**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.

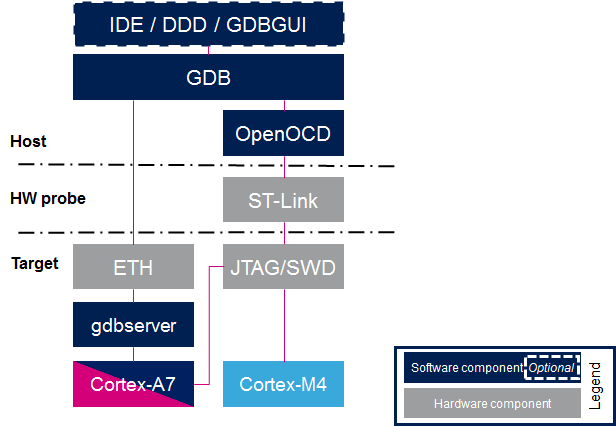
arm-none-eabi-gdb: GDB command you'll use to debug ARM EABI (bare-metal) targets.

OpenOCD: [Free software](https://en.wikipedia.org/wiki/Free_software) [on-chip debugging](https://en.wikipedia.org/wiki/In-circuit_emulation#On-chip_debugging), [in-system programming](https://en.wikipedia.org/wiki/In-system_programming) and [boundary-scan](https://en.wikipedia.org/wiki/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), [MIPS](https://en.wikipedia.org/wiki/MIPS_architecture) and [RISC-V](https://en.wikipedia.org/wiki/RISC-V) systems.

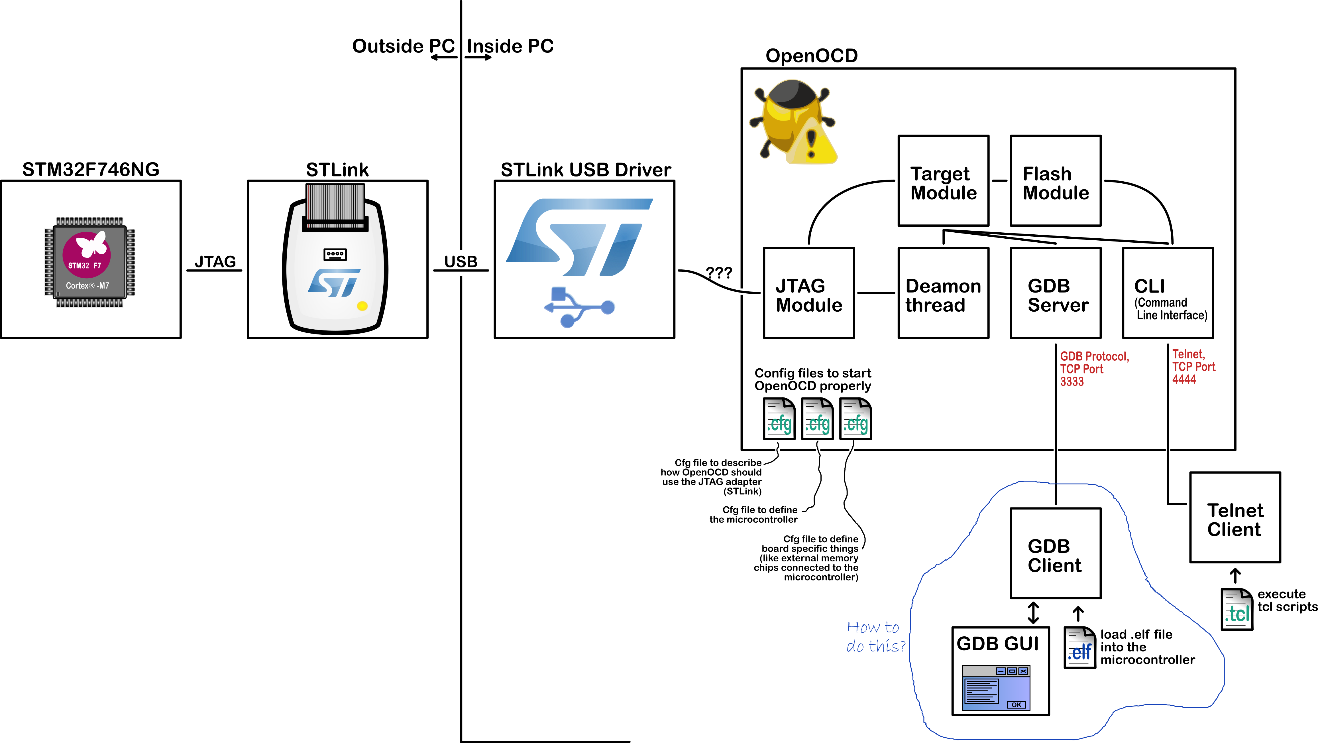
GDB: The GNU Project debugger (GDB), allows monitoring program execution, or what the program was doing at the moment it crashed.

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