THE GOOD THE BAD THE UGLY

May 19, 2026 CDT

208C+D (Level Two)

FROM MALLOC TO BOX: A PRACTICAL GUIDE TO RUSTIFICATION

CHRISTINA QUAST

MALLOC

The Good

4 Reasons to use Rust

Result type and ? operator

```
// C: int status = i2c_read(&buf);
// if(status < 0) return status;
fn read_i2c_sensor() -> Result<u8, i32> {
    if bus_busy() {
        Err(-5)
    } else { Ok(42) }
}
let data = read_i2c_sensor()?;
```

Result type and ? operator

```
// C: int status = i2c_read(&buf);
// if(status < 0) return status;
fn read_i2c_sensor() -> Result<u8, i32> {
    if bus_busy() {
        Err(-5)
    } else { Ok(42) }
}
let data = read_i2c_sensor()?;
```

Match pattern

```
match read_sensor() {
   Ok(0) => println!("Zero"),
   Ok(val) => println!("Read: {val}"),
   Err(e) => println!("Error: {e}"),
}
```

Result type and ? operator

```
// C: int status = i2c_read(&buf);
// if(status < 0) return status;
fn read_i2c_sensor() -> Result<u8, i32> {
    if bus_busy() {
        Err(-5)
    } else { Ok(42) }
}
let data = read_i2c_sensor()?;
```

Options type

```
// C: uint8_t *val = read_rx(); if(val != NULL) { ... }
fn read_uart() -> Option<u8> {
    if rx_ready() {
        Some(get_byte())
    } else { None }
}
if let Some(byte) = read_uart() {
    process(byte);
}
```

Match pattern

```
match read_sensor() {
    Ok(0) => println!("Zero"),
    Ok(val) => println!("Read: {val}"),
    Err(e) => println!("Error: {e}"),
}
```

Lambda functions & Iterators

```rust
let buffer = [0x10, 0x20, 0x30, 0x40];
let calibrated: Vec<u16> =
        buffer.iter()
        .map(|&val| (val as u16) + 5)
        .collect();
```

Result type and ? operator

```rust
// C: int status = i2c_read(&buf);
// if(status < 0) return status;
fn read_i2c_sensor() -> Result<u8, i32> {
    if bus_busy() {
        Err(-5)
    } else { Ok(42) }
}
let data = read_i2c_sensor()?;
```

Options type

```rust
// C: uint8_t *val = read_rx(); if(val != NULL) { ... }
fn read_uart() -> Option<u8> {
    if rx_ready() {
        Some(get_byte())
    } else { None }
}
if let Some(byte) = read_uart() {
    process(byte);
}
```

Match pattern

```rust
match read_sensor() {
    Ok(0) => println!("Zero"),
    Ok(val) => println!("Read: {val}"),
    Err(e) => println!("Error: {e}"),
}
```

Malloc vs Box

MEMORY MANAGEMENT LIFECYCLE COMPARISON: MALLOC/FREE vs. RUST BOX

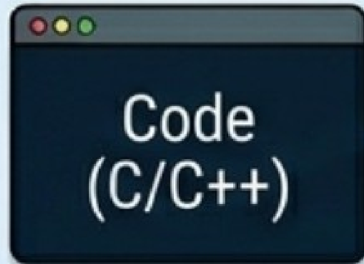
C/C++: MALLOC/FREE	RUST: BOX
MALLOC/FREE: EXPLICIT, MANUAL, PRONE TO ERRORS	**BOX:** IMPLICIT (DROP), AUTOMATIC, MEMORY SAFE

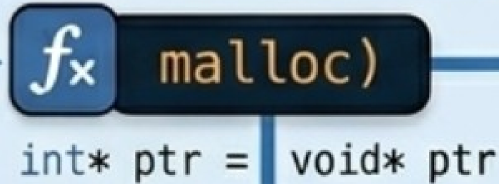
Malloc vs Box

MEMORY MANAGEMENT LIFECYCLE COMPARISON: MALLOC/FREE vs. RUST BOX

C/C++: MALLOC/FREE | RUST: BOX

ALLOCATION





int* ptr = | void* ptr

Raw pointer

Uninitialized data

MALLOC/FREE:
EXPLICIT, MANUAL, PRONE TO ERRORS

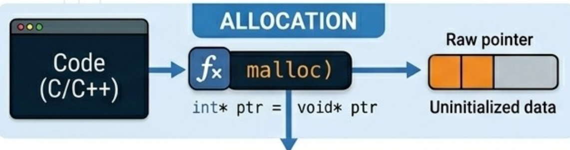
BOX:
IMPLICIT (DROP), AUTOMATIC, MEMORY SAFE

