
# Mastering Rust for Embedded Firmware Development

## ESE Konference Dec. 2024

# Embedded Rust Tool Chain

Rust for firmware development

## Aims and Topics

In this module we will learn about the various tools used in a Rust based embedded software development environment.

- Minimal environment requirements
- Interactive debugging
- Basic logging

## General Environment Setup with `rustup`

- `rustup` provides the support for:
  - `rustc` / `cargo`
- Install `rustup`
  - https://rust-lang.org/tools/install
- Or, update existing install
  - `rustup update`
- Check the environment:
  - `rustup show`

# The Rust Compiler

- rustc is is the main rust compiler
  - Open source, used for multiple targets
  - `rustup target list` to see supported targets
  - `rustup target add <target>` to add a target support
- Commercial alternatives exist for ISO 26262's ASIL D and IEC 61508's SIL 4 certification levels
  - Hitech (tailored for AURIX™ TC3x and TC4x microcontrollers)
  - AdaCore
  - Ferrocene

## Cross-Compilation with `cargo`

- Cargo is both a package manager and build tool
  - Dependencies are pulled from `crates.io`
  - Can be pointed to the file system or `github` repo
- The `cargo` command is typically used in favour of `rustc`
  - The `--target flag` is provided to `cargo` / `rust` for cross compiling

```
cargo build --target thumbv7em-none-eabihf
```

- The default generated output will be using the ELF format
  - Not to be confused with a bin file

## Beyond `cargo build`: `binutils`

- Cargo subcommands wrapper can use the LLVM tools for working with binaries and object files
  - `nm` lists labels used inside an object
  - `size` provide size information about sections
  - `objdump` disassembler

```
cargo install cargo-binutils
rustup component add llvm-tools
```

# Beyond `cargo build`: `cargo-generate`

`cargo-generate` used for creating new cargo projects from templates

- Augmented version of `cargo new`
- Facilitates the creation of baremetal projects
  - Provides missing linker scripts and projects settings
  - Only need to fill in the missing information (name, target hardware specifics)

```
cargo install cargo-generate
```

```
cargo generate --git https://github.com/rust-embedded/cortex-m-quickstart
cargo generate esp-rs/esp-idf-template cargo
```

## Cargo Runners

Runners are **configuration options** specifying a command to run your binary instead of executing it directly.

- Useful for cross-compilation scenarios (targets, debuggers, emulators..)
- Located inside: `.cargo/config.toml`

Running GDB for a Arm Cortex-M target:

```
[target.'cfg(all(target_arch = "arm", target_os = "none"))']
runner = "arm-none-eabi-gdb -q -x openocd.gdb"
```

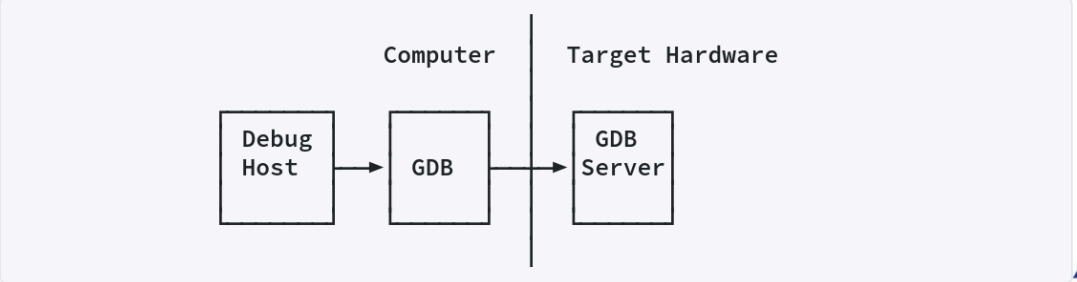
# The Embedded Software Developer's Tool Box

Traditional software development tools can be used with Rust binaries.

- Querying file:
  - `file exec.elf`
- Looking for strings in a file
  - `strings exec.elf`
- Reading ELF file information
  - `readelf -h exec.elf`
  - `readelf --segments exec.elf`
- Hex file dump:
  - `hexdump -C exec.elf`
  - `xxd exec.elf | less`
- Disassembling:
  - `objdump -d exec.elf`

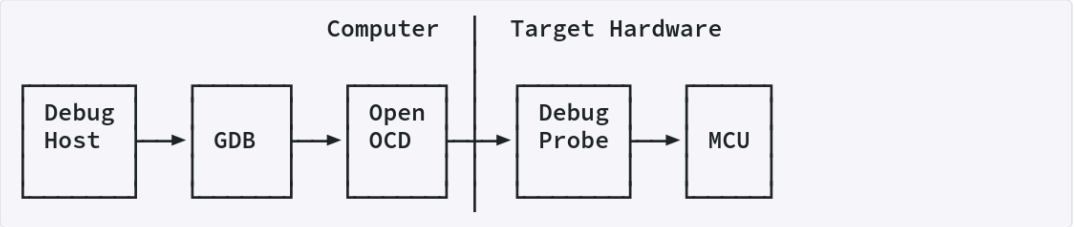
# GDB Overview

- The GNU Debugger is used to debug applications including Rust
- Command line driven ( `step` , `continue` , `break` ..)
- Can be used with:
  - `cargo-embed` , through probe.rs (simpler)
  - `OpenOCD` more traditional and generic



# OpenOCD Overview

- Open On-Chip Debugger interfaces GDB with a target debug adapter
  - Provide the GDB server required by GDB
  - Supports JTAG and SWD interfaces
- Some hardware solution bypass the need for OpenOCD:
  - Black magic probe (provides a GDB Server)



## Basic Printing Through Semihosting

Semihosting is a mechanism that lets embedded devices do **I/O on the host** and is mainly used to **log messages** to the host console.

- Minimal requirements (a debug session)
  - Lowest common denominator across most MCUs
  - Convenient to use
  - Very slow (hundreds of milliseconds)
- For Arm cortex-m MCUs use: `cortex-m-semihosting` crate
  - You do need to enable semihosting in OpenOCD from GDB first:

```
(gdb) monitor arm semihosting enable
```

# Semihosted Logging Example

- `hprintln!()` uses the same type of formatting as for `println!()` can be used
- The `dbg!()` macro can also be used for quick debugging

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

use panic_halt as _;
use cortex_m_rt::entry;
use cortex_m_semihosting::hprintln;

