Picozed Documentation

Release B

Architech

March 08, 2016

Contents

1 Notations **3**

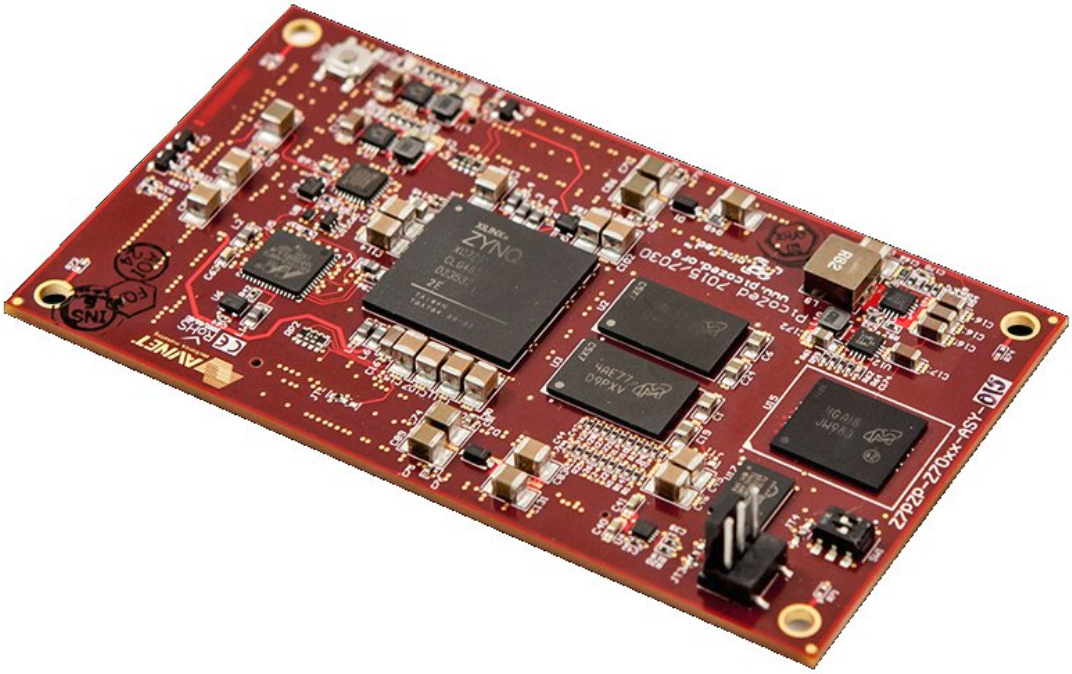
2 Chapters **5**

2.1 Unboxing 5
2.2 Quick start guide 5
2.3 SDK Architecture 40
2.4 Create SDK 60
2.5 BSP 62
2.6 Toolchain 65
2.7 The board 79
2.8 FAQ 85
2.9 Appendix 86

Version 2.2.0C

Copyright Architech

Date 08/03/16



Welcome to **Picozed** documentation!

Have you just received your Picozed board? Then you sure want to read the *Unboxing* Chapter first.

If you are a new user of the **Yocto based SDK** we suggest you to read the *Quick start guide* chapter, otherwise, if you want to have a better understanding of specific topics, just jump directly to the chapter that interests you the most.

Furthermore, we encourage you to read the official Yocto Project documentation.

CHAPTER 1

Notations

Throughout this guide, there are commands, file system paths, etc., that can either refer to the machine (real or virtual) you use to run the SDK or to the board.

Host

This box will be used to refer to the machine running the SDK

Board

This box will be used to refer to Picozed board

However, the previous notations can make you struggle with long lines. In such a case, the following notation is used.

If you click on *select* on the top right corner of these two last boxes, you will get the text inside the box selected. We have to warn you that your browser might select the line numbers as well, so, the first time you use such a feature, you are invited to check it.

Sometimes, when referring to file system paths, the path starts with **/path/to**. In such a case, the documentation is **NOT** referring to a physical file system path, it just means you need to read the path, understand what it means, and understand what is the proper path on your system. For example, when referring to the device file associated to your USB flash memory you could read something like this in the documentation:

Since things are different from one machine to another, you need to understand its meaning and corresponding value for your machine, like for example:

When referring to a specific partition of a device, you could read something like this in the documentation:

Even in this case, the things are different from one machine to another, like for example:

we are referring to the device /dev/sdb and in the specific to the partition 1. To know more details please refer to *device files* section of the *appendix*.

Chapters

2.1 Unboxing

Picozed comes with a very good and complete documentation. To verify the content of the box and to move the first steps with the board, visit the official documentation page, in particular have a look at "**PicoZed FMC Carrier Card Getting Started Guide v2.0**" document.

2.2 Quick start guide

This document will guide you from importing the virtual machine to debugging an *Hello World!* example on a customized Linux distribution you will generate with **OpenEmbedded/Yocto** toolchain.

2.2.1 Install

The development environment is provided as a virtual disk (to be used by a VirtualBox virtual machine) which you can download from this page:

Important: http://downloads.architechboards.com/sdk/virtual_machine/download.html

Important: Compute the MD5SUM value of the zip file you downloaded and compare it to the golden one you find in the download page.

Uncompress the file, and you will get a *.vdi* file that is our virtual disk image. The environment contains the SDK for all the boards provided by Architech, Picozed included.

Download VirtualBox



For being able to use it, you first need to install **VirtualBox** (version 4.2.10 or higher). You can get VirtualBox installer from here:

https://www.virtualbox.org/wiki/Downloads

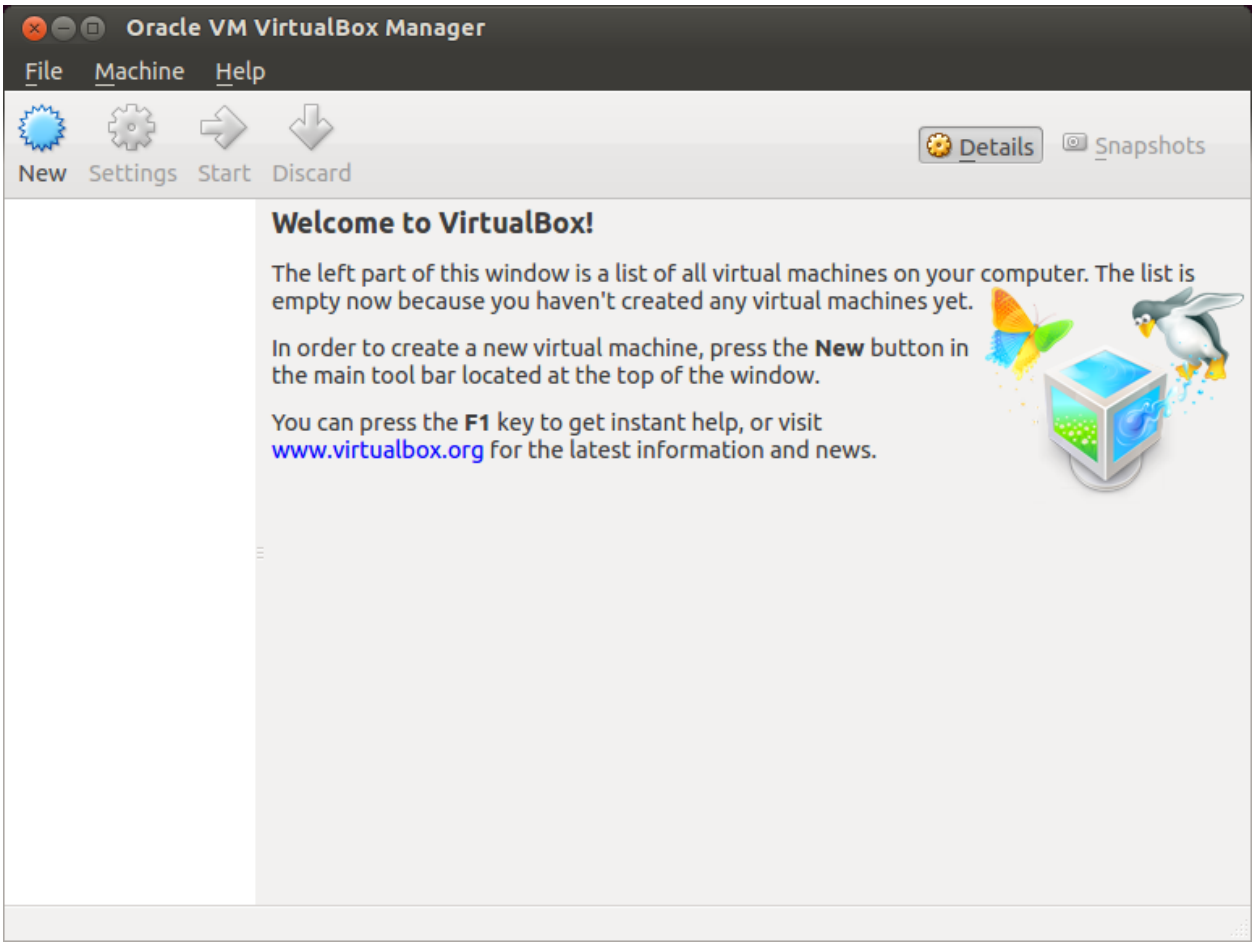
Download the version that suits your host operating system. You need to download and install the **Extension Pack** as well.

Important: Make sure that the extension pack has the same version of VirtualBox.

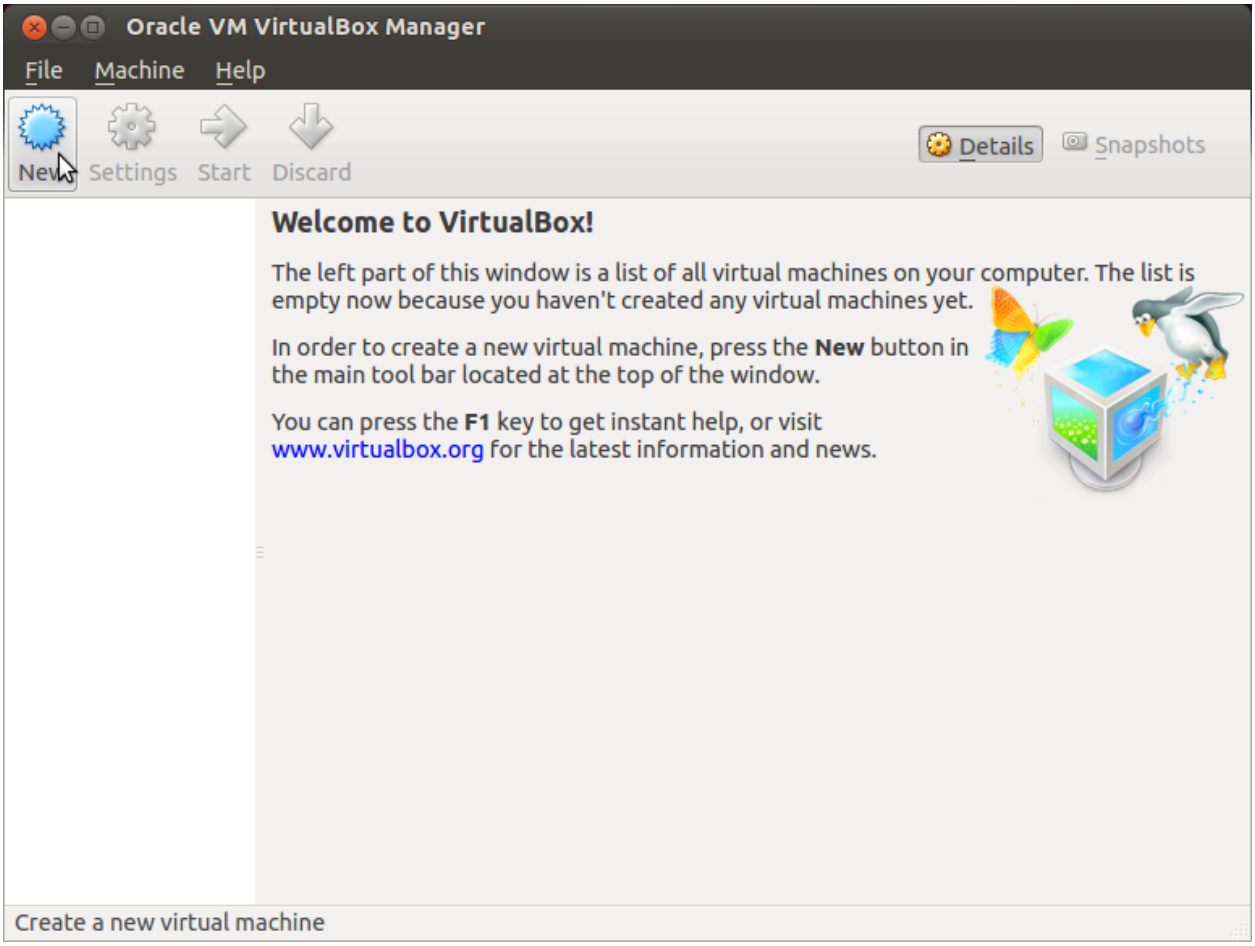
Install the software with all the default options.

Create a new Virtual Machine

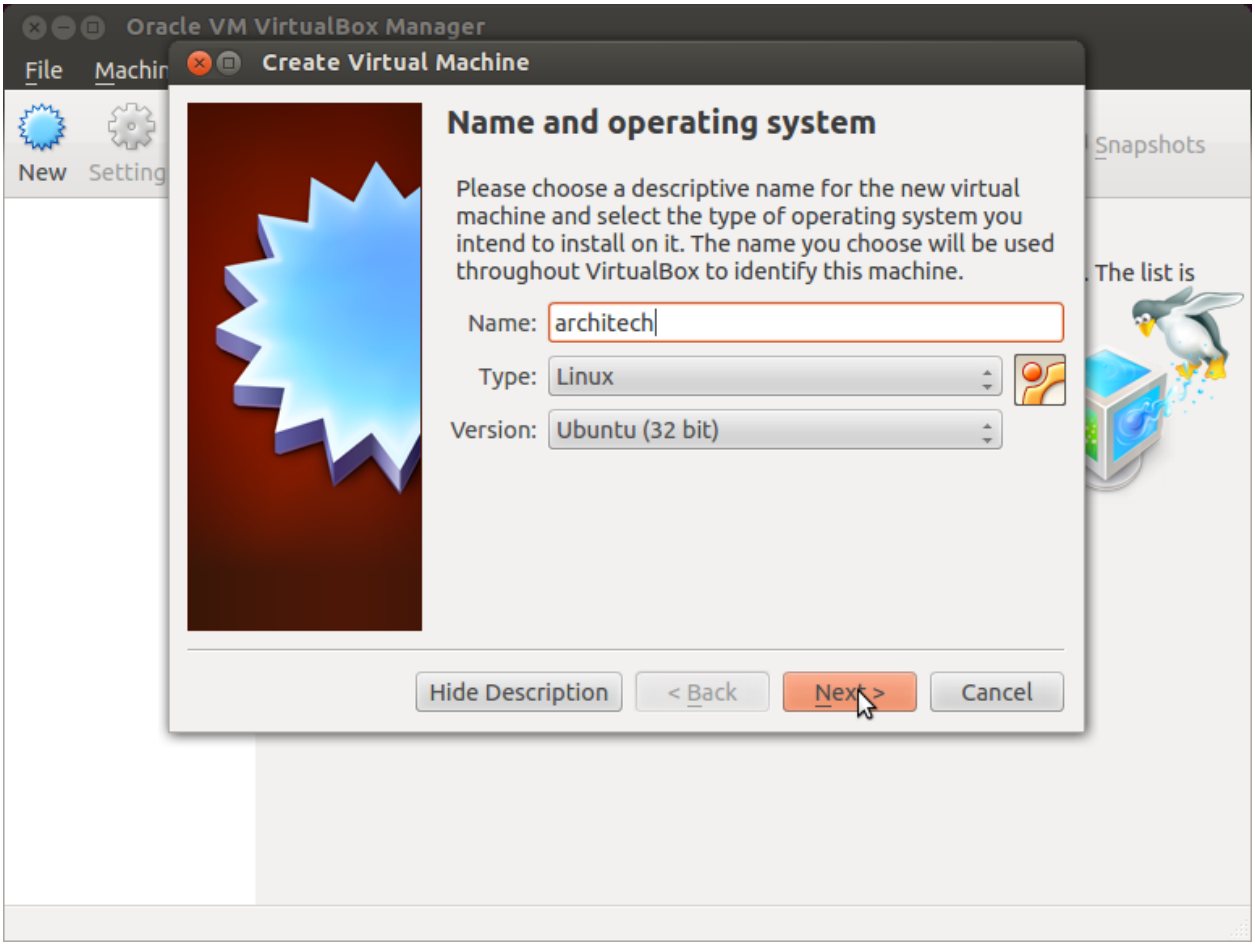
1. Run VirtualBox



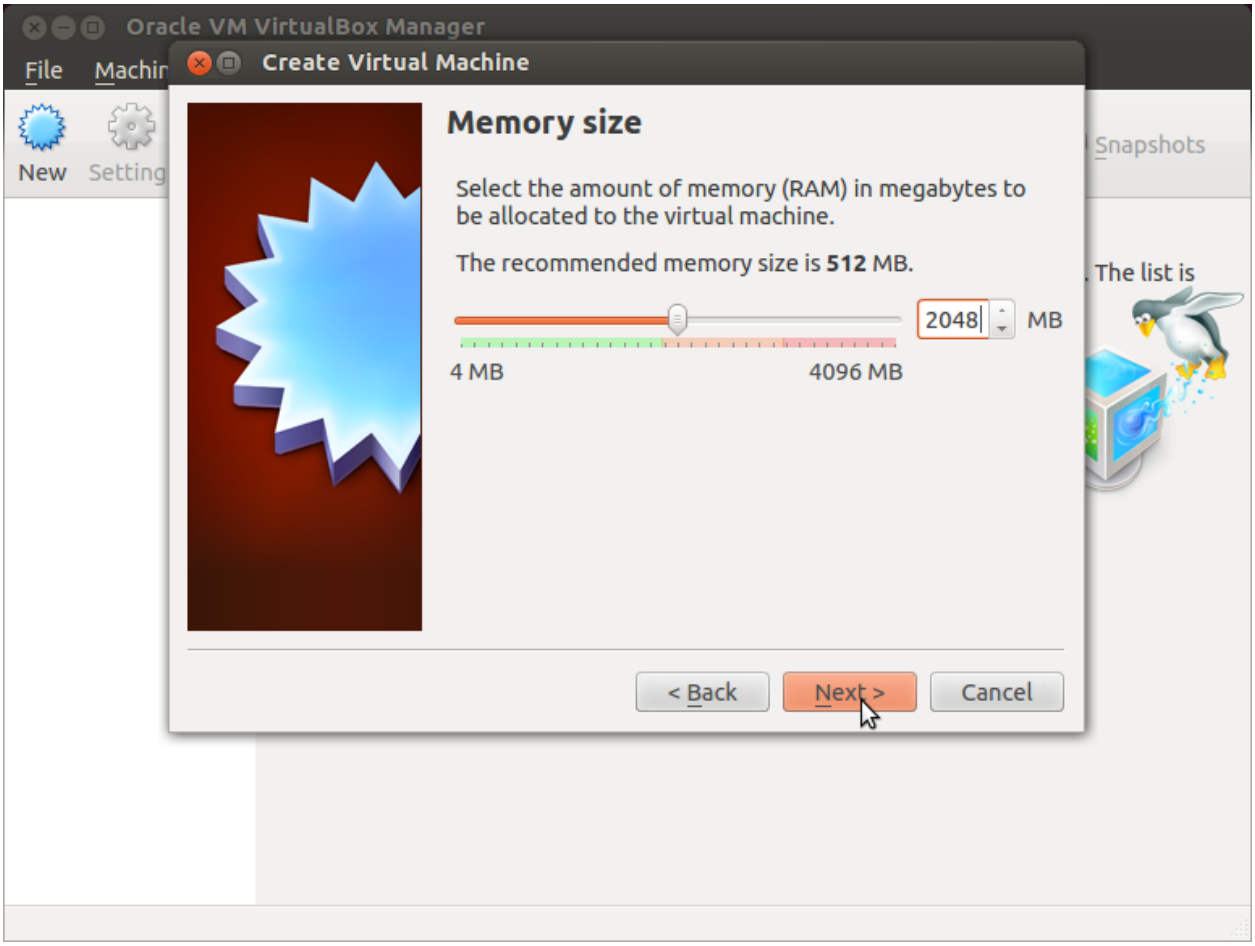
2. Click on *New* button



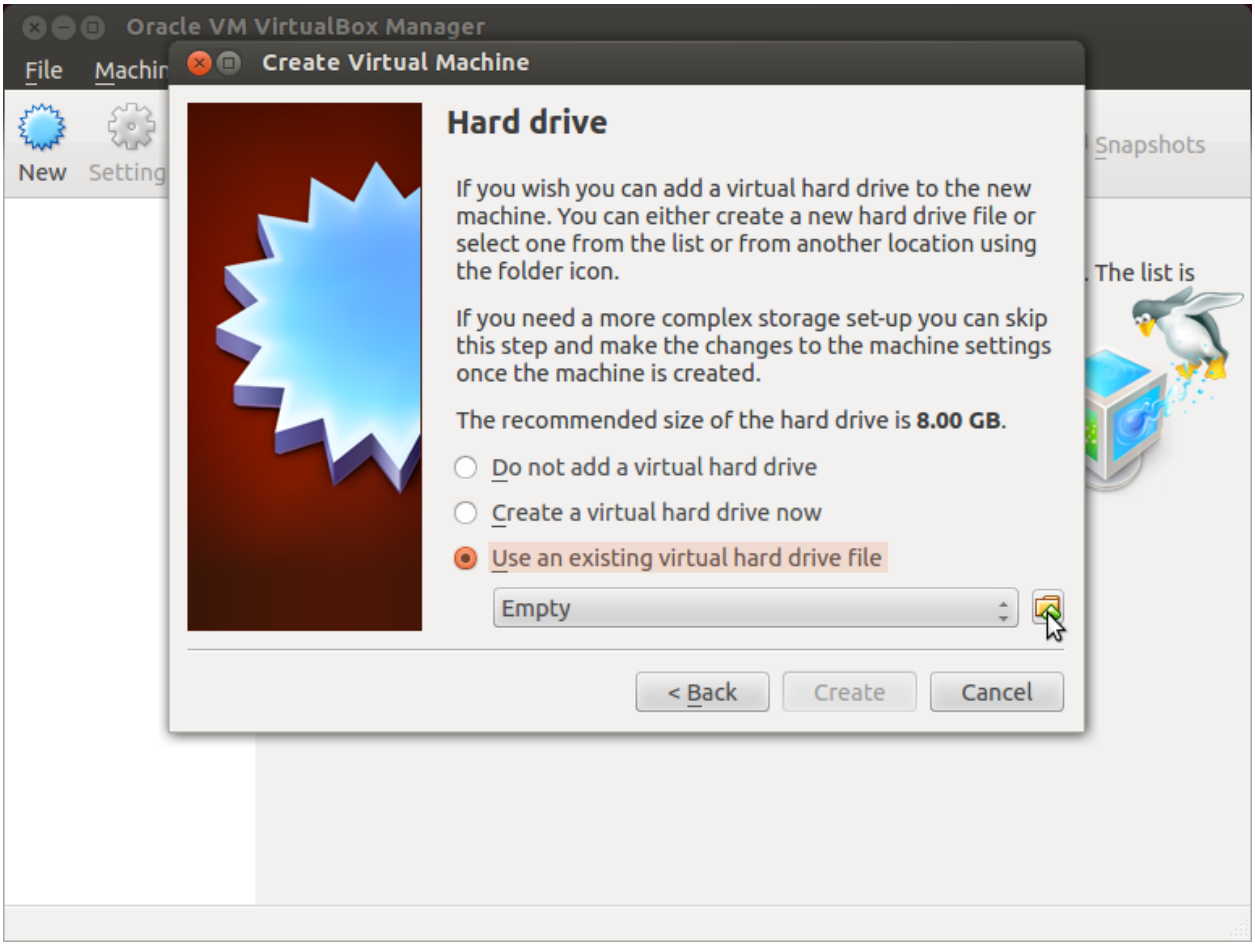
3. Select the name of the virtual machine and the operating system type



4. Select the amount of memory you want to give to your new virtual machine



5. Make the virtual machine use Architech's virtual disk by pointing to the downloaded file. Than click on *Create*.

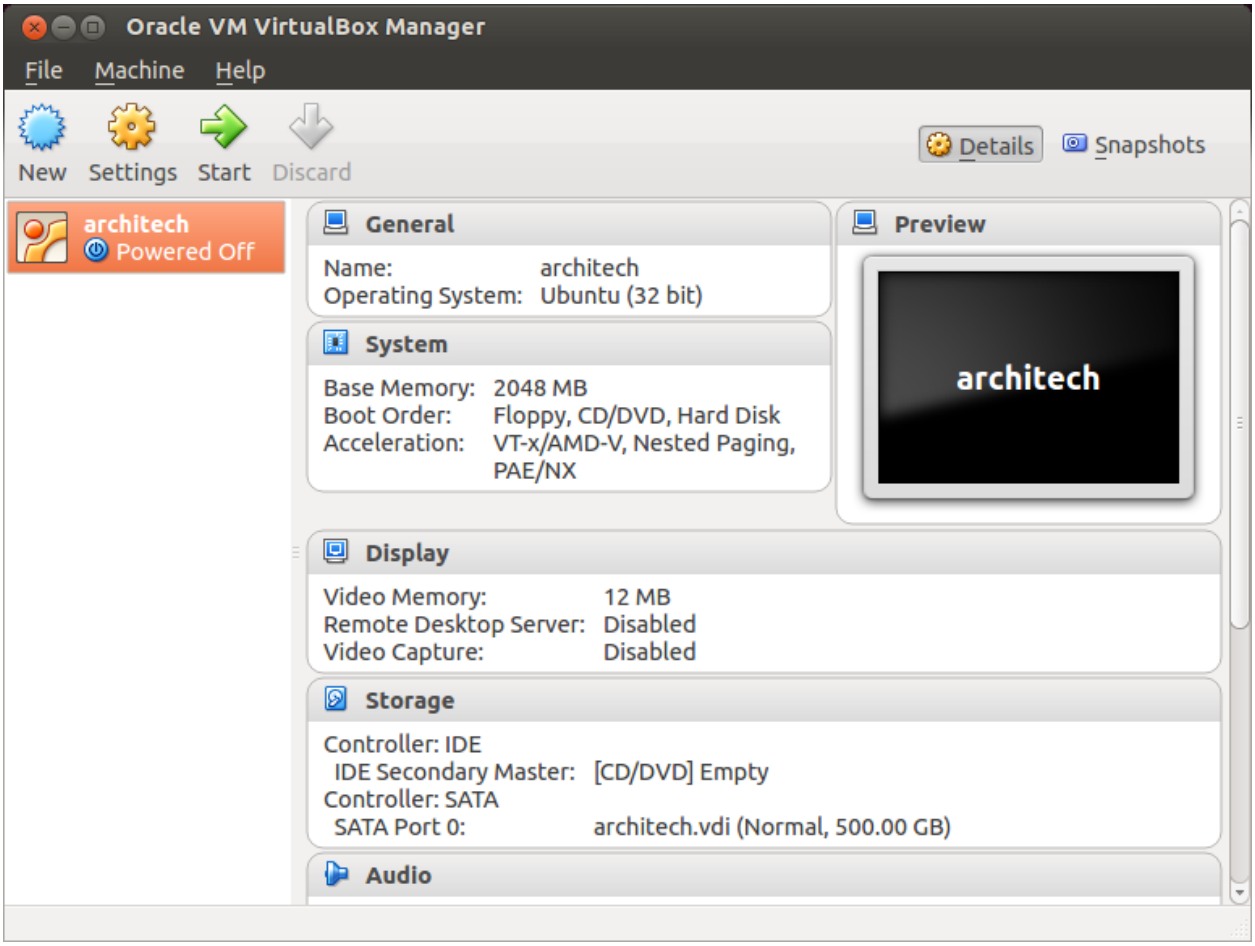


Setup the network

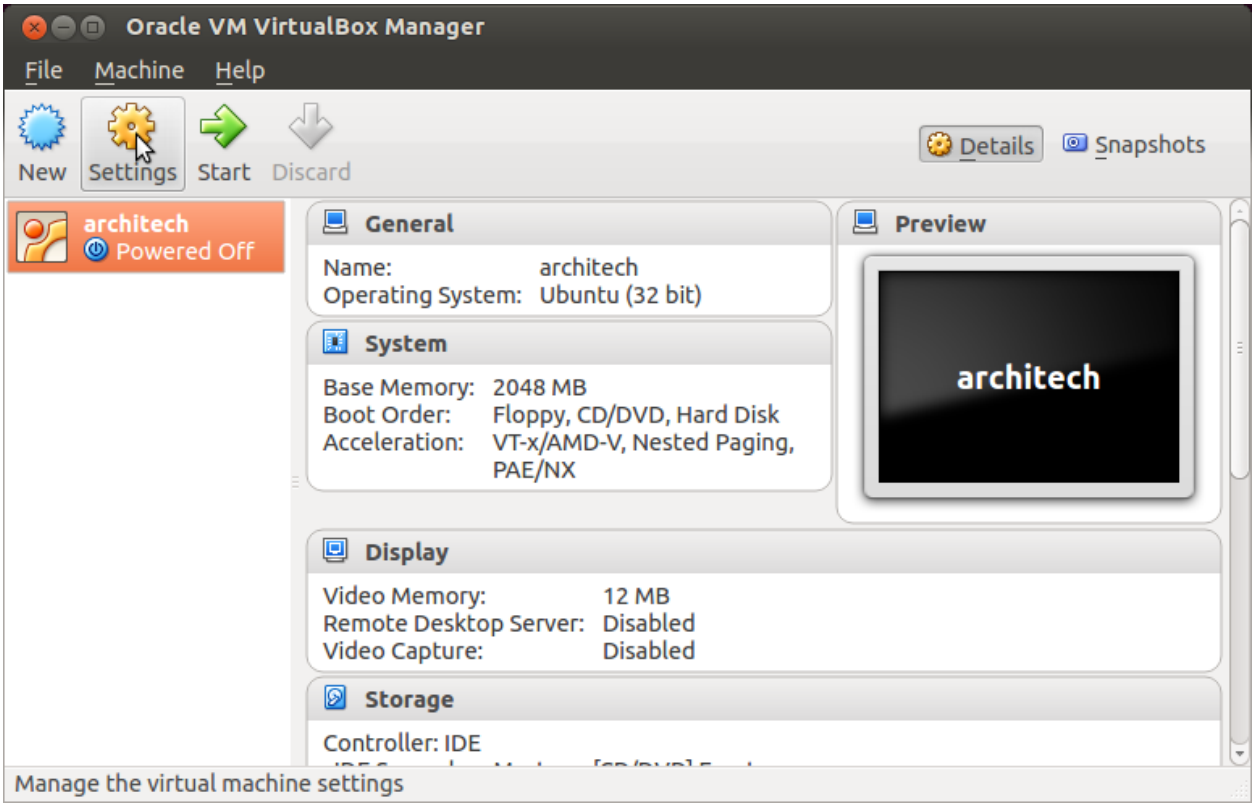
We need to setup a port forwarding rule to let you (later) use the virtual machine as a local repository of packages.

