Arduino / AVR Development Environment Setup

# Introduction

I develop complex applications and libraries for Atmel AVR microcontroller (on Arduino or directly on chip). I essentially work from Windows 10, although I am also familiar with Linux.

Currently, I don’t consider Arduino IDE suitable for professional environment.

I have decided to use a real IDE for my AVR developments; there are several good IDE supporting C and C++ on the market; some are more or less easy to configure for Arduino development.

I have selected NetBeans 8 for a couple of reasons:

* Easy support for remote build (very useful for Raspberry Pi development)
* Support for cross compilation (remote or local)
* It’s free
* It supports other languages that are useful for me: JavaScript, HTML, CSS
* It generates a Makefile that allows building a project from the command line (that can be useful when you run headless or you want to setup a CI environment)

Despite several drawbacks:

* UI is not as friendly as other IDE such as Eclipse or IntelliJ IDEA
* It does not officially support Python (which I also need for some Raspberry Pi developments)
* It is not super easy to set up for Arduino/AVR support, hence this document.

My Arduino projects use C++ 11 and sometimes heavily depend on C++ templating facilities.

For this reason and a few other constraints, I always try to use the latest Atmel AVR toolchain (based on GNU GCC) so that I have the best possible C++ support and the least bugs. My main constraint is that, with Arduino 1.6.x integrated toolchain, I always have had issues at link time with the provided GCC.

After my first attempts using the latest AVR Toolchain (3.5.3) on Windows, I found an issue with the archiver tool I was using (this one is not integrated in the toolchain, you have to get GNU binutils for Windows, i.e. Cygwin or MinGW).

Due to this latest issue, I decided to finally build all my projects on Linux, whilst still using NetBeans on Windows as this is my everyday working environment.

Hence I decided to try using **remote build from NetBeans on Windows onto a VirtualBox Linux guest**, running on the same machine. This setup is the purpose of this document.

# VirtualBox setup

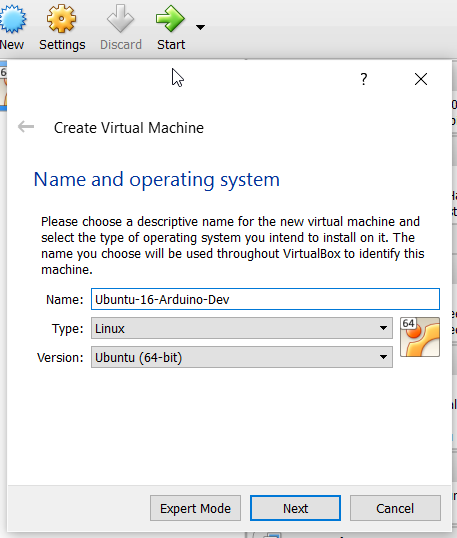
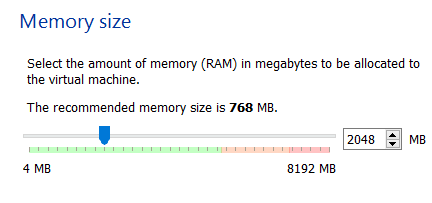
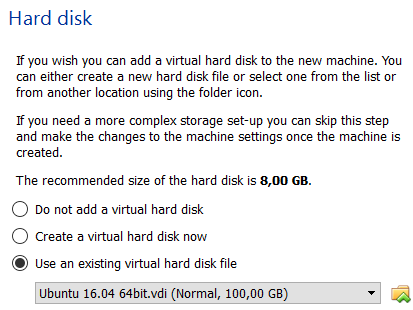
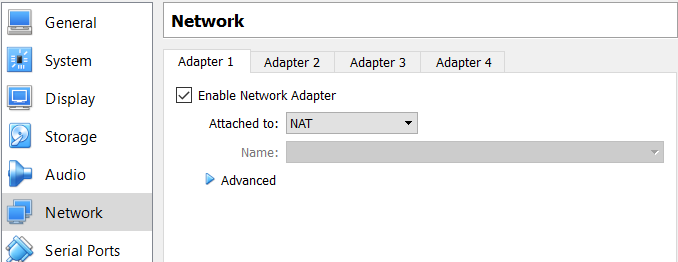
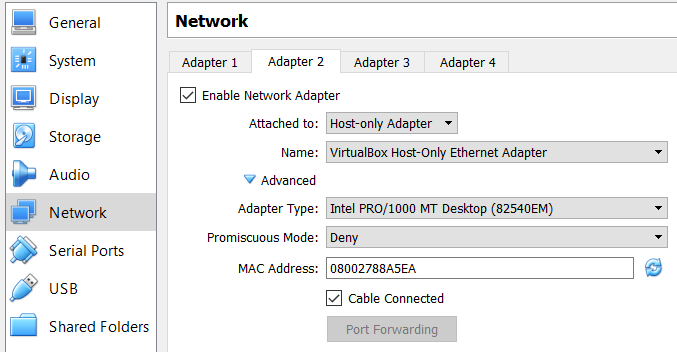
I currently use VirtualBox 5.0.26. I kept default preferences.

I have downloaded the following 64 bits image from <http://www.osboxes.org/ubuntu/>:

* Ubuntu 16.04 Xenial (Final Version)

