Hachiko Documentation

Release A

Architech

Mar 16, 2017

Contents

1 Notations **3**

2 Chapters **5**

2.1 Unboxing 5
2.2 Quick start guide 10
2.3 SDK Architecture 44
2.4 Create SDK 64
2.5 BSP 65
2.6 Toolchain 69
2.7 The board 83
2.8 FAQ 91
2.9 Appendix 93

Version 1.1.0A

Copyright (C)2016 Avnet Silica company

Date 08/08/2014

You can find the previous documentation: Here

Hachiko board is a deeply embedded board based on Renesas's RZ/A1 processor, which has a huge internal random access memory, enough to run a Linux distribution directly without the help of memory external to the SoC. The distribution you can run from within the internal RAM must be custom tailored for the final application. If you want to have more flexibility, the board allows you to mount an external memory chip so you can extend the amount of system memory. From the point of view of the SDK, there are differences regarding Linux kernel configuration, cross-toolchain, how the Linux distribution has to be composed, etc., hence there are two different SDKs for the two different configurations and two different documentations.

Important: We will call **Hachiko/SDRAM** the configuration that employs external memory, while we will call **Hachiko** the configuration without it.

This documentation is about **Hachiko**.

Have you just received your Hachiko 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 Hachiko 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*.

CHAPTER 2

Chapters

Unboxing

This amazing board gets the power directly from the USB, so it comes with no external power supply.

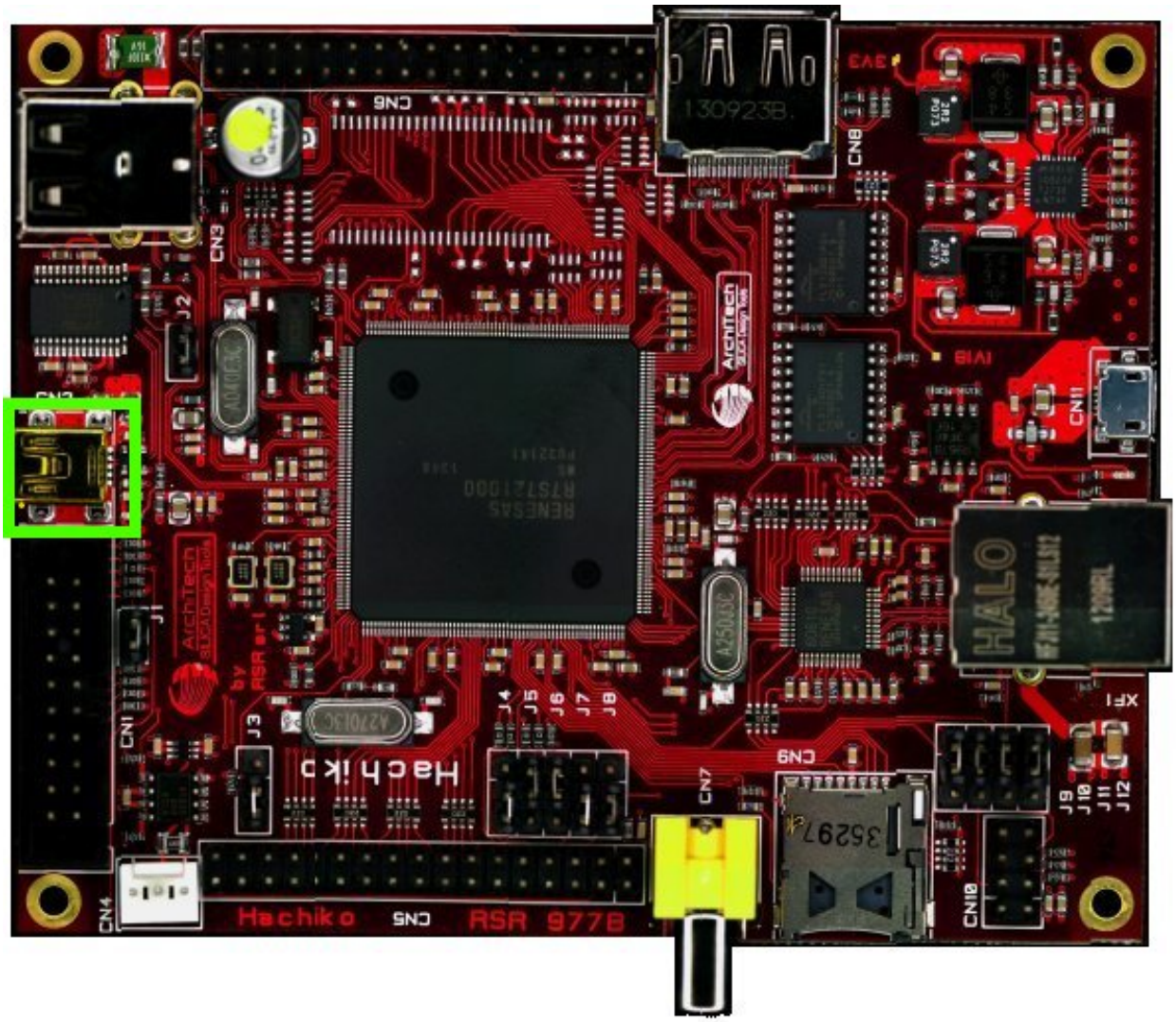
This is what the box looks like

And this is the content of box

The board itself has been programmed to boot to a Linux console of a really tiny distribution,
custom tailored to fit the 10MB of internal ram of Renesas's SoC.
First of all, make sure the board can boot from the pen drive by setting the jumpers J4 short pin2-3, J5 short pin1-2, J6 short pin1-2, J7 short pin 2-3 and J8 short pin2-3 with this configuration:

Shall we power on the board for the first time? Of course!

On Hachiko there is the dedicated serial console connector **CN2**



which you can connect, by means of a mini-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 ttyUSBX 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

Host

sudo dmesg -c

---

2. connect the mini-USB cable to the board
3. display the kernel messages

Host

dmesg

---

3. read the output

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

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

Host

sudo minicom -ws

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

Host

sudo apt-get install minicom

---

then you can setup your port with these parameters:

```
+-----------------------------------------------------------------------+
| A -    Serial Device      : /dev/ttyUSB0                              |
| B - Lockfile Location     : /var/lock                                 |
| C -   Callin Program      :                                           |
| D -  Callout Program      :                                           |
| E -    Bps/Par/Bits       : 115200 8N1                                |
| F - Hardware Flow Control : No                                        |
| G - Software Flow Control : No                                        |
|                                                                       |
|    Change which setting?                                              |
+-----------------------------------------------------------------------+
```

```
        | Screen and keyboard      |
        | Save setup as dfl        |
        | Save setup as..          |
        | Exit                     |
        | Exit from Minicom        |
        +--------------------------+
```

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

Give *root* to the login prompt:

Board

hachiko login: root

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.

Enjoy!

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.

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, Hachiko 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

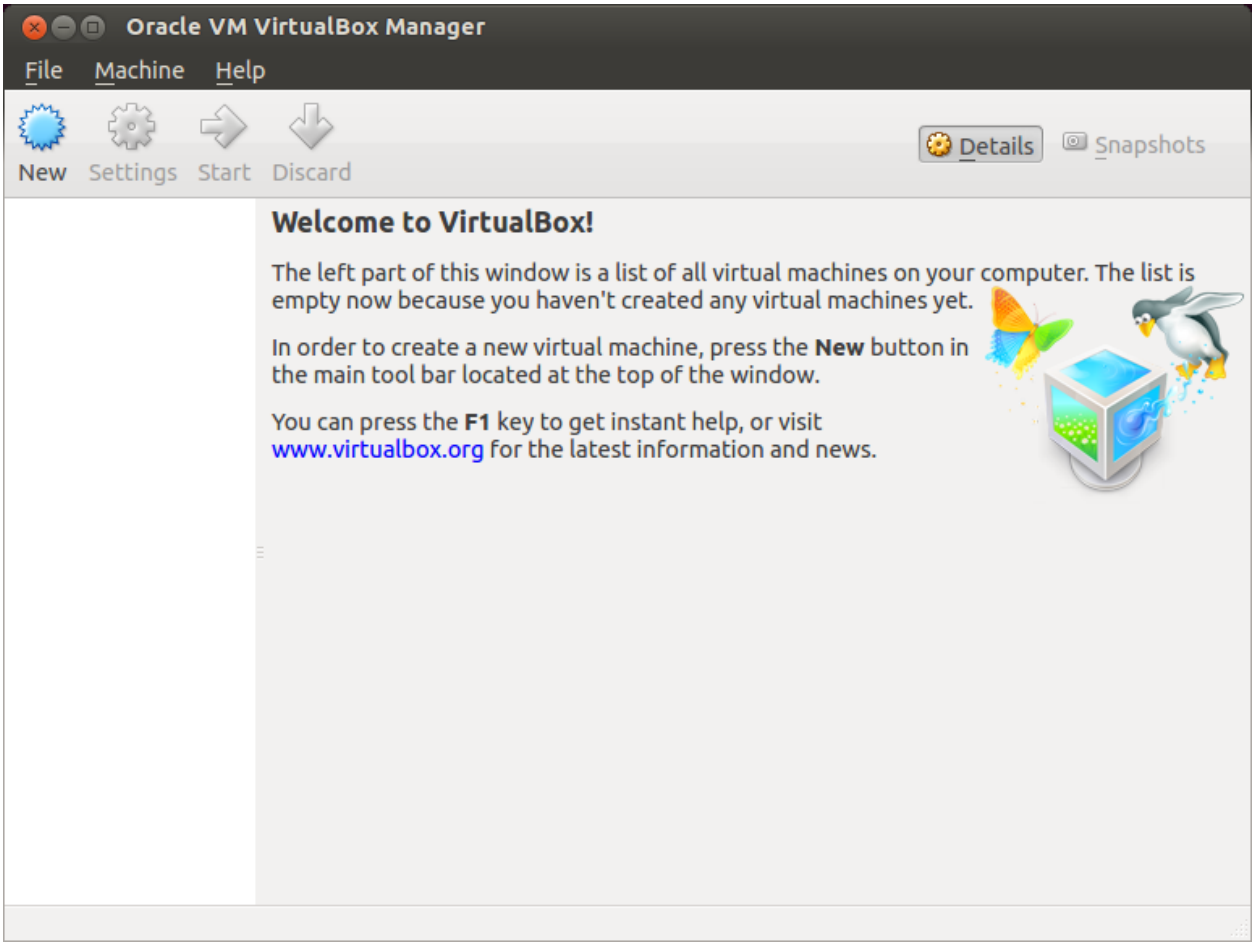
Download the version that suits your host operating system. You need to download and install the **Extension Pack** as well.

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

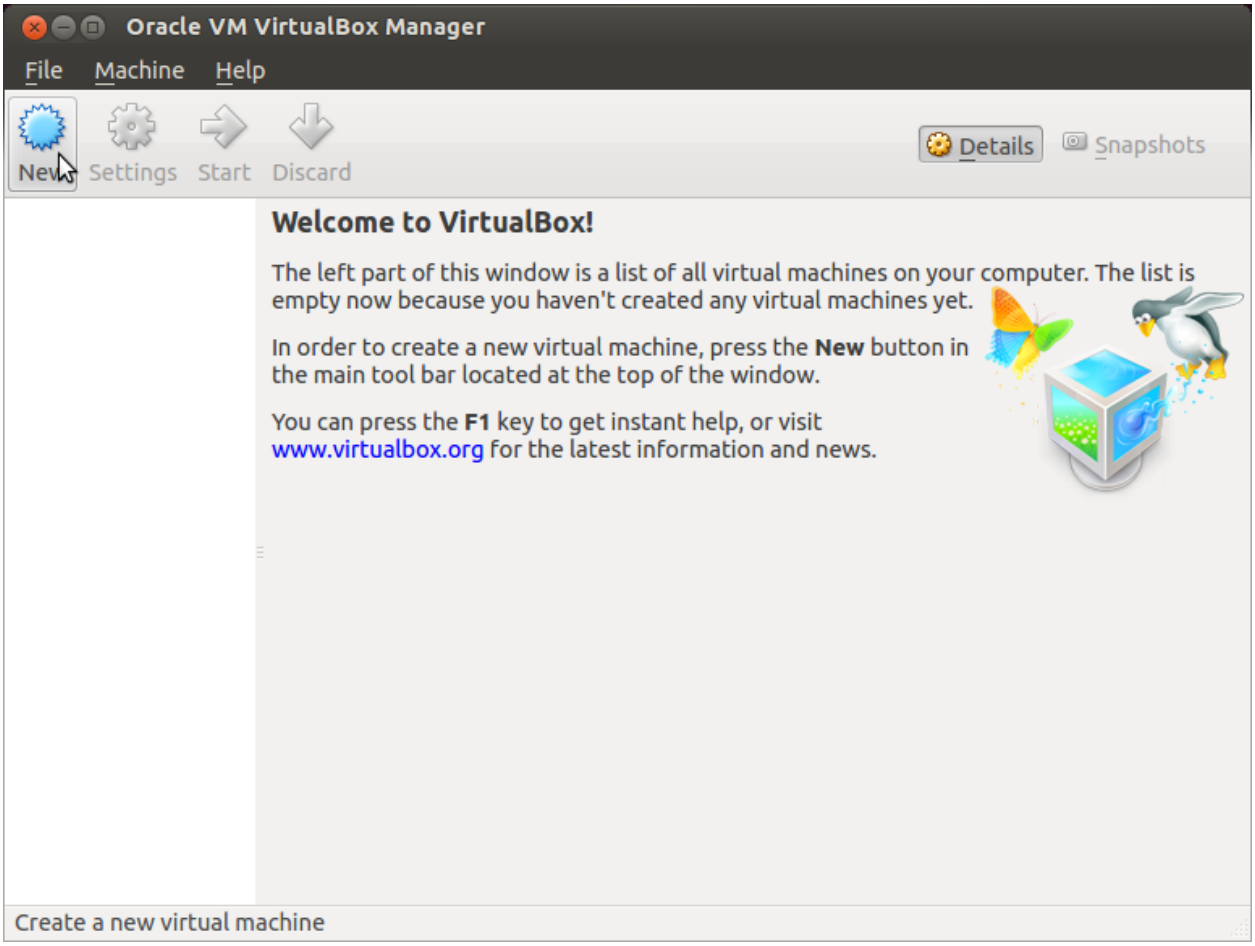
Install the software with all the default options.

Create a new Virtual Machine

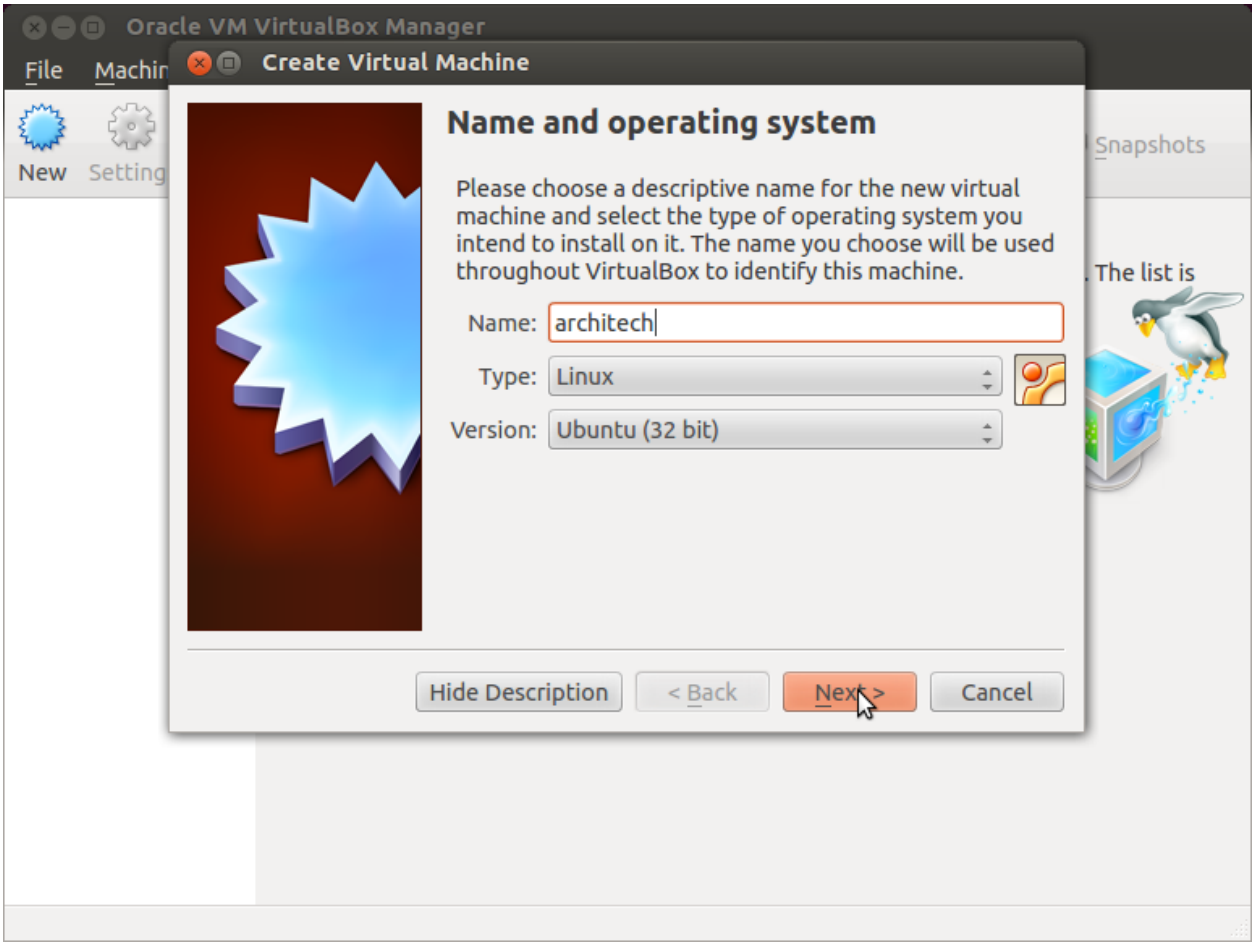
1. Run VirtualBox



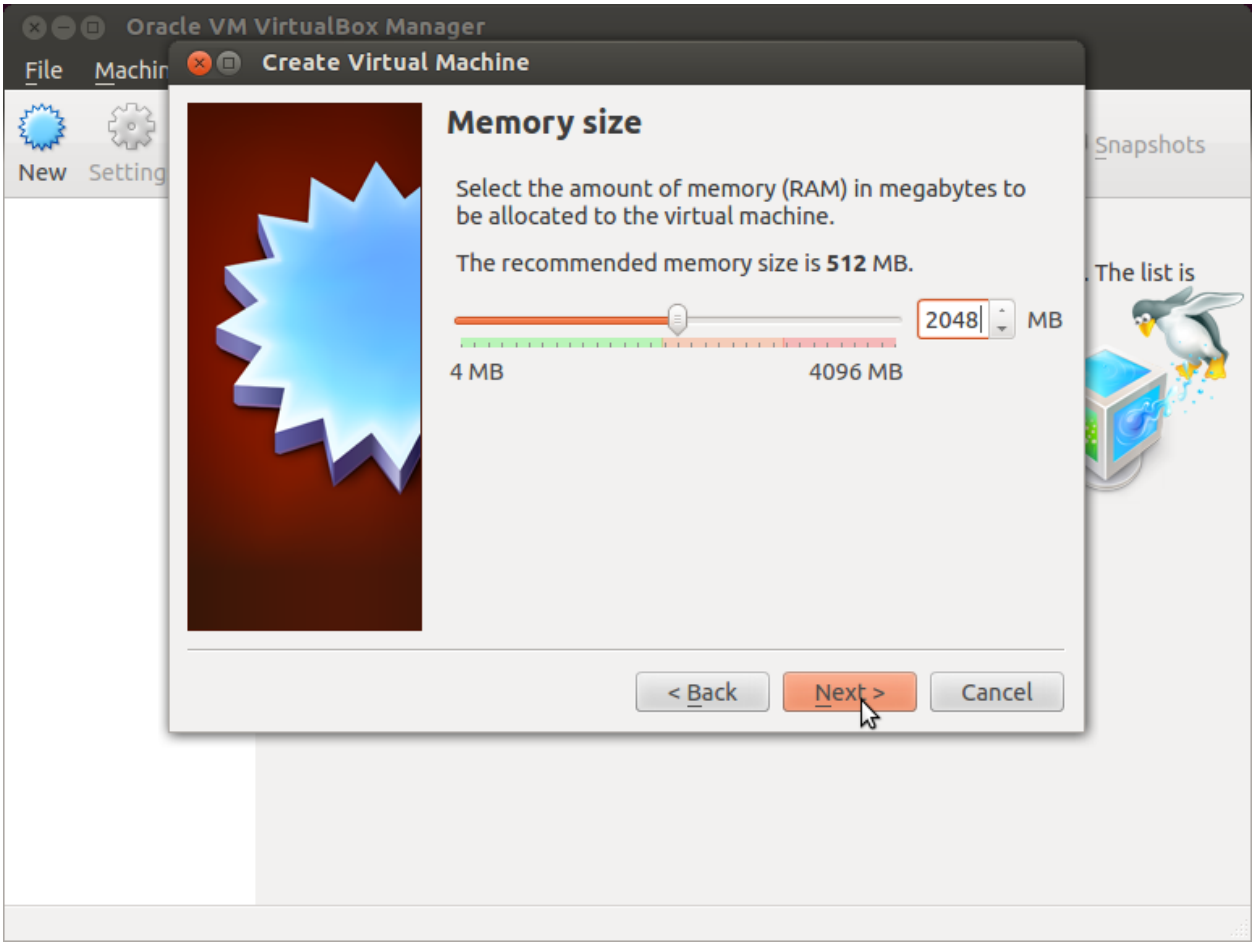
2. Click on *New* button



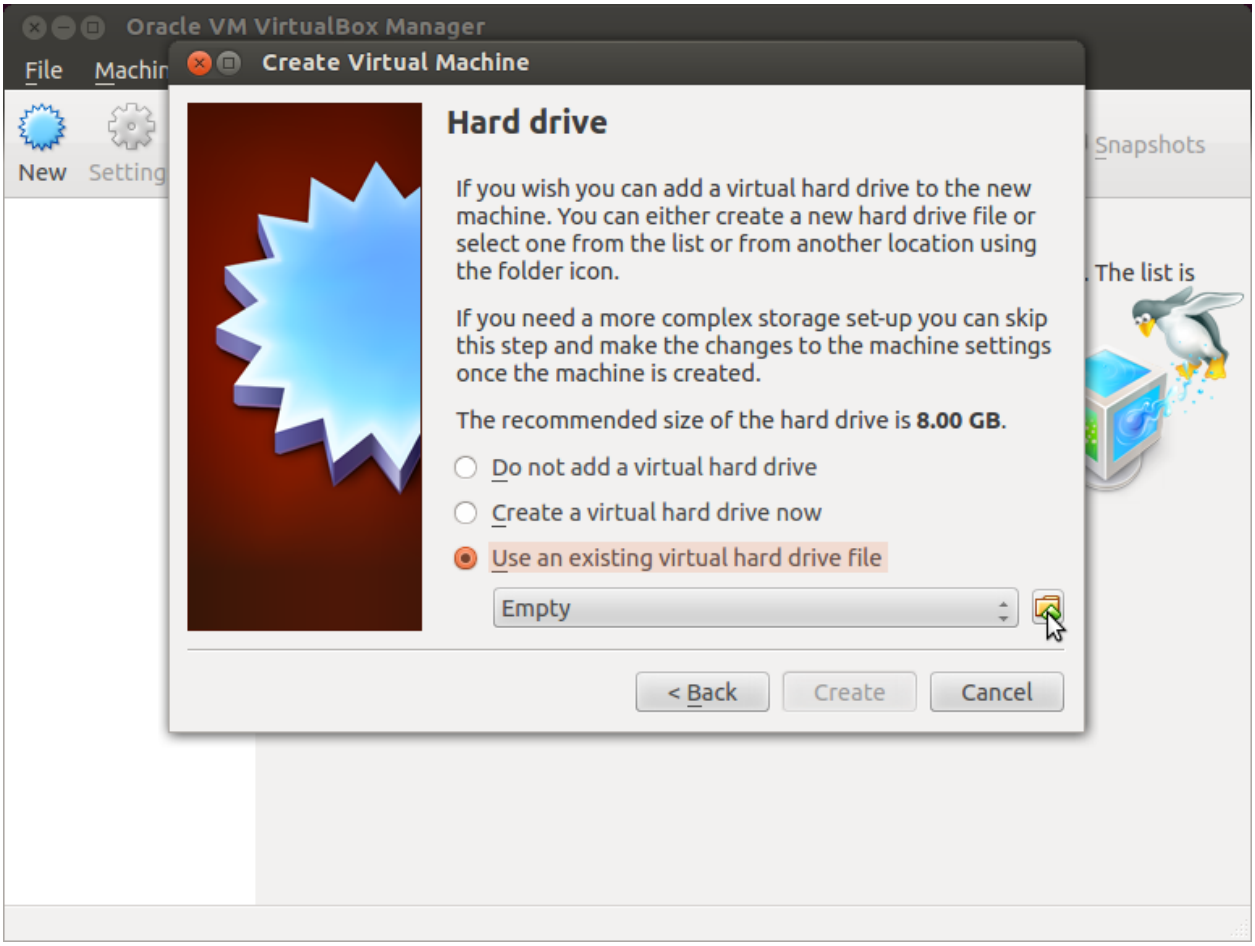
3. Select the name of the virtual machine and the operating system type



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*.

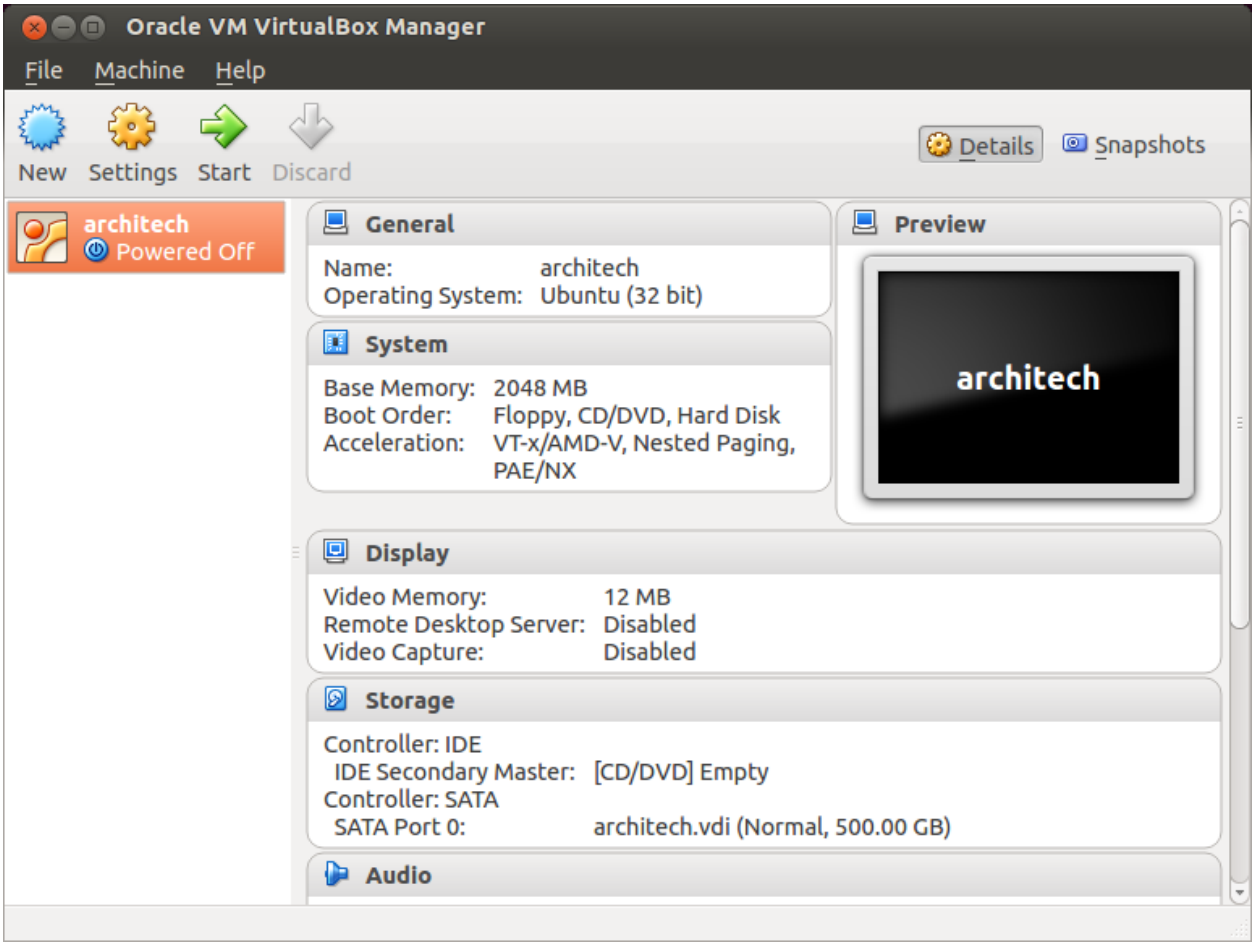


Setup the network

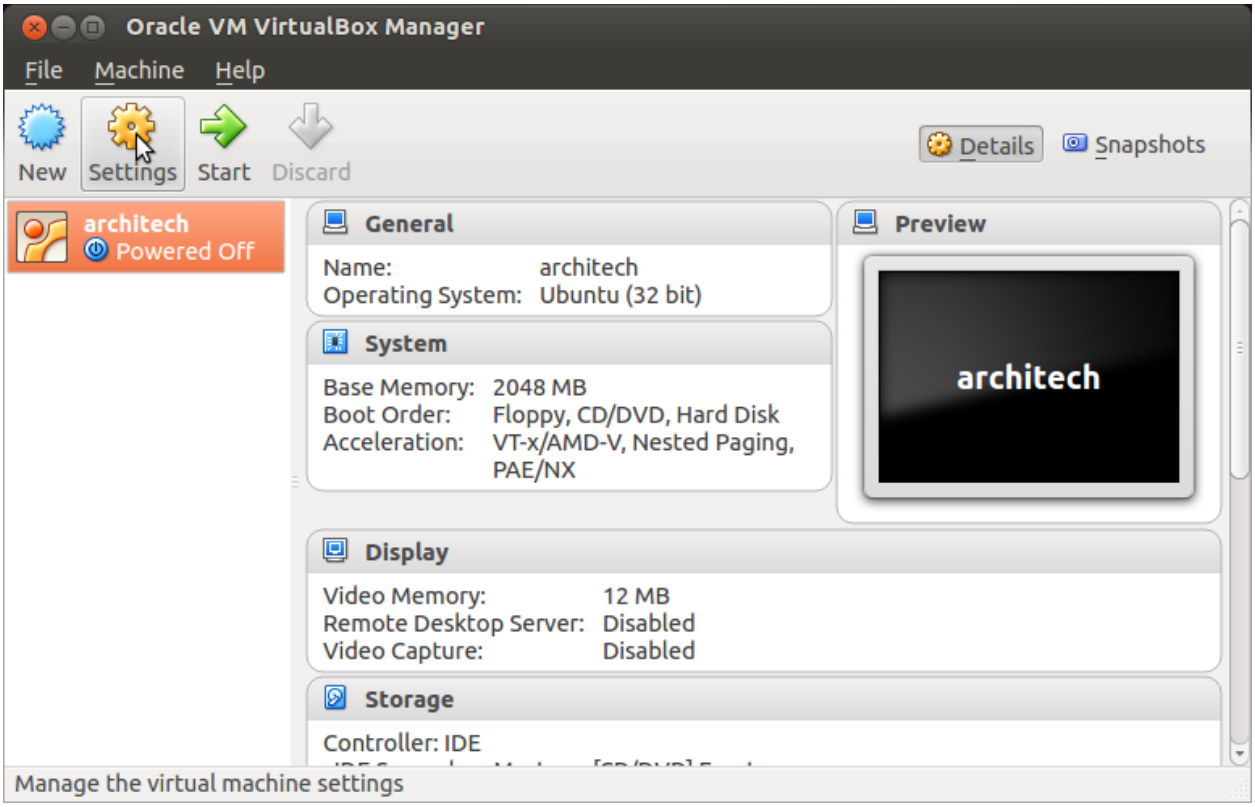
We need to setup a port forwarding rule to let you (later) use the virtual machine as a local repository of packages.