Malloc vs Box

MEMORY MANAGEMENT LIFECYCLE COMPARISON: MALLOC/FREE vs. RUST BOX

C/C++: MALLOC/FREE

RUST: BOX



USE

*ptr = 100;

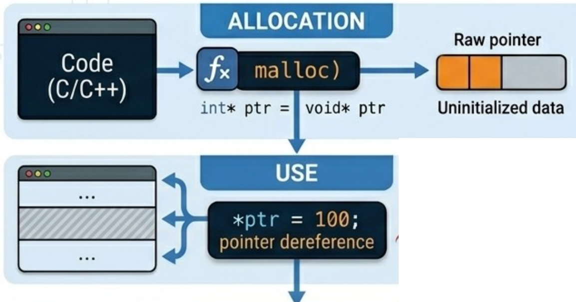
pointer dereference

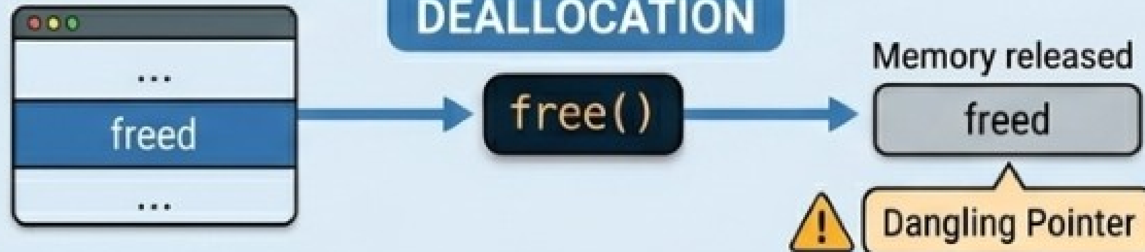
EXPLICIT, MANUAL, PRONE TO ERRORS

BOX:

IMPLICIT (DROP), AUTOMATIC, MEMORY SAFE

Malloc vs Box





TIC, MEMORY SAFE

Malloc vs Box

MEMORY MANAGEMENT LIFECYCLE COMPARISON: MALLOC/FREE vs. RUST BOX

C/C++: MALLOC/FREE	RUST: BOX

ALLOCATION

Code (C/C++) → malloc) → Raw pointer

int* ptr = void* ptr

Uninitialized data

USE

...

...

*ptr = 100;
pointer dereference

DEALLOCATION

...

freed

...

free()

RISKS
(Unsafe)

- Memory Leaks
- Double Free
- Invalid Access

MALLOC/FREE:
EXPLICIT, MANUAL, PRONE TO ERRORS

BOX:
IMPLICIT (DROP), AUTOMATIC, MEMORY SAFE

Malloc vs Box

MEMORY MANAGEMENT LIFECYCLE COMPARISON: MALLOC/FREE vs. RUST BOX

C/C++: MALLOC/FREE

ALLOCATI

Code (C/C++)

malloc

```
int* ptr = voi
```

USE

...

...

```
*ptr = 10
```

pointer derefe

DEALLOCATION

...

freed

...

free()

Memory released

freed

Dangling Pointer

RUST: BOX

ALLOCATION

Code (Rust)

```
Box::new()
let b =
```

Creates owned, initialized value (T)

100)

T>

Automatically returns a Box<T> smart pointer

MALLOC/FREE:

EXPLICIT, MANUAL, PRONE TO ERRORS

BOX:

IMPLICIT (DROP), AUTOMATIC, MEMORY SAFE

Malloc vs Box

MEMORY MANAGEMENT LIFECYCLE COMPARISON: MALLOC/FREE vs. RUST BOX

C/C++: MALLOC/FREE

ALLOCATION

Code (C/C++)

malloc)

int* ptr = void* ptr

Raw pointer

Uninitialized data

USE

...

