010123131

Software Development Practice

Handout #5

<rawat.s@eng.kmutnb.ac.th>

Last Update: 2024-07-22

Agenda

- C/C++ Cross Compilation & Toolchains for Linux
- GNU Toolchains for Embedded Processors
- C/C++ Cross-Compilation for Raspberry Pi SBC
- C/C++ Cross-Compilation for Arduino MCU Boards
- Introduction to Containerization with Docker
- Arduino CLI for Ubuntu Linux
- Python 3 Virtual Environment

Cross Compilation

- **Cross-compilation of C/C++ source code files** is the process of compiling code on one platform or architecture to produce executable files that can run on a different platform or architecture.
- This is often done when developing software for embedded systems, mobile devices, or other platforms where the target architecture is different from the development machine.

Cross Compilation

- **Cross-compilation for C/C++ source code** involves using a **cross-compiler** runs on one platform but generates code for a different platform.
  - Examples of CPU architectures: `x86`, `x86_64`, `amd64`, `arm64` (`aarch64`), `armhf`, `riscv64`, ...
- The **C/C++ cross-compiler** typically requires a set of header files and libraries for the target platform, which are often provided by a **toolchain for the target platform**.

Cross-Compilation Toolchains

- A **toolchain** is a set of distinct software development tools that are linked (or chained) together by specific stages such as **GCC**, **binutils** and **glibc** (standard C library) and **gdb** debugger (a portion of the **GNU Toolchain**).
- All the programs (like GCC) run on a host system of a specific architecture (such as x86), but they produce binary code to run on a different architecture (for example, ARM).

Reference: https://elinux.org/Toolchains

Cross-Compilation Toolchains

- A **bare-metal cross-compilation toolchain** is designed to generate code that runs directly on hardware, without the need for an operating system or other software layer to provide services or abstractions.
- This type of toolchain is commonly used for **embedded systems**, **microcontrollers (MCUs)**, and other low-level hardware applications.

Cross-Compilation Toolchains

- On the other hand, a **Linux-based cross-compilation toolchain** is designed to generate code that runs on top of a **Linux OS**.
- This type of toolchain is used for a wide range of applications, including desktop and server software, device drivers, and embedded systems that run a Linux-based OS.

Cross-Compilation Toolchains

- A **bare-metal cross-compilation toolchain** will typically provide only the most basic C and C++ libraries and headers.
- A **Linux-based cross-compilation toolchain** will include a wide range of libraries and headers for use with the Linux OS.

Cross-Compilation Toolchains

- A **bare-metal cross-compilation toolchain** may provide options for specifying the target hardware architecture, memory layout, and other low-level details.
- A **Linux-based cross-compilation toolchain** may provide options for specifying the Linux kernel version, target filesystem layout, and other higher-level details.

Cross-Compilation Toolchains

- **Compatibility issues** can occur due to different versions of **GLIBC dynamic library** on the target machine.
- Possible solutions:
  1) Statically link the binary file.
  2) Update the **GLIBC library** on the target machine if possible.
  3) Use a **Docker container** or **virtual machine**.

GLIBC

- **GLIBC (GNU C Library)** is the **C standard library** for the GNU system, maintained by the **GNU ("Nu") Project**.
- It is a collection of C programming language library functions that provide the essential low-level functionality required by most programs written in C.

GLIBC

- **GLIBC** is an important component of most Linux-based systems, and it is used by many open-source software projects.
- It provides a standardized interface between the Linux OS services and user-space applications, which makes it possible to write portable applications that can run on different Linux-based systems without modification.

Cross-Compilation for RPi

- Steps for cross-compiling C files on Linux / Ubuntu (`x86_64`) to target the Raspberry Pi (RPi) running Raspbian or Ubuntu OS (for `aarch64`).

```
# Install the cross-compiler and necessary libraries:
$ sudo apt install gcc-aarch64-linux-gnu libc6-dev-arm64-cross
# Write C/C++ code and compile it using the cross-compiler.
$ aarch64-linux-gnu-gcc -Wall -static main.c -o ./hello_pi
```

Use the `scp` command to copy the file (`./hello_pi`) from your local computer to the remote Raspberry Pi and try to run the program remotely using SSH.

Source Code File: `main.c`

```c
#include <stdio.h>
#include <time.h>

int main(int argc, char *argv[]) {
  time_t current_time;
  struct tm *local_time;

  // Get current time
  current_time = time(NULL);

  // Convert current time to local time
  local_time = localtime(&current_time);

  printf( "Hello on Raspberry Pi!\n" );
  printf( "Current date and time: %s", asctime(local_time) );
  return 0;
}
```

Cross Compilation with Static Linking for aarch64

```
$ aarch64-linux-gnu-gcc -Wall -static main.c -o hello_pi
```

```
$ ./hello_pi
-bash: ./hello_pi: cannot execute binary file: Exec format error
```

```
$ file ./hello_pi | tr ',' '\n' | sed '/^\s*$/d'

 ./hello_pi: ELF 64-bit LSB executable
 ARM aarch64
 version 1 (GNU/Linux)
 statically linked
 BuildID[sha1]=ba49af91da6baa790db1e25a711c9cd501ed3845
 for GNU/Linux 3.7.0
 not stripped
```

Local machine: x86_64, Windows 10, WSL2 / Ubuntu 22.04 LTS

```
# Check the CPU architecture of the local machine.
$ uname -m
x86_64

$ ldd --version | head -n 1
ldd (Ubuntu GLIBC 2.35-0ubuntu3.1) 2.35

$ cat /etc/os-release | head -n 5
PRETTY_NAME="Ubuntu 22.04.2 LTS"
NAME="Ubuntu"
VERSION_ID="22.04"
VERSION="22.04.2 LTS (Jammy Jellyfish)"
VERSION_CODENAME=jammy
```

Remote Machine: Raspberry Pi 4, aarch64, Raspbian OS (Debian 11, Bulleye)

```
# Check the CPU architecture of the remote machine.
$ ssh pi@raspberrypi 'uname -m'
aarch64

$ ssh pi@raspberrypi 'ldd --version | head -n 1'
ldd (Debian GLIBC 2.31-13+rpt2+rpi1+deb11u5) 2.31

$ ssh pi@raspberrypi 'cat /etc/os-release | head -n 5'
PRETTY_NAME="Debian GNU/Linux 11 (bullseye)"
NAME="Debian GNU/Linux"
VERSION_ID="11"
VERSION="11 (bullseye)"
VERSION_CODENAME=bullseye
```

Note: The versions of GLIBC on Ubuntu (amd64) and on Raspberry Pi (aarch64) are different. The program that was cross-compiled and linked dynamically may not work.

