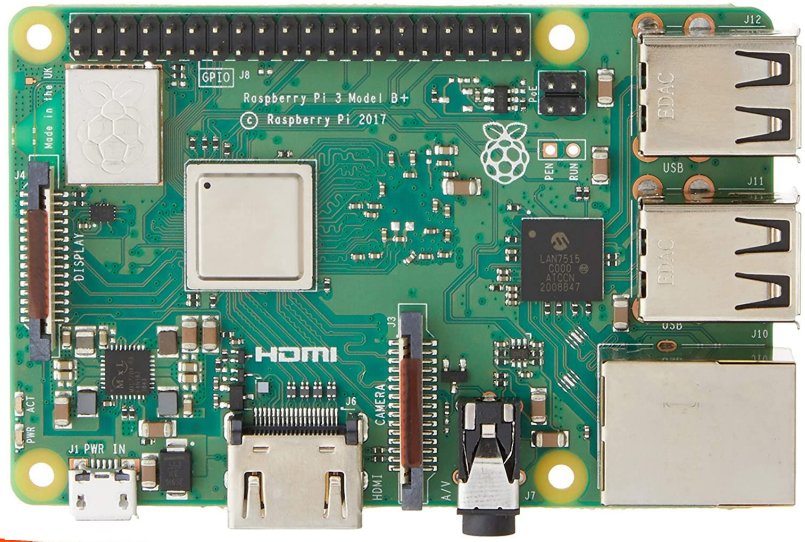
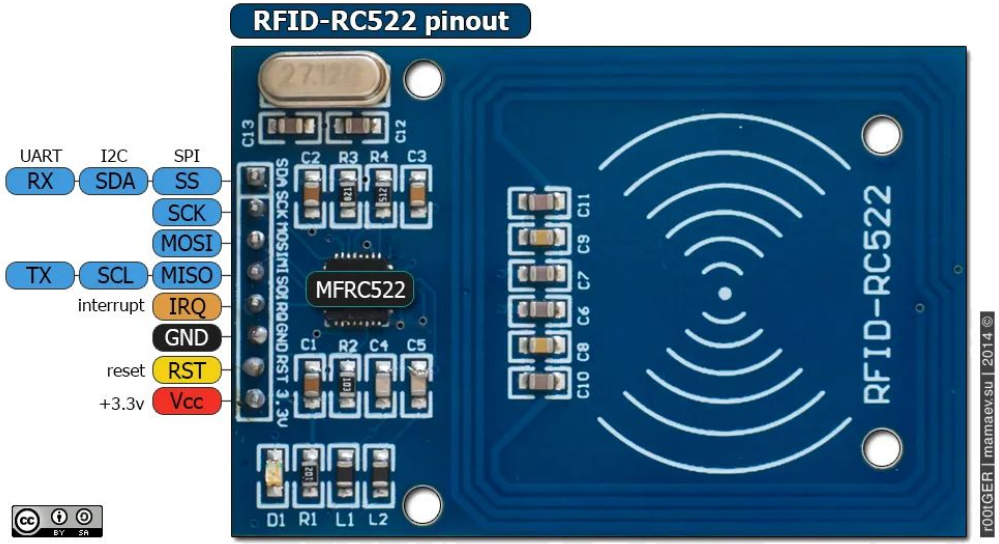
Developing a Linux driver in Rust for ARM

Who are we ?


GPIO
Raspberry Pi 3 Model B+
© Raspberry Pi 2017
HDMI
DISPLAY
CAMERA
J1 PWR IN


RFID-RC522 pinout
UART
I2C
SPI
RX
SDA
SS
SCK
MOSI
TX
SCL
MISO
interrupt
IRQ
GND
reset
RST
+3.3v
Vcc
MFRC522
RFID-RC522
r00tGER | mamaev.su | 2014 ©

Why Rust?