Note: The virtual machine must be off

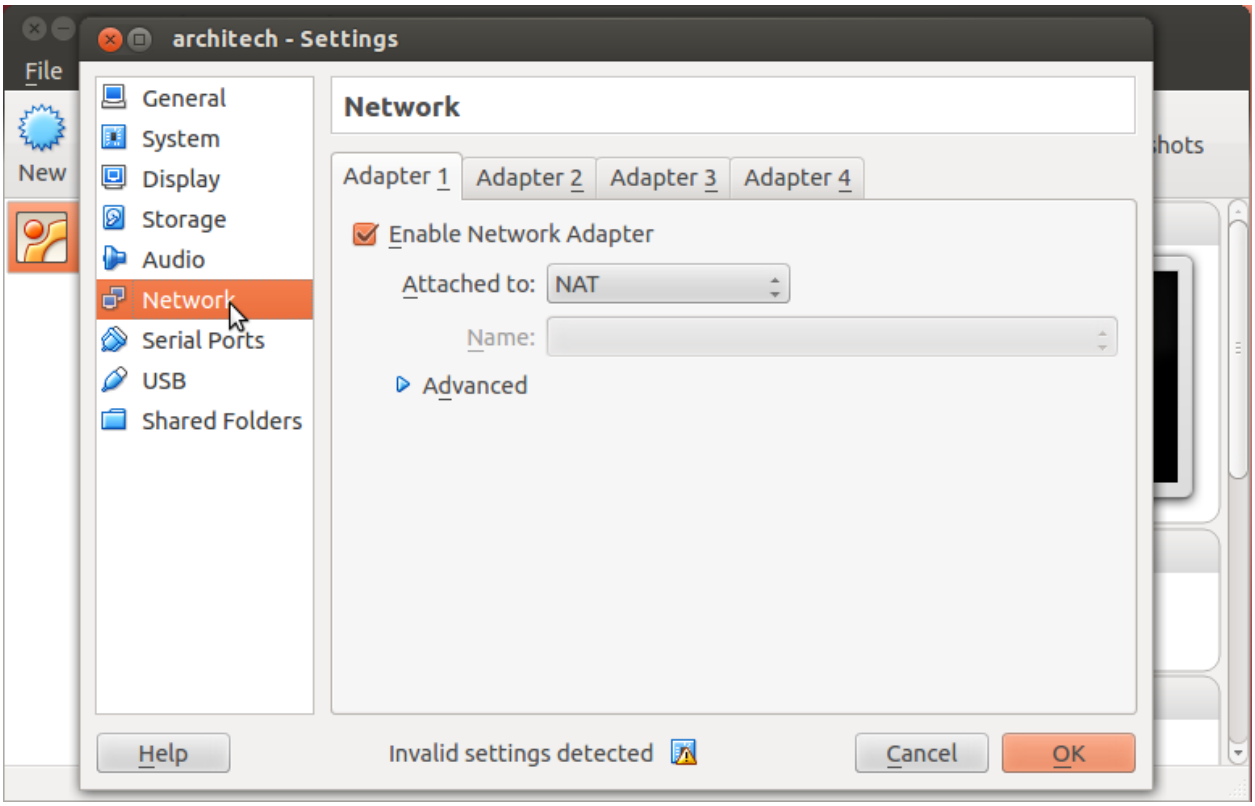
1. Select Architech's virtual machine from the list of virtual machines



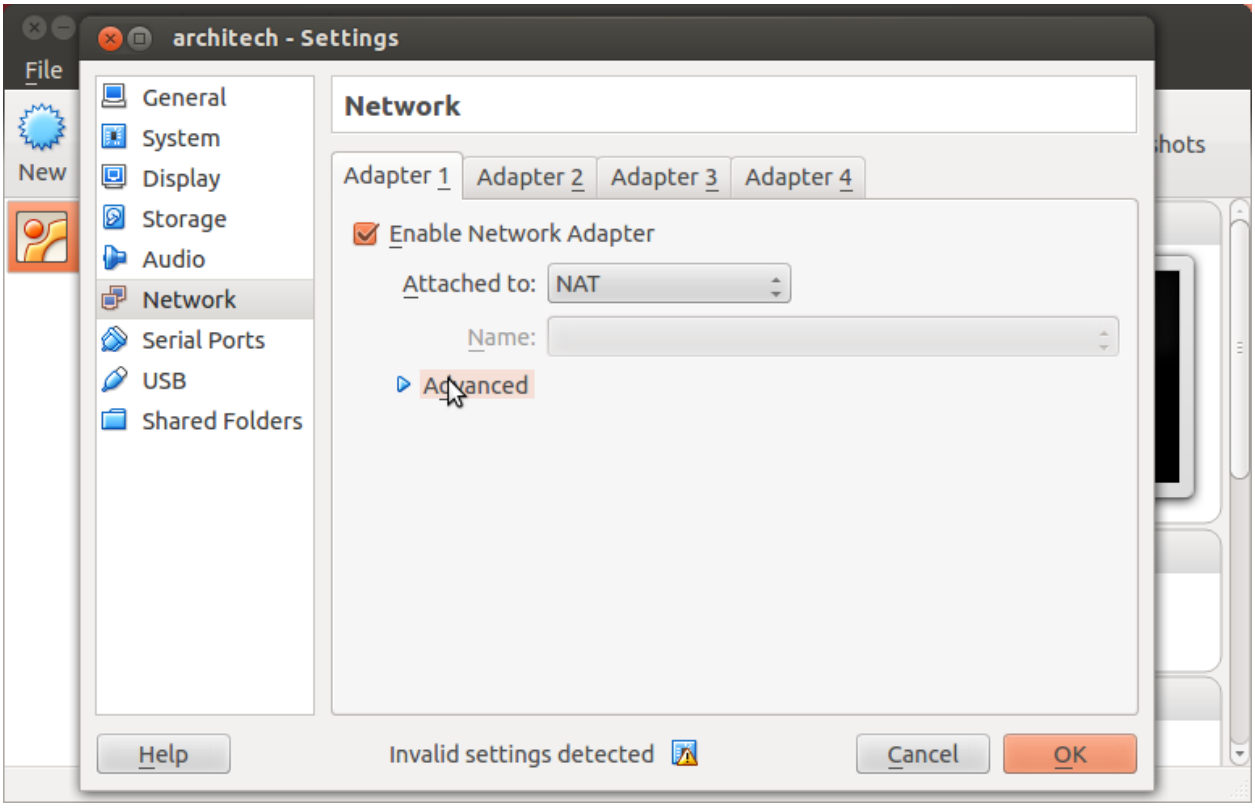
2. Click on *Settings*



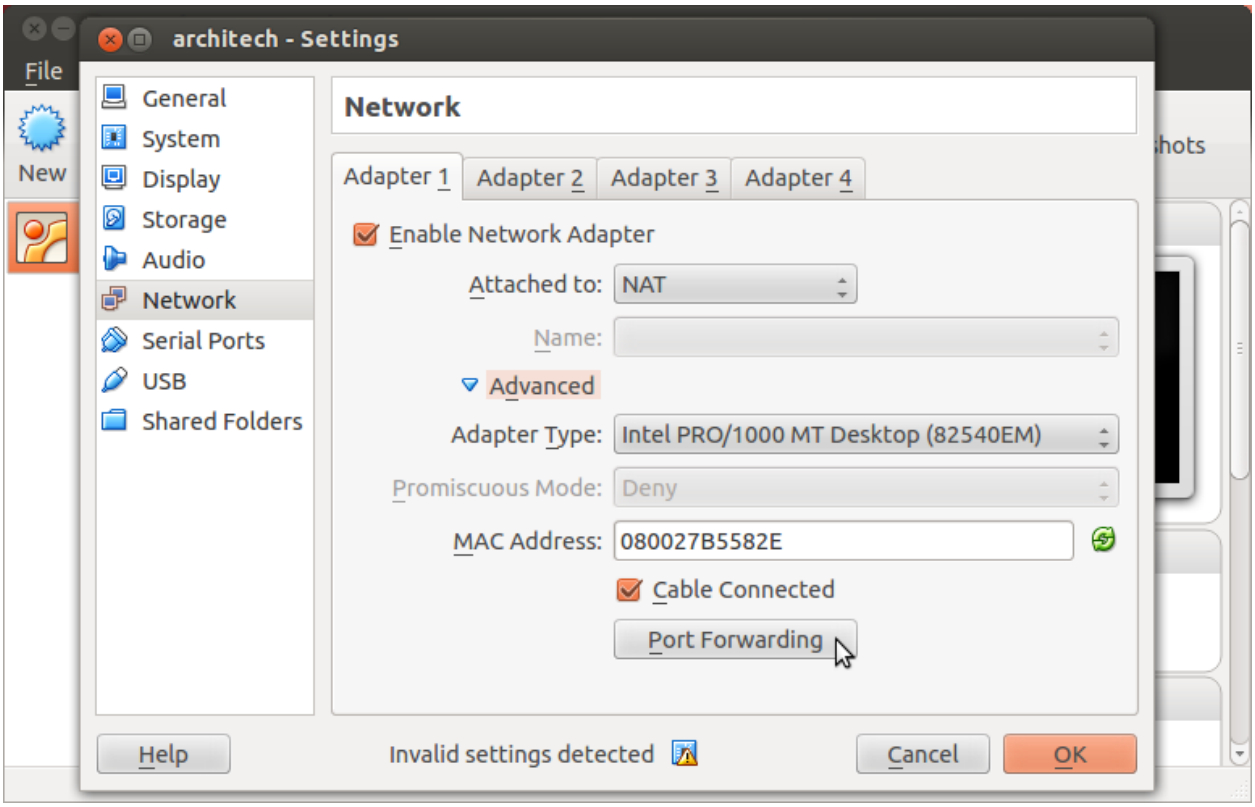
3. Select *Network*



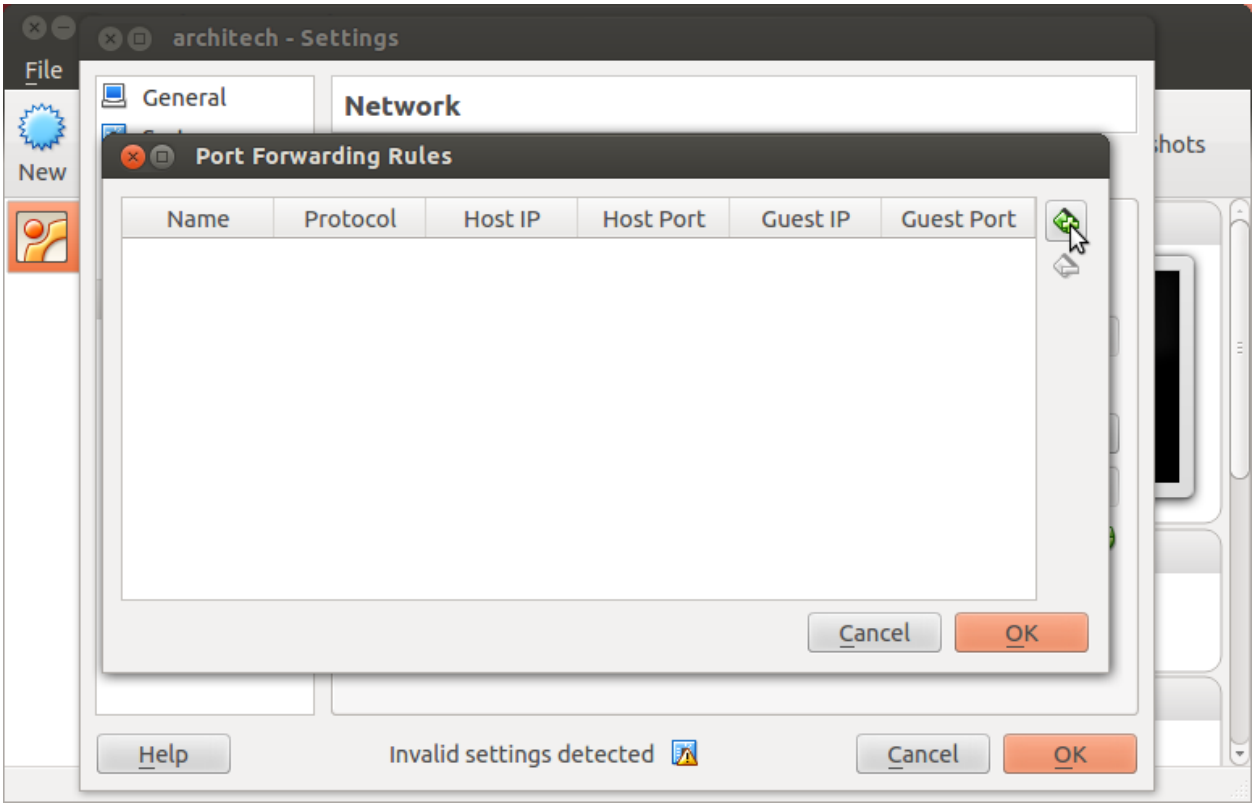
4. Expand *Advanced* of *Adapter 1*



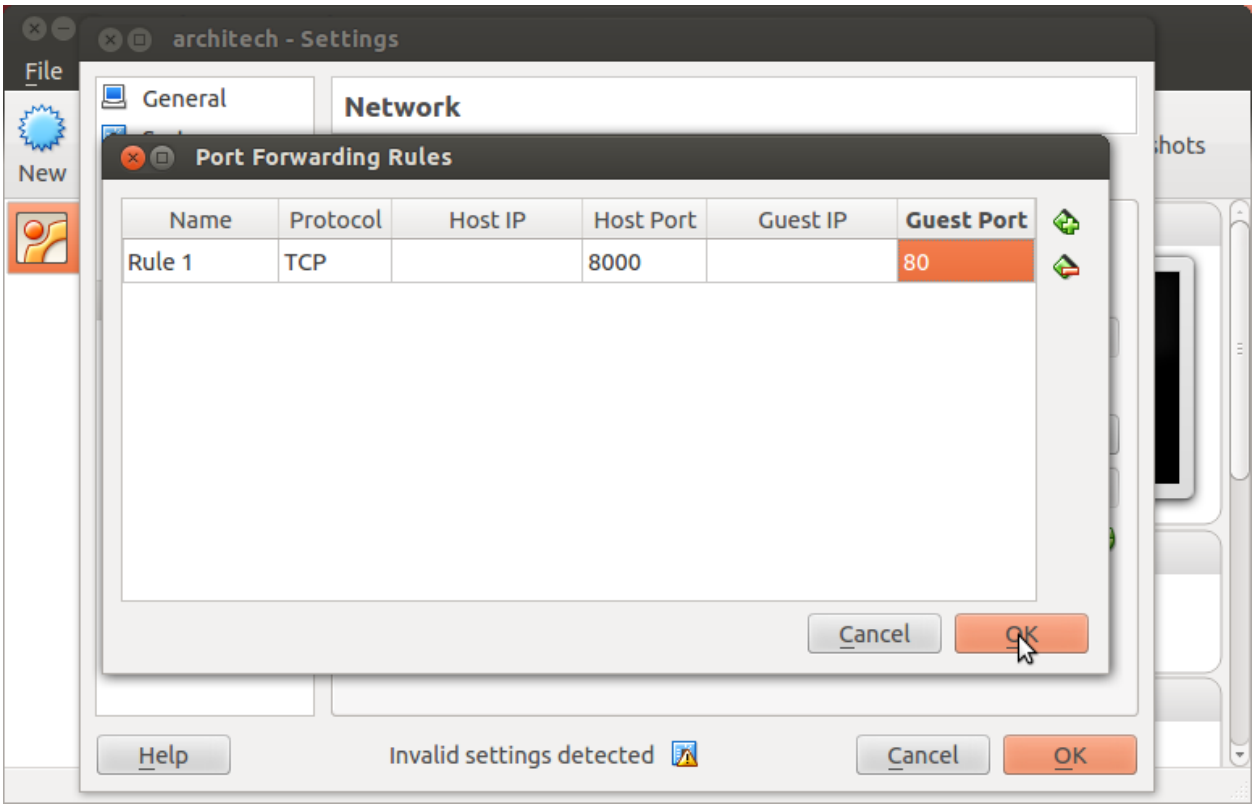
5. Click on *Port Forwarding*



6. Add a new *rule*



7. Configure the *rule*



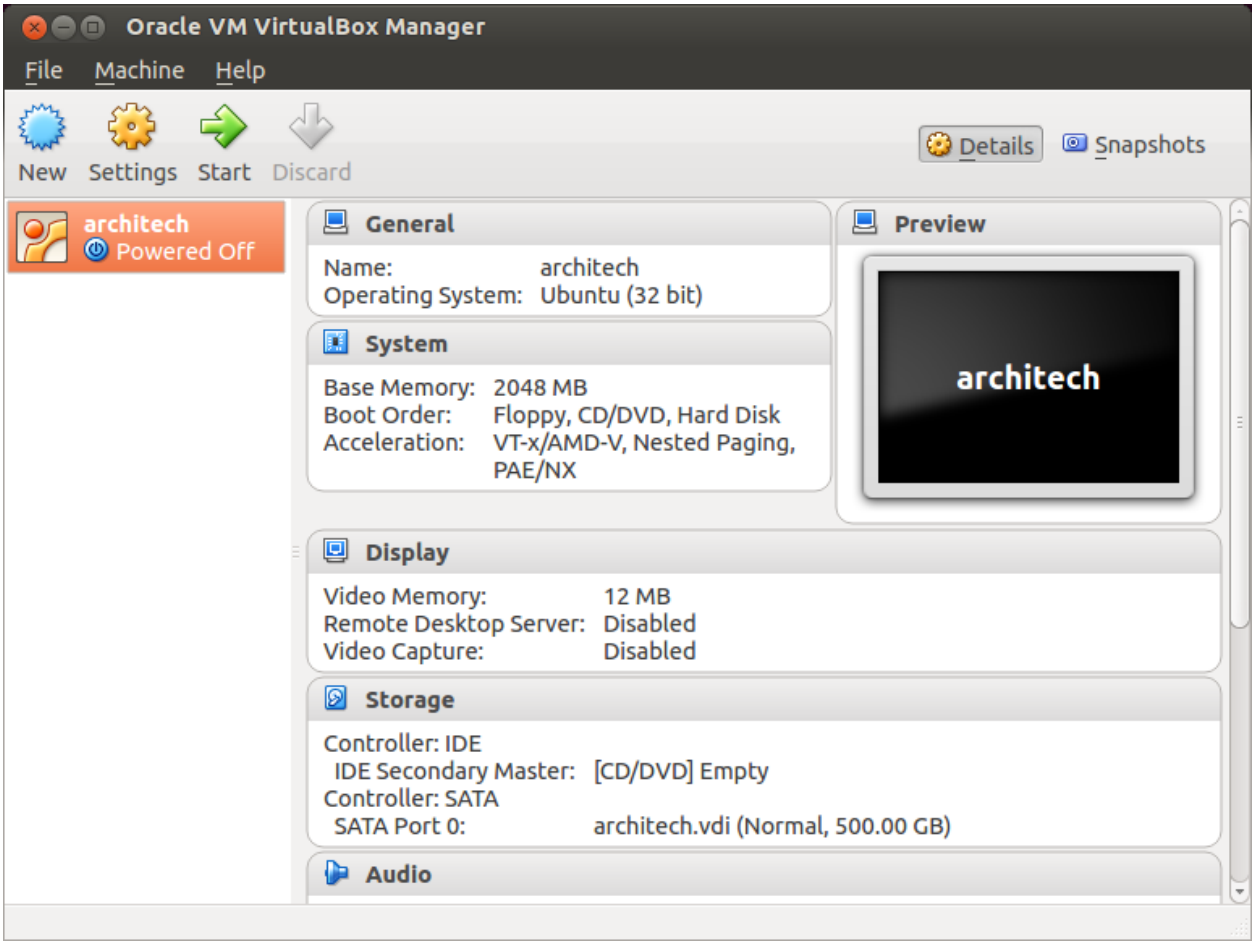
8. Click on *Ok*

Customize the number of processors

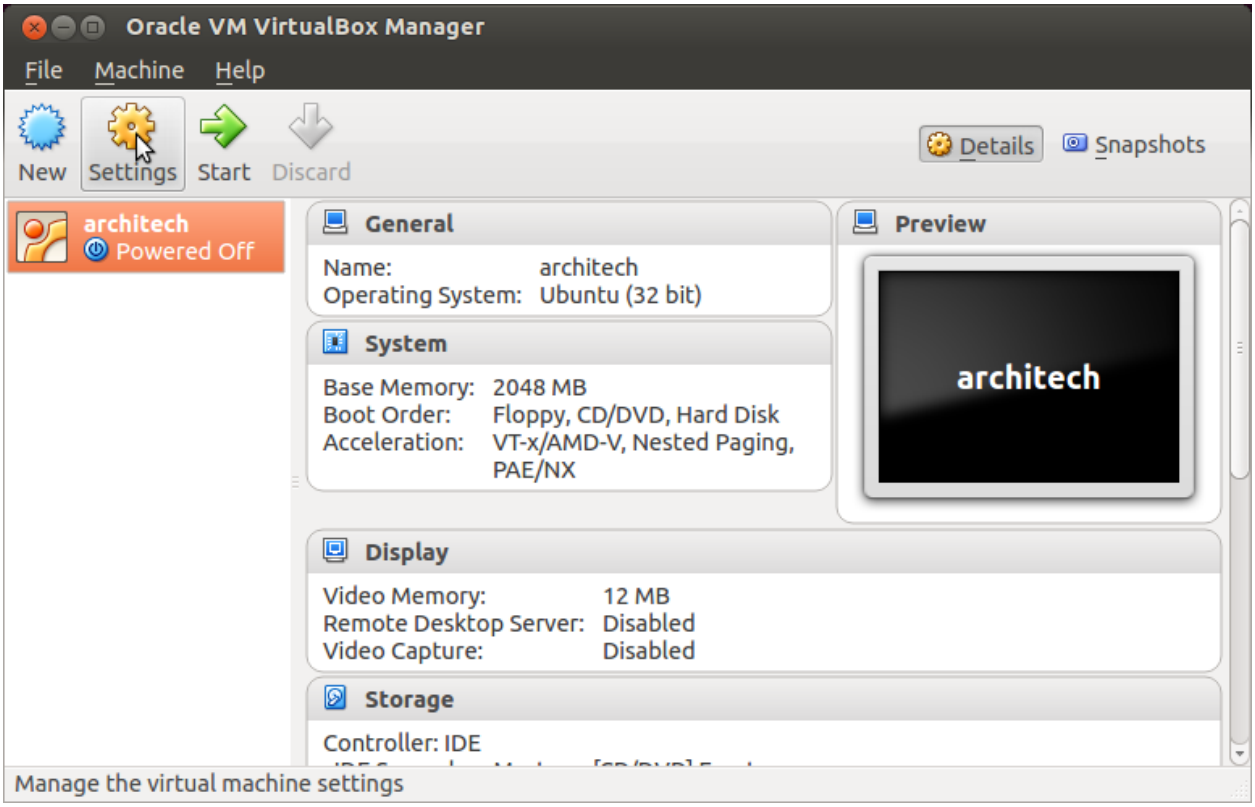
Building an entire system from the ground up is a business that can take up to several hours. To improve the performances of the overall build process, you can, if your computer has enough resources, assign more than one processor to the virtual machine.

Note: The virtual machine must be off

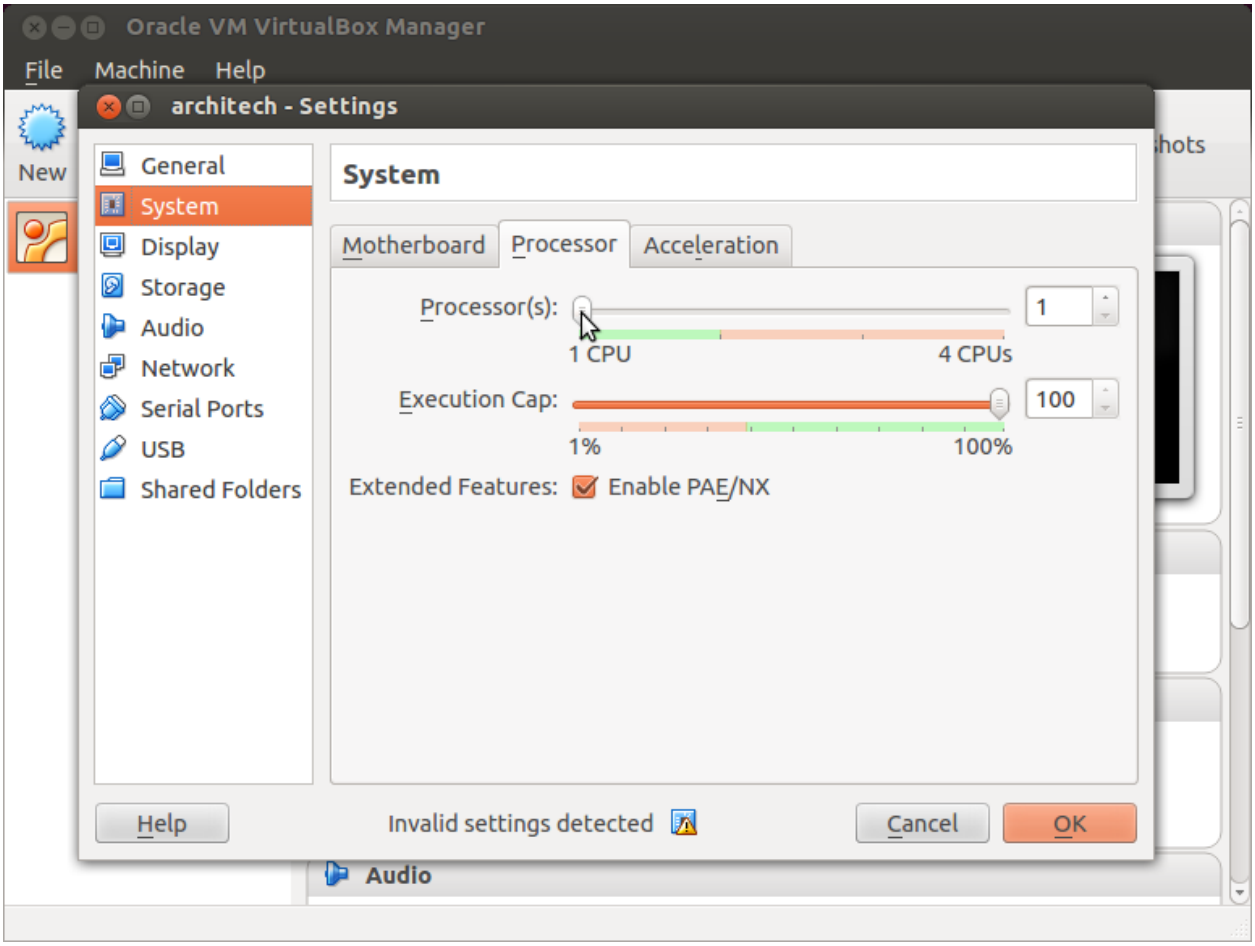
1. Select Architech's virtual machine from the list of virtual machines



2. Click on *Settings*



3. Select *System*
4. Select *Processor*
5. Assign the number of processors you wish to assign to the virtual machine

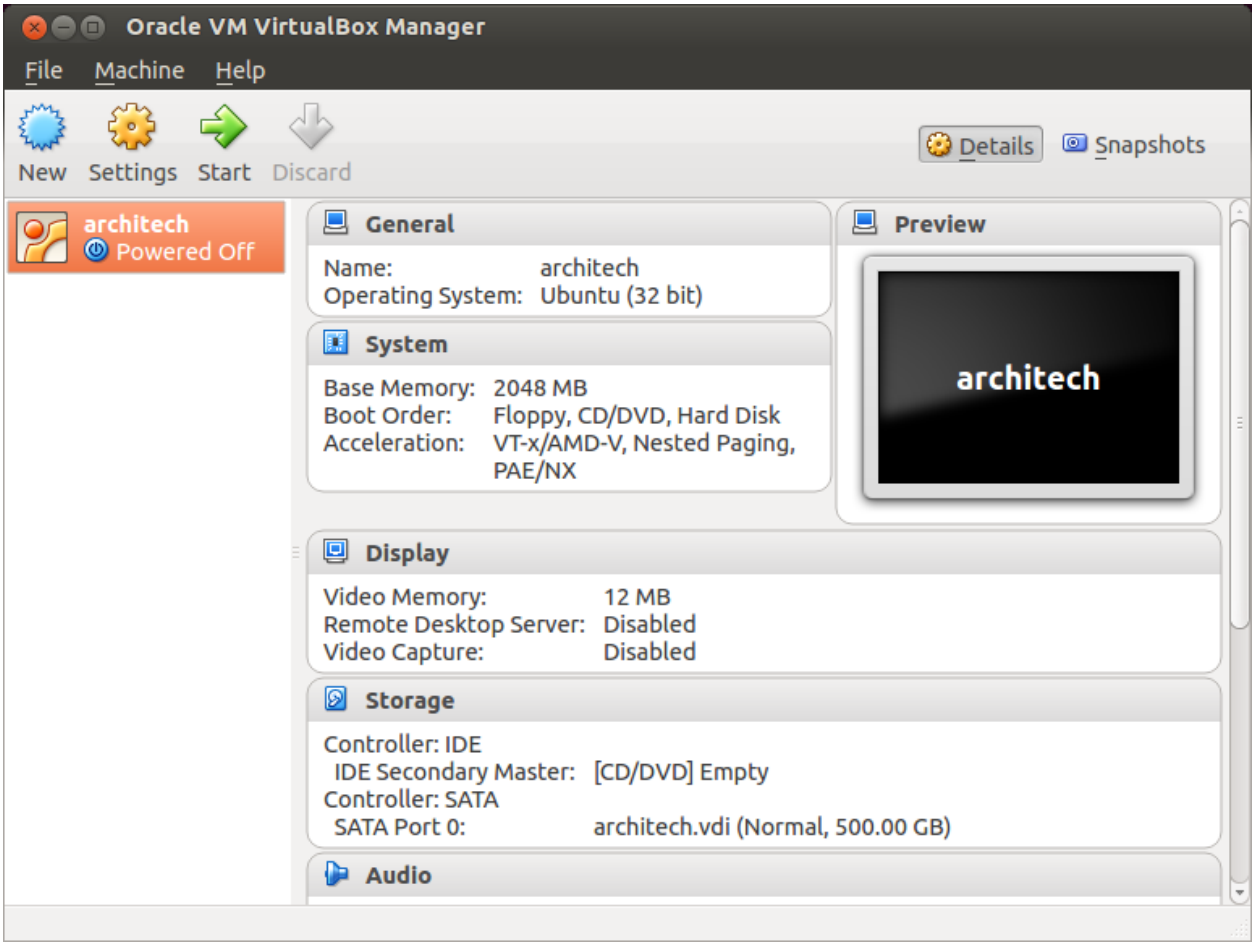


Create a shared folder

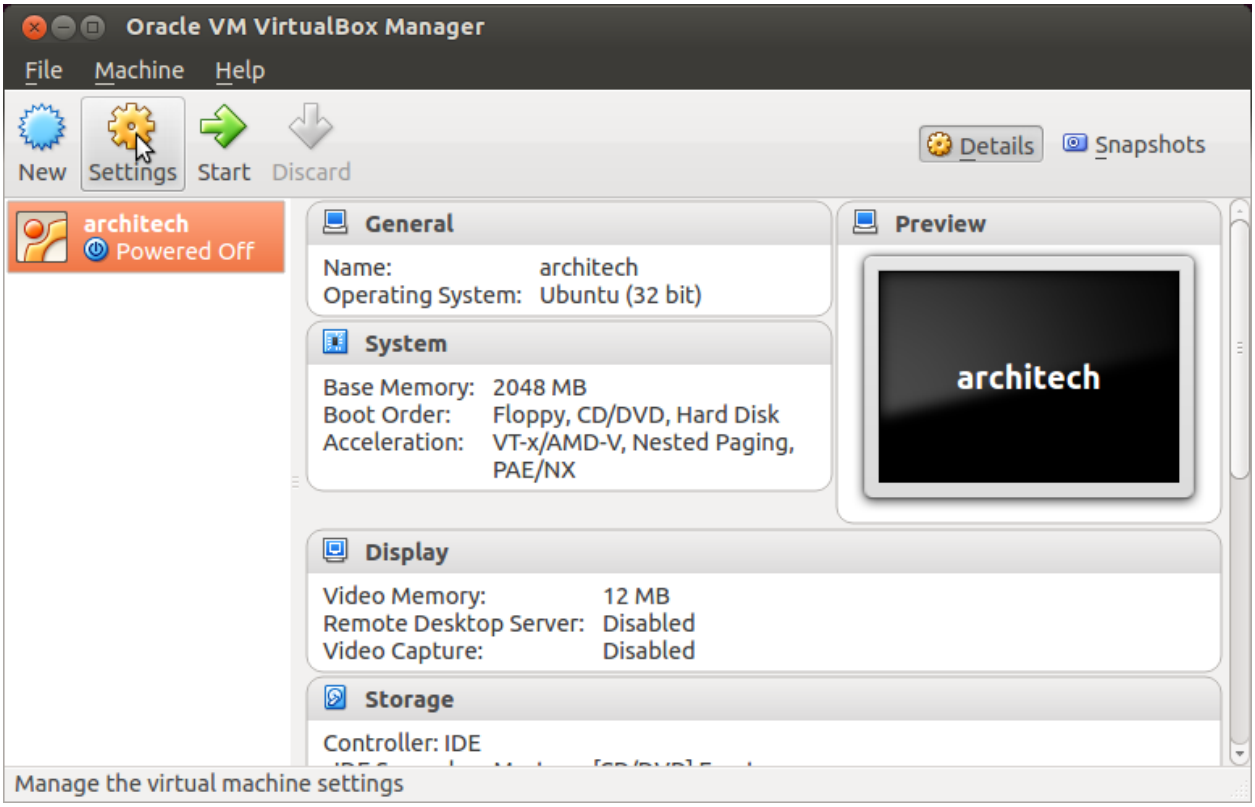
A shared folder is way for host and guest operating systems to exchange files by means of the file system. You need to choose a directory on your host operating system to share with the guest operating system.

Note: The virtual machine must be off

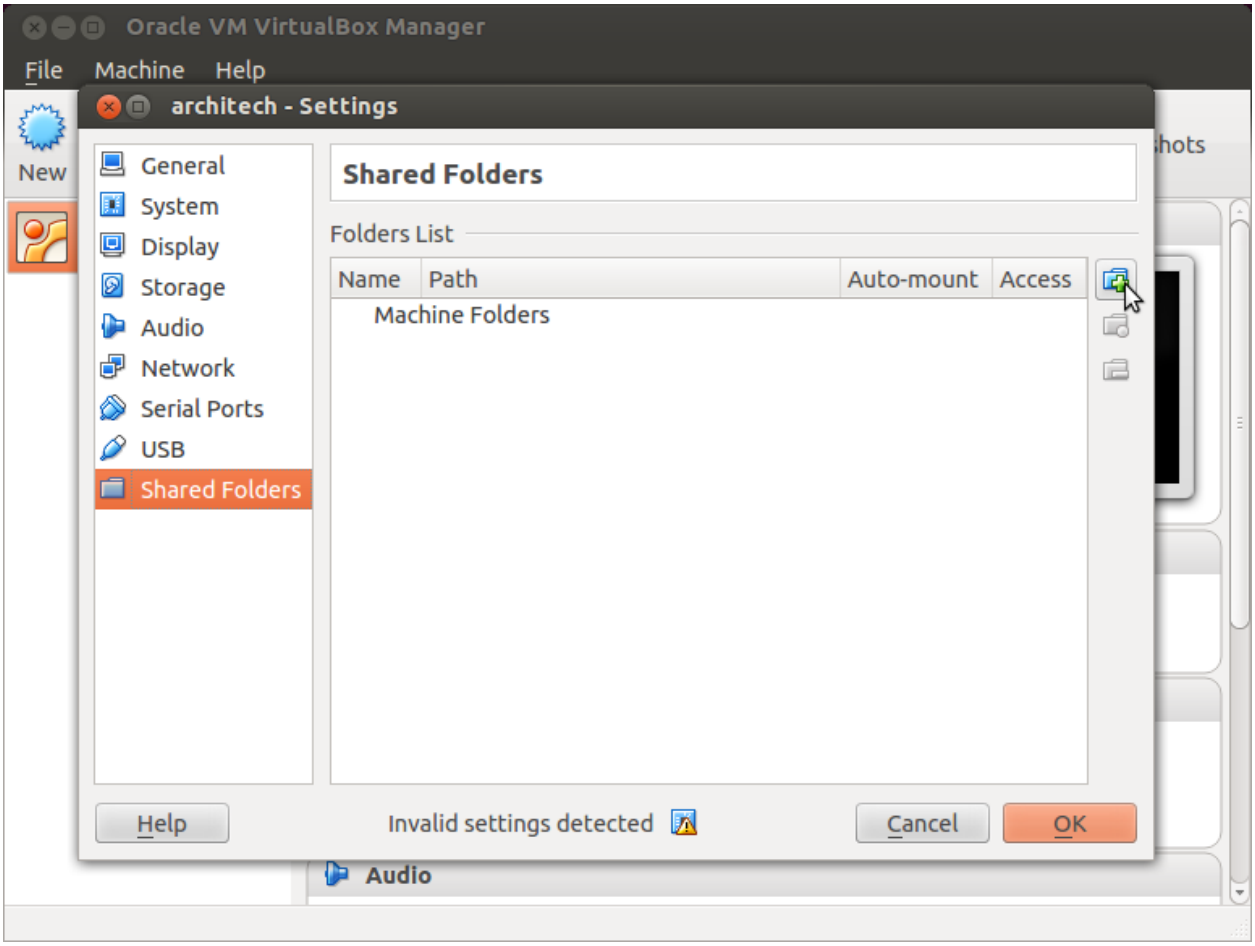
1. Select Architech's virtual machine from the list of virtual machines



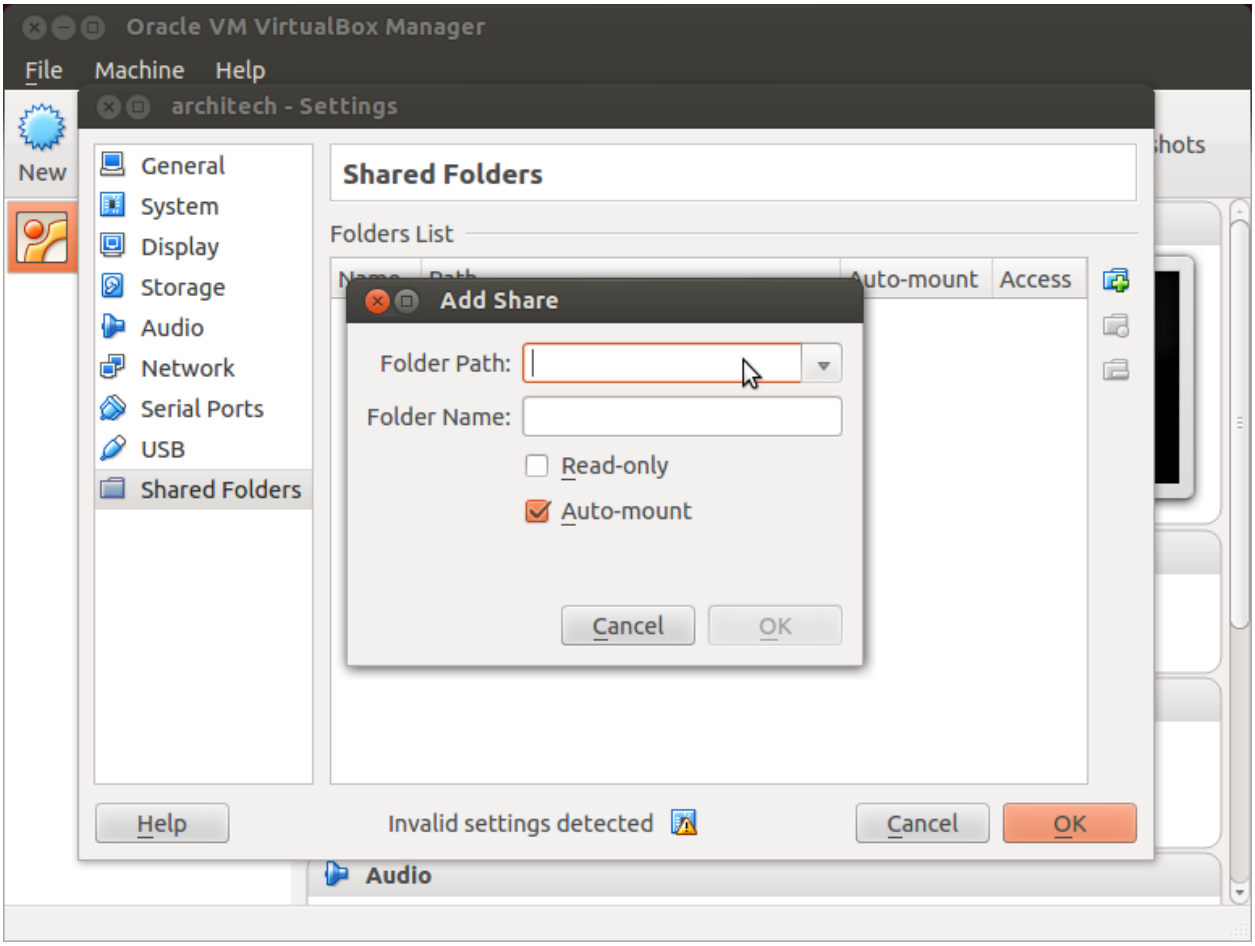
2. Click on *Settings*



3. Select *Shared Folders*
4. Add a new shared folder



5. Choose a directory to share on your host machine. Make sure *Auto-mount* is selected.



Once the virtual machine has been booted, the shared folder will be mounted under */media/* directory inside the virtual machine.

