rust generics are conceptually similar to C++ templates, therefore it is natural to represent Rust generics with the same UML representation.

In the images below, there is a simple example showing generics of a `struct` and a `trait`. The ≪`impl`≫ relationship means that there exists an implementation block that implements the trait for all possible types.

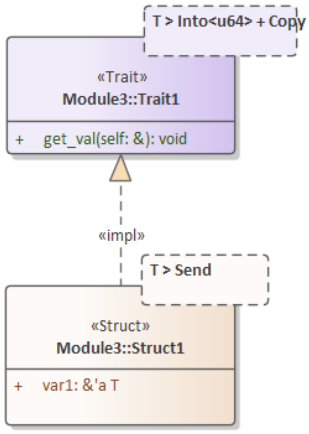


Figure 6.7: Simple representation of generic trait and implementation

```
pub trait Trait1<T>
where
    T: Into<u64> + Copy,
{
    fn get_val(&self) -> T;
}

pub struct Struct1<'a, T>
where
    T: Send,
{
    pub var1: &'a T,
```

```
}

impl<'a, T> Trait1<T> for Struct1<'a, T>
where
    T: Send + Into<u64> + Copy,
{
    fn get_val(&self) -> T {
        *self.var1
    }
}
```

When a more detailed representation is needed, the implementation can be placed separately from the trait and structure element.

E.g. when additional trait bounds should be expressed or when the location of the implementation is in the trait module instead of the struct module.

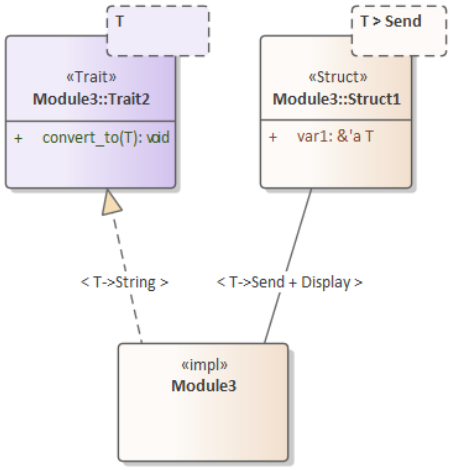


Figure 6.8: Detailed representation of generic trait and implementation

```
pub struct Struct1<'a, T>
where
    T: Send,
{
    pub var1: &'a T,
}

pub trait Trait2<T> {
    fn convert_to(self) -> T;
}

impl<'a, T> Trait2<String> for Struct1<'a, T>
where
    T: Send + Display,
{
    fn convert_to(self) -> String {
        self.var1.to_string()
    }
}
```

The representation may appear inconsistent compared to the type parameters. Rust lifetimes are syntactically represented as type parameters of generics and one can expect to represent them using UML template.

It has been decided to represent differently for the following reasons:

- Lifetimes are so widely used that pictures would become too complex
- Lifetimes have very different semantic compared to type parameters of generics

6.7 Advanced concepts

6.7.1 Ownership

Passing ownership of an object to a function is typically modelled in C++ by an `rvalue` reference (`&&`) on the callee side and `std::move` on the calling side. This added complexity makes this form of calling convention unpopular in C++ code, also because a borrowing call (`&` or `const &`) creates less effort on the caller's side.

For Rust passing ownership is the default calling convention unless you explicitly borrow the argument, either mutably or constant, to the callee for a limited amount of time. Typically, this borrowing ends once the called function returns.

So, sending a complex object to a single receiver will prefer passing ownership, sending an object to multiple receivers will use borrowing.

Thus calling an ara::com method will consume the arguments, pass their ownership to the called method and return ownership of the result object to the caller. Publishing an event to the network stack will consume the argument, but subscribers only receive a shared reference.

On the other hand, passing objects by value is likely an error for large objects, because it implicitly invokes the copy constructor. Thus, Rust requires an explicit `.clone()` until the object is `Copy` (the equivalent to trivially copyable in C++).

