Rust On L4Re

Integrating A Modern Systems Language Into A Microkernel Userland

Name	Sebastian Humenda
Mentor	Marcus Hähnel
Published at	Technische Universität Dresden, Germany
Date	14/06/2017

Contents

1 Introduction

Rust is a modern systems programming language, with a focus on safety, concurrency and speed. It propagates high-level abstractions at no or almost no cost, depending on the use case [11]. Rust's safety guarantees, enforced by the type system, complements strengths of L4, such as isolation and virtualisation of safety-critical systems.

The Rust compiler, also called Rustc, uses a novel ownership and borrowing model to infer life-time information for an object during compile time to automatically insert cleanup[1] code and to prevent "use after free" and other common bugs [11]. This eliminates the need for a garbage collector at runtime, memory management calls are generated statically. This enables the Rust compiler to give strong safety guarantees for all parts of the code which are not explicitly marked as unsafe. This allows for construction of safe network protocol stacks and potentially also for safe IPC communication in microkernel systems.

To ease understanding, I will introduce a few Rust-specific terms. General knowledge of systems programming and microkernels is assumed. Basic Rust knowledge is helpful.

In the Rust ecosystem, a crate is either a library or a binary,[2] which forms a unit. It is not meant as a replacement term and calling a library a library is fine, crate just makes clear that it is Rust code. Under the hood, this has a few more reasons. Crates bundle meta information and also serve as a unit for Rust's build system and package manager called Cargo. Project management, dependency resolution and code compilation is all handled by Cargo. It provides a central project configuration and offers commands to ease development from within IDEs. Each crate can be structured into modules. A module is a separate namespace, and its name is derived from the file or directory name. They don't have any other function, despite structuring the global namespace.

1.1 L4 Build System

The L4Re build system (**BID**) is based on GNU Make and provides both dependency management and cross-compilation for foreign architectures. It also makes it easy to create L4Re systems with only specific components enabled [3, 4].

To achieve this degree of flexibility, a lot of the actual rules are auto-generated, depend-ending on the environment and configuration settings. BID makes a lot of assumptions on the toolchain. For instance, it assumes that a compiler compiles to object files and that another program proceeds with the object file. This is different for Rust, which is

[1]These are mostly calls to `free()`, but can be anything.
[2]Crate is more often used for libraries.

explained in subsequent chapters. For cross-compilation, generated rules assume that a cross-compiler will have the architecture in the name. Rust in contrast is a cross-compiler itself and takes the information about a foreign architecture and environment from a target specification, given as command-line parameter.

The L4Re BID handles dependencies, as does Cargo. L4Re software is packaged and each package can declare dependencies on other packages. Ideally, a Rust programmer packages source for L4Re as for every other programming language. Details about the compiler and cross-compilation is handled transparently by BID.

1.2 Rust Development Tools

The Rust language comes with a variety of tools to aid development. Usually, these tools are installed using Rustup. Rustup manages different versions and branches of the Rust toolchain. Currently, there are Rust nightly (unstable), Rust beta and Rust stable. Rustup can enable one or more of these toolchains and can also install updates. It also manages foreign architectures or operating systems, called targets. When adding a target, the standard libraries and other tools for the request target are downloaded from the internet.

Cargo manages Rust projects: it builds executable and libraries, it downloads and checks dependencies and it can carry out tests. A global project configuration called `Cargo.toml` configures these aspects.
Conditional compilation in Rust is called a *feature*. It allows the exclusion or inclusion of parts of a project, depending on many constraints, for instance operating system or target platform. These features should not be confused with the feature keyword of Rust, which enables unstable features of Rust in a program. These features are also called language features.

Because Cargo is more than a dependency management utility, it makes sense to use of its capabilities. Since BID and Cargo both handle dependency resolution, the integration is challenging. For this reason, I will present in this work the integration of Rustc into BID, with the option of using Cargo later on. My focus has been to enable simple Rust programs to run on L4Re. I will also discuss shortcomings in the current Rust infrastructure and BID.