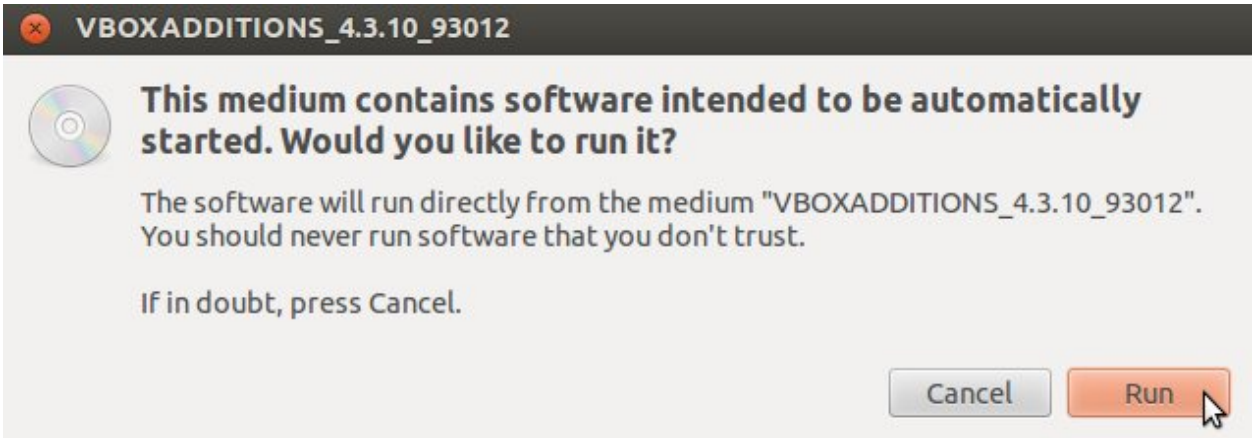
Install VBox Additions

The VBox addictions add functionalities to the virtual machine such as better graphic driver and more. It is already installed in the SDK but is important re-install it to configuring correctly the virtual machine with your operating system.

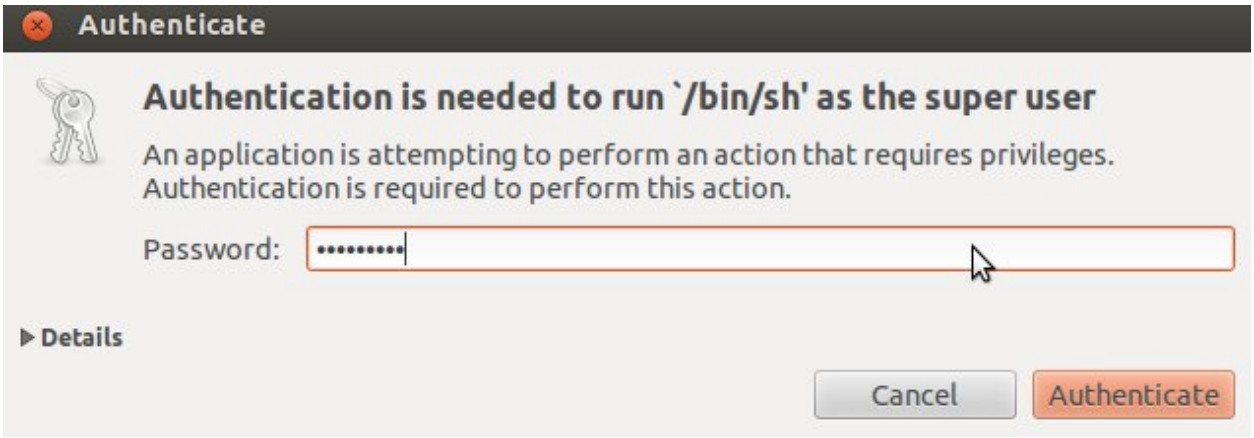
1. Starts the virtual machine



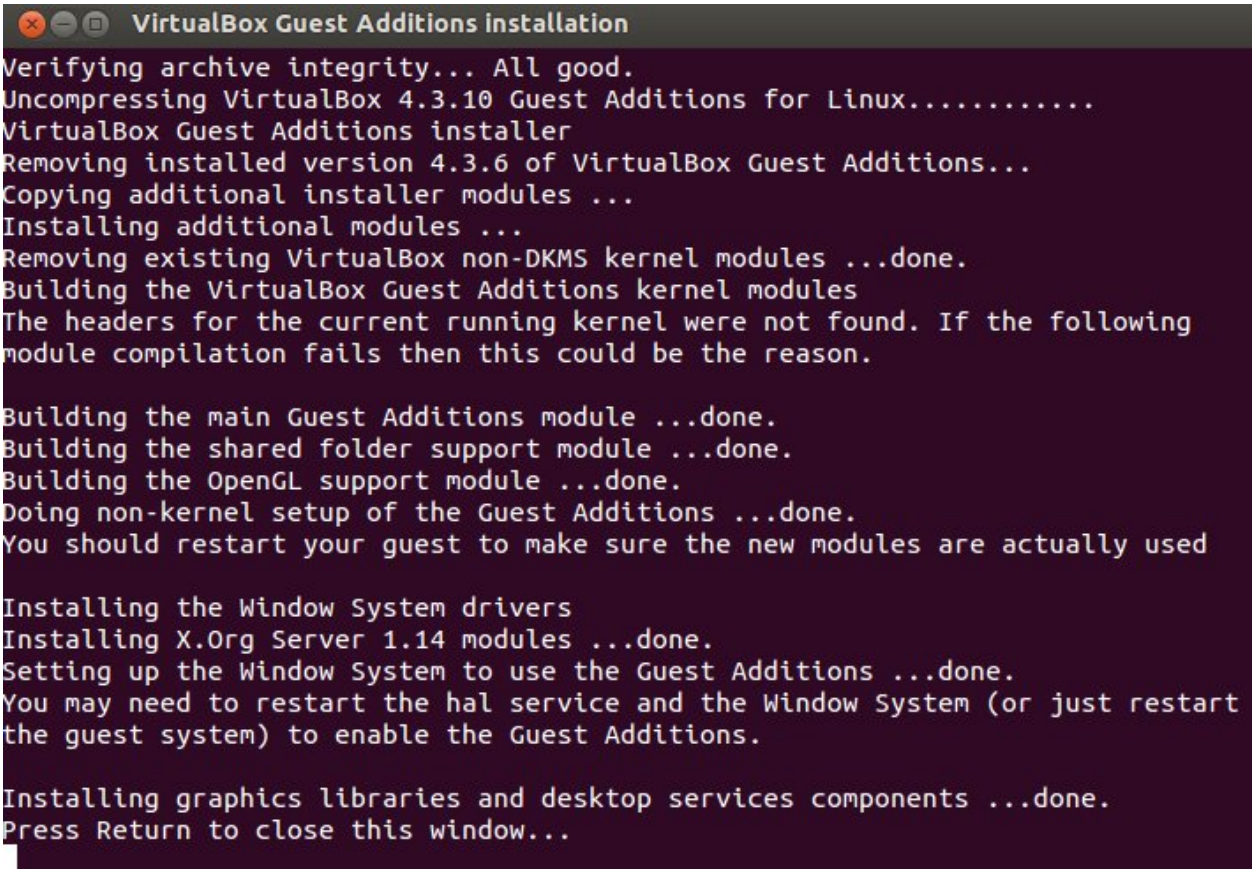
2. Click on the virtual box menu to the voice *Devices* and select *Insert Guest Additions CD Images...*. A message box will appear at the start of the installation, click on *run* button



4. To proceed are required admin privileges, so insert the password *architech* when asked



5. Then a terminal will show the installation progress. When finished, press *Enter* key

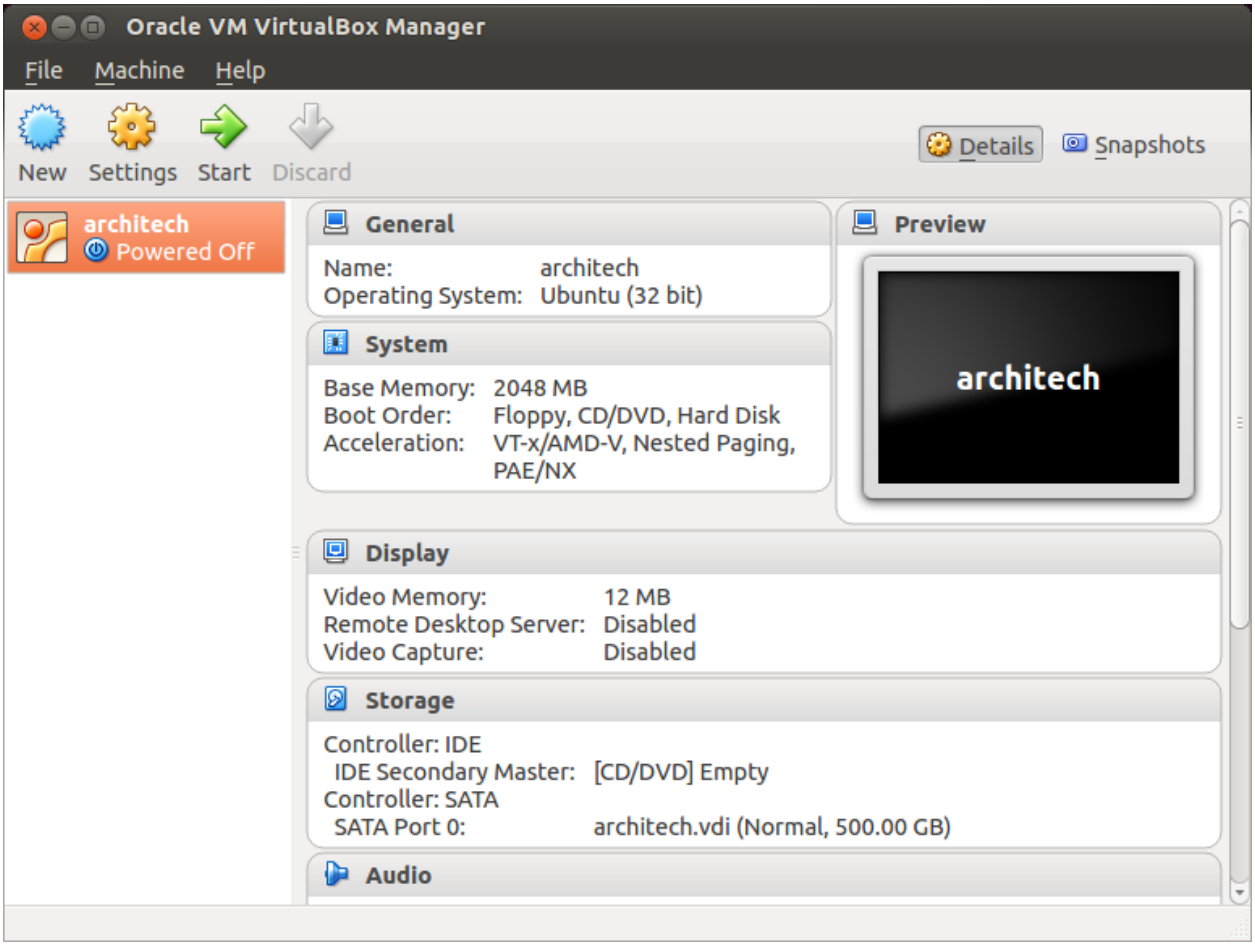


6. Before to use the SDK, it is required reboot the virtual machine

2.2.2 Build

Important: A working internet connection, several GB of free disk space and several hours are required by the build process

1. Select Architech's virtual machine from the list of virtual machines inside Virtual Box application



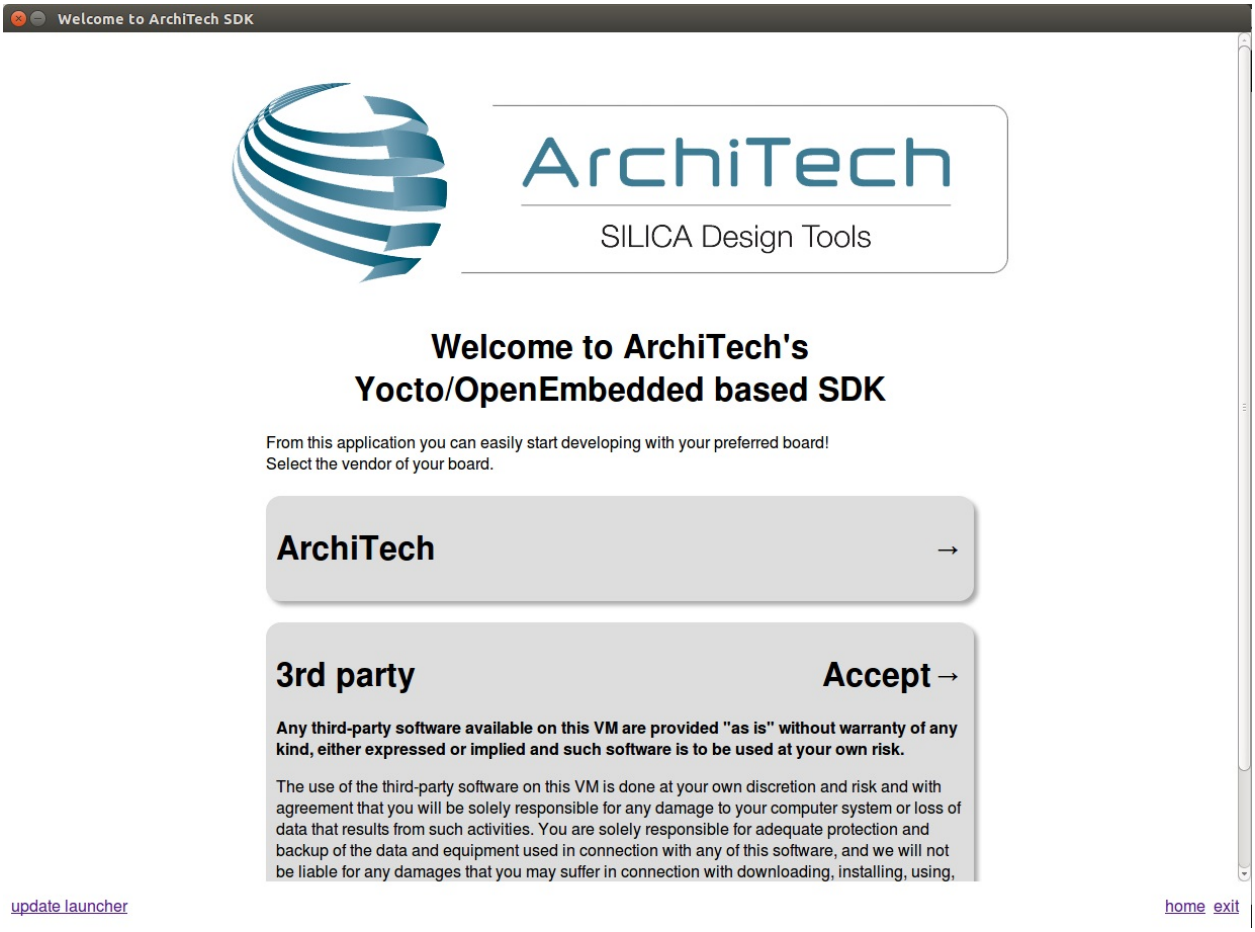
2. Click on the icon *Start* button in the toolbar and wait until the virtual machine is ready



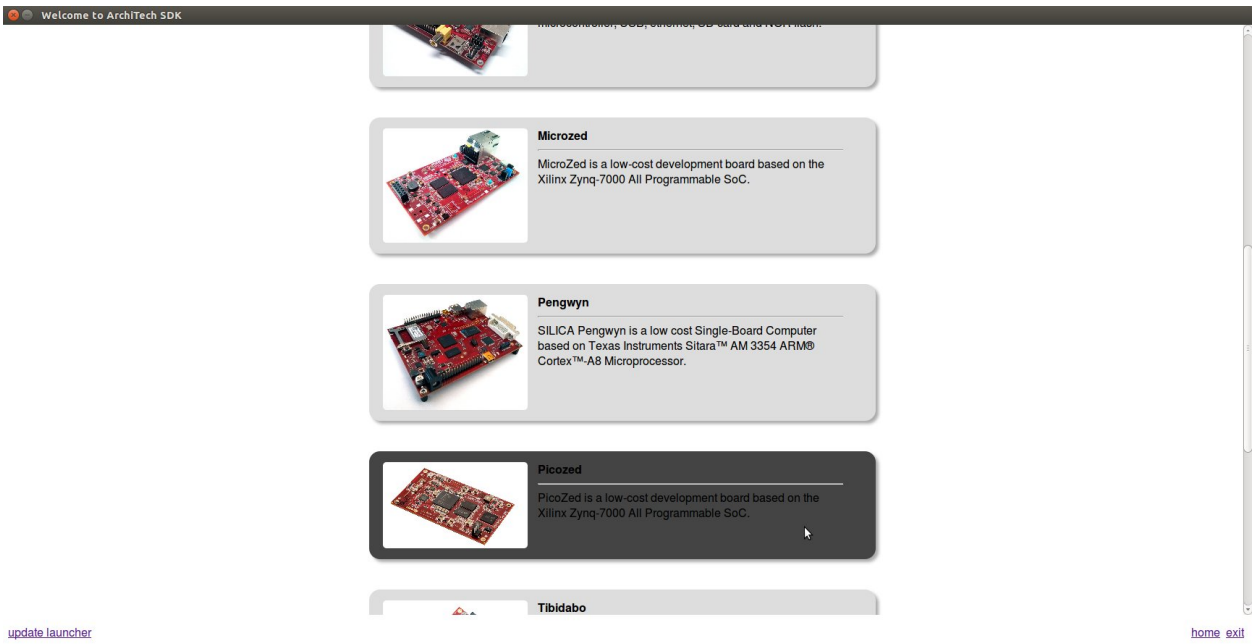
3. Double click on *Architech SDK* icon you have on the virtual machine desktop.



4. The first screen gives you two choices: *ArchiTech* and *3rd Party*. Choose *ArchiTech*.



5. Select Picozed as board you want develop on.



6. A new screen opens up from where you can perform a set of actions. Click on *Run bitbake* to obtain a terminal ready to start to build an image.



7. Open *local.conf* file:
8. Go to the end of the file and add the following lines:

This will trigger the installation of a features set onto the final root file system, like *tcf-agent* and *gdbserver*.

9. Save the file and close gedit.
10. Build *core-image-minimal-dev* image by means of the following command:

At the end of the build process, the image will be saved inside directory:

11. Setup *sysroot* directory on your host machine:

Note: **sudo** password is: "**architech**"

2.2.3 Deploy

> **Warning:** The following instruction will make you overwrite your SD card content, it will be lost forever! If you have important data on it, make sure you do a backup of your data on the SD card before catching up with the next steps.

Create two partitions on the SD card you mean to use to boot the board. The first one has to be a *FAT16* (name it **boot**), 64MB will be more than enough. Create the second partition as an *EXT2* (name it **rootfs**), make it big enough to fill the free space on the disk size.

Run this command:

to create the directory that will be used to save a few files you need to download from the Internet:

- Download file uEnv.txt

And build by yourself the boot.bin file, see xilinx documentation *documentation*. Alternatively in the PicoZed flash is present the u-boot that you can use by setting the SW1 switch (on the Picozed) in this way:



Now, we assume that the first partition of the SD card gets mounted (in your SDK virtual machine) under:

while the second partition gets mounted under:sdcard-jumpers.jpg

> **Warning:** If that's not the case for your configuration, please find out which are the proper mounting points for those two partitions on your system and replace them in the following instructions.

Ok then, we can finally deploy bootloader and kernel on the first partition of the SD card:

and the root file system on the second partition of the SD card:

Important: sudo password is **architech**

Make sure everything has been written on the SD card:

and unmount the SD card from your system.

2.2.4 Boot

In order to use the Picozed you have to plugin in it into the carrier board.



Make sure that Picozed boot mode SW switches are set like in the following picture:

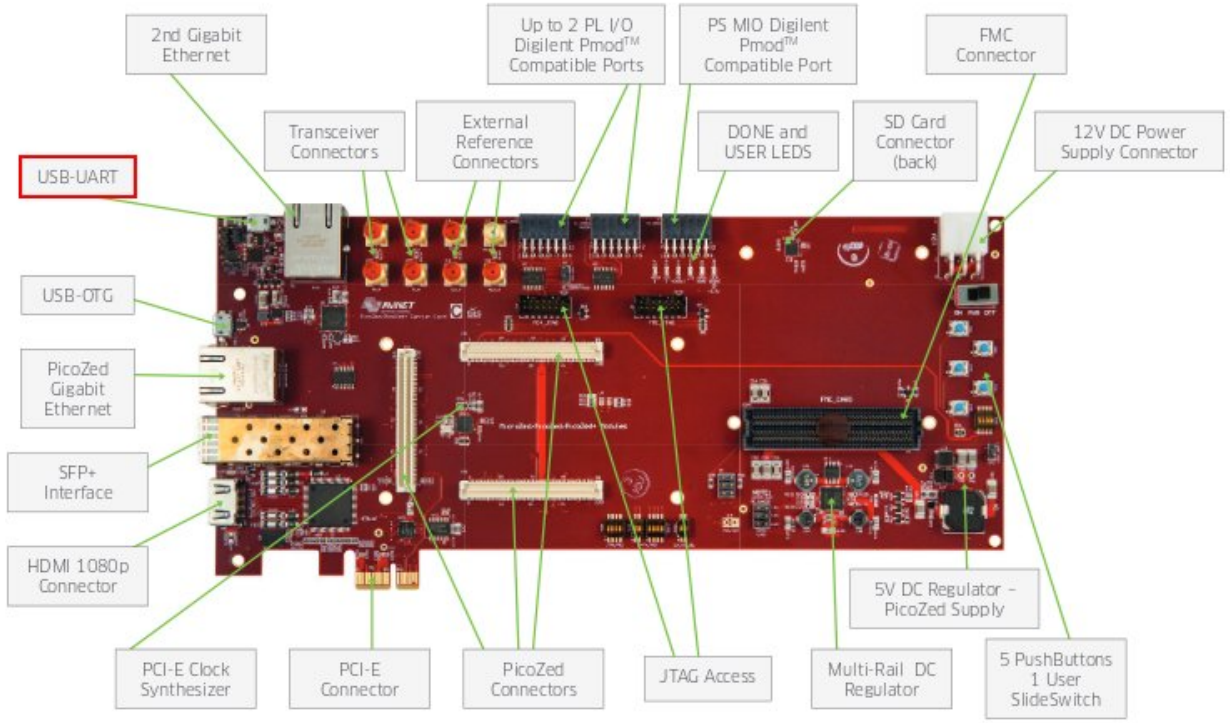


Insert the SD card you just prepared inside socket **J2** of the carrier board.

Power on the board and **AFTER** connect the micro-USB cable from your PC to Picozed carrier board connector **J6**.

And now proceed by setting up the serial console.

On Picozed the USB-UART port is used for serial console.



which you can connect, by means of a micro-USB cable, to your personal computer.

Note: Every operating system has its own killer application to give you a serial terminal interface. In this guide, we are assuming your **host** operating system is **Ubuntu**.

On a Linux (Ubuntu) host machine, the console is seen as a ttyACMX device and you can access to it by means of an application like *minicom*.

Minicom needs to know the name of the serial device. The simplest way for you to discover the name of the device is by looking to the kernel messages, so:

1. clean the kernel messages
2. connect the mini-USB cable to the board already powered-on
3. display the kernel messages
3. read the output

As you can see, here the device has been recognized as **/dev/ttyUSB0**.

Now that you know the device name, run *minicom*:

If minicom is not installed, you can install it with:

then you can setup your port with these parameters:

If on your system the device has not been recognized as */dev/ttyUSB0*, just replace */dev/ttyUSB0* with the proper device.

Once you are done configuring the serial port, you are back to *minicom* main menu and you can select *exit*.

Warning: The next procedure changes the boot mode of the u-boot. This step is necessary in order to use the files built by Yocto but could make not usable others sd-card built with other SDK.

Before starting to work with the SD-Card built by SDK, it is necessary modify u-boot environment variables. Turn on the board and before the countdown ends press *Enter* key to stop autoboot. Will be shown the u-boot prompt, then insert the following commands:

Note:

After the first run of picozed you have to stop the boot and give only the third command: **run uenvboot**

Give *root* to the login prompt:

and press *Enter*.

Note: Sometimes, the time you spend setting up minicom makes you miss all the output that leads to the login and you see just a black screen, press *Enter* then to get the login prompt.

2.2.5 Code

The time to create a simple *HelloWorld!* application using **Eclipse** has come.

Note:

Before to start remember to copy the cross-toolchain libreries to sysroot

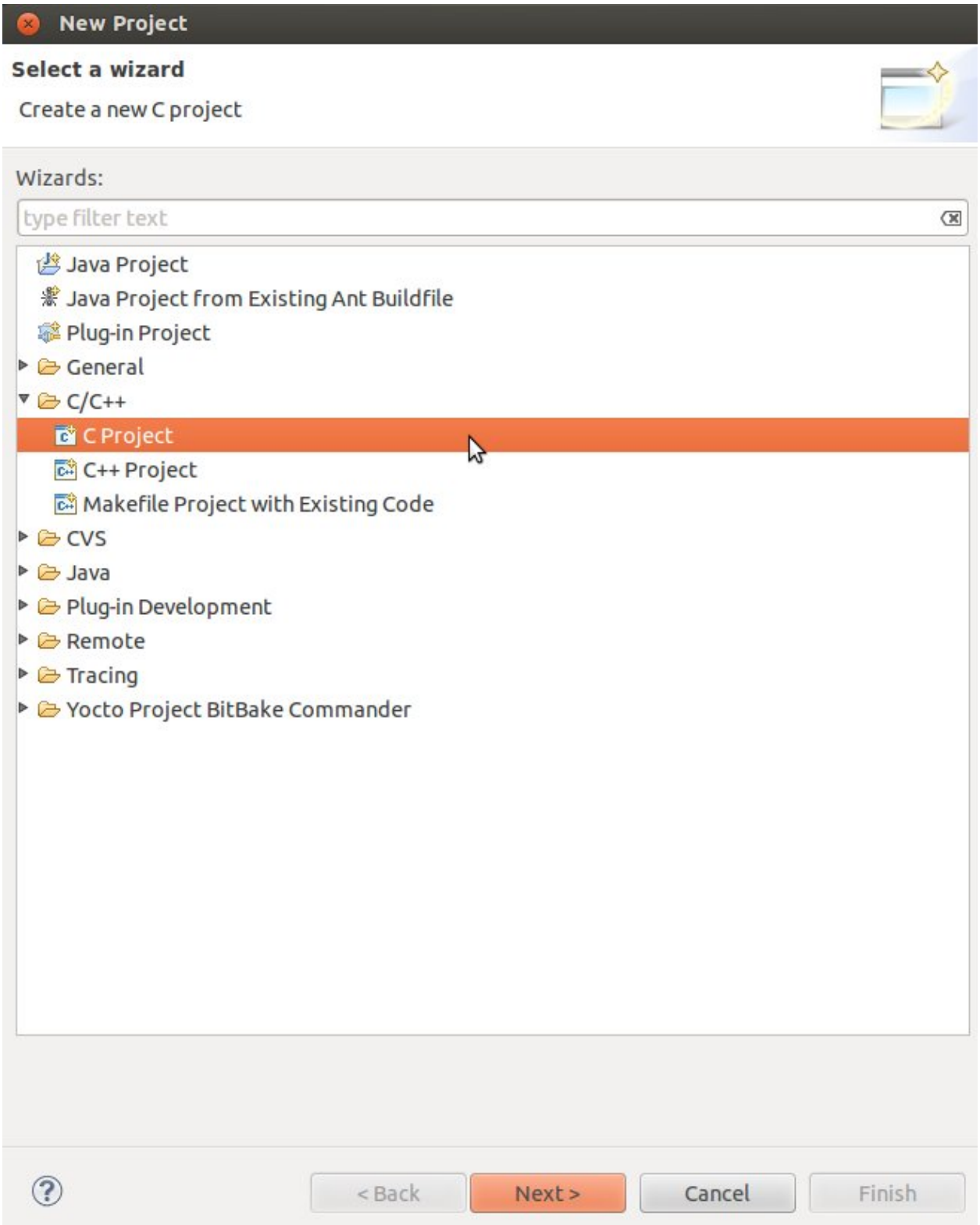
sudo cp -r
/home/architech/architech_sdk/architech/picozed/toolchain/sysroots/cortexa9-vfp-neon-poky-linux-gnueabi/*
/home/architech/architech_sdk/architech/picozed/sysroot

---

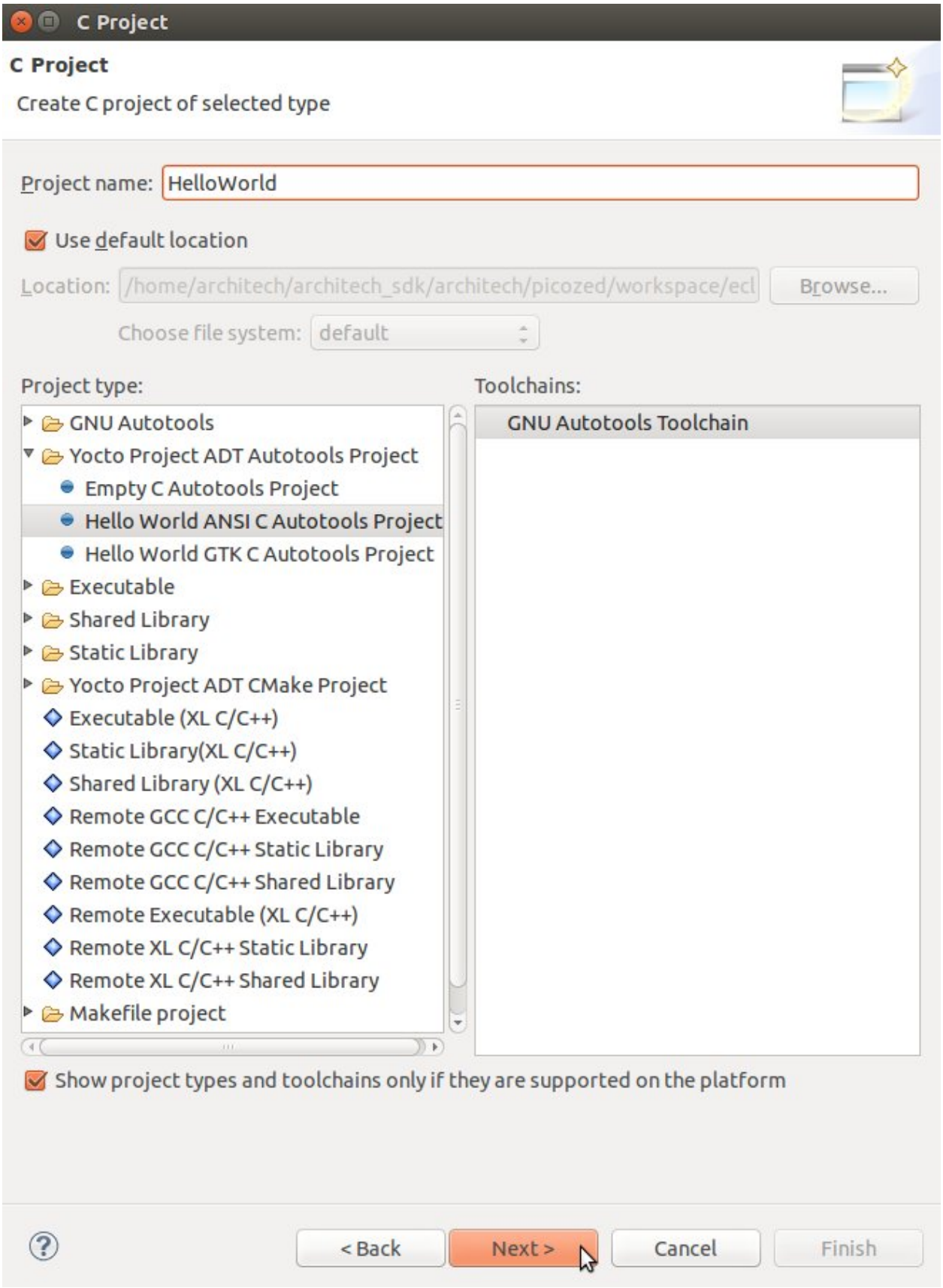
1. Return to the **Splashscreen**, which we left on Picozed board screen, and click on *Develop with Eclipse*.



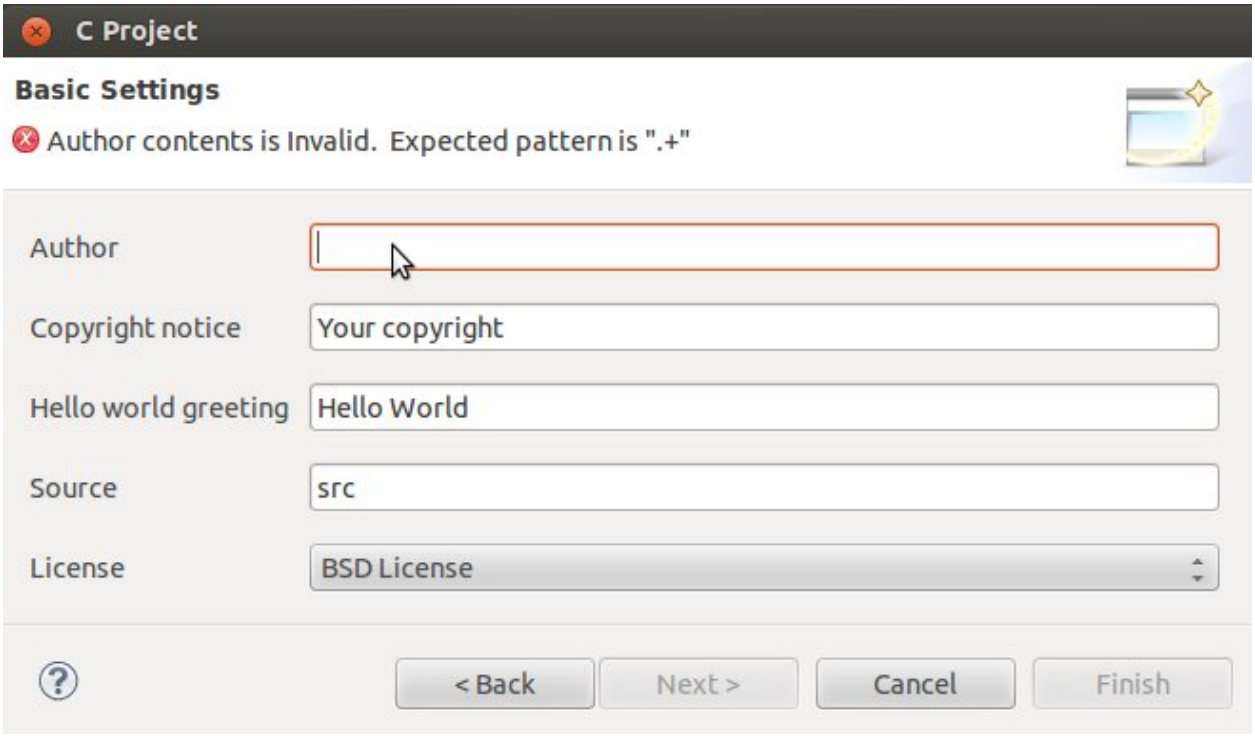
2. Go to *File→ New→ Project...*, in the node "C/C++" select *C Project* and press *next* button.