...

*ptr = 1
pointer derefe

DEALLOCA

...

freed

...

free()

Dangling Pointer

RUST: BOX

ALLOCATION

Code (Rust)

Box::new()

let b =

Creates owned, initialized value (T)

100}

T>

Automatically returns a Box<T> smart pointer

USE

...

T

...

*b;
(or implicit deref)

GUARANTEES
(Safe)

- No Leaks
- Automatic Deallocation
- Valid Access

MALLOC/FREE:
EXPLICIT, MANUAL, PRONE TO ERRORS

BOX:
IMPLICIT (DROP), AUTOMATIC, MEMORY SAFE

Malloc vs Box

MEMORY MANAGEMENT LIFECYCLE COMPARISON: MALLOC/FREE vs. RUST BOX

C/C++: MALLOC/FREE

ALLOCATION

Code (C/C++) → malloc() → Raw pointer: Uninitialized data

`int* ptr = void* ptr`

USE

`*ptr = 100;` pointer dereference

RISKS (Unsafe)
- Memory Leaks
- Double Free
- Invalid Access

DEALLOC

freed → free(

MALLOC

EXPLICIT, MANUAL, P

RUST: BOX

ALLOCATION

Code (Rust) → Box::new() → 100}

Creates owned, initialized value (T)

`let b =`

T> Automatically returns a Box<T> smart pointer

USE

T → `*b;` (or implicit deref)

GUARANTEES (Safe)
- No Leaks
- Automatic Deallocation
- Valid Access

OUT OF SCOPE

Box variable OUT OF SCOPE → Drop::drop() → Memory is instantly and automatically Freed

Malloc vs Box

MEMORY MANAGEMENT LIFECYCLE COMPARISON: MALLOC/FREE vs. RUST BOX

C/C++: MALLOC/FREE

ALLOCATION

Code (C/C++) → `malloc()` → Raw pointer: Uninitialized data

`int* ptr = void* ptr`

USE

`*ptr = 100;` pointer dereference

RISKS (Unsafe)
- Memory Leaks
- Double Free
- Invalid Access

DEALLOCATION

freed → `free()` → Memory released: freed

Dangling Pointer

RUST: BOX

ALLOCATION

Code (Rust) → `Box::new()` → `100}` Creates owned, initialized value (T)

`let b =` → T> Automatically returns a Box<T> smart pointer

USE

T → `*b;` (or implicit deref)

GUARANTEES (Safe)
- No Leaks
- Automatic Deallocation
- Valid Access

OUT OF SCOPE

Box variable OUT OF SCOPE → `Drop::drop()` → Memory is instantly and automatically Freed

MALLOC/FREE: EXPLICIT, MANUAL, PRONE TO ERRORS

BOX: IMPLICIT (DROP), AUTOMATIC, MEMORY SAFE

C cleanup with gotos

```c
int do_kernel_work(size_t size_a, size_t size_b) {
	int ret = ENOMEM;
	char *buf_a = kmalloc(size_a, GFP_KERNEL);
	if (!buf_a)
		goto just_return;
	char *buf_b = kmalloc(size_b, GFP_KERNEL);
	if (!buf_b)
		goto cleanup_a;
	// Core logic goes here
	ret = 0;

	kfree(buf_b);
cleanup_a:
	kfree(buf_a);
just_return:
	return ret;
}
```

The Drop trait

- Compiler's drop checks and writes drop glue
- Called on:
  - Function return (successful or on error with `?` operator), unless transfer of ownership
  - End of {} block
  - Assigning of new value to existing variable
  - `core::mem::drop(x)` (Manual drop)

```
use std::mem::drop;
struct Signal(&'static str);
impl Drop for Signal {
  fn drop(&mut self) {
    println!(">Dropping: {}", self.0);
  }
}
fn consume(_s: Signal) {
  println!("---Inside func body---");
} // <- '_s' goes out of scope and
  //     drops here
```

