Rust for embedded systems, a.k.a. Embedded Rust

4SE02

Guillaume Duc Samuel Tardieu

Février 2025

Introduction

Introduction

- Rust is a great language
- Very attractive for embedded development
  - Memory safety
  - Concurrency safety
  - No cost abstractions

Resources

- Embedded devices Working Group : https://github.com/rust-embedded/wg
- The Embedded Rust Book : https://docs.rust-embedded.org/book/index.html
- The embedonomicon : https://docs.rust-embedded.org/embedonomicon/index.html

Preamble : what operating system to use ?

A full-fledged operating system (e.g., Unix variants) provides services such as :

- memory management (allocation, protection, swap)
- time management (scheduling, time-measurement services)
- resources management (storage, network, display, input peripherals)

In the embedded world, we prefer to go either :

- bare-board : the application must handle all the hardware details ;
- with an embedded operating system : the application benefits from a limited set of features (e.g., hardware initialization, time management, memory protection)

Anatomy of an Embedded Rust Application

Small example

```
#![no_std]
#![no_main]

use panic_halt as _;

use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
    // init code
    loop {
        // application code
    }
}
```

`#![no_std]`

- Bare metal = no operating system nor memory allocator available
- Attribute `#![no_std]` to link with the *core-crate* instead of the *std-crate*
- With it, by default :
  - No heap allocation (unless you use `alloc` and define a global allocator, either from a crate such as `alloc-cortex-m` or a custom one)
  - No collections (you can use fixed-capacity collections with `heapless` crate, or define a global allocator)
  - libstd not available

`#![no_main]`

- The standard `main` interface makes some assumptions on the environment (command line arguments…)
- The attribute `#![no_main]` indicates that the program will not use the standard `main` interface
- The attribute `#[entry]` (defined in the `cortex-m-rt` crate) defines the entry point of your program (you can also do this manually using your linker script)

Panicking behavior

- Panicking is a core part of the Rust language (e.g. `unwrap()` on `Option<T>` and `Error<T>`, `panic!()`, and `assert!()` macros…)
- With `no_std` you need to define one panic handler
  - Signature : `fn …(&PanicInfo) -> !`
  - Declaration with the attribute `#[panic_handler]`
- Several crates implement a panic handler with different behavior : `panic-abort`, `panic-halt`…
  - Use : `use panic_abort as _;`

```
1 // Alternative hand-written implementation
2 #[panic_handler]
3 fn panic_break(info: &PanicInfo) -> ! {
4     breakpoint();  // Signal the debugger if attached
5     loop {}        // We cannot resume execution after a panic
6 }
```

`alloc` and global allocator

If you need dynamic memory allocation (for instance to use common collections), you need a memory allocator. Some are written for you (for instance in the crate `alloc-cortex-m`), but you can also define a custom one.

You first need to define a global allocator that implements the `GlocalAlloc` trait.

```
1  use core::alloc::{GlobalAlloc, Layout};
2
3  struct MyAllocator;
4
5  unsafe impl GlobalAlloc for MyAllocator {
6      unsafe fn alloc(&self, _layout: Layout) -> *mut u8 {
7          // ...
8      }
9      unsafe fn dealloc(&self, _ptr: *mut u8, _layout: Layout) {
10         // ...
11     }
12 }
```

`alloc` and global allocator

Next, you need to instantiate this allocator :

```
#[global_allocator]
static ALLOC: MyAllocator = MyAllocator {};
```

You also need to implement the Out Of Memory error behavior :

```
#![feature(alloc_error_handler)]

#[alloc_error_handler]
fn on_oom(_layout: Layout) -> ! {
    loop {}
}
```

`alloc` and global allocator

Once you global allocator is set up, you can access the collections and other dynamic memory allocation dependent tools from the `alloc` crate :

```
1  #![feature(alloc)]
2
3  extern crate alloc;
4
5  use alloc::vec::Vec;
```

Floating-point operations

Most floating-point operations (for example `f32::atan2(…)`) are defined in `std` and are not available in `no_std` mode.

Several crates (e.g., `micromath`) provide extension traits that add those operations back on predefined floating-point types.

```
1  use micromath::F32Ext;
2
3  // x and z are i16 retrieved from the accelerometer
4  let angle = (-x as f32).atan2(z as f32);
```

Access to peripherals

Access to peripherals

Levels of abstraction :

- Direct access to registers (with pointers)
- Peripheral Access Crate (PAC)
- Hardware Abstraction Layer Crate (HAL)

Direct access (first try)

```
1  struct Usart {
2      pub sr: u32,
3      pub dr: u32, // ...
4  }
5
6  let usart1 = 0x4001_1000 as *mut Usart;
7  let data = unsafe { (*usart1).dr };
```

Direct access (first try)

- Direct access to memory-mapped peripherals' registers with pointers
- Unsafe because Rust has no way to know that there is something at the address you use
- There are two mistakes in the previous code, would you find them ?