On Ubuntu x86_64 / amd64: Remove the cross-compiler for ARM64

```
# Show the host architecture
$ dpkg --print-architecture

# Show the foreign architectures (if added)
$ dpkg --print-foreign-architectures
```

```
# Remove the GNU C cross-compiler for the arm64
$ sudo apt remove --purge \
     gcc-aarch64-linux-gnu libc6-dev-arm64-cross
$ sudo dpkg --remove-architecture arm64
$ sudo apt autoremove && sudo apt autoclean
```

Pre-built Cross-Compiler (ARM GNU Toolchain) for Raspberry Pi OS

1) Go to: https://sourceforge.net/projects/raspberry-pi-cross-compilers/

2) Download files: “Bonus Raspberry Pi GCC 64-Bit Toolchains”.

- File: `cross-gcc-10.3.0-pi_3+.tar.gz` (for armhf / 32-bit)
- File: `cross-gcc-10.3.0-pi_64.tar.gz` (for arm64 / 64-bit)

3) Unpack the cross-compiler toolchain archive file for ARM64 on Linux Ubuntu.

```
# Extract files from the archive into a new directory.
$ tar xvfz "cross-gcc-10.3.0-pi_64.tar.gz"
$ export PATH="$PWD/cross-pi-gcc-10.3.0-64/bin:$PATH"
# Show the cross-compiler version.
$ aarch64-linux-gnu-gcc --version | head -n 1
aarch64-linux-gnu-gcc (GCC) 10.3.0
```

Pre-built Cross-Compiler (ARM GNU Toolchain) for Raspberry Pi OS

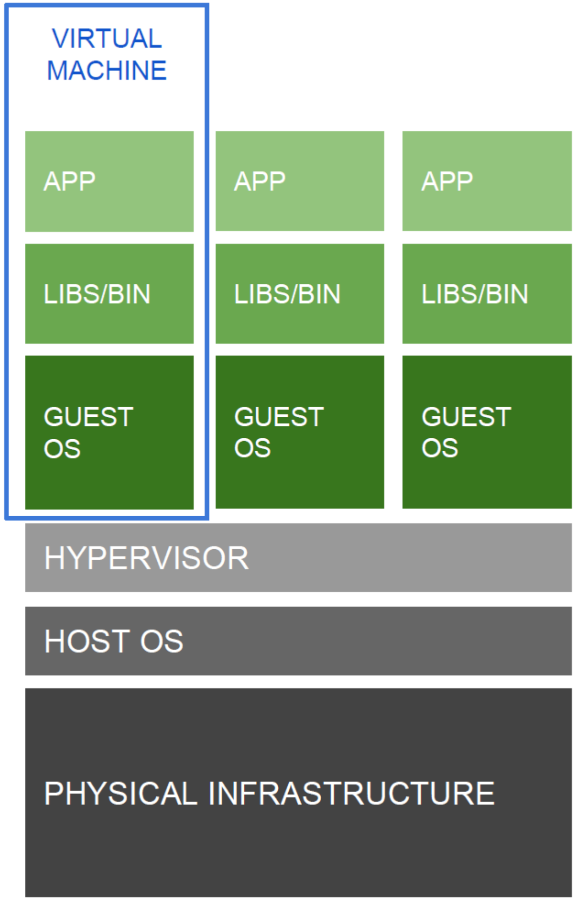
```
# Compile the source file (main.c).
$ aarch64-linux-gnu-gcc -Wall main.c -o hello_pi

# Show info about the executable file.
$ file ./hello_pi | tr ',' '\n' | sed '/^\s*$/d'
 ./hello_pi: ELF 64-bit LSB executable
 ARM aarch64
 version 1 (SYSV)
 dynamically linked
 interpreter /lib/ld-linux-aarch64.so.1
 for GNU/Linux 3.7.0
 with debug_info
 not stripped
```

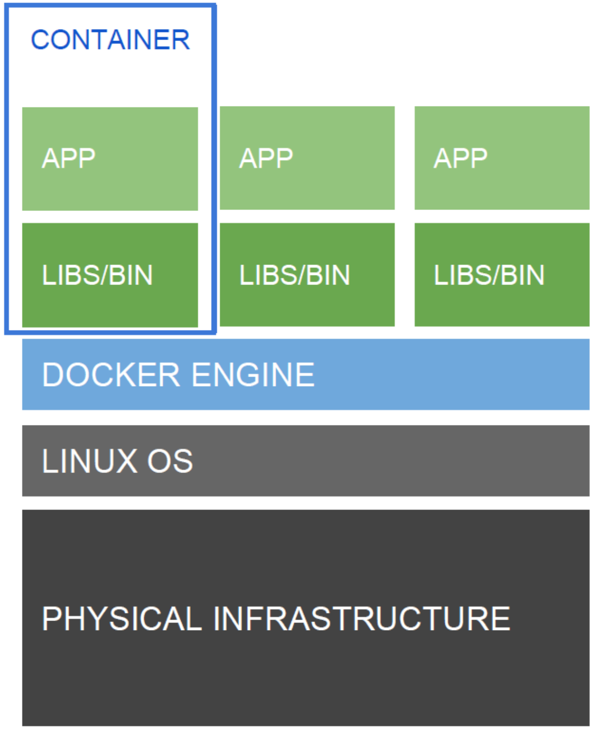
Use the `scp` command to copy the file (`./hello_pi`) from Ubuntu computer to the remote Raspberry Pi and run the program remotely using SSH.

Containerization

- **Containerization** is a method of packaging software and its dependencies into a standardized unit called a **container**.
- Containers based on containerization technologies such as **Docker** or **Kubernetes** provide a lightweight and isolated environment for running applications, making it easier to deploy and manage software across different systems.