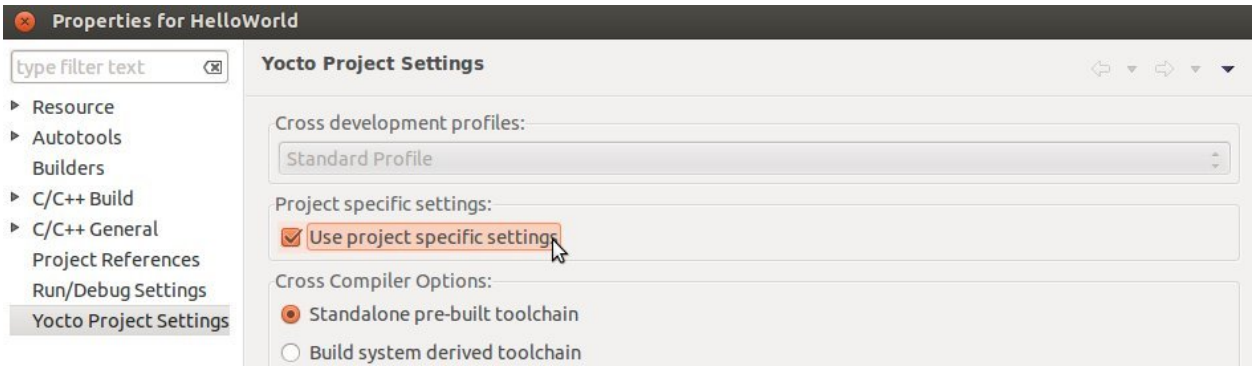
3. Insert *HelloWorld* as project name, open the node *Yocto Project ADT Autotools Project* and select *Hello World ANSI C Autotools Project* and press *next* button.


C Project
C Project
Create C project of selected type
Project name: HelloWorld
Use default location
Location: /home/architech/architech_sdk/architech/picozed/workspace/ecl
Browse...
Choose file system: default
Project type:
GNU Autotools
Yocto Project ADT Autotools Project
Empty C Autotools Project
Hello World ANSI C Autotools Project
Hello World GTK C Autotools Project
Executable
Shared Library
Static Library
Yocto Project ADT CMake Project
Executable (XL C/C++)
Static Library(XL C/C++)
Shared Library (XL C/C++)
Remote GCC C/C++ Executable
Remote GCC C/C++ Static Library
Remote GCC C/C++ Shared Library
Remote Executable (XL C/C++)
Remote XL C/C++ Static Library
Remote XL C/C++ Shared Library
Makefile project
Toolchains:
GNU Autotools Toolchain
Show project types and toolchains only if they are supported on the platform
< Back
Next >
Cancel
Finish

4. Insert *Author* field and click on *Finish* button. Select *Yes* on the *Open Associated Perspective?* question.



5. Open the windows properties clicking on *Project→ Properties* and select *Yocto Project Settings*. Check *Use project specific settings* in order to use the pengwyn cross-toolchain.



5. Click on *OK* button and build the project by selecting *Project→ Build All*.

2.2.6 Debug

Use an ethernet cable to connect the board (connector J11) to your PC. Configure your workstation ip address as 192.168.0.100. Make sure the board can be seen by your host machine:

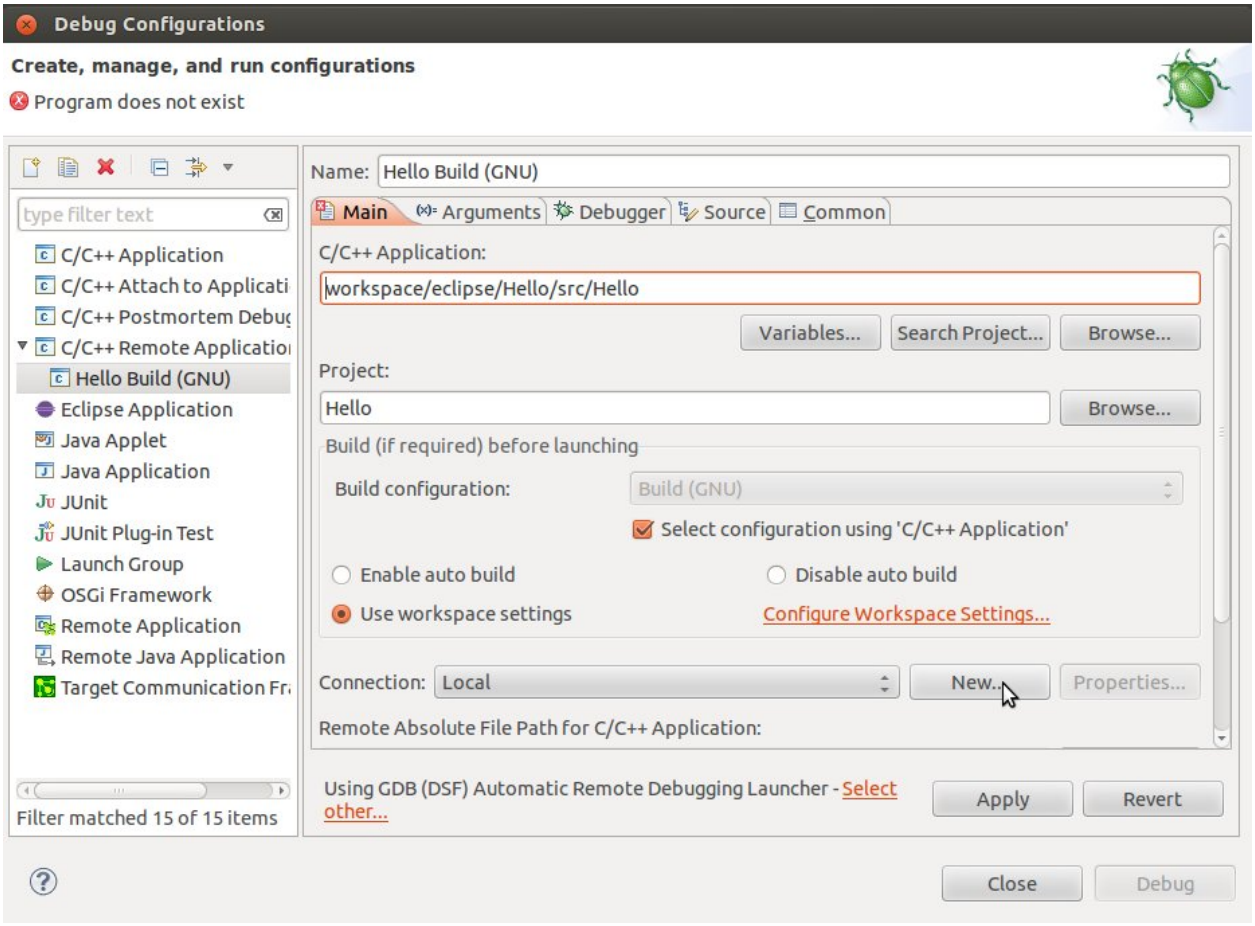
If the output is similar to this one:

then the ethernet connection is ok. Enable the remote debug with Yocto by typing this command on Picozed console:

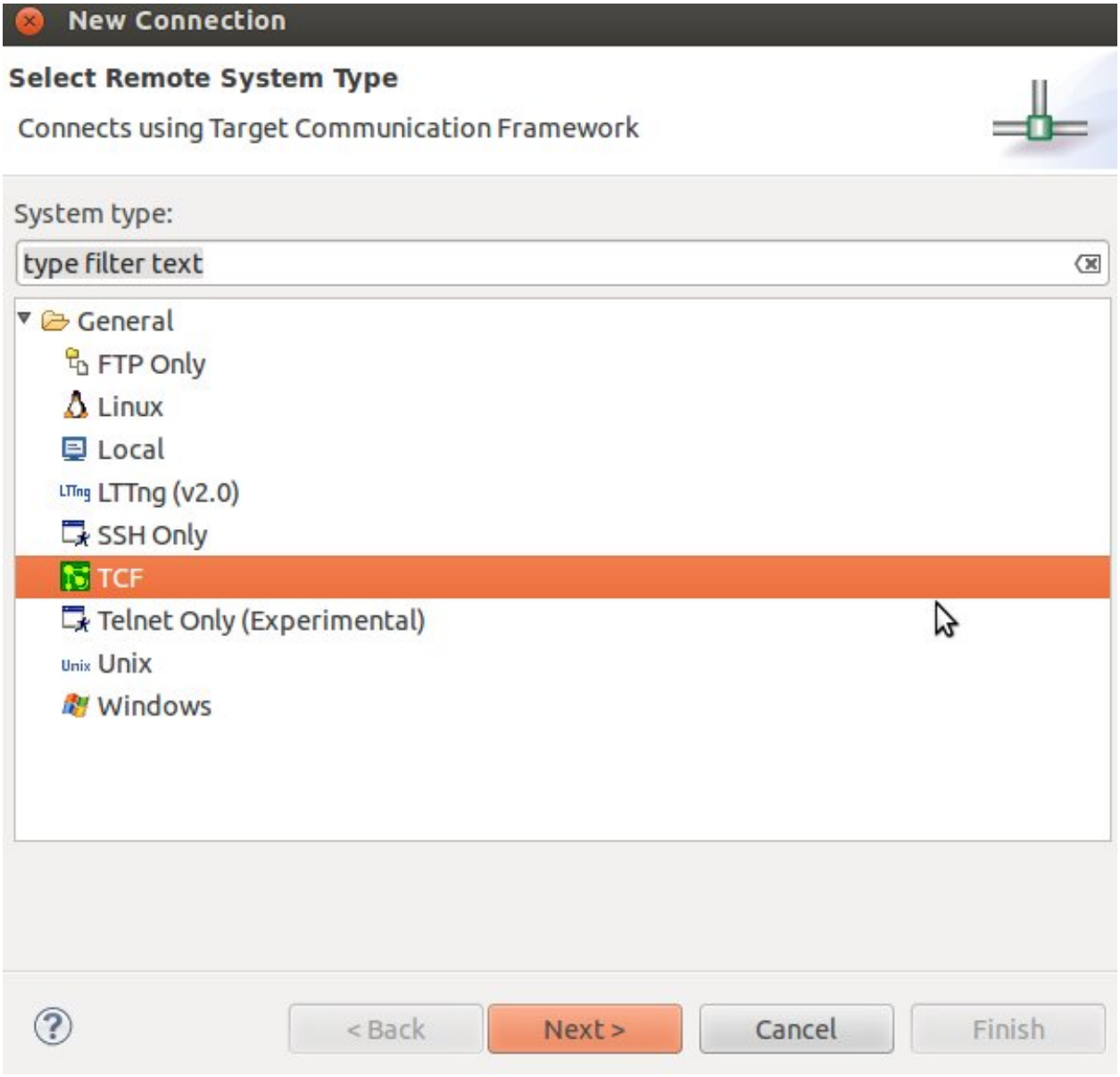
On the Host machine, follow these steps to let **Eclipse** deploy and debug your application:

- Select *Run→ Debug Configurations...*
- In the left area, expand *C/C++Remote Application*.

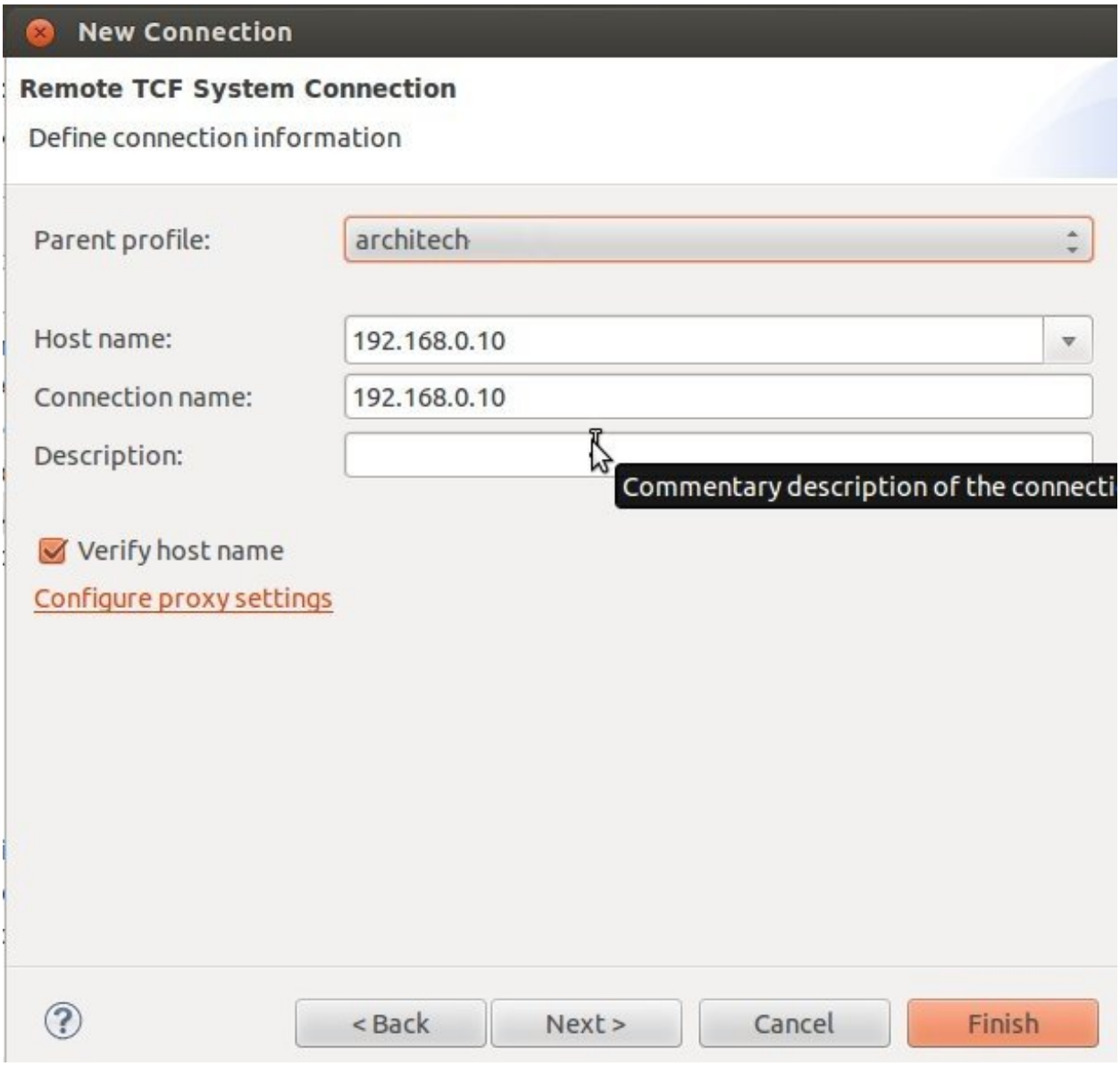
- Locate your project and select it to bring up a new tabbed view in the *Debug Configurations* Dialog.



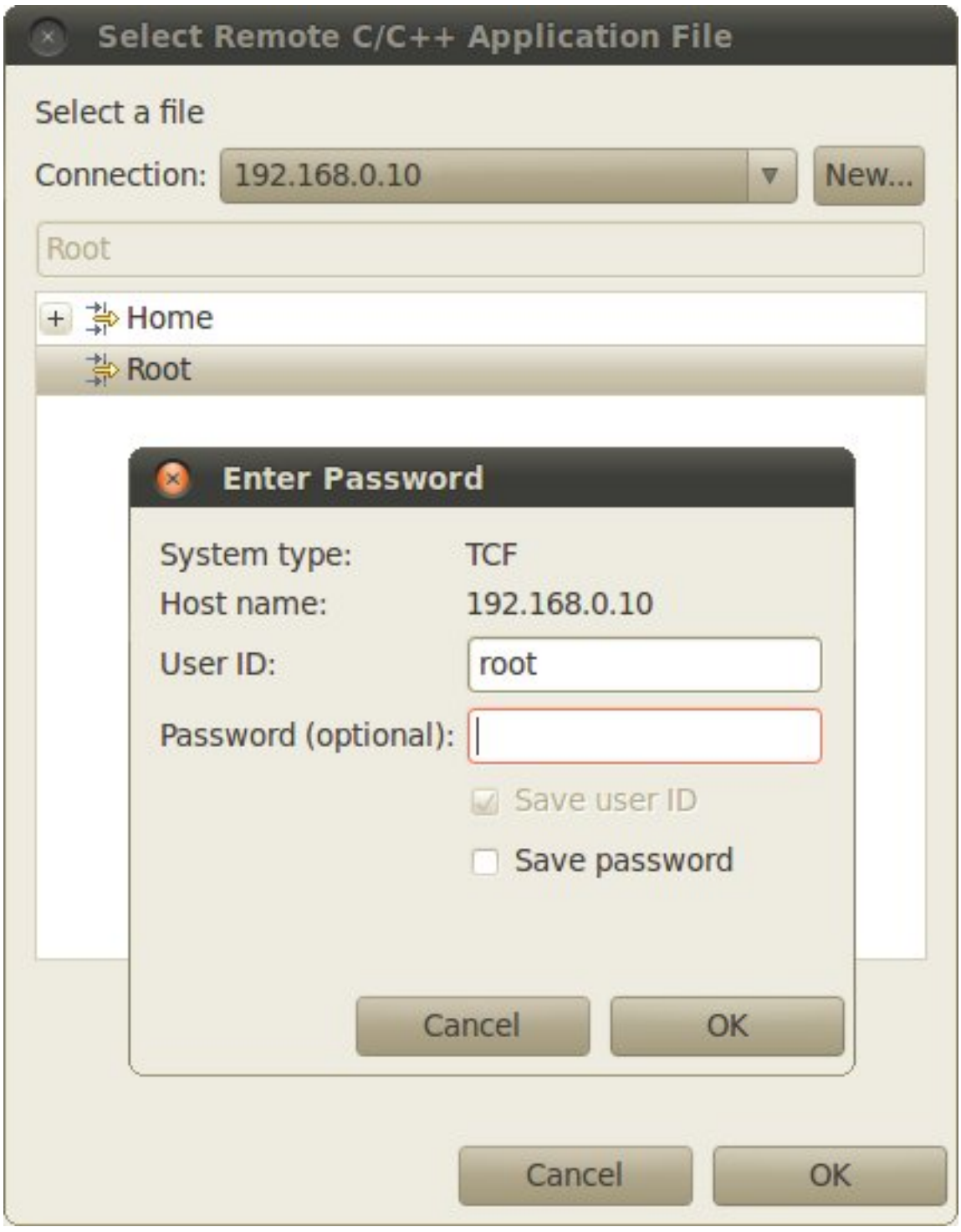
- Insert in *C/C++ Application* the filepath (on your host machine) of the compiled binary.
- Click on *New* button near the drop-down menu in the *Connection* field.
- Select *TCF* icon.



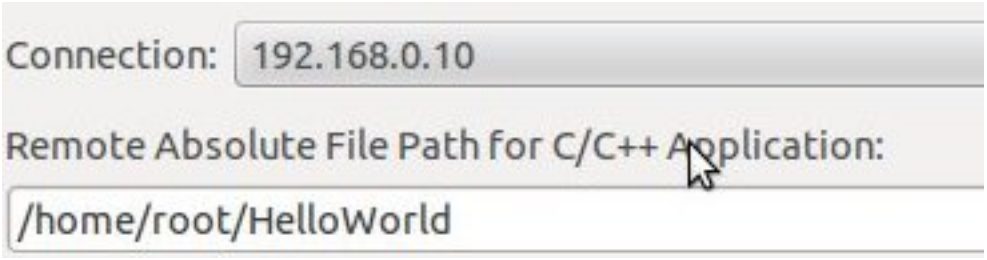
- Insert in *Host Name* and *Connection Name* fields the IP address of the target board. (e.g. 192.168.0.10)



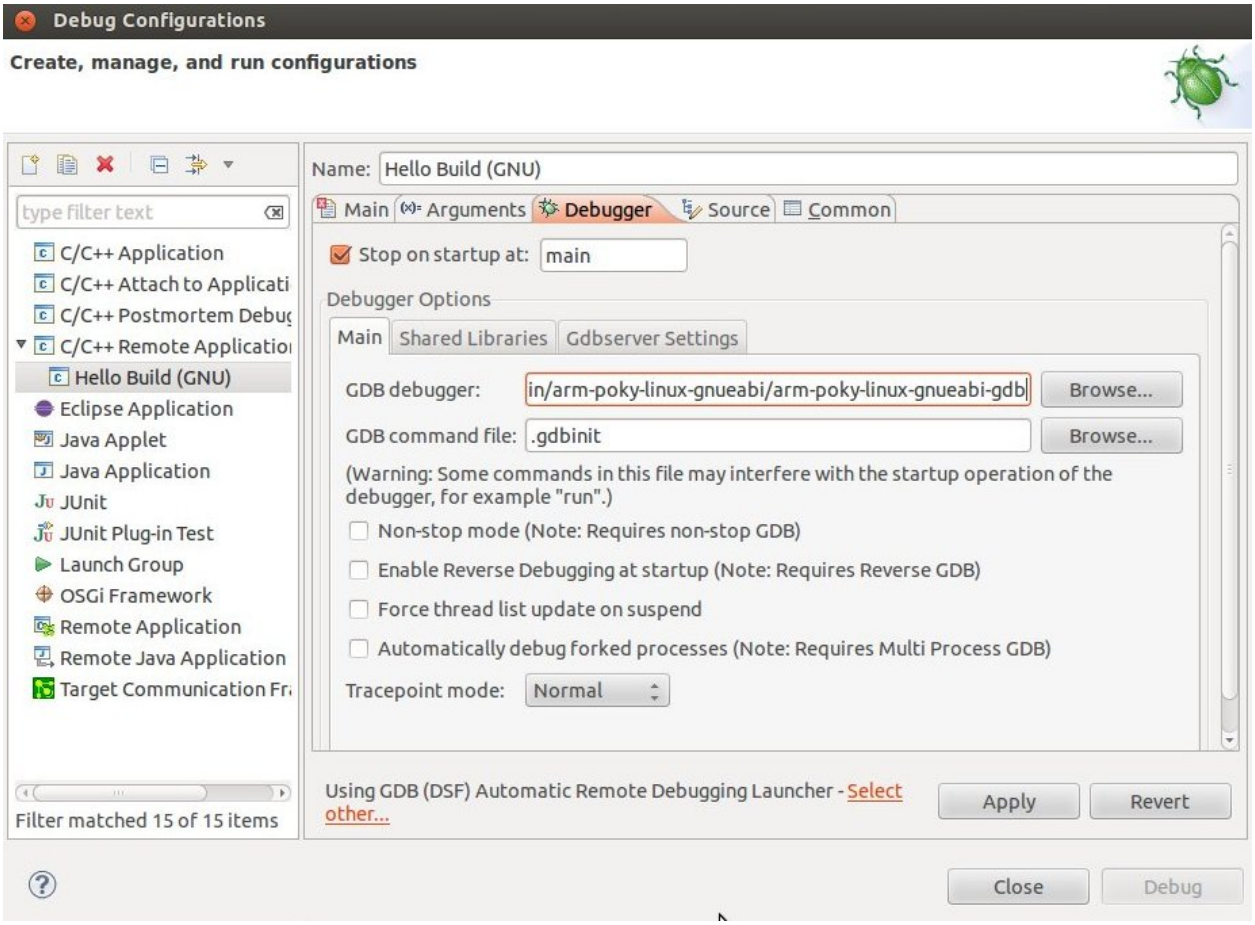
- Then press *Finish*.
- Use the drop-down menu now in the *Connection* field and pick up the IP Address you entered earlier.
- Enter the absolute path on the target into which you want to deploy the cross-compiled application. Use the *Browse* button near *Remote Absolute File Path for C/C++Application:* field. No password is needed.



- Enter also in the path the name of the application you want to debug. (e.g. HelloWorld)



- Select *Debugger* tab



- In GDB Debugger field, insert the filepath of gdb for your toolchain
- In *Debugger* window there is a tab named *Shared Library*, click on it.
- Add the libraries paths *lib* and *usr/lib* of the rootfs (which must be the same used in the target board)
- Click *Debug* to login.
- Accept the debug perspective.

Important: If debug does not work, check on the board if *tcf-agent* is running and *gdbserver* has been installed. You can ignore the message "Cannot access memory at address 0x0".

2.3 SDK Architecture

This chapter gives an overview on how the SDK has been composed and where to find the tools on the virtual machine.

2.3.1 SDK

The SDK provided by *Architech* to support Picozed is composed by several components, the most important of which are:

- **Yocto**,
- **Eclipse**, and
- **Qt Creator**

Regarding the installation and configuration of these tools, you have many options:

1. get a virtual machine with everything already setup,
2. download a script to setup your Ubuntu machine, or
3. just get the meta-layer and compose your SDK by hand

The method you choose depends on your level of expertise and the results you want to achieve.

If you are new to **Yocto** and/or **Linux**, or simply you don't want to read tons of documentation right now, we suggest you to download and *install the virtual machine* because it is the simplest solution (have a look at *VM content*), everything inside the virtual machine has been thought to work out of the box, plus you will get support.

If performances are your greatest concerns, consider reading Chapter *Create SDK*.

2.3.2 Virtual Machine

The development environment is provided as a virtual disk (to be used by a VirtualBox virtual machine) which you can download from this page:

Important: http://downloads.architechboards.com/sdk/virtual_machine/download.html

---

Important: Compute the MD5SUM value of the zip file you downloaded and compare it to the golden one you find in the download page.

Uncompress the file, and you will get a *.vdi* file that is our virtual disk image. The environment contains the SDK for all the boards provided by Architech, Picozed included.

Download VirtualBox



For being able to use it, you first need to install **VirtualBox** (version 4.2.10 or higher). You can get VirtualBox installer from here:

https://www.virtualbox.org/wiki/Downloads

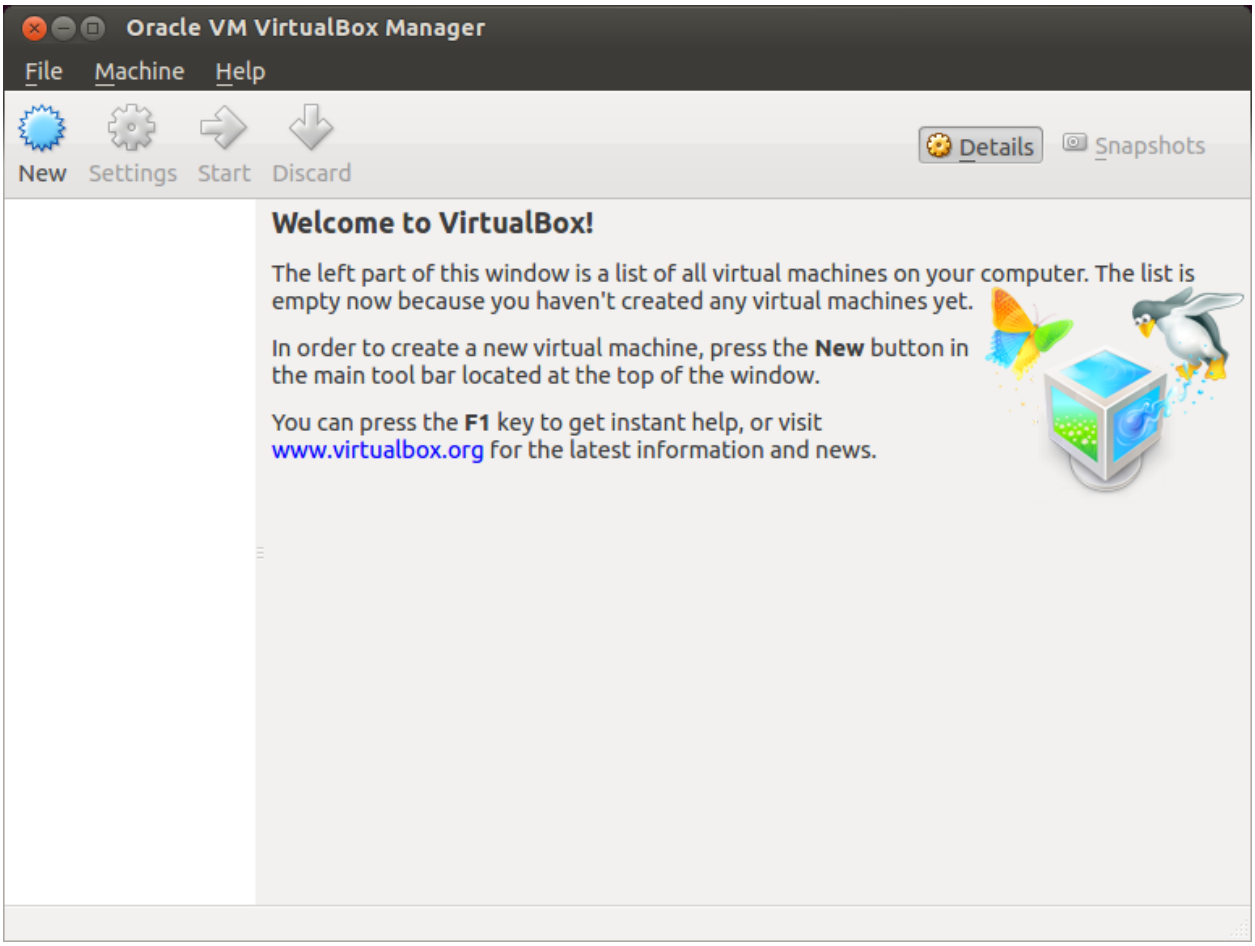
Download the version that suits your host operating system. You need to download and install the **Extension Pack** as well.

Important: Make sure that the extension pack has the same version of VirtualBox.

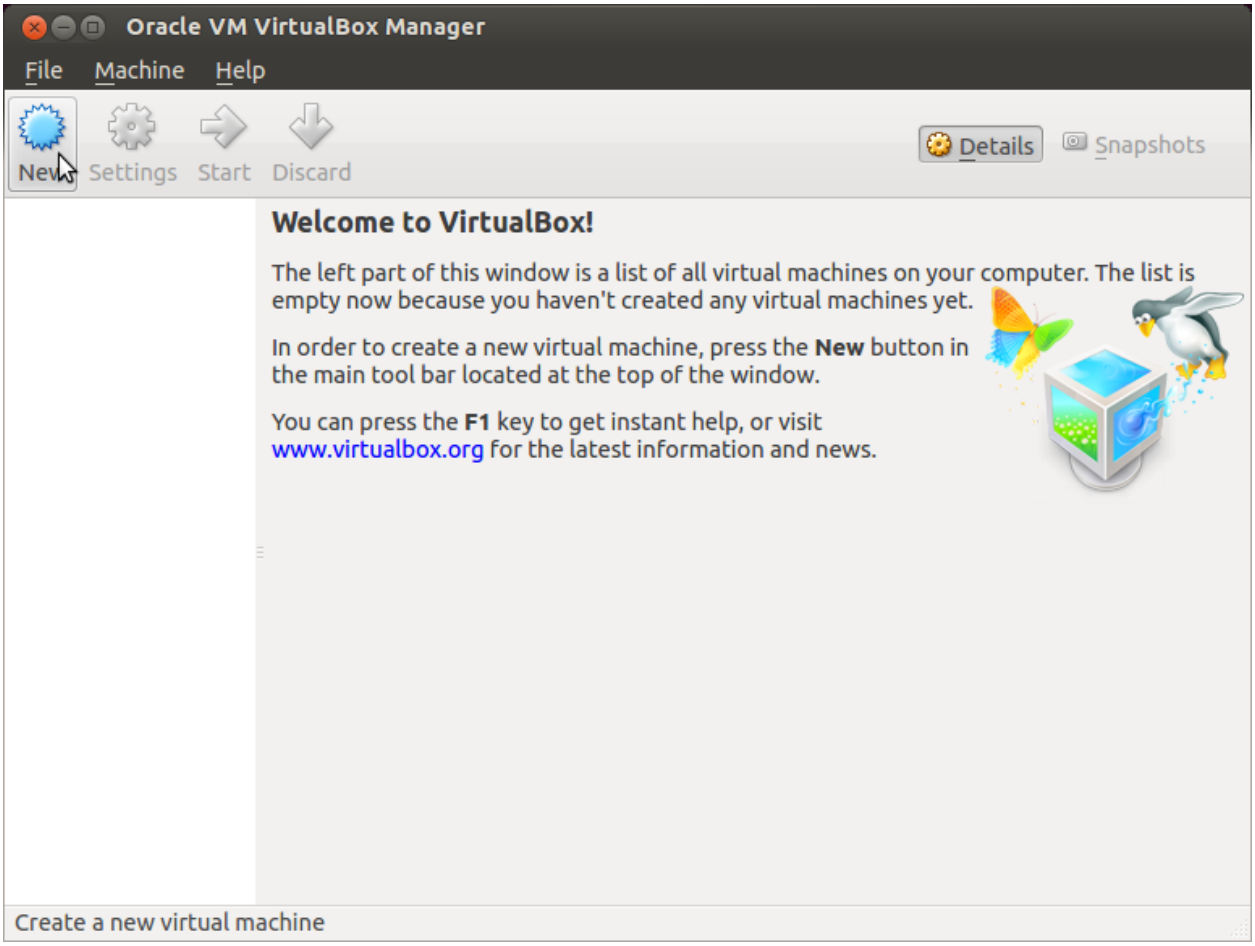
Install the software with all the default options.

