**Embedded Rust Quick Reference**

1. **Introduction: The compilation process** 
   1. **What is cross-compilation?**

Briefly, cross compilation enable us to compile and link programs for specific processor architectures and thus generating binaries with the instruction set that the remote processors can understand and execute.

The long answer: when you use your host computer to write computer program, build and run it on the same computer, it is called native compilation; your PC or laptop will act as both development and run environment for your computer program. So the compilation/linking process converts your program source code into executable machine code compatible to your host computer processor architecture.

Now say you want to develop programs which have to run on Embedded targets. These embedded targets are generally a microcontroller based hardware designed for a particular set of instructions called ISA(Instruction Set Architecture). The microcontroller/microprocessor architecture(E.g. ARM**x**, ARM cortex M-**x**, RISC V, Power Architecture) greatly varies from one to other based on the core and so does the ISA (E.G. thumbv6m-none-eabi -> Cortex-M.

thumbv7m-none-eabi -> Cortex-M3

armv7r-none-eabi -> little endian Cortex-R4

Cortex-R5ARMv6 -> ARM11 as found in the Raspberry Pi 1)

So now the computer programs you write and build on PC/laptop cannot directly run on Embedded target boards. Secondly these embedded target boards don't have any native build environment to build programs. So we use our PC/laptop, install a software which basically enables us to **write code using Host PC and compile it for embedded target boards**. These softwares are called **cross compiler, toolchain or cross compiler IDE**.

* 1. **What is LLVM?**

Like most compilers, rustc is composed of a "frontend" and a "backend". The "frontend" is responsible for taking raw source code, checking it for correctness, and getting it into a format X from which we can generate executable machine code. The "backend" then **takes that format X and produces** (possibly optimized) **executable machine** **code** for some platform.

Rustc's backend is LLVM, "a collection of modular and reusable compiler and toolchain technologies". In particular, the LLVM project contains a pluggable compiler backend (also called "LLVM"), which is used by many compiler projects, including the clang C compiler and our beloved rustc.

LLVM's "format X" is called LLVM IR, It is basically assembly code with additional low-level types and annotations added. These annotations are helpful for doing optimizations on the LLVM IR and **outputted machine code**. The end result of all this is (at long last) something executable (e.g. an ELF object or wasm).

There are a few benefits to using LLVM:

-We don't have to write a whole compiler backend.

-We benefit from the large suite of advanced optimizations that the LLVM project has been collecting.

-We can automatically compile Rust to any of the platforms for which LLVM has support.

* 1. **Does Rust support my device?**

It depends of 2 things:

1) Does the compiler support my device?

R: It depends on:

- Compilation **target** support: Rust support compiling your board´s ISA and architecture.

- **Architecture** support:: Rust support your board´s architecture and thus can generate LLVM code. However is possible that the ISA is not covered so manually handling is needed.

1. Does the crate ecosystem support my device?

R: Crate support means that exists libraries to facilitate handling of your board.

**2.0 Preparing cross compilation**

2.1 Tools

udev rules: These rules let you use USB devices like the F3 and the Serial module without root privilege.

Cargo-Binutils: Cargo subcommands to invoke the LLVM tools shipped with the Rust toolchain.

Itmdump: Tool to parse and dump [ITM](http://infocenter.arm.com/help/topic/com.arm.doc.ddi0314h/Chdbicbg.html) packets. ARM ITM (Instrumentation Trace Macroblock) allows tracing of software events, and also with the help of DWT (Debug, Watchpoint and Trace) the tracing of exceptions and data watchpoints. It also supports periodic sampling of PC values.

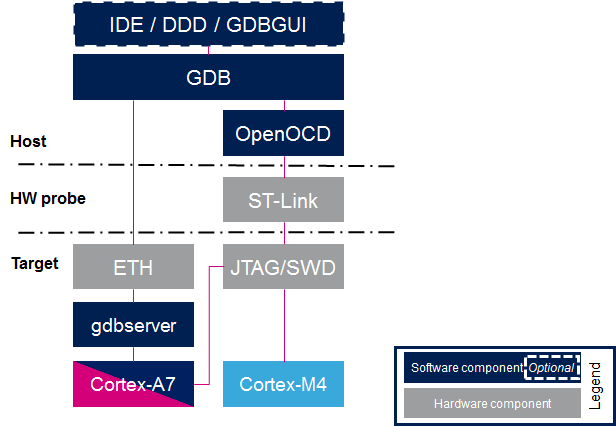
arm-none-eabi-gdb: GDB command you'll use to debug ARM EABI (bare-metal) targets.

OpenOCD: [Free software](https://en.wikipedia.org/wiki/Free_software" \o "Free software) [on-chip debugging](https://en.wikipedia.org/wiki/In-circuit_emulation" \l "On-chip_debugging" \o "In-circuit emulation), [in-system programming](https://en.wikipedia.org/wiki/In-system_programming" \o "In-system programming) and [boundary-scan](https://en.wikipedia.org/wiki/Boundary_scan" \o "Boundary scan)(Special IOs that allow test, override and observe the original signals produced by the board) testing tool for various [ARM](https://en.wikipedia.org/wiki/ARM_architecture" \o "ARM architecture), [MIPS](https://en.wikipedia.org/wiki/MIPS_architecture" \o "MIPS architecture) and [RISC-V](https://en.wikipedia.org/wiki/RISC-V" \o "RISC-V) systems.