Note: The virtual machine must be off

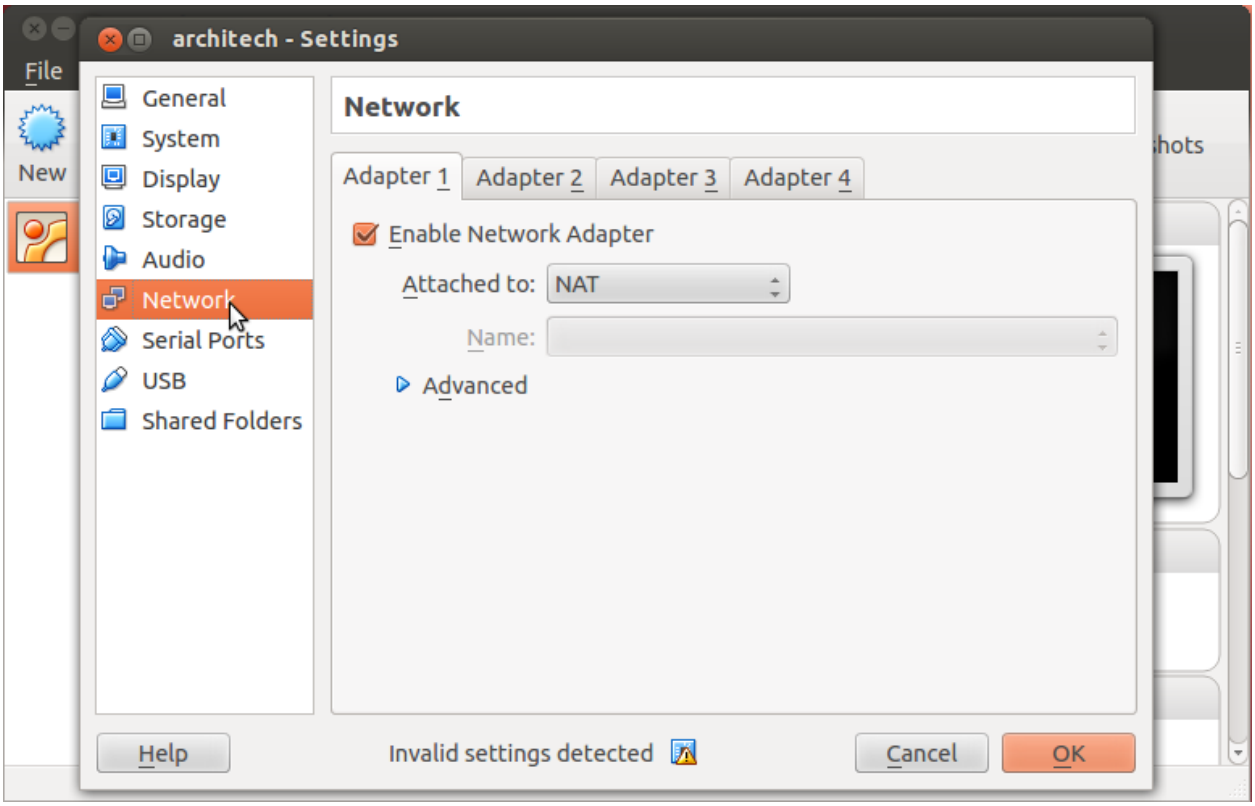
1. Select Architech's virtual machine from the list of virtual machines



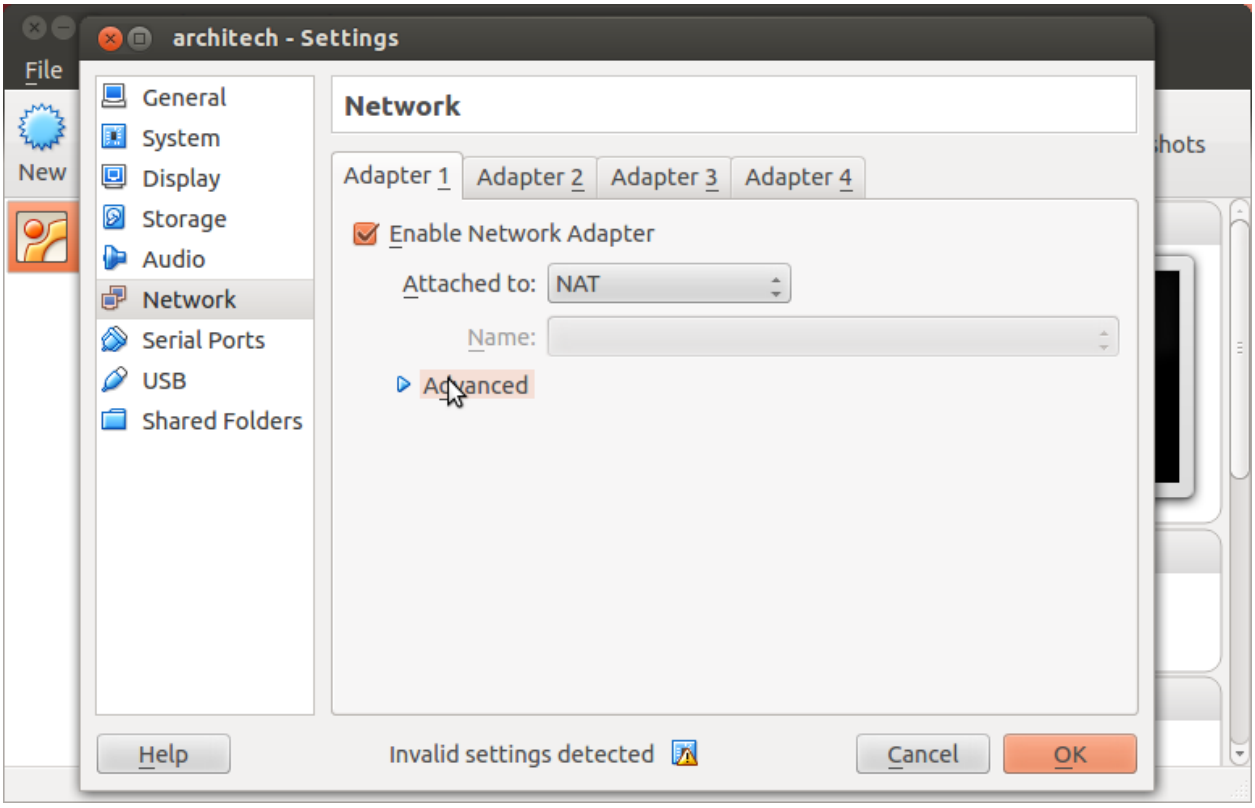
2. Click on *Settings*



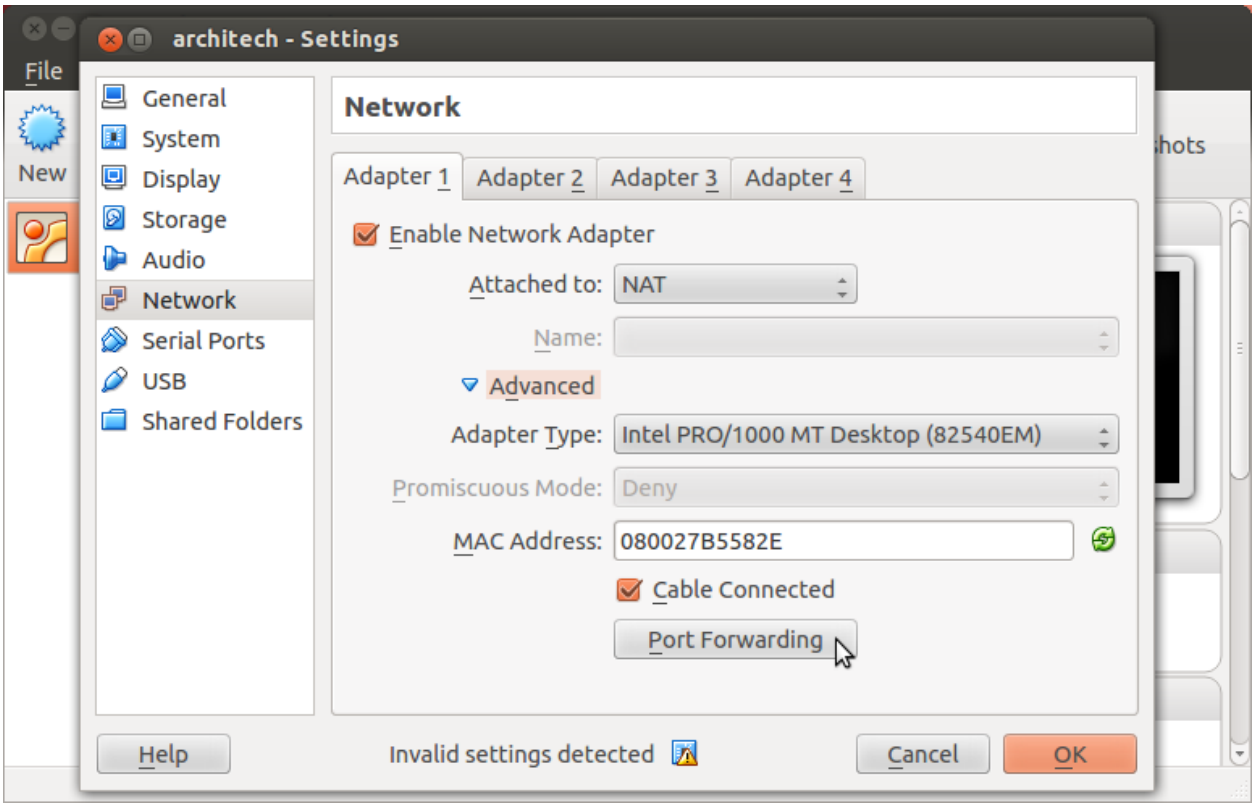
3. Select *Network*



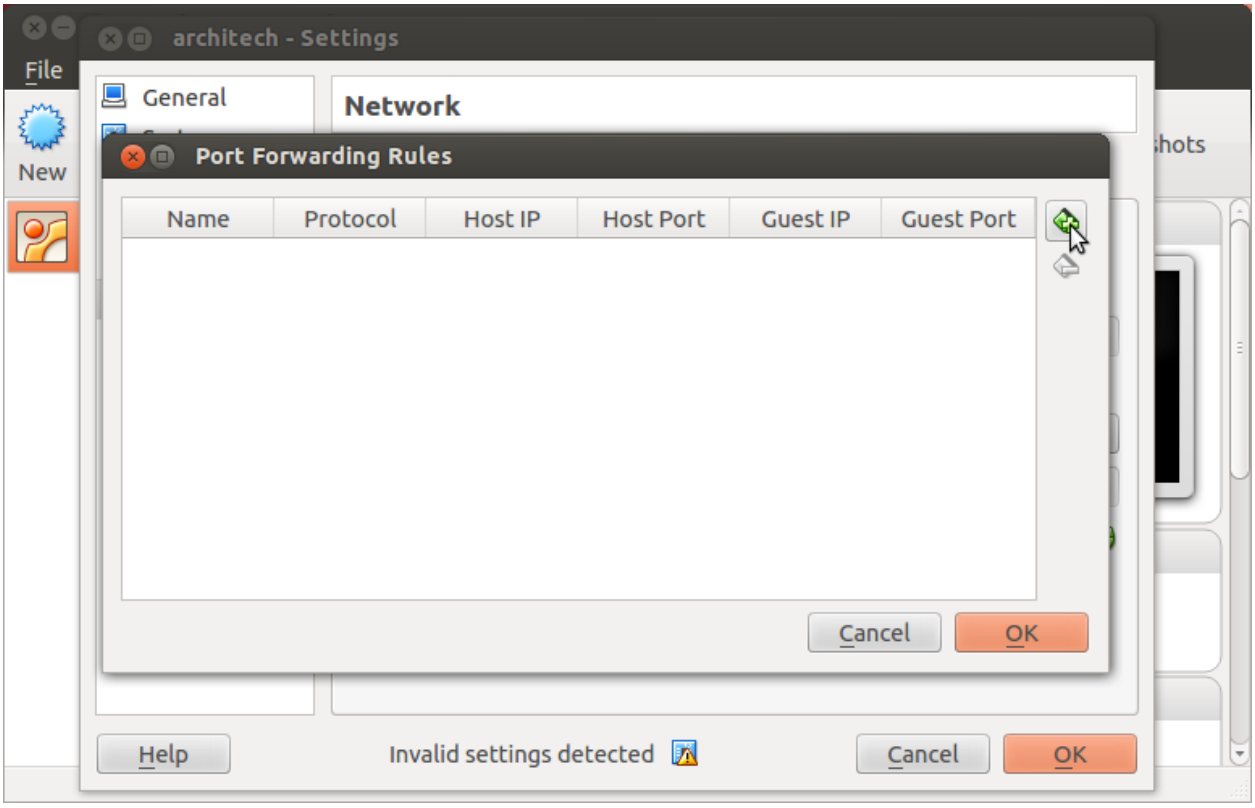
4. Expand *Advanced* of *Adapter 1*



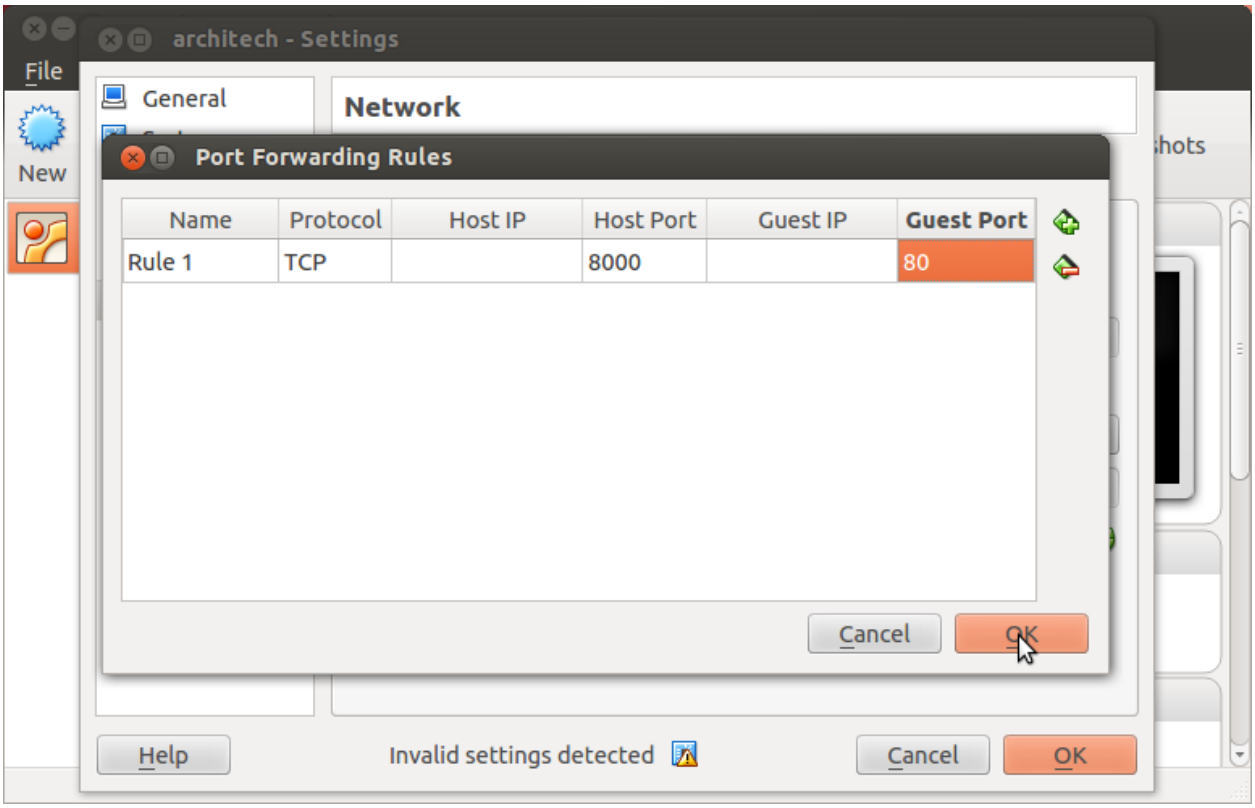
5. Click on *Port Forwarding*



6. Add a new *rule*



7. Configure the *rule*



8. Click on *Ok*

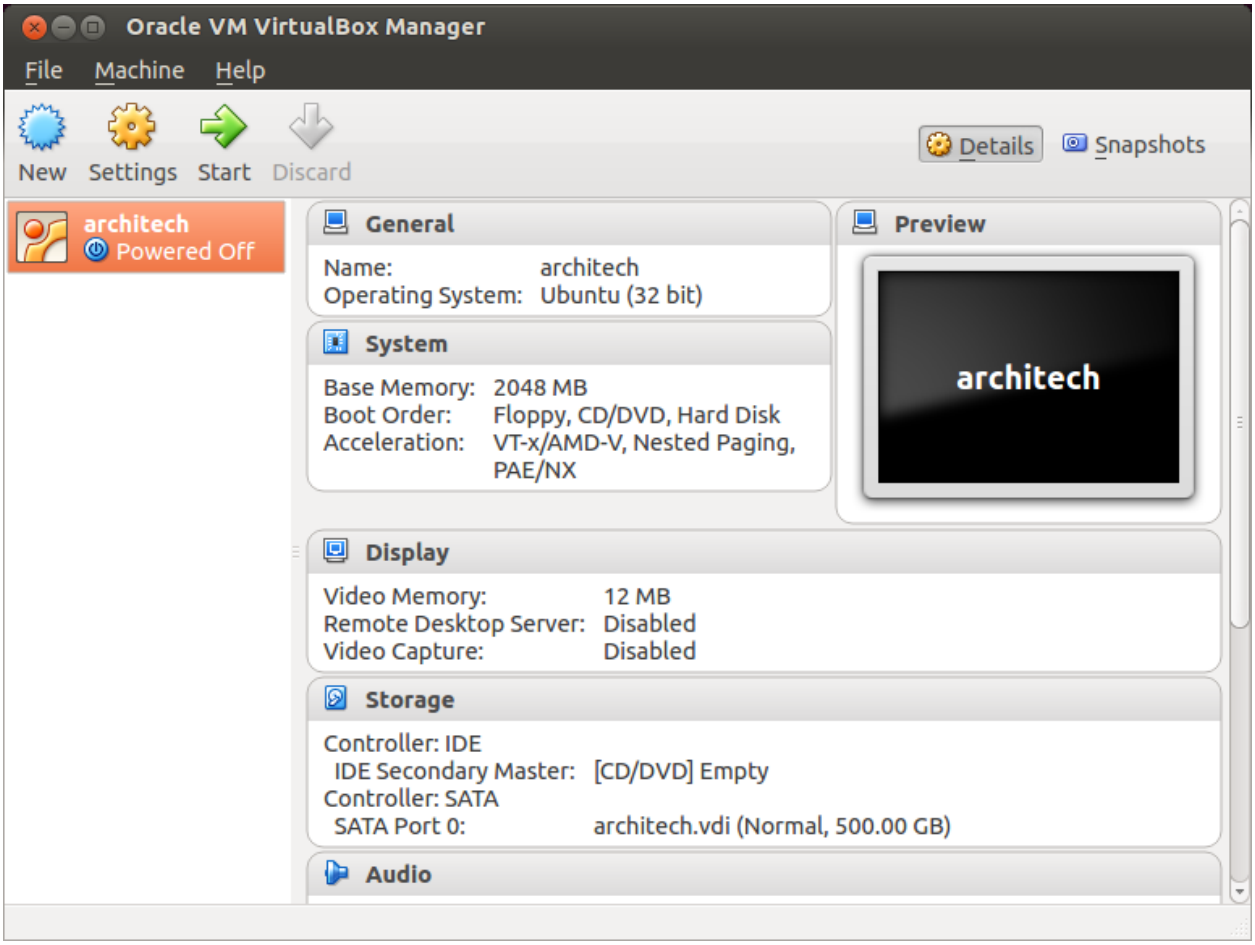
Customize the number of processors

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.

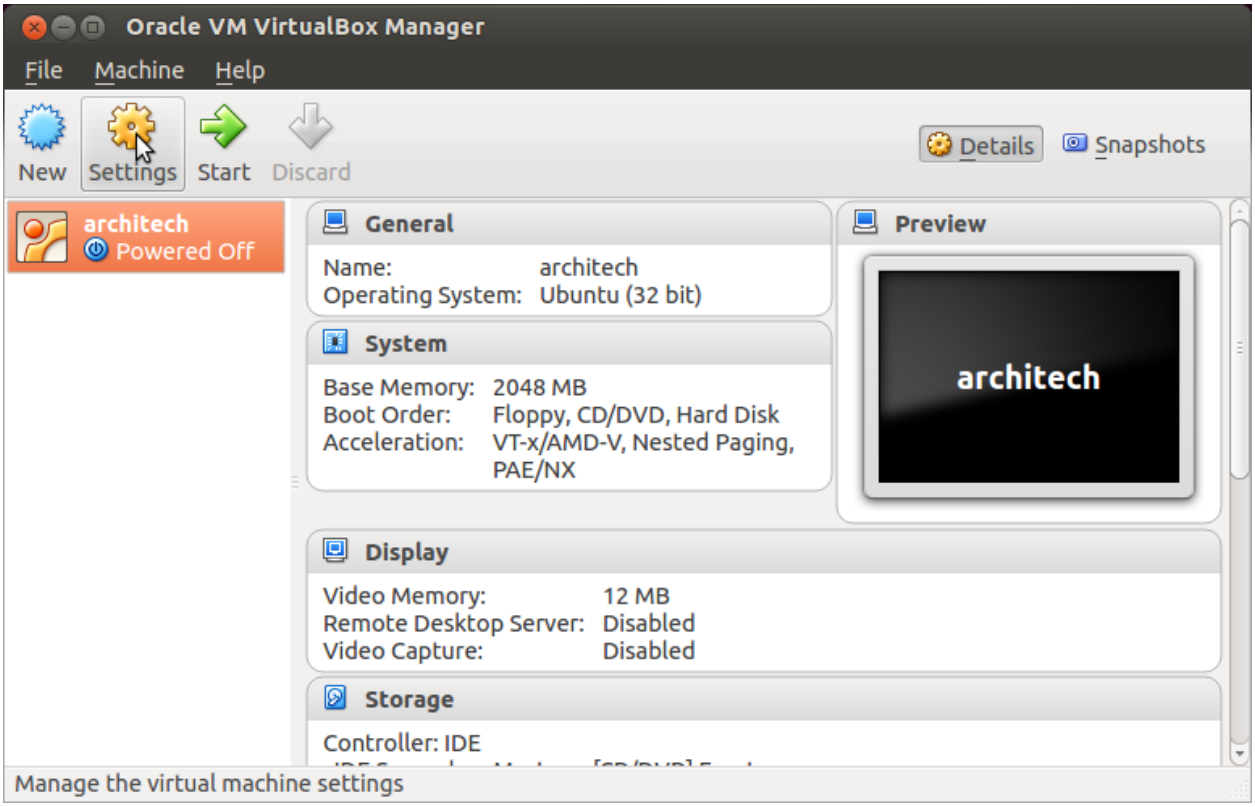
Note: The virtual machine must be off

---

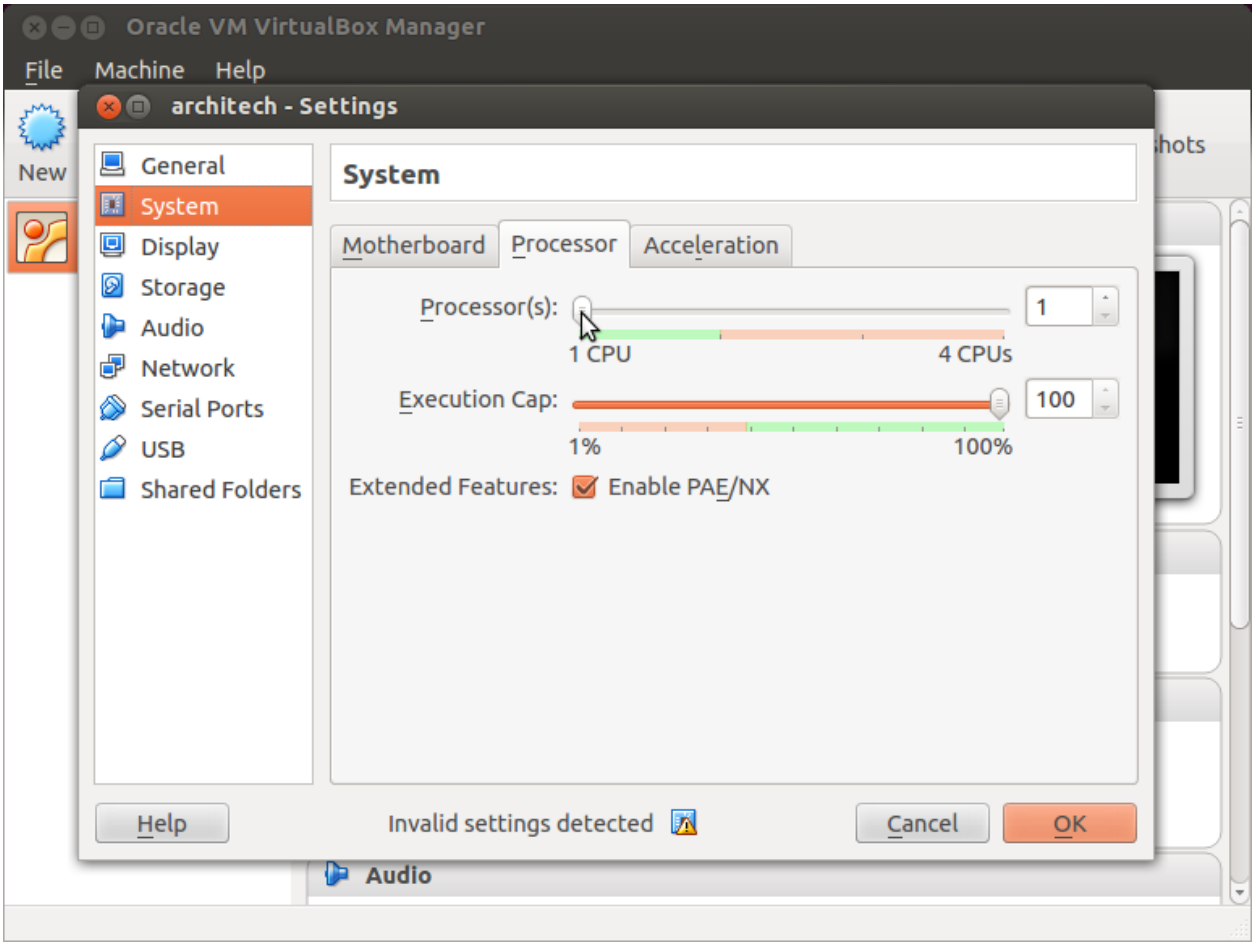
1. Select Architech's virtual machine from the list of virtual machines



2. Click on *Settings*



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



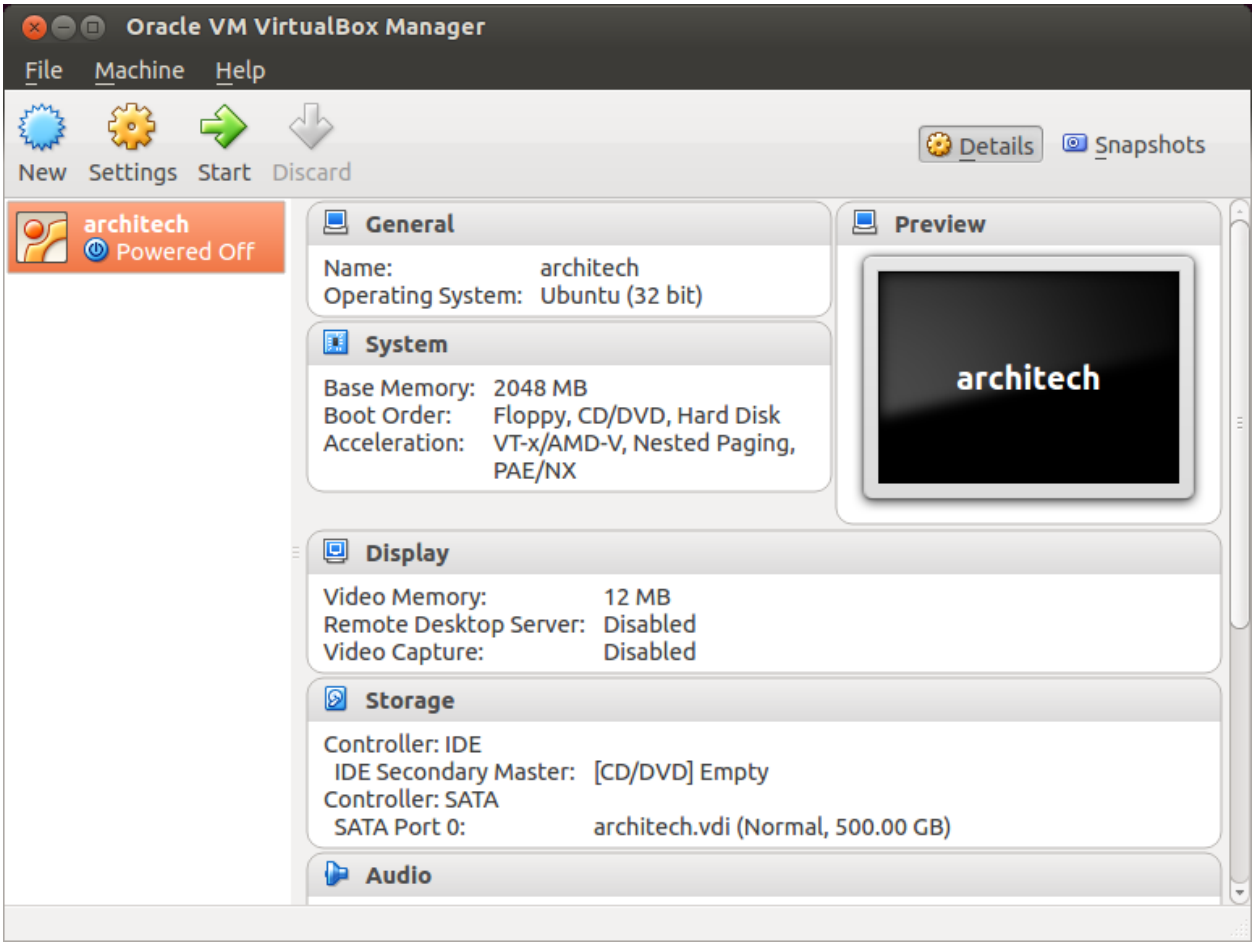
Create a shared folder

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.