Virtualization vs Containerization





Docker

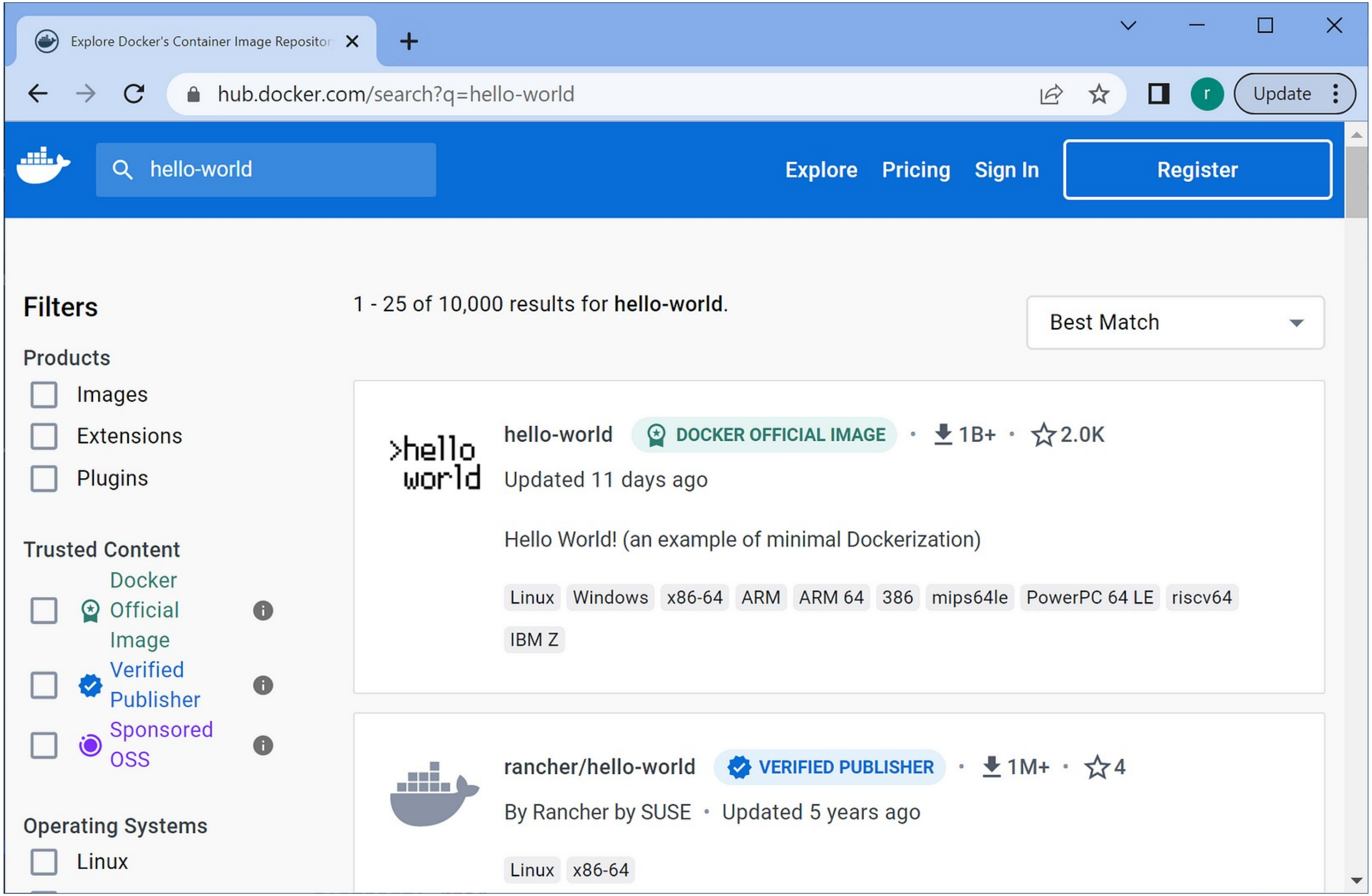
- **Docker** is an open-source platform used to create, run and deploy applications in **containers**.
- **Docker** is primarily implemented using **Golang** (**Go programming language**) developed by Google.
- Docker uses a **client-server architecture**, where the **Docker client** interacts with the **Docker daemon**, which is responsible for building, running, and distributing containers.

Key Components of Docker

- **Docker images**: They are read-only templates that contain everything needed to run an application.
- **Docker containers**: They are instances created from Docker images. They are isolated environments that encapsulate the application and its dependencies, providing consistency and reproducibility.
- **Docker Compose**: It is a tool that simplifies the management of **multi-container Docker applications**.

Key Components of Docker

- **Dockerfiles**: They are **Docker Compose files** used to create or build Docker images.
- **Docker Hub**: It is a cloud-based registry that hosts Docker images and serves as a central repository for sharing and discovering **pre-built Docker images**.


Explore Docker's Container Image Repositor
hub.docker.com/search?q=hello-world
Update
hello-world
Explore
Pricing
Sign In
Register
Filters
Products
Images
Extensions
Plugins
Trusted Content
Docker Official Image
Verified Publisher
Sponsored OSS
Operating Systems
Linux
1 - 25 of 10,000 results for hello-world.
Best Match
>hello world
hello-world
DOCKER OFFICIAL IMAGE
1B+
2.0K
Updated 11 days ago
Hello World! (an example of minimal Dockerization)
Linux
Windows
x86-64
ARM
ARM 64
386
mips64le
PowerPC 64 LE
riscv64
IBM Z
rancher/hello-world
VERIFIED PUBLISHER
1M+
4
By Rancher by SUSE
Updated 5 years ago
Linux
x86-64

Docker Images

- A **Docker image** is a standalone, and executable software package that includes everything needed to run an application, including the code, runtime, libraries, and system tools.
- When a **Docker image** is run, it creates an instance called a **Docker container**.
- Each **container** is isolated and independent, running in its own environment with its own filesystem, network, and resources.
- Multiple containers can be created from the same image.

Docker Images

- **Docker images** are dependent on a specific operating system or, more precisely, a specific **base image**.
- The **base image** serves as the foundation for the Docker image and includes the underlying OS and a minimal set of packages required for running applications.

Docker Installation on Ubuntu

```
# Install Docker on Raspbian OS 64-bit or WSL2 Ubuntu
# see: https://docs.docker.com/engine/install/debian/

$ sudo apt update && sudo apt upgrade -y
$ sudo apt install -y apt-transport-https ca-certificates \
  curl gnupg software-properties-common

# Download the Docker installation script.
$ curl -fsSL https://get.docker.com -o get-docker.sh

# Run the script with the help of the below command:
$ sudo sh get-docker.sh && rm -f get-docker.sh
```

Note: For Windows users, Microsoft recommends to use **Docker Desktop for Windows**.
https://www.docker.com/products/docker-desktop/

Docker Installation on Ubuntu

```
# Start the docker service manually.
$ sudo service docker start
$ sudo service docker status
$ sudo usermod -aG docker $USER
$ sudo docker version

# Add the current user to the docker group.
$ sudo usermod -aG docker $USER

## Logout and login again

# Show the Docker version and some info.
$ docker version
$ docker info
# Run Docker an official image in a container: hello-world.
$ docker run hello-world
```

ubuntu@ubuntu-desktop-vm: ~

```
ubuntu@ubuntu-desktop-vm:~$  sudo apt install -y apt-transport-https ca-certificates \
>   curl gnupg software-properties-common
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
ca-certificates is already the newest version (20211016ubuntu0.22.04.1).
curl is already the newest version (7.81.0-1ubuntu1.10).
gnupg is already the newest version (2.2.27-3ubuntu2.1).
software-properties-common is already the newest version (0.99.22.6).
apt-transport-https is already the newest version (2.4.9).
0 upgraded, 0 newly installed, 0 to remove and 10 not upgraded.
ubuntu@ubuntu-desktop-vm:~$ curl -fsSL https://get.docker.com -o get-docker.sh
ubuntu@ubuntu-desktop-vm:~$ sudo sh get-docker.sh && rm -f get-docker.sh
# Executing docker install script, commit: c2de0811708b6d9015ed1a2c80f02c9b70c8ce7b
+ sh -c apt-get update -qq >/dev/null
+ sh -c DEBIAN_FRONTEND=noninteractive apt-get install -y -qq apt-transport-https ca-certificates curl >/dev/null
+ sh -c install -m 0755 -d /etc/apt/keyrings
+ sh -c curl -fsSL "https://download.docker.com/linux/ubuntu/gpg" | gpg --dearmor --yes -o /etc/apt/keyrings/docker.gpg
+ sh -c chmod a+r /etc/apt/keyrings/docker.gpg
+ sh -c echo "deb [arch=amd64 signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu jammy stable" >
 /etc/apt/sources.list.d/docker.list
+ sh -c apt-get update -qq >/dev/null
+ sh -c DEBIAN_FRONTEND=noninteractive apt-get install -y -qq docker-ce docker-ce-cli containerd.io docker-compose-plugin dock
er-ce-rootless-extras docker-buildx-plugin >/dev/null
```