- Safety
- Speed
- Concurrency
- Expressivity
- No garbage collection and no manual memory management! (lifetime analysis)
- Zero-cost abstractions
- Zero setup cross compilation
- Strong ecosystem and tons of libraries at your fingertips!
- **Not for rewriting the kernel!**

First method: Calling Rust from C

```c
// mfrc522.c
void print(char *msg) {
      printk(KERN_INFO "%s", msg);
}

static int __init mfrc522_init(void) {
      hello_rust();
      return 0;
}

static void __exit mfrc522_exit(void) {
      pr_info("MFRC522-Rust exit\n");
}

module_init(mfrc522_init);
module_exit(mfrc522_exit);
```

```rust
// mfrc522.rs
#![no_std]

extern "C" {
    fn print(msg: *const u8) -> ();
}

pub extern "C" fn hello_rust() -> i32 {
    unsafe {
      print("Hello from Rust\n\0".as_bytes().as_ptr())
    };

    return 0;
}
```

C glue for Rust code

Pros

- It works (kinda…)
- 2 ways to go about it:
  - C -> Rust can’t “talk” with the kernel (compute only)
  - C -> Rust -> C can “talk” with the kernel

Cons

- Needs a C to Rust wrapper for each kernel function

```
// .cargo/config.toml
[build]
rustflags = [
     "-C", "relocation-model=static", "-C", "code-model=kernel", "-Z", "plt=y"
]

// Works… But "code-model=kernel" is only available for x86, so we removed it...
```

Issues with the method

- Weird requirements from the kernel's *modpost* tool
  - Issues with unwinding functions defined in *libgcc*

```
// Cargo.toml
[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"
```

```
ERROR: modpost: "__aeabi_unwind_cpp_pr1" [mfrc522.ko] undefined!
ERROR: modpost: "__aeabi_unwind_cpp_pr0" [mfrc522.ko] undefined!
```

```
// `cpp`? In my Rust code?
// Stack unwinding? But we abort on panics!
```

Issues with the method

- Common issue when cross-compiling for ARM?
- Since we don’t need unwinding code, we can write those functions ourselves, right?
  - More glue!

```rust
// If we arrive here, we're screwed anyway
#[no_mangle]
unsafe extern "C" fn __aeabi_unwind_cpp_pr1() -> ! { panic!() }
#[no_mangle]
unsafe extern "C" fn __aeabi_unwind_cpp_pr0() -> ! { panic!() }
```

Issues with the method

- *modpost* does not complain…
- ...but *insmod* does, missing relocation
- Only present when using Rust features: *pattern matching*, *traits*, *...*
  - Which defeats the purpose of using Rust...

Second method: Using rust-for-linux

What is Rust-for-Linux?

What is Rust for Linux?

- https://github.com/Rust-for-Linux/linux
- rust-for-linux@vger.kernel.org
- Goal: add support for the Rust language to the Linux kernel
- The code will be submitted to review to LKML
- First patch to linux-next: https://lkml.org/lkml/2021/4/14/1023

Compile rust-for-linux/linux

```
$ rustup default nightly-2021-02-20
...
$ rustup component add rust-src
...
$ cargo install --locked --version 0.56.0 bindgen
...
$ # Enable Rust support: General setup -> Rust support
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- LLVM=1 LLVM_IAS=1
...
```

Out of tree module in Rust for Linux

- Compiles correctly for arm64 defconfig...
- ...but not for our Raspberry Pi 3b+ config

```
error[E0463]: can't find crate for `core`
  |
  = note: the `target-10773475644222393364` target may not be installed

error: aborting due to previous error

For more information about this error, try `rustc --explain E0463`.
```

```
// target-10773475644222393364??
// Seems to be a cross-compilation issue
```

Rust for Linux, native compilation

- We could use a VM, but that’s heavy and not practical
- What about using a RasPi?
- Compilation time will increase
- What about chrooting? We do not need another running kernel