6.7.2 Structured concurrency

Structured concurrency, limiting the lifetime of theads and tasks, is uncommon with the most popular asynchronous executor, `tokio`. Thus `tokio` requires `'static` lifetime for futures. The `smol` executor permits shorter lifetimes down to matching its executor object.

It was a design decision to facilitate passing references to e.g. the ara environment, which represents the time span between `Initialize` and `Deinitialize`, proxy or skeleton objects by allowing a shorter than static lifetime for each matching `spawn` function.

`Send` bounds are necessary for multithreaded execution and can be activated via the `send` feature of the `ara` crate.

6.7.3 Detailed event handling

The following graph illustrates how calling a method on a services traverses several layers of Rust and C++ code, with autogenerated application specific parts marked with (gen).

The result is asynchronous, thus the control is given back to the executor until the result is announced via a callback from the C++ side and then passed via a channel to the suspended function.

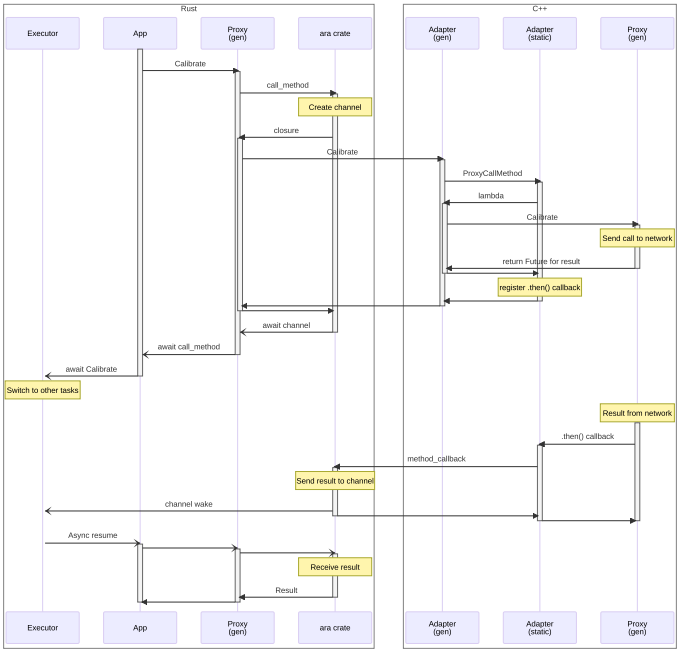


Figure 6.9: Calling a Method

Creating and offering a service are synchronous operations traversing many layers. Rust and C++ provide to each other function tables of operation callbacks.

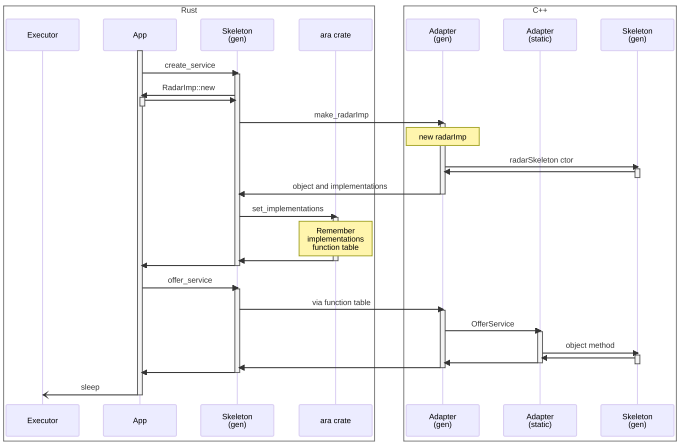


Figure 6.10: Creating a service

Invoking a method from the stack side calls into an asynchronous trait function implemented by the skeleton object. If the call directly returns a value, it is passed back to the caller and no asynchronous task is created.