Select ubuntu@ubuntu-desktop-vm: ~

```
+ sh -c docker version
Client: Docker Engine - Community
 Version:           24.0.1
 API version:       1.43
 Go version:        go1.20.4
 Git commit:        6802122
 Built:             Fri May 19 18:06:21 2023
 OS/Arch:           linux/amd64
 Context:           default

Server: Docker Engine - Community
 Engine:
  Version:          24.0.1
  API version:      1.43 (minimum version 1.12)
  Go version:       go1.20.4
  Git commit:       463850e
  Built:            Fri May 19 18:06:21 2023
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.6.21
  GitCommit:        3dce8eb055cbb6872793272b4f20ed16117344f8
 runc:
  Version:          1.1.7
  GitCommit:        v1.1.7-0-g860f061
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
```

Sample Dockerfile

Specify the base image

Set an environment variable

Run Linux commands

Set the working directory

```
FROM debian:bullseye

ENV LANG='C.UTF-8' LC_ALL='C.UTF-8'

# Install packages
RUN apt-get update && apt-get install -y \
        gcc-aarch64-linux-gnu g++-aarch64-linux-gnu \
        libc6-dev-arm64-cross

RUN dpkg --add-architecture arm64

WORKDIR /build
```

Using the GNU C/C++ ARM Cross-Compilation Toolchain inside a Docker container (with Debian Bullseye as base image).

Build Docker Image

```
# Build a Docker image (named “aarch64-toolchain”)
# from the Dockerfile in the current directory.
$ docker buildx build -t aarch64-toolchain ./

## Put the main.c file in the $HOME/ARM64 folder.

# Run the cross-compiler in a Docker container.
$ docker run -v $HOME/ARM64:/build aarch64-toolchain \
  aarch64-linux-gnu-gcc --version | head -n 1
aarch64-linux-gnu-gcc (Debian 10.2.1-6) 10.2.1 20210110

# Compile the C source code (main.c in $HOME/ARM64/) for RPi (64-bit).
$ docker run -v $HOME/ARM64:/build aarch64-toolchain \
  aarch64-linux-gnu-gcc -Wall -Os -lm main.c -o hello_pi
# List all local Docker images
$ docker images -a
REPOSITORY          TAG       IMAGE ID       CREATED              SIZE
aarch64-toolchain   latest    e366779dc947   About a minute ago   918MB
hello-world         latest    9c7a54a9a43c   2 weeks ago          13.3kB
```

Some Docker Commands

```
# List all local Docker images by IDs.
$ docker images -aq
# List the repository names and tags of local Docker images.
$ docker images --format "{{.Repository}}:{{.Tag}}"

# Shows the history of an Docker image.
$ docker history <image-repository:tag>
$ docker history hello-world:latest

# List all local Docker containers.
$ docker ps -a

# Stop and remove all local containers.
$ docker stop $(docker ps -aq)
$ docker rm $(docker ps -aq)

# Delete all local Docker images.
$ docker rmi $(docker images -aq)
```

Docker Removal

```
# Uninstall Docker

$ sudo systemctl stop docker
$ sudo systemctl disable docker
$ sudo groupdel docker

$ sudo apt-get purge docker-ce docker-ce-cli containerd.io \
  docker-buildx-plugin docker-compose-plugin docker-ce-rootless-extras

$ sudo rm -fr /var/lib/docker/
$ sudo rm -rf /var/lib/containerd/
```

Arduino Boards

- **Arduino microcontroller boards** are programmable circuit boards, which are designed for prototyping and creating interactive electronic projects.
- Arduino boards can be programmed using the Arduino programming language, which is a simplified version of C/C++.
- The **Arduino IDE** (Integrated Development Environment) provides a user-friendly interface for compiling, and uploading code to the Arduino board.

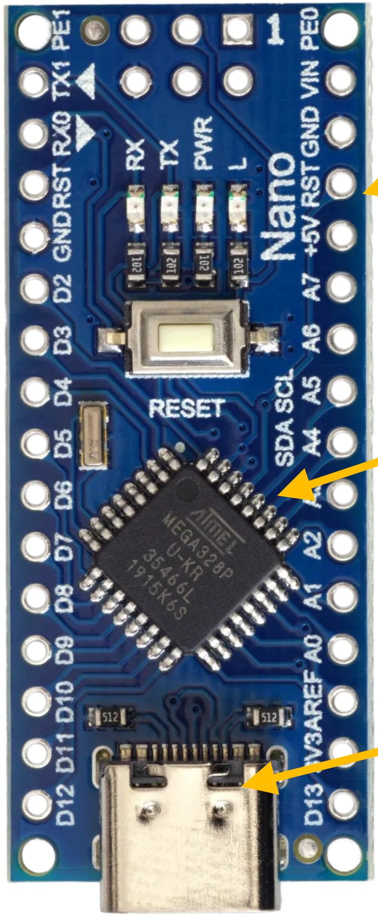
Arduino Boards

- Arduino boards come in different models, with different **microcontrollers** (MCUs) and hardware features.
  - **8-bit MCUs**: Arduino Uno, Arduino Nano, Arduino Mega, Arduino Leonardo, Arduino Nano Every
  - **32-bit MCUs**: Arduino DUE, Arduino MKR Series, Arduino Nano 33 BLE, ...

Arduino Boards

- The microcontroller on an Arduino-compatible board is pre-installed with an **Arduino bootloader**.
- The **Arduino bootloader** is a small program or piece of firmware that runs on the microcontroller of an Arduino board.
- It enables easy uploading of sketches (programs) to the Arduino board without the need for external programming hardware.