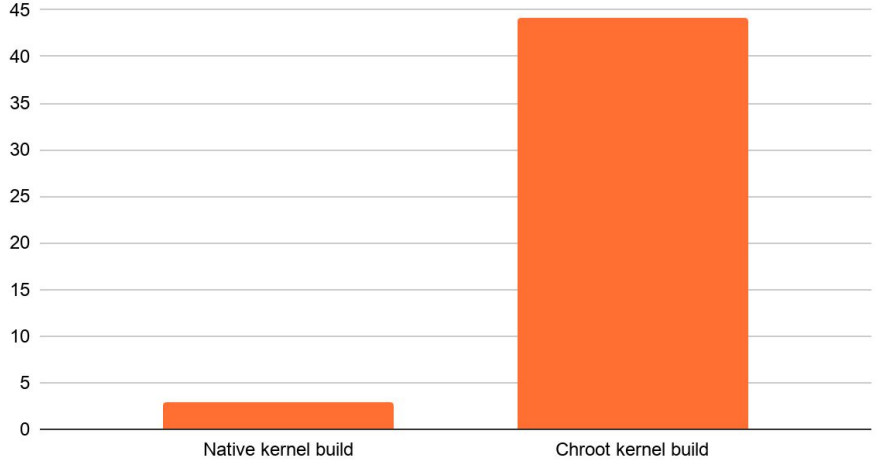
Chrooting in an aarch64 rootfs

- Impossible ?
- Use *qemu-user-static*
- Download a rootfs
- *chroot* into it
- Et voilà!

Compiling in the chroot

- Slooooooooow
- “Incremental” build time on chroot ~= full native build time
- Workflow less than pleasant.
- But out of tree module compilation works so…
- But still faster than on a Raspberry Pi
- After some time we switched to intree compilation

Native VS chroot compilation time (minutes)



Writing the driver

What do we need ?

- MODULE_{LICENSE, AUTHOR, DESCRIPTION} ✅
- module_{init, exit} ✅
- printk ✅
- misc devices API ✅
- spi API ❌

Rust API generation with bindgen

```
// rust/kernel/bindings_helper.h

/* SPDX-License-Identifier: GPL-2.0 */
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/module.h>
#include <linux/random.h>
#include <linux/slab.h>
...
#include <uapi/linux/android/binder.h>
```

```
// rust/bindings_generated.rs

pub const PLATFORM_DEVID_NONE: i32 = -1;
...
pub struct sysinfo {
    pub uptime: __kernel_long_t,
    ...
}
...
extern "C" {
    pub fn misc_register(misc: *mut
miscdevice) -> c_types::c_int;
}
```

Rust SPI API

- Add `#include <kernel/spi.h>` to rust/kernel/bindings_helper.h
- We now have generated Rust bindings for *spi_register*, *spi_write_then_read* and so on
- But our work is not done!
  - Simply using C functions from Rust negates the advantages of Rust
  - Rust-for-Linux implements high-level abstractions for interacting with the kernel
  - So we need to implement our abstraction too!

```c
static struct spi_driver mfrc522_spi_driver = {
	.driver = {
		.name = "mfrc522", .owner = THIS_MODULE,
	},
	.probe = mfrc522_spi_probe,
};

static int mfrc522_spi_probe(struct spi_device *client) {}

static int __init mfrc522_init(void) {
	int ret = spi_register_driver(&mfrc522_spi_driver);
	if (ret) {
		pr_err("[MFRC522] SPI Register failed\r\n");
		return ret;
	}
	return 0;
}
```

```rust
struct Mfrc522SpiMethods;

impl SpiMethods for Mfrc522SpiMethods {
    declare_spi_methods!(probe);

    fn probe(mut spi_device: SpiDevice) -> Result {}
}

impl KernelModule for Mfrc522Driver {
    fn init() -> Result<Self> {
        let spi = spi::DriverRegistration::new_pinned::<Mfrc522SpiMethods>(
            &THIS_MODULE,
            cstr!("mfrc522"),
        )?;
    }
}
```

How do we achieve this?

- Allow the user to register an SPI Driver (and unregister it safely)
- Use high level abstractions for the definitions of SPI Methods
- Add wrappers around usual SPI functions
  - *spi_write_then_read*
  - *spi_write*
  - *spi_read*
  - ...
- The user should only interact with safe, idiomatic Rust code

```rust
// Generated Rust representation of the spi_driver struct

#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct spi_driver {
    ...
    pub probe: Option<unsafe extern "C" fn(spi: *mut spi_device) -> c_types::c_int>,
    pub remove: Option<unsafe extern "C" fn(spi: *mut spi_device) -> c_types::c_int>,
    pub shutdown: Option<unsafe extern "C" fn(spi: *mut spi_device)>,
    ...
}
```

```rust
// The `probe` method is an option: Equivalent to NULL or a function
// It should take a pointer as argument, and return an integer
pub struct spi_driver {
    pub probe: Option<unsafe extern "C" fn(spi: *mut spi_device) -> c_types::c_int>,
}

// Equivalent to C's
struct spi_driver {
    int (*probe)(struct spi_device *spi);
}
```