The Drop trait

- Compiler's drop checks and writes drop glue
- Called on:
  - Function return (successful or on error with `?` operator), unless transfer of ownership
  - End of {} block
  - Assigning of new value to existing variable
  - `core::mem::drop(x)` (Manual drop)

```
fn main() {
  println!("--- Start of Main ---");

  // 1. STANDARD SCOPE BOUNDARY
  let _a = Signal("Std Scope Stack(1)");
  let _b = Signal("Std Scope Stack(2)");

  // 2. REASSIGNMENT DROP
  let mut reassigned =
      Signal("Orig. Value");
  println!("About to reassign...");
  reassigned = Signal("New Value");
  // <---[drop(old_reassigned)]
   ...
```

The Drop trait

- Compiler's drop checks and writes drop glue
- Called on:
  - Function return (successful or on error with `?` operator), unless transfer of ownership
  - End of {} block
  - Assigning of new value to existing variable
  - `core::mem::drop(x)` (Manual drop)

```
...
  // 3. MOVE / EXPLICIT DROP
  let moved =
    Signal("Moved into Function");
  consume(moved);
   // <--- Ownership shifts into Func
   // `consume`

  let explicit =
    Signal("Explicitly Dropped");
  drop(explicit);
   // <--- [drop(explicit)]
...
```

The Drop trait

- Compiler's drop checks and writes drop glue
- Called on:
  - Function return (successful or on error with `?` operator), unless transfer of ownership
  - End of {} block
  - Assigning of new value to existing variable
  - `core::mem::drop(x)` (Manual drop)

```
// 4. TEMPORARY VALUE DROP
// Temp. instance is created to read
// a field, then discarded instantly.
let _temp = Signal("Temp. Expr.").0;
// <---  [drop(temp. signal)]

// 5. SHADOWING (DEFERRED DROP)
let shadowed = Signal("Shadow(1st)");
let shadowed = Signal("Shadow(2nd)");
// <--- "Shadowed (1st)" is hidden,
// NOT dropped yet!
```

The Drop trait

- Compiler’s drop checks and writes drop glue
- Called on:
  - Function return (successful or on error with `?` operator), unless transfer of ownership
  - End of {} block
  - Assigning of new value to existing variable
  - `core::mem::drop(x)` (Manual drop)

```
...
 println!("---Reaching end of main---");
} // <--- 6. END OF BLOCK DROPS
  //(In reverse order of declaration)
  //    <- [drop(shadowed - Second)]
  //    <- [drop(shadowed - First)]
  //    <- [drop(reassigned - New Val.)]
  //    <- [drop(_b)]
  //    <- [drop(_a)]
```

The Drop trait

- Compiler's drop checks and writes drop glue
- Called on:
  - Function return (successful or on error with `?` operator), unless transfer of ownership
  - End of {} block
  - Assigning of new value to existing variable
  - `core::mem::drop(x)` (Manual drop)

```
$ cargo run
--- Start of Main ---
About to reassign...
> Dropping: Orig. Value
--- Inside func body ---
> Dropping: Moved into Function
> Dropping: Explicitly Dropped
> Dropping: Temp. Expr.
---Reaching end of main---
> Dropping: Shadow(2nd)
> Dropping: Shadow(1st)
> Dropping: New Value
> Dropping: Std Scope Stack _b
> Dropping: Std Scope Stack _a
```

TL;DR: The Good

- Using `Box` to mitigate use-after-free, double free, memory leaks compared to malloc
- `Drop` cleans up for you when variable goes out of scope

The Bad

Starting out with Rust

- Steep learning curve fighting the borrow checker
- Understanding and proving lifetimes to the compiler
- Initial prototyping feels slow until code finally runs
- Verbosity of error handling (`Option`, `Result`, unwrapping values, propagating errors through `?` operator, etc) compared to `NULL`
- Powerful macro system that rewrites code before it compiles
- Converting between many different String types (`String` (owned string), `&str` (string slice), `CString`, `&CStr`)

Easy in C but hard in Rust

- **Doubly linked lists (cyclic graphs)**: In Rust you can have **many immutable** references, OR **exactly one mutable** reference (&mut), but never both.
- **Quick prototyping**: In Rust you need to design data ownership model from line 1. Refactoring can require massive rewrites of code.

Easy in C but hard in Rust

- **Global state:** Rust compiler cannot guarantee that when you modify a global variable, this variable is not read or changed by another thread or async interrupt at exactly the same time
- **Pointer to field arithmetic**: Rust tracks "provenance" (where a pointer came from) and uses the "Aliasing" rule. You cannot cast a number to a pointer and de-reference it without `` `unsafe{}` ``.