#[entry]
fn main() -> ! {
    hprintln!("Hello, world!").unwrap();
    loop {}
}
```

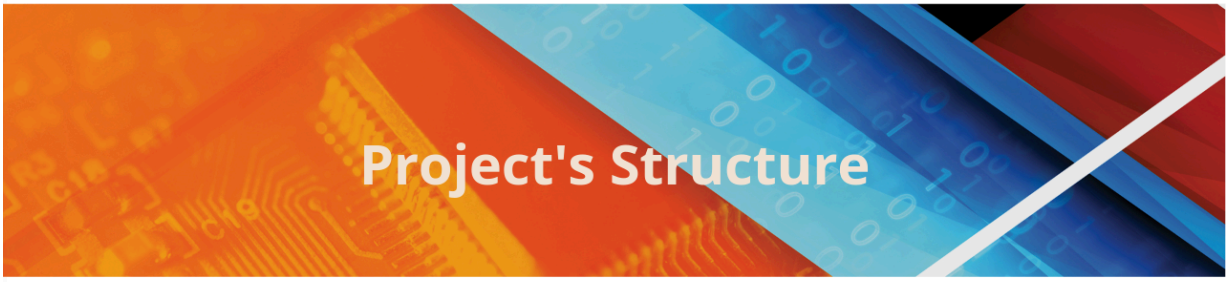
# Project Configuration and Hardware Platforms

Rust for firmware development

## Aims and Topics

In this module we will learn about the overall structure of a enbedded-Rust project as well as what constinute a `no_std` rust program.

- Project's structure
- The `no_std` embedded development environment
- Commonly used embedded crates



# Project's Files Overview

- `Cargo.toml` project configuration information, dependencies specification
- `config.toml` build information, run commands short cuts (flash/debug..)
- `memory.x` is a linker script short-hand
- Optionally, `build.rs` , `Embed.toml`

```
.
|__ .cargo
|   |__ config.toml
|__ Cargo.lock
|__ Cargo.toml
|__ memory.x
|__ build.rs [OPTIONAL]
|__ Embed.toml [OPTIONAL]
|__ src
    |__ main.rs
```

# Minimal `Cargo.toml` File

- `profile.dev`
  - Optimisation level set to 0
  - Debug info on
- `profile.release`
  - Optimisation level 3
  - Debug info off
- Panic strategy has to be defined for an embedded application

```
[package]
name = "crate_name"
version = "0.1.0"
authors = ["Author Name <author@example.com>"]

# the profile used for `cargo build`
[profile.dev]
panic = "abort" # disable stack unwinding on panic

# the profile used for `cargo build --release`
[profile.release]
panic = "abort" # disable stack unwinding on panic
```

## Customising Profiles Inside `Cargo.toml`

Profiles can be customized extensively.

- Optimisation level ( `opt-level` )
  - `0.. 3`
  - Image's size: `s` , `z`
- Debug symbols ( `debug` )
  - `true` / `false`
- Link-time optimisation ( `lto` )
  - `true` / `false`
- Panic ( `panic` )
  - `unwind` / `abort` (better for embedded)

```
[profile.dev]
opt-level = "0"
debug = true
panic = "abort"

[profile.release]
opt-level = "s"
debug = false
panic = "abort"
lto = true
```

# Adding Project's Dependencies to Cargo.toml

```
[dependencies]
# Regular dependencies for the Cortex-M architecture
cortex-m = "0.7"
cortex-m-rt = { version = "0.7", features = ["device"] }
panic-halt = "0.2"

# A hardware abstraction layer for a specific microcontroller family
stm32f1xx-hal = { version = "0.6", features = ["rt", "stm32f103", "medium"] }

# Use no default features to minimize the footprint
embedded-hal = { version = "0.2.4", default-features = false }

# Conditional dependency only for ARM architecture without an OS
[target.'cfg(all(target_arch = "arm", target_os = "none"))'.dependencies]
cortex-m-log = { version = "0.8", features = ["itm"] }

[dev-dependencies]
# Dependencies used only for compiling and running tests, not included in the release builds
proptest = "0.10.1"
```

## Cargo: The `config.toml` File

File used in the context of **embedded software** development.

- Defines flags for the **compiler**
- **Linker** arguments
  - Scripts, optimisations..
  - Additional debug features like defmt..
- **Build target** architecture for the Cargo build command
- **Runners** specifying how to run your application
  - Flash it
  - Run it on QEMU with GDB...

# `config.toml` File Example

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


rustflags = [
  "-C", "linker=flip-link",
  "-C", "link-arg=-Wl,--nmagic",
  "-C", "link-arg=-Wl,-Tlink.x",
  "-C", "link-arg=-Wl,-Tdefmt.x",
  "-C", "inline-threshold=5",
  "-C", "no-vectorize-loops",
]

[build]
# target = "thumbv6m-none-eabi"        # Cortex-M0 and Cortex-M0+
target = "thumbv7m-none-eabi"        # Cortex-M3
# target = "thumbv7em-none-eabihf"     # Cortex-M4F and Cortex-M7F (with FPU)
# target = "thumbv8m.base-none-eabi"   # Cortex-M23
# target = "thumbv8m.main-none-eabi"   # Cortex-M33 (no FPU)
```

## Cargo: The `memory.x` Linker Script

File used to describe the memory layout of the target device.

- This includes defining the sizes and boundaries of different memory regions
- Used by crates like `cortex-m-rt`, `esp32c_rt`, etc.
- The `link.x` cross references the `memory.x` definitions

```
MEMORY
{
  FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 256K
  RAM (rwx)  : ORIGIN = 0x20000000, LENGTH = 40K
}
```


## What Does `#![no_std]` Mean?

`#![no_std]` is a **crate level attribute** found inside the `main.rs` or `lib.rs` file.

- Indicates that the crate will link to the `core` crate instead of the `std` crate
- The `core` crate is a **subset** of the `std` crate that makes zero assumptions about the system the program will run on
  - Lacks the APIs for anything that involves heap memory allocations, networking and I/O
  - Must initialize its own runtime

## How Are Attributes Used?

Rust has a rich set of attributes that can be used to **control** various aspects of code **compilation** and **behaviour**.

- Crate-Level: `#![no_std]`, `#![feature]`, `#![no_main]`
- Conditional Compilation: `#[cfg]`, `#[cfg_attr]`
- Lint: `#[allow]`, `#[warn]`, `#[deny]`
- Derive: `#[derive]`
- Function and Method: `#[inline]`, `#[test]`
- Item: `#[repr]`
- Many more...

# Module and Item Attributes Example

Attributes that apply to modules, structs, enums, and other items.

`#[repr(...)]` : Controls the memory layout of structs and enums.

```
#[repr(C)]
struct MyStruct {
    field1: i32,
    field2: u8,
}
```

`#[repr(align(N))]` : Used to control the memory layout of structs and enums

```
#[repr(align(16))]
struct AlignedStruct {
    data: u32,
    more_data: u64,
}
```

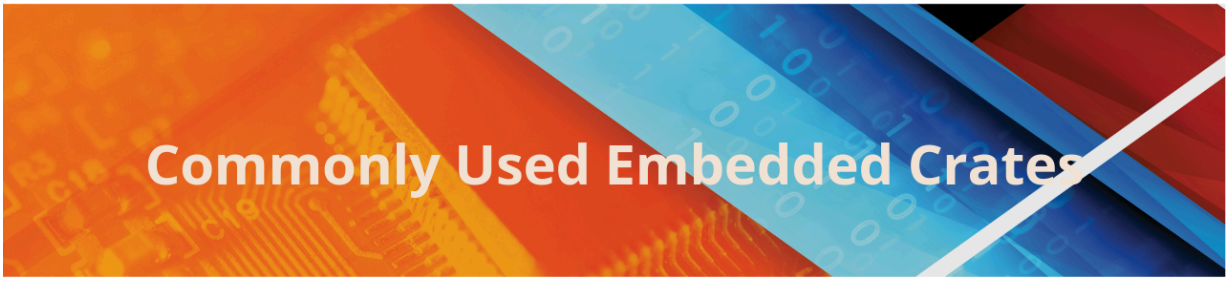
# Minimal Freestanding Rust Binary

```rust
#![no_std] // don't link the Rust standard library
#![no_main] // disable all Rust-level entry points

use core::panic::PanicInfo;

#[no_mangle] // don't mangle the name of this function
pub extern "C" fn _start() -> ! {
    // this function is the entry point, since the linker looks for a function
    // named `_start` by default
    loop {}
}

/// This function is called on panic.
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```



## Processors' Support

A **wealth** of processor architectures are **supported** by the Rust ecosystem either at the PAC level or PAC/HAL level.

- Arm based MCU/CPU
  - Infineon, NXP, STMicrolectronics, Nordic, Atmel, Renesas
- RISC-V based processors
- Infineon TriCore
- Expressif ESP32 processors
- Microchip AVR
- Texas Instruments MSP
- MIPS

## Arm Cortex-* Crates

- Cortex-A/R/M are at the core of many SoCs
- Cortex-M is the dominant generic MCU architecture
- Some relevant Cortex-M crates:
  - `cortex-m` : PAC for the core peripherals (timer, debugger, etc..)
  - `cortex-m-rt` : minimal runtime (entry point, interrupts, exception, ...)
  - `panic-itm` : Panic handler that sends messages over the ITM/SWO output
  - `panic-semihosting` : Panic handler that sends messages over semihosting

## The Run-time `cortex-m-rt` Crate

- Specifies how does our code gets located inside our MCU memory map
- Linker script (generic for Cortex-m)
- Locates the vector table
- Initialise static variables
- Enable FPU (if it exists)
- Provides function attributes
  - `#[entry]` to declare the entry point of the program
  - `#[exception]` to override an exception handler
    - All exceptions default to an infinite loop
  - `#[pre_init]` to run code before static variables are initialised

## `cortex-m-rt` Functions' Attributes Example

```
// IMPORTANT the standard `main` interface is not used because it requires nightly
#![no_main]
#![no_std]

// Some panic handler needs to be included. This one halts the processor on panic.
use panic_halt as _;

use cortex_m_rt::entry;

// use `main` as the entry point of this application
// `main` is not allowed to return
#[entry]
fn main() -> ! {
    // initialization
    loop {
        // application logic
    }
}
```

# Cortex-M Micro-Architectural Crate Example

The `SysTick` peripheral that's common to all Cortex-M based MCUs.

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

use cortex_m::peripheral::{syst, Peripherals}; // ← 💡 Cortex-M micro-architecutal crate
use cortex_m_rt::entry;
use panic_halt as _;

#[entry]
fn main() -> ! {
    let peripherals = Peripherals::take().unwrap();
    let mut systick = peripherals.SYST; //  ← 💡 Timer common to all Cortex-M
    systick.set_clock_source(syst::SystClkSource::Core);
    systick.set_reload(1_000);
    systick.clear_current();
    systick.enable_counter();
    while !systick.has_wrapped() {}
    loop {}
}
```

# Embedded Software Development

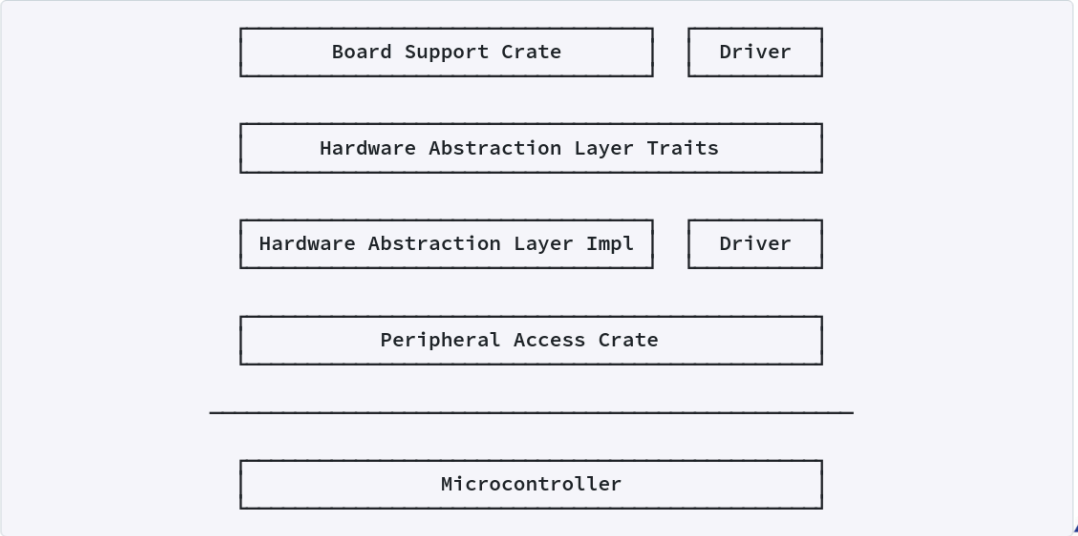
Rust for firmware development

## Aims and Topics

In this chapter we will learn about the specific aspects related to software development for an embedded application.