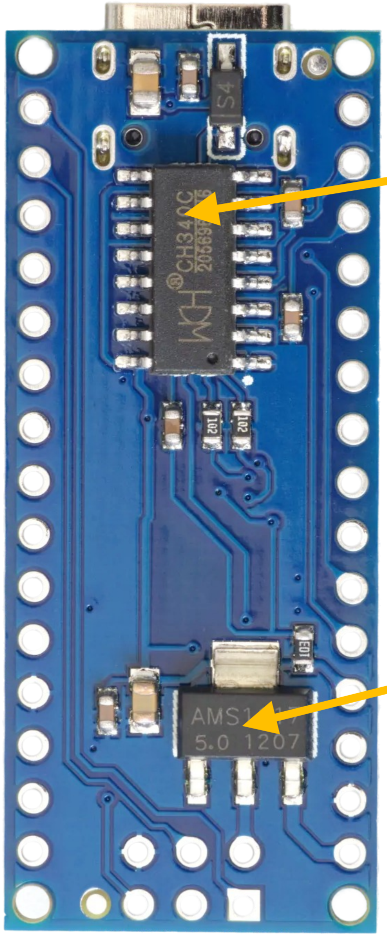
Arduino Nano (Clone)



Through Holes
for soldering pin
headers
(2.54mm spacing)