Create a new Virtual Machine

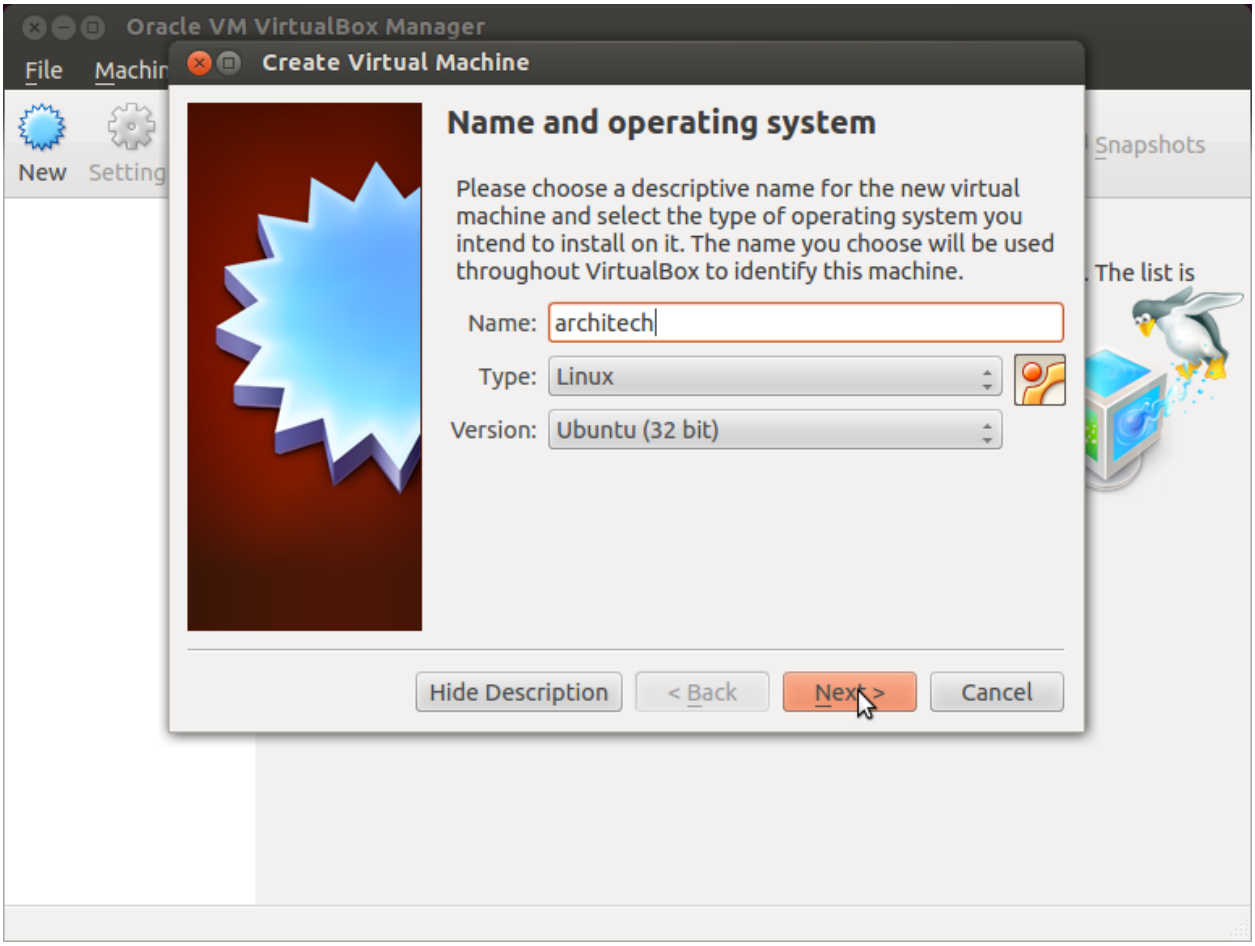
1. Run VirtualBox



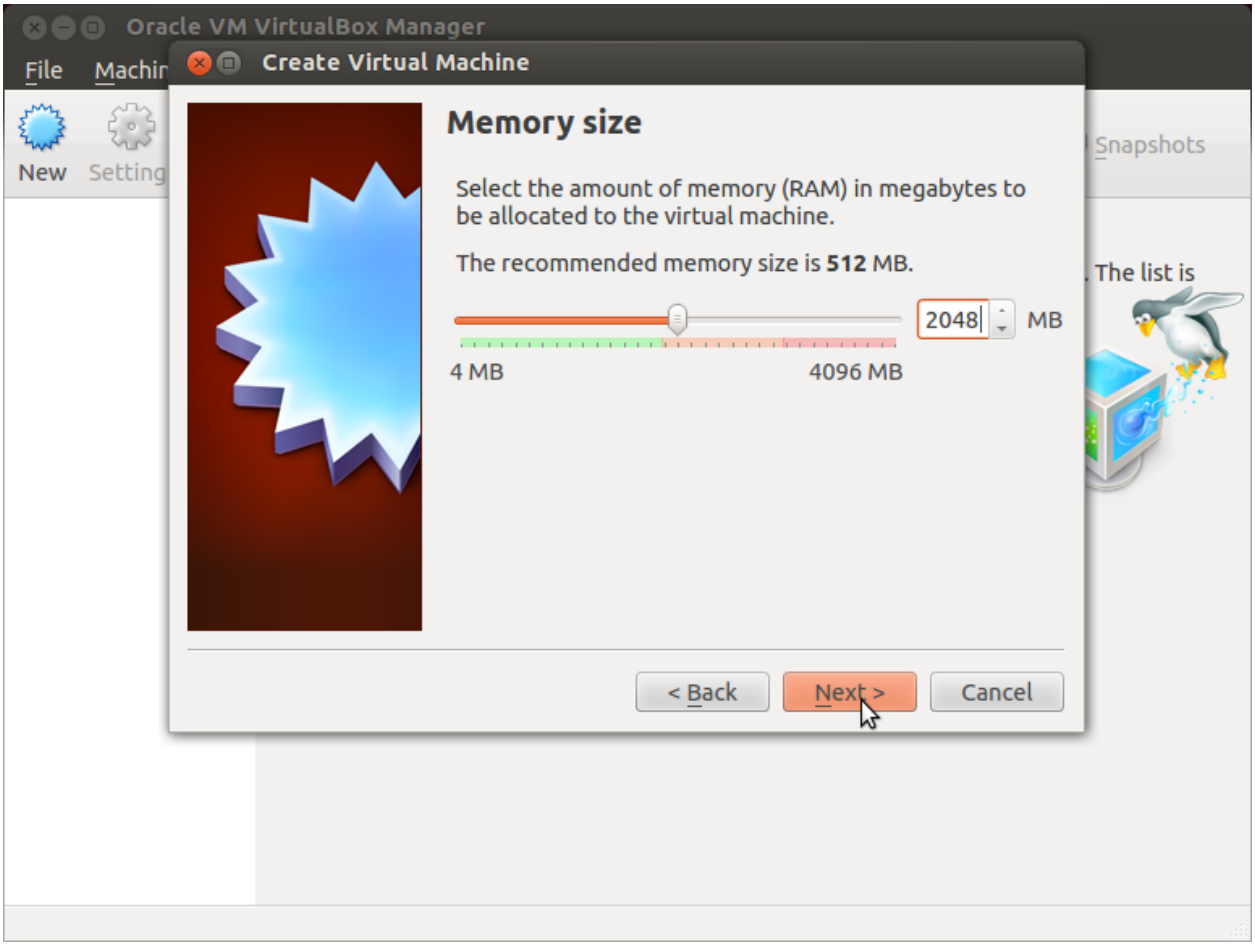
2. Click on *New* button



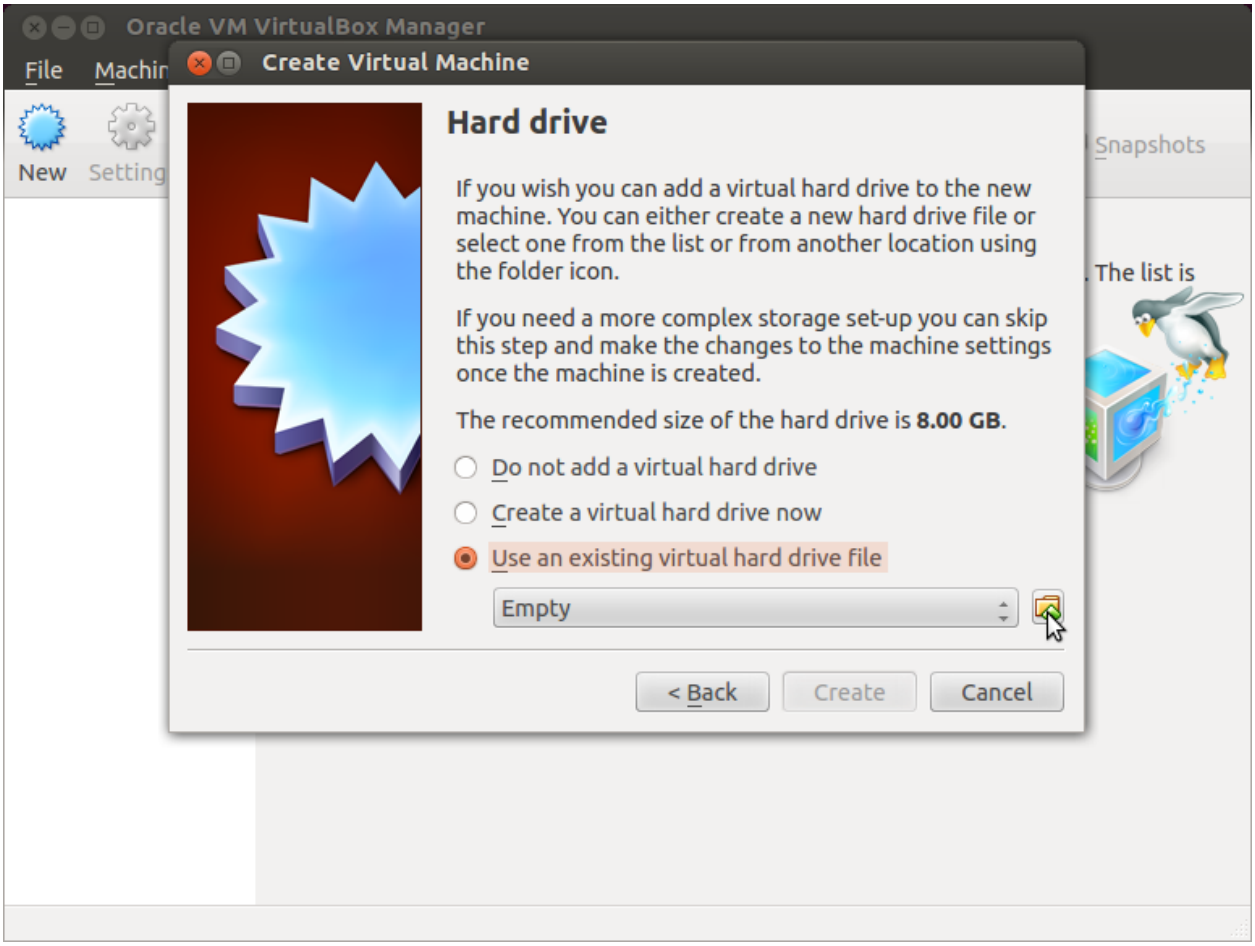
3. Select the name of the virtual machine and the operating system type



4. Select the amount of memory you want to give to your new virtual machine



5. Make the virtual machine use Architech's virtual disk by pointing to the downloaded file. Than click on *Create*.

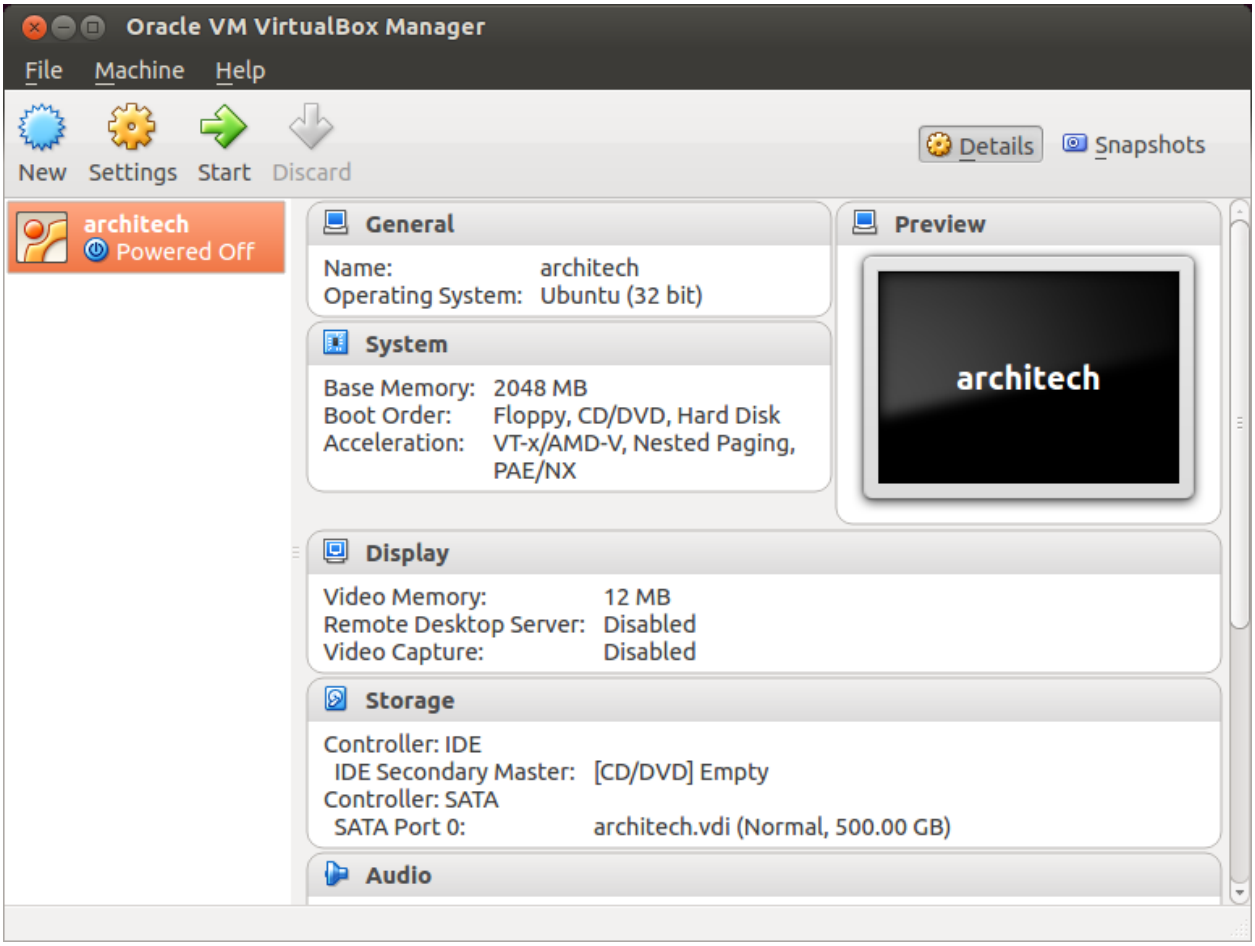


Setup the network

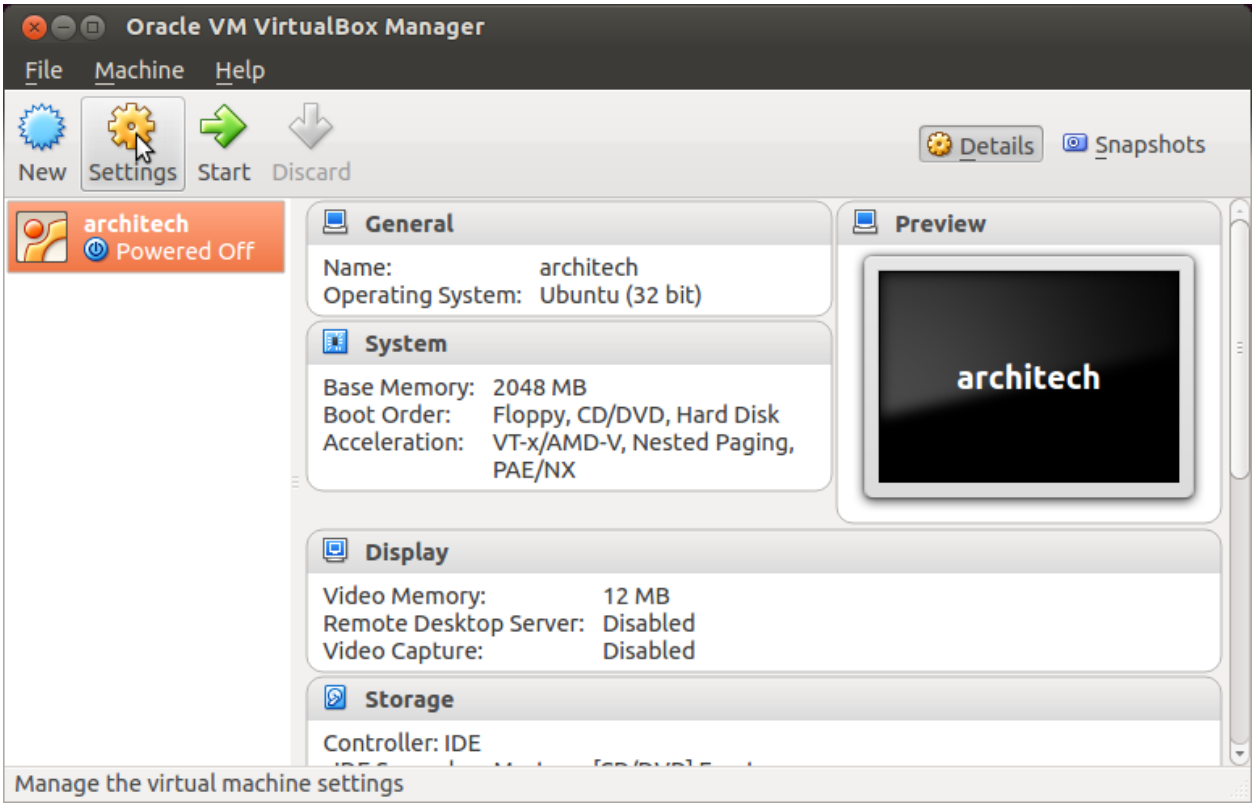
We need to setup a port forwarding rule to let you (later) use the virtual machine as a local repository of packages.

Note: The virtual machine must be off

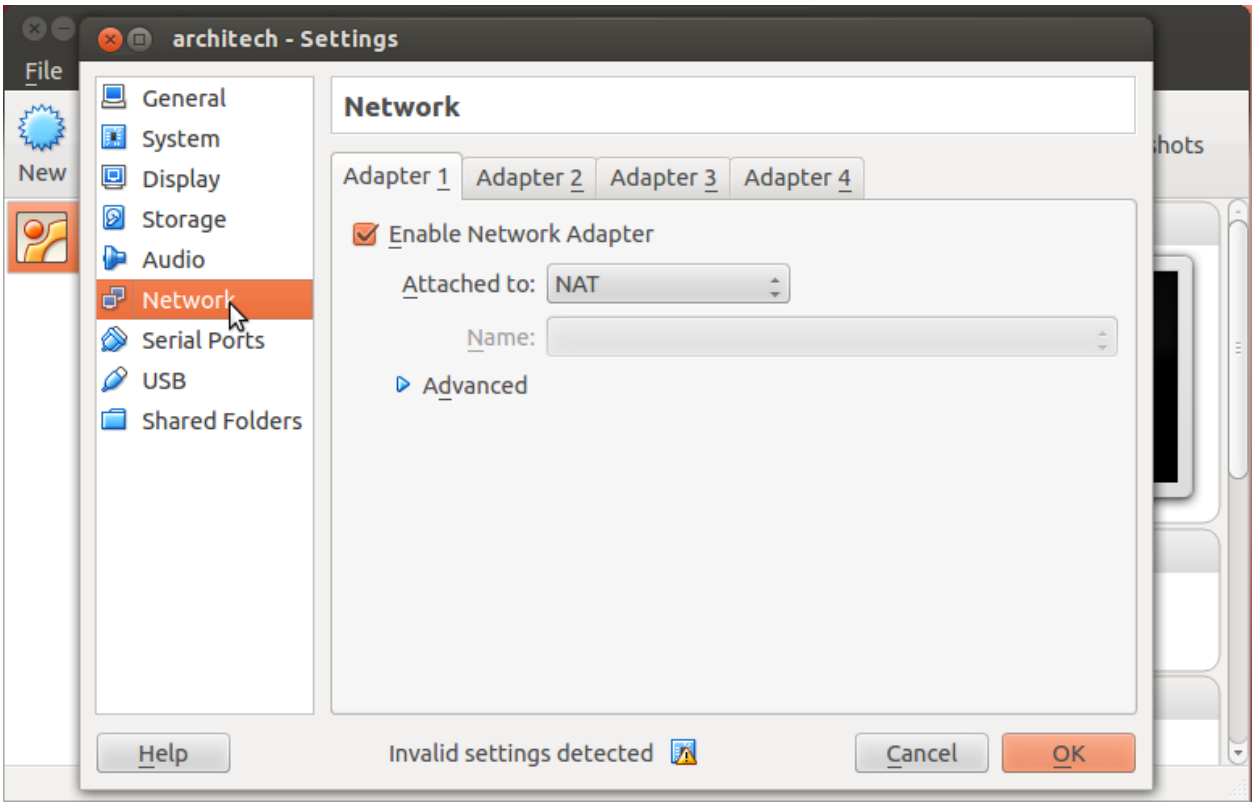
1. Select Architech's virtual machine from the list of virtual machines



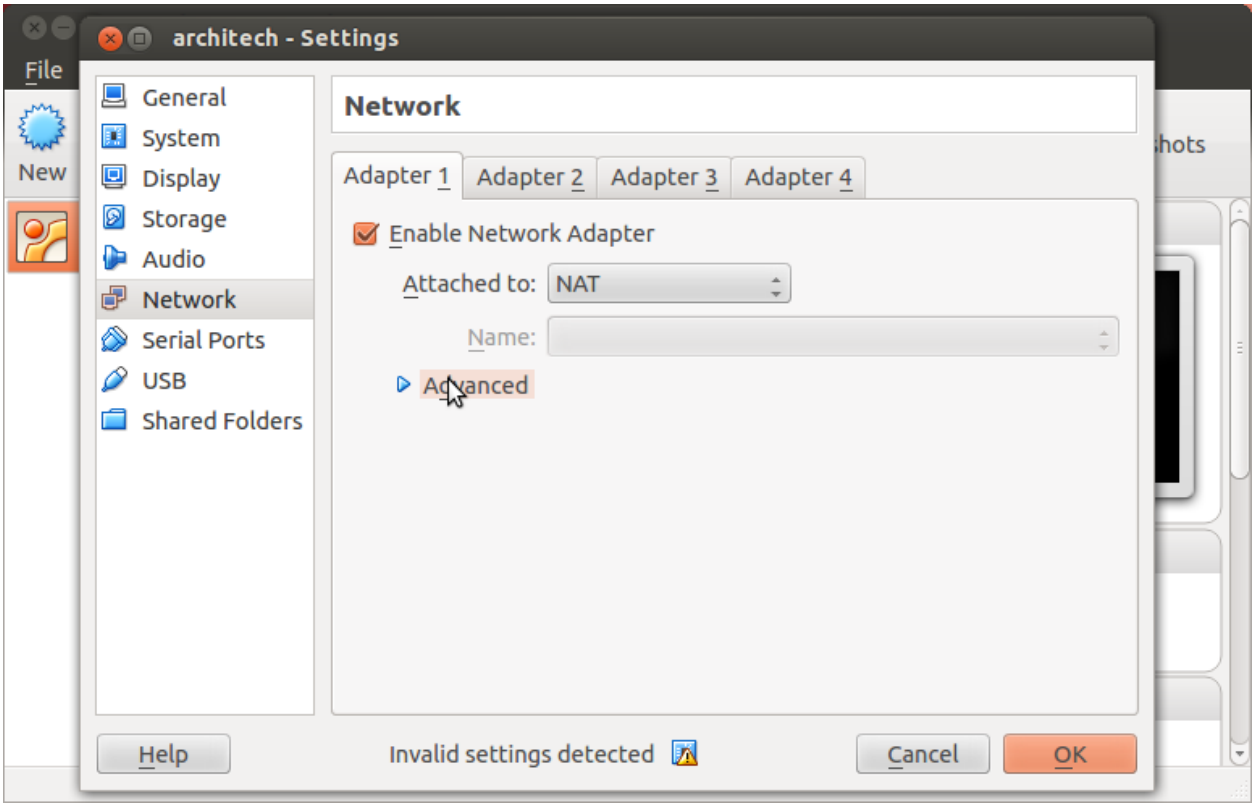
2. Click on *Settings*



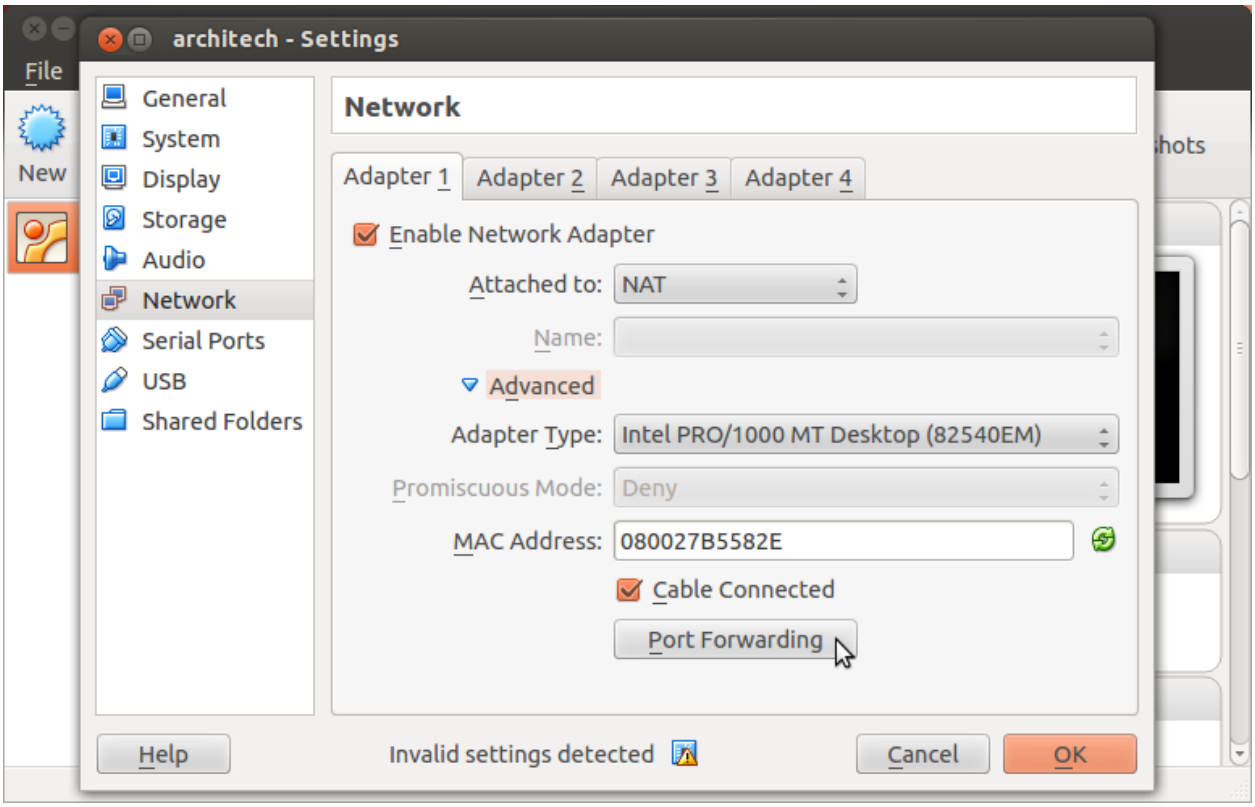
3. Select *Network*



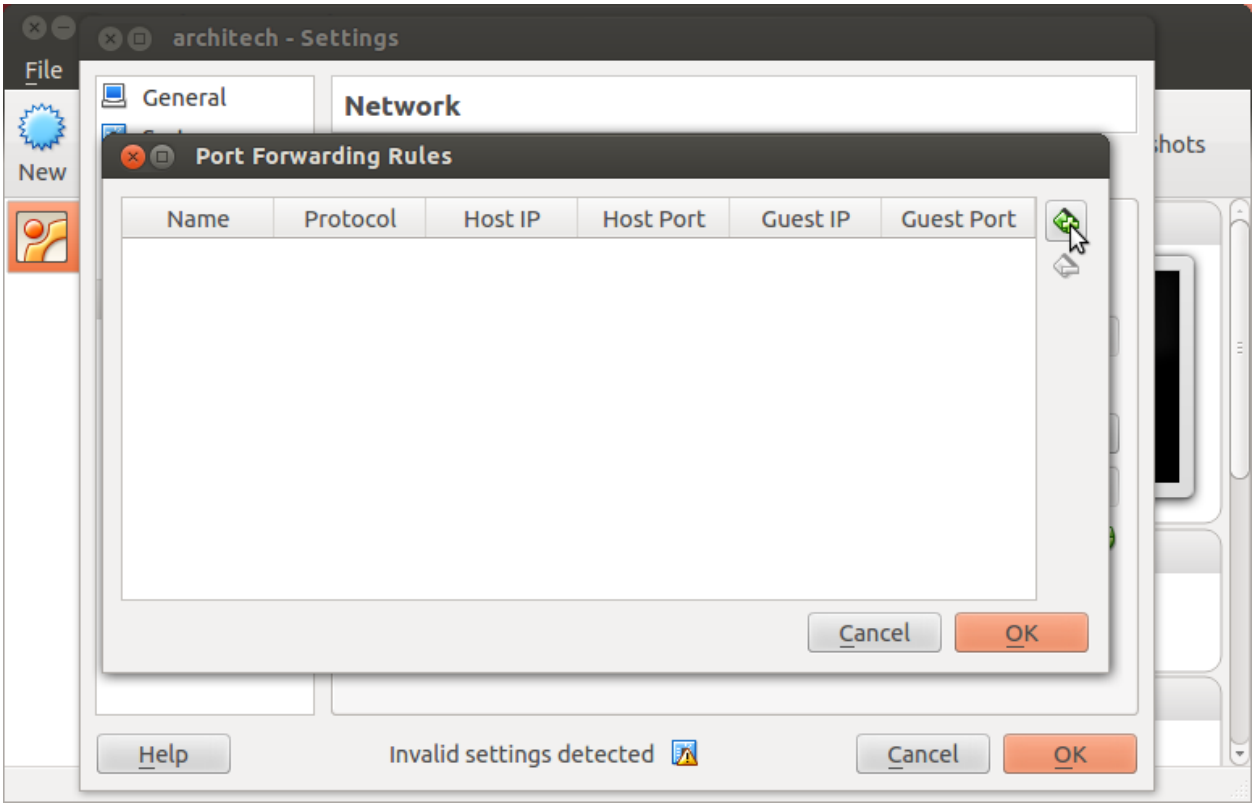
4. Expand *Advanced* of *Adapter 1*



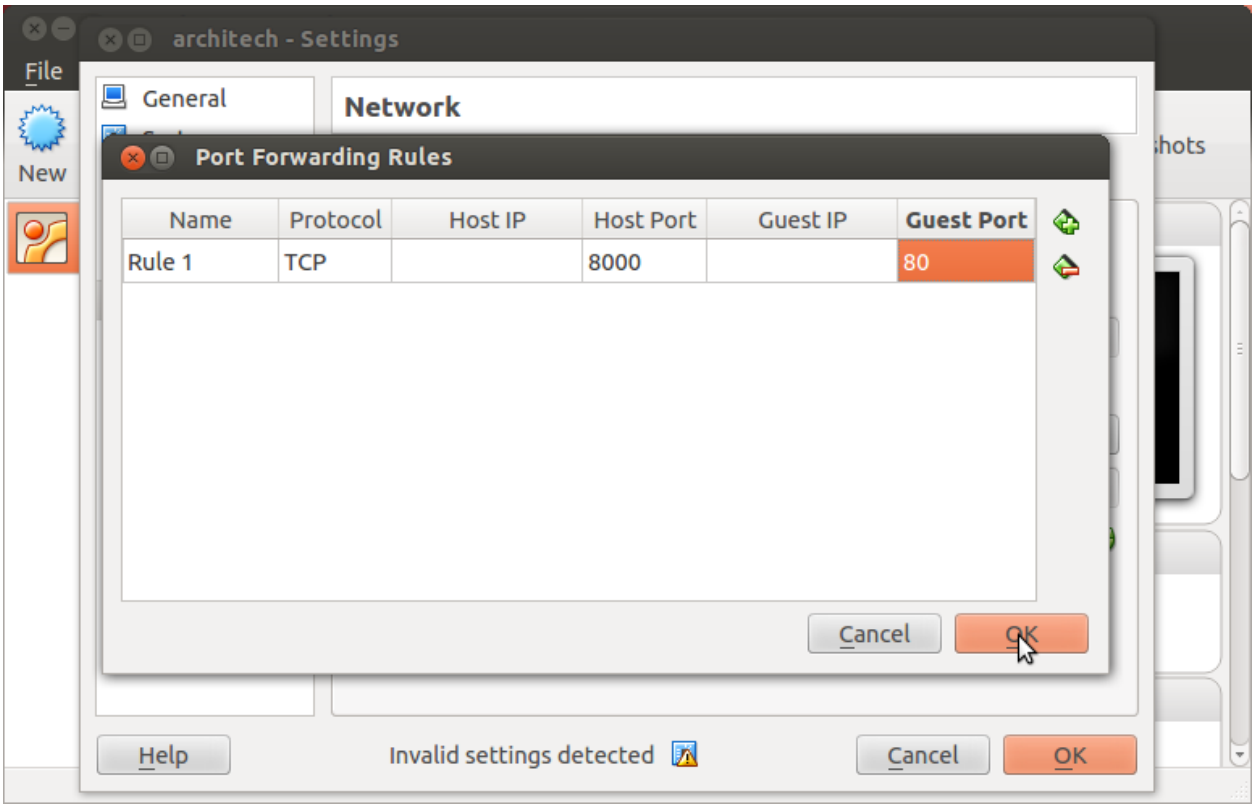
5. Click on *Port Forwarding*



6. Add a new *rule*



7. Configure the *rule*



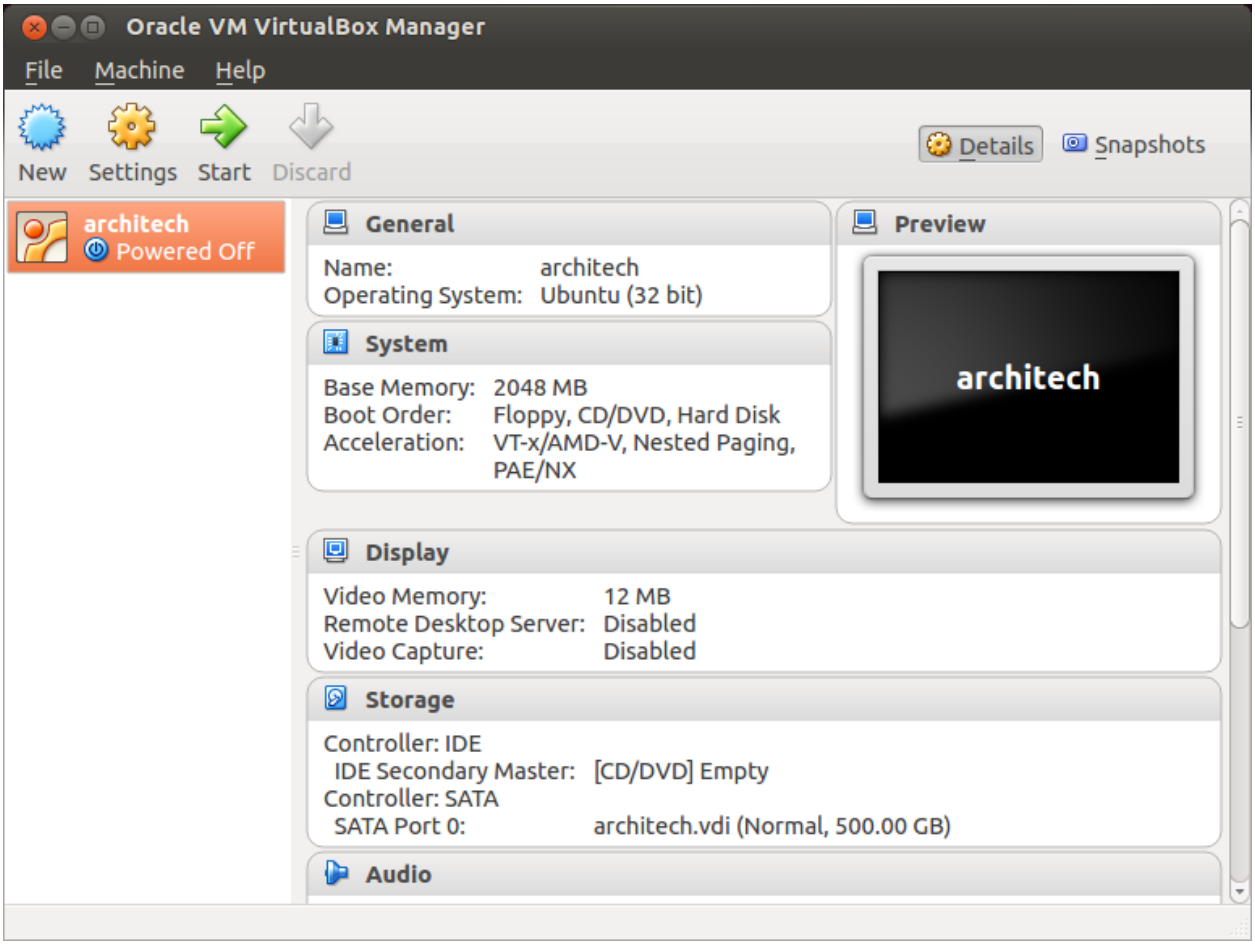
8. Click on *Ok*

Customize the number of processors

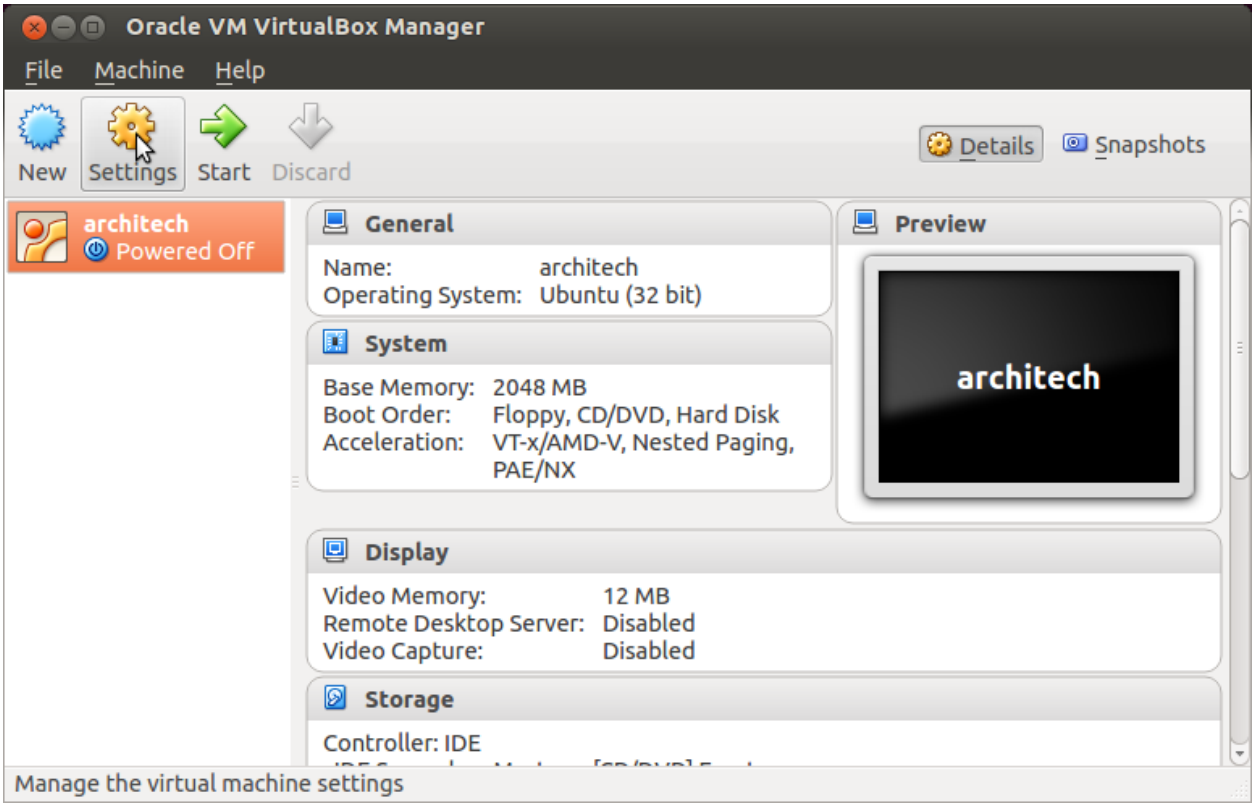
Building an entire system from the ground up is a business that can take up to several hours. To improve the performances of the overall build process, you can, if your computer has enough resources, assign more than one processor to the virtual machine.