Direct access (`repr(C)`)

- The first big mistake is that Rust gives few to no guarantees about the memory organization of the members of a structure (they can even be re-ordered for optimization)
- The attribute `#[repr(C)]` can be used to force Rust to use the C rules

```
1  #[repr(C)]
2  struct Usart {
3      pub sr: u32,
4      pub dr: u32, // ...
5  }
6
7  let usart1 = 0x4001_1000 as *mut Usart;
8  let data = unsafe { (*usart1).dr };
```

Direct access (with volatile)

- As with C, you also have to perform volatile access to prevent the compiler from optimizing memory accesses

```
#[repr(C)]
struct Usart {
    pub sr: u32,
    pub dr: u32, // ...
}

let usart1 = 0x4001_1000 as *mut Usart;
let data = unsafe { core::ptr::read_volatile(& (*usart1).dr) };
```

Direct access (with volatile)

- `core::ptr::read_volatile` and `core::ptr::write_volatile` performs volatile accesses
- From the documentation : *The compiler shouldn't change the relative order or number of volatile memory operations.*

Direct access (with `volatile_register` crate)

```rust
use volatile_register::{RW, RO};

#[repr(C)]
struct Usart {
    pub sr: RW<u32>,
    pub dr: RW<u32>, // ...
}

let usart1 = 0x4001_1000 as *mut Usart;
let data = unsafe { (*usart1).dr.read() }
```

Direct access (with `volatile_register` crate)

- This crate simplifies the access to volatile register
- But you still have to
  - Define by hand the structure of the registers blocks
  - Define by hand the addresses of the registers
  - Use masks to access the different parts of a register

Peripheral Access Crate

```rust
use stm32f4::stm32f401;

let mut peripherals = stm32f401::Peripherals::take().unwrap();
let usart1 = &peripherals.USART1;

// Read
let data = usart1.dr.read().dr().bits();

// Modify
usart1.cr1.modify(|_, w| w.te().set_bit());
usart1.cr1.modify(|r, w| w.re().bit(! r.re().bit()));

// Write
usart1.brr.write(|w| w.div_mantissa().bits(12));
```

Peripheral Access Crate

- A Peripheral Access crate hides low-level details (structure of the registers, addresses, etc.)
- It also provides an interface to manipulate (read, modify, write) the different parts of the peripheral's registers
  - Each register (e.g. `cr1`) in a register block (e.g. `USART1`) exposes several functions (depending on whether the register is read-only, write-only or read-write) : `read`, `modify`, `write`

Peripheral Access Crate (read)

You can read the content of the register :

```
let val = usart1.cr1.read().bits();
```

You can also read a field of the register :

```
1 let reader = usart1.cr2.read();
2 let clken = reader.clken().bit(); // 1 bit
3 let stop = reader.stop().bits(); // 2 bits
```

Peripheral Access Crate (write)

You can write the content of the register :

```
usart1.cr1.write(|w| unsafe{ w.bits(value) });
```

Or write only the fields you need (the other field will have their reset value) :

```
usart1.cr2.write(|w| w.clken().set_bit().stop().bits(0b10));
```

Peripheral Access Crate (modify)

You can modify the content of certain fields without modifying the value of the other fields (read-modify-write) :

```
usart1.cr1.modify(|r, w| w.re().bit(! r.re().bit() ));
```

Hardware Abstraction Layer

```rust
use stm32f4xx_hal as hal;
use hal::{rcc::Clocks, serial::{config, Serial}, stm32};

let p = stm32::Peripherals::take().unwrap();
let rcc = p.RCC.constrain();
let clocks: Clocks = rcc
    .cfgr
    .sysclk(64.mhz())
    .pclk1(32.mhz())
    .pclk2(64.mhz())
    .freeze();
let gpioa = dp.GPIOA.split();
let tx_pin = gpioa.pa9.into_alternate_af7();
let rx_pin = gpioa.pa10.into_alternate_af7();
let usart1_config = config::Config {
    baudrate: 9_600.bps(),
    wordlength: config::WordLength::DataBits8,
    parity: config::Parity::ParityNone,
    stopbits: config::StopBits::STOP1,
};

let usart1 = Serial::usart1(p.USART1, (rx_pin, tx_pin), usart1_config, clocks).unwrap();
usart1.write(b'A').unwrap();
```

Hardware Abstraction Layer

Various HAL work differently. On LPC55S69, the HAL tracks the state of the various subsystems and can switch between disabled (default) and enabled states. Most functions only work when the subsystem has been enabled :

```
#[init] fn init(cx: init::Context) {
  let mut hal = hal::Peripherals::from((cx.device, cx.core));

  let mut iocon = hal.iocon.enabled(&mut hal.syscon);
  let mut gpio = hal.gpio.enabled(&mut hal.syscon);
  let pins = hal::Pins::take().unwrap();

  let blue_led = pins
    .pio1_4
    .into_gpio_pin(&mut iocon, &mut gpio)
    .into_output_high();
  …
}
```

Using types, Rust can ensure at compile-time that the subsystem has been properly enabled.