TL;DR: The Bad

- Rust is harder to start with, easier to maintain at scale.

The Ugly

Foreign Function Interface(FFI) and C bindings

include/linux/miscdevice.h

```
struct miscdevice {
  int minor;
  const char *name;
  const struct file_operations *fops;
  struct list_head list;
  struct device *parent;
  struct device *this_device;
  const struct attribute_group **groups;
  const char *nodename;
  umode_t mode;
};
```

Foreign Function Interface(FFI) and C bindings

include/linux/miscdevice.h

```
struct miscdevice {
  int minor;
  const char *name;
  const struct file_operations *fops;
  struct list_head list;
  struct device *parent;
  struct device *this_device;
  const struct attribute_group **groups;
  const char *nodename;
  umode_t mode;
};
```

rust/bindings/bindings_generated.rs

```
#[repr(C)]
#[derive(Copy, Clone, MaybeZeroable)]
pub struct miscdevice {
    pub minor: ffi::c_int,
    pub name: *const ffi::c_char,
    pub fops: *const file_operations,
    pub list: list_head,
    pub parent: *mut device,
    pub this_device: *mut device,
    pub groups: *mut *const attribute_group,
    pub nodename: *const ffi::c_char,
    pub mode: umode_t,
}
```

Foreign Function Interface(FFI) and C bindings

rust/bindings/bindings_generated.rs

```
impl Default for miscdevice {
    fn default() -> Self {
        let mut s =
            ::core::mem::MaybeUninit::<Self>::uninit();
        unsafe {
            ::core::ptr::write_bytes(s.as_mut_ptr(), 0, 1);
            s.assume_init()
        }
    }
}
```

Foreign Function Interface(FFI) and C bindings

include/linux/miscdevice.h

```
extern int misc_register(
    struct miscdevice *misc);
extern void misc_deregister(
    struct miscdevice *misc);
```

Foreign Function Interface(FFI) and C bindings

include/linux/miscdevice.h

```
extern int misc_register(
    struct miscdevice *misc);
extern void misc_deregister(
    struct miscdevice *misc);
```

rust/bindings/bindings_generated.rs

```
extern "C" {
    pub fn misc_register(misc: *mut
                  miscdevice) -> ffi::c_int;
}
extern "C" {
   pub fn misc_deregister(misc: *mut
                  miscdevice);
}
```

Foreign Function Interface(FFI) and C bindings

rust/kernel/miscdevice.rs

```rust
to_result(unsafe { bindings::misc_register(slot) })
```

rust/kernel/error.rs

```rust
pub fn to_result(err: crate::ffi::c_int) -> Result {
    if err < 0 {
        Err(Error::from_errno(err))
    } else {
        Ok(())
    }
}
```

Foreign Function Interface(FFI) and Rust bindings

drivers/android/binder/rust_binder_main.rs

```
1 /// # Safety
2 /// Only called by binderfs.
3 #[no_mangle]
4 unsafe extern "C" fn rust_binder_new_context(
5     name: *const kernel::ffi::c_char,
6 ) -> *mut kernel::ffi::c_void {
7     // SAFETY: The caller will always provide a valid c string here.
8     let name = unsafe { kernel::str::CStr::from_char_ptr(name) };
9     match Context::new(name) {
10         Ok(ctx) => Arc::into_foreign(ctx),
11         Err(_err) => core::ptr::null_mut(),
12     }
13 }
```

PhantomData, Opaque and Arc

- **Opaque**: Hide C struct's internals from Rust
- **PhantomData**: Type safety, labels resources with its specific driver type
- **Arc**: Keeps C resource alive while in use, cleanup when last reference gone
- **UnsafeCell:** Treat memory as volatile and potentially changing
- **PhantomPinned**: Pinned in memory, cannot be moved by compiler

rust/kernel/block/mq/tag_set.rs

```
#[pin_data(PinnedDrop)]
#[repr(transparent)]
pub struct TagSet<T: Operations> {
    #[pin]
    inner:
Opaque<bindings::blk_mq_tag_set>,
    _p: PhantomData<T>,
}
```

rust/kernel/block/mq/gen_disk.rs

```
pub struct GenDisk<T: Operations> {
    _tagset: Arc<TagSet<T>>,
    gendisk: *mut bindings::gendisk,
}
```