- Embedded Rust ecosystem
- Mixing Rust with assembly


# Abstraction Layers



## Unsafe Rust

Memory locations of mapped hardware can be accessed just like in embedded C but it is `unsafe`

- Large amount of technical knowledge needed when controlling hardware
  - Read the (very large) manual!
- Very error prone
  - No safety net, no validity or dependencies checks

# Unsafe Code Example

```
static HELLO: &[u8] = b"Hello World!";

#[entry]
fn main() -> ! {
    let vga_buffer = 0xb8000 as *mut u8; // raw pointer to memory

    for (i, &byte) in HELLO.iter().enumerate() {
        unsafe { // unsafe block for dereferencing the raw pointers
            *vga_buffer.offset(i as isize * 2) = byte;
            *vga_buffer.offset(i as isize * 2 + 1) = 0xb;
        }
    }

    loop {}
}
```

# Peripheral Access Crates

PACs provide low level access to memory mapped peripherals inside a given MCU

- Adds zero cost abstraction
- Usually auto-generated by the `SVD2Rust` program
  - Using the CMIS-SVD format
- Limited usefulness due to MCU complexity
  - Maximum control of the hardware
  - Stepping stone to the creation of an HAL

## The PAC Application Programming Interface

A singleton design pattern is used for peripherals

- Ensures only one instance of device peripherals can be used

```
let core_peripherals = stm32f401_pac::Peripherals::take().unwrap();
```

- `take` () method returns an `Option<T>`.
  - First call returns an instance; Subsequent calls return `None`.
- The `core_peripheral` handle is used to create instances for specific peripherals.

# Peripheral Access Manipulation

- Access format
  - `read` : Read values from device registers.
  - `modify` : Modify specific bits in device registers.
  - `write` : Write new values to device registers.
- Usage Format:

`[Peripheral Handle].[Peripheral Register Name].[Operation]`

```
let dp = stm32f401_pac::Peripherals::take().unwrap();
// Enabling the clocks for GPIOA and GPIOC
dp.RCC.write(|w| w.gpioaen().set_bit().gpiocen().set_bit());
// Toggling PA6 Output Using the modify Method
dp.GPIOA.odr.modify(|r, w| w.odr6().bit(!r.odr6().bit()));
```

# PAC Use Example

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

use panic_halt as _;

use cortex_m_rt::entry;
use nrf52833_pac::Peripherals;

#[entry]
fn main() -> ! {
    if let Some(p) = Peripherals::take() {
        p.P0.pin_cnf[28].write(|w| w.dir().output());   // <-┐ This configures pins 28 and 21 of port P0 as output pins
        p.P0.pin_cnf[21].write(|w| w.dir().output());   // <-┘

        p.P0.out.write(|w| unsafe { w.bits(1 << 21) }); // <-- Setting the initial state of pin 21 to high

        let mut count: u8 = 0;
        loop {
            count += 1;

            if count & 1 == 1 {
                p.P0.out.write(|w| unsafe { w.bits(1 << 21) }); // <-- Setting the pin 21 to high
            } else {
                p.P0.out.write(|w| unsafe { w.bits(0) });       // <-- Setting the pin 21 to low
            }

            for _ in 0..50_000 {
                cortex_m::asm::nop();
            }
        }
    };

    loop {
        continue;
    }
}
```

# What is an HAL?

An Hardware Abstraction Layer provides **portability** for embedded systems with varying peripherals capabilities across different vendors and families

- Uniform interface for different hardware platforms through standardisation
- Developers can write device independent applications
- Abstraction of platform-specific details
- Active community and ecosystem support
  - `crates.io`
  - HAL implementation crates on https://github.com/rust-embedded/awesome-embedded-rust

# HAL Example

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

use panic_halt as _;

use embedded_hal::digital::OutputPin; // Provides required traits
use cortex_m_rt::entry;
use nrf52833_hal as hal;
use nrf52833_hal::{gpio::Level, pac::Peripherals};

#[entry]
fn main() -> ! {
    if let Some(p) = Peripherals::take() {
        let port0 = hal::gpio::p0::Parts::new(p.P0); // ◄——— Split the GPIO port P0 into individual pins for configuration

        // Configure P0.28 and P0.21 as output pins
        let mut col1 = port0.p0_28.into_push_pull_output(Level::Low); // ◄——— Configure P0.28 as a push-pull output, initially set to Low
        let mut row1 = port0.p0_21.into_push_pull_output(Level::High); // ◄——— Configure P0.21 as a push-pull output, initially set to High
        let mut count: u8 = 0;
    loop {
            count += 1;

            if count & 1 == 1 {
                row1.set_high().unwrap(); // ◄——— Set P0.21 (row1) to High (turn on the corresponding output)
            } else {
                row1.set_low().unwrap(); // ◄——— Set P0.21 (row1) to Low (turn off the corresponding output)
            }

            for _ in 0..50_000 {
                cortex_m::asm::nop();
            }
        }
    }
}
```

# Embedded Rust Board Support Package

The BSPs are an addition to HALs as they abstract board specific feature for given boards

- Defines specific pins, MCU configurations, hardware for a board

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

use panic_halt as _;
use cortex_m_rt::entry;
use embedded_hal::{delay::DelayNs,
                   digital::OutputPin};
use microbit::{board::Board,
               hal::timer::Timer};

#[entry]
fn main() -> ! {
    let mut board = Board::take().unwrap();
    let mut timer = Timer::new(board.TIMER0);
    let _ = board.display_pins.col1.set_low();
    let mut row1 = board.display_pins.row1;
```

```
// continuing...
    loop {
        let _ = row1.set_low();
        timer.delay_ms(1_000u32);
        let _ = row1.set_high();
        timer.delay_ms(1_000u32);
    }
}
```


# Why Use Intrinsics?

- Intrinsics provide access to specific CPU instructions that are not typically exposed by higher-level languages
- Useful for performance optimisation and accessing specialised hardware features
  - Limited scope: `wfe`, `wfi`, `dsb`, `isb`, `dmb`, `bkpt`...

```
use cortex_m::asm::isb;
fn main() {
    // Use the Instruction Synchronization Barrier
    unsafe {
        isb();
    }
}
```

# Inline Assembly

Inline assembly is provided by the macro `asm!()`

- Can be used to embed handwritten assembly in the output generated by `rustc`
  - x86, x86-64, AArch64, AArch32, RISC-V, LoongArch, TriCore...

- Inference example:

```
fn reverse_bytes_manual(x: u32) -> u32 {
    ((x & 0x000000FF) << 24) |
    ((x & 0x0000FF00) << 8) |
    ((x & 0x00FF0000) >> 8) |
    ((x & 0xFF000000) >> 24)
}
```

- Inline assembly example:

```
fn reverse_bytes(x: u32) -> u32 {
    let result: u32;
    unsafe {
        asm!(
            "rev {0}, {1}",
            out(reg) result,  // Output operand
            in(reg) x         // Input operand
        );
    }
    result
}
```

# Module Level Inline Assembly

The `global_asm!()` macro is used for defining startup code, interrupt vector tables, or other essential assembly routines that are too large or complex for `asm!()`.

- Functions can be written completely in assembly
  - Used for startup routines that initialise hardware and memory before Rust's main function starts
  - Used for defining an interrupt vector table

```
global_asm!(r#"
.globl my_assembly_func

my_assembly_func:
    bx lr
"#);

extern {
    fn my_assembly_func();
}
```

# Mixing Rust with C

Rust for firmware development

## Aims and Topics

In this module we will look at the foreign functions' interface and how to use it to mix both C and Rust code.

- FFI and ABI in Rust
- Calling C from Rust
- Calling Rust from C

## What is a Foreign Function Interface?

FFI is a system-level mechanism for **interacting** with different programming **languages**.

- In the Rust context we want to:
  - Facilitate the **reuse** of existing **libraries** written in languages like C within Rust code
  - **Expose Rust** functions for use in C or C++ projects, enhancing safety through Rust's guarantees

# Procedure Call Standards in Rust, C, and Assembly

- The PCS is a component of the ABI that focuses on **function calls**
  - Rust relies on the ABI defined by the target platform (AArch64, AArch32, etc.)
  - Pragmatically, Rust commonly uses the C calling conventions
- **AArch32 PCS Example:**
  - First 4 function arguments map to R0-R3
  - Additional arguments are stacked
  - 64-bit types map to consecutive registers (even index first)
  - The stack pointer is 8-byte aligned at the object's boundaries

# C ABI: Register Usage Example in Calls

# C ABI: Data Alignments of Arrays

All values have an **alignment** and **size**.

- The alignment of a value specifies which **addresses** are **valid** for storing the value.
  - For example, a `u32` has a size of 4 bytes and an alignment of 4 (`size_of::<u32>()`).
- Proper alignment ensures **efficient** data access and can **prevent hardware exceptions** on some architectures.
- An array of `[T; N]` (a fixed-size array in Rust) has a size of `size_of::<T>() * N` and the same alignment as `T`

# C ABI: Data Alignments of Structures

The alignment of a `struct` with a C representation is the alignment of the most-aligned field in it

```rust
use std::mem::{size_of, align_of};

#[repr(C)]
struct ExampleStruct {
    field1: u8,       // 1-byte alignment
    field2: u32,      // 4-byte alignment
    field3: i16,      // 2-byte alignment
}

fn main() {
    // Assertions for size and alignment checks
    assert_eq!(size_of::<ExampleStruct>(), 12);
    assert_eq!(align_of::<ExampleStruct>(), 4);
}
```

```rust
use std::mem::{size_of, align_of};

#[repr(C)]
struct ExampleStruct {
    field_1: u8,         // 1-byte alignment
    padding_1: [u8; 3],  // Padding for alignment
    field_2: u32,        // 4-byte alignment
    field_3: i16,        // 2-byte alignment
    padding_2: [u8; 2],  // Padding for alignment
}

fn main() {
    // Assertions for size and alignment checks
    assert_eq!(size_of::<ExampleStruct>(), 12);
    assert_eq!(align_of::<ExampleStruct>(), 4);
}
```

# C ABI: Data Alignments of Enums

The use of `#[repr(C)]` to ensure that enums have a predictable memory layout compatible with C.

- The Enum size determination is based on the largest variant and the discriminant

```
use std::mem::{size_of, align_of};
#[repr(C)]
enum Status {
    Idle,                    // No associated data
    Active(u32),             // Single u32 data
    Error { code: u16, desc: u16 }, // Struct-like variant with two u16 fields
}

fn main() {
    println!("Size of Status enum: {}", size_of::<Status>());
    // Expected: 8 bytes (4 bytes discriminant + 4 bytes largest variant)

    println!("Alignment of Status enum: {}", align_of::<Status>());
    // Expected: 4 bytes, matching the strictest field (u32)
}
```


# Naming Convention Compatibility

Rust has well-defined **naming conventions** to ensure code consistency and safety.

- Compiler generates **warnings** if Rust naming conventions are violated
- C/C++ do not enforce strict naming conventions
- Crate attributes are required to **suppress warnings** when interfacing with C/C++ code

```
#![allow(non_upper_case_globals)]
#![allow(non_camel_case_types)]
#![allow(non_snake_case)]
```

## Declaring External Functions in Rust

The `extern` keyword is used to declare functions that are implemented in other languages.

- Alternatively, use `extern "aapcs"` for ARM-specific calling conventions

```
extern "C" {
    fn SysTick_Handler();
}
```

- This assumes the existence of the C function with the following definition:

```
void SysTick_Handler(void) {/* some code */}
```

# Handling Scalar Data Types

The `core::ffi` module in Rust, provides various type definitions for C types.

| C type | core::ffi equivalent |
|---|---|
| char | c_char |
| signed char | c_schar |
| unsigned char | c_uchar |
| int | c_int |
| unsigned int | c_uint |
| short | c_short |
| unsigned short | c_ushort |

| C type | core::ffi equivalent |
|---|---|
| long | c_long |
| unsigned long | c_ulong |
| long long | c_longlong |
| unsigned long long | c_ulonglong |
| double | c_double |
| float | c_float |
| void pointer | c_void |

# Example of Scalar Types Usage

```
#![allow(non_upper_case_globals)]
#![allow(non_camel_case_types)]
#![allow(non_snake_case)]

use core::ffi::{c_char, c_int};

extern "C" {
    fn microbit_display_scroll(s: *const c_char);
    fn wait_ms(s: c_int);
}

fn main() {
    unsafe {
        // Calling the external C function within an unsafe block
        microbit_display_scroll("Hello Rust!\0".as_ptr());
        wait_ms(2000);
    }
}
```

# Direct and Indirect Linking

- Direct Linking: Ensures that your Rust application links directly to a specific C library using the `#[link]` attribute.
- Indirect Linking: Managed by build environments like Cargo, where linking is specified in a `build.rs` file or through Cargo configurations.

```
// Direct linking
#[link(name = "microbit_Screen")]

extern "C" {
    fn microbit_display_scroll(s: *const c_char);
    fn wait_ms(s: c_int);
}
```

# Working with C enum and struct types

1. Define the enum or stuct in C
2. Define the matching version in Rust with the `#[repr(C)]` attribute

```
enum IconNames {
    Heart = 0,
    SmallHeart = 1,
    Yes
};
```

```
#[repr(C)]
pub enum IconNames {
    Heart = 0,
    SmallHeart = 1,
    Yes
}

extern "C" {
    fn showIcon(icon: IconNames, interval: c_uint);
}
```

```
typedef struct TPoint { uint8_t x; uint8_t y; };
```

```
#[repr(C)]
pub struct TPoint {
    pub x: u8,
    pub y: u8,
}

extern "C" {
    // External C function that accepts a TPoint
    fn update_led_position(point: TPoint);
}

fn main() {
    let mut point = TPoint { x: 10, y: 20 };

    // C function that updates the LED's position
    unsafe {
        update_led_position(point);
    }
}
```


## Rust's Library Specification...

Rust functions are placed in the `lib.rs` file

- `#[no_mangle]`: Avoids the Rust compiler symbols' mangling
- `extern "C"`: Makes functions adhere to the C calling convention