2 Integration Challenges

Rust is a language built on top of the LLVM framework, to allow for high-performance optimized code for all platforms that LLVM targets. However, Rust handles the build process different from other compilers. So for instance, the Rust compiler compiles an object file and passes this object file with custom options directly to the linker, instead of leaving the task to an external build utility.[3] This has several reasons, for instance, generic functions can be monomorphized acrosss libraries and Link-Time Optimisations (**LTO**) can happen independent from the linker. Furthermore, the Rust compiler handles incremental compilation itself and by calling the linker directly, the number of object files and their order is independent from the surrounding build system [8].

Rust doesn't use the C-ABI (except if requested) and doesn't have a stable ABI yet. Standard file formats as `.a` for static libraries and `.so` for dynamic libraries can only be used, if a C-ABI is explicitly exposed.

2.1 Unstable ABI

Rust does not have a stable ABI, which is still subject to discussion. Defining a new ABI needs great care, because if the ABI were released too quickly, changes to core aspects of the language would be impossible.
The C-ABI offers too little type information for Rust. For instance, generic types, trait definitions[4] and other type information cannot be exposed. C++ for instance works around this problem by banning templates into header files[5] and hence does a from-source compilation for each newly built binary [6].

The standard library of Rust makes use of features, which are only available in the unstable branch of Rust. These features have not been stabilized yet, because their design has not been fully discussed and they also partly influence the ABI stabilization. This means that a Rust nightly compiler is required for building the stdlib [1]. This is discussed in more detail in Section 3.2 on page 7.

[3]For other languages, for instance for C, the compiler can call the linker automatically as well. However, this can be easily decoupled.
[4]A trait in Rust is similar to traits in other languages and defines methods and functions or types for a type.
[5]This is true for libraries.

2.2 Dependency Management

As mentioned earlier, I have not integrated Cargo into BID. Instead, the dependency management is done by BID. This means that dependencies have to be specified in the package `Makefile` of a project.[6]

Every Rust program, unless explicitly disabled, dependends on the `std` crate (standard library). `std` in turn depends on roughly a dozen libraries and acts as a facade to these low-level implementations. This has the advantage that Rust programs can be customized to different systems, by either using the standard library or by using only a few of its dependencies. The `std` crate has not been ported yet. Therefore, each L4Re package written in Rust needs to declare the libraries of the standard library it depends on manually. For an example, see Section 4.2 on page 11.

2.3 Conditional Compilation

Cargo offers conditional compilation with a mechanism called **feature**. A feature should not be confused with a *language feature*.[7] Features are normally activated and deactivated in the `Cargo.toml` configuration file. Cargo takes care of passing the correct flags to manage feature activation to Rustc. Due to my decision to integrate Rustc first, there is no easy way for the application developer to pass features to Rustc at the moment. As a work-around, a developer can extend the `RSFLAGS` variable in the Makefile of the L4Re Package. For an example Makefile, see Section 4.1 on page 9.

3 Port And Implementation

In the following sections, I will introduce the changes which I made to Rust and BID. The current BID extensions are very experimental. They are subject to change, while the integration of the L4Re target into Rust progresses. The aim is to enable Rust to compile code for L4Re without any patches.

[6]An example Makefile will be shown in the next section.

[7]A language feature is a way of enabling additional language extensions in nightly versions of the compiler. These are experimental and might either be stabilized in the future or removed again.

3.1 Cross-Compilation

Rustc by itself is already a cross-compiler. It uses so called target specifications to learn about the properties of the target system. These are shipped with Rustc for the major platforms, but can be easily specified as a JSON file. JSON target specifications are very helpful for prototyping, because they can be passed to Rustc via `--target=`. Since target specification are very sensible to compiler-internal changes, they are not stable. For the longer term, target specifications should always go into the Rust compiler directly.