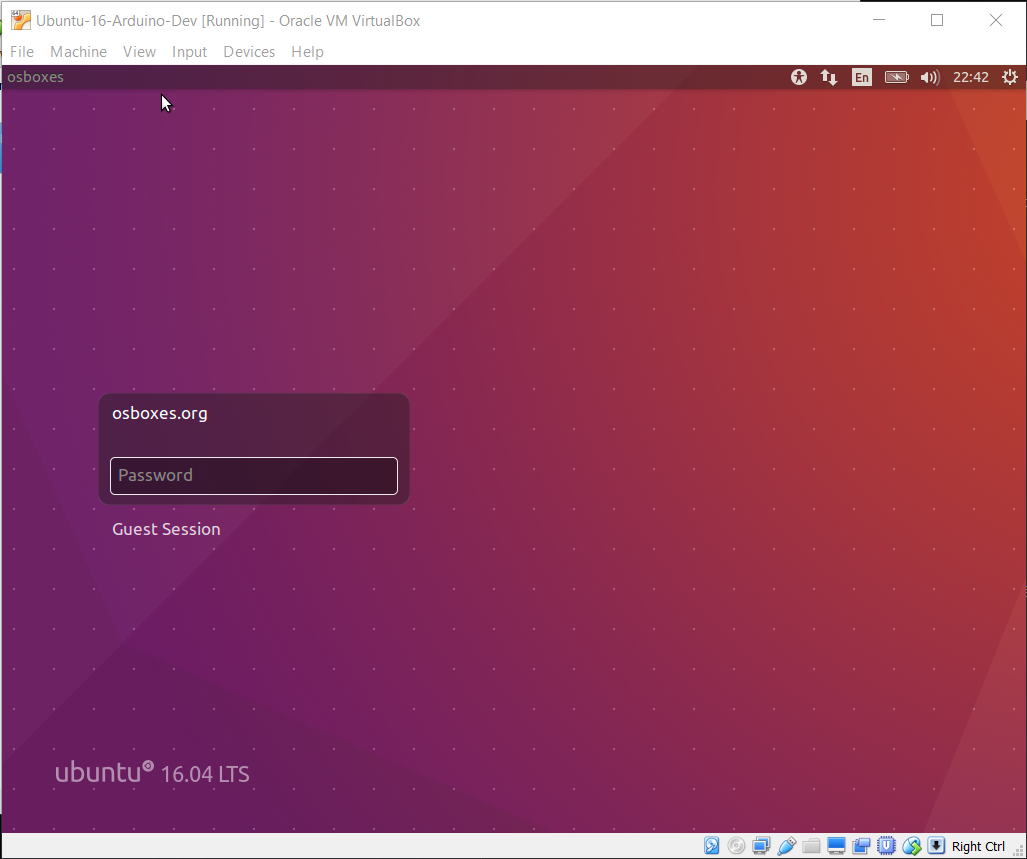
I had also tried Debian I found it was more difficult to setup for my needs, hence I won’t describe it here.

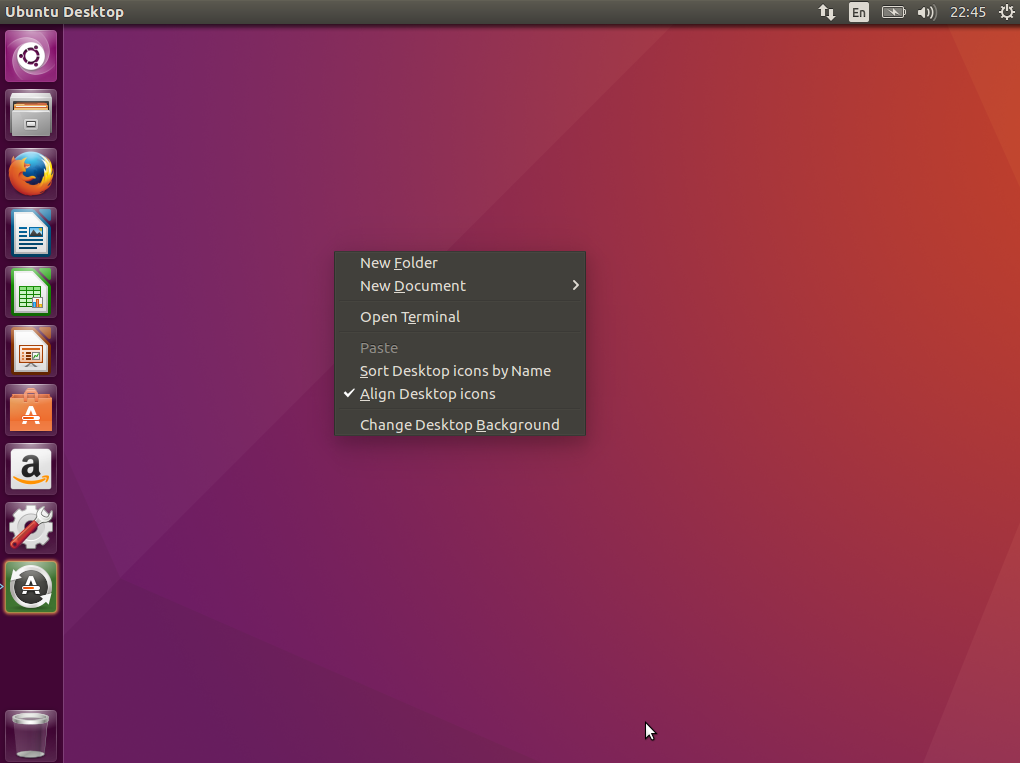
## Ubuntu 16

1. Create a new Virtual Machine, name it Ubuntu-16- -Arduino-Dev  
   
2. Select RAM for this VM, I recommend at least 2GB  
   
3. Use the existing VDI you got for Ubuntu image “Ubuntu 16.04 64bit.vdi”  
   
4. Update Settings of the new VM
   1. Keep defaults for General, System settings, Display, Storage, Audio, Serial Ports, USB and User Interface
   2. For Network, keep the default Adapter 1, attached to NAT:  
        
      Then enable Adapter 2 and attach it to “Host-only Adapter” and keep all other default settings:  
      
   3. Shared Folders settings: this change is optional, it is useful only if you consider copying files from Windows to Linux (or vice-versa), which normally should not happen, except potentially for the AVR-GNU-toolchain archive that you may already have downloaded on Windows but which you could as well get directly from your Linux VM through wget.