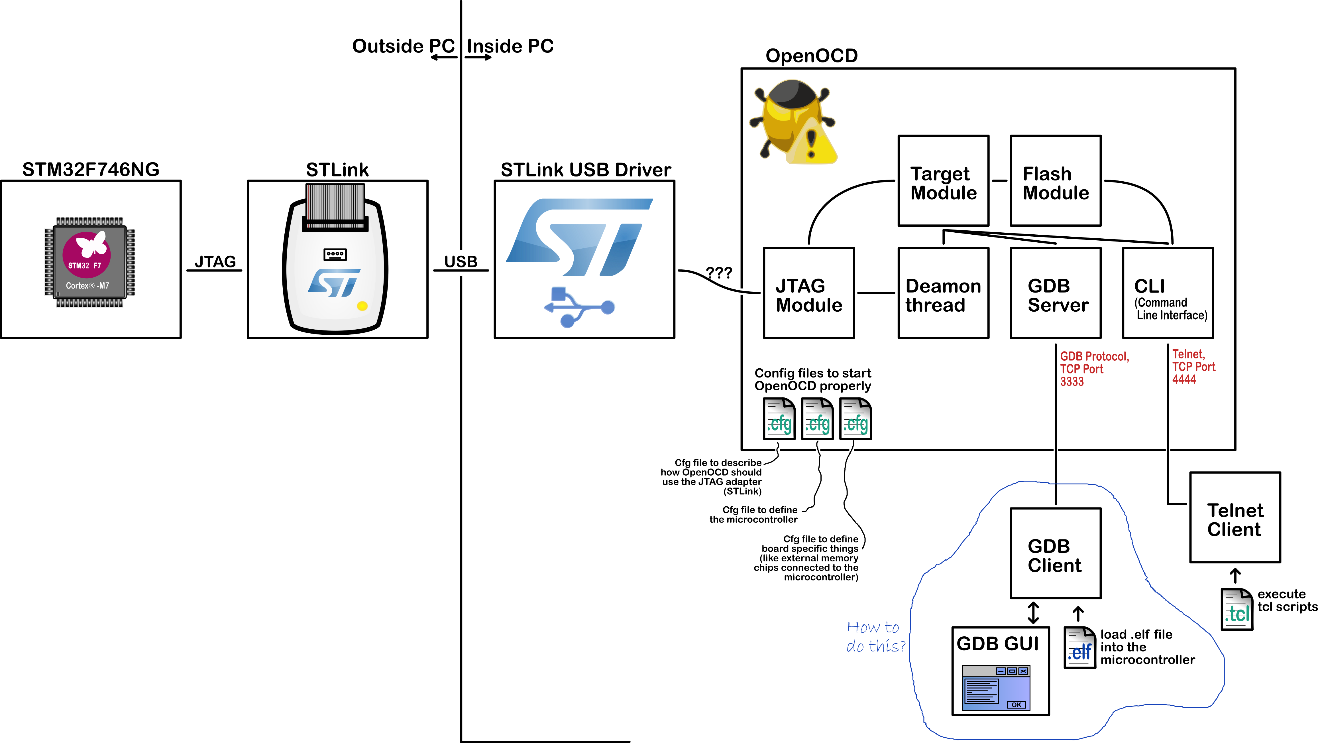
GDB: The GNU Project debugger (GDB), allows monitoring program execution, or what the program was doing at the moment it crashed.

JTAG: Originally designed to test assembled PCBs. Nowadays allow us to test, debug and program. Its connected to the boundary electronic allowing live data, regrettably it is serially connected (permits daisy chain on multiple devices) so many clock pulses are required.

2.2 Target Hardware connection



2.2.1 GDB path, either ethernet or ST-link



2.1.2 Debug architecture

**2. Getting started**

**2.1 Two general Embedded Programming classifications:**

Hosted Environments

These kinds of environments are close to a normal PC environment. What this means is that you are provided with a System Interface E.G. POSIX that provides you with primitives to interact with various systems, such as file systems, networking, memory management, threads, etc. Standard libraries in turn usually depend on these primitives to implement their functionality. You may also have some sort of sysroot and restrictions on RAM/ROM-usage, and perhaps some special HW or I/Os. Overall it feels like coding on a special-purpose PC environment.

Bare Metal Environments

In a bare metal environment no code has been loaded before your program. Without the software provided by an OS we can not load the standard library. Instead the program, along with the crates it uses, can only use the hardware (bare metal) to run. To prevent rust from loading the standard library use **no\_std**. The platform-agnostic parts of the standard library are available through libcore. libcore also excludes things which are not always desirable in an embedded environment. One of these things is a memory allocator for dynamic memory allocation. If you require this or any other functionalities there are often crates which provide these.