ATmega328P MCU
(5V, 16MHz),
pre-installed
with an Arduino
bootloader

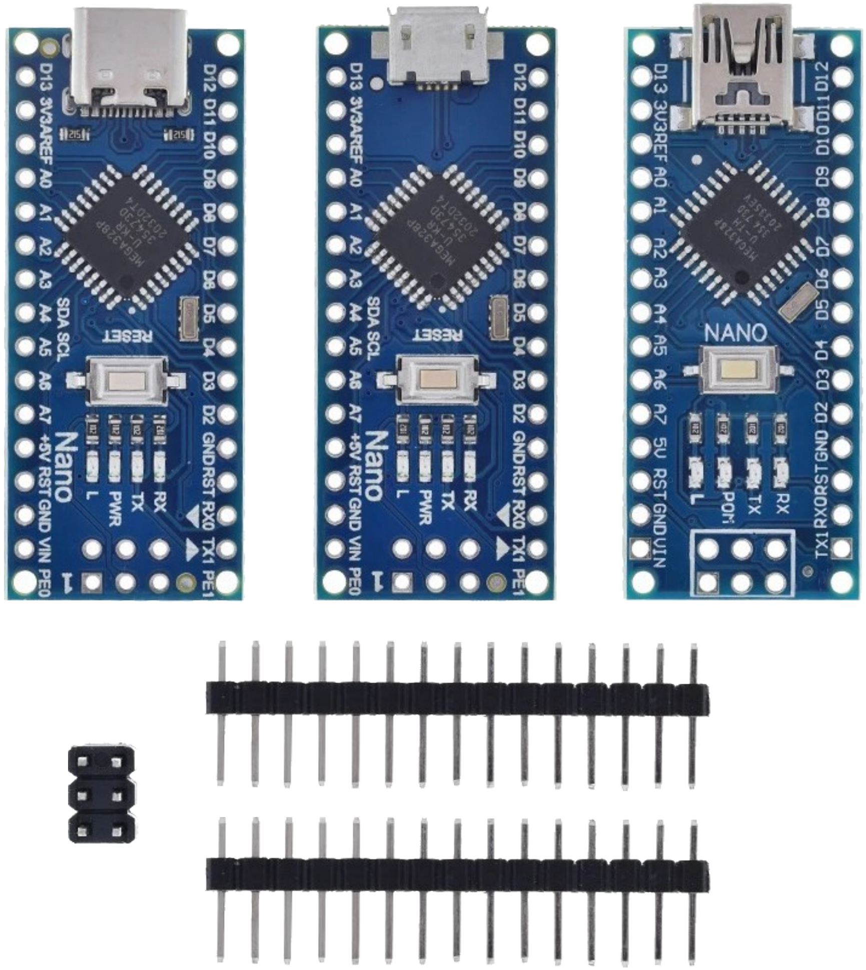
USB Type-C
Port

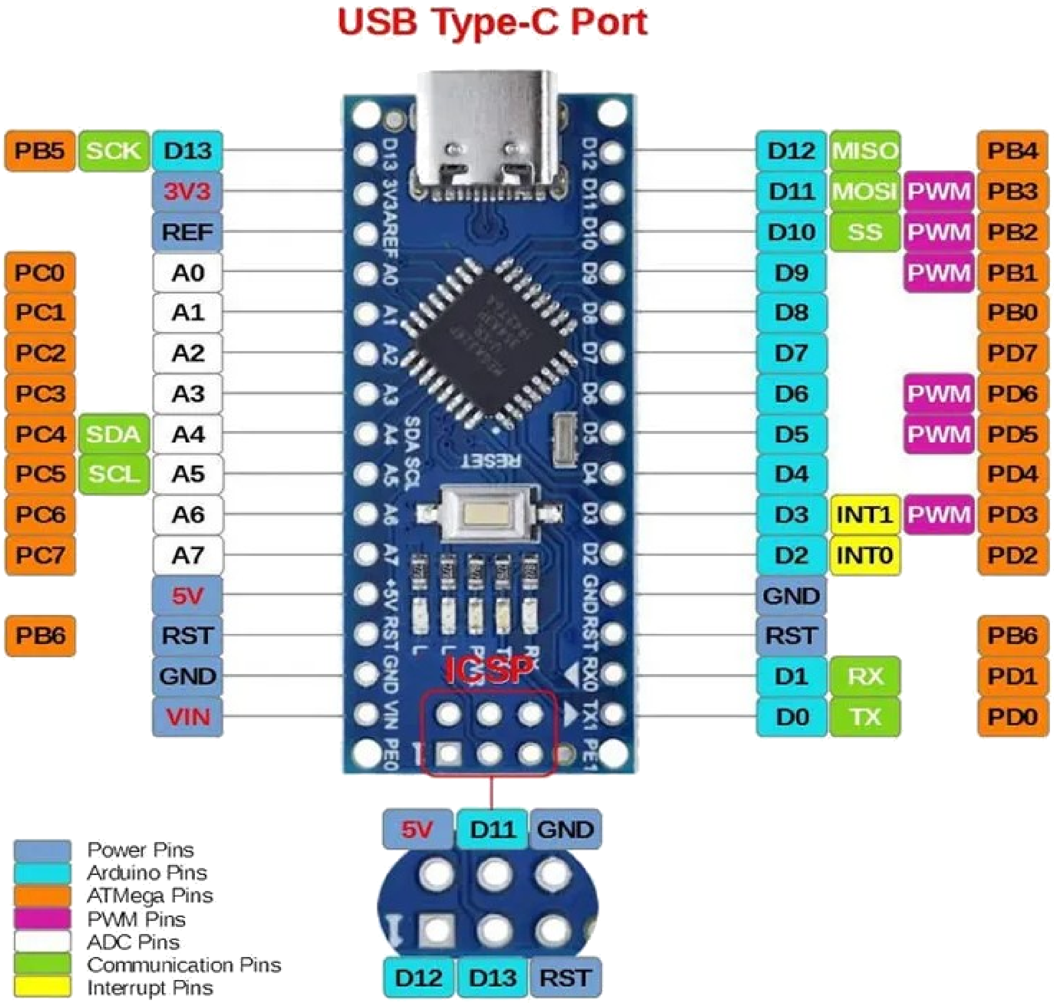


CH340
USB-to-Serial
Chip

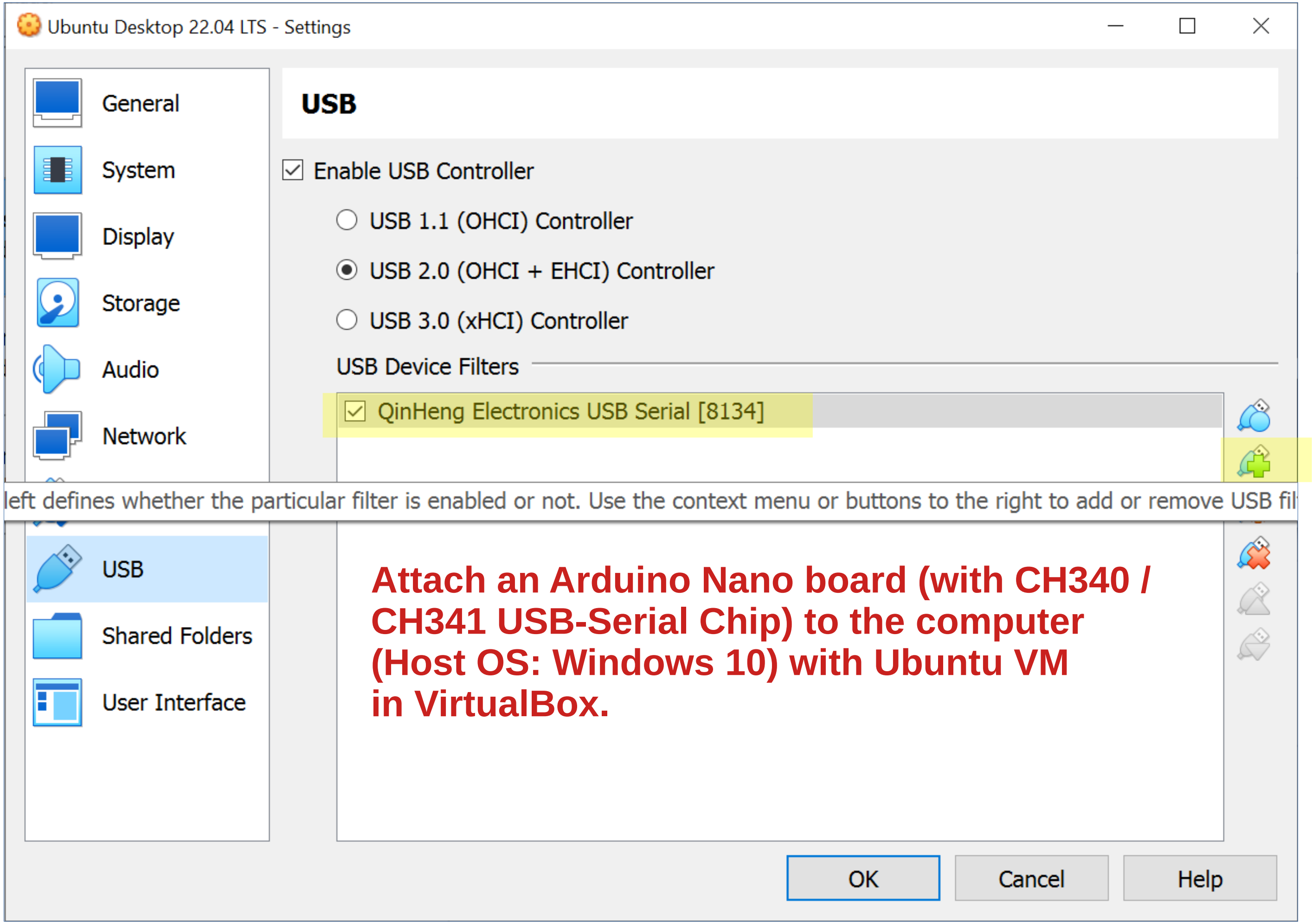
AMS117 5V
Voltage
Regulator

Arduino Nano (Clone)





Add an USB device so that it can be accessed by the Ubuntu VM in VirtualBox.



Attach an Arduino Nano board (with CH340 / CH341 USB-Serial Chip) to the computer (Host OS: Windows 10) with Ubuntu VM in VirtualBox.

List USB devices on Ubuntu Desktop 22.04 LTS (Virtual Machine)

ubuntu@ubuntu-desktop-vm: ~

```
ubuntu@ubuntu-desktop-vm:~$
ubuntu@ubuntu-desktop-vm:~$ lsusb
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 003: ID 1a86:7523 QinHeng Electronics CH340 serial converter
Bus 002 Device 002: ID 80ee:0021 VirtualBox USB Tablet
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
ubuntu@ubuntu-desktop-vm:~$ sudo dmesg | grep usb | tail -n 6
[    4.710408] usbcore: registered new interface driver usbserial_generic
[    4.710417] usbserial: USB Serial support registered for generic
[    4.820582] usbcore: registered new interface driver ch341
[    4.820591] usbserial: USB Serial support registered for ch341-uart
[    4.855623] usb 2-2: ch341-uart converter now attached to ttyUSB0
[    5.384056] usb 2-2: usbfs: interface 0 claimed by ch341 while 'brltty' sets config #1
ubuntu@ubuntu-desktop-vm:~$
```

Remove brltty and ModemManager (not used).

```
# Check whether the brltty service is enabled.
$ systemctl list-units | grep brltty

# Remove brltty (braille display driver) and ModemManager.
$ sudo apt-get --purge remove brltty modemmanager
$ sudo apt autoremove && sudo apt autoclean
$ sudo reboot
```

```
# Add the current user to the dialout group.
$ sudo usermod -aG dialout $USER
# Start a new shell with same user.
$ exec su -l $USER
# Show the user name and the user's groups.
$ whoami && groups
```

Show the USB device associated with the Arduino Nano board.

```
ubuntu@ubuntu-desktop-vm: ~
ubuntu@ubuntu-desktop-vm:~$
ubuntu@ubuntu-desktop-vm:~$ sudo dmesg | grep usb | tail -n 6
[    2.610939] hid-generic 0003:80EE:0021.0001: input,hidraw0: USB HID v1.10 Mouse [VirtualBox USB Tablet] on usb-0000:00:06.0-1/input0
[    5.262099] usbcore: registered new interface driver usbserial_generic
[    5.262112] usbserial: USB Serial support registered for generic
[    5.279165] usbcore: registered new interface driver ch341
[    5.279180] usbserial: USB Serial support registered for ch341-uart
[    5.373482] usb 2-2: ch341-uart converter now attached to ttyUSB0
ubuntu@ubuntu-desktop-vm:~$ ls -la /dev/ttyUSB*
crw-rw---- 1 root dialout 188, 0 May 21 22:38 /dev/ttyUSB0
ubuntu@ubuntu-desktop-vm:~$
```

File: main.c

```c
#define F_CPU   16000000UL // set the CPU speed to 16MHz

#include <avr/io.h>       // for PORTx, DDRx, ... I/O registers
#include <util/delay.h>   // for _delay_ms();

int main(){
  // set direction of PB5 pin to output (onboard LED)
  DDRB |= (1<<5); // set DDB5 bit
  while(1) {
    PORTB |= (1<<5);  // output high to PB5 (set bit)
    _delay_ms(500);
    PORTB &= ~(1<<5); // output low to PB5 (clear bit)
    _delay_ms(500);
  }
}
```

"This is a sample C code written in a 'bare-metal style' and targeted at the ATmega328P MCU on the Arduino Uno/Nano board. It can be compiled with the AVR-GCC toolchain to build the firmware (binary) file.