Note that it seems Ubuntu has an issue with VirtualBox with the NAT Adapter, which makes it impossible to use DNS. From your Windows command line you need to perform the following command to solve this issue:

> VBoxManage.exe modifyvm "Ubuntu-16-Arduino-Dev" --natdnshostresolver1 on

Now you can start the new VM, after a few seconds, you should see a new window similar to this:  


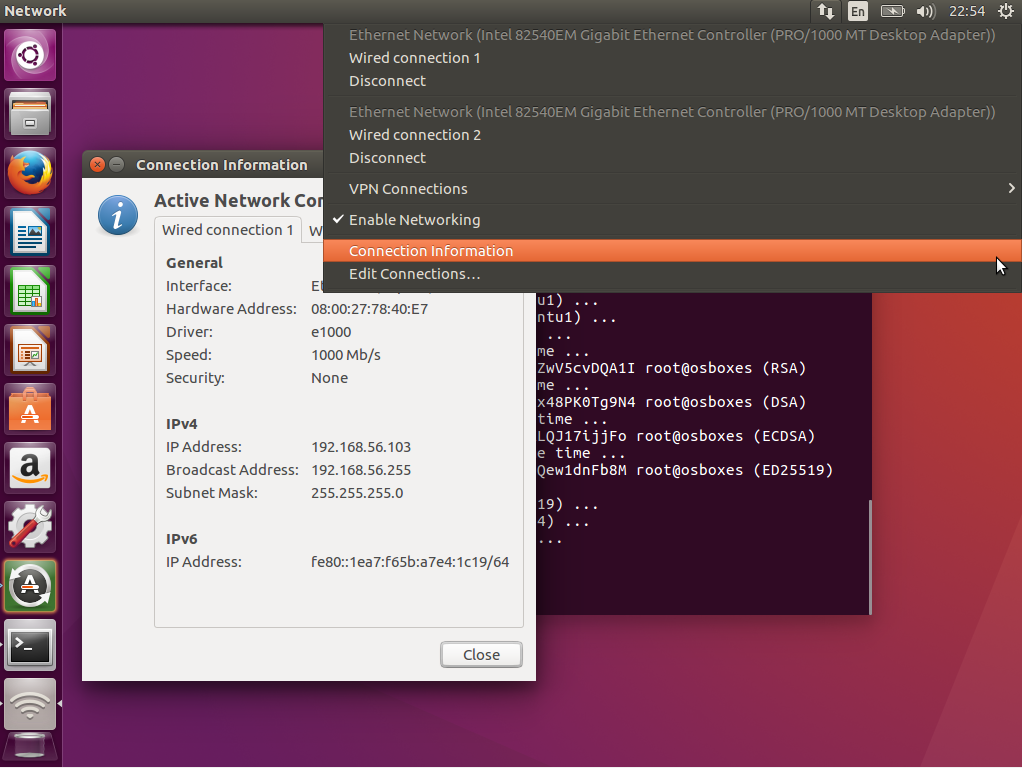
After signing in with osboxes.org (password: osboxes.org), you get Ubuntu desktop.  


Open a new terminal from which you’ll need to install ssh, then get AVR GNU Toolchain:

> sudo apt-get update

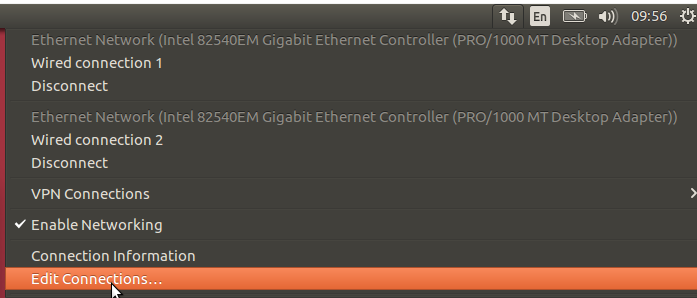
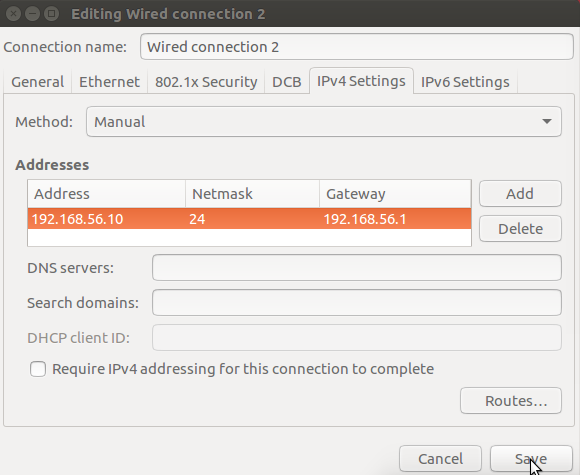
> sudo apt-get install openssh-server

> sudo ufw allow 22

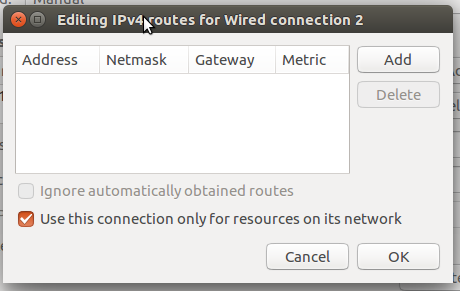
After that, you should be able to SSH to your VM. You can check that with Putty but you first need to find out the IP address for your VM:  


The dialog “Connection Information” should show 2 connections, one is used by NAT adapter, the other by the Host-only adapter: 192.168.56.103 in my case.

Note that this IP is currently assigned by the VirtualBox internal DHCP, and thus may change after restarting the VM. Thus, you’ll have to change it to a fixed IP.

For this, from Ubuntu desktop, open the “Edit Connections” dialog:  
  


Note that you have to select an address that is out of the DHCP range configured in VirtualBox for the Host-only network adapter. You can leave the DNS IP empty as you don’t need any DNS on this network adapter.

Finally, you should ensure this connection is not used for accessing the “outside world” (e.g. Internet) as that would always fail. For that, Click the “Routes…” button in the previous dialog above:  


Check the box “Use this connection only for resources on its network”.

Once you have finished setup of this connection, you have to restart your VM.