Why external functions are unsafe

- Since the compiler cannot guarantee the safety of external C functions, calling them is inherently unsafe
- We need to make sure to document unsafe code as well as check it carefully, to avoid undefined behavior
    - Which we encountered… But more on that later

```rust
// We could have the user write the following
unsafe extern "C" fn my_probe(spi: *mut spi_device) -> c_types::c_int {
    pr_info!("Probing successful!\n");

    0
}
```

High level SPI methods

- But we can't let the user write unsafe functions!
- What's the point of using Rust if it's just to interact with C?
- It makes the user interact with raw pointers and C types
  - This is quite annoying in Rust!
  - Rust is written using references instead of pointers
  - And using *i32* instead of *c_types::c_int*
  - c_types are FFI safe and correspond to the actual C types and API, but they're still annoying to use

```rust
// Ideally, this is what `probe` should look like

fn my_probe(mut spi: SpiDevice) -> Result {
    pr_info!("Probing successful!\n");

    Ok(())
}
// The function is safe
// We use high level types such as `SpiDevice`
// It returns using a high level abstraction, `Results`, instead of integers
```

```rust
// Here's what ours look like (cropped a little)
fn probe(mut spi_device: SpiDevice) -> Result {
    pr_info!("[MFRC522-RS] SPI Registered\n");

    let version = match Mfrc522Spi::get_version(&mut spi_device) {
        Ok(v) => v,
        Err(_) => return Err(kernel::Error::from_kernel_errno(-1)),
    };

    pr_info!("[MFRC522-RS] MFRC522 {:?} detected\n", version);

    Ok(())
}
```

Converting safe, high-level functions into low level ones

- Since we interact with C, we cannot pass our high-level function as a member to our spi_driver. We *need* to give it an unsafe function using C types and pointers.
- We can convert a Rust function into a C function using Rust macros! (Don’t do this)

```
macro_rules! spi_method {
    (fn $method_name:ident (mut $device_name:ident : SpiDevice) -> Result $block:block) => {
        unsafe extern "C" fn $method_name(dev: *mut kernel::bindings::spi_device) ->
kernel::c_types::c_int {
            use kernel::spi::SpiDevice;

            fn inner(mut $device_name: SpiDevice) -> Result $block

            match inner(SpiDevice::from_ptr(dev)) {
                Ok(_) => 0,
                Err(e) => e.to_kernel_errno(),
            }
        }
    };
}
```

```
// This is the invocation, what you use to call the macro
(fn $method_name:ident (mut $device_name:ident : SpiDevice) -> Result $block:block)
//  ^^^^^^^^^^^^^^^^^^^       ^^^^^^^^^^^^^^^^^^^                        ^^^^^^^^^^^^
// These are our macro parameters
// The rest are just matching keywords for invoking the macro!
```

```
// We could also use this match arm, for example
(penguins $method_name:ident love $device_name:ident crabs $block:block)
```

```
        // This is the generated code. $Arguments are expanded from the invocation
        // This expansion is done at the AST level, not as tokens in the source
        unsafe extern "C" fn $method_name(dev: *mut kernel::bindings::spi_device) ->
kernel::c_types::c_int {
            use kernel::spi::SpiDevice;

            fn inner(mut $device_name: SpiDevice) -> Result $block

            match inner(SpiDevice::from_ptr(dev)) {
                Ok(_) => 0,
                Err(e) => e.to_kernel_errno(),
            }
        }
```

```
macro_rules! spi_method {
    (fn $method_name:ident (mut $device_name:ident : SpiDevice) -> Result $block:block) => {
        unsafe extern "C" fn $method_name(dev: *mut kernel::bindings::spi_device) ->
kernel::c_types::c_int {
            use kernel::spi::SpiDevice;

            fn inner(mut $device_name: SpiDevice) -> Result $block

            match inner(SpiDevice::from_ptr(dev)) {
                Ok(_) => 0,
                Err(e) => e.to_kernel_errno(),
            }
        }
    };
}
```

```rust
// Let's use our macro for our previously defined probe

    fn my_probe(mut spi: SpiDevice) -> Result {
        pr_info!("Probing successful!\n");

        Ok(())
    }
```

```
// Let's use our macro for our previously defined probe
spi_method! {
    fn my_probe(mut spi: SpiDevice) -> Result {
        pr_info!("Probing successful!\n");

        Ok(())
    }
}

// Done!
```

```rust
// Or the alternative...
spi_method! {
    penguins my_probe love spi crabs {
        pr_info!("Probing successful!\n");

        Ok(())
    }
}
```

```rust
// Let's look at the generated code
unsafe extern "C" fn my_probe(dev: *mut kernel::bindings::spi_device) -> kernel::c_types::c_int {
    use kernel::spi::SpiDevice;

    fn inner(mut spi: SpiDevice) -> Result {
        pr_info!("Probing successful!\n");

        Ok(())
    }

    match inner(SpiDevice::from_ptr(dev)) {
        Ok(_) => 0,
        Err(e) => e.to_kernel_errno(),
    }
}
```

Probe function, macro edition

Pros

- Easy to implement
- Easy to use

Cons

- Unmaintainable
- *Awful* compiler errors on misuse

What's the alternative

- We can use Rust's object paradigm to create a *trait*
  - This defines an interface which a type can implement

```rust
// This is similar to Java interfaces for example. This one has default implementations
pub trait SpiMethods {
    fn probe(mut _spi_dev: SpiDevice) -> Result {
        Ok(())
    }

    fn remove(mut _spi_dev: SpiDevice) -> Result {
        Ok(())
    }

    fn shutdown(mut _spi_dev: SpiDevice) {}
}
```

What's the alternative

- Using genericity, we can implement functions with the correct valid signature
- Which take any high level Rust type

```rust
// This function has the right signature for a probe function, but is generic!
// Will generate a function specific to this type
// We can have trait bounds (T: SpiMethods) to make sure that the types we use it on implement the
// required methods
unsafe extern "C" fn probe_wrapper<T: SpiMethods>(s: *mut bindings::spi_device) -> c_types::c_int {
    // We call the probe function from the type given as generic parameter
    match T::probe(SpiDevice::from_ptr(s)) {
        Ok(_) => 0,
        Err(e) => e.to_kernel_errno(),
    }
}
```

```
// The register method is generic too, with a trait bound too
pub fn register<T: SpiMethods>(self: Pin<&mut Self>) -> Result {
    // Since our wrappers have the right signature, we can just give them to our low level
    // spi_driver!
    this.spi_driver.probe = Some(DriverRegistration::probe_wrapper::<T>);
    this.spi_driver.remove = Some(DriverRegistration::remove_wrapper::<T>);
    this.spi_driver.shutdown = Some(DriverRegistration::shutdown_wrapper::<T>);
}
```

What's the alternative

- However, we can't use the kernel's default implementations, since we always pass functions and never NULL
- We need a way for the user to specify when they want to implement a function, and when to use the Kernel's default
- We could use reflection: If the type *T* implements the function *probe*, use it, otherwise use NULL... This is really slow!
- We can use a virtual/truth table by having the user declare what functions they will implement. This is what's being done for the *FileOperations* abstraction

```rust
pub struct ToUse {
    pub probe: bool,
    pub remove: bool,
    pub shutdown: bool,
}

// We can define constants inside of a trait for implementers to use
pub trait SpiMethods {
    const TO_USE: ToUse;

    ...
}
// Since it's not really pretty nor interesting, we can wrap the declaration of this constant
// in a macro
```

```rust
impl SpiMethods for Mfrc522SpiMethods {
    declare_spi_methods!(probe);

    ...
    // Remember to implement probe properly, or the trait's default implementation (not the
    // kernel's) will be used. This implementation basically does nothing.
    // This macro basically declares the `TO_USE` constant with all fields set to false,
    // except `TO_USE.probe` which is set to true
}
```

```rust
pub fn register<T: SpiMethods>(self: Pin<&mut Self>) -> Result {
    this.spi_driver.probe = if T::TO_USE.probe {
        Some(DriverRegistration::probe_wrapper::<T>)
    } else {
        None
    };
    this.spi_driver.remove = if T::TO_USE.remove {
        Some(DriverRegistration::remove_wrapper::<T>)
    } else {
        None
    };
    // Ugly code, but this is just from the low-level implementation. The user only deals with
    // idiomatic Rust.
}
```

```
pub fn register<T: SpiMethods>(self: Pin<&mut Self>) -> Result {
    this.spi_driver.probe = T::TO_USE.probe.then_some(DriverRegistration::probe_wrapper::<T>);

    // Rust has us covered!
}
```

Probe function, trait edition

Pros

- Better type safety

Cons

- Harder to understand and implement

```rust
pub fn register(self: &mut Self) -> Result {
    let mut spi_driver =
    bindings::spi_driver::default();
    ...
    // Here spi_driver is copied...
    self.spi_driver = Some(spi_driver);
    let res = unsafe {
    // Inside an unsafe block, rustc can't guarantee
    // that spi_driver will not be used outside the
    // scope of the current function
bindings::__spi_register_driver(this.this_module.0,
&mut spi_driver)
    };
    ...
}
```

```
[   36.266704] mfrc522: [MFRC522-RS] Init
[   36.267087] mfrc522: [MFRC522-RS] SPI Registered
[   36.272868] Unable to handle kernel paging request at virtual address ffff80001087b7d0
[   36.282778] Mem abort info:
[   36.286541]   ESR = 0x96000007
[   36.290281]   EC = 0x25: DABT (current EL), IL = 32 bits
[   36.296322]   SET = 0, FnV = 0
[   36.300027]   EA = 0, S1PTW = 0
[   36.303836] Data abort info:
[   36.307375]   ISV = 0, ISS = 0x00000007
[   36.311887]   CM = 0, WnR = 0
[   36.315514] swapper pgtable: 4k pages, 48-bit VAs, pgdp=00000000013f1000
[   36.322967] [ffff80001087b7d0] pgd=0000000001810003, p4d=0000000001810003, pud=0000000001811003, pmd=00000000
[   36.337034] Internal error: Oops: 96000007 [#1] PREEMPT SMP
[   36.343378] Modules linked in: mfrc522 hci_uart btqca btbcm bluetooth ecdh_generic ecc 8021q mrp garp stp llc
[   36.378161] CPU: 3 PID: 539 Comm: systemd-udevd Tainted: G        W         5.12.0-rc4+ #25
[   36.388189] Hardware name: Raspberry Pi 3 Model B Plus Rev 1.3 (DT)
[   36.395343] pstate: 80000005 (Nzcv daif -PAN -UAO -TCO BTYPE=--)
[   36.402250] pc : dev_uevent+0x138/0x1e8
[   36.406966] lr : uevent_show+0x90/0x118
[   36.411677] sp : ffff80001079bba0
[   36.415849] x29: ffff80001079bbc0 x28: ffff670b41b9d400
[   36.422066] x27: 0000000000000000 x26: 0000000000000001
[   36.428281] x25: 0000000000000000 x24: ffff670b49b42700
[   36.434496] x23: ffffdc99f37a4168 x22: ffff670b4191a880
[   36.440711] x21: ffff670b42475800 x20: ffff670b47f59000
[   36.446900] x19: ffff670b42475800 x18: 0000000000000000
[   36.453060] x17: 0000000000000000 x16: 0000000000000000
[   36.459216] x15: ffff670b4342f0d0 x14: 0000000000000000
[   36.465374] x13: 0000000000000001 x12: 0000000000000000
[   36.471522] x11: 0000000000000001 x10: ffff670b47f58000
[   36.477660] x9 : 0de4e88efe7dd600 x8 : ffff80001087b7d0
[   36.483806] x7 : 0000000000000000 x6 : 000000000000003f
[   36.489957] x5 : 0000000000000040 x4 : ffff80001079bb80
[   36.496107] x3 : 0000000000000001 x2 : ffff670b47f59000
[   36.502254] x1 : ffff670b42475800 x0 : ffff670b4191a880
[   36.508389] Call trace:
[   36.511582]  dev_uevent+0x138/0x1e8
[   36.515824]  uevent_show+0x90/0x118
[   36.520048]  dev_attr_show+0x20/0x58
[   36.524339]  sysfs_kf_seq_show+0xa0/0x110
[   36.529046]  kernfs_seq_show+0x2c/0x9c
[   36.533465]  seq_read_iter+0x11c/0x3b4
[   36.537858]  kernfs_fop_read_iter+0x68/0x188
[   36.542758]  vfs_read+0x290/0x2bc
[   36.546666]  ksys_read+0x74/0xe0
[   36.550464]  __arm64_sys_read+0x1c/0x28
[   36.554867]  el0_svc_common+0x90/0x110
[   36.559170]  do_el0_svc+0x24/0x80
[   36.563029]  el0_svc+0x28/0x88
[   36.566622]  el0_sync_handler+0x84/0xe4
[   36.571006]  el0_sync+0x154/0x180
[   36.574856] Code: aa1403e0 97f8c37e f9403668 b40000c8 (f9400102)
[   36.581549] ---[ end trace 556d98a75f645ca9 ]---
```

Rust for Linux

Pros

- It works!

Cons

- Extremely recent
- Subject to changes
- A lot of work still needs to be done

MFRC522 driver: C vs Rust

```c
enum Mfrc522Command {
    Idle = 0,
    Mem
};

void test(enum Mfrc522Command e) {
    printf("%i\n", e);
}

int main(void) {
    test(Idle);
    test(42);
}

// Compiles fine
```

```rust
enum Mfrc522Command {
    Idle = 0b0000,
    Mem = 0b0001,
}
fn test(e: Mfrc522Command) {
    println!("{:?}", e);
}

fn main() {
    test(Mfrc522Command::Idle);
    test(42);
}
```

```
error[E0308]: mismatched types
  --> src/main.rs:12:10
     |
12   |     test(42);
     |          ^^ expected enum `Mfrc522Command`,
found integer
```

```c
struct driver_command {
  const char *input; u8 cmd;
};

struct driver_command commands[MFRC522_CMD_AMOUNT] = {
  { .input = "mem_write", .cmd = MFRC522_CMD_MEM_WRITE },
  ...
  { .input = "version", .cmd = MFRC522_CMD_GET_VERSION },
  { .input = "debug", .cmd = MFRC522_CMD_DEBUG },
};

struct driver_command *find_cmd_from_token(const char *token)
{
  size_t i;

  for (i = 0; i < MFRC522_CMD_AMOUNT; i++)
    if (!strcmp(token, commands[i].input))
      return &commands[i];

  return NULL;
}
```

```rust
pub enum Cmd {
  MemWrite,
  MemRead,
  GetVersion,
  GenRand,
}

impl Cmd {
  pub fn from_str(cmd: &str) -> Option<Cmd> {
    match cmd {
      "mem_read" => Some(Cmd::MemRead),
      "mem_write" => Some(Cmd::MemWrite),
      "get_version" => Some(Cmd::GetVersion),
      "gen_rand_id" => Some(Cmd::GenRand),
      _ => None,
    }
  }
}
```

```c
static inline int
spi_read(struct spi_device *spi,
         void *buf, size_t len)
{
    struct spi_transfer t = {
            .rx_buf = buf,
            .len = len,
        };

    return spi_sync_transfer(spi, &t, 1);
}
```

```rust
#[inline]
fn write(dev: &mut SpiDevice, tx_buf: &[u8]) -> Result {
    Spi::write_then_read(dev, tx_buf, &mut [0u8; 0])
}

#[inline]
fn read(dev: &mut SpiDevice, rx_buf: &mut [u8]) -> Result {
     Spi::write_then_read(dev, &[0u8; 0], rx_buf)
}
```

Merging our work

GIST


ksquirrel commented 6 days ago
Review of 0861303f78de :
Commit 7e65b95 : Looks fine!
Commit cdefa5d : Looks fine!
Commit f650715 : Looks fine!
Commit 0861303 : Looks fine!

Any questions?

- esteban.blanc@lse.epita.fr
- martin.schmidt@lse.epita.fr
- cohenarthur.dev@gmail.com

https://github.com/ks0n/linux
https://github.com/ks0n/mfrc522-linux