File: `main.c` (for AVR ATmega328P), using nano as text-based editor

```
ubuntu@ubuntu-desktop-vm: ~/AVR

  GNU nano 6.2                         main.c
#define F_CPU   16000000UL // set the CPU speed to 16MHz

#include <avr/io.h>        // for PORTx, DDRx, ... I/O registers
#include <util/delay.h>    // for _delay_ms();

int main(){
  // set direction of PB5 pin to output
  DDRB |= (1<<5); // set DDB5 bit
  while(1) {
    PORTB |= (1<<5);  // output high to PB5 (set bit)
    _delay_ms(500);
    PORTB &= ~(1<<5); // output low to PB5 (clear bit)
    _delay_ms(500);
  }
}

^G Help       ^O Write Out  ^W Where Is   ^K Cut        ^T Execute    ^C Location
^X Exit       ^R Read File  ^\ Replace    ^U Paste      ^J Justify    ^/ Go To Line
```

To save and exit: Ctrl+O, Enter and Ctrl-X

Install the AVR-GCC toolchain on Ubuntu and compile the C source code.

```
# Install the toolchain for AVR-GCC.
$ sudo apt install -y build-essential \
  gcc-avr binutils-avr avr-libc avrdude

# Show the version of the AVR-gcc compiler.
$ avr-gcc --version | head -n 1

# Compile the source code into an .elf file.
$ avr-gcc -Os -Wall -mmcu=atmega328p -lc -lm -o main.elf ./main.c

# Convert the ELF file into a .hex file (Intel hex file).
$ avr-objcopy -j .text -j .data -O ihex main.elf main.hex

# Upload the firmware file (.hex) to the Arduino board.
$ avrdude -p atmega328p -c arduino -b 115200 -P /dev/ttyUSB0 \
 -D -Uflash:w:main.hex:i
```

```
# Remove the AVR-GCC toolchain
$ sudo apt remove --purge gcc-avr binutils-avr avr-libc avrdude
```

Check the dependencies of a meta-package.

```
$ apt-cache depends build-essential

build-essential
 |Depends: libc6-dev
  Depends: <libc-dev>
    libc6-dev
  Depends: gcc
  Depends: g++
  Depends: make
    make-guile
  Depends: dpkg-dev
```

```
$ apt-cache depends gcc-avr
gcc-avr
  Depends: libc6
  Depends: libgmp10
  Depends: libmpc3
  Depends: libmpfr6
  Depends: zlib1g
  Depends: binutils-avr
  Conflicts: avr-libc
  Suggests: gcc-doc
  Suggests: gcc
  Suggests: avr-libc
```

Compile the C source code for AVR and upload the binary file to the board.

ubuntu@ubuntu-desktop-vm: ~/AVR

```
ubuntu@ubuntu-desktop-vm:~/AVR$  avr-gcc --version | head -n 1
avr-gcc (GCC) 5.4.0
ubuntu@ubuntu-desktop-vm:~/AVR$ avr-gcc -Os -Wall -mmcu=atmega328p -lc -lm -o main.elf ./main.c
ubuntu@ubuntu-desktop-vm:~/AVR$ avr-objcopy -j .text -j .data -O ihex main.elf main.hex
ubuntu@ubuntu-desktop-vm:~/AVR$ avrdude -p atmega328p -c arduino -b 115200 -P /dev/ttyUSB0 \
>  -D -Uflash:w:main.hex:i

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.01s

avrdude: Device signature = 0x1e950f (probably m328p)
avrdude: reading input file "main.hex"
avrdude: writing flash (182 bytes):

Writing | ################################################## | 100% 0.06s

avrdude: 182 bytes of flash written
avrdude: verifying flash memory against main.hex:
avrdude: load data flash data from input file main.hex:
avrdude: input file main.hex contains 182 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.04s

avrdude: verifying ...
avrdude: 182 bytes of flash verified
```

Makefile

```
# Set the firmware file name
FIRMWARE=main
# Set the target MCU
MCU=atmega328p
# Set the serial port for firmware uploading
PORT=/dev/ttyUSB0
# Set the serial baudrate
BAUDRATE=115200
# Set executables
CC=avr-gcc
OBJCOPY=avr-objcopy
AVRDUDE=avrdude
# Enable compilation warning and optimize code for size
CFLAGS +=-std=gnu99 -Wall -Os -mmcu=$(MCU)
# Set linker flags
LDFLAGS +=-lc -lm
# Define C source files (all .c files in the project directory)
SRCS = $(wildcard *.c)
# Define object files
OBJ_FILES = $(SRCS:.c=.o)
# Define Phony targets
.PHONY: all clean flash
all: main
	@echo "done..."
main: $(OBJ_FILES)
	$(CC) $(CFLAGS) $(LDFLAGS) $^ -o $@.elf
	$(OBJCOPY) -j .text -j .data -O ihex $@.elf $(FIRMWARE).hex
flash: $(FIRMWARE).hex
	$(AVRDUDE) -p $(MCU) -c arduino -b $(BAUDRATE) -P $(PORT) \
	-D -Uflash:w:$(FIRMWARE).hex:i
%.o: %.c # use pattern rules
	$(CC) $(CFLAGS) -c $<
clean:
	rm -f *.o *.elf *.hex *.map
```

```
$ make -f Makefile clean
$ make -f Makefile all flash
```

Dockerfile for the AVR-GCC toolchain on Ubuntu 22.04.

```
FROM ubuntu:22.04

# Update package repositories and install required dependencies
RUN apt-get update && apt-get install -y \
    build-essential \
    gcc-avr \
    binutils-avr \
    avr-libc \
    avrdude && \
    apt-get autoclean -y && \
    apt-get autoremove -y

WORKDIR /build
```

Build a Docker image from the Dockerfile

```
$ docker build -t avr-toolchain ./
$ docker images -a # List all Docker images built locally.
```

Use AVR-GCC toolchain inside a Docker container.

```
# Show the version of the AVR GCC toolchain.
$ docker run -v $(pwd):/build avr-toolchain \
  avr-gcc --version | head -n 1

# Compile the main.c file in ${{PWD}/build/ and generate an ELF file.
$ docker run -it -v $(pwd):/build avr-toolchain \
  avr-gcc -Os -Wall -mmcu=atmega328p -lc -lm -o main.elf ./main.c

# Convert the ELF file into a .hex file (Intel hex file).
$ docker run -it -v $(pwd):/build avr-toolchain \
  avr-objcopy -j .text -j .data -O ihex main.elf main.hex

# Upload the firmware file (.hex) to the Arduino board.
$ docker run -it --privileged -v $(pwd):/build avr-toolchain \
  avrdude -p atmega328p -c arduino -b 115200 -P /dev/ttyUSB0 \
  -D -Uflash:w:main.hex:i
```

Arduino CLI

- **Arduino CLI (Command Line Interface)**, is a software tool designed to facilitate interaction with Arduino boards and libraries via the command line, eliminating the need for the **Arduino IDE**.
- It enables users to compile, upload, and manage Arduino sketches and libraries effortlessly.
- Its capabilities extend to automation, scripting, and seamless integration into various development workflows.