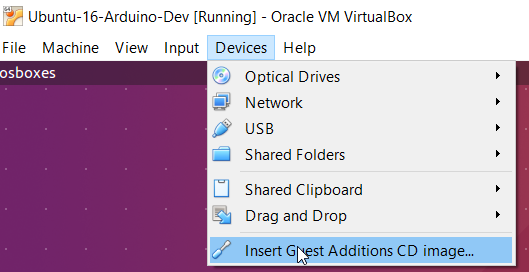
Now you’ll need to get the AVR GNU Toolchain from Atmel: <http://www.atmel.com/tools/atmelavrtoolchainforlinux.aspx>

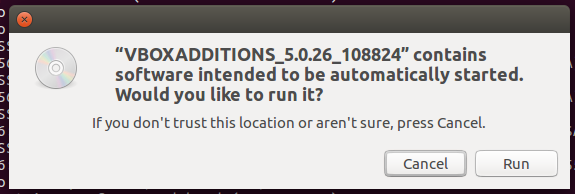


You will be requested for some personal information before you can get a download link by email. Just copy this link and paste it in your Linux terminal:

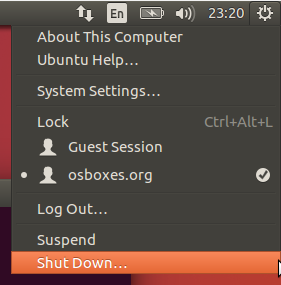
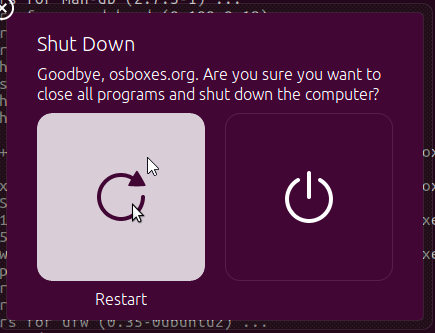
> wget <pasted link>

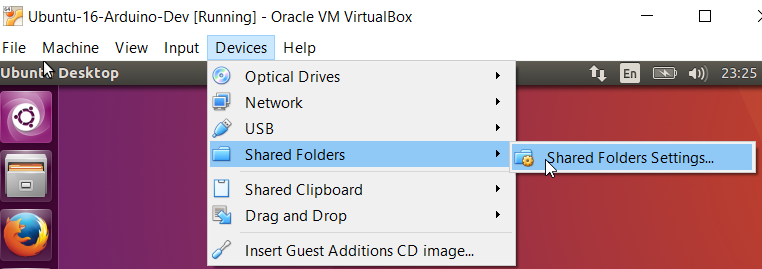
This should create a file named “avr8-gnu-toolchain-3.5.3.1700-linux.any.x86\_64.tar.gz” (or similar) on your home directory.

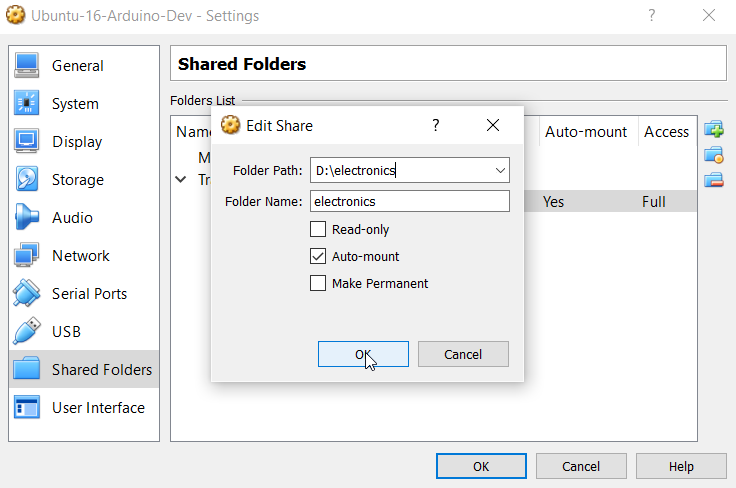
If you already have this file on your Windows machine, you don’t need to download it a second time, you can also share it with your Linux VM. For that, you first need to install VirtualBox Guest Additions to your VM:  


A popup window shall appear in your VM desktop:  


Click “Run” and follow instructions. Installation shall take some minutes to complete.

When done, you shall restart your Linux VM:  
  


Upon restart, you need to add as a Shared Folder the Windows directory that contains the AVR GNU Toolchain archive:  




Since you are not supposed to share folder forever, no need to make it permanent (you will only use it to copy the toolchain archive now). Although the auto-mount is normally not necessary, I found it was much easier to access the shared folder when this was checked (I had many issues with mount command in Ubuntu).

Restart your VM, login and open a terminal. You should be able to see the shared folder. However, to access it, you’ll need to add yourself to the group “vboxsf”:

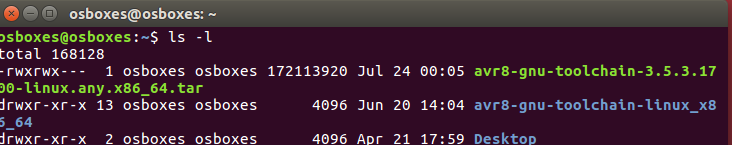
> sudo usermod –aG vboxsf osboxes

> ls /media/sf\_electronics

> cp /media/sf\_electronics/avr8-gnu-toolchain-3.5.3.1700-linux.any.x86\_64.tar.gz .

> gunzip avr8-gnu-toolchain-3.5.3.1700-linux.any.x86\_64.tar.gz

> tar xvf avr8-gnu-toolchain-3.5.3.1700-linux.any.x86\_64.tar

This shall have created a directory named “avr8-gnu-toolchain-linux\_x86\_64” on your Linux home directory:  


One additional step consists in updating the path in Linux so that it includes ~/avr8-gnu-toolchain-linux\_x86\_64/bin which contains all GNU-based executables for AVR (all the binaries are prefixed with “avr-“).