A JSON target specification is basically a mapping of different key-value pairs. Some of the possible values include the pointer width, properties of atomic counters, additional linker arguments, etc. With the help of this, compiling a crate for Windows on a GNU/Linux system is merely

```
$ cargo build -- --target x86_64-pc-windows-msvc
```

For this work, a JSON target specification has been written, compatible with Rust 1.18. The specification is copied to the object directory before the build and passed to each invocation of Rustc. An excerpt of the current target specification looks like this:

```
{
  "arch": "x86_64",
  "cpu": "x86-64",
  "os": "L4Re",
  "env": "uclibc",
  "vendor": "unknown",
  "target-family": "unix",
  "linker-flavor": "ld",
  "target-endian": "little",
  "target-pointer-width": "64",
}
```

Listing 1: Excerpt Of The Current Rust Target Specification In JSON

3.2 Compilation Of Libraries

Rustc compiles libraries by reading a file called `lib.rs`[8] and follows all references to other files. It hence makes all additional BID/Make logic to compile other files of a library superfluous. The output is not a static `ar` (archive) file, but a **rlib**, which is an internal Rust library format for static libraries. Rust introduced a special file format, because the objects (**.o**) within an archive file cannot store the additional type information required

[8]This is a convention and can be configured.

for Rust compilation. Because of the ABI instability, this rlib file is specific to the Rust compiler version.[9] The `rlib` format is implementation-specific and can change between Rust versions; at the moment, it is an `ar` archive with additional type information, for instance about generic types or trait objects. See Section 2.1 on page 5.

As of Rust 1.18, the Rust standard library cannot be built with stable Rust, because not all relevant language features have been stabilized yet. To resolve this contradiction, each stable Rust compiler also contains a copy of Rust nightly, which is used for building the standard library. To enable this, an undocumented environment variable has to be set. For the BID Makefiles, the following line enables the additional features:

```
export RUSTC_BOOTSTRAP=1
```

Rlib's are used for static linking and are prefered, because only with their amount of information, the compiler can fully optimize the resulting binary. Nevertheless, Rust also supports dynamic linking (`.so`) and classic static libraries (`.a`). This has the disadvantage that a C-ABI must be exposed, which is unsafe and looses all relevant type information for optimization.

At the time of writing, I have been able to build the following libraries for L4Re: `alloc`, `alloc_system`, `backtrace`, `collections`, `compiler_builtins`, `core`, `libc`, `panic_abort`, `panic_unwind`, `rand`, `std_unicode`, `unwind`.

Due to the modular architecture, changes are only required in `std` and `libc` and the patch for the latter is already merged into the original project.[10]

3.3 Compilation Of Binaries

After object files have been compiled, they are linked to a final binary. BID has a special set of rules to guide this process, which differs slightly from the one of other operating systems. L4Re uses the ELF binary format and ships a custom linker script to instruct the linker how to arrange the ELF sections. This is necessary, because the L4Re linker script is not part of a normal GCC distribution. Additionally, custom startup code / teardown code[11], as well as additional predefined static symbols, are used.

As I have explained before, Rust handles these steps transparently and calls *cc* or *ld* directly for linking. This conflicts with BID, which is designed to manage this very

[9]The format of the RLIB file format is described here: `https://github.com/rust-lang/rust/blob/master/src/librustc_metadata/schema.rs",`

[10]The GitHub pull request can be found at `https://github.com/rust-lang/libc/pull/607`.

[11]These files are called `crt0.o` and `crtn.o` and are compiled into each executable.

process. The Rust compiler offers the possibility to modify the linker argument list, by modifying options in the target specification, which I explained on page 7. Most important are the `pre-link-args` and `post-link-args` and the same for objects. This allows for relatively fine-graned control over the linker command line. The positioning of objects and linker arguments is not trivial and due to time constraints, I have not integrated L4Re-linking support into Rustc. Instead, binaries are compiled as a static library and have to export a main function with the C ABI. This way, the usual L4Re linking step can be utilized and no special treatment is required. The additional function in the Rust source code looks like this:

```
#[no_mangle]
pub extern "C" fn main(_: i32, _: *const *const u8) -> i32 {
    // ...
    0
}
```

Listing 2: C-ABI compatible main function in Rust

Please see the next section for an explanation of the code.