PhantomData, Opaque and Arc

- **Opaque**: Hide C struct's internals from Rust
- **PhantomData**: Type safety, labels resources with its specific driver type
- **Arc**: Keeps C resource alive while in use, cleanup when last reference gone
- **UnsafeCell:** Treat memory as volatile and potentially changing
- **PhantomPinned**: Pinned in memory, cannot be moved by compiler

rust/library/core/src/marker.rs

```
/// guaranteed for all `T`:
/// `size_of::<PhantomData<T>>() == 0`
/// `align_of::<PhantomData<T>>() == 1`
#[lang = "phantom_data"]
pub struct PhantomData<T: PointeeSized>;
#[repr(transparent)]
pub struct Opaque<T> {
    value: UnsafeCell<MaybeUninit<T>>,
    _pin: PhantomPinned,
}
```

ForeignOwnable: Trait

rust/kernel/types.rs

```
pub unsafe trait ForeignOwnable: Sized {
 const FOREIGN_ALIGN: usize;
 type Borrowed<'a>;

 // Convert Rust owned object to foreign owned (C void*)
 fn into_foreign(self) -> *mut c_void;
 // Convert foreign owned object to Rust owned
 unsafe fn from_foreign(ptr: *mut c_void) -> Self;

 // Borrow foreign owned object immutably/muttably
 unsafe fn borrow<'a>(ptr: *mut c_void) -> Self::Borrowed<'a>;
 unsafe fn borrow_mut<'a>(ptr:*mut c_void) -> Self::BorrowedMut<'a>;
}
```

ForeignOwnable: into_foreign()

rust/kernel/block/mq/gen_disk.rs

```
// 1. Convert Rust obj to raw C void * ptr
let data = queue_data.into_foreign();
// 2. Pass that pointer to the C allocator
let gendisk = from_err_ptr(unsafe {
    bindings::__blk_mq_alloc_disk(
        tagset.raw_tag_set(),
        &mut lim,
        data, // C now "owns" ptr
              // to 'queuedata' field
        static_lock_class!().as_ptr(),
    )
})?;
```

ForeignOwnable: ScopeGuard

- If crash or return early before completing setup --> turn pointer BACK into a Rust object so it can be dropped normally
- If we reach the end successfully --> "dismiss" the guard because C now safely owns the pointer.

rust/kernel/block/mq/gen_disk.rs

```
 let recover_data =
     ScopeGuard::new(|| {
       // SAFETY: T::QueueData was
       // created by the call
       // to `into_foreign()`
       // above
       drop(unsafe{
T::QueueData::from_foreign(data)});
 });

 // ... perform C setup ...
 recover_data.dismiss();
```

ForeignOwnable: from_foreign()

- Get the raw void pointer back from the C queue
- Tell C to delete the disk
- Reclaim ownership: Turn the C pointer back into a Rust object, which can properly dropped

rust/kernel/block/mq/gen_disk.rs

```
impl<T: Operations> Drop for GenDisk<T> {
  fn drop(&mut self) {
    let queue_data =
      unsafe {
        (*(*self.gendisk).queue).queuedata
      };
    unsafe {
        bindings::del_gendisk(self.gendisk)
    };
    drop(unsafe{
      T::QueueData::from_foreign(queue_data)
    });
  }
}
```

TL;DR: The Ugly

- `bindgen` automates creating C bindings for Rust from C headers
- The Rust code you write is only as safe as your unsafe Rust and FFI-C interface
- With `ForeignOwnable`, the programmer has to pass objects explicitly between the Rust and C world

Gradual Migration

Phase 1 (**Isolate**): Use bindgen to create raw bindings for existing C data structures.

Phase 2 (**Encapsulate**): Wrap the raw structures in safe Rust types (using Opaque<T> and PhantomData to enforce kernel lifetime invariants).

Phase 3 (**Replace**): Rewrite leaf-node modules (like individual hardware drivers) first, keeping the core subsystem core in C until dependencies are reduced.