Tools

ARM Cortex-M targets

Install precompiled `core` crate for a specific architecture :

```
$ rustup target add thumbv7em-none-eabihf
```

List of targets :

- `thumbv6m-none-eabi` : Cortex-M0 and Cortex-M0+
- `thumbv7m-none-eabi` : Cortex-M3
- `thumbv7em-none-eabi` : Cortex-M4 and Cortex-M7 (no FPU)
- `thumbv7em-none-eabihf` : Cortex-M4F and Cortex-M7F (with FPU)
- `thumbv8m.base-none-eabi` : Cortex-M23
- `thumbv8m.main-none-eabi` : Cortex-M33 (no FPU)
- `thumbv8m.main-none-eabihf` : Cortex-M33 (with FPU)

Project configuration

If you want to start a new project for Cortex-M, you can clone the cortex-m-quickstart project template. Or you can setup your project manually.

Project configuration : Cargo.toml

Two dependencies are essential :

- `cortex-m`
- `cortex-m-rt`

You can add :

- panic behavior crate (e.g. `panic-halt`)
- peripheral access crate (e.g. `stm32f3`)
- hardware abstraction layer crate (e.g. `stm32f4xx-hal`)
- RTIC (`cortex-m-rtic`)
- ...

Project configuration : .cargo/config.toml - Target

```
# .cargo/config.toml

[build]
# Pick ONE of these compilation targets
# target = "thumbv6m-none-eabi"        # Cortex-M0 and Cortex-M0+
target = "thumbv7m-none-eabi"        # Cortex-M3
# target = "thumbv7em-none-eabi"       # Cortex-M4 and Cortex-M7 (no FPU)
# target = "thumbv7em-none-eabihf"     # Cortex-M4F and Cortex-M7F (with FP
# target = "thumbv8m.base-none-eabi"   # Cortex-M23
# target = "thumbv8m.main-none-eabi"   # Cortex-M33 (no FPU)
# target = "thumbv8m.main-none-eabihf" # Cortex-M33 (with FPU)
```

Project configuration : .cargo/config.toml - Linker flags

```
# .cargo/config.toml

[target.'cfg(all(target_arch = "arm", target_os = "none"))']

rustflags = [
  # This is needed if your flash or ram addresses are not aligned to 0x1000
  # See https://github.com/rust-embedded/cortex-m-quickstart/pull/95
  "-C", "link-arg=--nmagic",

  # LLD (shipped with the Rust toolchain) is used as the default linker
  "-C", "link-arg=-Tlink.x",
]
```

Project configuration : .cargo/config.toml - Runner

It is possible to configure the behavior of `cargo run` :

```
# .cargo/config.toml

[target.thumbv7m-none-eabi]
# runner = "qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb \
#    -nographic -semihosting-config enable=on,target=native -kernel"
runner = "arm-none-eabi-gdb -q -x openocd.gdb"
```

Project configuration : memory.x

The file `memory.x` is merged with the linker script provided by the crate `cortex-m-rt`.

It must contains at least the memory layout of the targeted chip :

```
MEMORY
{
  FLASH : ORIGIN = 0x00000000, LENGTH = 256K
  RAM : ORIGIN = 0x20000000, LENGTH = 64K
}
```

By default :

- The section containing the interrupt vectors table and the sections `.text` and `.rodata` are stored in the region `FLASH`
- The section `.data` is stored in `FLASH` and copied in `RAM`
- The sections `.bss` is created in `RAM`
- The stack pointer is initialized at the end of `RAM`

Project configuration : memory.x

It is also possible to define more memory regions and change the locations of sections. Example :

```
MEMORY
{
  FLASH : ORIGIN = 0x00000000, LENGTH = 256K
  RAM : ORIGIN = 0x20000000, LENGTH = 64K
  CCRAM : ORIGIN = 0x10000000, LENGTH = 8K
}

_stack_start = ORIGIN(CCRAM) + LENGTH(CCRAM);
_stext = ORIGIN(FLASH) + 0x40c;
```

ARM semihosting

Semihosting (as in "not on an operating system but almost as-if") is a mechanism in which some `svc` or `bkpt` instructions are interpreted as a system call. This allows an embedded program to exchange data with the host, or even read or write files on the host.

It works as follows (on Cortex-M, using `bkpt`) :

- The syscall number is placed into `r0`
- The argument (or pointer to an argument block) is placed into `r1`
- `bkpt 0xab` is issued

The probe or the debugger checks the `bkpt` argument using the IP register. If it is equal to `0xab`, and `r0` denotes a valid syscall, it is executed on the host.

ARM semihosting pitfalls

Semihosting is not a flawless solution though :

- It is slow, as it requires to stop the processor and have the probe or the debugger exchange data with the target memory. Segger RTT proprietary feature is much faster when using a Segger probe.
- If no probe is connected, caution must be taken not to use semihosting instructions as by default `bkpt` will generate a hardfault when no probe is present.

Knurling

Knurling is an ongoing project led by company Ferrous Systems aiming to provide better tools for Rust embedded debugging :

- `probe-rs` : interact with various probes (CMSIS-DAP, JLink, STLink)
- `cargo-flash` : easily flash your embedded board
- `probe-run` : run embedded programs just like native ones, including tests
- `defmt` : deferred formatting to format log entries on the host rather than on the target
- `flip-link` : stack overflow detection