```
#![no_std]

use core::panic::PanicInfo;

#[panic_handler]
fn handle_panic(_info: &PanicInfo) -> ! { loop {} }

#[no_mangle]
pub extern "C" fn rust_callee(a: i32, b: i32, c: i32, d: i32, e: i32, f: i32) -> i32 {
    a + b + c + d + e + f
}
```

## Rust's Library Specification

The compiler needs additional library definition directives inside the `Cargo.toml` file

- `crate-type`: Determines the type of library to generate, such as static or dynamic libraries, to match the linking requirements
  - Add to the Cargo.toml file the following directive:

```
[lib]
name = "rust_lib"
crate-type = ["staticlib"]  # static library (.a)
#            ["cdylib"]       dynamic libraby (.so)
```

## On the C Side

- Declare the Rust function in a C-compatible header file for proper linkage
- Use the `extern` keyword to reference Rust functions within C source files

```c
// rust_lib.h
int rust_callee(int a, int b, int c, int d, int e, int f);
```

```c
// main.c
extern int rust_callee(int a, int b, int c, int d, int e, int f);

int main(void) {
    rust_callee(1, 2, 3, 4, 5, 6);
    return 0;
}
```

## Compiling and Linking

Assuming the C file is place at the crate's root. We can compile is as shown:

- Linker Arguments `-l` and `-L`: Indicate the library name and the directory containing the Rust object files
- Setting `LD_LIBRARY_PATH`: Specifies the runtime library search path for dynamic libraries

```
gcc call_rust.c -o call_rust -lrust_lib -L./target/debug

LD_LIBRARY_PATH=./target/debug ./call_rust
```

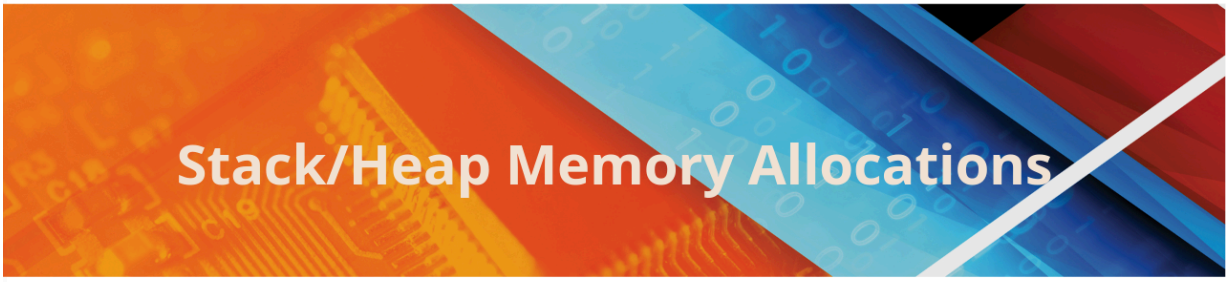
# Memory Management in no_std

Rust for firmware development

# Aims and Topics

In this module we will look at the various kinds of memory used by a embedded-Rust program and look at pragmatic options for using complex types.

- Stack and heap memory allocation
- Custom allocators for heap usage
- Alternative to using an allocator



## Why Should I Care About It?

- Efficient Resource Utilisation in Embedded Rust:
  - Limited memory in microcontrollers (MCUs)
  - Proper management prevents memory exhaustion
- Real-time Constraints:
  - Memory management consumes CPU time
- Safety and Security:
  - Prevents overflows and ensures proper memory access

# Memory in an Embedded Program

| | Content | Size | Lifetime | Cleanup |
|---|---|---|---|---|
| Stack ▼ | +Function arguments<br>+Local variables<br>+Known size at compile time | Dynamic / fixed upper limit | Lifetime of function | Automatic. When function returns |
| ▲ Heap | +Values that live beyond a function's lifetime<br>+Values access by multiple threads<br>+Large values<br>+Unknown size at compile time | Dynamic | Determined by programmer | Manual Malloc/ free MMU/MPU Operations |
| Static | +Program' binary<br>+Static variables<br>+String literals | Fixed | Lifetime of program | Automatic. When program terminates |

# Rust Use of Stack Memory Resources

- Efficient and quick memory for local variables:
  - Often cached in embedded systems
- Local variables in functions:
  - Local variables declared in a function are added to that function's stack frame
- Type restrictions:
  - Only values with a size known at compile time (e.g., `i8`, `u32`) can exist on the stack

# Rust Use of Heap Memory Resources

- Heap Resources:
  - Used for values with variable sizes at runtime (e.g., a vector of characters)
  - Elements such as characters can be added or removed during runtime

```
fn main() { //                ┌──── (HEAP)
  //                          ▼
    let a = String::from("Ferris");
} //   ▲
  //   │
  //   └──── Fat pointer [ptr, len, capacity]
  //           (STACK)
```


## Rationale for Custom Allocators

- Custom allocators can provide essential features in embedded systems:
  - Control over memory usage and allocation patterns
  - Real-time pedictability
  - Eliminates the need for an operating system
  - Targets specialised memory types
- Examples: `embedded_alloc`, `buddy_system_allocator`, `talc`, `alloc_no_std_lib`

## The `embedded-alloc` custom allocator

- MCU-agnostic heap allocator for embedded systems
  - Requires an implementation of the `critical-section` feature
- Linked-List First Fit (LLFF) heap allocator implementation:
  - Handles varying block sizes effectively
  - Performance may degrade with heavy heap usage

# Critical Section Implementations

- A critical section is a fundamental primitive for controlling concurrency
- Ensures mutually exclusive access to a resource
  - Can be acquired by only one thread at a time
- Applicable to various implementations (unicore, multicore, bare metal, RTOS...)

```
[dependencies]
cortex-m = { version = "0.7.7", features = ["critical-section-single-core"]}
embedded-alloc = "0.5.1"
```

# `embedded-alloc` Example

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

extern crate alloc;
use alloc::{boxed::Box, vec::Vec};
use cortex_m_rt::entry;
use stm32f1xx_hal;
use embedded_alloc::Heap;
use panic_halt as _;

#[global_allocator]
static HEAP: Heap = Heap::empty();

#[entry]
fn main() -> ! {
    {
        use core::mem::MaybeUninit;
        const HEAP_SIZE: usize = 1024; // 1KiB
        static mut HEAP_MEM: [MaybeUninit<u8>; HEAP_SIZE] = [MaybeUninit::uninit(); HEAP_SIZE];
        unsafe { HEAP.init(HEAP_MEM.as_ptr() as usize, HEAP_SIZE) }
    }

    let mut vec = Vec::new();
    vec.push(true);
    vec.push(false);

    let my_box = Box::new(42);

    #[allow(clippy::empty_loop)]
    loop {}
}
```