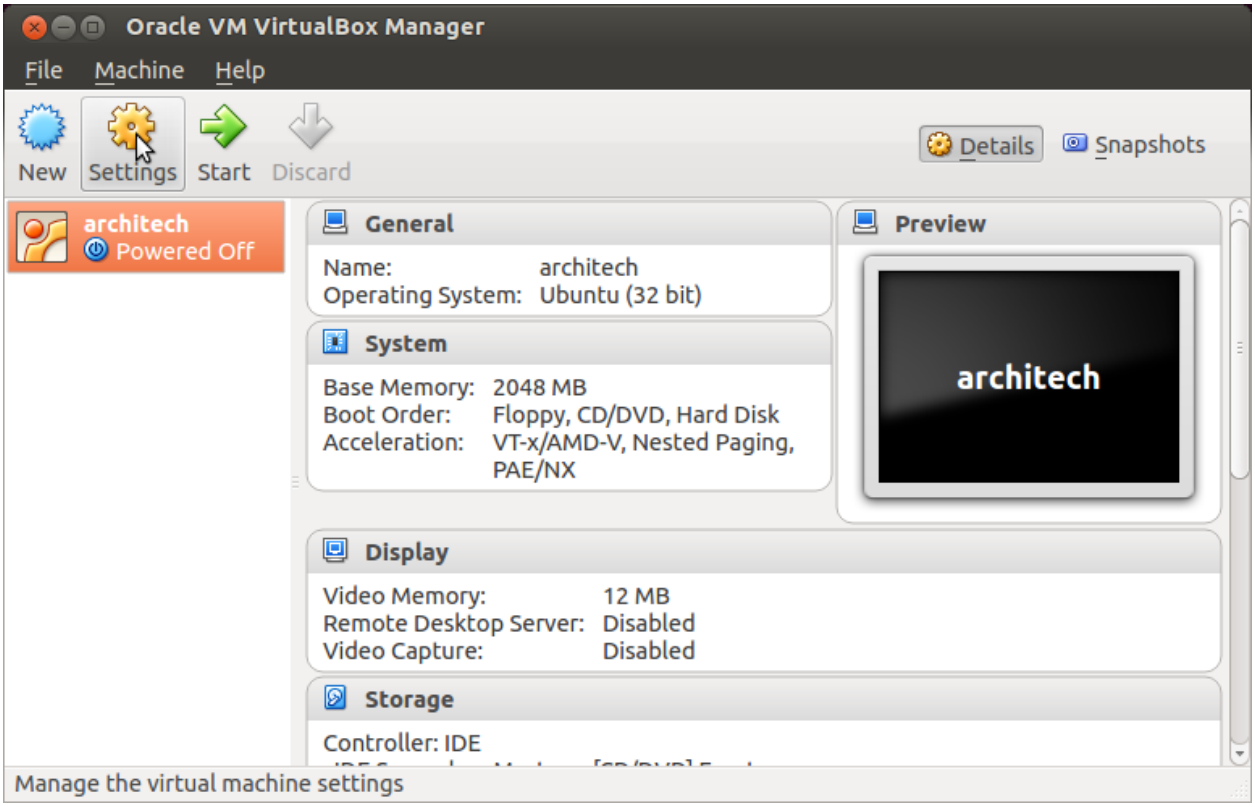
Note: The virtual machine must be off

---

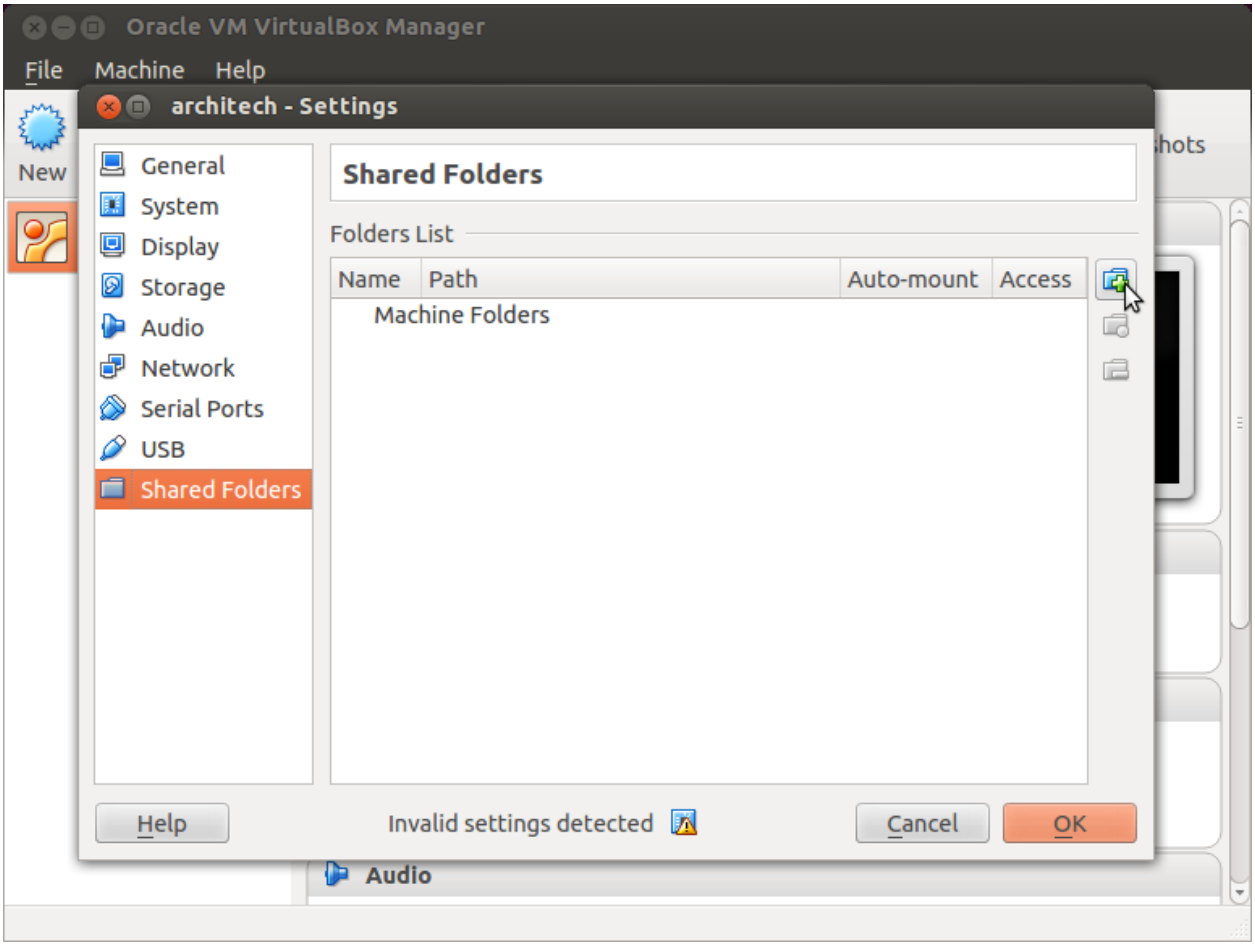
1. Select Architech's virtual machine from the list of virtual machines



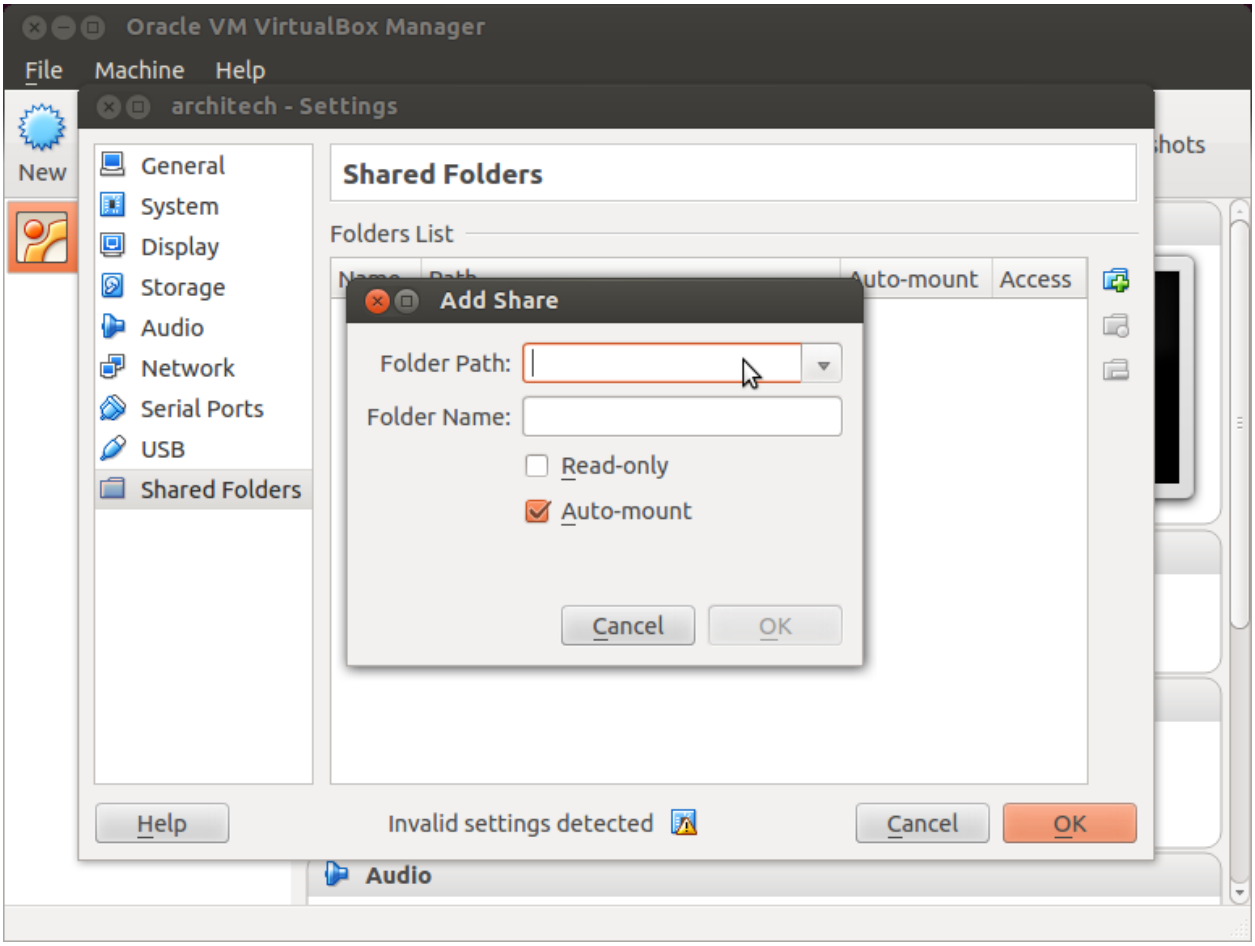
2. Click on *Settings*



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.



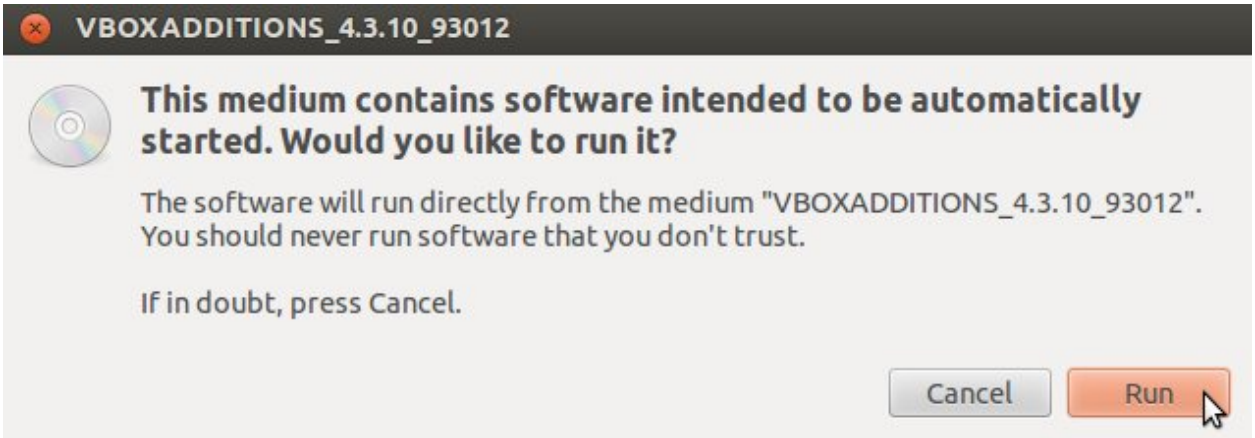
Once the virtual machine has been booted, the shared folder will be mounted under */media/* directory inside the virtual machine.

Install VBox Additions

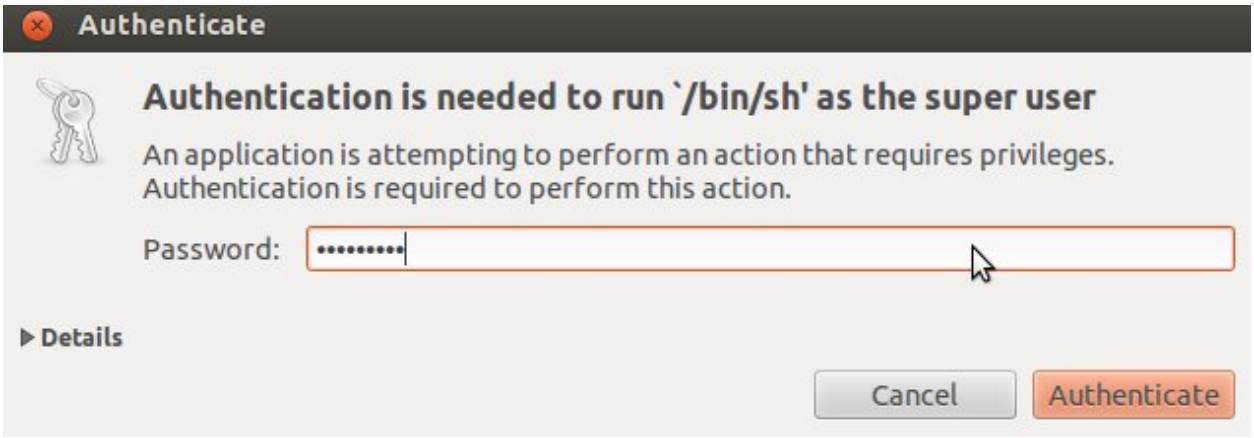
The VBox addictions add functionalities to the virtual machine such as better graphic driver and more. It is already installed in the SDK but is important re-install it to configuring correctly the virtual machine with your operating system.

1. Starts the virtual machine

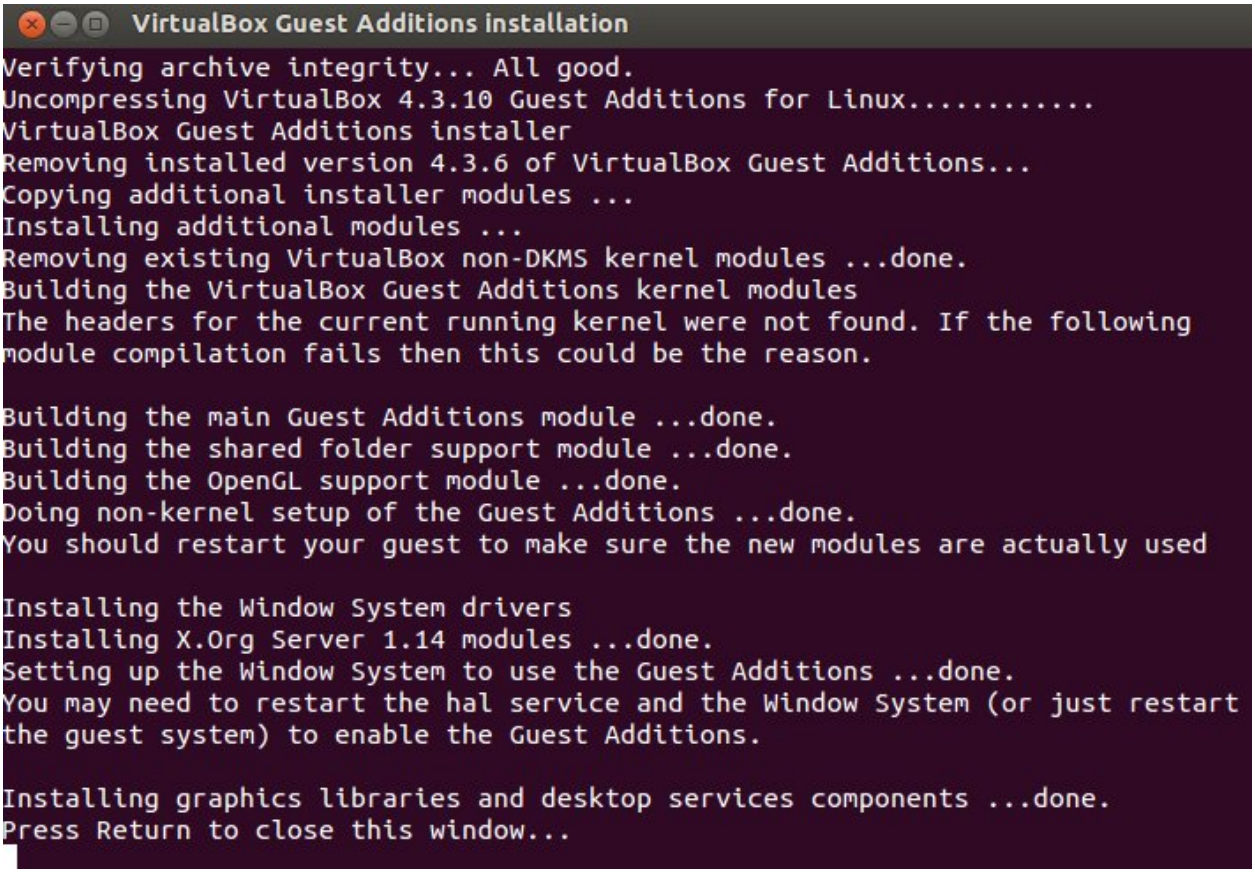
2. Click on the virtual box menu to the voice *Devices* and select *Insert Guest Additions CD Images...*. A message box will appear at the start of the installation, click on *run* button



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

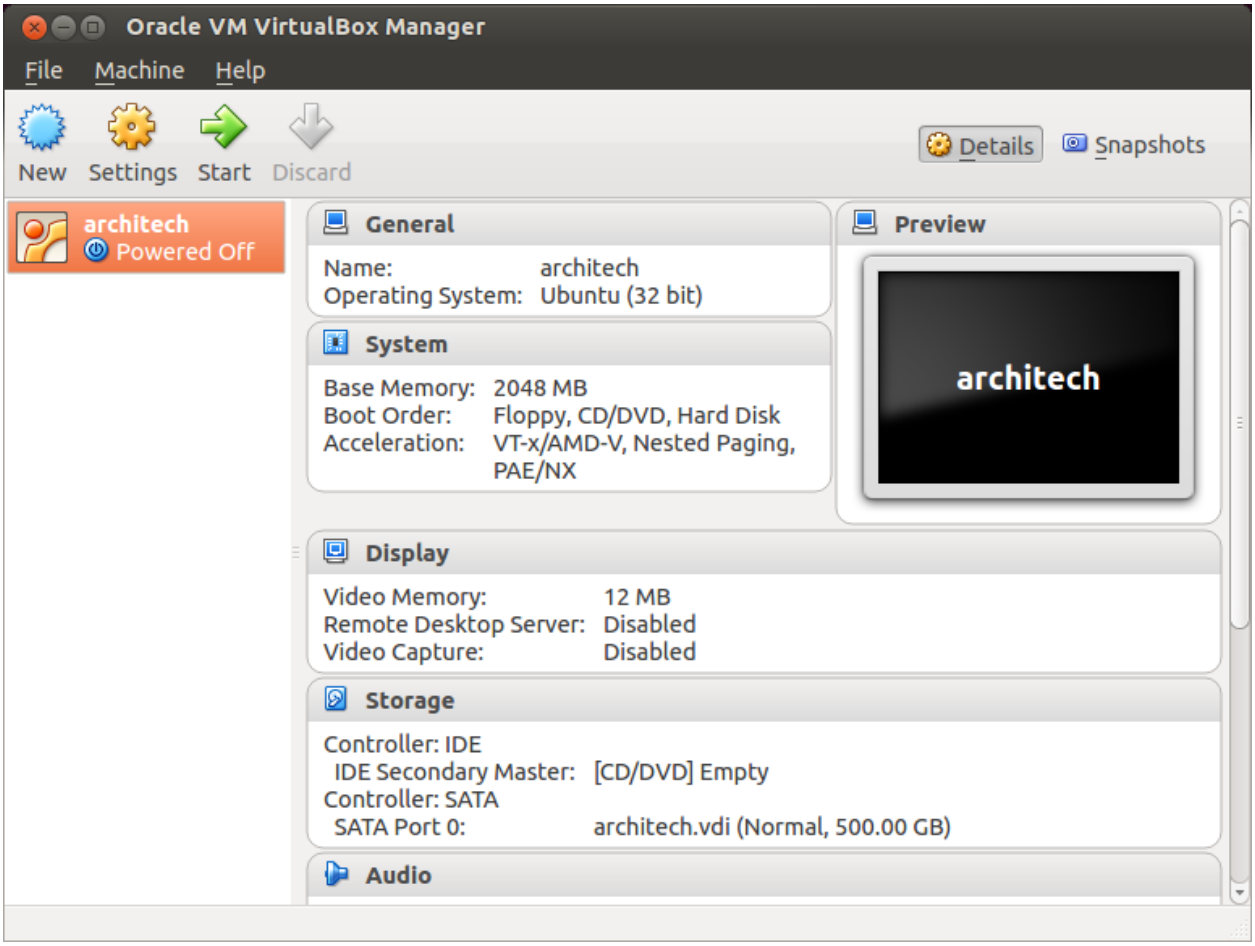


6. Before to use the SDK, it is required reboot the virtual machine

Build

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

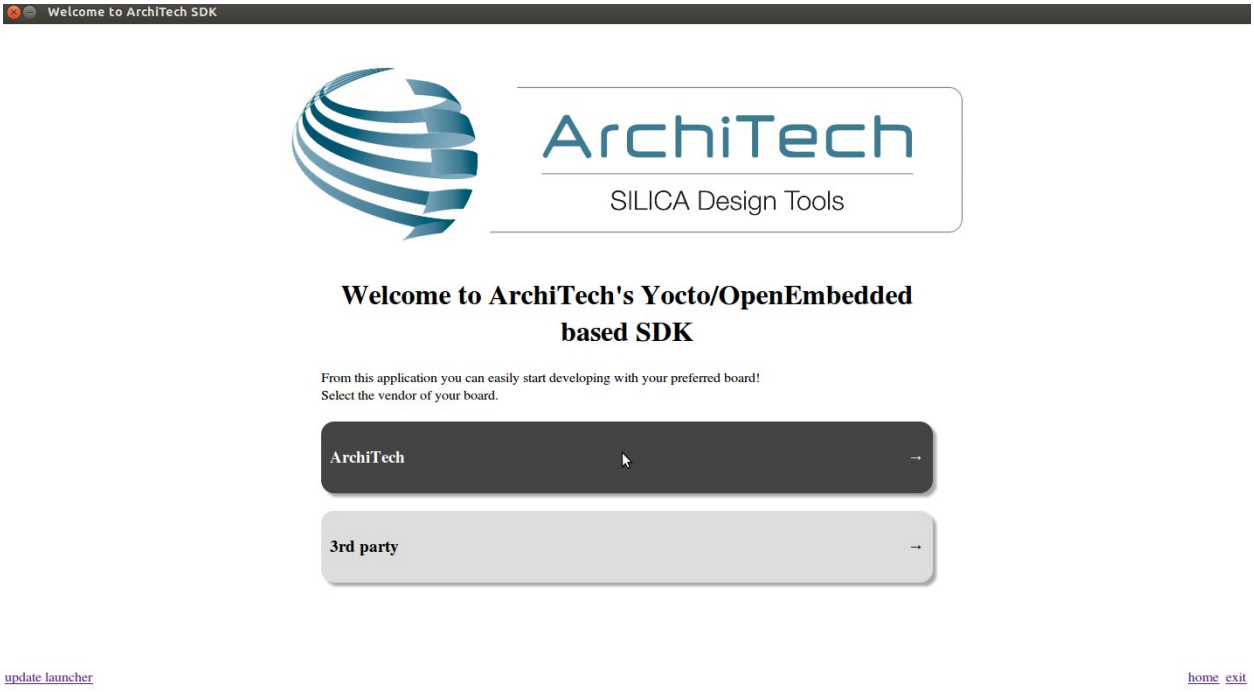
1. Select Architech's virtual machine from the list of virtual machines inside Virtual Box application



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 Hachiko 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 *tiny-image* 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**"

Deploy

To deploy the root file system, you are going to need an USB flash drive.

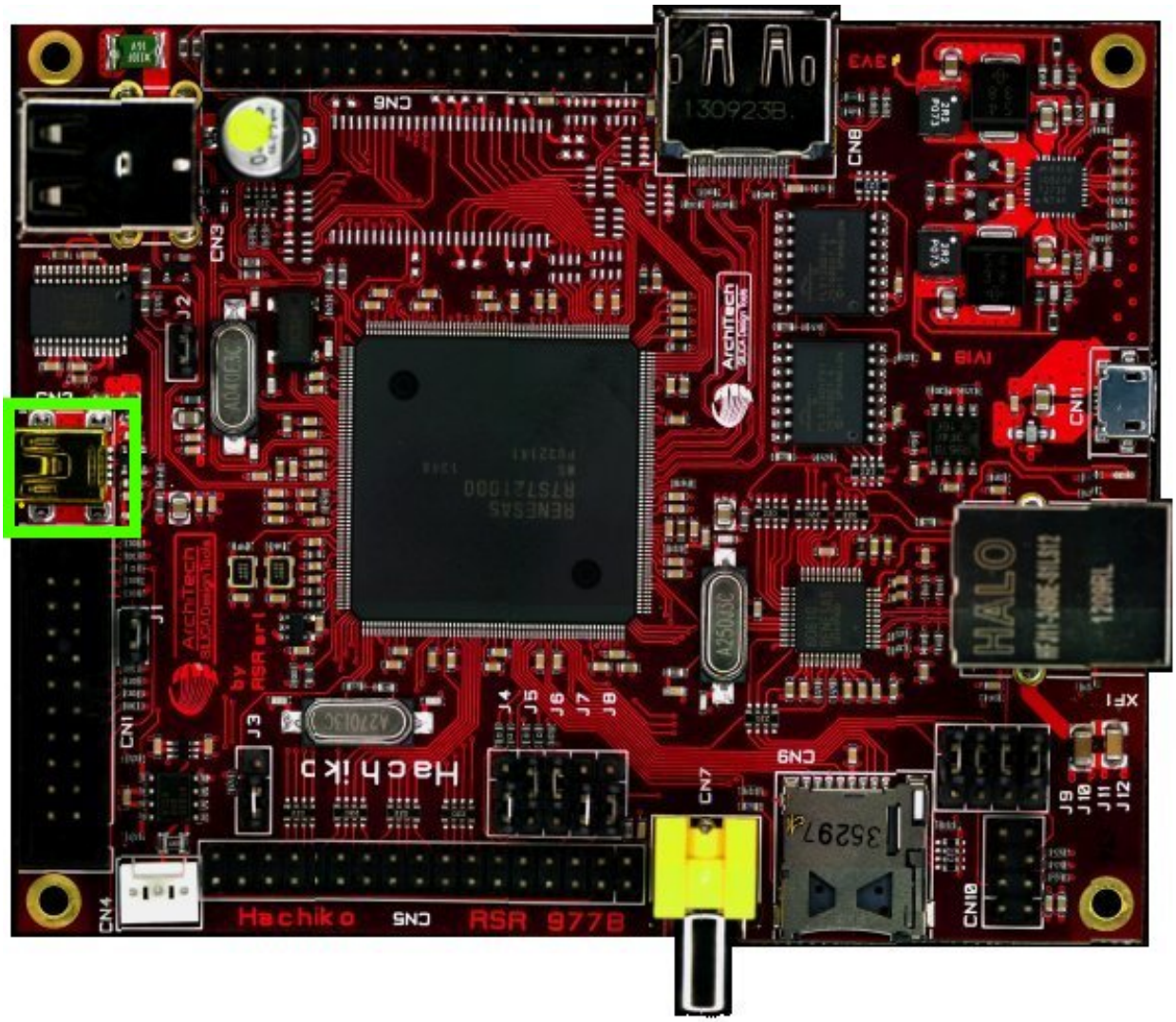
> **Warning:** The USB flash drive content will be lost forever!

1. Umount the device:
2. Align a new partition to the first sector of your USB device:
3. Format it as an *EXT2* partition:
4. Extract your image inside such a media:
5. Copy the kernel to the USB flash drive:
6. Now copy the device tree:

7. Unmount the flash drive from your system
8. Insert the USB Flash drive in the USB port at the bottom of the USB connector of Hachiko board

Boot

On Hachiko there is the dedicated serial console connector **CN2**



which you can connect, by means of a mini-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 ttyUSBX 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

Host

sudo dmesg -c

2. connect the mini-USB cable to the board
3. display the kernel messages

Host

dmesg

3. read the output

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

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

Host

sudo minicom -ws

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

Host

sudo apt-get install minicom

then you can setup your port with these parameters:

```
+-----------------------------------------------------------------------+
| A -    Serial Device      : /dev/ttyUSB0                              |
| B - Lockfile Location     : /var/lock                                 |
| C -   Callin Program      :                                           |
| D -  Callout Program      :                                           |
| E -    Bps/Par/Bits       : 115200 8N1                                |
| F - Hardware Flow Control : No                                        |
| G - Software Flow Control : No                                        |
|                                                                       |
|    Change which setting?                                              |
+-----------------------------------------------------------------------+
        | Screen and keyboard      |
        | Save setup as dfl        |
        | Save setup as..          |
        | Exit                     |
        | Exit from Minicom        |
        +--------------------------+
```

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

Give *root* to the login prompt:

Board

```
hachiko login: root
```

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.

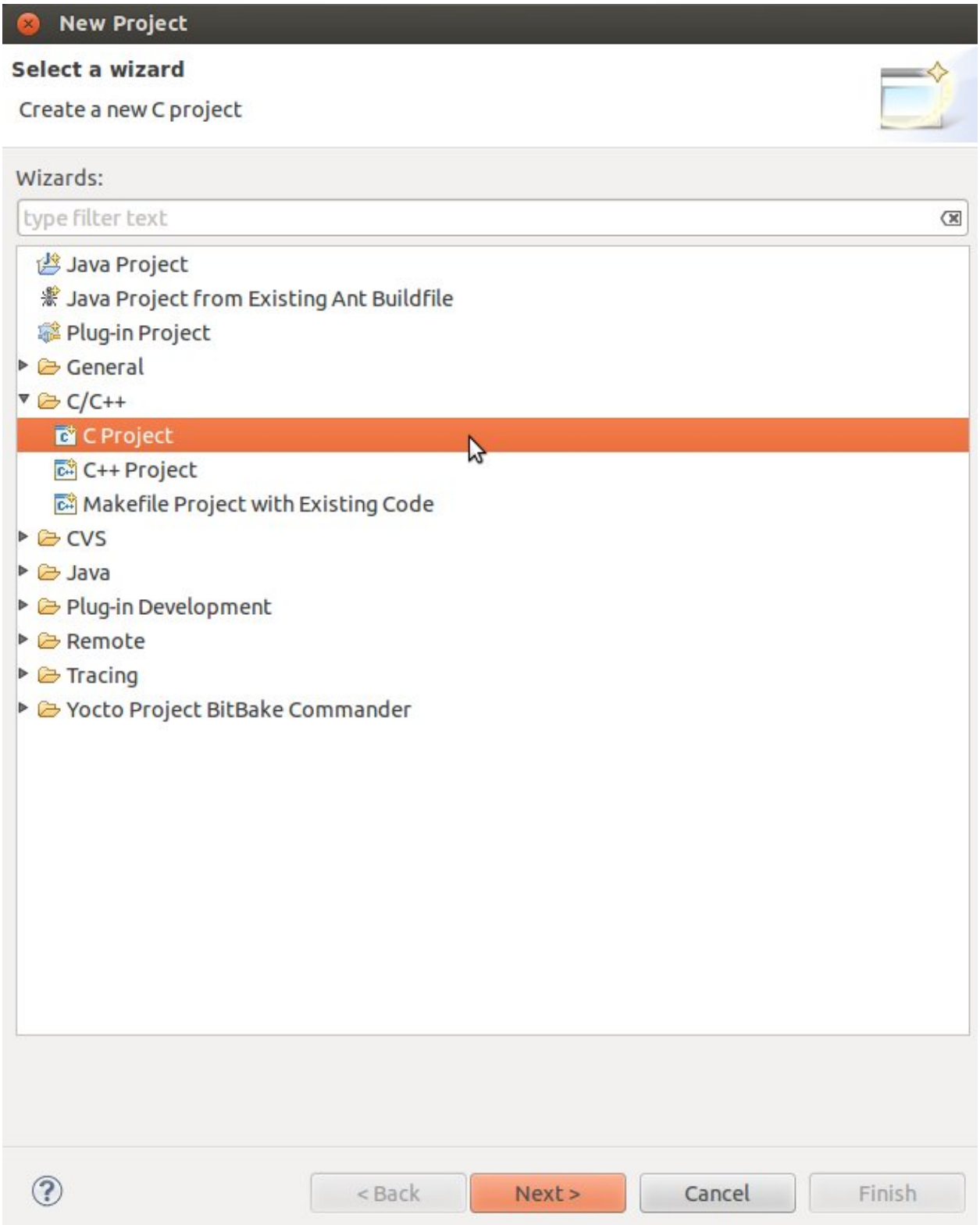
CN2 is used also to provide the power to the board, so, as soon as your serial terminal emulator is ready you start getting messages from the board.

Code

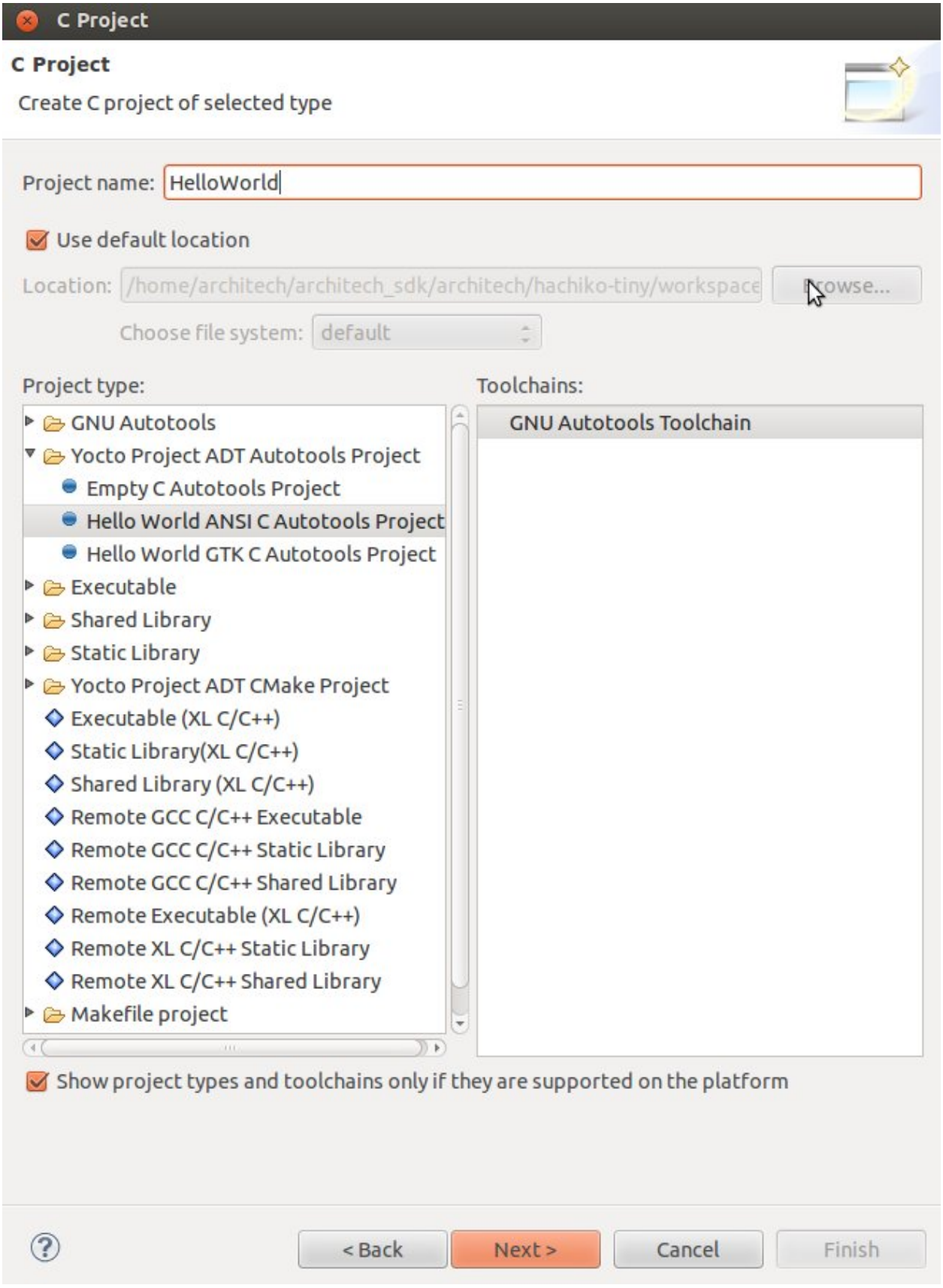
The time to create a simple *HelloWorld!* application using **Eclipse** has come.