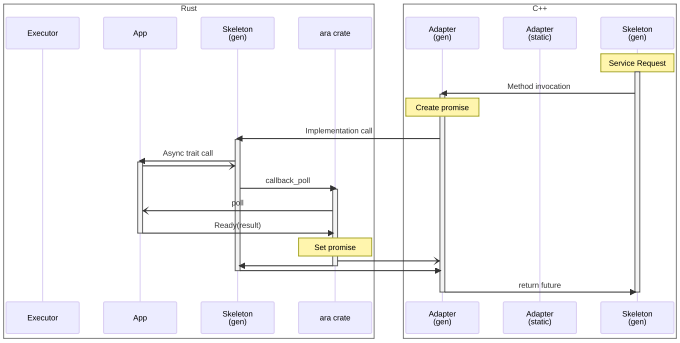


Figure 6.11: Fast method invocation

Please note that the `async` function feature of the Rust programming language will hide the `poll` function and its results `Ready(X)` and `Pending` from the application programmer. `Pin` and `Waker` are also implicitly added by the `async`/`await` syntactic sugar.

If the invoked method depends on an external event or method invocation, it internally returns Pending, this requires moving the promise to the heap and registering a new task with the executor.

If the method is not ready when unconditionally invoked next time by the executor, it will register a waking signal and return control to the executor.

Then the signal wakes up the task fulfilling the promise, which then sends the result via the ara::com stack on the C++ side.

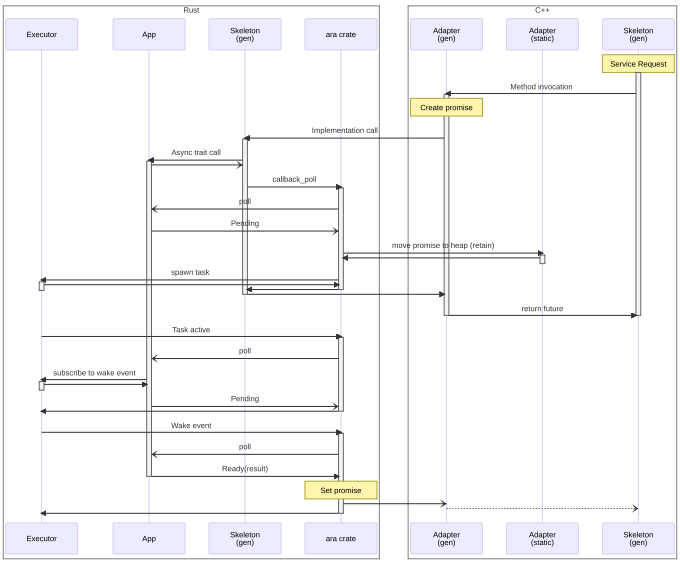


Figure 6.12: Slow method invocation

7 Recommend further readings

7.1 Recommended

Ide for Rust language:
IDE for i.e. VS Code, Vim - [14]

Performance measurement, finding most used parts of code, profiling:
Performance measurement package for Rust - `criterion` crate at [13]

Profiling method on Linux - profiling chapter of [15]

Rust coding style formatter:
Rust standard formatter, shipped with the compiler - [16]

Static analyzer:
Rust language Lint tool for static analysis of code [17]

Runtime test tools:
Unit test coverage and extension tool - `cargo-llvm-cov` crate at [18]

Runtime tester of possible bugs which language compiler could miss or ignore - [19]

Crate for testing potential concurrency errors - `loom` crate at [18]

Build checkers:
Checker for included in project cargo versions and known vulnerabilities - `cargo-checkmate` crate at [18]

Dependencies trustworthiness checker - `cargo-crev` crate at [18]

Dependencies trustworthiness checker - `cargo-vet` crate at [18]

7.2 Outlook

The infrastructure for asynchronous Rust, or more generally user space task management and user space I/O, is currently seeing a lot of innovation. `io_uring` changes currently form a large part of Linux kernel innovations and user space libraries are extended to support these APIs.