## The `heapless` Crate

- Provides fixed-capacity vectors and strings
- More versatile than `arrayVec`:
  - Implements a map and set based on a hash table
  - Includes more complex structures such as:
    - `Box`, `Arc`, `BinaryHeap`, queues, deques, maps, and more

# `heaples::Vec` Example

```
use heapless::Vec;

let mut my_vec: Vec<i32,16> = Vec::new();
let _ = my_vec.push(1963);
let _ = my_vec.push(9);

// Try to add elements to the my_vec
if my_vec.push(1986).is_err() {
    writeln!(serial, "Failed to add element to my_vec").unwrap();
}

// Access elements by index
if let Some(value) = my_vec.get(1) {
    writeln!(serial, "Value at index 1: {}", value).unwrap();
} else {
    writeln!(serial, "No value at index 1").unwrap();
}

// Iterate over elements
for value in &my_vec {
    writeln!(serial, "Iterating value: {}", value).unwrap();
}

// Pop an element
if let Some(value) = my_vec.pop() {
    writeln!(serial, "Popped value: {}", value).unwrap();
} else {
    writeln!(serial, "No value to pop").unwrap();
}
```

# `heapless::String` Example

```
use heapless::String;

let mut my_string: String<6> = String::new();
let _ = my_string.push_str("Ferris");
writeln!(serial, "{} quotes: i don't believe in isims.\r", my_string).unwrap();

let my_string: String<9> = String::<9>::from_str("Bueller's").unwrap();
writeln!(serial, "{} quotes: i don't believe in isims.\r", my_string).unwrap();

let mut subject = String::<9>::new(); // Create a new String with capacity 9
let _ = subject.push_str("Ferris"); // Push "Ferris" into the String

// Try to append " Bueller's" to the String
if subject.push_str(" Bueller's").is_ok() {
    writeln!(serial, "{} quotes: I don't believe in isms.\r", subject).unwrap();
} else {
    writeln!(serial, "String error").unwrap();
}
```

## Compromise Container Alternatives

- Store a certain number of elements inline
  - Fall back to the heap (entirely) for larger allocations
- Improves cache locality
- Reduces allocator traffic for small workloads
- `Smallvec` or `Tinyvec` crates
  - Requires an custom allocator to use the heap in no_std

```
use smallvec::SmallVec;

// spills into the heap over 4 elements
let mut v = SmallVec::<[u8; 4]>::new();
```

# Bringing a Baremetal System to Life

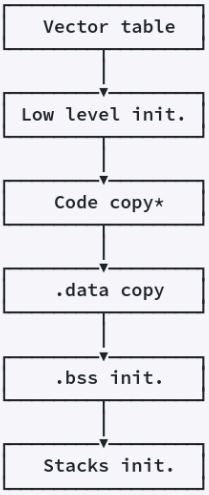
Rust for firmware development

## Aims and Topics

In this chapter we will look at the steps involved in starting a main function onto a deeply embedded device.

- Tailoring the image memory map to you target
- Rust library initialisation
- Streamlining bare-metal development
- Exception handlers

# General boot sequence overview

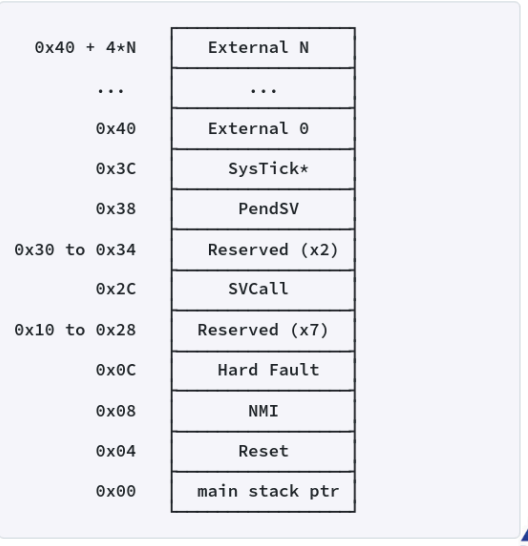



1. The vector table provide the reset handler's address
2. System devices' initialisation may be required
3. The code can be optionally moved to the RAM
4. Static data requires initialisation
5. Optional additional stacks can be configured

# The Vector Table: Cortex-M Example

Vector tables will **differ** from one processor's family to another.

- They contain either **jump** instructions or a **destination** address
  - Both internal and external exceptions are listed
- With Arm Cortex-M processors, it also contains the MSP address



# Writing an Exceptions' Table in Rust

The vector table entries can either be a pointer to an exception handler or a reserved placeholder.

```
pub union Vector {
    reserved: u32,
    handler: unsafe extern "C" fn(),
}

extern "C" {
    fn NMI();
    fn HardFault();
    fn SVCall();
    fn PendSV();
    fn SysTick();
}
```

```
#[link_section = ".vector_table.exceptions"]
#[no_mangle]
pub static EXCEPTIONS: [Vector; 14] = [
    Vector { handler: NMI },
    Vector { handler: HardFault },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { handler: SVCall },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { handler: PendSV },
    Vector { handler: SysTick },
];
```


# General Form of a Reset Handler Function

- The reset handler has some specific memory placement requirements
- We need to provide section attributes for the benefit of the linker

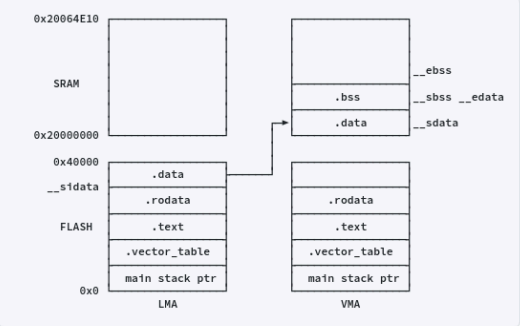
```
pub fn reset_handler() -> ! { // <-- returns the never type
  // handler's code here....
  loop {}
}

// The reset vector, a pointer into the reset handler
#[link_section = ".vector_table.reset_vector"]
#[no_mangle]
pub static RESET_VECTOR: unsafe extern "C" fn() -> ! = reset_handler;
```

# Providing Symbols for the Reset Handler

- The initialisation algorithm relies on the existence of address symbols provided by the linker script
  - Those symbols will be used to perform a move operation

```
static mut __sbss: u32; // Start of .bss section
static mut __ebss: u32; // End of .bss section
static mut __sdata: u32; // Start of .data section
static mut __edata: u32; // End of .data section
static __sidata: u32; // LMA of .data section
```



# Basic Reset Handler: Symbols Definition

- The placement symbols are defined by the linker script and defined as extern values

```
pub fn reset_handler() -> ! { // <-- returns the never type
    extern "C" {
        // These symbols come from `linker.ld`
        static mut __sbss: u32; // Start of .bss section
        static mut __ebss: u32; // End of .bss section
        static mut __sdata: u32; // Start of .data section
        static mut __edata: u32; // End of .data section
        static __sidata: u32; // LMA of .data section
    }

// More code...
  loop {}
}
```

# Basic Reset Handler: R/W Data Section Initialisation

- Data section containing global initialised values

```
pub fn reset_handler() -> ! {
    // linker placement symbols

    // Initialise Data
    unsafe {
        let mut sdata: *mut u32 = &mut __sdata;
        let edata: *mut u32 = &mut __edata;
        let mut sidata: *const u32 = &__sidata;

        while sdata < edata {
            write_volatile(sdata, read(sidata));
            sdata = sdata.offset(1);
            sidata = sidata.offset(1);
        }
    }

// More code...
```

# Basic Reset Handler: Zero Initialised Data Section

- BSS sections only require a zeroing out

```
pub fn reset_handler() -> ! {
    // linker placement symbols

    // Initialise Data

    // Initialise (Zero) BSS
    unsafe {
        let mut sbss: *mut u32 = &mut __sbss;
        let ebss: *mut u32 = &mut __ebss;

        while sbss < ebss {
            write_volatile(sbss, zeroed());
            sbss = sbss.offset(1);
        }
    }

    // Call user's main function
    main()
```

# Linker Script with Explicit Address Markers

```
SECTIONS
{
  /* MORE CODE */

  .text :
  {
    __stext = .;
    *(.text .text.*);
    . = ALIGN(4);
    __etext = .;
  } > FLASH

  .rodata :
  {
    __srodata = .;
    *(.rodata .rodata.*);
    . = ALIGN(4);
    __erodata = .;
  } > FLASH

  .data :
  {
    __sdata = .;
    *(.data .data.*);
    . = ALIGN(4);
    __edata = .;
  } RAM AT > FLASH

  /* LMA of .data */
  __sidata = LOADADDR(.data);

  /* .bss */
  .bss :
  {
    __sbss = .;
    *(.bss .bss.*);
    . = ALIGN(4);
    __ebss = .;
  } > RAM
}
```

## Using `#![no_std]` Crate Wide Attribute

- A crate wide attribute is used to disable the standard library
- Ensures compatibility with environments lacking an OS

```
#![no_std]

fn main() {
    // Bare metal code here
}
```

# The Smallest `#![no_std]` Program

- The `#![no_main]` attribute which means that the program won't use the standard main function as its entry point
  - This code produces an empty binary!

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

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_panic: &PanicInfo<'_>) -> ! {
    loop {}
}
```

## The Cortex-M Runtime (rt) Crate

- Contains all the required parts to build a `no_std` application
  - Vector table configuration
  - Static variables' initialisation
  - Enabling of the FPU
- Specialised attribute definition
  - `#[entry]` program's entry point definition
  - `#[exception]` overriding of existing exception handlers
  - `#[pre-init]` custom code to execute before variable initialisation

# Pragmatic, Minimal Bare-Metal Example

Pragmatically, we do not want to have to re-invent the wheel.

- "rt" core crates provide attributes and boiler-plate initialisation code to simplify the creation of an application's run-time

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

use panic_halt as _;
use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
    loop {}
}
```

## Project's Cargo File Configuration

- The `Cargo.toml` file will lists your project's dependencies
- Use crates.io to find a suitable crate

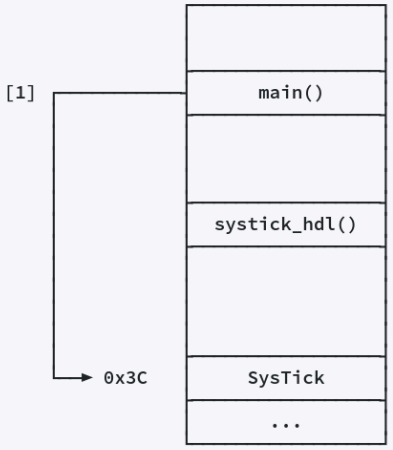
```
cortex-m = "0.6.0"
cortex-m-rt = "0.6.10"
stm32l4xx-hal = {version="*", features=["stm32l4x2", "rt"] }
```

# Interrupts on MCUs

- Interrupts are device specific
  - Number
  - Source
  - Priority
- Generic requirements:
  - Setup the peripheral for generating interrupts
  - Setup the priority level to enable pre-emption
  - Enable the source and the interrupt controller

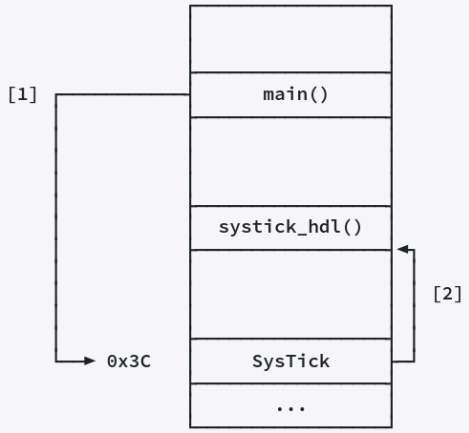
# SysTick Exception Example

- On an exception the core vectors into the relevant entry inside the vector table



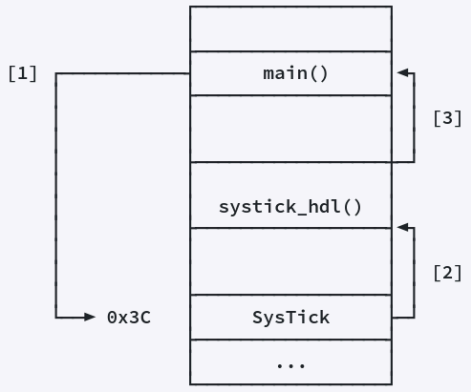
# Jumping to the Exception Handler

- The address held inside the vector table is loaded into the Program Counter



# Execution and Return

- The specific handler is executed and the processor returns the main application



## Handlers with the `cortex_m_rt` Crate

- Handlers are defined by overriding default implementations
  - Use the semantics: `#[exception] fn Name(..`
- Supported handlers' name:

```
DefaultHandler
NonMaskableInt
HardFault
MemoryManagement(*)
BusFault(*)
UsageFault(*)
SecureFault(*)
SVCall
DebugMonitor
PendSV
SysTick(**)
```

# Example: SysTick Handler in Rust

- Interrupts functions are identical to plain Rust functions
  - Return nothing and take no arguments

```
use cortex_m_rt::exception;

#[exception]
fn SysTick() {
    static mut COUNT: i32 = 0;

    // `COUNT` is safe to access and has type `&mut i32`
    *COUNT += 1;

    println!("{}", COUNT);
}
```