4 Working Example

In the previous sections, I have discussed some of the challenges of porting Rust to the L4Re platform and detailled some of the current shortcomings. With a set of patches, compiling Rust code on L4Re is possible. Some of these changes have already been integrated into the upstream Rust project. The following section will show in practical examples, how a basic program might look like. It is assumed that the reader is familiar with the structure of a L4Re package.

4.1 Makefile

I have changed BID in a few places. This mainly affects binary and library rules. BID generates rules automatically, depending for instance on the programming language used. If it is told to build a file ending on `.rs`, it will try to find compilation rules for this suffix.

In the following Makefile, a package with no subdirectories and with a single source file is assumed:

```
1  PKGDIR   ?= .
   L4DIR                ?= $(PKGDIR)/../..

   TARGET               = agoodname
5  SRC_RS               = main.rs

   REQUIRE_LIBS += libpanic-rust libcollections-rust libcore-rust liballoc-rust
   export RUSTC_BOOTSTRAP=1

10 include $(L4DIR)/mk/prog.mk
```

Listing 3: Makefile For BID With Example Dependencies

With the variable `SRC_RS`, I have added a way for the programmer to specify the location of the source files and at the same time to select the programming language for compilation. Since Rustc traverses files automatically from the given entry file, no further files need to be specified.

In line 4, the target is defined. Since this target has no file extension, BID invokes the rules for creating a binary. Normally, this would mean compiling object files and linking them. Since this is hard to achieve with Rust, I wrote a rule to compile Rust source code to a static library and another rule to link the static library to an executable.
If the target in the Makefile ends on `.rlib`, the rule to compile a static Rust library is invoked.

Line 7 lists all the crates that the example project depends on. These are a selection of the standard library crates. Listing these packages is a current work-around, because the `std` crate cannot be built for L4Re yet. As soon as this works, listing these libraries will be not necessary anymore. This line could be extended by more Rust crates (or even C libraries), as long as they are packaged withing the L4Re source tree.

In line 8, the Rust compiler is instructed to expose nightly features from a stable version of the compiler. As mentioned in Section 3.2 on page 8, we need to do this to use some of the features present in the core crates. This line should not be used anymore as soon as the `std` crate is available.

If desired, additional arguments can be passed to the Rust compiler, for instance to increase the optimization level or to activate a feature by extending the `RS_FLAGS` variable. To build a well-optimized binary, one could add the following line to the Makefile:

```
RS_FLAGS += -C opt-level=3
```

4.2 Rust Code

The `std` crate facade is not available for L4Re yet and therefore the programmer has to do extra work to use the core libraries. In the following code example, a vector from the collections crate is initialized on the heap and its contents are printed:

```rust
#![no_std]
#![feature(collections)]
#![feature(lang_items)]
#![feature(panic_unwind)]

extern crate panic_unwind;
extern crate libc;
#[macro_use]
extern crate collections;

use libc::*;

/// Empty panic formatter...
#[lang="panic_fmt"]
#[no_mangle]
pub fn panic_fmt() -> ! {
    loop { } // no handling
}

#[no_mangle]
pub extern "C"
fn main(_: i32, _: *const *const u8) -> i32 {
    let nums = vec![1, 2, 3];
    unsafe {
        printf("I can count: \0".as_ptr());
    }
    for elem in nums {
        unsafe {
            printf("%i, \0".as_ptr(), elem);
        }
    }
    unsafe {
        printf("\n\0".as_ptr());
    }
    0
}
```

Listing 4: RustDemonstration Of Core Library Usage

The #![no_std] directive disables the default usage of the standard library. Most of the traits and default macros available in every Rust program will not be available, because they are automatically imported from the prelude,[12]

[12] A small set of macros, types and traits are automatically imported in every Rust program. They are defined in std::prelude.

The feature directives, from line 2 onwards, are part of Rust nightly and enable (*language*) features which are not part of official Rust. This is normally part of the `std` crate.
The extern crate keyword instructs the compiler to search for the requested library and use it. A more detailled explanation can be found in [10].