Note: The virtual machine must be off

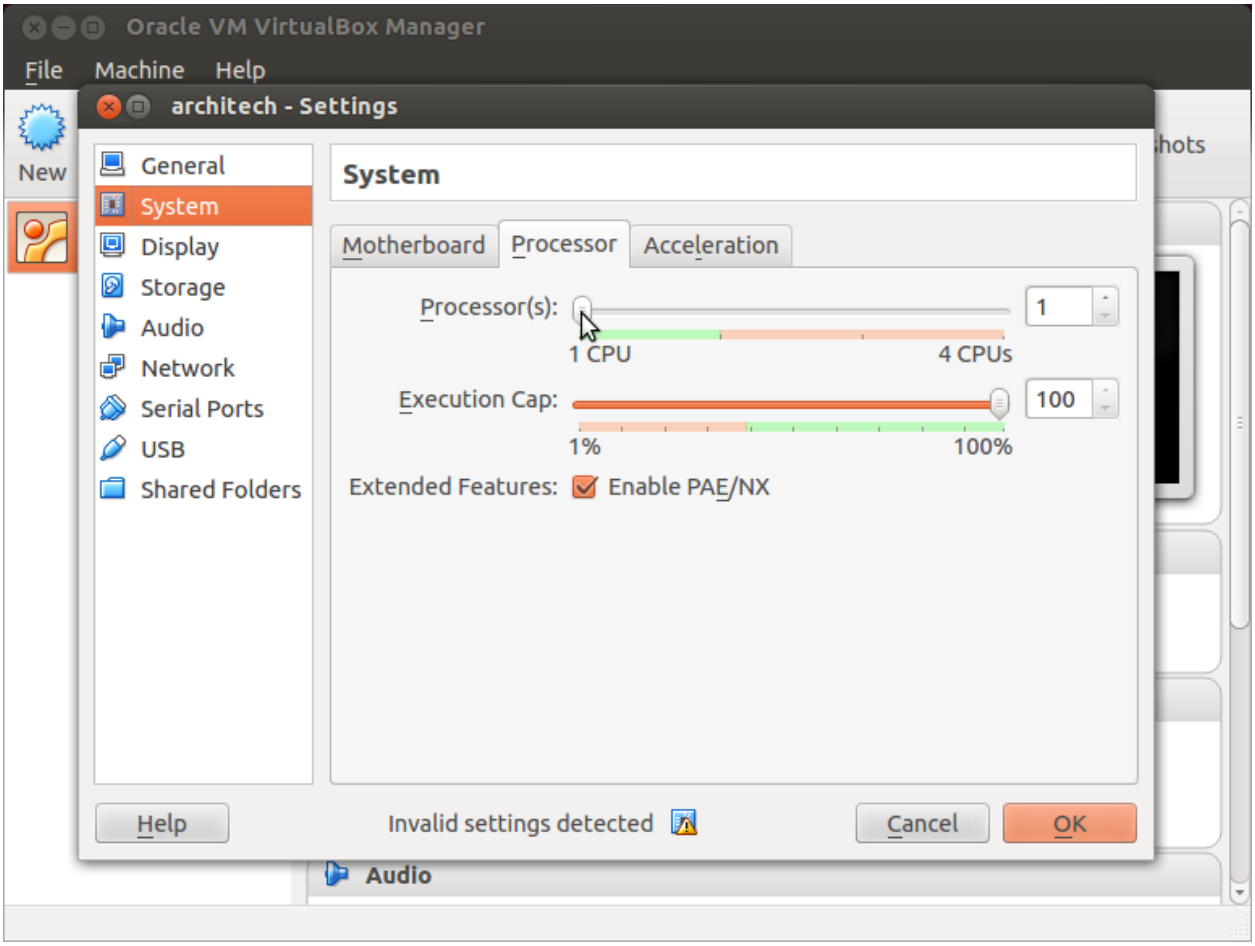
1. Select Architech's virtual machine from the list of virtual machines



2. Click on *Settings*



3. Select *System*
4. Select *Processor*
5. Assign the number of processors you wish to assign to the virtual machine

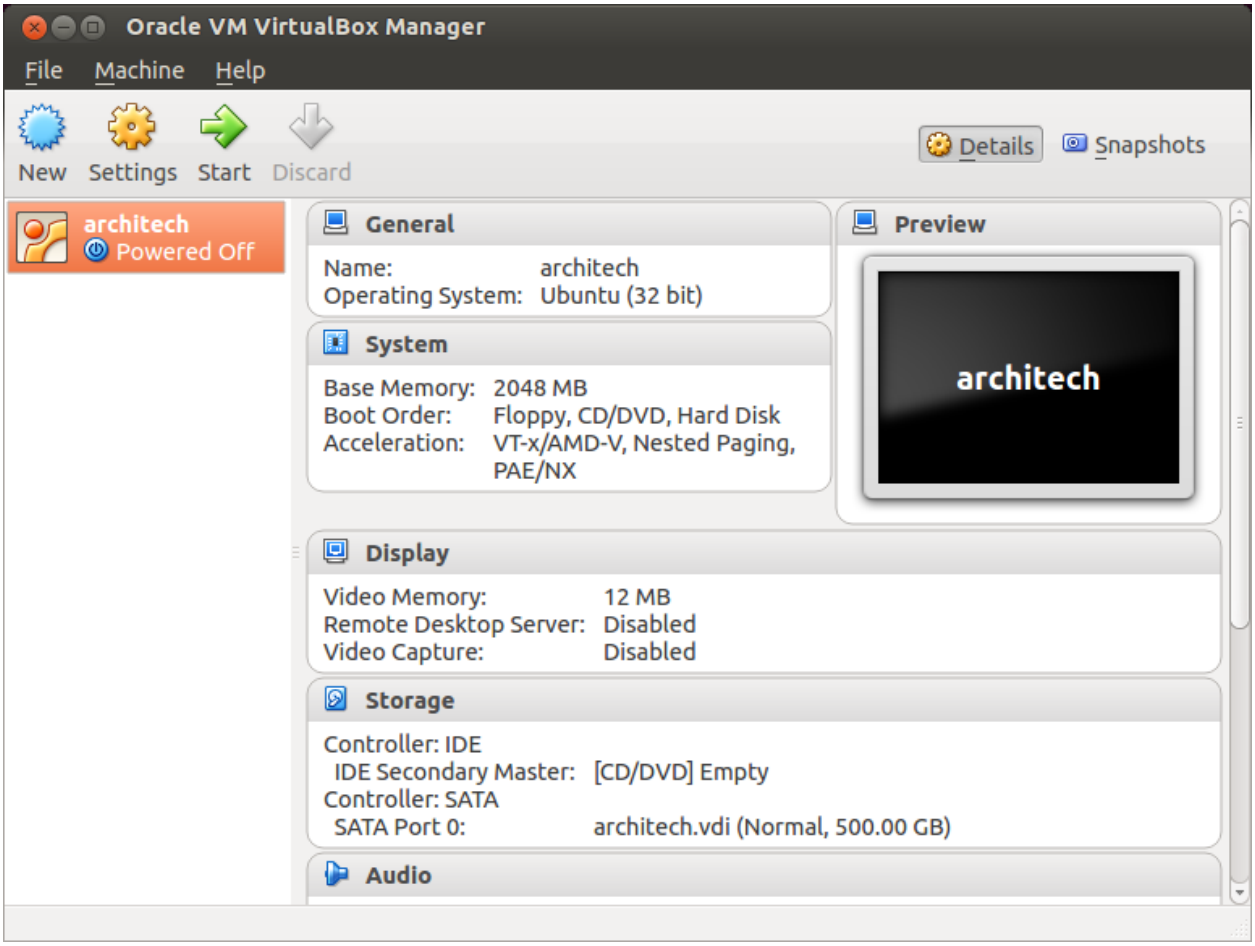


Create a shared folder

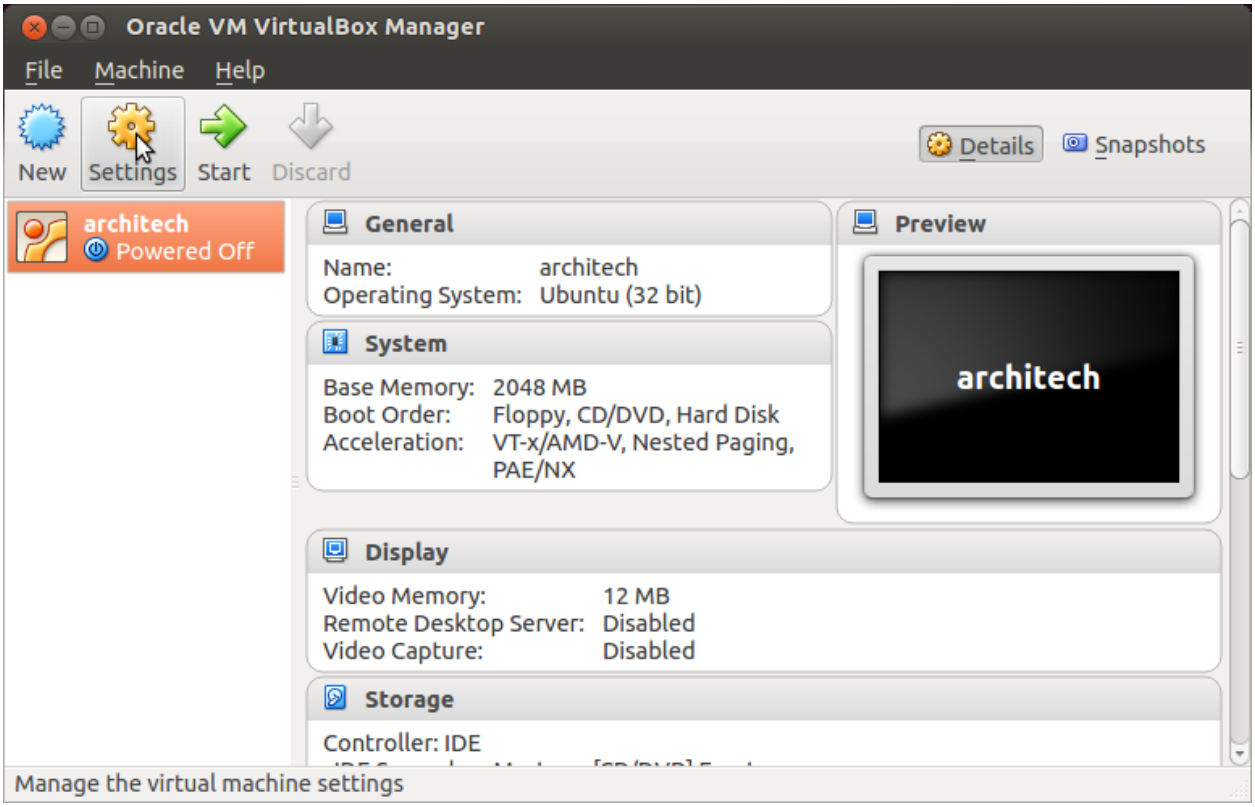
A shared folder is way for host and guest operating systems to exchange files by means of the file system. You need to choose a directory on your host operating system to share with the guest operating system.

Note: The virtual machine must be off

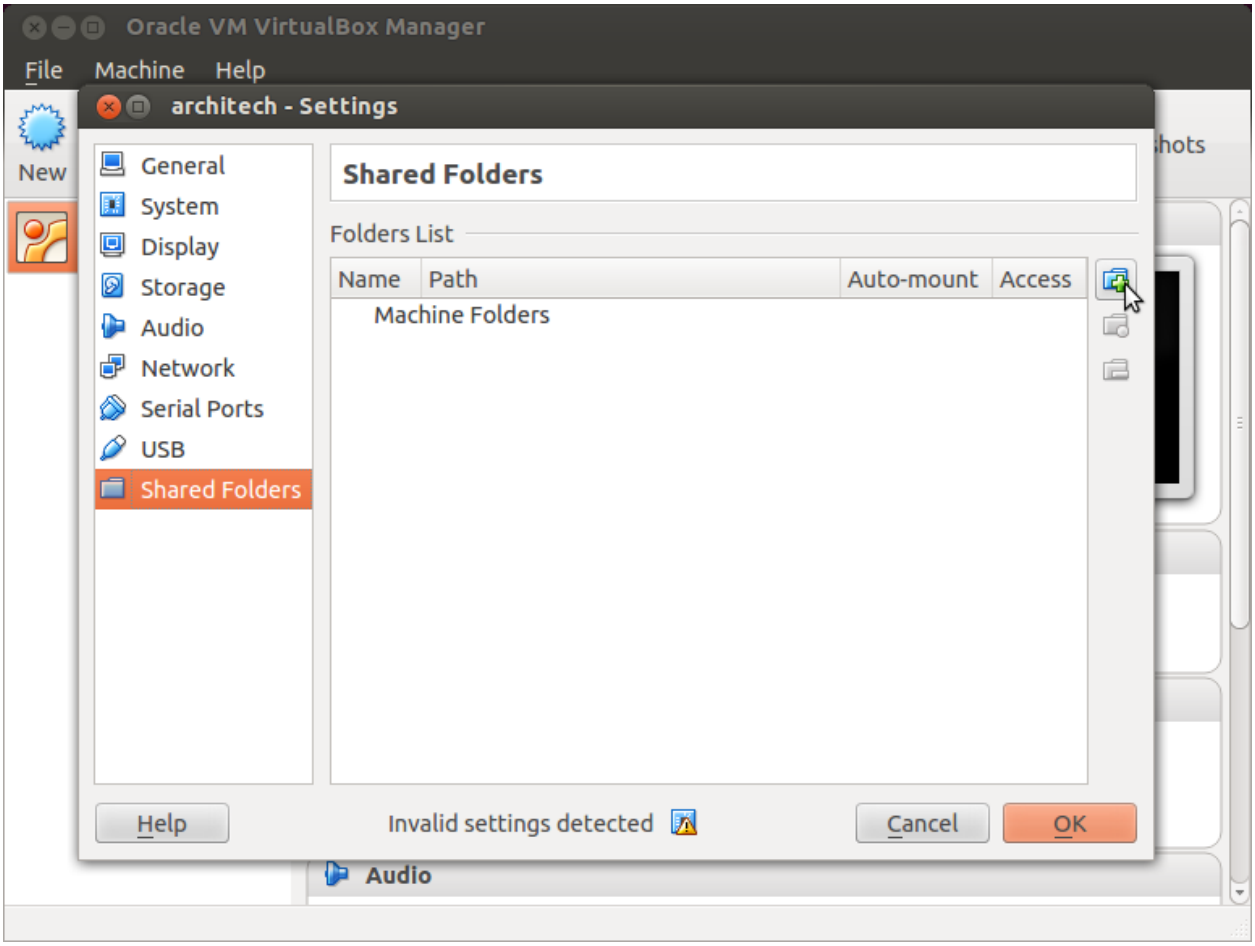
1. Select Architech's virtual machine from the list of virtual machines



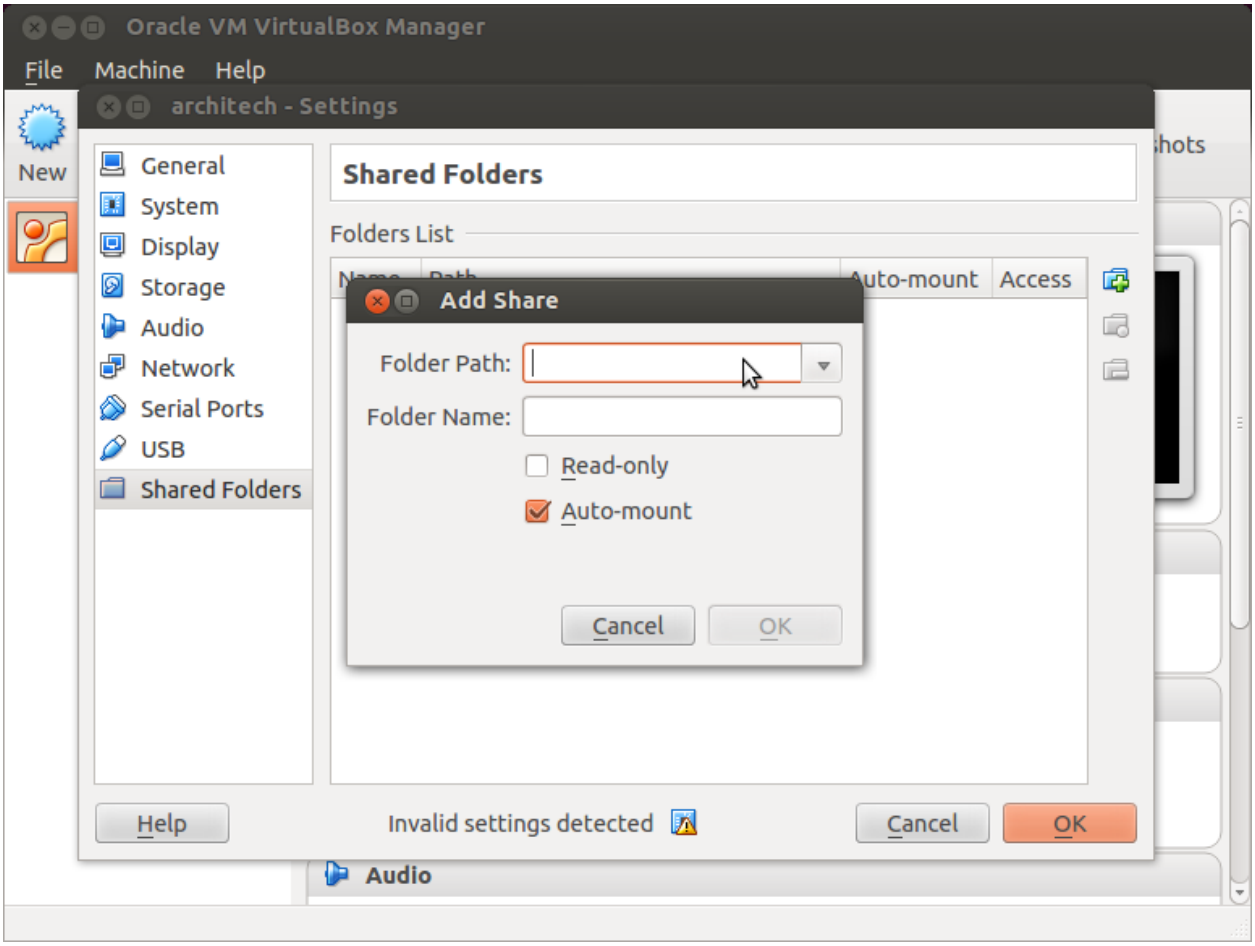
2. Click on *Settings*



3. Select *Shared Folders*
4. Add a new shared folder



5. Choose a directory to share on your host machine. Make sure *Auto-mount* is selected.



Once the virtual machine has been booted, the shared folder will be mounted under */media/* directory inside the virtual machine.

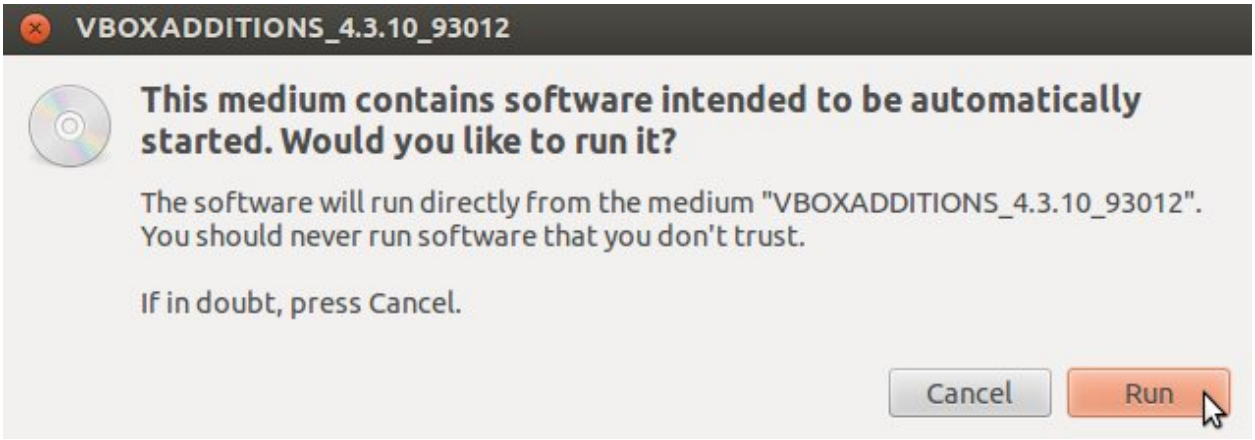
Install VBox Additions

The VBox addictions add functionalities to the virtual machine such as better graphic driver and more. It is already installed in the SDK but is important re-install it to configuring correctly the virtual machine with your operating system.

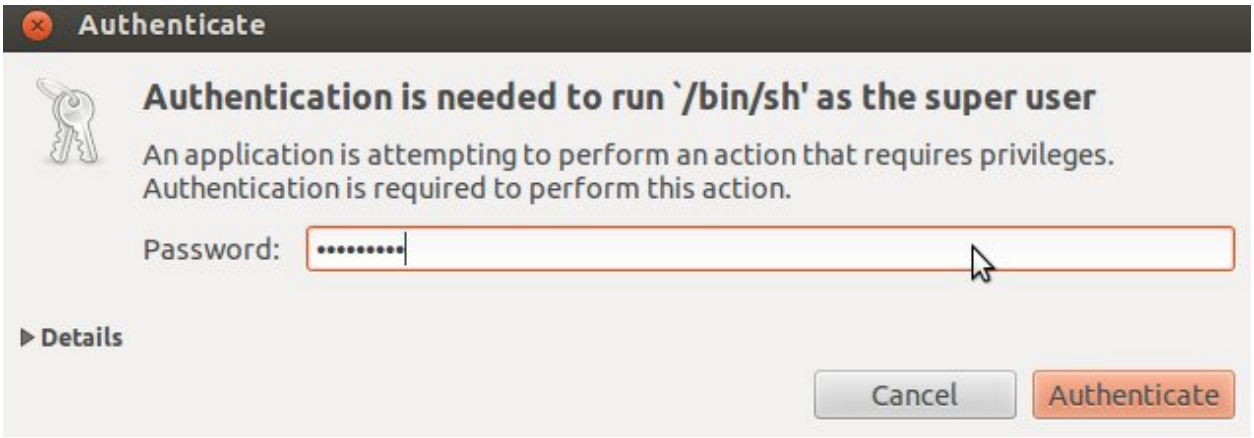
1. Starts the virtual machine



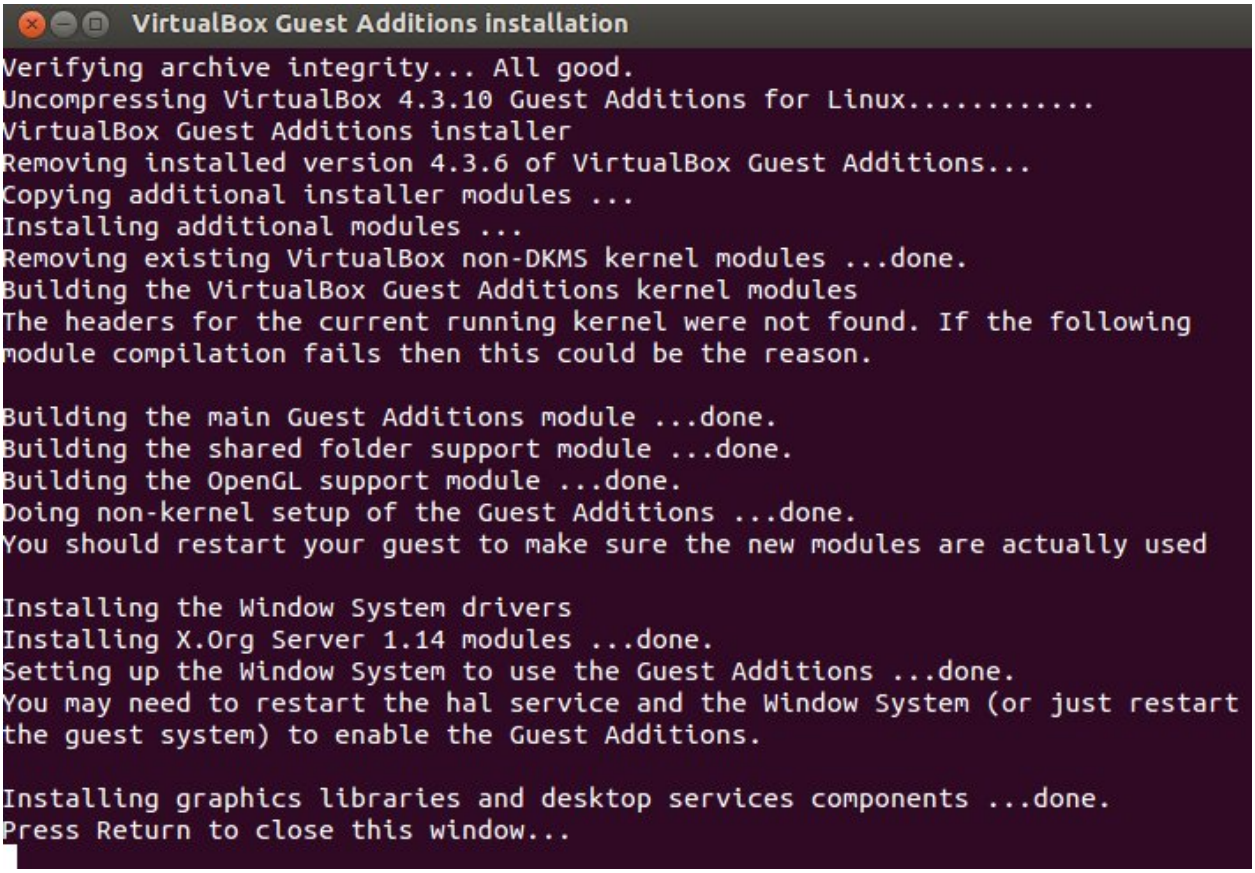
2. Click on the virtual box menu to the voice *Devices* and select *Insert Guest Additions CD Images....* A message box will appear at the start of the installation, click on *run* button



4. To proceed are required admin privileges, so insert the password *architech* when asked



5. Then a terminal will show the installation progress. When finished, press *Enter* key



6. Before to use the SDK, it is required reboot the virtual machine

2.3.3 VM content

The virtual machine provided by Architech contains:

- A splash screen, used to easily interact with the boards tools
- Yocto/OpenEmbedded toolchain to build BSPs and file systems
- A cross-toolchain (derived from Yocto/OpenEmbedded) for all the boards
- Eclipse, installed and configured
- Qt creator, installed and configured

All the aforementioned tools are installed under directory **/home/architech/architech_sdk**, its sub-directories main layout is the following:

picozed directory contains all the tools composing the ArchiTech SDK for Picozed board, along with all the information needed by the splash screen application. In particular:

- *eclipse* directory is where Eclipse IDE has been installed
- *qtcreator* contains the installation of Qt Creator IDE
- *splashscreen* directory contains information and scripts used by the splash screen application,
- *sysroot* is supposed to contain the file system you want to compile against,
- *toolchain* is where the cross-toolchain has been installed installed

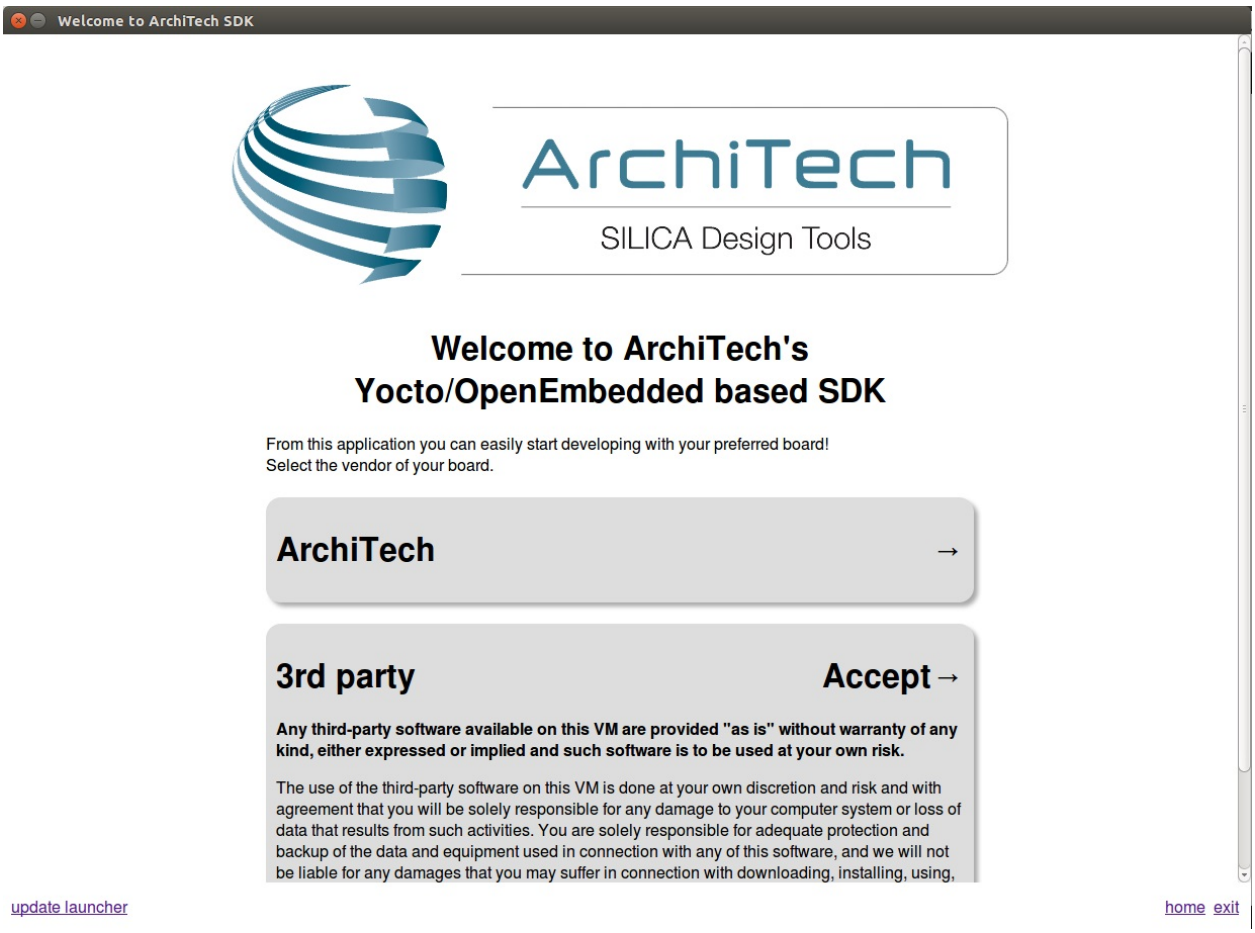
- *workspace* contains the the workspaces for Eclipse and Qt Creator IDEs
- *yocto* is where you find all the meta-layers Picozed requires, along with Poky and the build directory

Splash screen

The splash screen application has been designed to facilitate the access to the boards tools. It can be opened by clicking on its *Desktop* icon.



Once started, you can can choose if you want to work with Architech's boards or with partners' ones. For Picozed, choose **ArchiTech**.



A list of all available Architech's boards will open, select Picozed.

A list of actions related to Picozed that can be activated will appear.



2.4 Create SDK

If you have speed in mind, it is possible to install the SDK on a native Ubuntu machine (other Linux distributions may support this SDK with minor changes but won't be supported). This chapter will guide you on how to clone the entire SDK, to setup the SDK for one board or just **OpenEmbedded/Yocto** for Picozed board.

2.4.1 Installation

Architech's Yocto based SDK is built on top of **Ubuntu 14.04 32bit**, hence all the scripts provided are proven to work on such a system.

If you wish to use another distribution/version you might need to change some script option and/or modify the scripts yourself, remember that you won't get any support in doing so.

Install a clone of the virtual machine inside your native machine

To install the same tools you get inside the virtual machine on your native machine you need to download and run a system wide installation script:

where *-g* option asks the script to install and configure a few graphic customization, while *-p* option asks the script to install the required packages on the machine. If you want to install the toolchain on a machine not equal to Ubuntu 14.04 32bit then you may want to read the script, install the required packages by hand, and run it without options. You might need to recompile the Qt application used to render the splashscreen.

At the end of the installation process, you will get the same tools installed within the virtual machine, that is, all the tools necessary to work with Architech's boards.

Install just one board

If you don't want to install the tools for all the boards, you can install just the subset of tools related to Picozed:

1. Install the following packages, it can require for a while:
2. Install repo tool, if you already have it go to the next step
3. Make sure directory *~/bin* is included in your *PATH* variable by printing its content
4. If *~/bin* directory is not included, add this line to your *~/.bashrc*
5. Install and setup git:
6. Finally install the board sdk:

before build an image with bitbake open the file */your/path/Picozed/yocto/build/conf/local.conf* and edit these variables:

and change them in:

2.4.2 Yocto

If you have launched machine_installer or run_install.sh script, yocto is already installed. The following steps are useful for understood how the sdk works "under the hood".