#![no\_std] is a crate-level attribute that indicates that the crate will link to the core-crate instead of the std-crate.

| **feature** | **no\_std** | **std** |
| --- | --- | --- |
| heap (dynamic memory) | \* | ✓ |
| collections (Vec, HashMap, etc) | \* | ✓ |
| stack overflow protection | ✘ | ✓ |
| runs init code before main | ✘ | ✓ |
| libstd available | ✘ | ✓ |
| libcore available | ✓ | ✓ |
| writing firmware, kernel, or bootloader code | ✓ | ✘ |

F2.1.1. Features available with no\_std vs std

**2.2 Tools**

QEMU: Generic and open source machine emulator and virtualizer. In this case we use the variant that can fully emulate ARM systems. We use QEMU to run embedded programs on the host.

GDB with ARM support: A debugger is a very important component of embedded development as you may not always have the luxury to log stuff to the host console.

OpenOCD: GDB isn't able to communicate directly with the ST-Link debugging hardware on your development board. It needs a translator and the Open On-Chip Debugger, OpenOCD, is that translator. OpenOCD is a program that runs on your laptop/PC and translates between GDB's TCP/IP based remote debug protocol and ST-Link's USB based protocol. It knows how to interact with the memory mapped registers used by the ARM **CoreSight** debug peripheral. It is these CoreSight registers that allow for:

* Breakpoint/Watchpoint manipulation
* Reading and writing of the CPU registers
* Detecting when the CPU has been halted for a debug event
* Continuing CPU execution after a debug event has been encountered

It also knows how to erase and write to the microcontroller's FLASH

cargo-binutils: collection of Cargo subcommands that make it easy to use the LLVM tools that are shipped with the Rust toolchain. These tools include the LLVM versions of objdump, nm and size and are used for inspecting binaries.

Cargo-generate or git: quickstart template. Bare metal programs are non-standard (no\_std), they require some adjustment in the linking process to declare the memory layout of the particular target; This requires some additional files (like linker) scripts and settings (like linker flags). cargo-generate creates project template to fill missing information.

**2.3 Minimal Program**

#![no\_std]#![no\_main]

extern crate panic\_halt;

use cortex\_m\_rt::entry;

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

loop {

// your code goes here

}}

#![no\_std] indicates that this program will not link to the standard crate, std. Instead it will link to its subset: the core crate.

#![no\_main] indicates that this program won't use the standard main interface that most Rust programs use.

extern crate panic\_halt: This crate provides a panic\_handler that defines the panicking behavior of the program.

#[entry] is an attribute provided by the **cortex-m-rt crate** that's used to mark the entry point of the program. As we are not using the standard main interface we need another way to indicate the entry point of the program and that'd be #[entry].

fn main() -> !. Our program will be the only process running on the target hardware so we don't want it to end! We use a divergent function (the -> ! bit in the function signature) to ensure at compile time that'll be the case.

**cortex-m-rt crate:** Startup code and minimal runtime for Cortex-M microcontrollers. It crates takes care of:

* The memory layout of the program. In particular, it populates the vector table so the device can boot correctly, and properly dispatch exceptions and interrupts.
* Initializing static variables before the program entry point.
* Enabling the FPU before the program entry point if the target is thumbv7em-none-eabihf.

**2.4 Inspect Program**

cargo-readobj can print the ELF headers to confirm that this is an ARM binary. E.g. cargo readobj --bin app -- -file-headers. Note that:

--bin {project} is sugar for inspect the binary at target/$TRIPLE/debug/{project}

--bin {project} will also (re)compile the binary, if necessary

Important: ELF files contain metadata like debug information so their size on disk does not accurately reflect the space the program will occupy when flashed on a device. Always use cargo-size to check how big a binary really is.

**cargo-size** can print the size of the linker sections of the binary.

$ cargo size --bin {project} --release -- -A

app :

section size addr

.vector\_table 1024 0x0

.text 92 0x400

.rodata 0 0x45c

.data 0 0x20000000

.bss 0 0x20000000

.debug\_str 2958 0x0

.debug\_loc 19 0x0

.debug\_abbrev 567 0x0

.debug\_info 4929 0x0

.debug\_ranges 40 0x0

.debug\_macinfo 1 0x0

.debug\_pubnames 2035 0x0

.debug\_pubtypes 1892 0x0

.ARM.attributes 46 0x0

.debug\_frame 100 0x0

.debug\_line 867 0x0

Total 14570

**A refresher on ELF linker sections**

.vector\_table is a non-standard section that we use to store the vector (interrupt) table

.text contains the program instructions

.rodata contains constant values like strings

.data contains statically allocated variables whose initial values are not zero

.bss also contains statically allocated variables whose initial values are zero

.ARM.attributes and the .debug\_\* sections contain metadata and will not be loaded onto the target when flashing the binary.

**Disasembly**

To generate the dissasembly, use: cargo objdump --bin app --release -- -disassemble -no-show-raw-insn -print-imm-hex