Panics are Rusts way of reacting to unrecoverable failures. There are many panic strategies available, depending on the use case. The default way is the unwind library, which can, if requested, display a backtrace of the panic. For embedded systems, a direct abort strategy might be more appropriate, to save space and time. On line 14, an empty panic formatter, which leads to an endless loop on a panic, is used for demonstration purposes.

The main function is a C-ABI compatible function (as can be seen from its signature). The `std` crate defines an entry point for the program, which internally generates a C-ABI compatible start function, which calls the Rust main function [9]. Since the `std` crate is not present, this has to be done for all L4Re Rust programs manually.

Explaining the contents of the main function in detail is beyond the scope of this work, but it should be noted that the vector type comes from the `collections` crate and the printf function from the `libc` crate. For comparison, the same program with `std` looks like this:

```
fn main() {
    let nums = vec![1, 2, 3];
    print!("I can count: ");
    for elem in nums {
        print!("{}, ", elem);
    }
    println!();
    0
}
```

Listing 5: Simple Demonstration Program With Standard Library Enabled

5 Future Work

In the previous sections, I have given an overview about the current state of Rust on L4Re. Rust programs can be built and can interface with the native libraries, using the C-API, to interact with the operating system. However, the examples also made clear that the expected convenience is still missing. Therefore, porting the `std` crate to L4Re is a priority. This would not only add additional ease of use for application developers, it would also allow the re-use of existing Rust software.

As a next step, the L4Re target specification should become part of the compiler. This way, BID would only need to pass the location of the linker script, the start-up files and the library paths to `rustc`. This would heavily simplify the integration into BID:

1. The intermediate compilation step to a static library could be omitted, because Rustc would be able to invoke the linker with the correct arguments for L4Re. As a consequence, the main function wouldn't need to be exported as a C-ABI compatible function.

2. Instead of embedding the compiler, the build system of the Rust ecosystem, Cargo, could be integrated into the BID. This would bring the additional benefit, that Cargo would take care of feature activation and of version clashes between crates. It would also allow for seemless integration into IDEs for Rust L4Re developers.

The `libc` crate still lacks most of the definitions present in uClibc. Extending the Rust bindings to capture the most relevant would make it easier for application programmers to interface with the POSIX compatibility layer.

As soon as the target specification is officially part of the rust compiler and L4Re is an official platform, adding Rust support to Rustc could be as easy as

```
rustup component add l4re
```

This would make packaging Rust's standard library within the L4Re source tree obsolete. This is currently necessary, because patched versions of the low-level crates need to be built. This adds complexity and maintenance overhead. Getting all patches upstream is hence very important.

References

[1] May 10, 2017.
Web: https://github.com/nox/rust-rfcs/blob/master/text/1184-stabilize-no_std.md.

[2] May 12, 2017.
Web: https://github.com/rust-lang/rfcs/issues/1675.

[3] May 24, 2017.
Web: https://l4re.org/doc/l4re_build_system.html.

[4] Apr. 25, 2017.
Web: https://os.inf.tu-dresden.de/l4env/doc/html/bid-spec/node13.html.

[5] May 10, 2017.
Web: https://web.archive.org/web/20140815054745/http://blog.mozilla.org/graydon/2010/10/02/rust-progress/.

[6] May 22, 2017.
Web: https://github.com/rust-lang/rfcs/issues/600.

[7] May 26, 2017.
Web: https://doc.rust-lang.org/std/prelude/.

[8] July 14, 2017.
Web: https://github.com/rust-lang/rust/issues/38913.

[9] July 14, 2017.
Web: https://github.com/rust-lang/rust/blob/master/src/libstd/rt.rs#L32.

[10] Steve Klabnik and Carol Nichols: *The Rust Language*. No Starch, Aug. 2017.

[11] Nicholas D. Matsakis and Felix S. Klock II: “The Rust Language”. In: *Ada Lett.* 34.3 (Oct. 2014), pp. 103–104. ISSN: 1094-3641. DOI: 10.1145/2692956.2663188.
Web: http://doi.acm.org/10.1145/2692956.2663188.