Actually the easiest way to do so on Ubuntu is to add a link named “bin” on your home directory and make it point to ~/avr8-gnu-toolchain-linux\_x86\_64/bin:

> ln –s avr8-gnu-toolchain-linux\_x86\_64/bin bin

Last, you have to install avrdude so that your builds will be able to upload AVR programs to the targets (Arduino through USB or an AVR MCU directly through an ISP programmer):

> sudo apt-get install avrdude

This will install avrdude 6.2 on your VM at /usr/bin. You’ll also have to install a configuration file for avrdude; I use the one provided with Arduino 1.6.9 (can be found at hardware\tools\avr\etc\avrdude.conf).

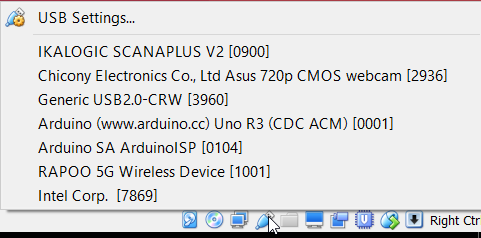
Personally, I set it as default configuration file so that I don’t have to specify it in command line, for that it must be copied to /etc:

> sudo cp avrdude.conf /etc

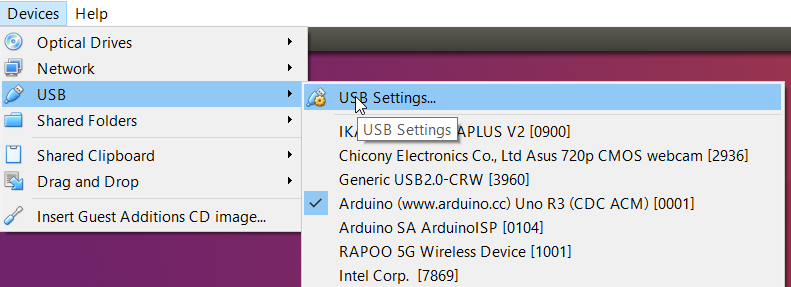
> sudo chmod a+r /etc/avrdude.conf

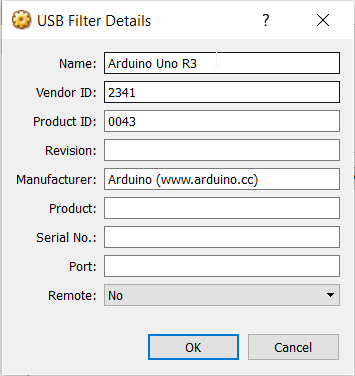
## USB USAGE

In order to effectively use avrdude on the Linux VM, it is necessary to ensure that USB ports, connected to an Arduino or an ISP programmer, are available to Linux.

With VirtualBox, it is easy to manually attach an USB port to a running VM:  


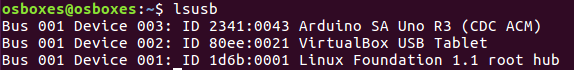
However, with this method, you will have to attach the device every time you restart the VM.

If you want your device always attached to the VM, you can also define a “USB device filter” through USB settings:  




VirtualBox will automatically fill all the fields for the selected device; it is up to you to make the filter less rigid by removing some fields, like in the screenshot above.

You can check your USB device has been successfully added to the Linux VM:

> lsusb [-vvv]  


In order to use this USB device from any program, in particular from the NetBeans build, you can use the device alias found on /dev/serial/by-id:

> ls –l /dev/serial/by-id  


As you can see, this links to /dev/ttyACM0 here. You can either use the device or the alias for your programs.

However, in order to get access to the device, you will have to grant yourself permissions:

In order to use this USB device from any program, in particular from the NetBeans build, you can use the device alias found on /dev/serial/by-id:

> sudo usermod –aG dialout osboxes  
> sudo chmod a+rw /dev/ttyACM0

Then you are able to use the device, e.g. in a serial terminal or avrdude.

The command line below shows stty settings that work well:

> stty -F /dev/ttyACM0 cs8 115200 ignbrk -brkint -icrnl -imaxbel -opost -onlcr -isig -icanon -iexten -echo -echoe -echok -echoctl -echoke noflsh -ixon –crtscts

**Important!** If you intend to use an ISP programmer that does not get viewed directly as a tty device (e.g. ArduinoISP), then you will need to ensure that avrdude is run as suid so that it is executed in SSH as root, this can be done with the simple command line below:

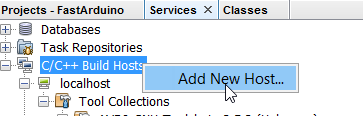
> sudo chmod +s /usr/bin/avrdude

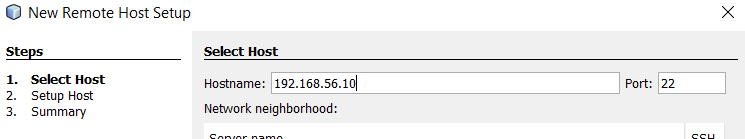
Please note that this is normally unsafe to do so, but considering your VM shall not be accessible from outside your Windows box, I would not consider that an issue.

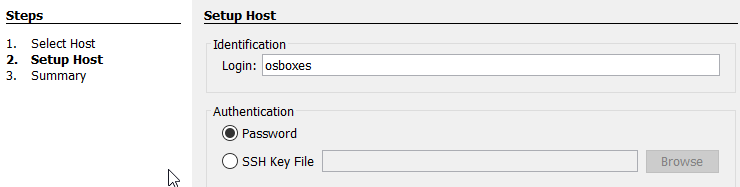
References:  
<https://mightyohm.com/blog/2010/03/run-avrdude-without-root-privs-in-ubuntu/>  
<https://learn.adafruit.com/usbtinyisp/avrdude>

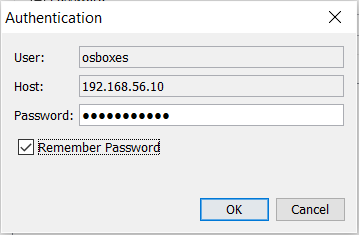
# NetBeans setup

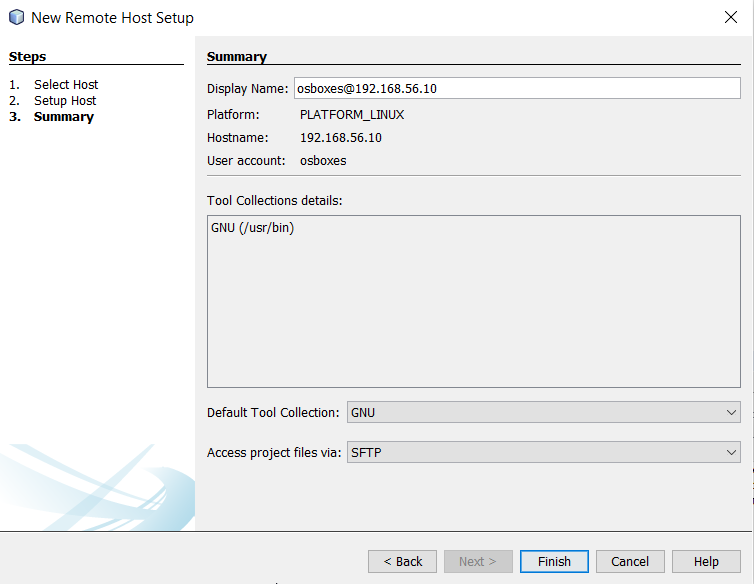
## Remote Build setup

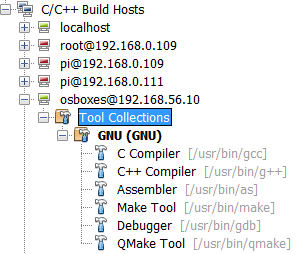
First, we need to setup a new Tools Collection on a new C/C++ Build host:  


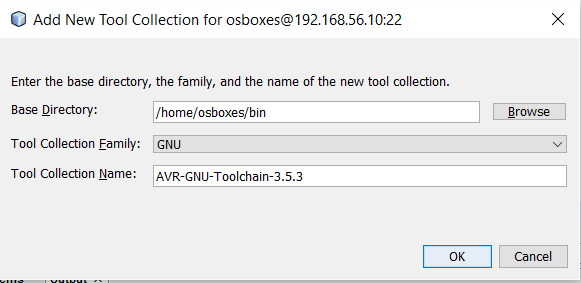
Type the IP you have configured for your Ubuntu VM:  


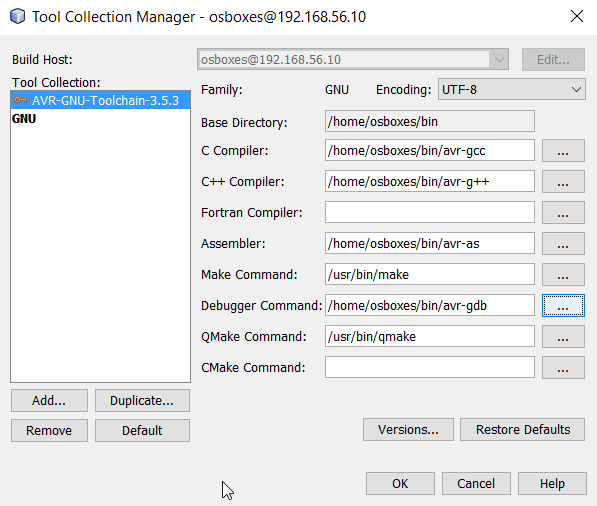
Enter the login name on your VM:  


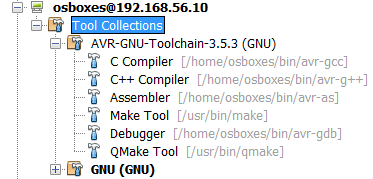
Set and remember the password:  


NetBeans will find GNU GCC collection on the system (but not the AVR GNU toolchain):  


The new build host has been created with new Tools Collection:  


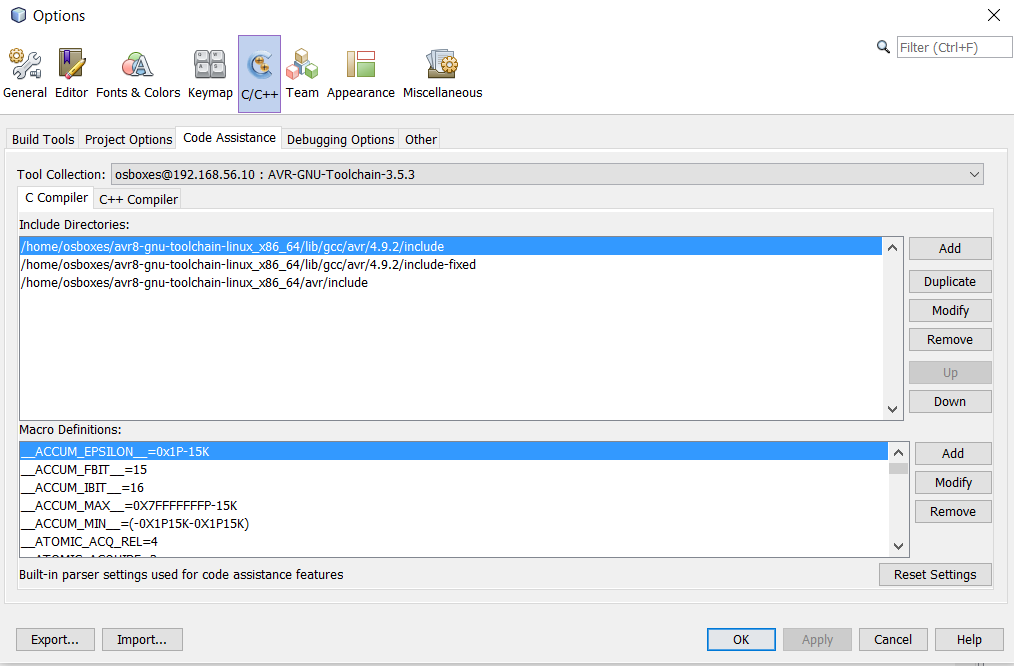
Now we need to add a new Tools collection for AVR:  


Then you have to individually set the path to each tool in the collection:  


Now this is what you should see in the C/C++ Build Hosts:  


TODO Make it the default.

## NetBeans Options

Next, we must setup C/C++ options for the new Tools Collection:  


In my own projects, for Code assistance, I typically add some specific macro definitions, in addition to all macros automatically set by the AVR Toolchain:  
 ARDUINO\_UNO  
 \_\_AVR\_ATmega328P\_\_  
 F\_CPU=16000000UL

The first is used in my project headers to determine the AVR target and then direct conditional compilation based on this target. The second is used by AVR headers to determine the MCU target; normally this id defined in command line when invoking the compiler but here, we need it on Netbeans code assistance.

All that is useful only during code edition, as building will override this macro through a specific Makefile in the project.

## Project setup

TODO Show new project creation in Netbeans

I always create one configuration per possible AVR target in my projects:

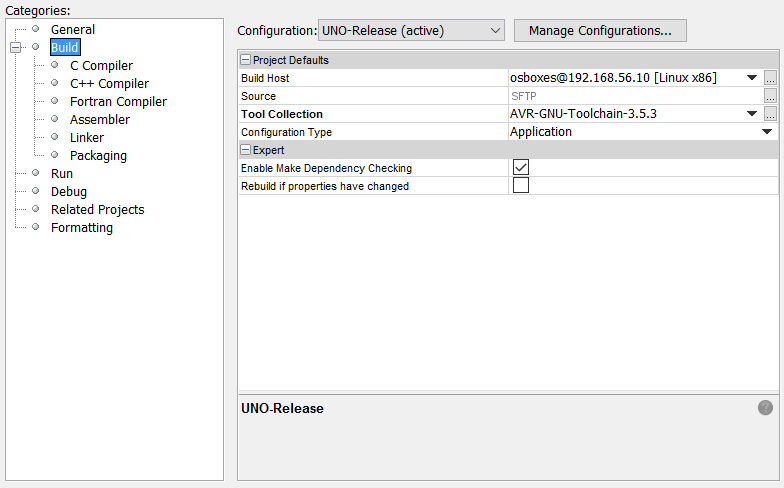
* UNO-Release (directly maps to Arduino UNO)
* MEGA-Release (directly maps to Arduino MEGA 2560)
* ATmega328-Release (maps to breadboard ATmega328P with internal RC clock at 8MHz)
* ATtiny84-Release (maps to breadboard ATtiny84 with internal RC clock at 8MHz)

I create the first configuration from scratch (as described in the following sections) and then duplicate it to change only the minimum stuff:

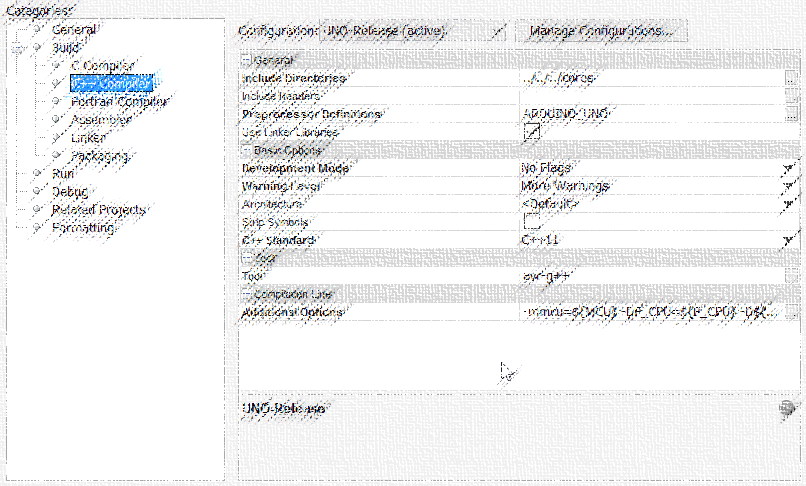
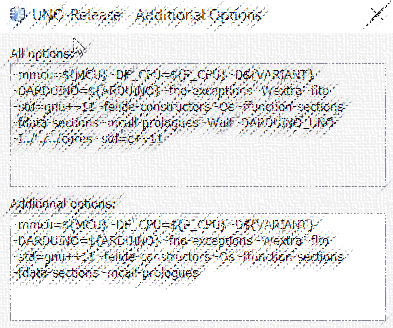
TODO screenshot to duplicate config

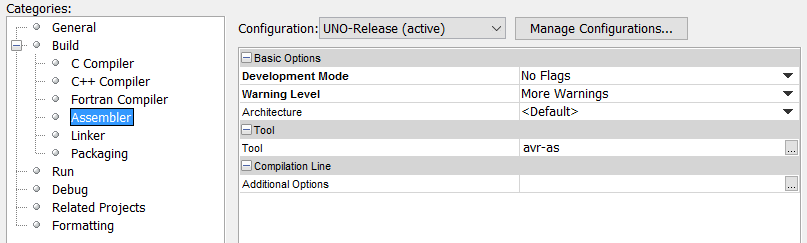
First I create the configuration for UNO-Release:

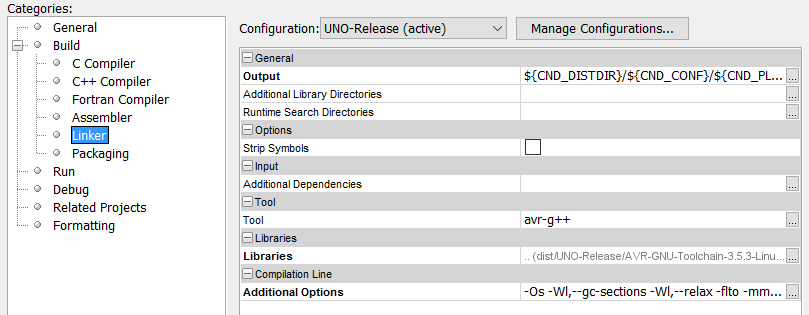
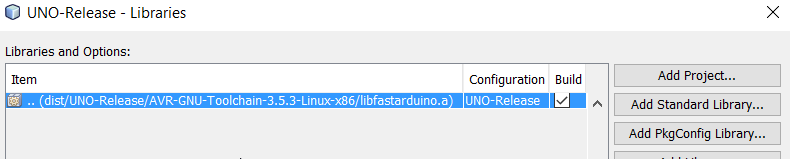
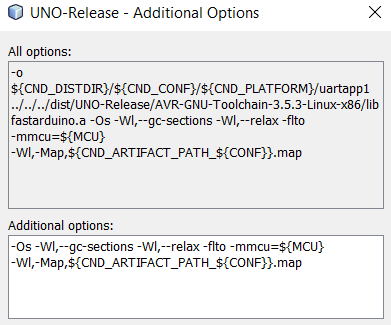
TODO screenshot to create config

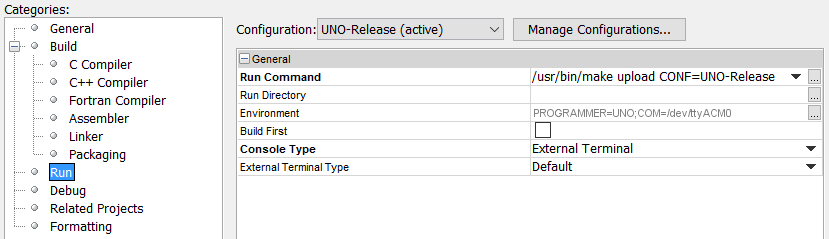
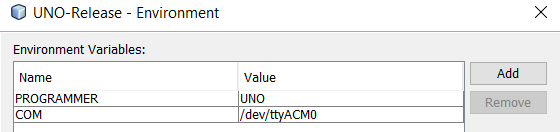
Here are the Global build settings:  


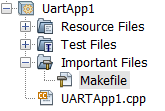
Since we do not use the C compiler, there is no need to perform any specific setting for it.

Here are the -important- settings for C++ compiler:  
  
  
-mmcu=${MCU} -DF\_CPU=${F\_CPU} -D${VARIANT} -fno-exceptions -Wextra -flto -std=gnu++11 -felide-constructors -Os -ffunction-sections -fdata-sections -mcall-prologues

Here are the settings for the Assembler:  


Here are settings for the linker:  
  
  
  
-Os -Wl,--gc-sections -Wl,--relax -flto -mmcu=${MCU} -Wl,-Map,${CND\_ARTIFACT\_PATH\_${CONF}}.map

Here are the setting for Run, which will actually build then upload the program to the target:  
  


For each project, I update the Makefile that is initially generated by Netbeans:  


I add the following at the beginning of the file (after the header comments):

# JFP BEGIN special variables stuff

# set variables based on each configuration

ifeq ($(findstring UNO,${CONF}),UNO)

VARIANT=ARDUINO\_UNO

MCU=atmega328p

ARCH=avr5

F\_CPU=16000000L

else

ifeq ($(findstring ATmega328,${CONF}),ATmega328)

VARIANT=arduino/atmega328p

MCU=atmega328p

ARCH=avr5

F\_CPU=8000000L

else

ifeq ($(findstring MEGA,${CONF}),MEGA)

VARIANT=ARDUINO\_MEGA

MCU=atmega2560

ARCH=avr6

F\_CPU=16000000L

#TODO Add other targets here (Leonardo, ATtiny)

endif

endif

endif

# Set upload options

ifeq (${PROGRAMMER},)

PROGRAMMER=UNO

endif

AVRDUDE\_OPTIONS=-p ${MCU}

ifeq (${PROGRAMMER},ISP)

AVRDUDE\_OPTIONS+= -c arduinoisp

endif

ifeq (${PROGRAMMER},SHIELD)

AVRDUDE\_OPTIONS+= -c stk500v1 -b 19200 -P ${COM}

endif

ifeq (${PROGRAMMER},UNO)

AVRDUDE\_OPTIONS+= -c arduino -b 115200 -P ${COM}

endif

ifeq (${PROGRAMMER},MEGA)

AVRDUDE\_OPTIONS+= -c wiring -b 115200 -P ${COM}

endif

#TODO need to add options for other programmers someday

# JFP END special variables stuff

This initializes several variables that are needed elsewhere in the make targets.

The first half is used for building the program for a given target.

The second half is used for uploading the built program to the target through the given means.

Then I change “.build-post” target to ensure the linked program is properly transformed for AVR target:

.build-post: .build-impl

# Add your post 'build' code here...

avr-objcopy -O ihex ${CND\_ARTIFACT\_PATH\_${CONF}} ${CND\_ARTIFACT\_PATH\_${CONF}}.hex

avr-nm --synthetic -S -C --size-sort ${CND\_ARTIFACT\_PATH\_${CONF}} >${CND\_ARTIFACT\_PATH\_${CONF}}.nm.txt

avr-objdump -m ${ARCH} -x -d -C ${CND\_ARTIFACT\_PATH\_${CONF}} >${CND\_ARTIFACT\_PATH\_${CONF}}.dump.txt

avr-size -C --mcu=${MCU} ${CND\_ARTIFACT\_PATH\_${CONF}}

Please note the optional avr-nm and avr-objdump that I often use to control the detailed output (assembly code) of the build.

Finally I add a new target “upload”:

upload:

avrdude ${AVRDUDE\_OPTIONS} -Uflash:w:${CND\_ARTIFACT\_PATH\_${CONF}}.hex:i

Note that no configuration file is provided to avrdude, because I have copied the Arduino provided avrdude.conf to the default location expected by avrdude.

Sometimes, I also add more specific targets for direct MCU programming (no Arduino board) where I would need to set the fuses myself.

TODO show example