Install Arduino CLI on Ubuntu VM.

```
# Install the Arduino CLI tool to the /usr/local/bin directory.
$ curl -fsSL https://raw.githubusercontent.com/arduino/arduino-cli/master/install.sh \
  | sudo BINDIR=/usr/local/bin sh
# Show the version of the Arduino-CLI tool.
$ `which arduino-cli` version

# Update the index of Arduino cores to the latest version.
$ arduino-cli core update-index

# Install Arduino Core for AVR.
$ arduino-cli core install arduino:avr
```

Create / Build / Upload Arduino Sketch Using Arduino CLI.

```
# Create a new Arduino sketch named “led_blink”
$ mkdir -p $HOME/Arduino && cd $HOME/Arduino
$ arduino-cli sketch new led_blink -f
$ cd led_blink/

# Use the nano editor to edit the Arduino Sketch file.
$ nano led_blink.ino

# Build the Arduino sketch for the Arduino Nano board.
$ arduino-cli compile --fqbn arduino:avr:nano

# Upload the Arduino sketch to the targte board (Arduino Nano).
$ arduino-cli upload --fqbn arduino:avr:nano -v -p /dev/ttyUSB0

# Open Arduino Serial monitor to receive messages over Serial.
$ arduino-cli monitor -p /dev/ttyUSB0 --config baudrate=115200
```

Arduino Sketch File: `led_blink.ino`

```
#define LED_PIN LED_BUILTIN

void setup() {
  Serial.begin( 115200 ); // Open the serial port
  pinMode( LED_PIN, OUTPUT ); // Set output pin direction
}

void loop() {
  bool state = digitalRead( LED_PIN ); // Read the LED state
  state = !state; // Modify/Toggle the LED state
  digitalWrite( LED_PIN, state ); // Update the LED output
  Serial.print( "LED state: " );  // Send a message to Serial
  Serial.println( state );
  delay(100);
}
```

Create / Build / Upload Arduino Sketch Using Arduino CLI

ubuntu@ubuntu-server-vm: ~/Arduino/led_blink

```
ubuntu@ubuntu-server-vm:~/Arduino/led_blink$ arduino-cli compile --fqbn arduino:avr:nano
Sketch uses 2060 bytes (6%) of program storage space. Maximum is 30720 bytes.
Global variables use 200 bytes (9%) of dynamic memory, leaving 1848 bytes for local variables. Maximum is 2048 bytes.

Used platform Version Path
arduino:avr   1.8.6   /home/ubuntu/.arduino15/packages/arduino/hardware/avr/1.8.6
ubuntu@ubuntu-server-vm:~/Arduino/led_blink$ arduino-cli upload --fqbn arduino:avr:nano -v -p /dev/ttyUSB0
"/home/ubuntu/.arduino15/packages/arduino/tools/avrdude/6.3.0-arduino17/bin/avrdude" "-C/home/ubuntu/.arduino15/package
s/arduino/tools/avrdude/6.3.0-arduino17/etc/avrdude.conf" -v -V -patmega328p -carduino "-P/dev/ttyUSB0" -b115200 -D "-U
flash:w:/tmp/arduino/sketches/5C963C4FB47790C44BB49B8C27DD2AE3/led_blink.ino.hex:i"

avrdude: Version 6.3-20190619
         Copyright (c) 2000-2005 Brian Dean, http://www.bdmicro.com/
         Copyright (c) 2007-2014 Joerg Wunsch

         System wide configuration file is "/home/ubuntu/.arduino15/packages/arduino/tools/avrdude/6.3.0-arduino17/etc/
avrdude.conf"
         User configuration file is "/home/ubuntu/.avrduderc"
         User configuration file does not exist or is not a regular file, skipping

         Using Port                    : /dev/ttyUSB0
         Using Programmer              : arduino
         Overriding Baud Rate          : 115200
         AVR Part                      : ATmega328P
         Chip Erase delay              : 9000 us
```

Use Arduino Serial Monitor.

ubuntu@ubuntu-server-vm: ~/Arduino/led_blink

```
ubuntu@ubuntu-server-vm:~/Arduino/led_blink$ arduino-cli monitor -p /dev/ttyUSB0 --config baudrate=115200
Monitor port settings:
baudrate=115200
Connected to /dev/ttyUSB0! Press CTRL-C to exit.
LED state: 1
LED state: 0
LED state: 1
LED state: 0
LED state: 1
LED state: 0
LED state: 1
LED state: 0
LED state: 1
LED state: 0
LED state: 1
LED state: 0
^C
ubuntu@ubuntu-server-vm:~/Arduino/led_blink$
```

Use Python Virtual Environment.

```
# Install Python3 Virtual Environment.
$ sudo apt install -y python3 python3-pip python3-venv
$ python3 --version
$ pip3 --version

# Create a new virtual environment named 'pyenv'.
$ python3 -m venv 'pyenv'

# Activate the virtual environment.
# To deactivate, use the command line: deactivate
$ source pyenv/bin/activate

# Install or update the python serial package.
$ pip install pyserial -U
```

File: `read_serial.py`

```
import serial
import time

# Serial port settings
port = '/dev/ttyUSB0'
baudrate = 115200

# Open the serial port
ser = serial.Serial(port, baudrate, timeout=0.5)

try:
    while True:
        # Read a line from the serial port
        line = ser.readline()
        if line is not None:
            line = line.decode().strip()
            # Check if the line is not empty
            if line:
               print("Received:", line)
except KeyboardInterrupt:
    # Close the serial port on Ctrl+C
    ser.close()
```

```
$ python ./read_serial.py
```

Run a Python script in a Python virtual environment.

ubuntu@ubuntu-desktop-vm: ~/Arduino/led_blink

```
(pyenv) ubuntu@ubuntu-desktop-vm:~/Arduino/led_blink$ python ./read_serial.py
Received: LED state: 1
Received: LED state: 0
Received: LED state: 1
Received: LED state: 0
Received: LED state: 1
Received: LED state: 0
Received: LED state: 1
Received: LED state: 0
Received: LED state: 1
Received: LED state: 0
Received: LED state: 1
Received: LED state: 0
^C(pyenv) ubuntu@ubuntu-desktop-vm:~/Arduino/led_blink$
```