Since `async` became part of the Rust language in 2018 the most widespread `async` infrastructure, `tokio` has gone a long way to its current stable form. C++20 brought `co_await`, which is the exact C++ equivalent of Rust's `async`. But `async` function in traits are still missing from the Rust core language (requiring runtime memory allocation via `async-trait` as a workaround) and there is no equivalent for thread scopes.

As nano-coroutines offer significant advantage to model network protocols and I/O drivers for microcontrollers, Rust's `async` (e.g. embassy project) and `co_await` will see more adoption in resource constrained small embedded systems. Thus, introducing this future technology in the Rust binding for AUTOSAR Adaptive paves the road for its use in a future AUTOSAR Classic binding. Furthermore, existing `async` network protocol libraries (e.g. for SOME/IP) facilitate the creation of future fully Rust and `async` frameworks which support the necessary subset of an AUTOSAR stack for interoperability.

The increasing support for functional programming introduced by Rust compared to C will speed up the adoption of higher order abstractions into the embedded software industry, helping to cut down code maintenance costs due to better readability and increased automated checking. This is especially true for object lifetimes and ownership across library boundaries, as currently there is no widely adopted and machine verifiable lifetime notation for C and C++ and this leads to higher programming efforts and bugs.

Rust is currently fashionable for formal proofs of program correctness, facilitated by its increased guarantees for memory insulation or freedom from interference and the much stricter aliasing rules. As guidelines for Rust in functional safety get created and more and more automated verification tools are worked on, Rust is well positioned for safety critical systems.

7.2.1 WebAssembly Interface Types

Wrapping a function like FindService which returns a Result<Vector<InstanceId>> in a C interface is error-prone, because C doesn't support any generic or templated types, neither does it provide a standardized way to forward ownership of heap allocated data structures to called functions.

Over the past three years, the component model working group of the WebAssembly standard committee created a language neutral binary interface description which directly supports Result, Vector (called list<> there) and object handles (resources).

Since this directly adds support for a multitude of languages, including Rust, Python, Go, JavaScript, C++ and Kotlin, a future version of the AUTOSAR Rust binding will define and migrate to this language neutral layer.

This will also enable low-cost sub-application-sized modules, often called microservices, and creating stacks in other languages which then also support the C++ API by a shared implementation written in C++.

A Appendix

[1] Glossary
AUTOSAR_FO_TR_Glossary

[2] Programming rules to develop secure applications with Rust
https://www.ssi.gouv.fr/en/guide/programming-rules-to-develop-secure-
-applications-with-rust/

[3] High Assurance Rust, Developing Secure and Robust Software
https://highassurance.rs/

[4] SAE JA1020 – Recommendations for the Rust Programming Language in Safety-Related Systems

[5] Ferrocene Language Specification
https://spec.ferrocene.dev/

[6] Learn Rust
https://doc.rust-lang.org/learn/

[7] Comprehensive Rust
https://google.github.io/comprehensive-rust/

[8] Rust for the Polyglot Programmer
https://www.chiark.greenend.org.uk/~ianmdlvl/rust-polyglot/index.html

[9] Rust for C++ Programmers
https://github.com/nrc/r4cppp

[10] Rust Cheat Sheet
https://cheats.rs/

[11] Asynchronous programmming in Rust
https://rust-lang.github.io/async-book/

[12] Async Rust vs RTOS showdown!
https://tweedegolf.nl/en/blog/65/async-rust-vs-rtos-showdown

[13] Crate documentations
https://docs.rs/

[14] Rust analyzer
https://rust-analyzer.github.io/

[15] Rust SIMD Performance Guide
https://rust-lang.github.io/packed_simd/perf-guide/

[16] Format Rust code
https://github.com/rust-lang/rustfmt

[17] Clippy rules
https://rust-lang.github.io/rust-clippy/master/

[18] The Rust community's crate registry
https://crates.io/

[19] MIRI - An interpreter for Rust's mid-level intermediate representation
https://github.com/rust-lang/miri