Installation with repo

The easiest way to setup and keep all the necessary meta-layers in sync with upstream repositories is achieved by means of Google's **repo** tool. The following steps are necessary for a clean installation:

1. Install repo tool, if you already have it go to step 4
2. Make sure directory *~/bin* is included in your *PATH* variable by printing its content
3. If *~/bin* directory is not included, add this line to your *~/.bashrc*
4. Open a new terminal
5. Change the current directory to the directory where you want all the meta-layers to be downloaded into
6. Download the manifest
7. Download the repositories

By the end of the last step, all the necessary meta-layers should be in place, anyway, you still need to edit your **local.conf** and **bblayers.conf** to compile for picozed-zynq7 machine and using all the downloaded meta-layers.

Updating with repo

When you want your local repositories to be updated, just:

1. Open a terminal
2. Change the current directory to the directory where you ran repo init
3. Sync your repositories with upstream

Install Yocto by yourself

If you really want to download everything by hand, just clone branch *dizzy* of *meta-xilinx*:

and have a look at the README file.

To install *Eclipse*, *Qt Creator*, *cross-toolchain*, *NFS*, *TFTP*, etc., read **Yocto/OpenEmbedded** documentation, along with the other tools one.

2.5 BSP

The Board Support Package is composed by a set files, patches, recipes, configuration files, etc. This chapter gives you the information you need when you want to customize something, fix a bug, or simply learn how the all thing has been assembled.

2.5.1 U-boot

Get sources

The bootloader used by Picozed is **u-boot**. If you want to browse/modify the sources first you have to get them. There are two viable ways to do that:

- if you already built Picozed's bootloader with *Bitbake*, then you already have them on your (virtual) disk, otherwise
- you can download them.

Bitbake will place *u-boot* sources under:

this means that within the virtual machine you will find them under:

We suggest you to **don't work under Bitbake build directory**, you will pay a speed penalty and you can have troubles syncronizing the all thing. Just copy the sources some place else and do what you have to do.

If you didn't build them already with *Bitbake*, or you just want to make every step by hand, you can always get the sources from the Internet by cloning the proper repository and checking out the proper commit:

and in order to compile the u-boot, run the following commands:

The Cross-Toolchain

Suppose you modified something and you want to recompile the sources to test your patches, well, you need a cross-toolchain (see *Cross compiler* Section). If you are not working with the virtual machine, the most comfortable way to get the toolchain is to ask *Bitbake* for it:

When *Bitbake* finishes, you will find an install script under directory:

Host

path/to/build/tmp/deploy/sdk/

Install the script, and you will get under the installation directory a script to source to get your environment almost in place for compiling. The name of the script is:

Anyway, the environment is not quite right for compiling the bootloader and the Linux kernel, you need to unset a few variables:

Inside the virtual machine, the toolchain is already installed under:

In the very same directory there is a file, **environment-nofs**, that you can source that takes care of the environment for you when you want to compile the bootloader or the kernel

Build

Ok, now you a have working environment to compile *u-boot*, just do:

if you omit *-j* parameter, *make* will run one task after the other, if you specify it *make* will parallelize the tasks execution while respecting the dependencies between them. Generally, you will place a value for *-j* parameter corresponding to the double of your processor's cores number, for example, on a quad core machine you will place *-j 8*.

Once the build process is complete, you will find **u-boot** file in your sources directory, that's your binary. However, *u-boot* file alone is not able to boot the board, you are going to need a **First Stage Bootloader** and a **Bitstream** to make the board properly boot.

2.5.2 Linux Kernel

Like we saw for the *bootloader*, the first thing you need is: sources. Get them from *Bitbake* build directory (if you built the kernel with it) or get them from the Internet.

Bitbake will place the sources under directory:

If you are working with the virtual machine, you will find them under directory:

We suggest you to **don't work under Bitbake build directory**, you will pay a speed penalty and you could have troubles syncronizing the all thing. Just copy them some place else and do what you have to do.

If you didn't build them already with *Bitbake* or you just want to do make every step by hand, you can always get them from the Internet by cloning the proper repository and checking out the proper hash commit:

and by properly patching the sources:

If you don't use our SDK then use the following commands to patch the sources:

Source the script to load the proper evironment for the cross-toolchain (see *Cross compiler* Section) and you are ready to customize and compile the kernel:

Now you need compile the devicetree file:

By the end of the build process you will get **uImage** and **devicetree** under *arch/arm/boot*.

Host

~/Documents/linux-xlnx/arch/arm/boot/uImage ~/Documents/linux-xlnx/arch/arm/boot/dts/picozed-zynq7.dtb

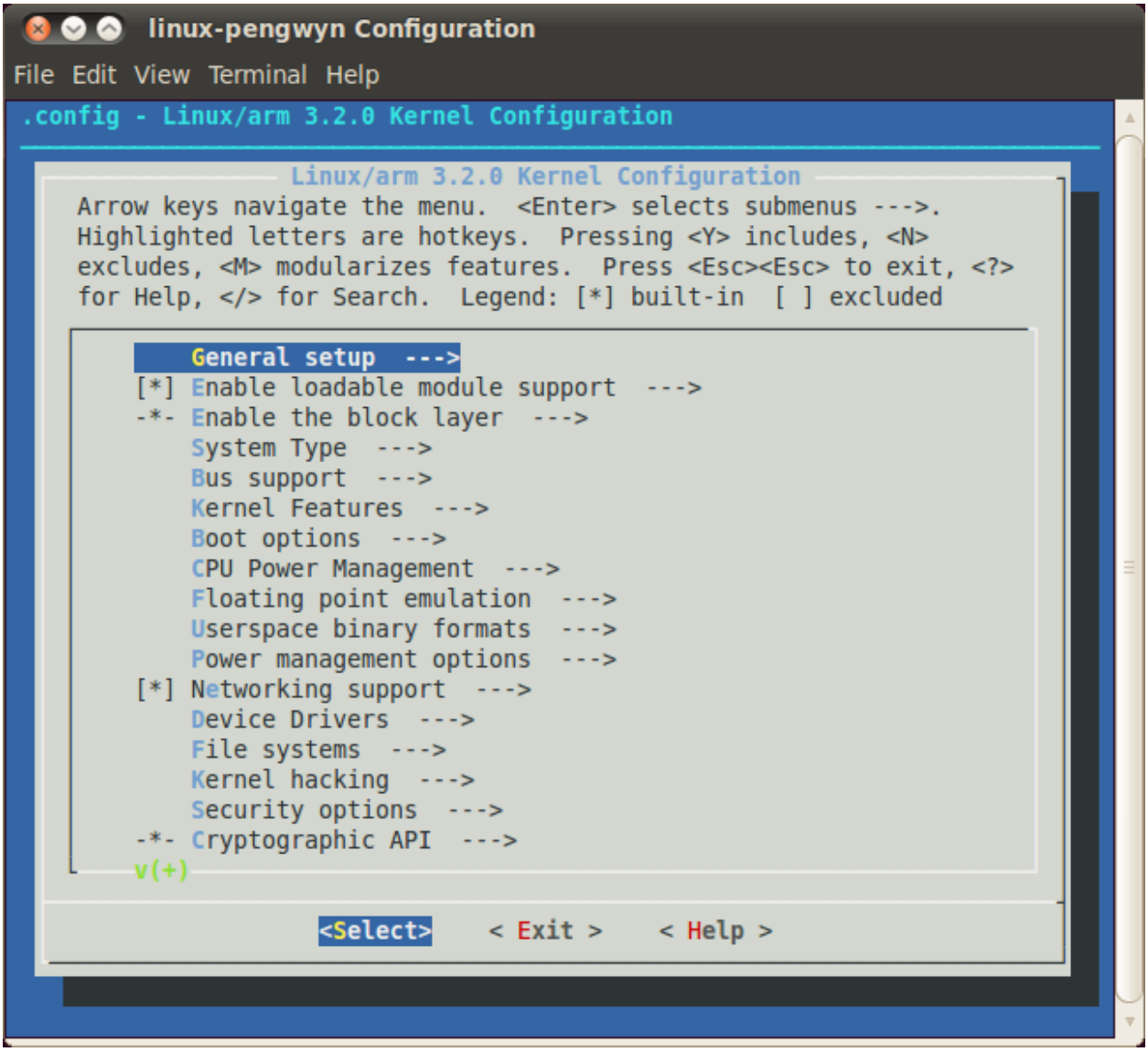
---

Enjoy!

Build from bitbake

The most frequent way of customization of the Linux Kernel is to change the .config file that contains the Kernel options. Setup the environment and run:

a new window, like the following one, will pop-up:



follow the instructions, save and exit, than you ready to generate your preferred image based on your customized kernel. If you prefer, you can build just the kernel running:

At the end of the build process, the output file (uImage.bin), along with the built kernel modules, will be placed under *tmp/deploy/images/picozed-zynq7/* inside your build directory, so, if you are building your system from the default directory, the destination directory will be */home/architech/architech_sdk/architech/picozed/yocto/build/tmp/deploy/images/picozed-zynq7/*.

2.5.3 Meta Layer

A Yocto/OpenEmbedded meta-layer is a directory that contains recipes, configuration files, patches, etc., all needed by *Bitbake* to properly "see" and build a BSP, a distribution, a (set of) package(s), whatever. **meta-xilinx** is a meta-layer which defines the BSP for Microzed device, Picozed included. You can get it with *git*:

This layer depends with this one which defines the BSP for Xilinx devices:

```
git clone git://git.yoctoproject.org/meta-xilinx.git
cd meta-xilinx/
```

git checkout 2af8d2a0e63aa371045895da03ba2bf98b51adb4

Please, refer to the *README* file contained inside the meta-layer directory.

The machine name corresponding to Picozed is **picozed-zynq7**.

2.5.4 Root FS

The final root file system will be packaged as a *.tar.gz* file that, at the end of the build process, *Bitbake* will let you find it under directory:

this means that within the SDK the actual path of the directory is:

To deploy the root file system, you are going to need an SD card with two partitions on it.

The first partition must be formatted as **FAT16**, its size must be sufficient to contain all the following files (64MB are more than enough):

1. **UBOOT.BIN**, read directly by the processor at boot, containing the *first stage bootloader*, the *bitstream* (optional), and *u-boot*
2. **uEnv.txt**, the bootscript with customizations
3. **uImage**, the Linux kernel
4. **devicetree.dtb**, the device tree binary file

To have a better understanding of those components and how to boot the board please refer to *Let's boot* Section.

The second partition, our root file system partition, can be formatted as **EXT2**.

We assume that the second partition of the SD card gets mounted (in your SDK virtual machine) under:

Warning: If that's not the case for your configuration, please find out what is the proper mounting point for such a partition on your system and replace it in the following instructions.

Untar the file corresponding to your root file system inside such a partition:

where <*image*> is the name of the recipe you used to build your root file system. For example, if you built core-image-minimal-dev with *Bitbake*, then the name of the tarball will be *core-image-minimal-dev-picozed-zynq7.tar.gz*

Important: sudo password is **architech**

2.6 Toolchain

Once your (virtual/)machine has been set up you can compile, customize the BSP for your board, write and debug applications, change the file system on-the-fly directly on the board, etc. This chapter will guide you to the basic use of the most important tools you can use to build customize, develop and tune your board.

2.6.1 Bitbake

Bitbake is the most important and powerful tool available inside Yocto/OpenEmbedded. It takes as input configuration files and recipes and produces what it is asked for, that is, it can build a package, the Linux kernel, the bootloader, an entire operating system from scratch, etc.

A **recipe** (*.bb* file) is a collection of metadata used by BitBake to set **variables** or define additional build-time **tasks**. By means of *variables*, a recipe can specify, for example, where to get the sources, which build process to use, the license of the package, an so on. There is a set of predefined *tasks* (the fetch task for example fetches the sources from the network, from a repository or from the local machine, than the sources are cached for later reuses) that executed one after the other get the job done, but a recipe can always add custom ones or override/modify existing ones. The most fine-graned operation that Bitbake can execute is, in fact, a single task.

Environment

To properly run Bitbake, the first thing you need to do is setup the shell environment. Luckily, there is a script that takes care of it, all you need to do is:

Inside the virtual machine, you can find *oe-init-build-env* script inside:

If you omit the build directory path, a directory named **build** will be created under your current working directory.

By default, with the SDK, the script is used like this:

Your current working directory changes to such a directory and you can customize configurations files (that the environment script put in place for you when creating the directory), run Bitbake to build whatever pops to your mind as well run hob. If you specify a custom directory, the script will setup all you need inside that directory and will change your current working directory to that specific directory.

Important: The build directory contains all the caches, builds output, temporary files, log files, file system images... everything!

The default build directory for Picozed is located under:

and the splash screen has a facility (a button located under Picozed's page) that can take you there with the right environment already in place so you are productive right away.

Important:

If you don't use the default build directory you need setup the local.conf file. See the paragraph below.

Configuration files

Configuration files are used by Bitbake to define variables value, preferences, etc..., there are a lot of them. At the beginning you should just worry about two of them, both located under *conf* directory inside your build directory, we are talking about **local.conf** and **bblayers.conf**.

local.conf contains your customizations for the build process, the most important variables you should be interested about are: **MACHINE**, **DISTRO**, **BB_NUMBER_THREADS** and **PARALLEL_MAKE**. *MACHINE* defines the target machine you want compile against. The proper value for Picozed is picozed-zynq7:

DISTRO let you choose which distribution to use to build the root file systems for the board. The default distribution to use with the board is:

BB_NUMBER_THREADS and *PARALLEL_MAKE* can help you speed up the build process. *BB_NUMBER_THREADS* is used to tell Bitbake how many tasks can be executed at the same time, while *PARALLEL_MAKE* contains the **-j** option to give to *make* program when issued. Both *BB_NUMBER_THREADS* and *PARALLEL_MAKE* are related to the number of processors of your (virtual) machine, and should be set with a number that is two times the number of processors on your (virtual) machine. If for example, your (virtual) machine has/sees four cores, then you should set those variables like this:

bblayers.conf is used to tell Bitbake which meta-layers to take into account when parsing/looking for recipes, machine, distributions, configuration files, bbclasses, and so on. The most important variable contained inside *bblayers.conf* is **BBLAYERS**, it's the variable where the actual meta-layers layout get specified.

All the variables value we just spoke about are taken care of by Architech installation scripts.

Command line

With your shell setup with the proper environment and your configuration files customized according to your board and your will, you are ready to use Bitbake. The first suggestion is to run:

Bitbake will show you all the options it can be run with. During normal activity you will need to simply run a command like:

for example:

Such a command will build bootloader, Linux kernel and a root file system. *core-image-minimal-dev* tells Bitbake to execute whatever recipe

you just place the name of the recipe without the extension *.bb*.

Of course, there are times when you want more control over Bitbake, for example, you want to execute just one task like recompiling the Linux kernel, no matter what. That action can be achieved with:

where *-c compile* states the you want to execute the *do_compile* task and *-f* forces Bitbake to execute the command even if it thinks that there are no modifications and hence there is no need to to execute the same command again.

Another useful option is *-e* which gets Bitbake to print the environment state for the command you ran.

The last option we want to introduce is *-D*, which can be in fact repeated more than once and asks Bitbake to emit debug print. The amount of debug output you get depend on many times you repeated the option.

Of course, there are other options, but the ones introduced here should give you an head start.

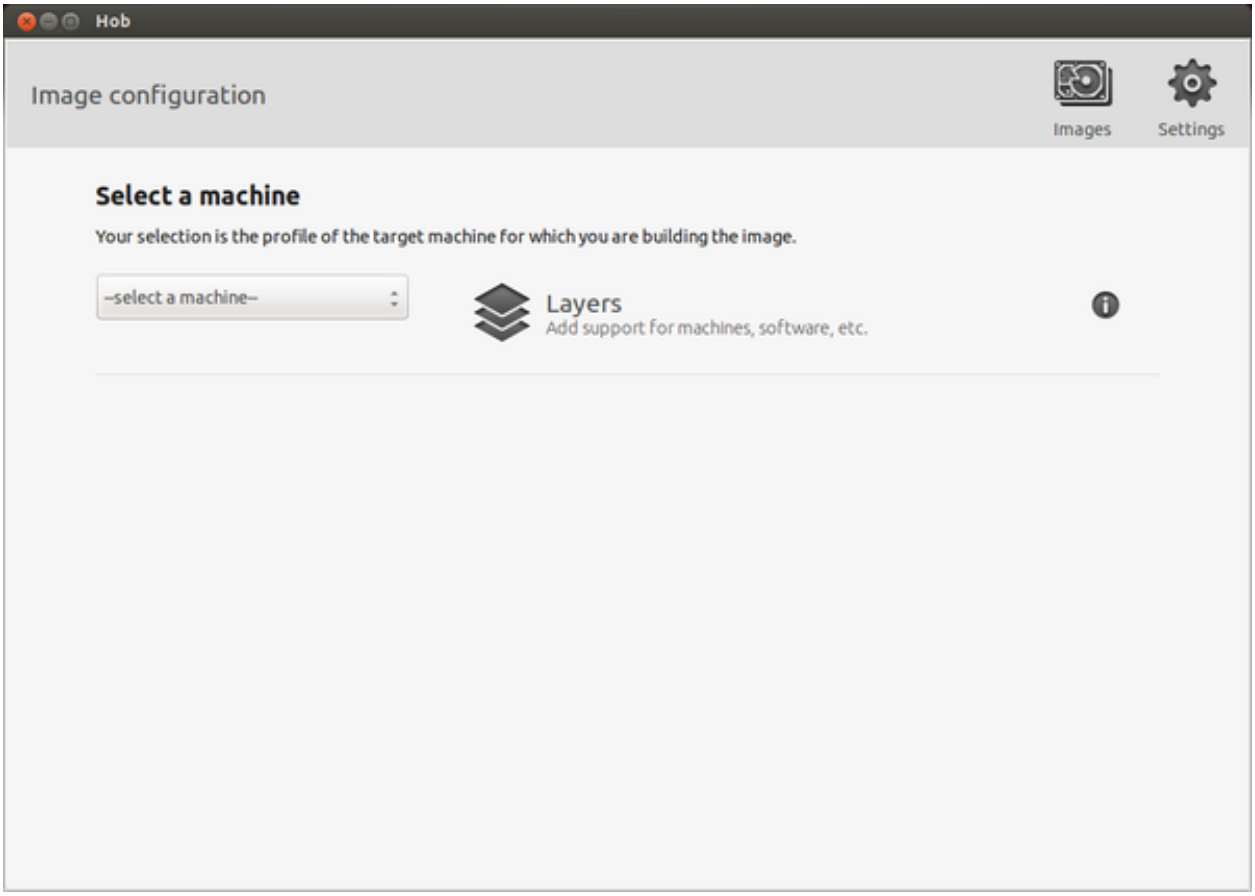
2.6.2 Hob

Hob is a graphical interface for Bitbake. It can be called once Bitbake environment has been setup (see *Bitbake*) like this:

Host

hob

once open, you are required to select the machine you want to compile against



after that, you can select the image you want to build and, of course, you can customize it.

Note:

Remove **tools-debug** from **local.conf** file in order to use correctly HOB.

2.6.3 Eclipse

Eclipse is an integrated development environment (IDE). It contains a base workspace and the Yocto plug-in system to compile and debug a program for Picozed. Hereafter, the operating system that runs the IDE/debugger will be named host machine, and the board being debugged will be named target machine. The host machine could be running as a virtual machine guest operating system, anyway, the documentation for the host machine running as a guest operating system and as host operating system is exactly the same.

To write your application you need:

- a root file system filesystem (you can use *bitbake*/*hob* to build your preferred filesystem) with development support (that is, it must include all the necessary libraries, header files, the *tcf-agent* program and *gdbserver*) included
- a media with the *root filesystem* installed and, if necessary, the bootloader
- Picozed *powered up* with the aforementioned root file system
- a working *serial console* terminal
- a working *network* connection between your workstation and the board (connector *J11*), so, be sure that:

1. your board has ip address 192.168.0.10 on interface eth0, and
2. your PC has an ip address in the same family of addresses, e.g. 192.168.0.100.

Note:

Before to start remember to copy the cross-toolchain libreries to sysroot

cp -r /home/architech/architech_sdk/architech/picozed/toolchain/sysroots/cortexa9-vfp-neon-poky-linux-gnueabi/* /home/architech/architech_sdk/architech/picozed/sysroot

---

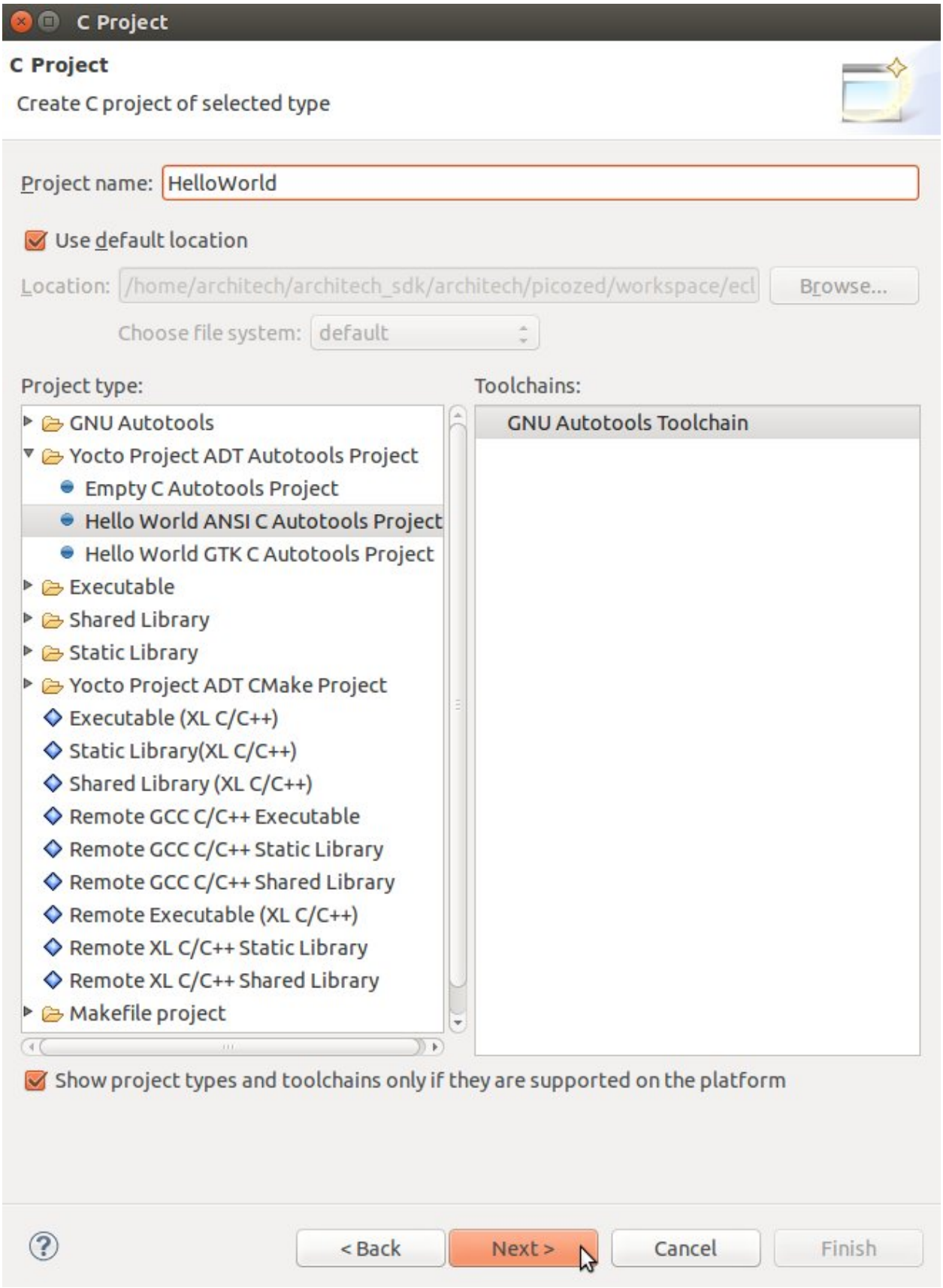
Creating the Project

You can create two types of projects: Autotools-based, or Makefile-based. This section describes how to create Autotools-based projects from within the **Eclipse IDE**. Launch Eclipse using Architech Splashscreen just click on **Develop with Eclipse**.



To create a project based on a Yocto template and then display the source code, follow these steps:

- Select File→New→Project...
- Under *C/C++*, double click on *C Project* to create the project.
- Click on "Next" button
- Expand *Yocto Project ADT Autotools Project*.
- Select *Hello World ANSI C Autotools Project*. This is an Autotools-based project based on a Yocto Project template.

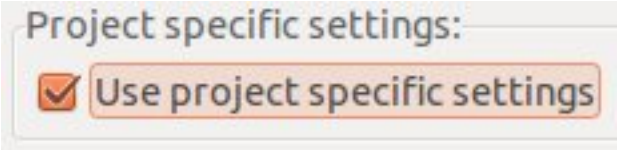

C Project
C Project
Create C project of selected type
Project name: HelloWorld
Use default location
Location: /home/architech/architech_sdk/architech/picozed/workspace/ecl
Browse...
Choose file system: default
Project type:
GNU Autotools
Yocto Project ADT Autotools Project
Empty C Autotools Project
Hello World ANSI C Autotools Project
Hello World GTK C Autotools Project
Executable
Shared Library
Static Library
Yocto Project ADT CMake Project
Executable (XL C/C++)
Static Library(XL C/C++)
Shared Library (XL C/C++)
Remote GCC C/C++ Executable
Remote GCC C/C++ Static Library
Remote GCC C/C++ Shared Library
Remote Executable (XL C/C++)
Remote XL C/C++ Static Library
Remote XL C/C++ Shared Library
Makefile project
Toolchains:
GNU Autotools Toolchain
Show project types and toolchains only if they are supported on the platform
< Back
Next >
Cancel
Finish

- Put a name in the Project *name:* field. Do not use hyphens as part of the name.
- Click *Next*.
- Add information in the *Author* and *Copyright* notice fields.
- Be sure the *License* field is correct.
- Click *Finish*.

Note: If the "open perspective" prompt appears, click *Yes* so that you enter in C/C++ perspective. The left-hand navigation panel shows your project. You can display your source by double clicking on the project source file.



- Select *Project→Properties→Yocto Project Settings* and check *Use project specific settings*

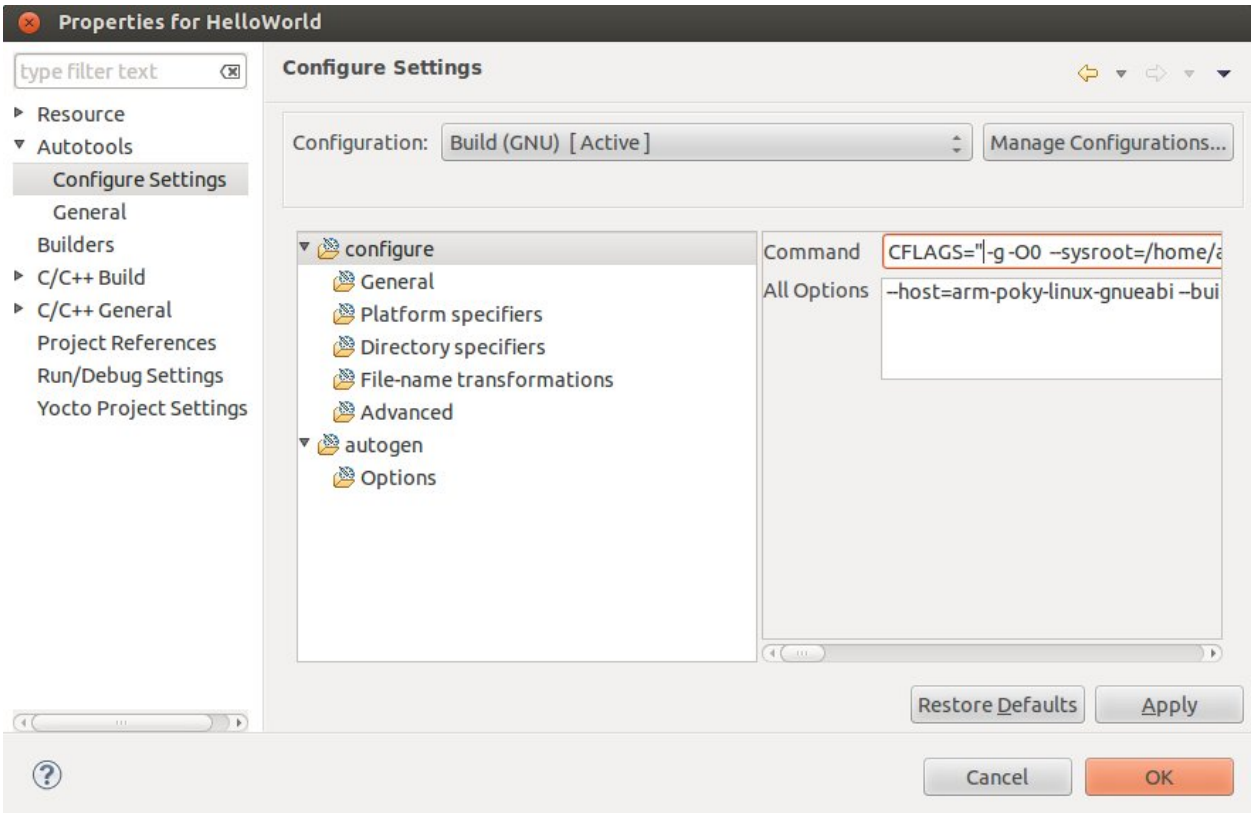


Building the Project

To build the project, select Project→Build Project. The console should update with messages from the cross-compiler. To add more libraries to compile:

- Click on Project→Properties.
- Expand the box next to Autotools.
- Select Configure Settings.
- In CFLAGS field, you can add the path of includes with -Ipath_include
- In LDFLAGS field, you can specify the libraries you use with -lname_library and you can also specify the path where to look for libraries with -Lpath_library
- Click on Project→Build All to compile the project

Note: All libraries must be located in */home/architech/architech_sdk/architech/picozed/sysroot* subdirectories.

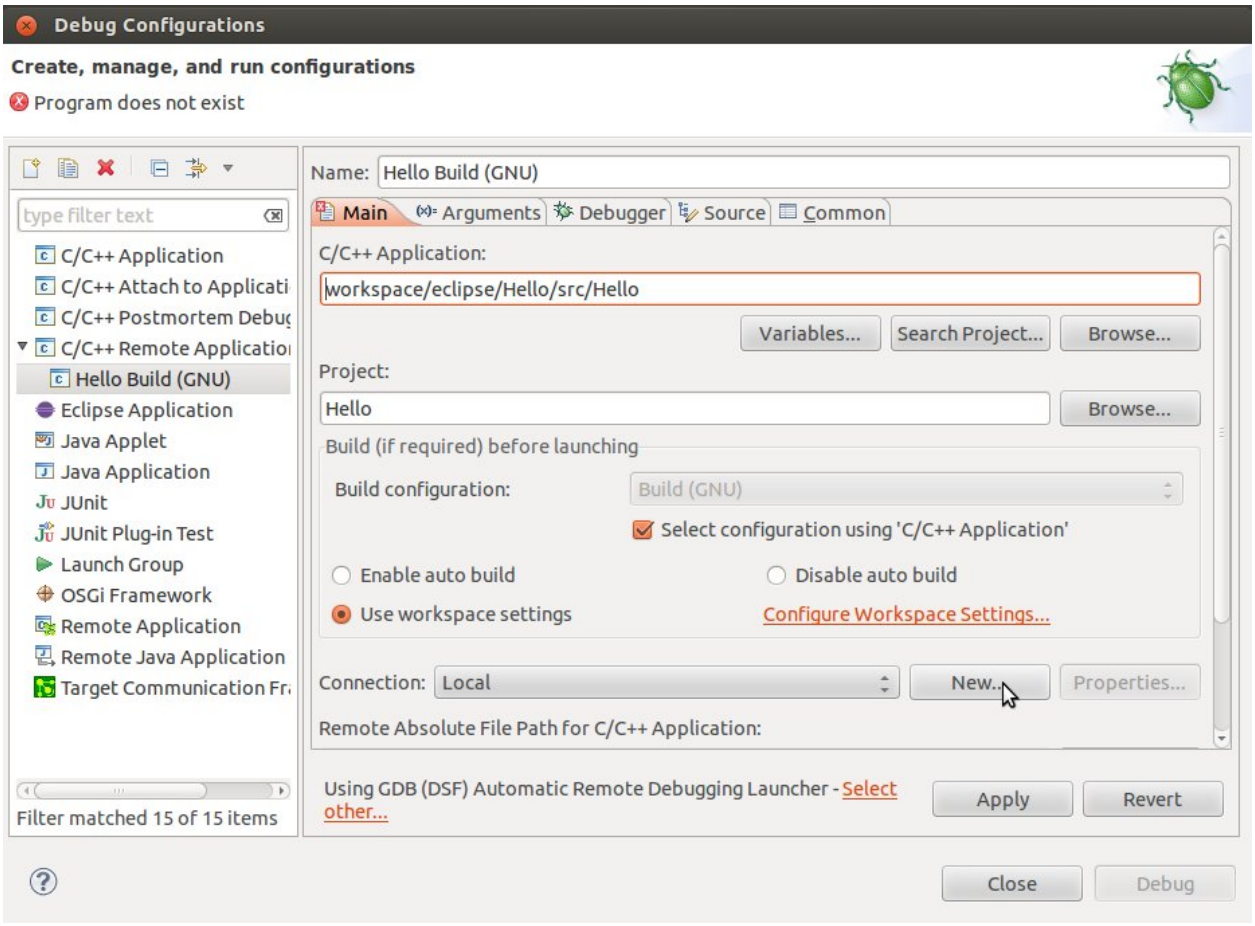


Deploying and Debugging the Application

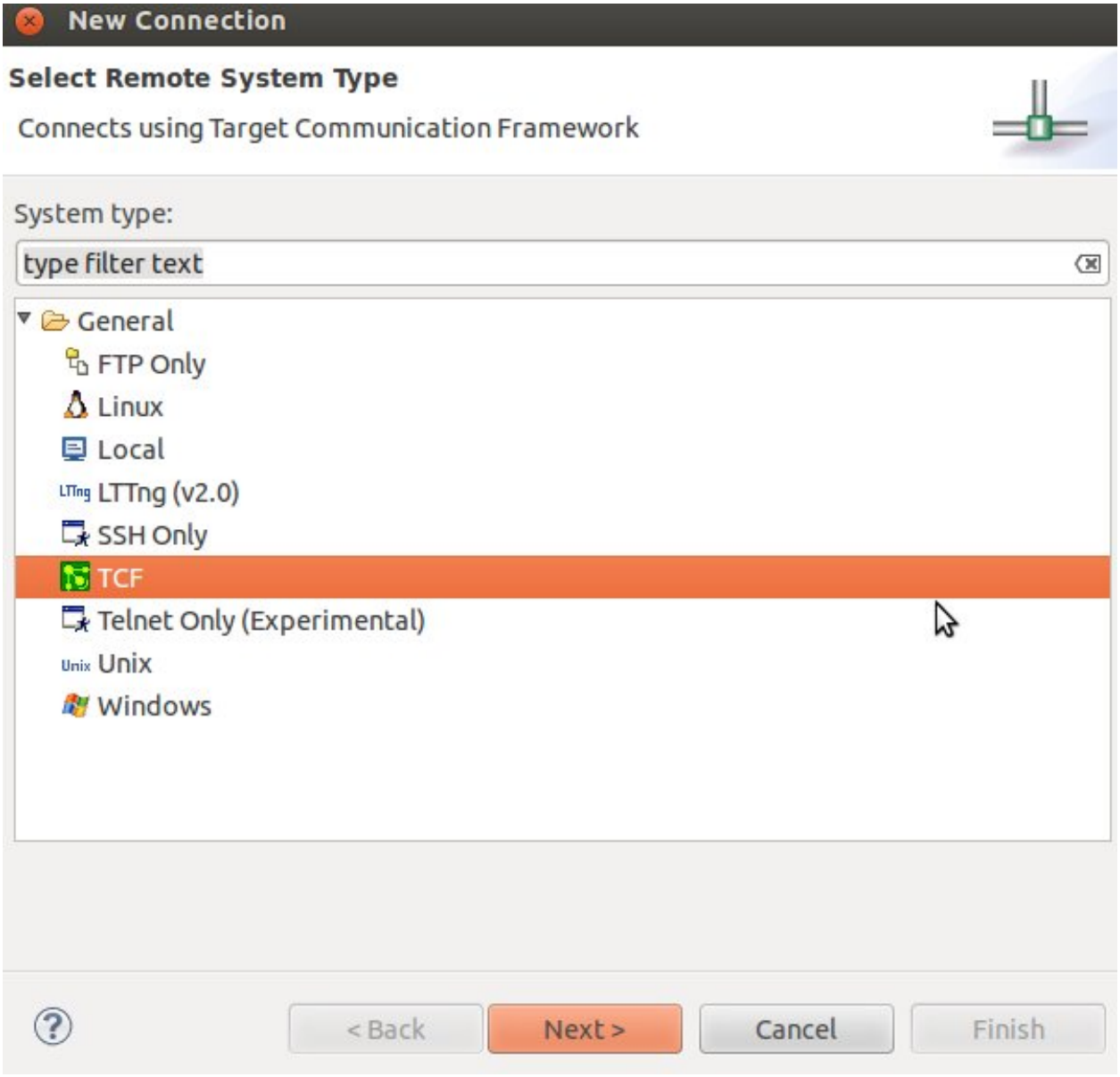
Connect Picozed console to your PC and power-on the board. Once you built the project and the board is running the image, use minicom to run **tcf-agent** program in target board:

On the Host machine, follow these steps to let **Eclipse** deploy and debug your application:

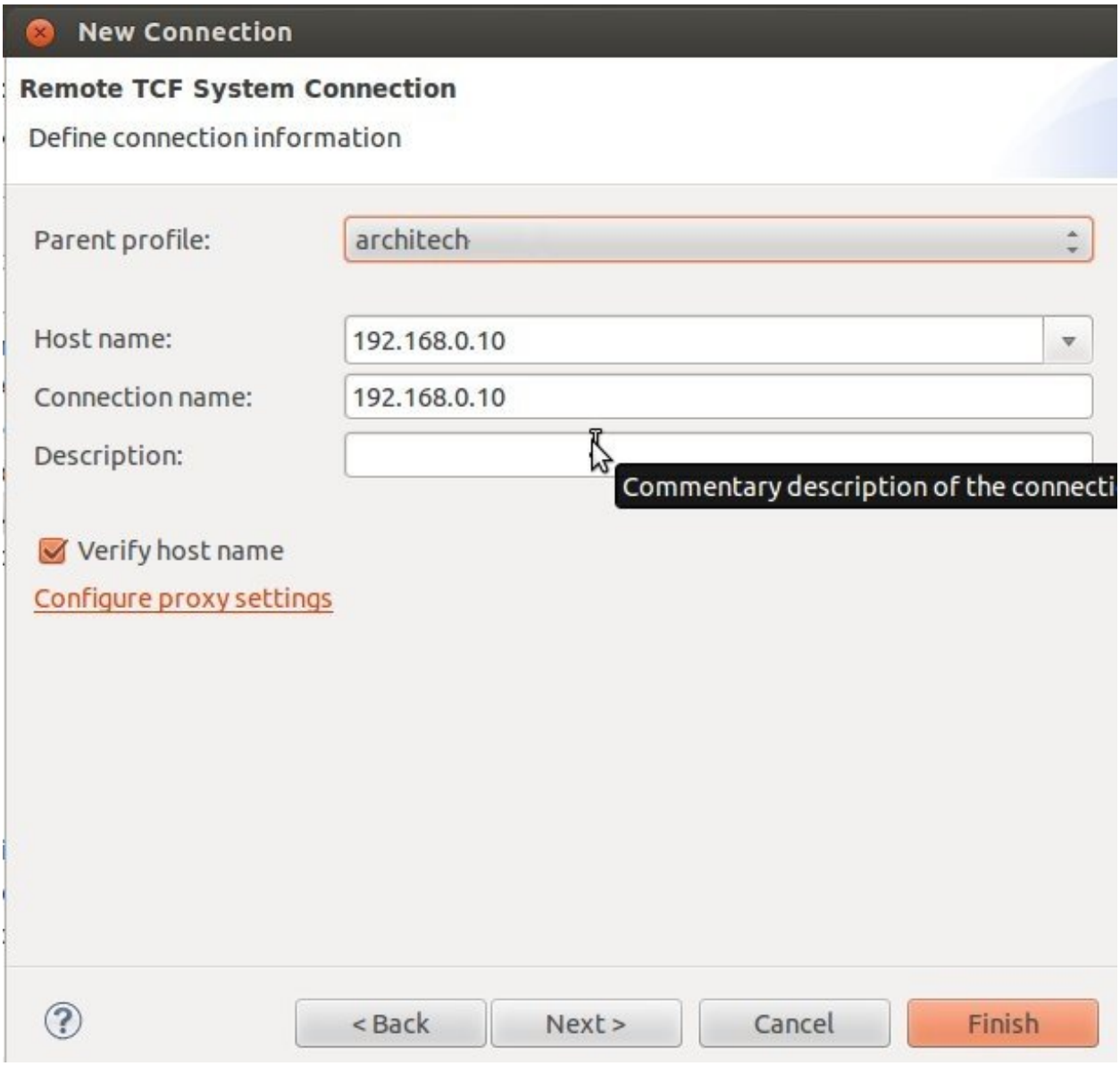
- Select Run→Debug Configurations...
- In the left area, expand *C/C++ Remote Application*.
- Locate your project and select it to bring up a new tabbed view in the *Debug Configurations* Dialog.



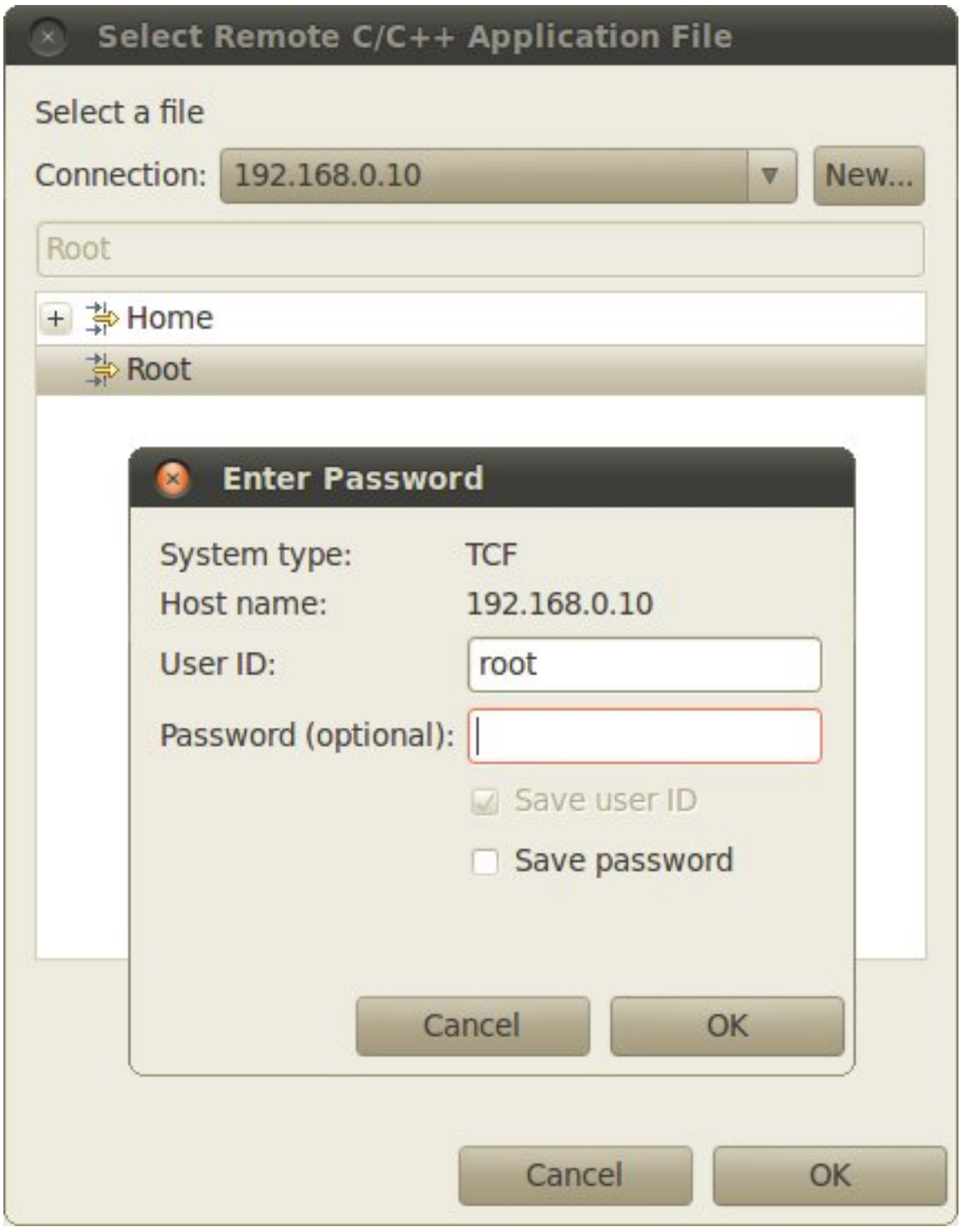
- Insert in *C/C++ Application* the filepath of your application binary on your host machine.
- Click on "New" button near the drop-down menu in the *Connection* field.
- Select *TCF* icon.



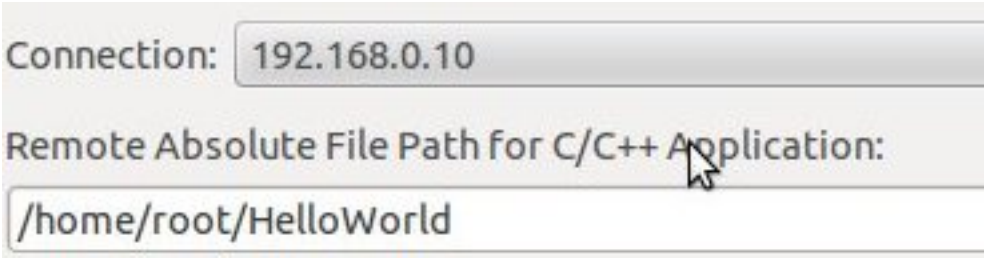
- Insert in *Host Name* and *Connection Name* fields the IP address of the target board. (e.g. 192.168.0.10)



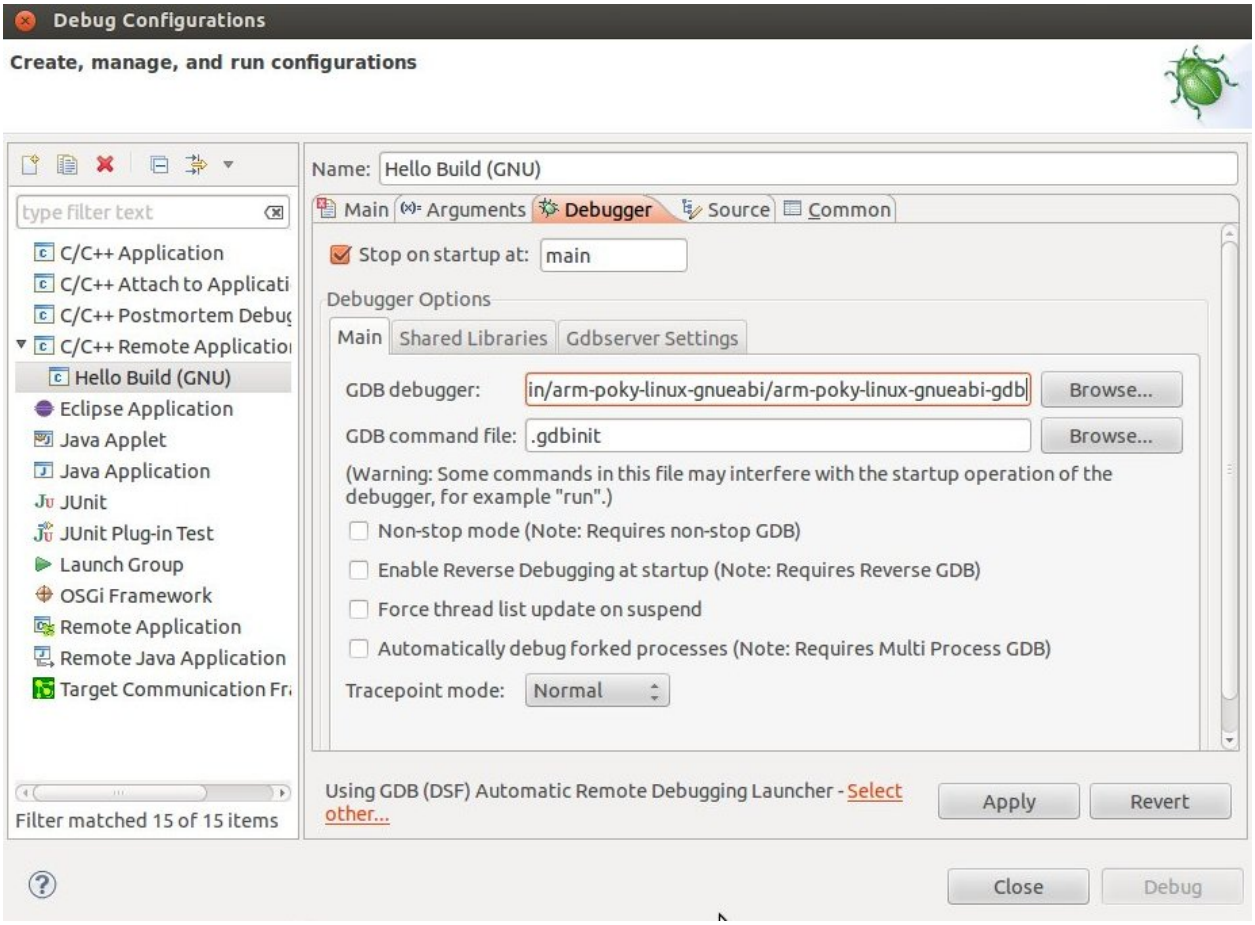
- Press *Finish*.
- Use the drop-down menu now in the *Connection* field and pick the IP Address you entered earlier.
- Enter the absolute path on the target into which you want to deploy the application. Use *Browse* button near *Remote Absolute File Path for C/C++Application:* field. No password is needed.



- Enter also in the target path the name of the application you want to debug. (e.g. HelloWorld)



- Select *Debugger* tab



- In GDB Debugger field, insert the filepath of gdb for your toolchain
- In *Debugger* window there is a tab named *Shared Library*, click on it.
- Add the libraries paths *lib* and *usr/lib* of the rootfs (which must be the same used in the target board)
- Click *Debug* to bring up a login screen and login.
- Accept the debug perspective.

Important: If debug does not work, check on the board if *tcf-agent* is running and *gdbserver* has been installed.

2.6.4 Cross compiler

Yocto/OpenEmbedded can be driven to generate the cross-toolchain for your platform. There are two common ways to get that:

or

The first method provides you the toolchain, you need to provide the file system to compile against, the second method provides both the toolchain and the file system along with -dev and -dbg packages installed.

Both ways you get an installation script.

The virtual machine has a cross-toolchain installed for each board, each generated with *meta-toolchain*. To use it just do:

to compile Linux user-space stuff. If you want to compile kernel or bootloader then do:

and you are ready to go.

2.6.5 Opkg



Opkg (Open PacKaGe Management) is a lightweight package management system. It is written in C and resembles apt/dpkg in operation. It is intended for use on embedded Linux devices and is used in this capacity in the OpenEmbedded and OpenWrt projects.

Useful commands:

- update the list of available packages:
- list available packages:
- list installed packages:
- install packages:
- list package providing <file>
- Show package information
- show package dependencies:
- remove packages:

Force Bitbake to install Opkg in the final image

With some images, *Bitbake* (e.g. *core-image-minimal*) does not install the package management system in the final target. To force *Bitbake* to include it in the next build, edit your configuration file

and add this line to it:

Create a repository

opkg reads the list of packages repositories in configuration files located under */etc/opkg/*. You can easily setup a new repository for your custom builds:

1. Install a web server on your machine, for example **apache2**:
2. Configure apache web server to "see" the packages you built, for example:
3. Create a new configuration file on the target (for example */etc/opkg/my_packages.conf*) containing lines like this one to index the packages related to a particular machine:

To actually reach the virtual machine we set up a port forwarding mechanism in Chapter *Virtual Machine* so that every time the board communicates with the workstation on port 8000, VirtualBox actually turns the communication directly to the virtual machine operating system on port 80 where it finds *apache* waiting for it.

4. Connect the board and the personal computer you are developing on by means of an ethernet cable
5. Update the list of available packages on the target

Update repository index

Sometimes, you need to force bitbake to rebuild the index of packages by means of:

2.7 The board

This chapter introduces the board, its hardware and how to boot it.

2.7.1 Hardware

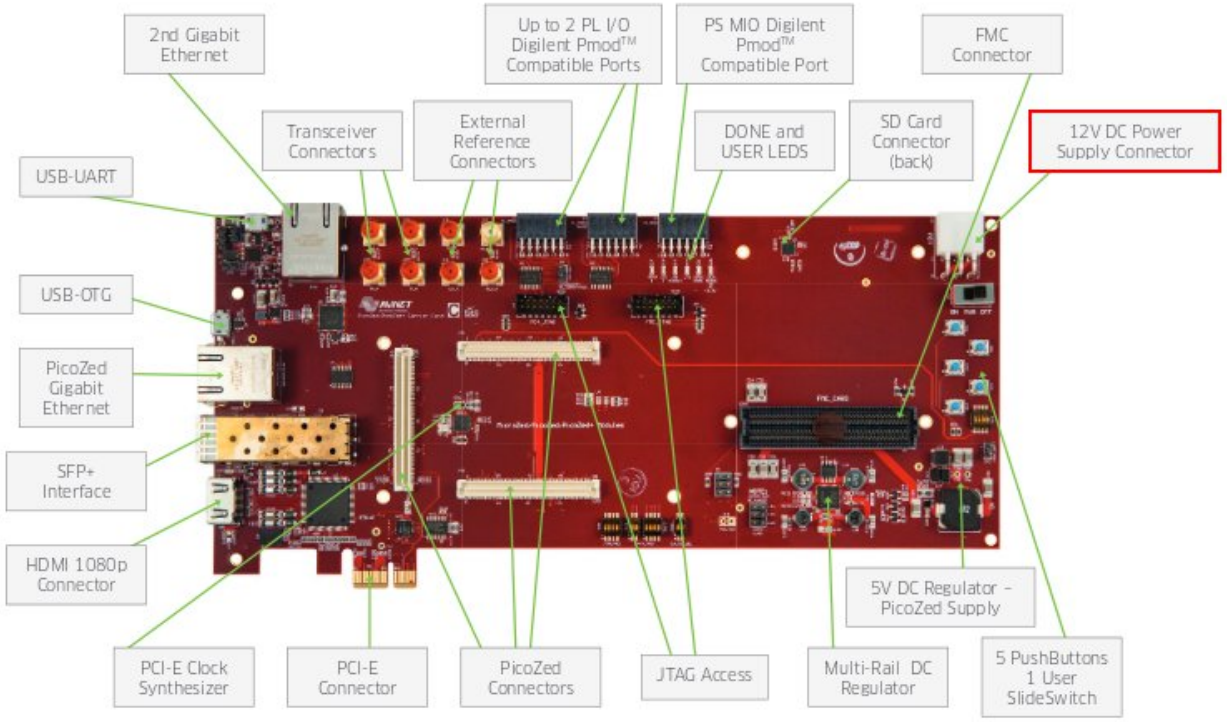
The hardware documentation of Picozed can be found here:

http://picozed.org/support/documentation/4701

In order to use the Picozed you have to plugin in it into the carrier board.

2.7.2 Power-On

In order to use the Picozed you have to plugin in it into the carrier board. Picozed takes the power from the connector *J14* if the carrier board. The board is shipped with an external power adapter.

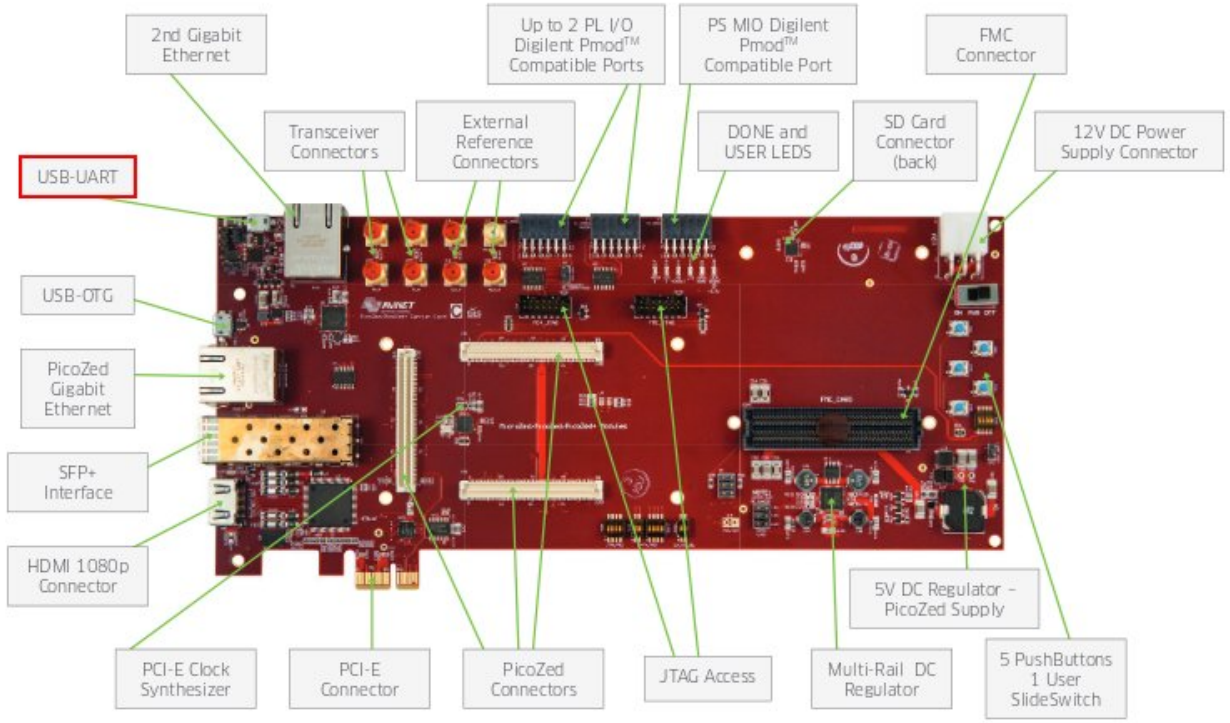


On connector *J6* you can have the serial console, so, during your daily development use, you would just connect your workstation to the board using the micro-USB.

Warning: Connect the micro USB only **AFTER** have powered the board.

2.7.3 Serial Console

On Picozed the USB-UART port is used for serial console.



which you can connect, by means of a micro-USB cable, to your personal computer.

Note: Every operating system has its own killer application to give you a serial terminal interface. In this guide, we are assuming your **host** operating system is **Ubuntu**.

On a Linux (Ubuntu) host machine, the console is seen as a ttyACMX device and you can access to it by means of an application like *minicom*.

Minicom needs to know the name of the serial device. The simplest way for you to discover the name of the device is by looking to the kernel messages, so:

1. clean the kernel messages
2. connect the mini-USB cable to the board already powered-on
3. display the kernel messages
3. read the output

As you can see, here the device has been recognized as **/dev/ttyUSB0**.

Now that you know the device name, run *minicom*:

If minicom is not installed, you can install it with:

then you can setup your port with these parameters:

If on your system the device has not been recognized as */dev/ttyUSB0*, just replace */dev/ttyUSB0* with the proper device.

Once you are done configuring the serial port, you are back to *minicom* main menu and you can select *exit*.

2.7.4 Let's boot

In order to use the Picozed you have to plugin in it into the carrier board.



To properly boot the board, you need several components:

- an **SD card**,
- file **BOOT.BIN**, (if you don't want to use u-boot from the flash)
- the **Linux kernel image**,
- the **device tree blob**,
- a **bootscript**, and
- a **root file system**.

This section will guide you through all that stuff.

SD card

The SD card has to be prepared with two partitions:

- a *FAT16* partition able to contain the following files: *BOOT.BIN*, *uEnv.txt*, *uImage* and *devicetree.dtb*. 64MB will be more than enough
- an *EXT2* partition to contain the root file system

The SD card has to be inserted in **J6**.

Furthermore, on the board there is a set of switches related to the boot process. Verify Picozed boot mode (JP1, JP2 and JP3) jumpers are set like in the following picture:



that means *boot from SD card*, as described in the Hardware User Guide:

http://picozed.org/support/documentation/4701

BOOT.BIN

To boot the board, the first thing you should care about is the *boot file*. The *boot file* is composed by up to three components:

1. a **First Stage BootLoader**,
2. a valid **Bitstream** (this is optional), and
3. **u-boot**.

Its name is **BOOT.BIN**.

This guide won't treat the creation of the *Bitstream* and the *First Stage BootLoader*. For information about those components, please refer to Xilinx's official documentation:

- Bitstream,
- First Stage BootLoader.

A guide on how to prepare file **BOOT.BIN** is available here:

http://www.wiki.xilinx.com/Prepare+boot+image

This SDK doesn't relies on a pre-generated *BOOT.BIN*, you need build one.

BOOT.BIN has to be copied in the first partition of the SD card you use to boot the board.

Kernel

Yocto generates the Linux kernel image ready to be deployed on the board when you build *virtual/kernel* or an image (see *Bitbake* Section for more information on how to use *Bitbake*). You will find it inside directory:

uImage has to be copied to the first partition of the SD card.

Device Tree

The Flattened Device Tree (FDT) is a data structure for describing the hardware in a system. It is a derived from the device tree format used by *Open Firmware* to encapsulate platform information and convey it to the operating system. The operating system uses the FDT data to find and register the devices in the system.

The Device Tree Source (*.dts*) file is a text file containing the specification. The Device Tree Blob file (*.dtb*) is the blob version of the source one, and it is passed to the Linux Kernel at boot.

You can get the Device Tree Blob file from here:

Download devicetree.dtb

You can get the corresponding Device Tree Source from here:

Download devicetree.zip

Bootscript

With our flow, the default settings of *u-boot* will cause the Linux boot process to fail. You need to customize it by means of a file named **uEnv.txt** with these commands in it:

Important: Make sure uEnv.txt is terminated by an empty line.

If you prefer, you can download file *uEnv.txt* from here:

Download file uEnv.txt

u-boot will look for *uEnv.txt* automatically at boot.

uEnv.txt has to be copied to the first partition of the SD card.

RootFS

Every time you build an image recipe with *Bitbake* you get a root file system. All the built root file systems are stacked under directory:

To deploy the root file system, clear the second partition of the SD card and untar the root file system tarball Yocto generated directly to the second partition of the SD card.

Compose the SD card

Warning: The following instruction will make you overwrite your SD card content, it will be lost forever! If you have important data on it, make sure you do a backup of your data on the SD card before catching up with the next steps.

To sum up, the first time you create your SD card, create two partitions on it. The first one has to be a *FAT16* (name it **boot**), 64MB will be more than enough. Create the second partition as an *EXT2* (name it **rootfs**), make it big enough to fill the free space on the disk size.

You are going to need the following files: BOOT.BIN, uEnv.txt, uImage, devicetree.dtb, <*image*>-picozed-zynq7.tar.gz. <*image*> is the recipe name used to build your image, for example: core-image-minimal-dev, so that the rootfs tarball name would be core-image-minimal-dev-picozed-zynq7.tar.gz.

Now, we assume that the first partition of the SD card gets mounted (in your SDK virtual machine) under:

while the second partition gets mounted under:

> **Warning:** If that's not the case for your configuration, please find out which are the proper mounting points for those two partitions on your system and replace them in the following instructions.

Furthermore, we assume you previously downloaded files *BOOT.BIN*, *uEnv.txt*, and *devicetree.dtb* inside directory:

Ok then, we can finally deploy bootloader and kernel on the first partition of the SD card:

and the root file system on the second partition of the SD card:

If you just need to install a new root file system on your SD card, you can execute just the last step.

2.7.5 Network

Within Linux, you can see the network interface as **eth0**.

2.8 FAQ

2.8.1 Virtual Machine

What is the password for the default user of the virtual machine?

The password for the default user, that is **architech**, is:

Host

architech

What is the password of sudo?

The default password of **architech** is **architech**. If you are searching more information about **sudo** command please refer to *sudo* section of the *appendix*.

What is the password for user root?

By default, Ubuntu 14.04 32bit comes with no password defined for **root** user, to set it run the following command:

Host

sudo passwd root

Linux will ask you (twice, the second time is just for confirmation) to write the password for user root.

What are device files? How can I use them?

Please refer to *device files* section of the *appendix*.

I have problems to download the vm, the server cut down the connection

The site has limitation in bandwith. Use download manager and do not try to speed up the download. If you try to download fastly the server will broke up your download.

2.8.2 Picozed

2.9 Appendix

In this page you can find some useful info about how Linux works. If you are coming from Microsoft world, the next paragraphs can help you to have a more soft approach to Linux world.

2.9.1 sudo command

sudo is a program for Unix-like computer operating systems that allows users to run programs/commands with the security privileges of another user, normally the superuser or root. Not all the users can call sudo, only the **sudoers**, **architech** (the default user of the virtual machine) user is a sudoer. When you run a command preceeded by sudo Linux will ask you the user password, for **architech** user the password is **architech**.

2.9.2 Device files

Under Linux, (almost) all hardware devices are treated as files. A device file is a special file which allows users to access an hardware device by means of the standard file operations (open, read, write, close, etc), hiding hardware details. All device files are in */dev* directory. In order to access a filesystem in Linux you first need to mount it. Mounting a filesystem simply means making the particular filesystem accessible at a certain point in the Linux directories tree. In Linux, memory cards are generally named starting with *mmcblk*. For example if you insert 2 memory cards in 2 different slots of the same computer, Linux will create 2 device files:

The number identifies a specific memory card. A memory card itself can have one or more partitions. Even in this case, Linux will create a device file for every partition present in the sd card. So, for example if the "mmcblk0" countains 3 partitions, the operating system will add these files under */dev* directory:

Not all devices are named according to the aforementioned naming scheme. For example, usb pens and hard disks are named with *sd* followed by a letter which is incremented every time a new device gets connected (starting with *a*), as opposed to the naming scheme adopted by SD cards where a number (starting with *0*) was incremented. A machine with an hard disk and two pen drives would tipically have the following devices:

Usually */dev/sda* file is the primary hard disk (this might depend on your hardware).

As memory cards, the pen can have one or more partitions, so if for example we have a pen drive which has been recognized as *sdc*, and the pen drive has 2 partitions on it, we will have the following device files:

Commands like mount, umount, dd, etc., use partition device files. FIXME mkfs

> **Warning:**
> Be very careful when addressing device files, addressing the wrong one may cost you the loss of important data

2.9.3 Disks discovery

When dealing with plug and play devices, it is quite comfortable to take advantage of **dmesg** command. The kernel messages (printk) are arranged into a ring buffer, which the user can be easly access by means of **dmesg** command.

Every time the kernel recognizes new hardware, it prints information about the new device within the ring buffer, along with the device filename. To better filter out the information regarding the plug and play device we are interested in, it is better if we first clean up the ring buffer:

now that the ring buffer has been emptied, we can plug the device and, after that, display the latest messages from the kernel:

On the Ubuntu machine (with kernel version *3.2.0-65-generic*) this documentation has been written with, we observed the following messages after inserting a pen drive:

As you can see, the operating system have recognized the usb device as *sdb* (this translates to */dev/sdb*) and its only partition as *sdb1* (this translates to */dev/sdb1*)

The most useful command to gather information about mass storage devices and related partitions is **fdisk**. On the very same machine of the previous example, the execution of this command:

produces the following output:

The machine has two mass storage devices, a 500GB hard disk and a 1GB USB pen disk. As you can see from the output, *sudo fdisk -l* command lists information regarding the disks seen by the kernel along with the partitions found on them, disk after disk. The first disk (sda) presented by *fdisk* is the primary hard disk (where Linux is running), it has 4 partitions, two of which (sda1 and sda2) are used by a Microsoft operating system while the other two (sda3 and sda4) are used by a Linux operating system. The second disk (sdb) depicted by *fdisk* is an USB disk with a single FAT32 partition (sdb1)

As already stated, in order to access a filesystem in Linux you first need to mount it. Mounting a partition means binding a directory to it, so that files and directories contained inside the partition will be available in Linux filesystem starting from the directory used as mount point.

2.9.4 mount command

Suppose you want to read a file named *readme.txt* which is contained inside the USB disk of the previous example, in the main directory of the disk. Before accessing the device you must understand if it is already mounted. **mount** is the command that lets you control the mounting of filesystems in Linux. It is a complex command that permits to mount different devices and different filesystems. In this brief guide we are using it only for a very common use case. Launching **mount** without any parameter lists all mounted devices with their respective mounting points. Every line of the list, describes the name of the mounted device, where it has been mounted (path of the directory in the Linux filesystem, that is the mount point), the type of filesystem (ext3, ext4, etc.), and the options used to mount it (read and write permissions,etc.). Launching the command on the same machine of the previous section example, we don't find the device */dev/sdb1*.

```
$ mount
/dev/sda2 on /media/windows7 type fuseblk (rw,noexec,nosuid,nodev,allow_other,blksize=4096)
/dev/sda3 on / type ext4 (rw,errors=remount-ro)
proc on /proc type proc (rw,noexec,nosuid,nodev)
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
none on /sys/fs/fuse/connections type fusectl (rw)
none on /sys/kernel/debug type debugfs (rw)
none on /sys/kernel/security type securityfs (rw)
udev on /dev type devtmpfs (rw,mode=0755)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=0620)
tmpfs on /run type tmpfs (rw,noexec,nosuid,size=10%,mode=0755)
none on /run/lock type tmpfs (rw,noexec,nosuid,nodev,size=5242880)
none on /run/shm type tmpfs (rw,nosuid,nodev)
binfmt_misc on /proc/sys/fs/binfmt_misc type binfmt_misc (rw,noexec,nosuid,nodev)
```

rpc_pipefs on /run/rpc_pipefs type rpc_pipefs (rw)
vmware-vmblock on /run/vmblock-fuse type fuse.vmware-vmblock
(rw,nosuid,nodev,default_permissions,allow_other)
gvfs-fuse-daemon on /home/roberto/.gvfs type fuse.gvfs-fuse-daemon (rw,nosuid,nodev,user=roberto)

This tells us that the USB disk has not been mounted yet.

The mount operation requires three essential parameters: - the device to mount - the directory to associate - the type of filesystem used by the device

Thanks to the previously introduced **fdisk** command, we know the partition to mount (*/dev/sdb1*) and the type of filesystem used (FAT32). The directory to bind can be anything you like, by convention the user should mount his own devices under */media* or */mnt*. We haven't created it yet, so:

At this point, we have the information we need to execute the mounting. To semplify our life, we leave the duty of understanding what filesystem is effectively used by the device to the **mount** command by using option *-t auto* (if we would have wanted to tell mount exactly which filesystem to use we would have written *-t vfat*), like

The partition is now binded to */media/usbdisk* directory and its data are accessible from this directory.

now we can open the file, read it and, possibly, modify it.

When you want to disconnect the device, you need the inverse operation of **mount** which is **umount**. This command saves all data still contained in RAM (and waiting to be written on the device) and unbind the directory from the device file.

Once the directory */media/usbdisk* is unmounted it's empty, feel free to delete it if doesn't interest you anymore. It is now possible to remove the device from the machine.

What if you wanted to know the amount of free disk space available on a mounted device?

df command shows the disk space usage of all currently mounted partitions. For every partition, **df** prints its device file, size, free and used space, and the partition mount point. On our example machine we have:

-h option tells **df** to print sizes in human readable format.

D

Debug, 72

P

Project, 69