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

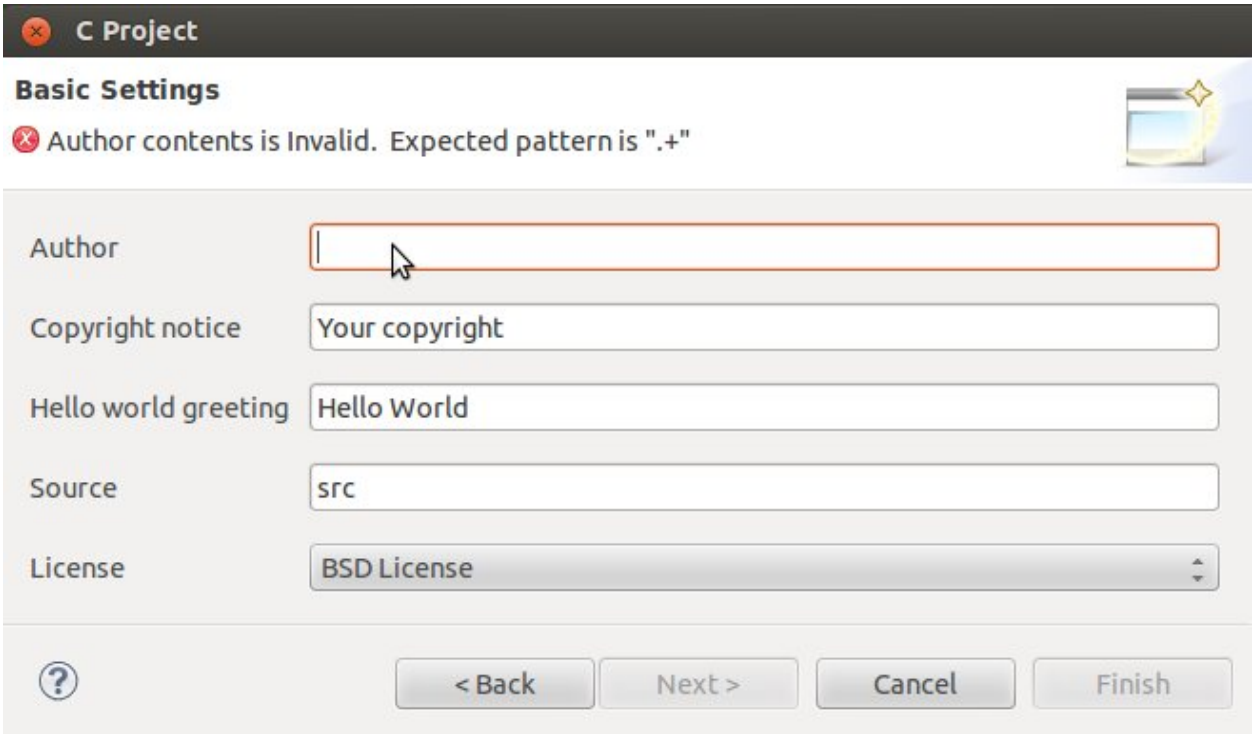
2. Go to *File→ New→ Project...*, in the node "C/C++" select *C Project* and press *next* button.



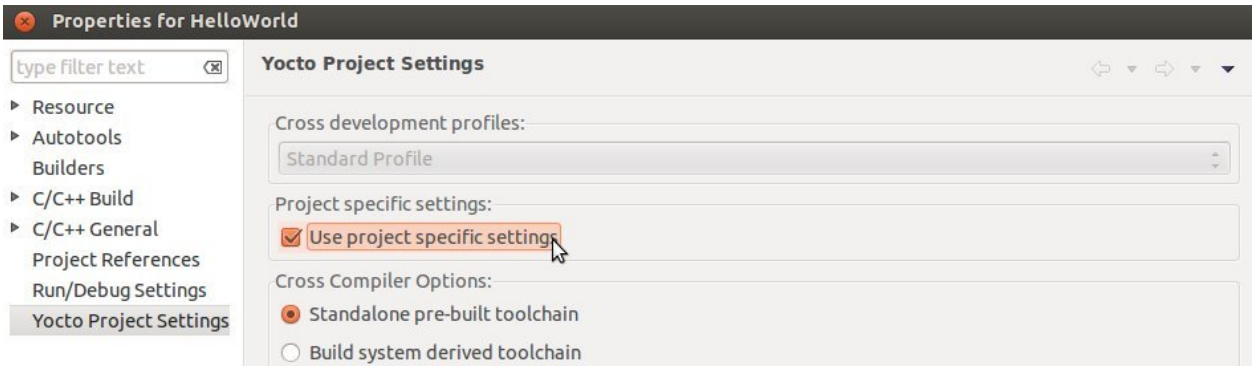
3. Insert *HelloWorld* as project name, open the node *Yocto Project ADT Autotools Project* and select *Hello World ANSI C Autotools Project* and press *next* button.


C Project
C Project
Create C project of selected type
Project name: HelloWorld
Use default location
Location: /home/architech/architech_sdk/architech/hachiko-tiny/workspace
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*.

Debug

Use an ethernet cable to connect the board (connector XF1) to your PC. Configure your workstation ip address as 192.168.0.100. Make sure the board can be seen by your host machine:

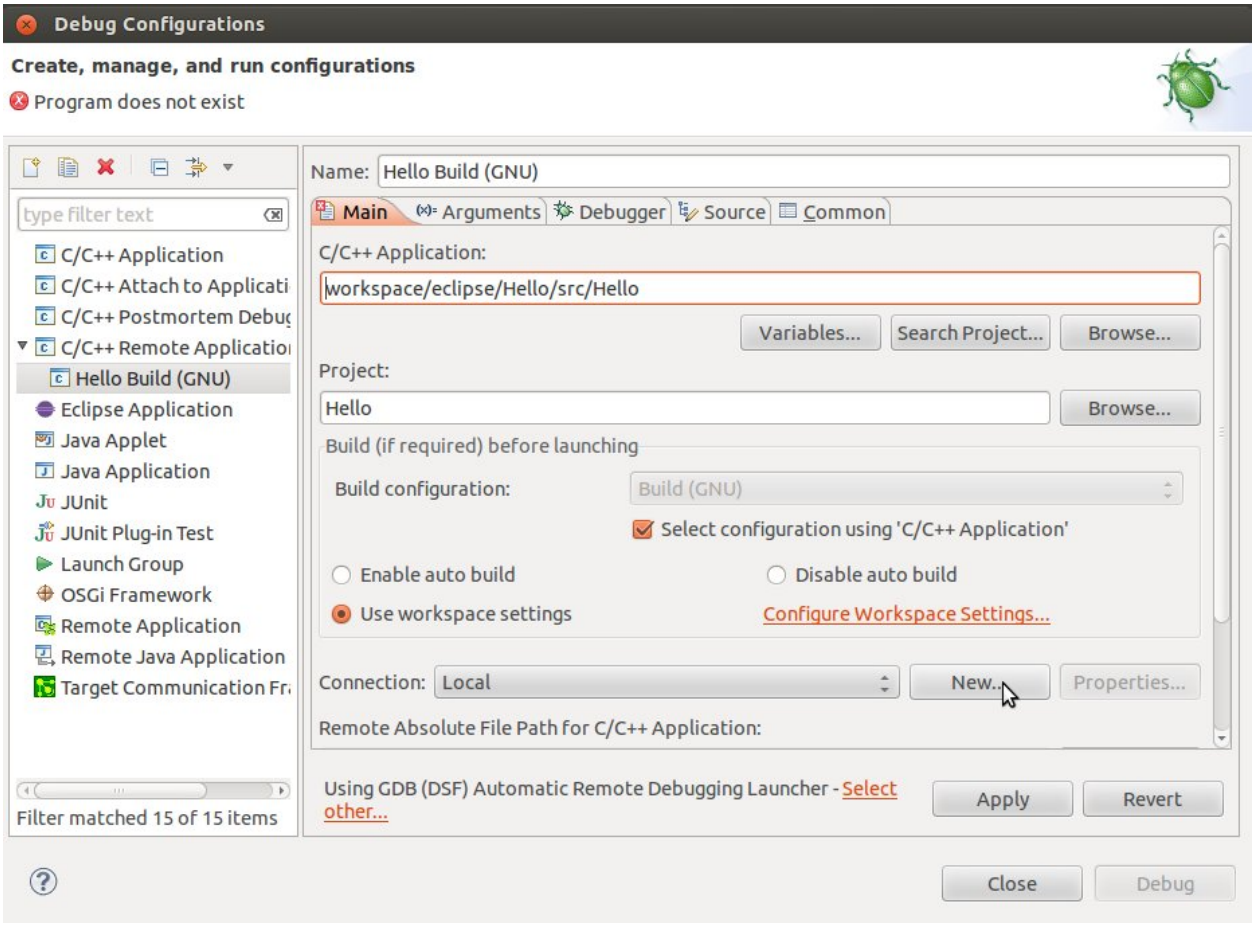
If the output is similar to this one:

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

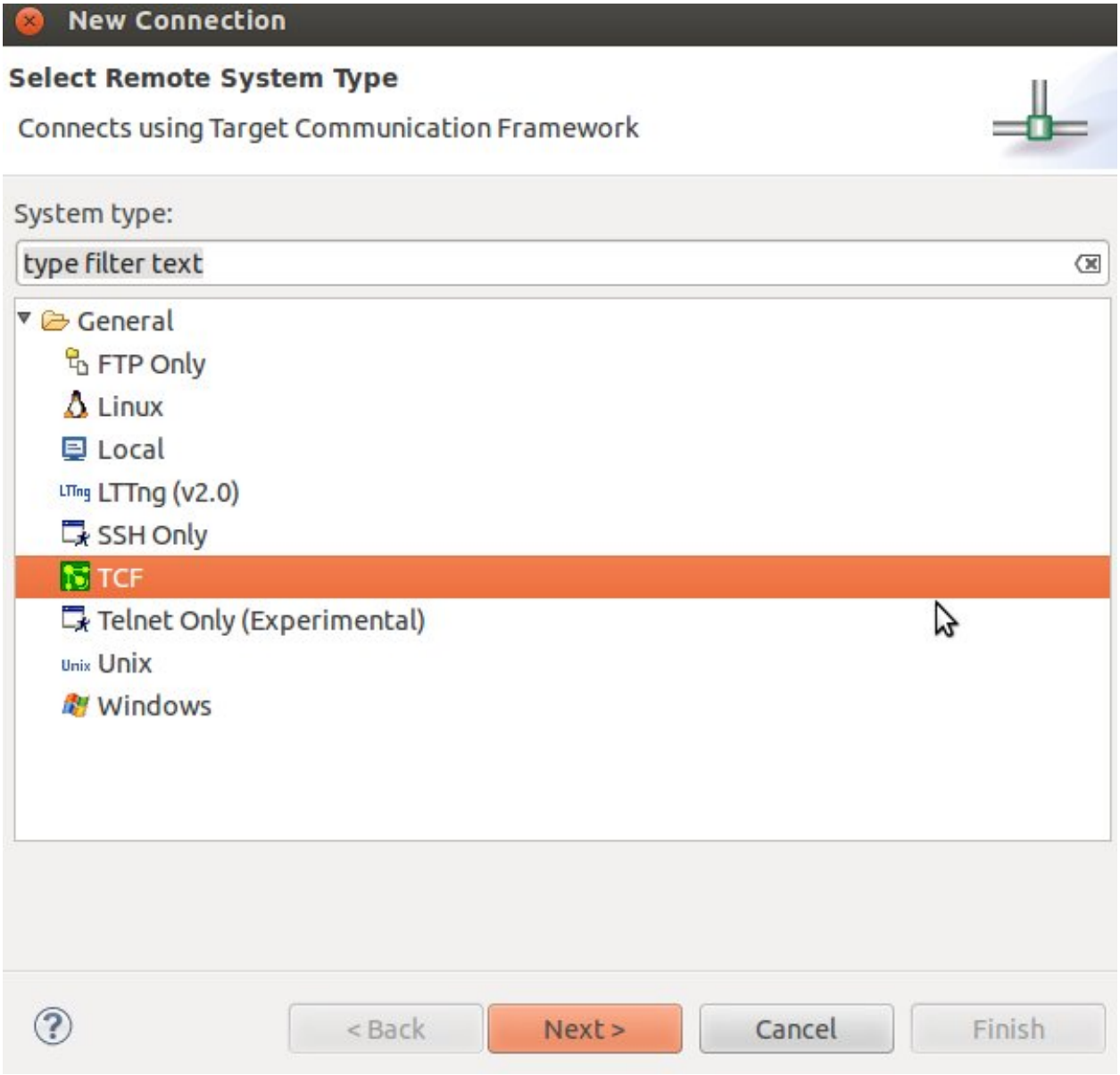
On the Host machine, follow these steps to let **Eclipse** deploy and debug your application:

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

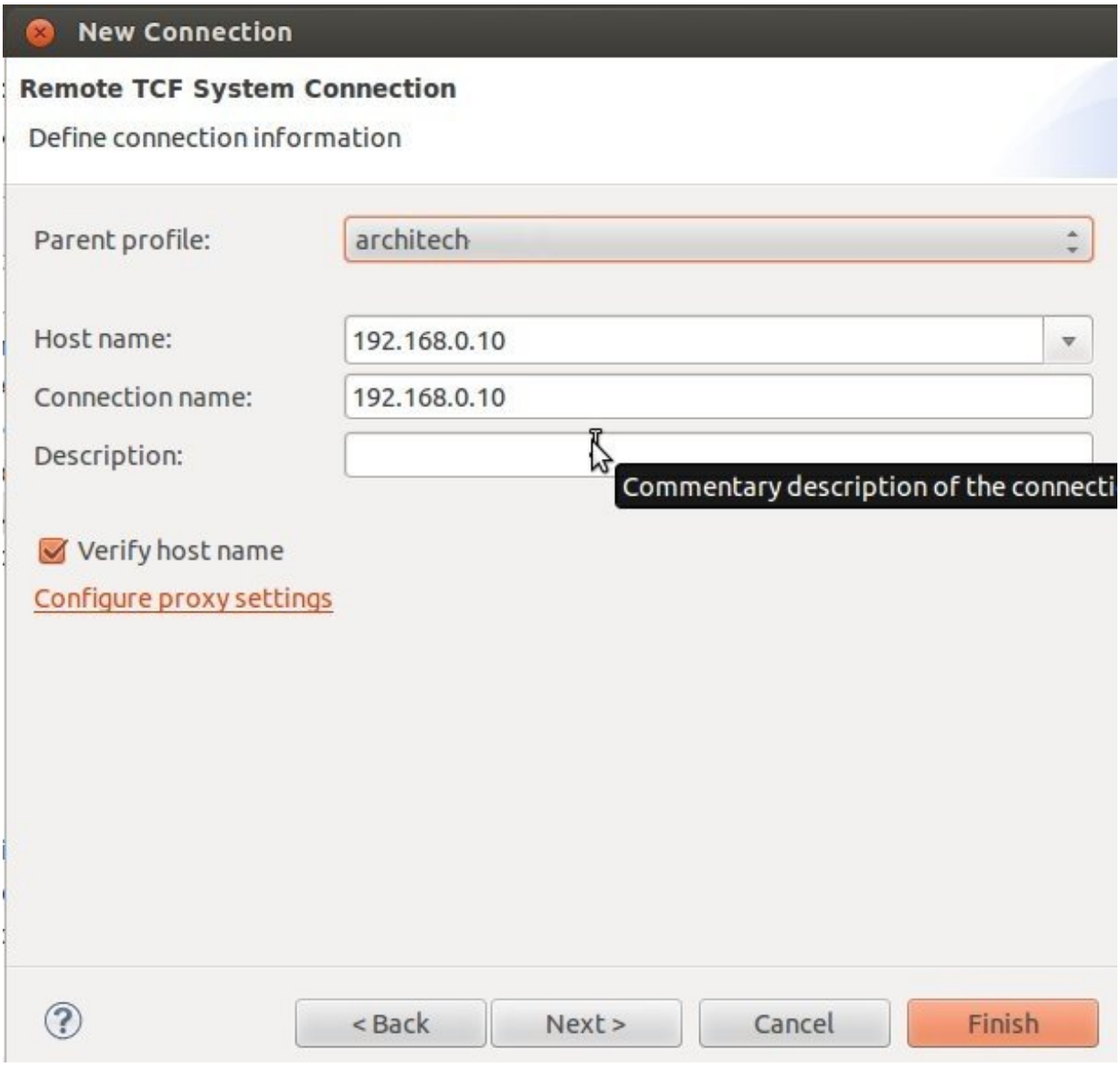
- Locate your project and select it to bring up a new tabbed view in the *Debug Configurations* Dialog.



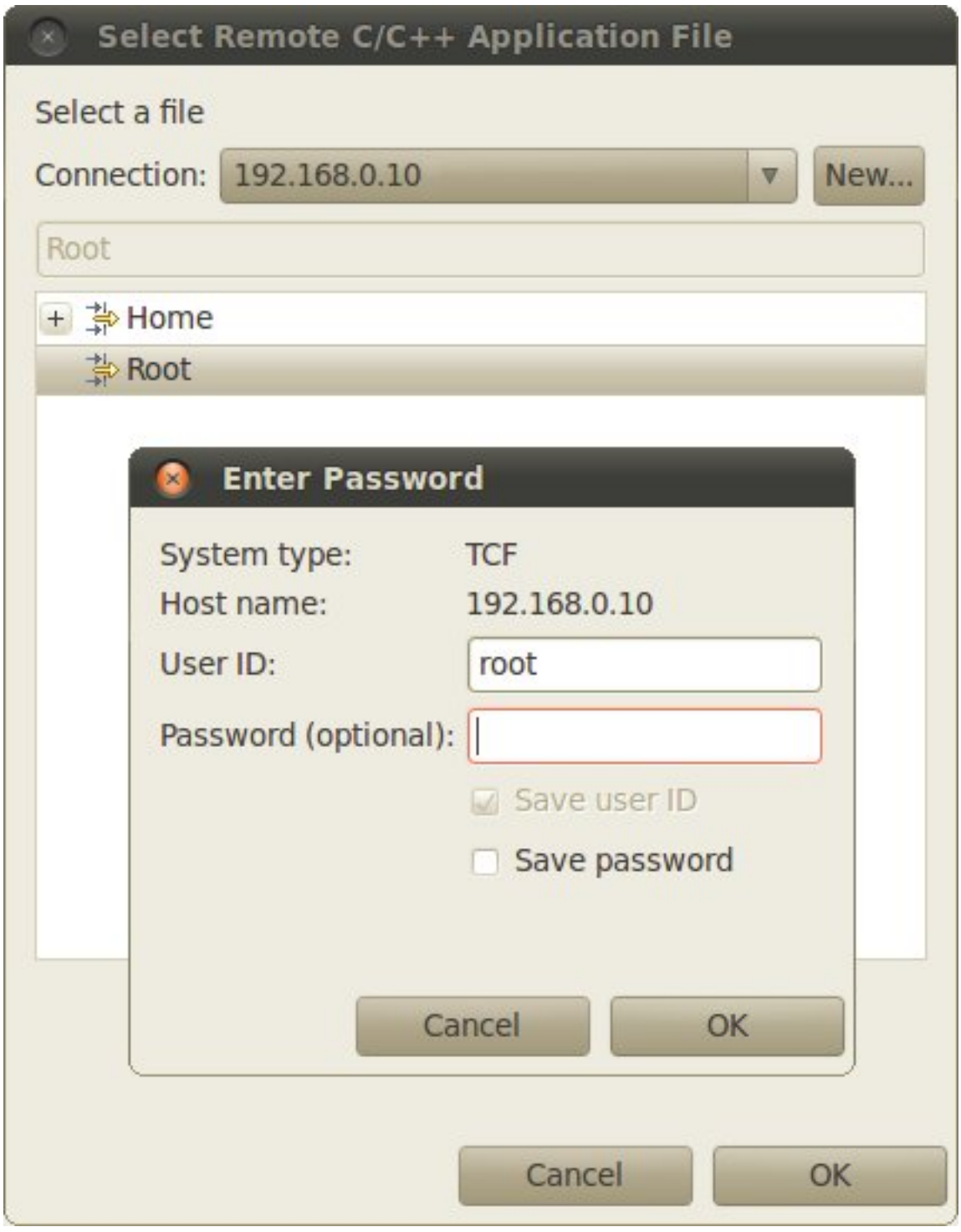
- Insert in *C/C++ Application* the filepath (on your host machine) of the compiled binary.
- Click on *New* button near the drop-down menu in the *Connection* field.
- Select *TCF* icon.



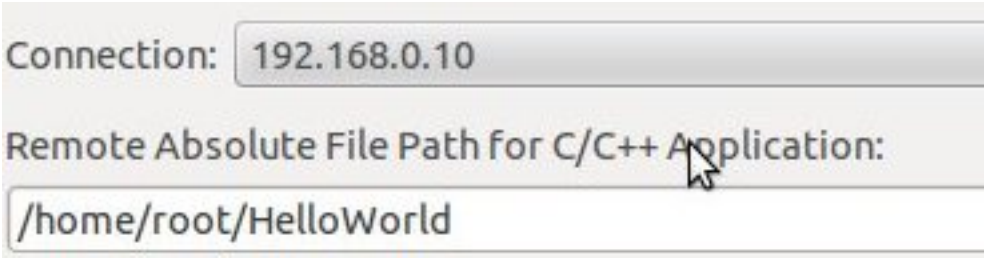
- Insert in *Host Name* and *Connection Name* fields the IP address of the target board. (e.g. 192.168.0.10)



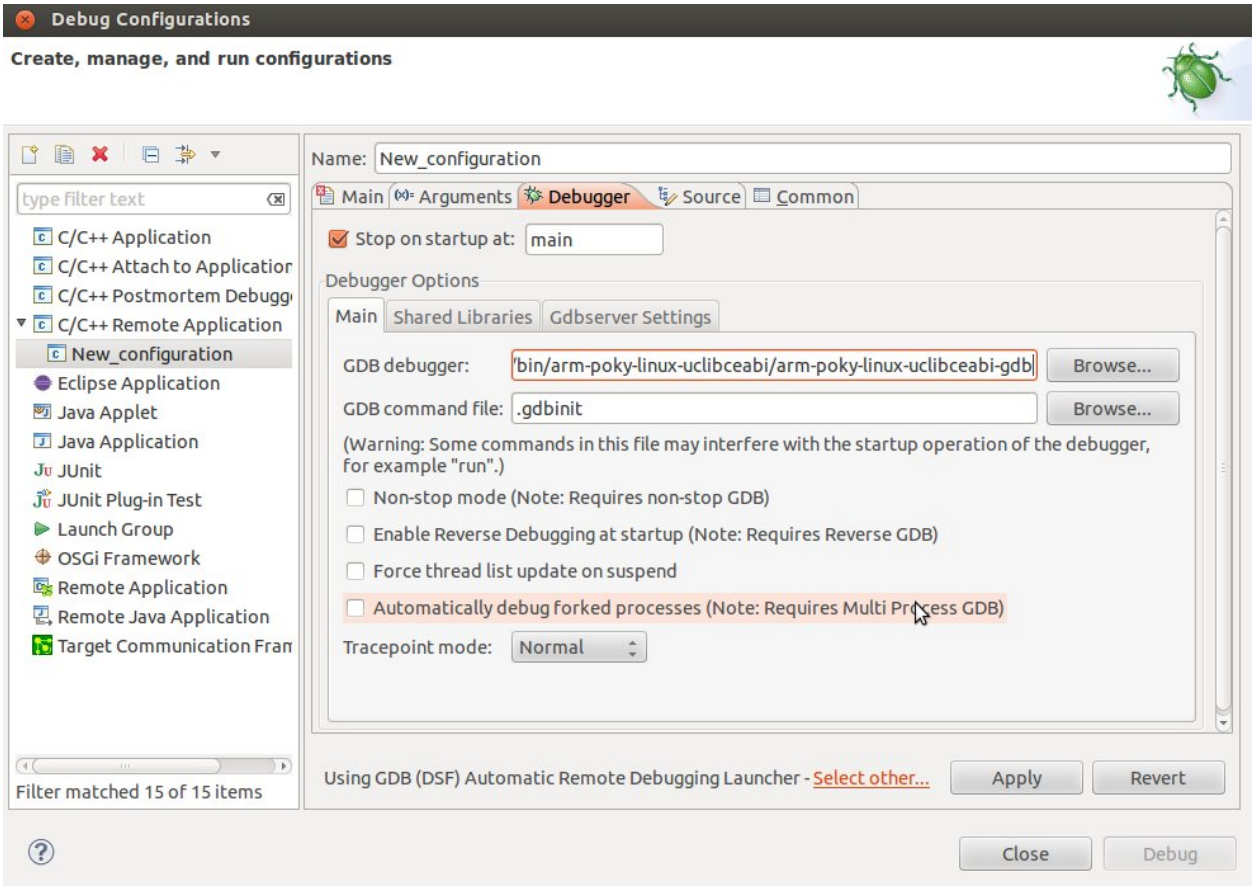
- Then press *Finish*.
- Use the drop-down menu now in the *Connection* field and pick up the IP Address you entered earlier.
- Enter the absolute path on the target into which you want to deploy the cross-compiled application. Use the *Browse* button near *Remote Absolute File Path for C/C++Application:* 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".

SDK Architecture

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

SDK

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

- **Yocto**, and
- **Eclipse**

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*.

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, Hachiko 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

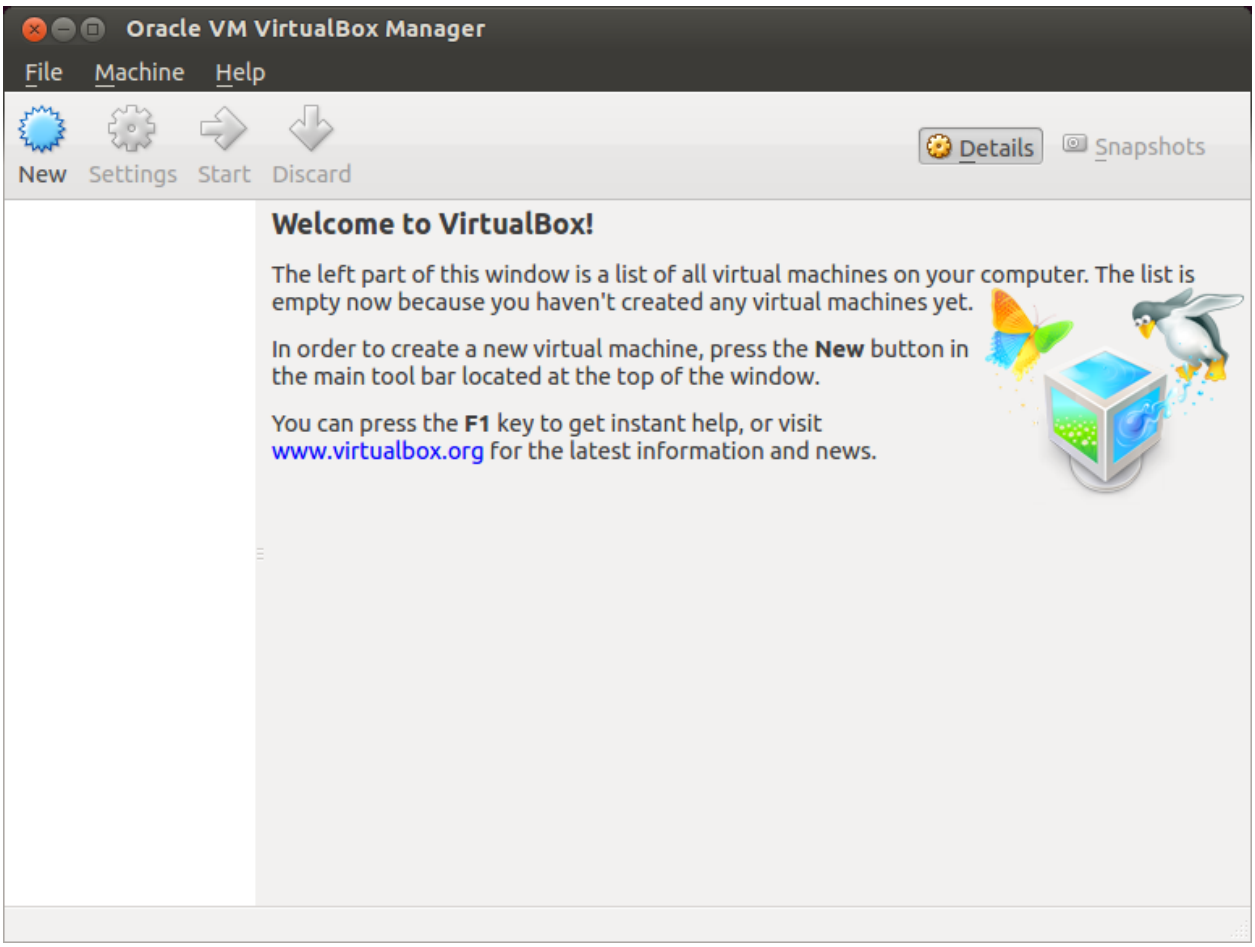
Download the version that suits your host operating system. You need to download and install the **Extension Pack** as well.

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

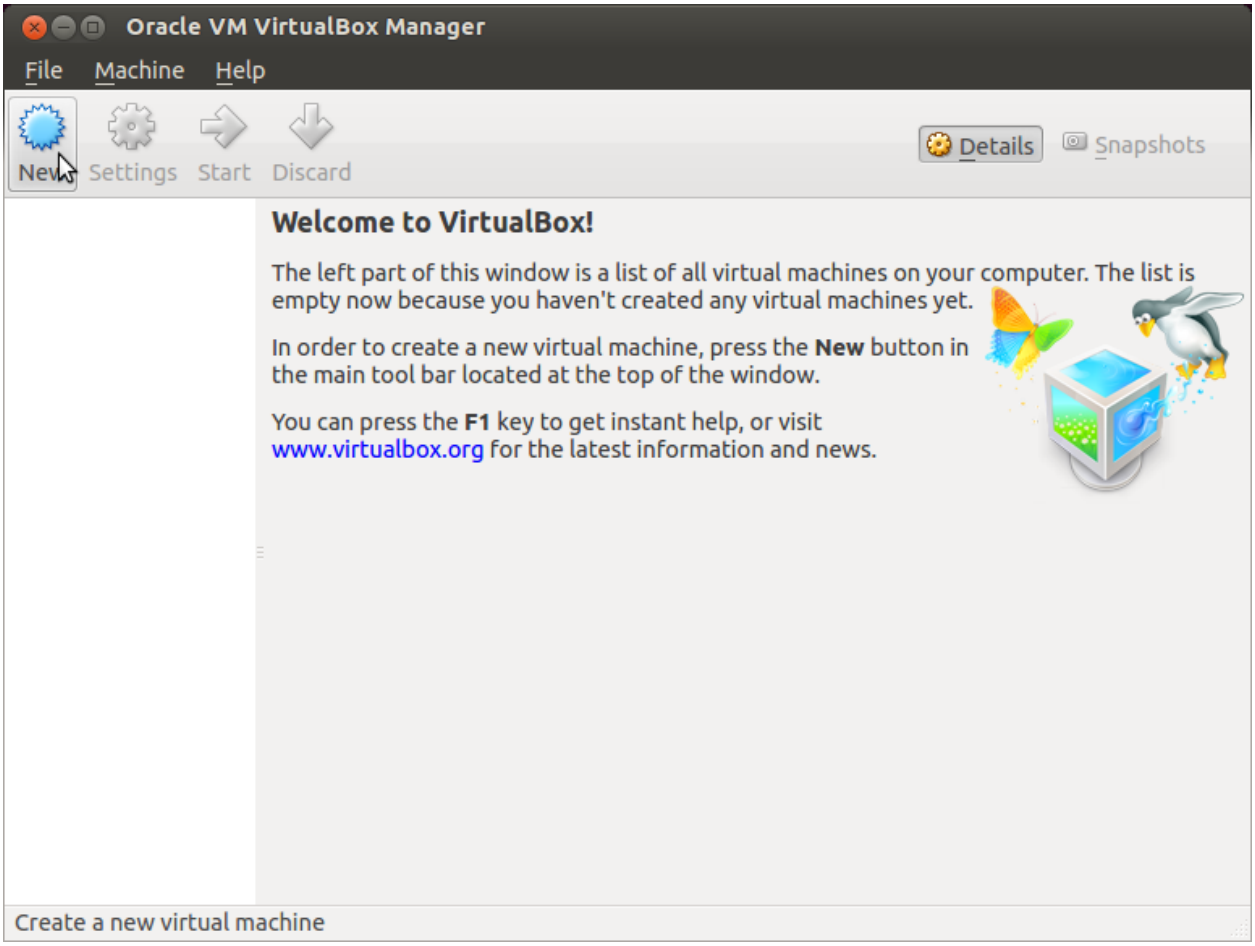
Install the software with all the default options.

Create a new Virtual Machine

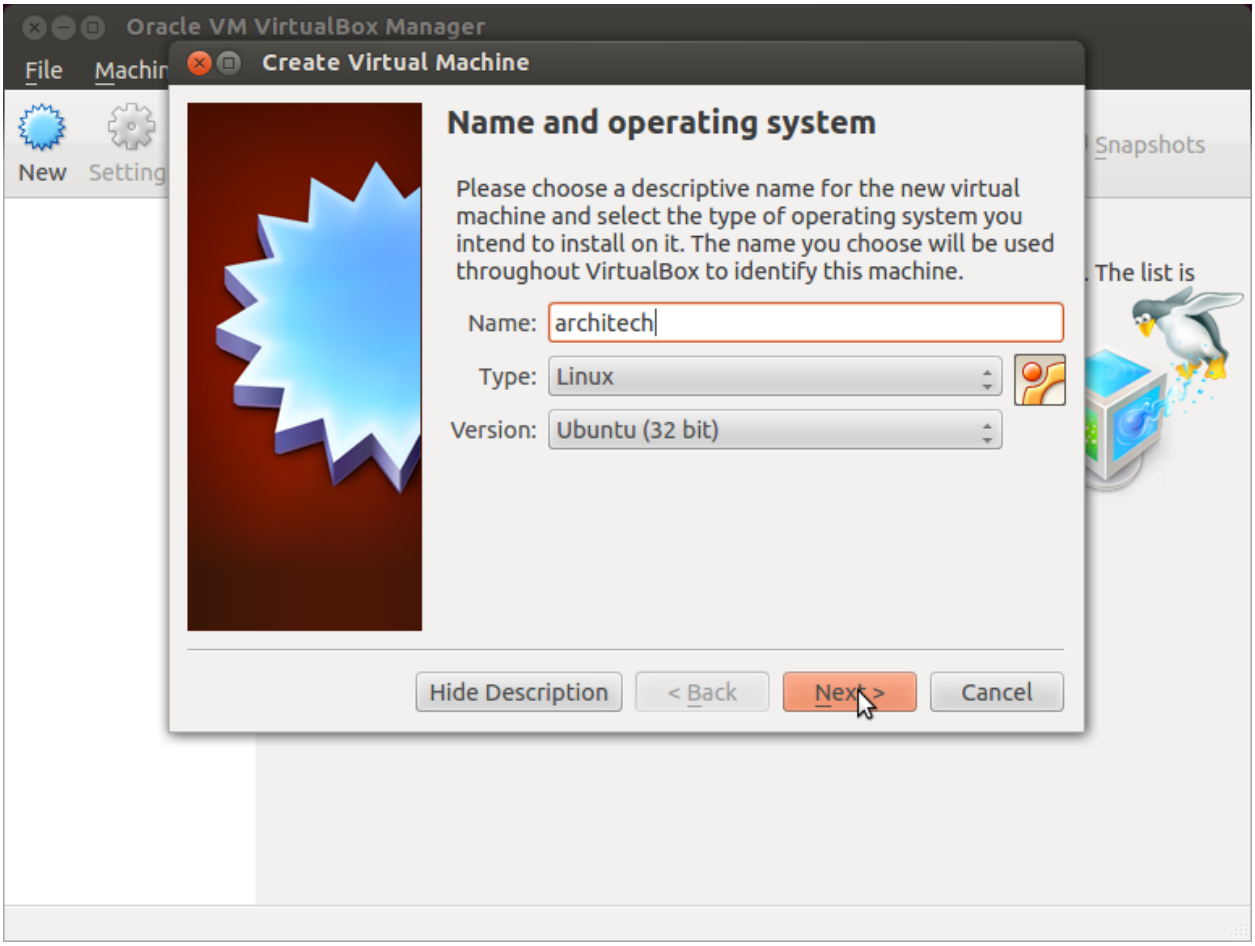
1. Run VirtualBox



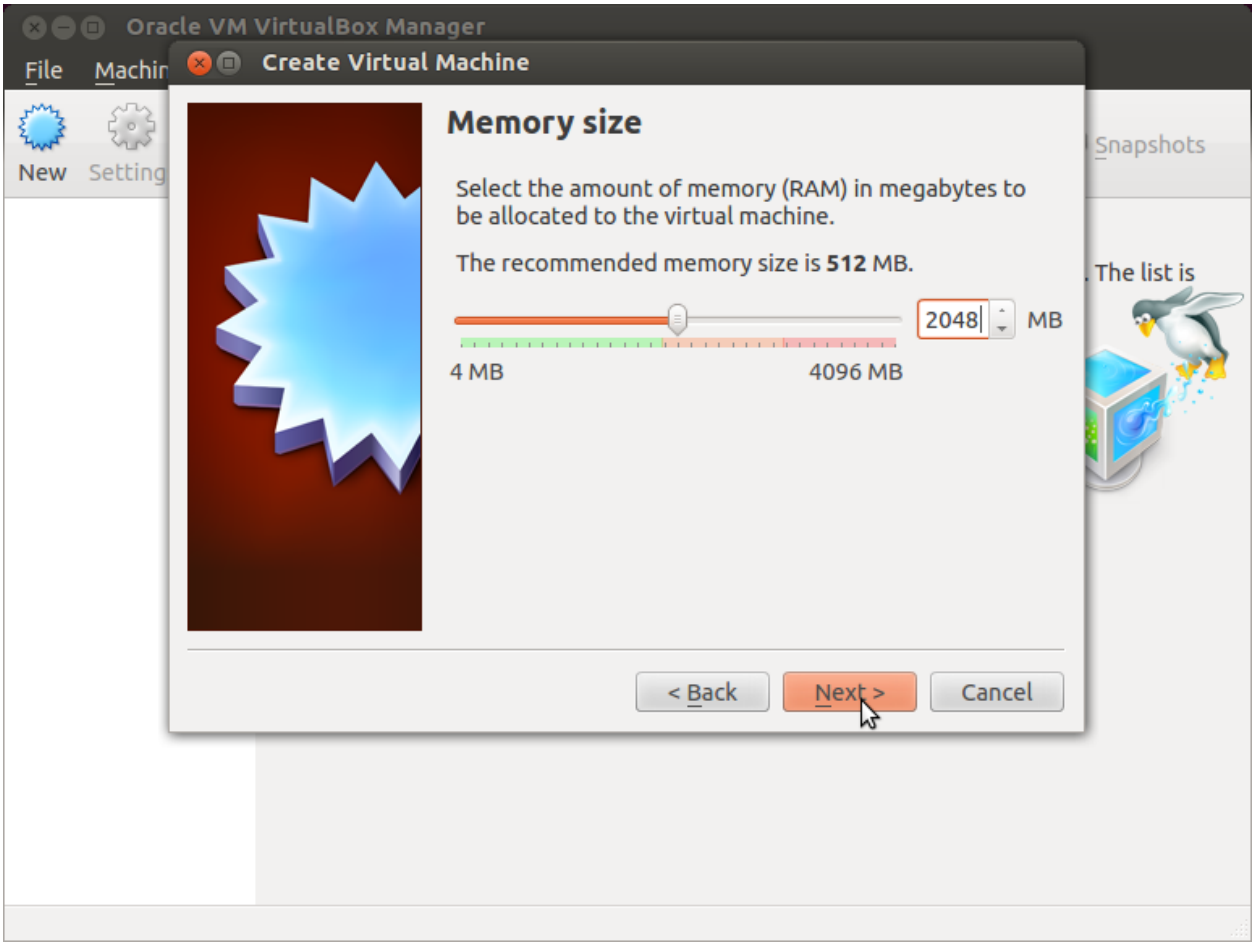
2. Click on *New* button



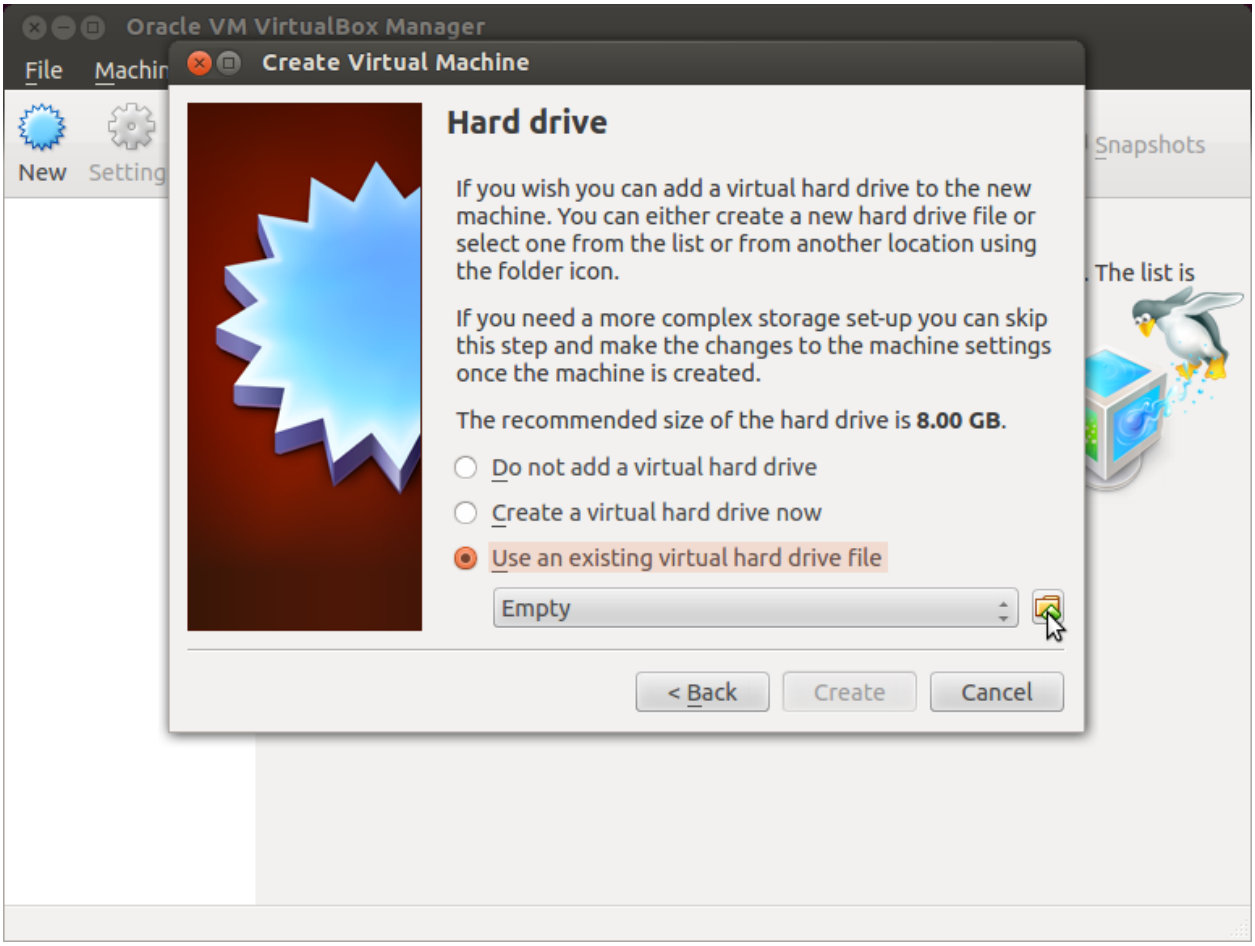
3. Select the name of the virtual machine and the operating system type



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*.

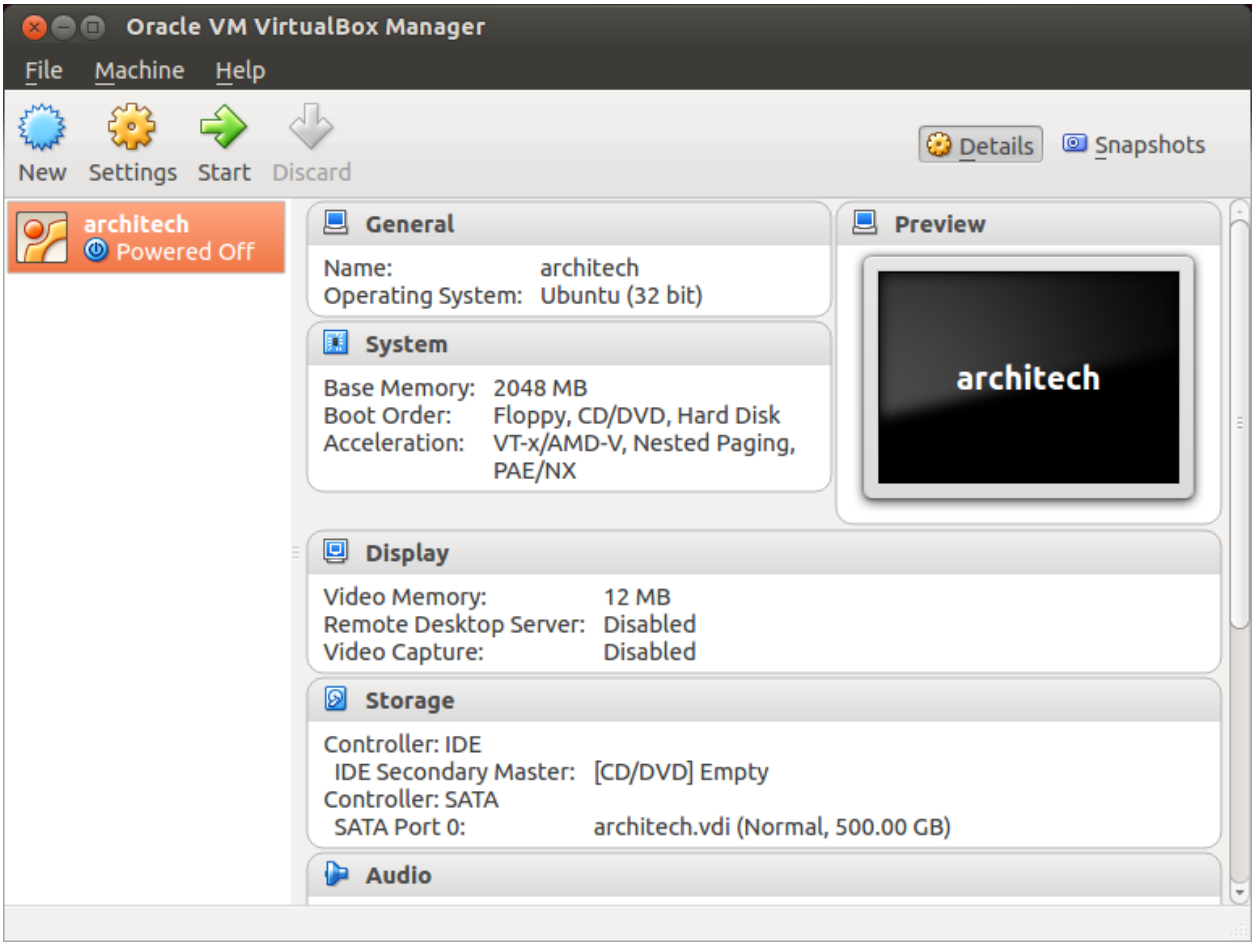


Setup the network

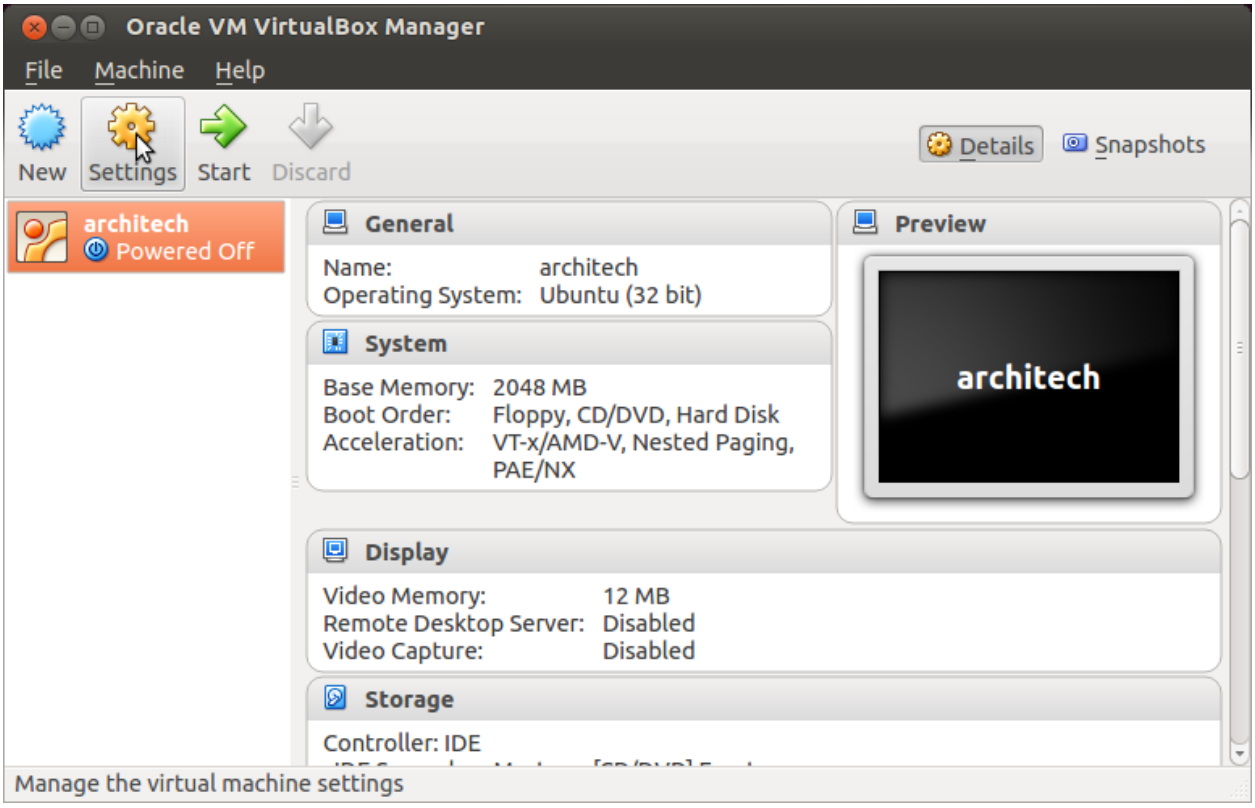
We need to setup a port forwarding rule to let you (later) use the virtual machine as a local repository of packages.

Note: The virtual machine must be off

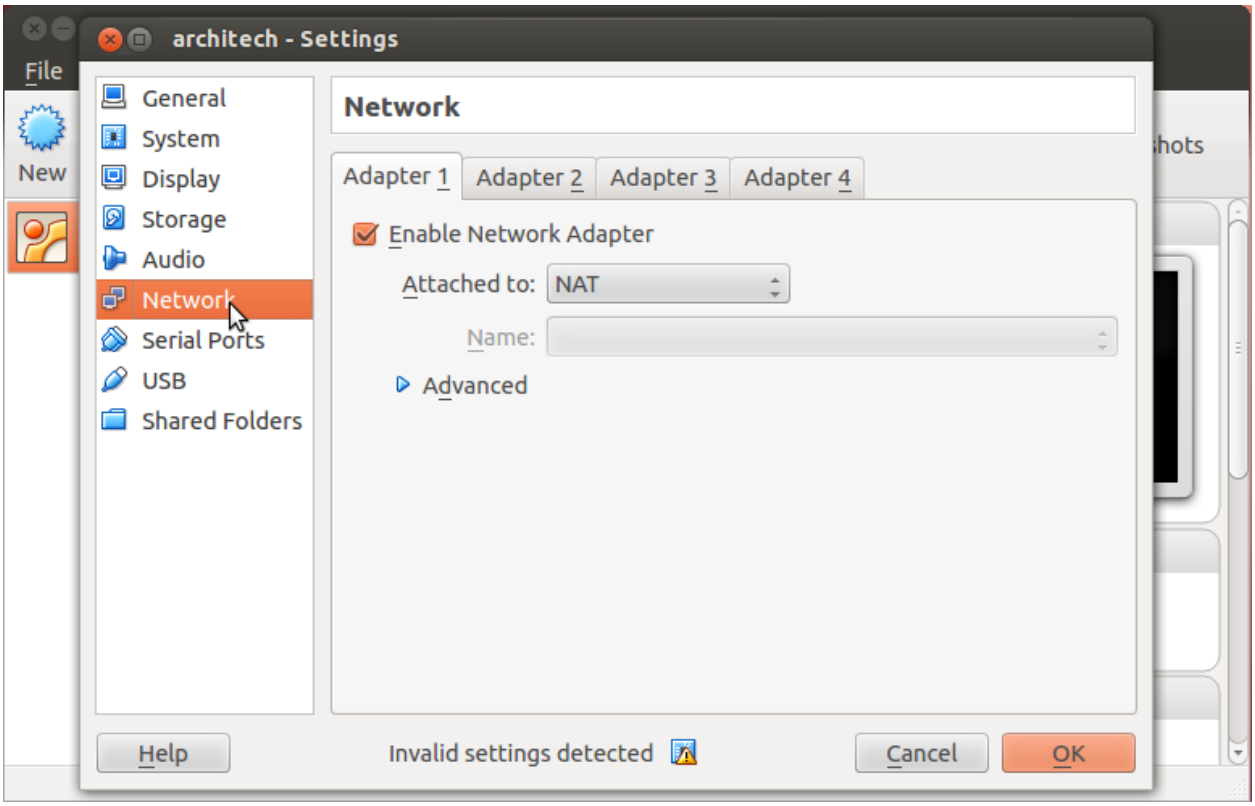
1. Select Architech's virtual machine from the list of virtual machines



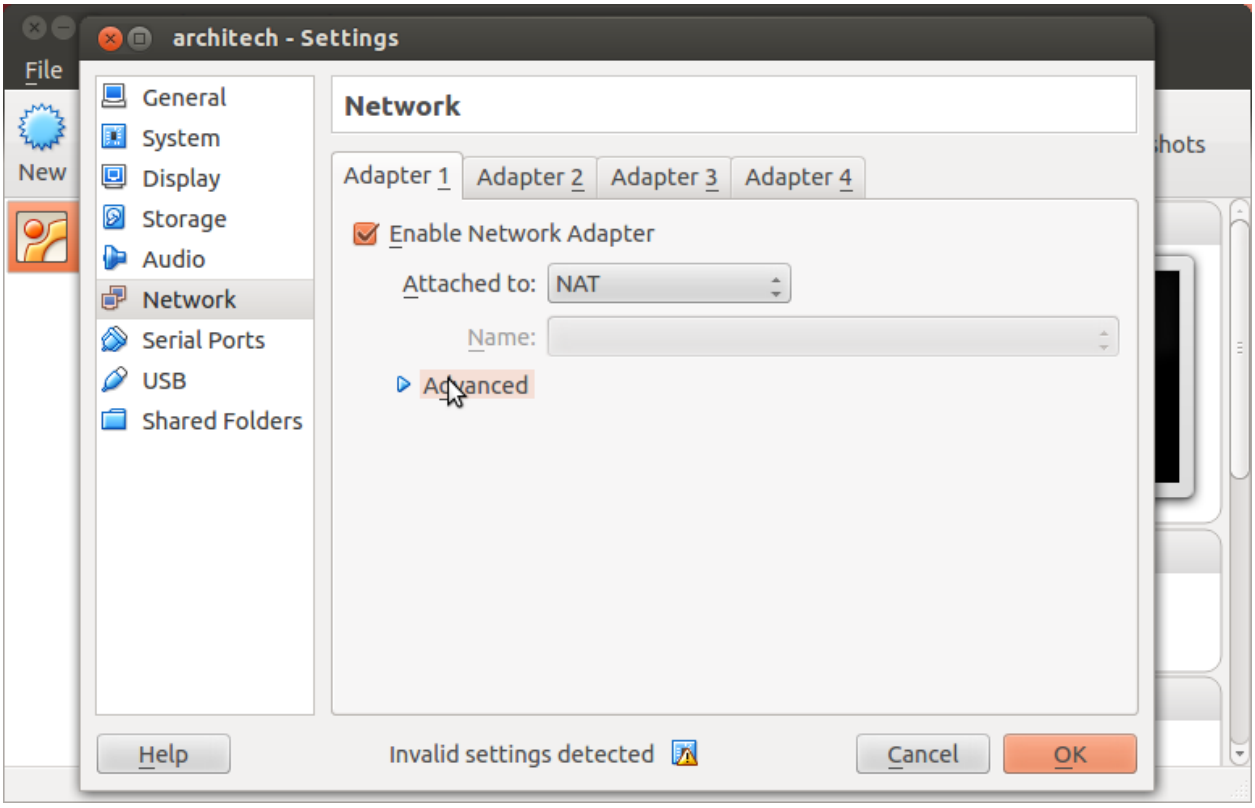
2. Click on *Settings*



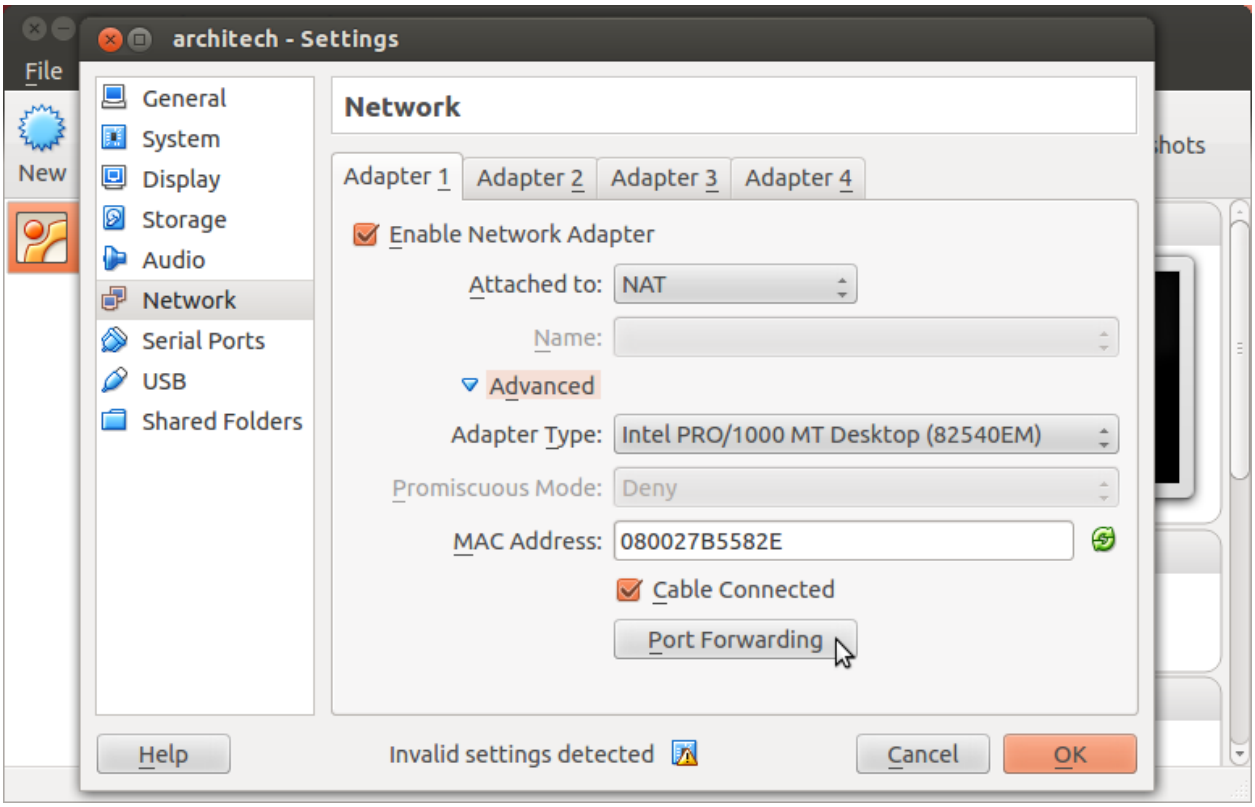
3. Select *Network*



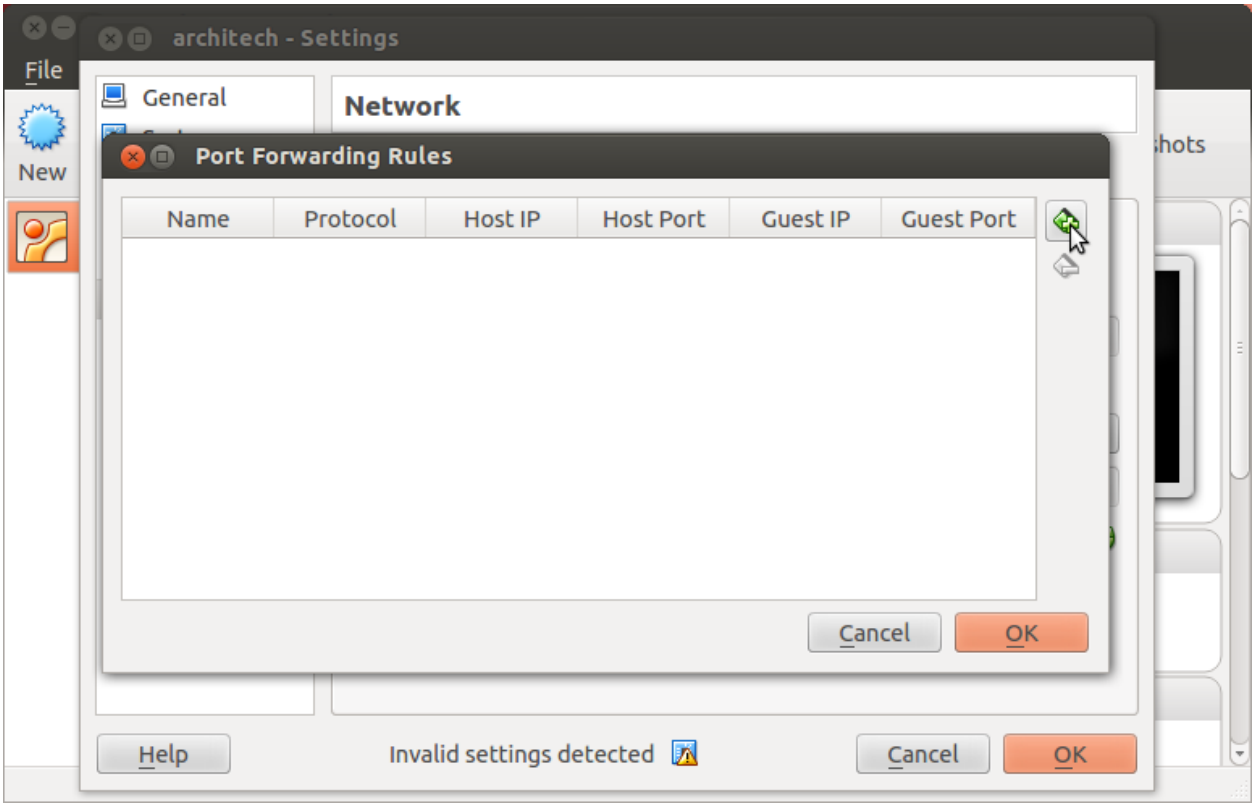
4. Expand *Advanced* of *Adapter 1*



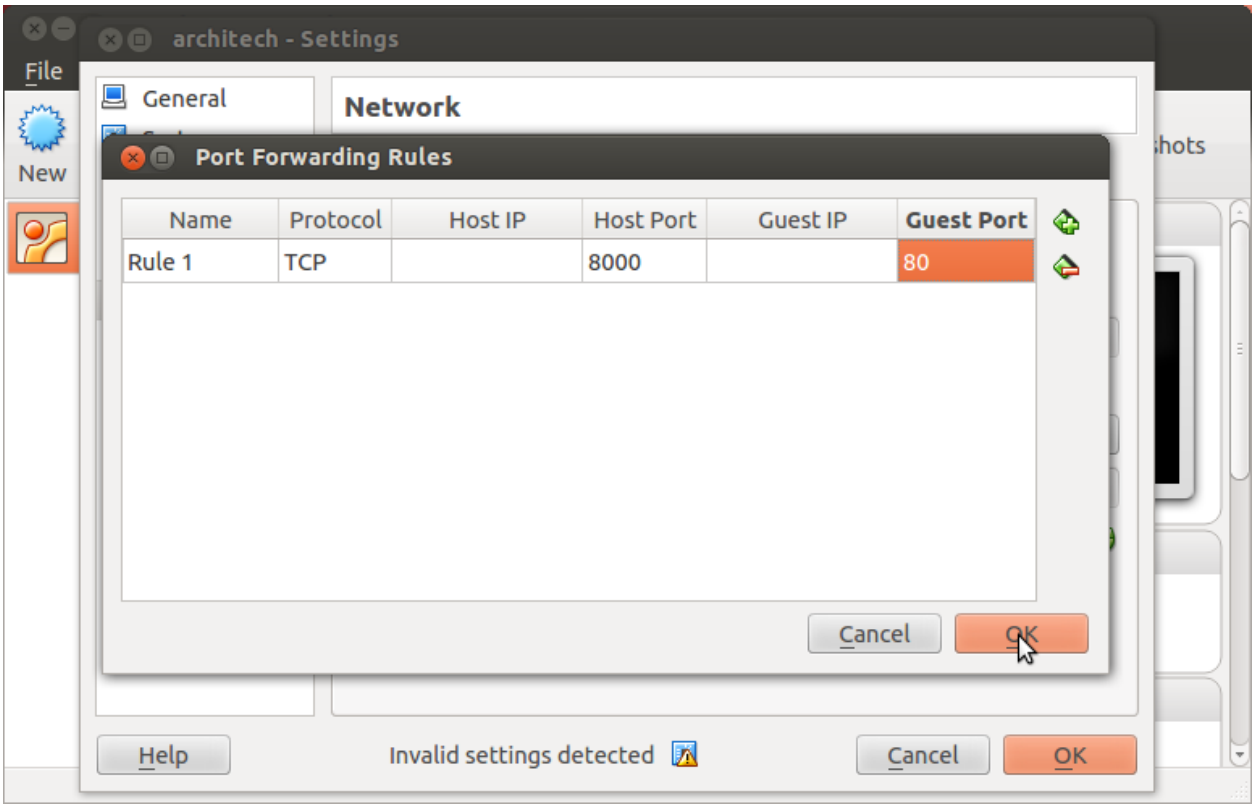
5. Click on *Port Forwarding*



6. Add a new *rule*



7. Configure the *rule*



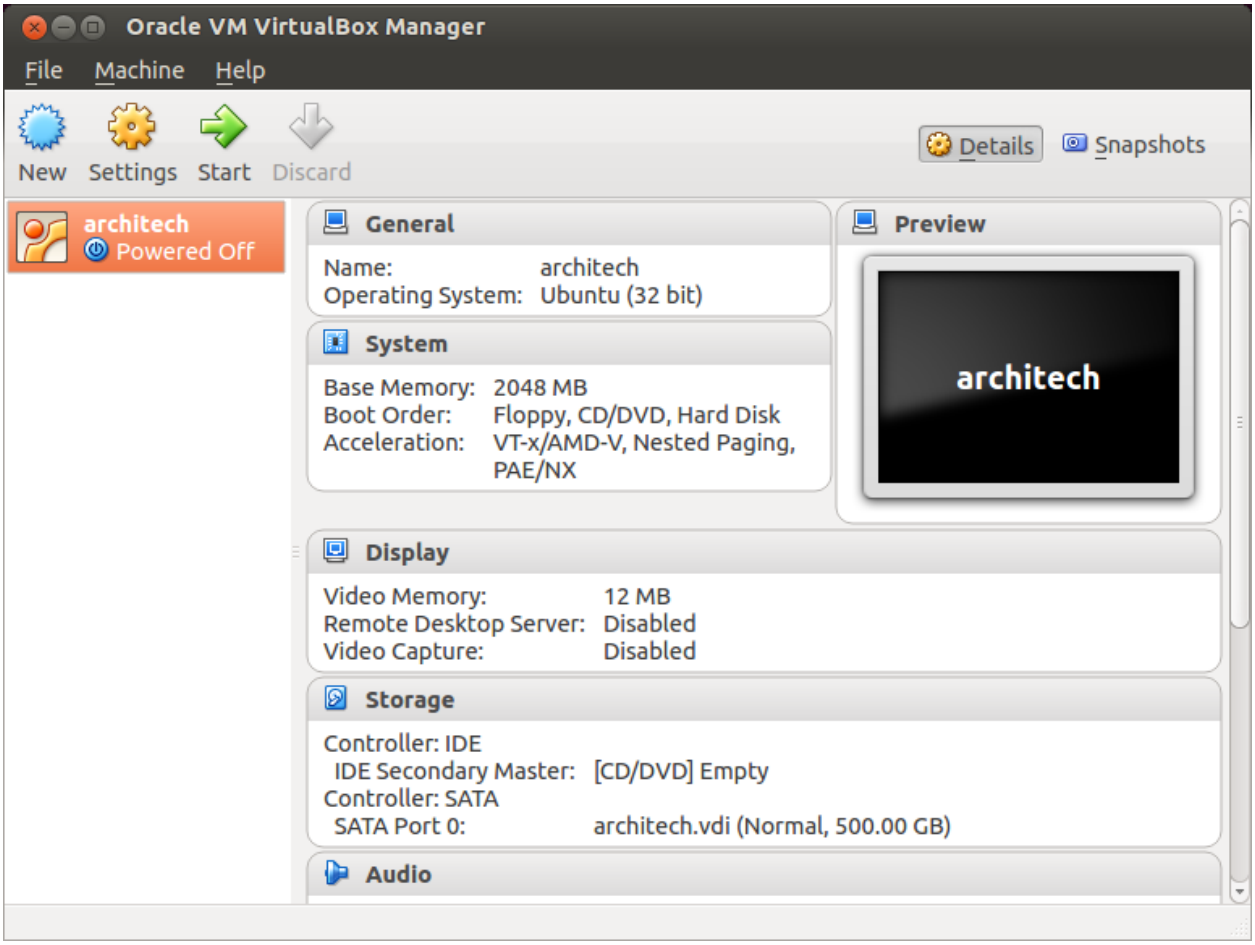
8. Click on *Ok*

Customize the number of processors

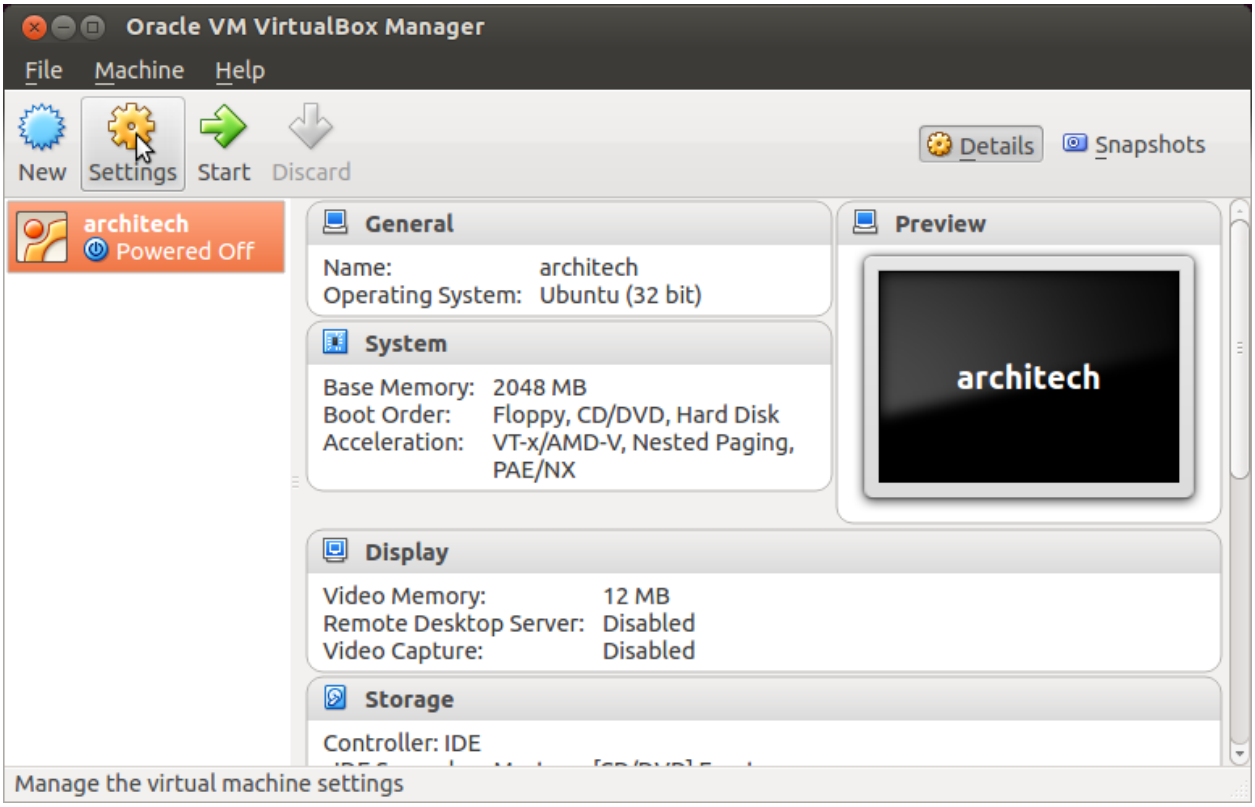
Building an entire system from the ground up is a business that can take up to several hours. To improve the performances of the overall build process, you can, if your computer has enough resources, assign more than one processor to the virtual machine.

Note: The virtual machine must be off

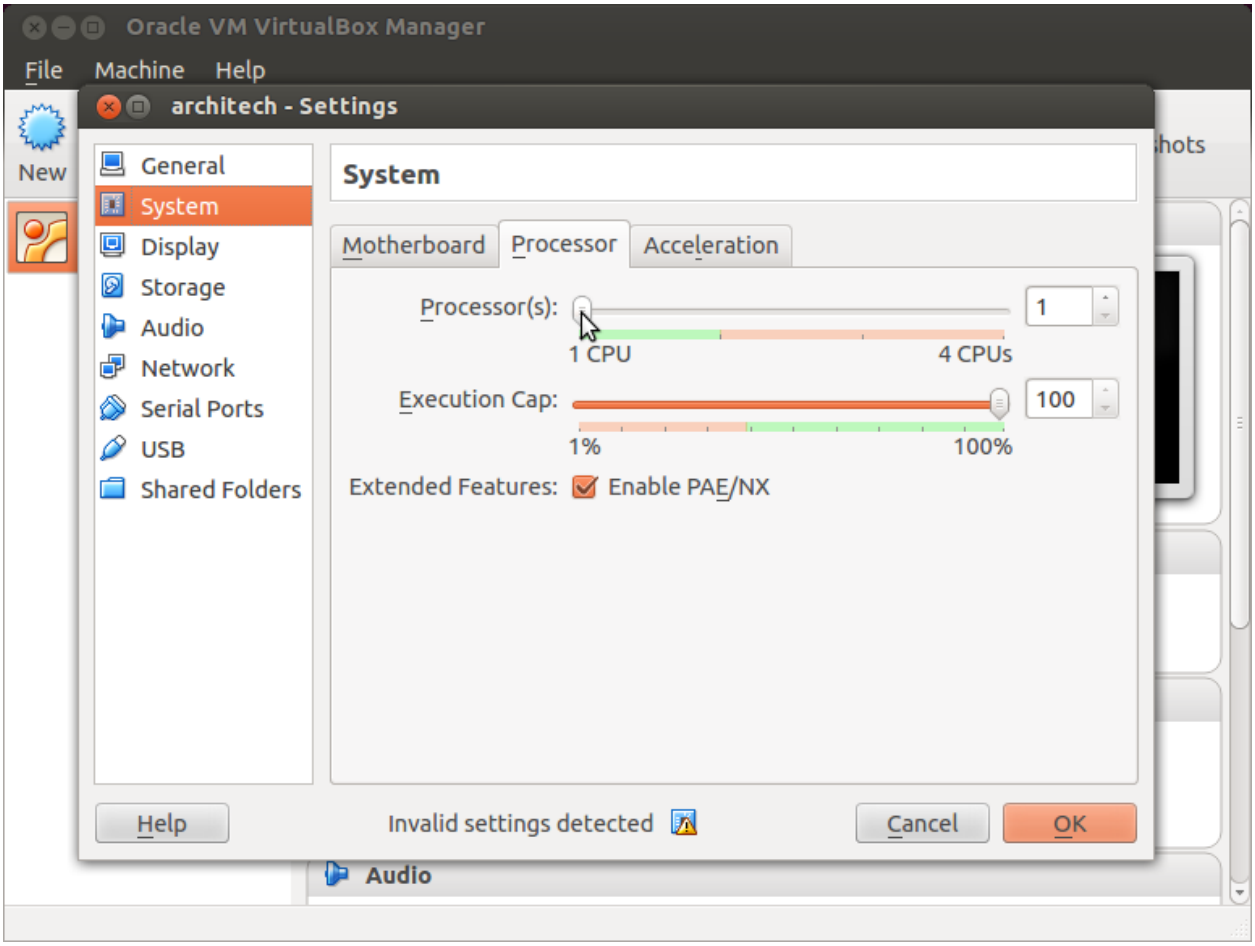
1. Select Architech's virtual machine from the list of virtual machines



2. Click on *Settings*



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



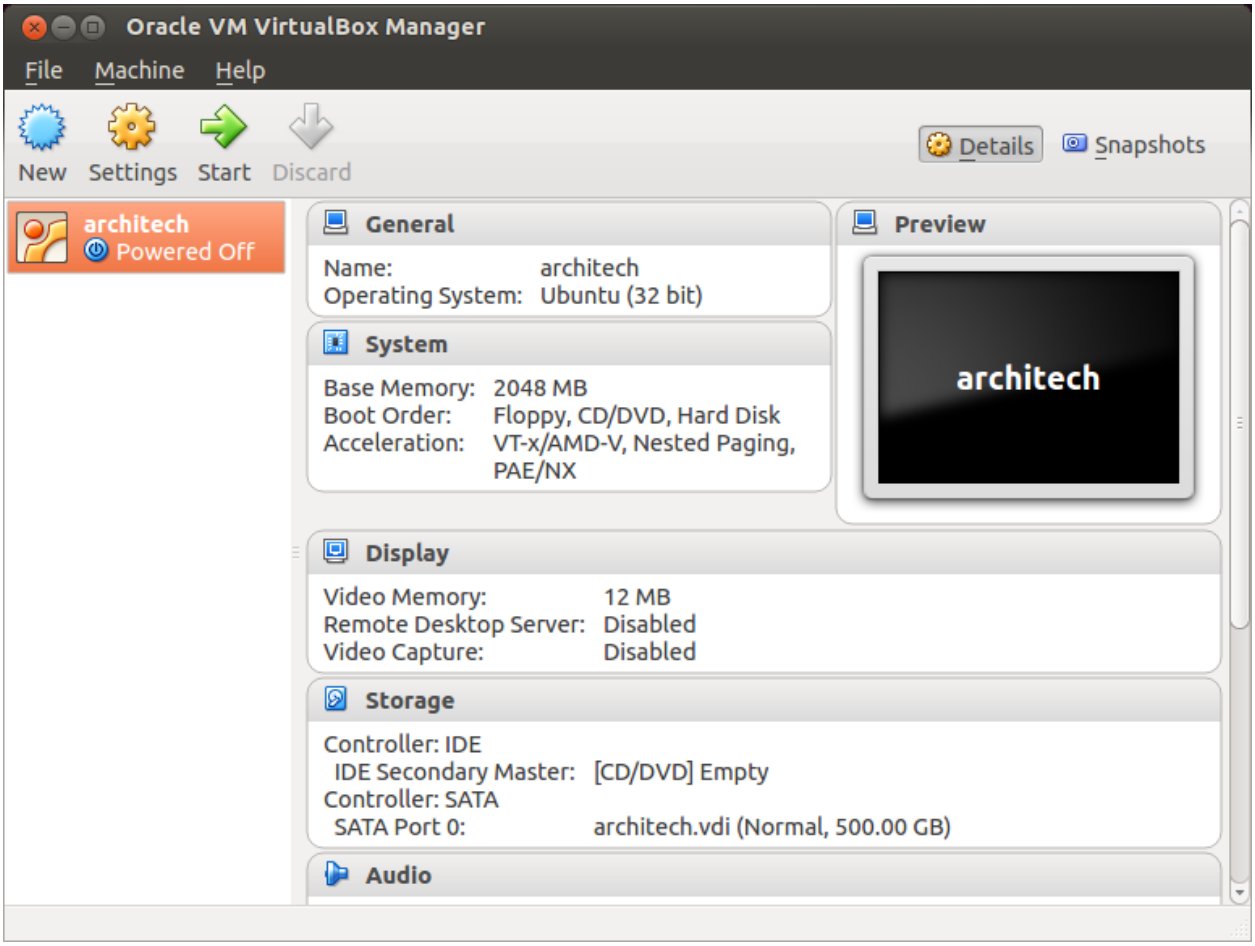
Create a shared folder

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.

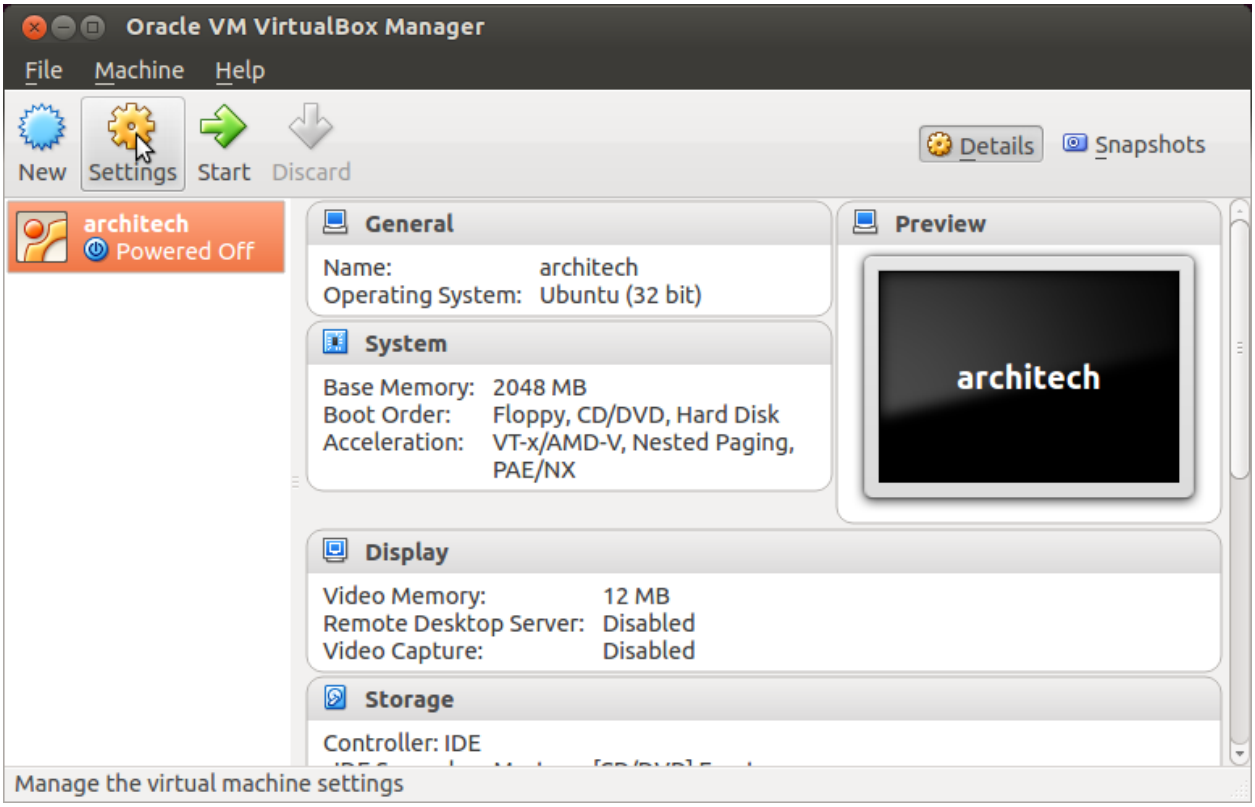
Note: The virtual machine must be off

---

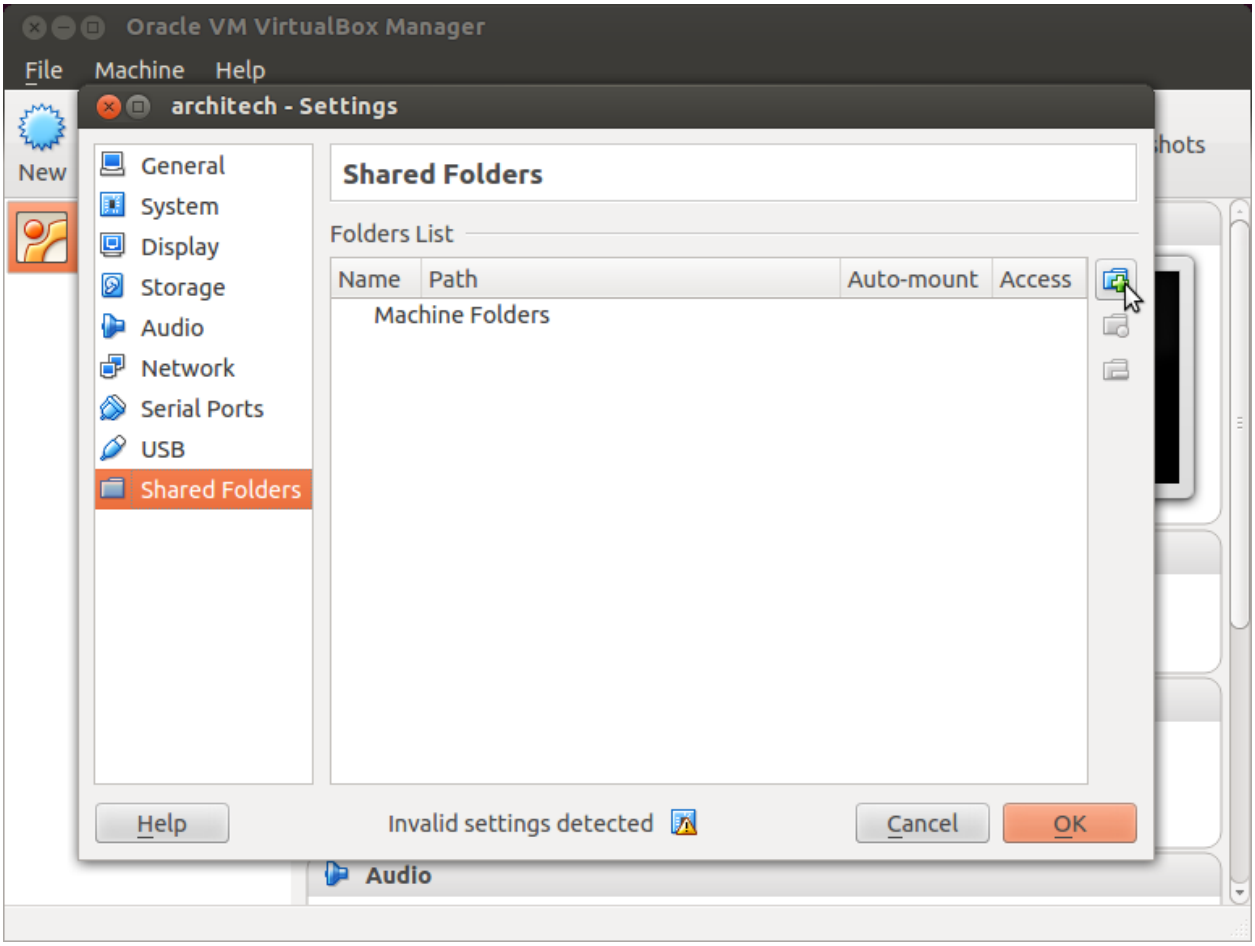
1. Select Architech's virtual machine from the list of virtual machines



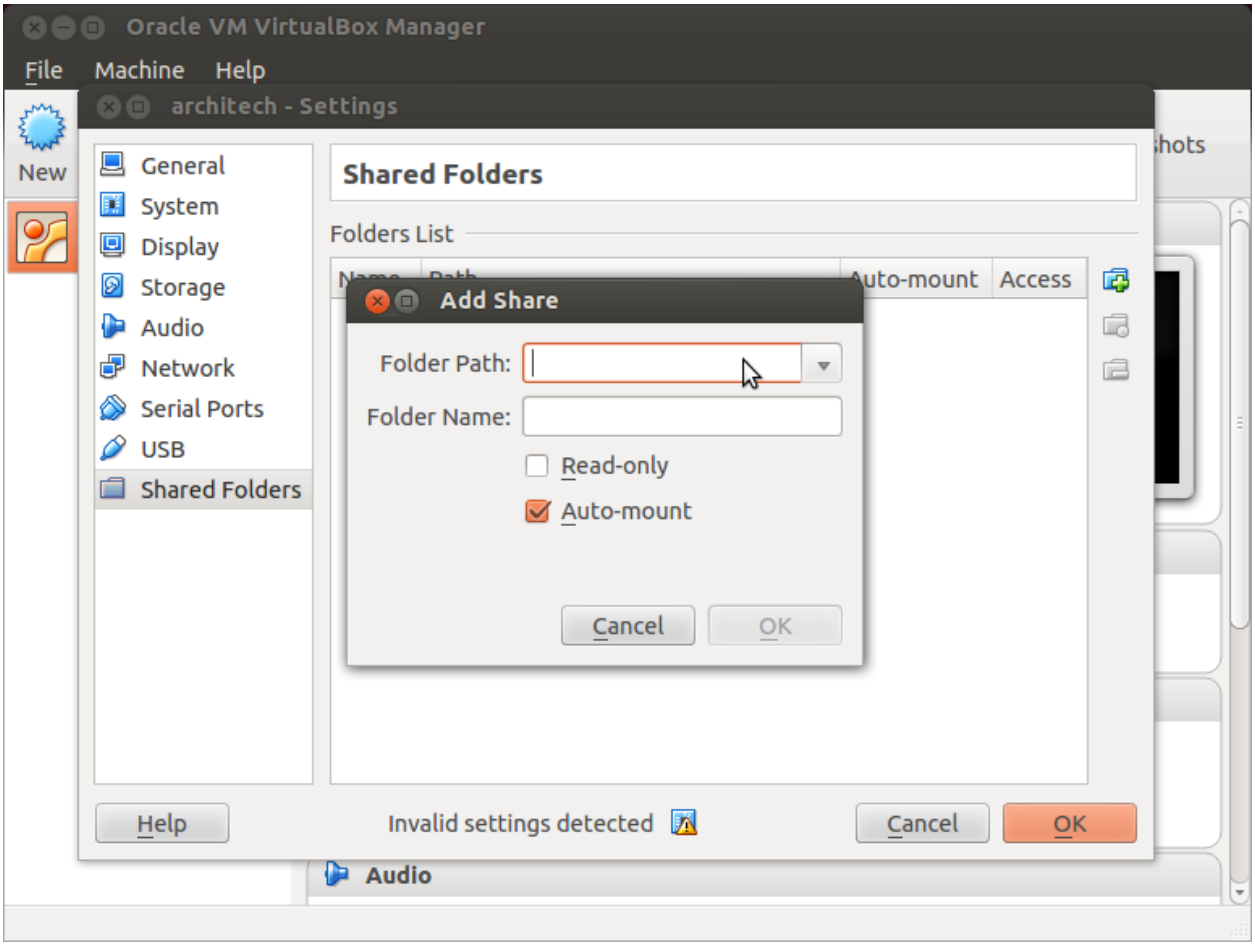
2. Click on *Settings*



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.



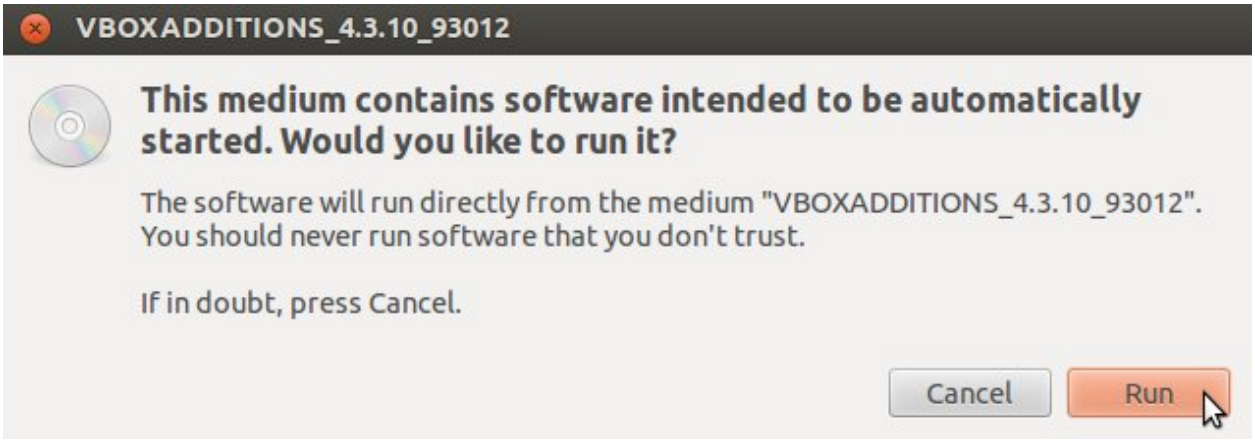
Once the virtual machine has been booted, the shared folder will be mounted under */media/* directory inside the virtual machine.

Install VBox Additions

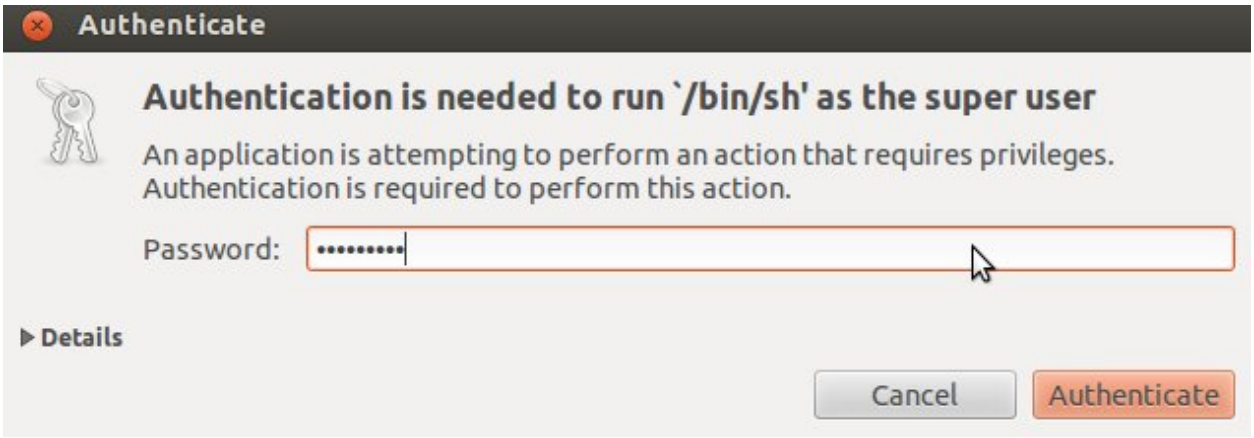
The VBox addictions add functionalities to the virtual machine such as better graphic driver and more. It is already installed in the SDK but is important re-install it to configuring correctly the virtual machine with your operating system.

1. Starts the virtual machine

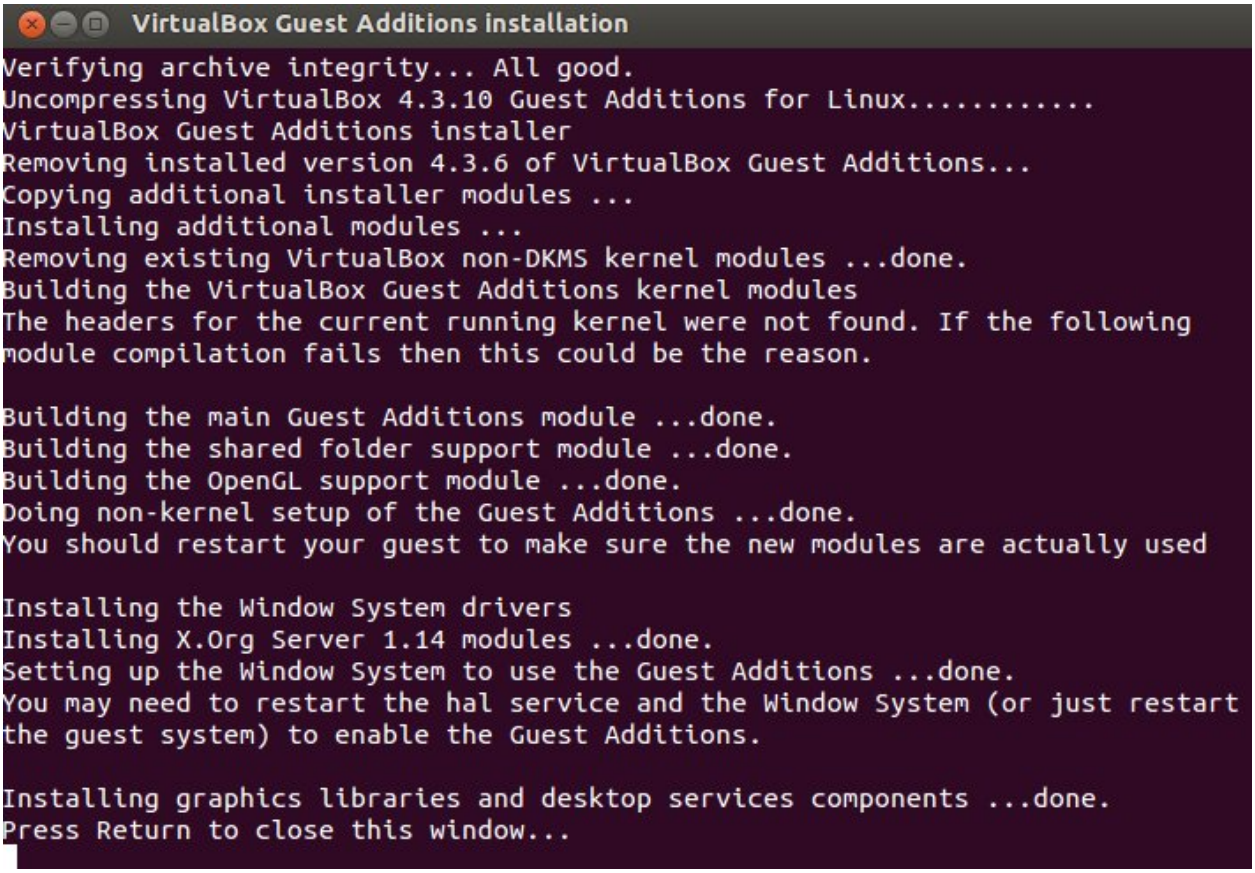
2. Click on the virtual box menu to the voice *Devices* and select *Insert Guest Additions CD Images...*. A message box will appear at the start of the installation, click on *run* button



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

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:

hachiko-tiny directory contains all the tools composing the ArchiTech SDK for Hachiko board, along with all the information needed by the splash screen application. In particular:

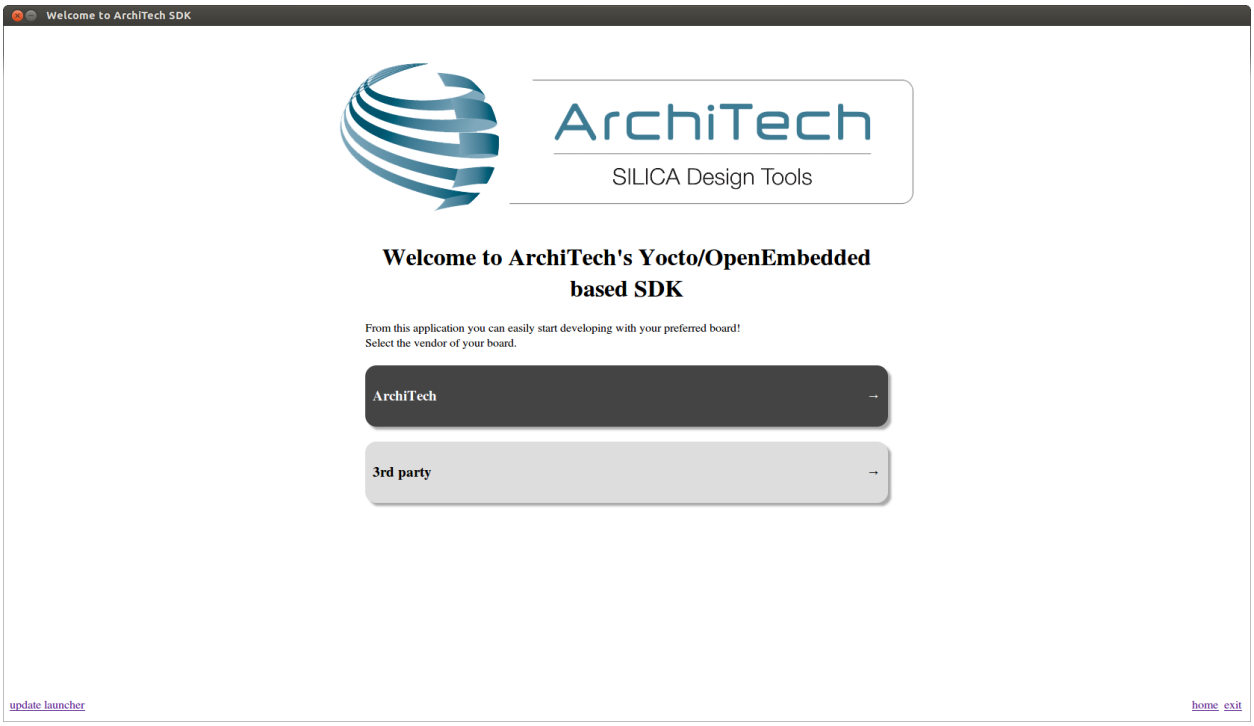
- *eclipse* directory is where Eclipse IDE has been installed
- *java* directory is where the Java Virtual Machine has been installed (needed by Eclipse)
- *qtcreator* contains the installation of Qt Creator IDE
- *splashscreen* directory contains information and scripts used by the splash screen application,
- *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 Hachiko requires, along with Poky and the build directory

Splash screen

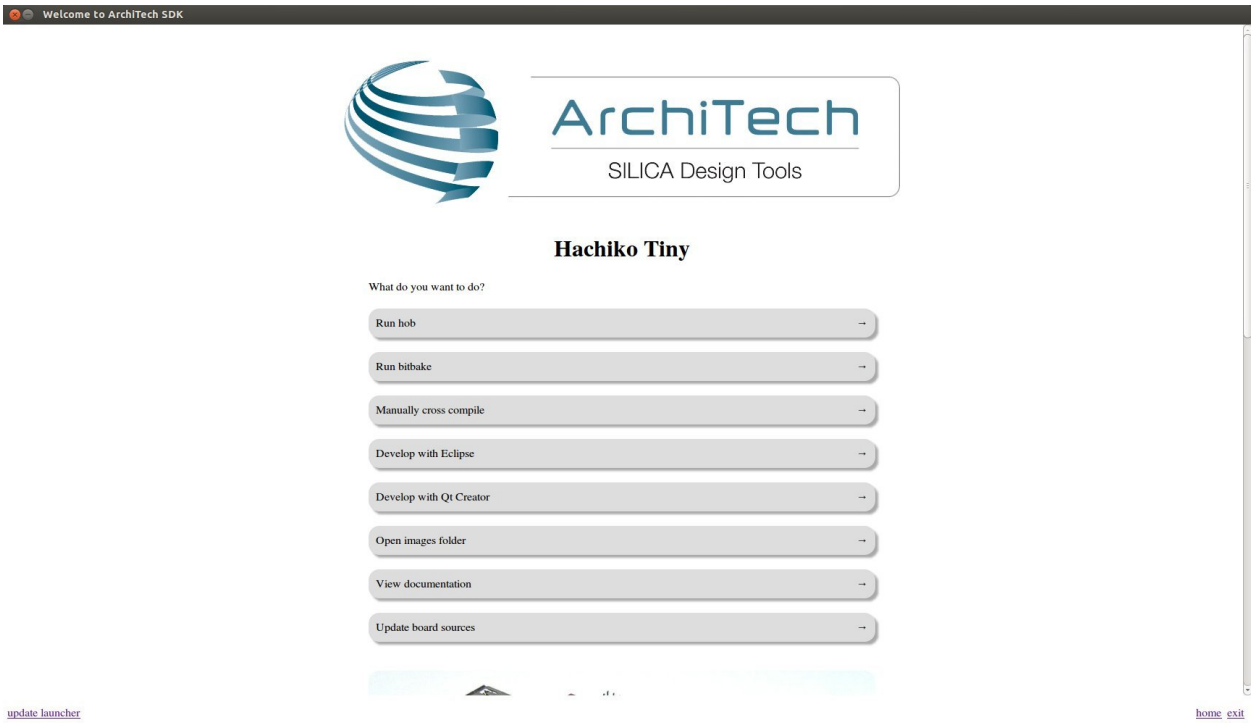
The splash screen application has been designed to facilitate the access to the boards tools. It can be opened by clicking on its *Desktop* icon.

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



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

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



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 Hachiko board.

Installation

Architech's Yocto based SDK is built on top of **Ubuntu 12.04 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 12.04 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 Hachiko:

This script needs the same tools/packages required by *machine_install*

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 hachiko 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 *dora* of *meta-hachiko*:

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.

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.

U-boot

The bootloader used by Hachiko board is **U-Boot**. If you need to modify the bootloader or to recompile it you have two ways to get the sources:

- use the sources you find in Yocto build directory after having compiled at least once a yocto image, or
- download the official u-boot release, than patch it with the BSP patches for Hachiko board.

Anyway, we will assume in this guide that u-boot sources will be copied to:

and such directory does not yet exists on your PC. Of course, you are free to choose the path you like the most for u-boot sources, just remember to replace the path used in this guide with your custom path.

From Yocto sources

The first way is based on the sources set up by the Yocto build system. However, it is never advisable to work with the sources in the Yocto build directory, if you really want to modify the source code inside the Yocto environment we strongly suggest to refer to the official Yocto documentation. To avoid messing up Yocto recipes and installation, it is desirable to copy the patched u-boot sources you find in the build directory elsewhere. The directory we are talking about is this one:

Replace:

all over this chapter with your custom build directory path if you are not working with the default SDK build directory.

From the official u-boot release

The second way is to get the official U-Boot sources and patch them with Hachiko BSP patches. Hachiko board uses U-Boot version 2013.04, which can be downloaded from:

ftp://ftp.denx.de/pub/u-boot/u-boot-2013.04.tar.bz2.

with this command:

then extract the tarball:

Patches are to be found in the Yocto meta-layer **meta-hachiko**. You can use them right away if you are working with the SDK:

However, if you are not working with the official SDK the most general solution to check them out and patch the sources is:

Configuration and board files for Hachiko board are in:

Suppose you modified something and you wanted to recompile the sources to test your patches, well, you need a cross-toolchain (see *Cross compiler* Section). Luckily, the SDK already contains the proper cross-toolchain. To use it to compile the bootloader or the operating system kernel, just run:

then you can run these commands to compile it:

Once the build process completes, you can find *u-boot.bin* file inside directory */home/architech/Documents/u-boot*.

If you are not working with the virtual machine, you need to get the toolchain from somewhere. The most comfortable way to get the toolchain is to ask *Bitbake* for it:

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

Run the script and you 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 first to get it ready:

Here you go, you now have the proper working environment to compile *u-boot* (or the Linux kernel).

Linux Kernel

Build from sources

As seen for **U-Boot** in the previous section, the first step is to get the kernel sources, which in this guide we assume that they will be copied to directory:

You are free to choose the path you like the most for the kernel sources, just remember to replace the path used in this guide with your custom path. The suggested way, even for the kernel, to get the sources is to take them from the Yocto build directory (copying them in the aforementioned directory to avoid messing up Yocto):

or from the vanilla kernel tarball at this URL:

http://kernel.org/pub/linux/kernel/v3.0/linux-3.8.13.tar.bz2.

and patch it using the patches found in Hachiko BSP meta-layer:

If you are not developing from within the official SDK, the most general solution to check them out and patch the sources is:

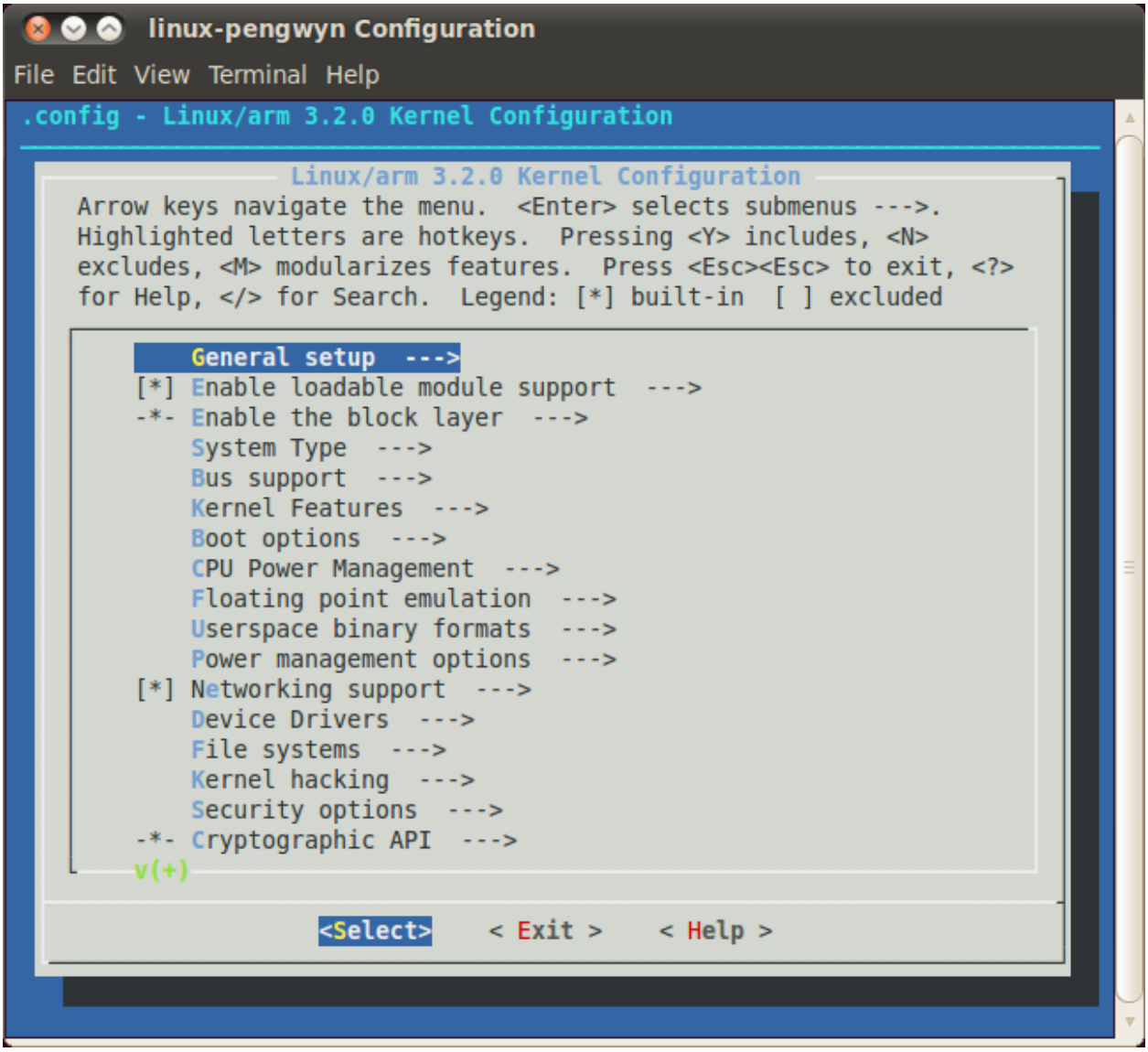
To compile the kernel without bitbake just execute these commands:

If you are not developing from within the original SDK, you are going to need to *install the cross toolchain*. The output of the compilation process is:

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/hachiko/* 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/hachiko-tiny/yocto/build/tmp/deploy/images/hachiko/*.

Meta Layer

A Yocto/OpenEmbedded meta-layer is a directory that contains recipes, configuration files, patches, and others things all needed by Bitbake to properly "see" and build a BSP, a distribution, and a (set of) package(s).

meta-hachiko is stored in a git repository that can be cloned with:

For information about Yocto, Bitbake and the directories tree inside the meta-layer, please refer to the Yocto documentation.

Hachiko meta-layer defines two different machines: **hachiko** and **hachiko64**, the latter to be used when an external SDRAM is available on the board. Whereas *hachiko64* machine can be used for virtually any distro and image available on Yocto, *hachiko* machine must use a custom tailored distro and image to be able to fit in the limited amount of SRAM available on-chip. This documentation is about **hachiko** machine.

To manually select board and distribution for *Bitbake*, make sure that file *local.conf*, that in the SDK has this path:

contains the assignment of these variables:

When asked to build an image, *Bitbake*/*Hob* produces several files as output, of which these are needed to build the whole system:

- the Linux kernel - file *uImage*
- the device tree - file *uImage-rza1-hachiko.dtb*
- U-boot bootloader - file *u-boot.bin*
- the root file system - file **.tar.bz2*

Within the SDK, the files will be emitted to directory:

If you are not working with the SDK, just replace:

with your build directory path.

Root FS

By default, Hachiko Yocto/OpenEmbedded SDK will generate two different types of files when you build an image:

- **.tar.bz2*, and
- **.jffs2*

The *.tar.bz2* file can be expanded in your final medium partition (flash memory or USB stick) or on your host development system and used for build purposes with the Yocto Project. File *.jffs2* can be written out "as is" on the final medium (usually flash NOR partition) with, for example, *dd* program:

or

Important: Be very careful when you use *dd* or *flashcp* to write to a device to pick up the right device

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.

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 Hachiko is located under:

and the splash screen has a facility (a button located under Hachiko'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 Hachiko is hachiko:

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. *tiny-image* 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.

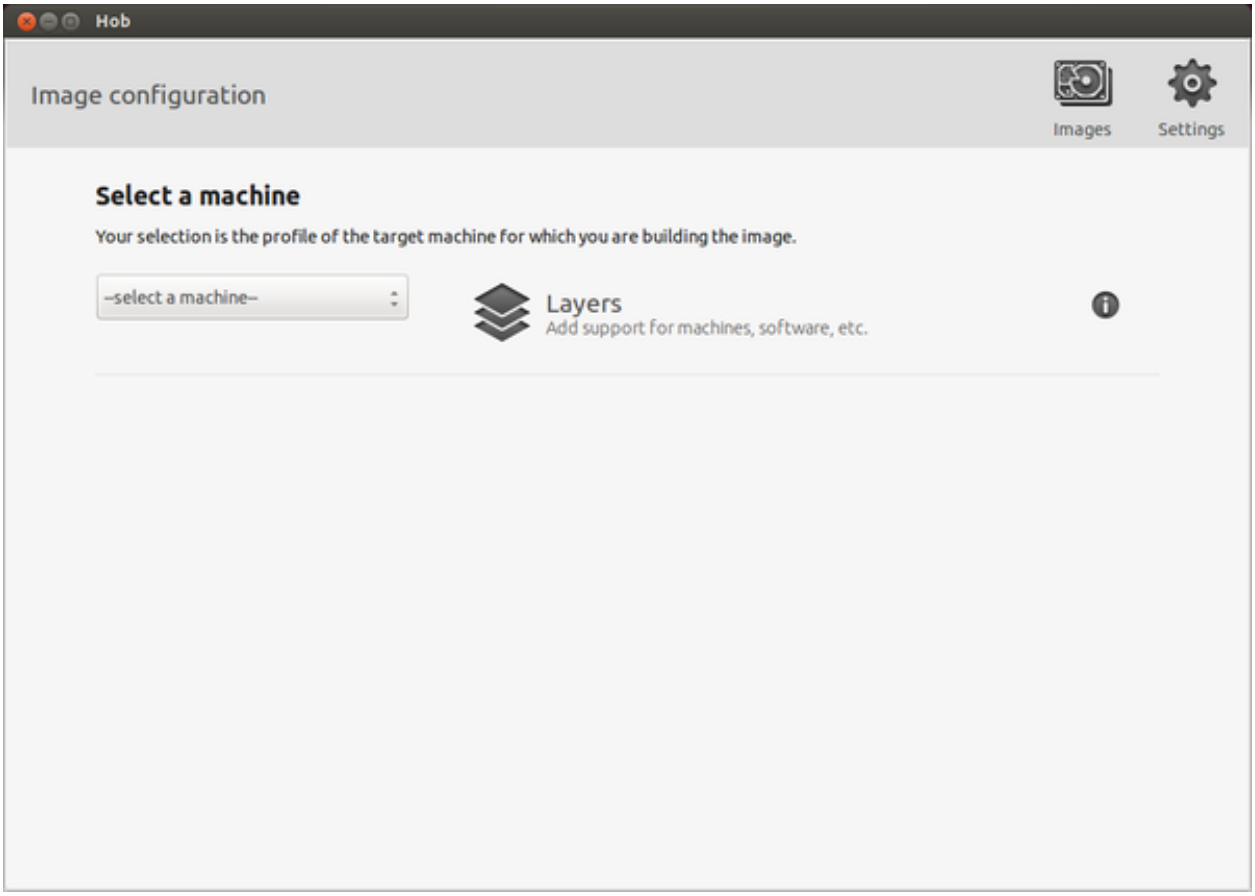
Hob

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.

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 Hachiko. 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.

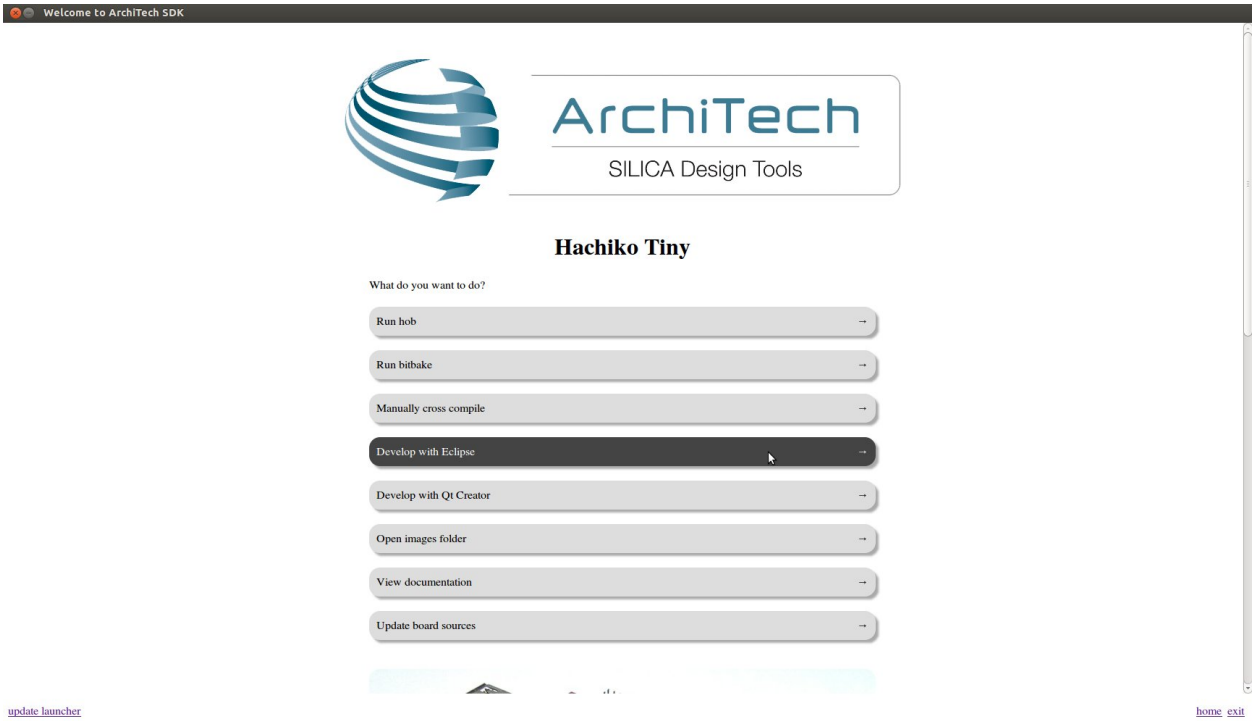
To write your application you need:

- 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
- Hachiko *powered up* with the aforementioned root file system
- a working *serial console* terminal
- a working *network* connection between your workstation and the board (connector *XF1*), 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.

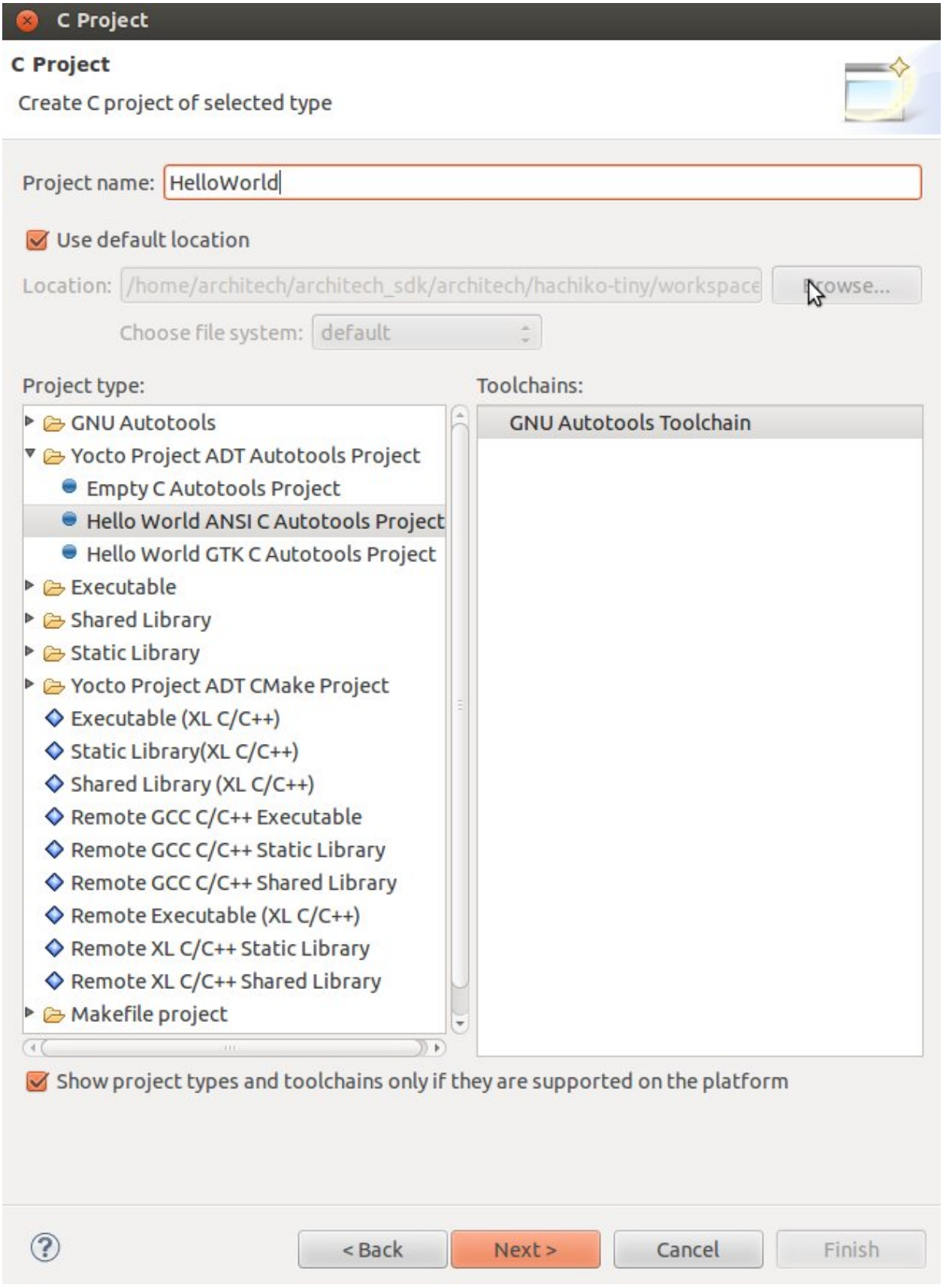
Creating the Project

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:

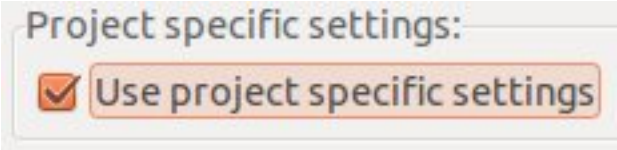
- Select File→New→Project...
- Under *C/C++*, double click on *C Project* to create the project.
- Click on "Next" button
- Expand *Yocto Project ADT Autotools Project*.
- Select *Hello World ANSI C Autotools Project*. This is an Autotools-based project based on a Yocto Project template.


C Project
C Project
Create C project of selected type
Project name: HelloWorld
Use default location
Location: /home/architech/architech_sdk/architech/hachiko-tiny/workspace
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*

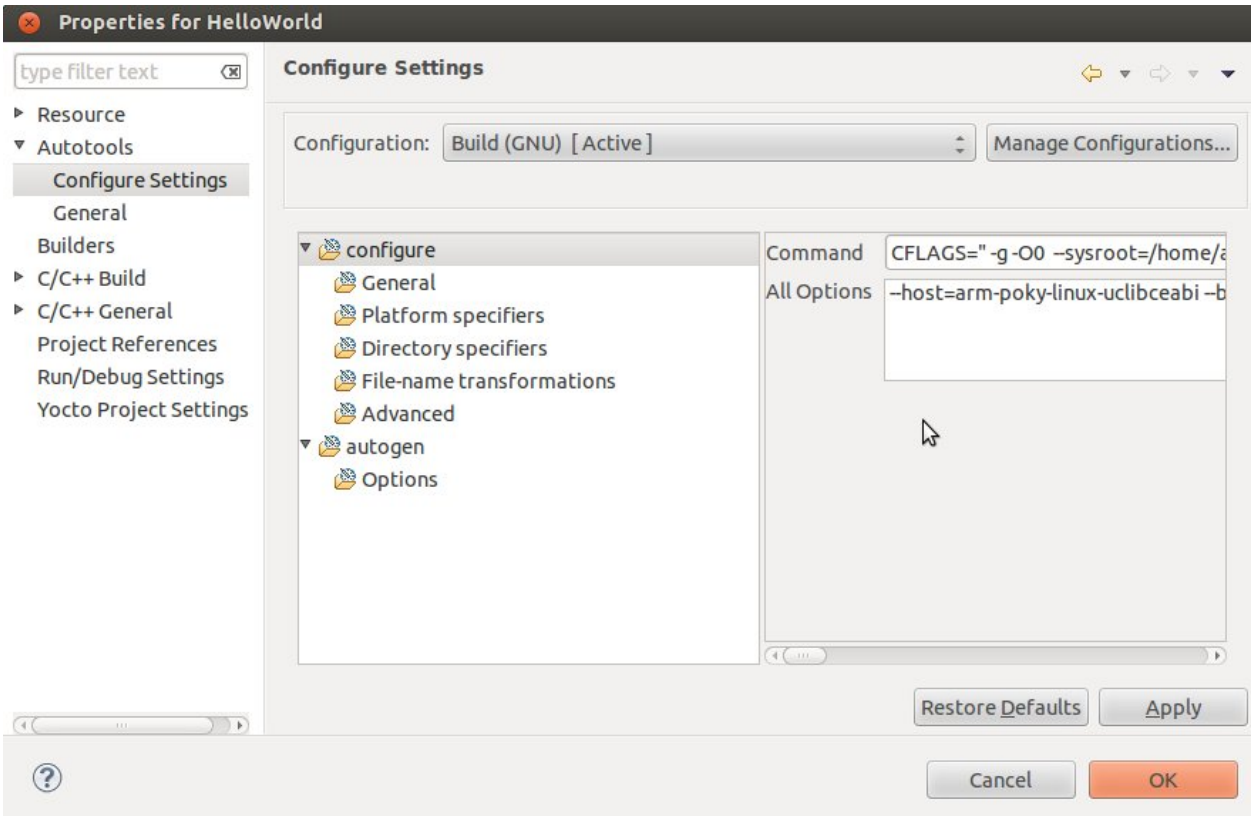


Building the Project

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/hachiko-tiny/sysroot* subdirectories.

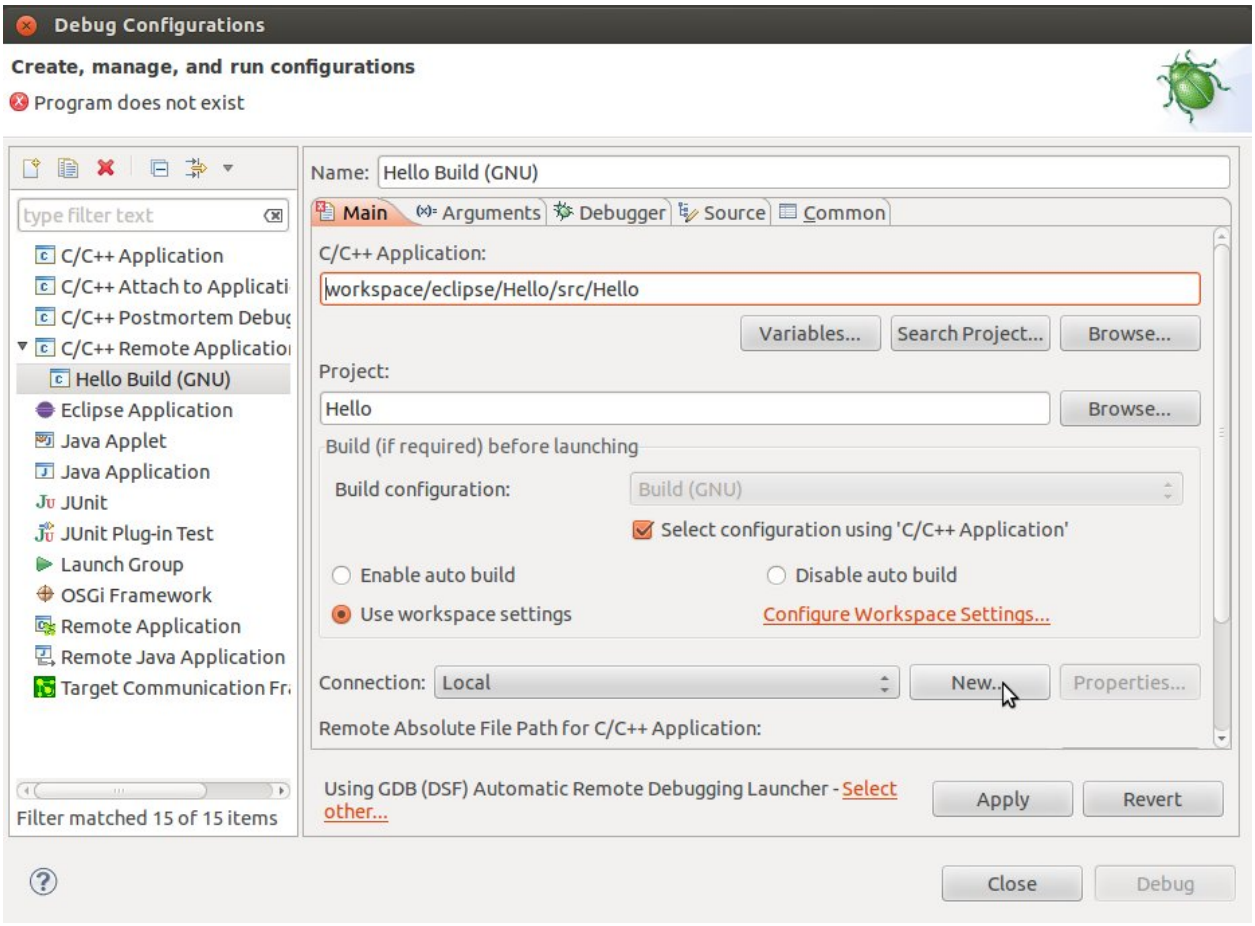


Deploying and Debugging the Application

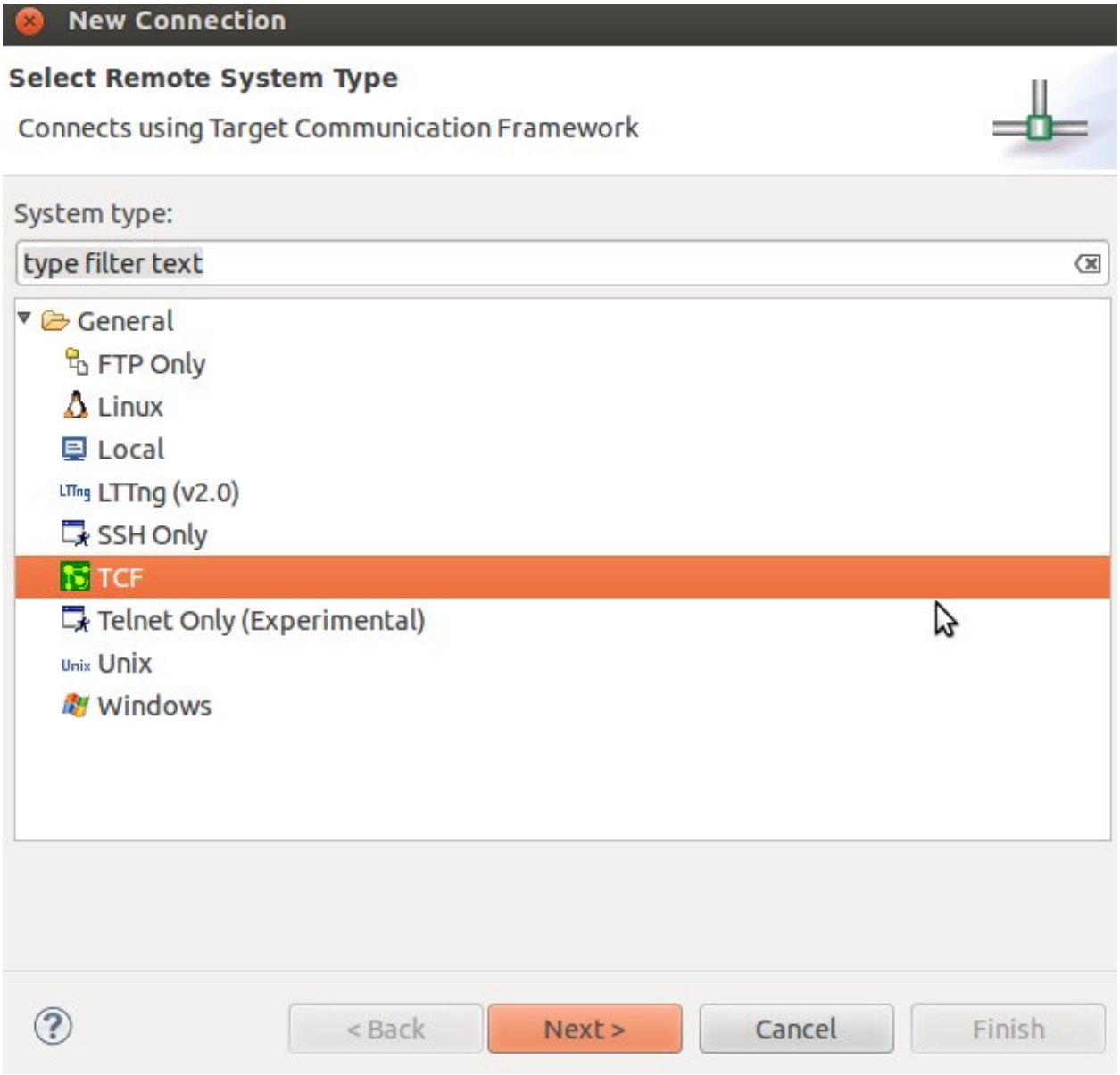
Connect Hachiko console to your PC and power-on the board. Once you built the project and the board is running the image, use minicom to run **tcf-agent** program in target board:

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

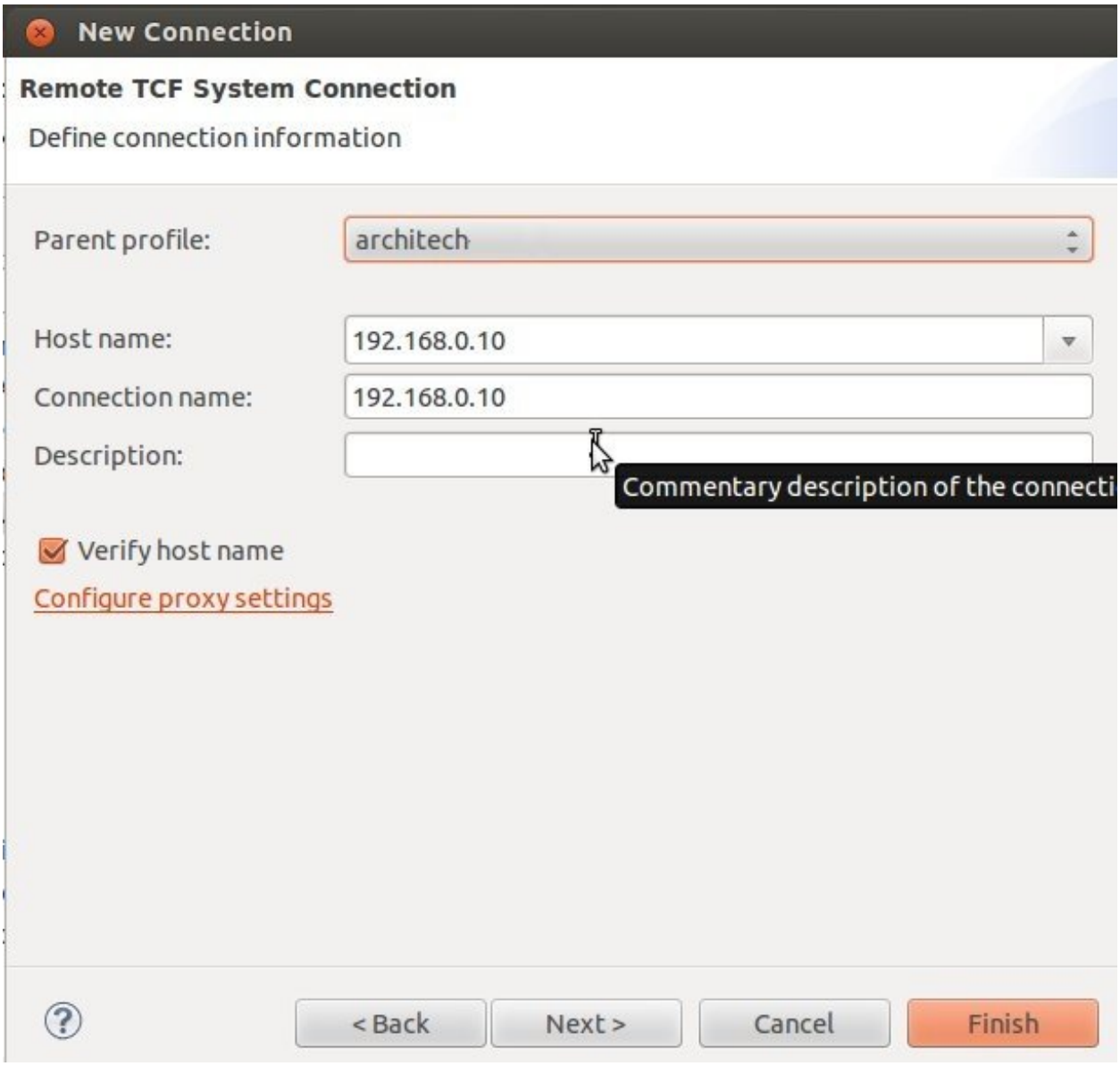
- Select Run→Debug Configurations...
- In the left area, expand *C/C++ Remote Application*.
- Locate your project and select it to bring up a new tabbed view in the *Debug Configurations* Dialog.



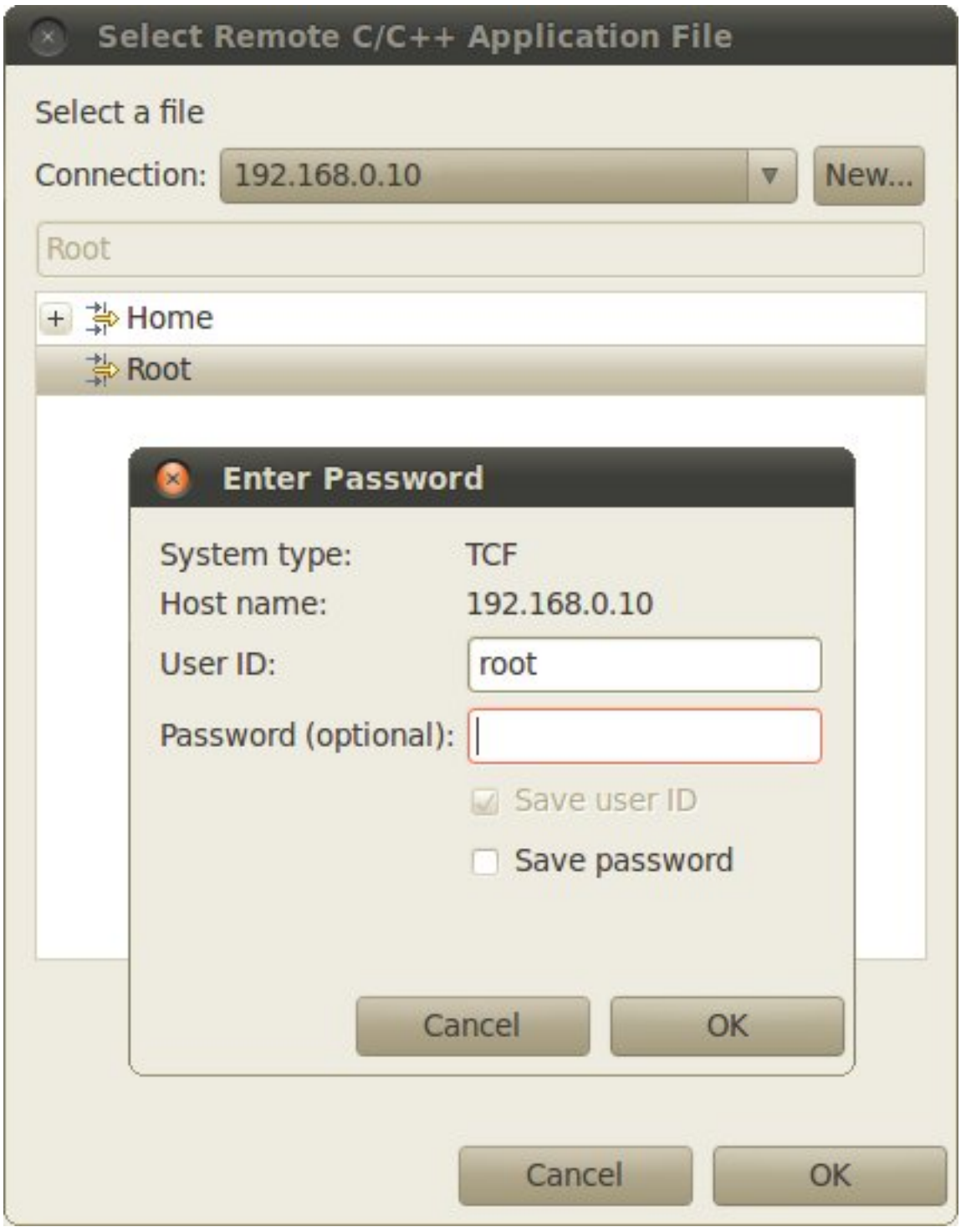
- Insert in *C/C++ Application* the filepath of your application binary on your host machine.
- Click on "New" button near the drop-down menu in the *Connection* field.
- Select *TCF* icon.



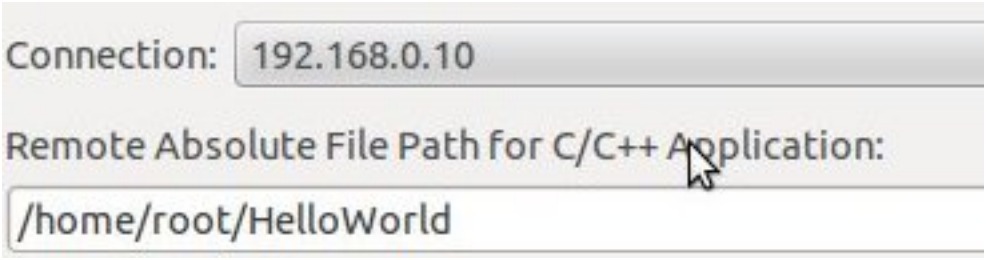
- Insert in *Host Name* and *Connection Name* fields the IP address of the target board. (e.g. 192.168.0.10)



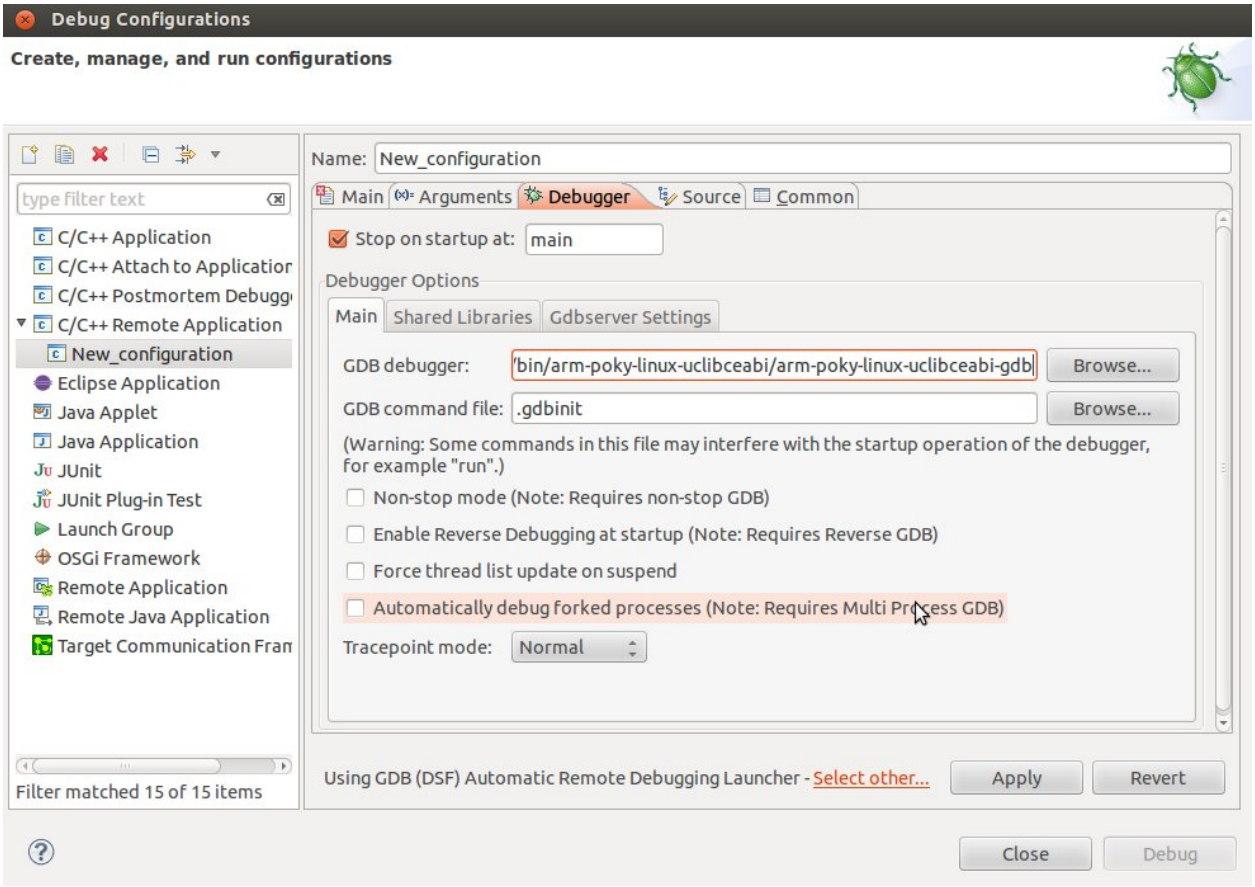
- Press *Finish*.
- Use the drop-down menu now in the *Connection* field and pick the IP Address you entered earlier.
- Enter the absolute path on the target into which you want to deploy the application. Use *Browse* button near *Remote Absolute File Path for C/C++Application:* field. No password is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

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.

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:

The board

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

Hardware

The hardware documentation of Hachiko can be found here:

http://downloads.architechboards.com/doc/Hachiko/download.html

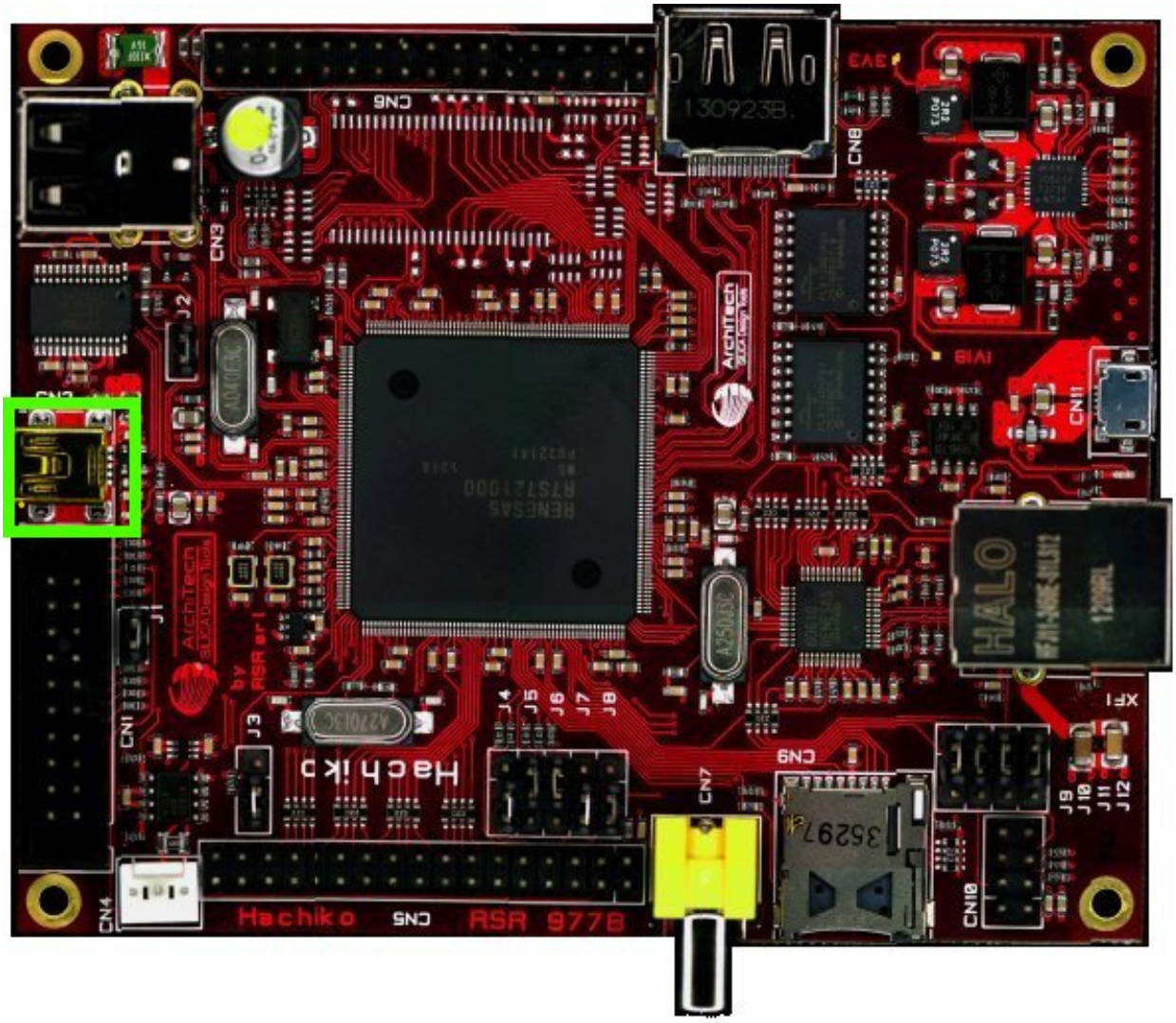
Power-On

Hachiko takes the power from the micro-USB connector *CN11* and/or the mini-USB connector *CN2*.

On connector *CN2* you can also have the serial console, so, during your daily development use, you would just connect your workstation to the board using a mini-USB to connector *CN2*. If you connect some power ungry device to the board, you can give more power to the board by connecting your workstation (or a smartphone-like battery charger for example) to the board by means of a micro-USB cable to board connector *CN11*.

Serial Console

On Hachiko there is the dedicated serial console connector **CN2**



which you can connect, by means of a mini-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 ttyUSBX 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

Host

sudo dmesg -c

---

2. connect the mini-USB cable to the board
3. display the kernel messages

Host

dmesg

3. read the output

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

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

Host

sudo minicom -ws

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

Host

sudo apt-get install minicom

then you can setup your port with these parameters:

```
+-----------------------------------------------------------------------+
| A -    Serial Device      : /dev/ttyUSB0                              |
| B - Lockfile Location     : /var/lock                                 |
| C -   Callin Program      :                                           |
| D -  Callout Program      :                                           |
| E -    Bps/Par/Bits       : 115200 8N1                                |
| F - Hardware Flow Control : No                                        |
| G - Software Flow Control : No                                        |
|                                                                       |
|    Change which setting?                                              |
+-----------------------------------------------------------------------+
        | Screen and keyboard      |
        | Save setup as dfl        |
        | Save setup as..          |
        | Exit                     |
        | Exit from Minicom        |
        +--------------------------+
```

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

Bootstrap

In the latest release only bootmode 3 is supported by Hachiko board. That is the board boots from the serial flash memory connected to the SPI multi I/O bus. To enable bootmode 3 the jumpers J4, J5 and J6 must be respectively in position 0, 1 and 1. When set for bootmode 3, the bootloader (U-Boot) must be flashed on the serial NOR at address 0x18000000 (SPI multi I/O bus space). Hachiko boards already have the latest version of U-Boot flashed on SPI NOR at factory time.

For more information:

- Hachiko schematics

- RZ/A1H User's Manual

Write to NOR

If for any reason U-Boot is corrupted or broken or simply the board refuses to boot again, the only way to recover the board is to try to flash a new U-Boot binary on the serial NOR. Flashing the NOR without U-Boot or Linux on the board being accessible is an easy task with the correct setup.
Renesas already has brief information about how to use a PARTNER-Jet or ULINK2 to write U-Boot in serial flash. Follows an extract from Renesas documentation.

Using PARTNER-Jet

- **Write to serial flash:**
    1. Execute PARTNER-Jet_ARM_V5_71b.exe
    2. Create a new project
    3. Copy "JETARM.CFG" and "usrflash_arm.MON" from flash directory to created project.
    4. Push power on button of PARTNER-JET
    5. Write U-boot image data for serial flash boot to 0x18000000.

Using ULINK2

To use ULINK2, DS-5 from ARM Ltd. and binary writing tool provided from Renesas are needed. As for the way to obtain these tools, please ask Renesas electronics sales representative. After the installation of DS5 and binary writing tool, please go to the next step.

- **Writing to serial flash:**
    1. Clear *RZ_A1H_spibsc_boot_init\Debug\RZ_A1H_spibsc_boot_init.bin* directory in DS-5 workspace.
    2. Rename U-boot image file for serial flash boot *u-boot.bin* as *VECTOR_TABLE* and put it on the above directory.
    3. Run DS-5.
    4. Connect ULINK2 to the board and poweron the board.
    5. Click script tag and double click *RZ_A1H_sflash_boot_user.ds*

The process using the ULINK2 is described here more in detail.

Software you need:

- *SPIBSCBoot.zip* file containing the DS-5 workspace, the script file and a launcher configuration (download link)
- DS-5 (preferred and tested version: *DS500-BN-00003-r5p0-15rel1*)

Setup of the environment and first run:

1. Install the DS-5 and use the activation code found here http://ds.arm.com/renesas/rza-starter-kit/ to enable the Renesas compiler when a valid licence is asked the first time DS-5 is started

2. Unzip the content of *SPIBSCBoot.zip*. Three more files should be available: *Connection_Only.launch*, *Workspace.zip* and *script.zip*. Unzip also *Workspace.zip* and *script.zip*
3. Start DS-5 and import into workspace all the three projects ([File] >> [Import Existing Projects into Workspace]) taking care of select [Copy projects into workspace] when importing
4. Import also the launch configuration (file *Connection_Only.launch*) ([File] >> [Import] >> [Launch configurations])
5. Copy the script directory into workspace directory and copy only the script *RZ_A1H_sflash_boot_user.ds* in the workspace directory (outside the script dir)
6. Open [Run] >> [Debug Configurations...] menu, then click on [DS-5 Debugger] >> [Connction_only]. In the [Connection] tab select [Renesas]/[RZ/A Series-R7S72100]/[Bare Metal Debug]/[Debug of Coretex-A9 via ULINK2] and press the [Browse] button to identify and select the attached debugger. Press Apply to save changes.
7. Rename *u-boot.bin* file as *VECTOR_TABLE* and copy it in *RZ_A1H_spibsc_boot_init\Debug\RZ_A1H_spibsc_boot_init.bin* directory inside the workspace
8. Connect the ULINK2 and power on the board. Select again [Run] >> [Debug Configurations...] menu, click on [DS-5 Debugger] >> [Connction_only] and on the Debug button.
9. When the DS-5 Dubug perspective is open, in the [Script] tab of DS-5 import the script *RZ_A1H_sflash_boot_user.ds*
10. Pause the microprocessor with the button in the toolbar and double-click on the script to run it
11. When the download script runs, the flash downloader is launched and a message is shown in the [Application Console]. Answer Y to the first question and N to the second one
12. When downloading finishes, it is possible to disconnect from target

For all the subsequent times it will be enough:

1. Open the DS-5
2. Switch to DS-5 Debug perspective
3. Select the launcher from the [Debug control] tab
4. Press the [Connect to Target] button in the toolbar
5. Double click on the script in the [Script] tab
6. Answer Y and N to the two questions in the [Application Console]
7. Disconnect from the target using the [Disconnect from the Target] button in the toolbar

For a more detailed description of the entire process, see:

- Renesas Application Note: Example of Downloading to Serial Flash Memory Using the Semihost Function of ARM DS-5"
- Renesas Applicaton Note: Example of Booting from Serial Flash Memory

---

Note:

* To change file to program in serial NOR it is enough to overwrite again the file named *VECTOR_TABLE* as seen at step (8)
* It is not possible to reprogram the NOR flash if the board successfully boots Linux. To be able to write the NOR using the DS-5 it is required that the board stops in U-Boot or at first stage bootloader (before U-Boot is loaded)

U-Boot

To check if the board is correctly programmed, connect the board to a terminal emulator on your computer. To see how this can be achieved please refer to section *serial_console_label*.

If everything is setup correctly you should be able to see the bootstrap process and the U-Boot output. In particular, as soon as the board is powered a countdown is started and displayed on the serial output. If a key is pressed before the countdown expires the autoboot stops, otherwise Linux is loaded from USB or SPI NOR.

On Hachiko board you can boot using the USB or the serial NOR. During the boot process, if U-Boot detects a correct kernel and rootfs on the USB drive it will boot from this USB device, otherwise it will switch to SPI NOR. In case no correct linux kernel is detected, the boot stops in the U-Boot console.

For a brief documentation about U-Boot:

- Renesas U-Boot documentation:

Boot from USB

Booting from USB requires that an USB pen drive is prepared with all the files needed for booting Linux and that it is correctly partitioned.

Important: The only USB port that it is possible to use for booting is the USB port at the bottom of the USB connector.

USB partitioning

The USB pen driver is required to have one single *EXT2* partition with a start sector of the partition below the 63rd sector. It is possible to use tools as *fdisk* or *cfdisk* to partition the USB drive.

Before use these commands you need unmount the device:

After that you can use the following commands:

As alternative it is possible to use the sfdisk tools to have the partition correctly aligned to the first sector:

To format the partition it is enough:

USB content

When booting from USB, U-Boot expects to find a valid single *EXT2* partition in the USB pen drive containing the rootfs. Moreover U-Boot needs to find in */boot* directory a valid kernel image and a valid *DTB* file respectively named *uImage* and *rza1-hachiko.dtb*.

When using Yocto to generate the rootfs we need to extract the compressed rootfs found in

in the partition on the USB and copy the kernel in */boot/uImage* and *DTB* file in */boot/rza1-hachiko.dtb*.

Briefly, to have a bootable USB stick after having compiled an image with Yocto:

1. Create one *EXT2* partition in the USB stick
2. Extract the content of your *.tar.bz2* rootfs tarball file in the *EXT2* partition
3. Copy *uImage* to */boot*

4. Copy *uImage-rza1-hachiko.dtb* to */boot* and rename it as *rza1-hachiko.dtb*

At this point it is possible to boot Linux by inserting the USB pen drive in the correct USB port and power on the board.

Boot from NOR

When no USB device is attached or the kernel image is not valid, U-Boot tries to boot from SPI NOR. In Hachiko board the NOR is required to contain all the needed files in the first serial flash memory on channel 0.

Note: A valid NOR Linux image is programmed at factory time in Hachiko NOR, so it is possible to start using Hachiko board right away.

NOR Partitioning

The serial flash memory is divided into 5 partitions according to the following scheme (the base address is 0x18000000):

```
0x18000000-0x18080000 spibsc0_loader  (offset: 0x00000000)
0x18080000-0x180c0000 spibsc0_bootenv (offset: 0x00080000)
0x180c0000-0x184c0000 spibsc0_kernel  (offset: 0x00c0000)
0x184c0000-0x18500000 spibsc0_dtb     (offset: 0x004c0000)
0x18500000-0x1c000000 spibsc0_rootfs  (offset: 0x00500000)
```

```
spibsc0_loader: contains u-boot (u-boot.bin)
spibsc0_bootenv: contains u-boot environment
spibsc0_kernel: contains the Linux kernel (uImage)
spibsc0_dtb: contains the DTB file (rza1-hachiko.dtb)
spibsc0_rootfs: contains the rootfs
```

NOR content

To write in NOR and replace or update the content of the NOR partitions you can go through U-Boot or Linux. It is strongly recommended to use Linux for writing new data in NOR partitions, especially when no external SDRAM is available.

Using U-Boot [not recommended]

Using U-Boot for writing or updating data in SPI NOR is not advisable especially when no external SDRAM is available.

Warning: The operation is prone to failure, use it at your own risk.

The process of writing data in serial NOR using U-Boot goes through 3 main steps: 1) load the file to write in a temporary RAM location, 2) erase data on the NOR partition and 3) write the new data.

1. Assuming you have the file on the USB pen drive you have to load it in RAM using the following commands:

The output from the command is also the size of the file loaded, info useful for step (3).

RAM ranges:

```
0x20000000 - 0x20A00000
```

Please, note that while *spibsc0_loader*, *spibsc0_kernel*, and *spibsc0_dtb* partitions of the flash memory contain raw data respectively for *u-boot.bin*, *uImage* and *rza1-hachiko.dtb*, whereas partition *spibsc0_rootfs* contains raw data for the filesystem that, in our case, is a *JFFS2* image. That is, when writing a new rootfs in *spibsc0_rootfs* partition it is needed to use the image file *.jffs2* generated by Yocto.

2. To erase data on the NOR partition;

where $OFFSET is the partition offset and $SIZE its size in bytes.

3. To write new data:

where $RAM_ADDR is the temporary RAM location holding our file (tipically 0x20000000), $OFFSET is the partition offset and $SIZE is the file size in bytes as obtained by the output of the command ext2load in step (1).

For more informations about flash managing with U-Boot refer to:

- Renesas U-Boot documentation

Using Linux

To use linux for writing or updating data on the serial NOR you are going to need **MTD utils**. It is possible to compile a small image containing the MTD utils with Yocto by means, for example, of image *core-image-minimal-mtdutils* that can be generated by *Bitbake* with this command line:

Warning:

If you are using the **Hachiko-tiny** which is the board without external RAM, in order to compile successfully this image you have to abilitate the IPv6 in the uClibc. To do so open the file:
/home/architech/architech_sdk/architech/hachiko-tiny/yocto/meta-hachiko/conf/distro/tiny-linux-uclibc.conf
and at the line *DISTRO_FEATURES_NET = "ipv4"* change it with **DISTRO_FEATURES_NET = "ipv4 ipv6"**

In Linux, the process is made easier by the MTD framework that remap each NOR partition to a different device file. In particular:

Again the process goes through 2 steps: (1) erasing the content of the serial NOR partition and (2) write the new data.

1. To erase the content of the partition the tool flash_erase can be used. For raw files as *u-boot.bin*, *uImage* or *rza1-hachiko.dtb*, the tool can be used as follow:

This command completely erases the content of the partition. For the root file system the command is slightly different, since being *spibsc0_rootfs* a *JFFS2* partition, it requires proper formatting, so for mtd4 device you need to run this command:

2. To write the new data on the serial NOR the tool flashcp is used. Again for raw file the simple syntax is:

For rootfs we have two different ways to write data in *spibsc0_rootfs* partition:

1. Using the image file *.jffs2* generated by Yocto
2. If you wish to use the image file *.tar.bz2* instead, you need to mount the partition and decompress the file content in place.

For more information on how to manage flash storage with Linux:

http://free-electrons.com/blog/managing-flash-storage-with-linux/

Network

The network PHY is provided by Renesas's chip 60610 (*U6*). Within Linux, you can see the network interface as **eth0**.

If you want that configuration to be brought up at boot you can add a few line in file */etc/network/interfaces*, for example, if you want *eth0* to have a fixed ip address (say 192.168.0.10) and MAC address of value 1e:ed:19:27:1a:b6 you could add the following lines:

FAQ

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 passowrd 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 12.04 32bit comes with no password defined for **roor** 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.

Hachiko

How to switch from hachiko to hachiko/SDRAM and viceversa

Switching from hachiko to hachiko/SDRAM or viceversa (adding or removing the external SDRAM) is a delicate operation that involves flashing a new U-Boot and using a new Kernel. To flash a new U-Boot it is needed to follow Section *Write to NOR*, writing the new U-Boot from the U-Boot itself or from Linux. After the first boot with the new U-Boot it is always suggested to give the following commands to reset the U-Boot environment stored in the second partition of the serial NOR:

and reboot the board.

Also the Linux kernel must be recompiled for the new configuration and substituted to the old one in the USB pend drive or in the NOR.

Bootargs and Framebuffer

Hachiko kernel uses two parameters passed by U-Boot at boot time to enable / disable / configure the framebuffer. These two parameters are vdc5fb0 for channel 0 and vdc5fb1 for channel 1 of the video display controller.

The default value for these parameters is:

hachiko/SDRAM:

Board

vdc5fb0=3 vdc5fb1=4

Hachiko:

Board

vdc5fb0=0 vdc5fb1=0

The meaning of the values are here reported:

Channel 0 (**vdc5fb0**):

- (**0**) unuse
- (**1**) / (**2**) reserved for RZARSK dev board
- (**3**) LCD parallel out enabled

Channel 1 (**vdc5fb1**):

- (**0**) unuse
- (**1**) / (**2**) / (**3**) reserved for RZARSK dev board
- (**4**) LVDS output

It is possible to modify the default setting in U-Boot with the command:

Board

env set fbparam vdc5fb0=$B vdc5fb1=$A

---

with $A and $B the new set of parameters. To make the configuration permanent:

Board

saveenv

Note: For the hachiko board without external SDRAM the usage of framebuffer can result in instability if not used with care.

Change Framebuffer Resolution

The default kernel shipped has the following default resolutions:

LCD parallel out:

```
480x272
```

LVDS output:

```
800x480
```

to change them the file arch/arm/mach-shmobile/rskrza1-vdc5fb.c must be modified. Specifically the two structures containing the screen timings are:

Channel 0 (parallel LCD):

```
struct fb_videomode videomode_wqvga_lcd_kit
```

Channel 1 (LVDS):

```
struct fb_videomode videomode_lvds
```

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.

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**.

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

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.

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.

Index

D

Debug, 76

P

Project, 72