A porting Example

“Rustifying” a PHY Driver

drivers/net/phy/Makefile

```
ifdef CONFIG_ROCKCHIP_RUST_PHY
obj-$(CONFIG_ROCKCHIP_PHY)        += rockchip_rust.o
else
obj-$(CONFIG_ROCKCHIP_PHY)        += rockchip.o
endif
```

“Rustifying” a PHY Driver

drivers/net/phy/Kconfig

```
config ROCKCHIP_RUST_PHY
	bool "Rust driver for Rockchip Ethernet PHYs"
	depends on RUST_PHYLIB_ABSTRACTIONS && ROCKCHIP_PHY
	help
	  Uses the Rust reference driver for Rockchip
	  PHYs (rockchip_rust.ko). The features are equivalent.
	  It supports the integrated Ethernet PHY.
```

“Rustifying” a PHY Driver

drivers/net/phy/rockchip_rust.rs

```
#define INTERNAL_EPHY_ID                                0x1234d400
static struct phy_driver rockchip_phy_driver[] = {
        .phy_id                 = INTERNAL_EPHY_ID,
        .phy_id_mask            = 0xfffffff0,
        .name                   = "Rockchip integrated
EPHY",
        /* PHY_BASIC_FEATURES */
        .flags                  = 0,
    ...
}
```

“Rustifying” a PHY Driver

drivers/net/phy/rockchip_rust.rs

```
kernel::module_phy_driver! {
    drivers: [PhyRockchip],
    device_table: [
        DeviceId::new_with_driver::<PhyRockchip>(),
    ],
    name: "rust_rockchip_phy",
    author: "John Doe <john.doe@mail.com>",
    description: "Rust Rockchip PHY driver",
    license: "GPL",
}
```

"Rustifying" a PHY Driver

drivers/net/phy/rockchip_rust.rs

```
struct PhyRockchip;
#[vtable]
impl Driver for PhyRockchip {
    const FLAGS: u32 = 0;
    const NAME: &'static CStr = c_str!("Rockchip integrated EPHY");
    const PHY_DEVICE_ID: DeviceId =
            DeviceId::new_with_custom_mask(0x1234d400, 0xfffffff0);
  ...
}
```

“Rustifying” a PHY Driver

drivers/net/phy.rs

```
#[vtable]
pub trait Driver {
    /// Issues a PHY software
reset.
    fn soft_reset(_dev: &mut
Device) -> Result {
        Err(code::ENOTSUPP)
    }
}
```

drivers/net/phy/rockchip_rust.rs

```
struct PhyRockchip;
#[vtable]
impl Driver for PhyRockchip {
    fn soft_reset(dev: &mut
phy::Device) -> Result {
        dev.genphy_soft_reset()
    }
}
```

“Rustifying” a PHY Driver

File sizes

```
% wc -l rockchip*
200 drivers/net/phy/rockchip.c
131 drivers/net/phy/rockchip_rust.rs
% ls -lh rockchip*ko
-rw-r--r-- 1 chrysh chrysh 14K
drivers/net/phy/rockchip.ko
-rw-r--r-- 1 chrysh chrysh 12K
drivers/net/phy/rockchip_rust.ko
```

Full phy driver code:

TL;DR

- Rust has fewer LoCs and is easier to read
- You are dependent on subsystem maintainer to implement the base infrastructure
- Migrate gradually: Don't rewrite everything at once; isolate through FFI and build safe abstractions step-by-step.

TTL;DR

- **Hardware/FFI is always unsafe**. Isolate raw manipulation strictly behind transparent, audited unsafe barriers.
- **Don't port C architecture line-by-line.** Embrace explicit ownership boundaries. Swap complex pointer graphs for flat, index-based patterns in your internal logic.
- **Development time will increase.** Expect longer design phases upfront compared to C.
- **Safety by design.** Rust eliminates entire classes of vulnerabilities (like memory corruption and data races) at compile time. Once it compiles, it provides a foundational baseline of safety that manual C code review can't match.

Thank you for your attention! 🦀

Freelancer, available for:

- C-to-Rust conversions
- Workshops
- Linux kernel driver development
- Embedded firmware development

✉️ mail@christina-quast.de

🌐 https://christina-quast.de

✍️ https://rust.christina-quast.de

Comments? Suggestions? Your feedback goes here: