Tibidabo Documentation

Release C

Architech

Mar 16, 2017

Contents

Version 1.1.0C

Copyright (C)2016 Avnet Silica company

Date 19/12/2014

Welcome to **Tibidabo** documentation!

You can find the previous documentation: Here
Have you just received your Tibidabo 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 Tibidabo 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 powerful board comes with this beautiful box

freescale
ArchiTech
Design. Build. Bring it all together.
ArchiTech Tibidabo
Freescale i.MX 6Quad High End
Multimedia Development Kit
www.architechboards.org/tibidabo
CE RoHS

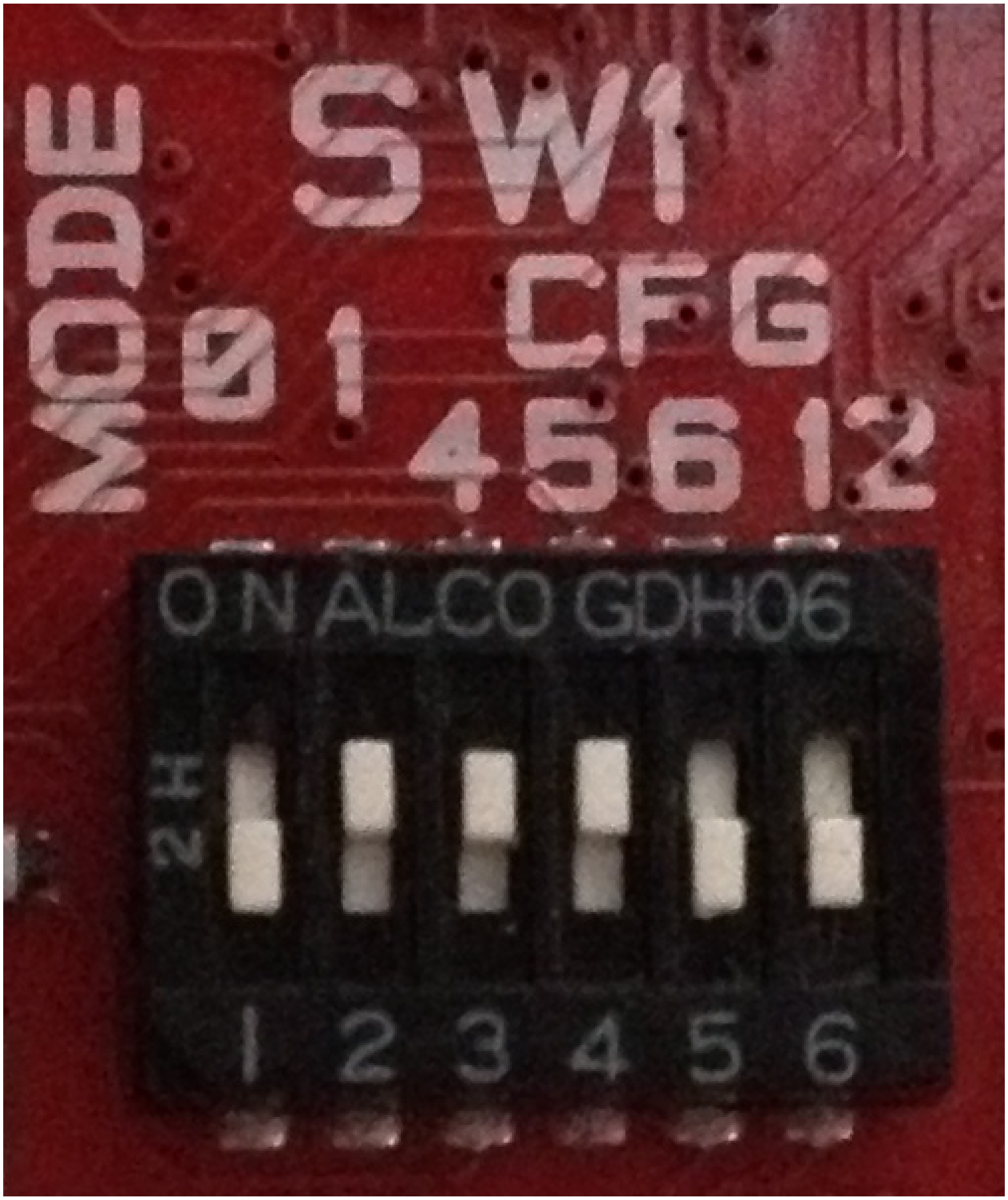
Tibidabo feeds its horses by means of an external power supply, which is included in the package and has several socket adapters.

ArchiTech
SILICA Design Tools
ArchiTech Tibidabo
An iMX 6Quad on a high end multimedia Linux Board
Quickstart guide
Thank you for buying Silica ArchiTech Tibidabo demo board.
Requirements and Quickstart:
Silica ArchiTech Tibidabo development board website
http://www.silica.com/product/architech-tibidabo.html

The SPI NOR on the board has been programmed to let Tibidabo boot a *core-image-minimal* image generated with *Yocto*.

What are we waiting for? Lets boot the board!

1. First of all, make sure SW1 has this configuration



2. Connect the HDMI connector (**CN8**) to your monitor/television by means of an HDMI cable
3. Connect a USB keyboard to the board (connector **CN18**)
4. Take the socket adapter compatible with your country, plug it in the power adapter. When in position, you should hear a slight *click*
5. Power on the board connecting the external power adapter to Tibidabo connector **CN19**

6. The login is **root**

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, Tibidabo 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

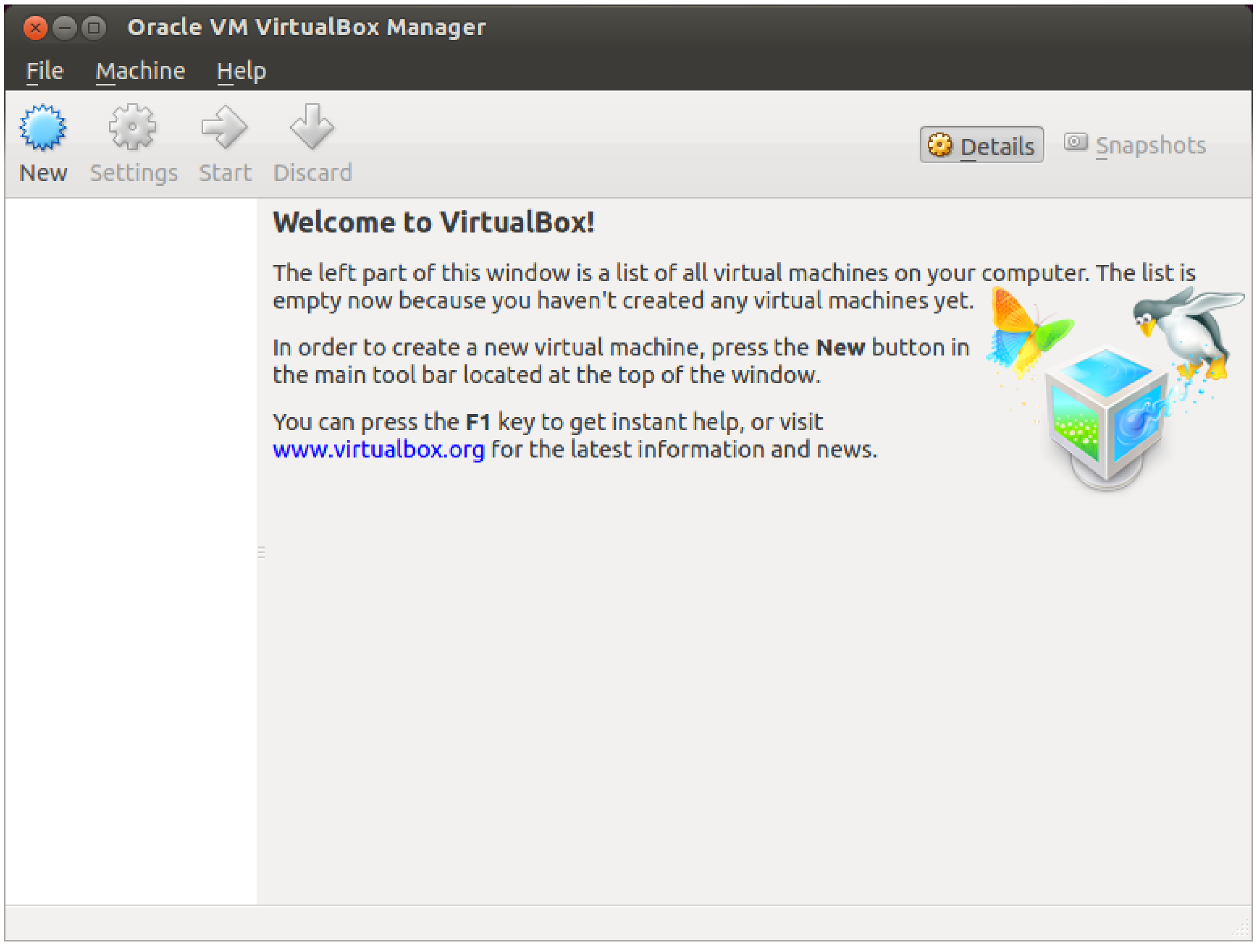
Download the version that suits your host operating system. You need to download and install the **Extension Pack** as well.

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



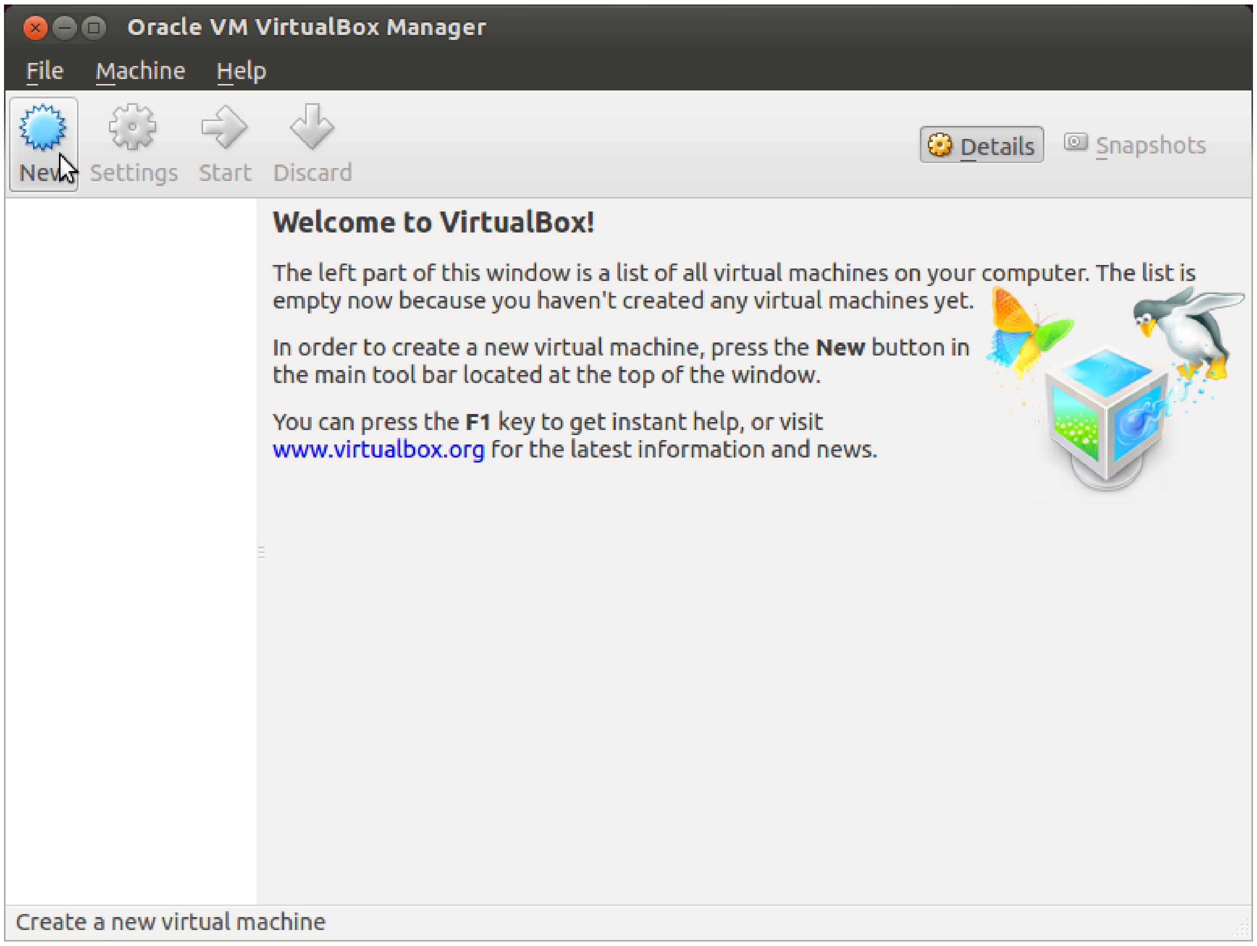
Welcome to VirtualBox!

The left part of this window is a list of all virtual machines on your computer. The list is empty now because you haven't created any virtual machines yet.

In order to create a new virtual machine, press the **New** button in the main tool bar located at the top of the window.

You can press the **F1** key to get instant help, or visit www.virtualbox.org for the latest information and news.

2. Click on *New* button



Welcome to VirtualBox!

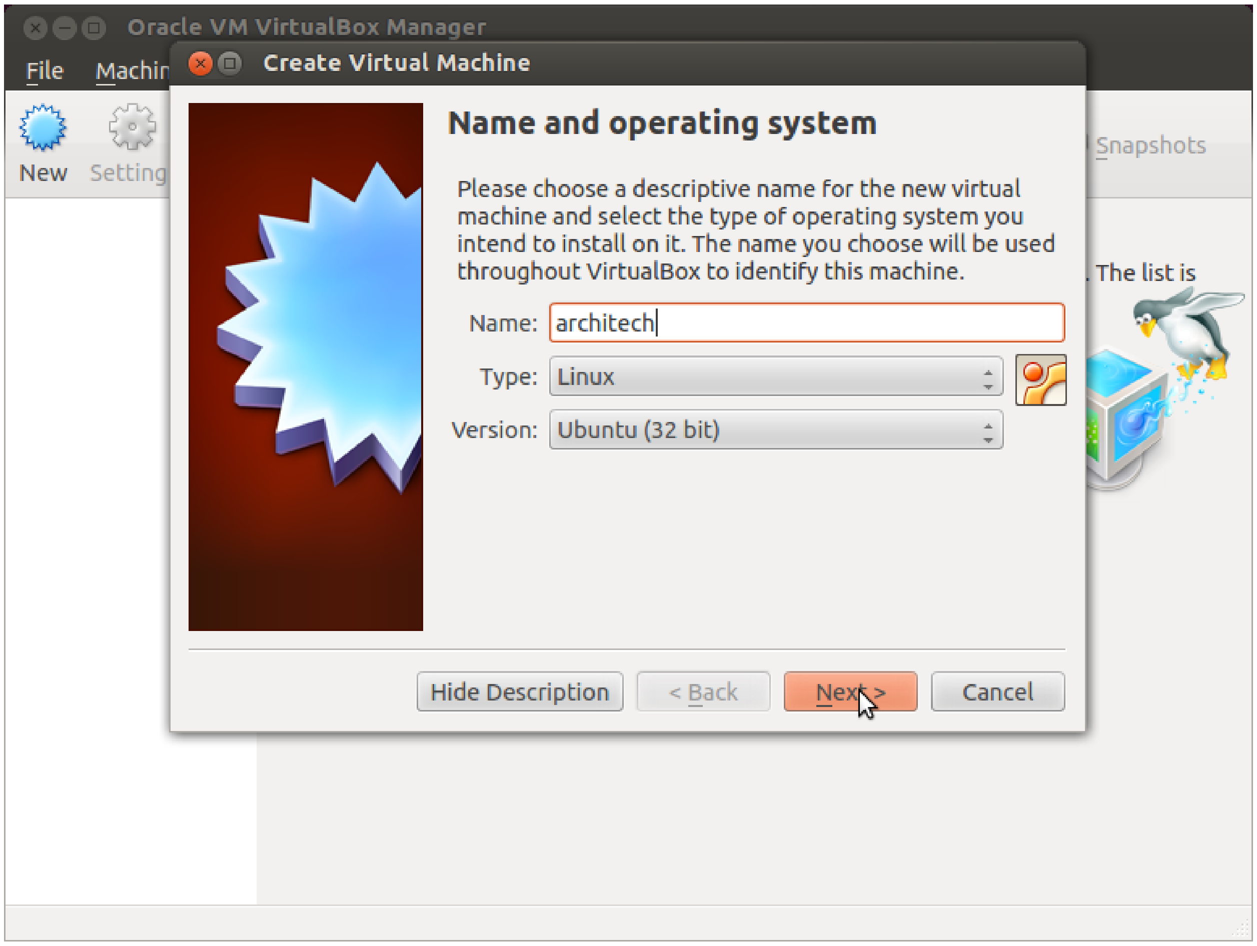
The left part of this window is a list of all virtual machines on your computer. The list is empty now because you haven't created any virtual machines yet.

In order to create a new virtual machine, press the **New** button in the main tool bar located at the top of the window.

You can press the **F1** key to get instant help, or visit www.virtualbox.org for the latest information and news.

Create a new virtual machine

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



Name and operating system

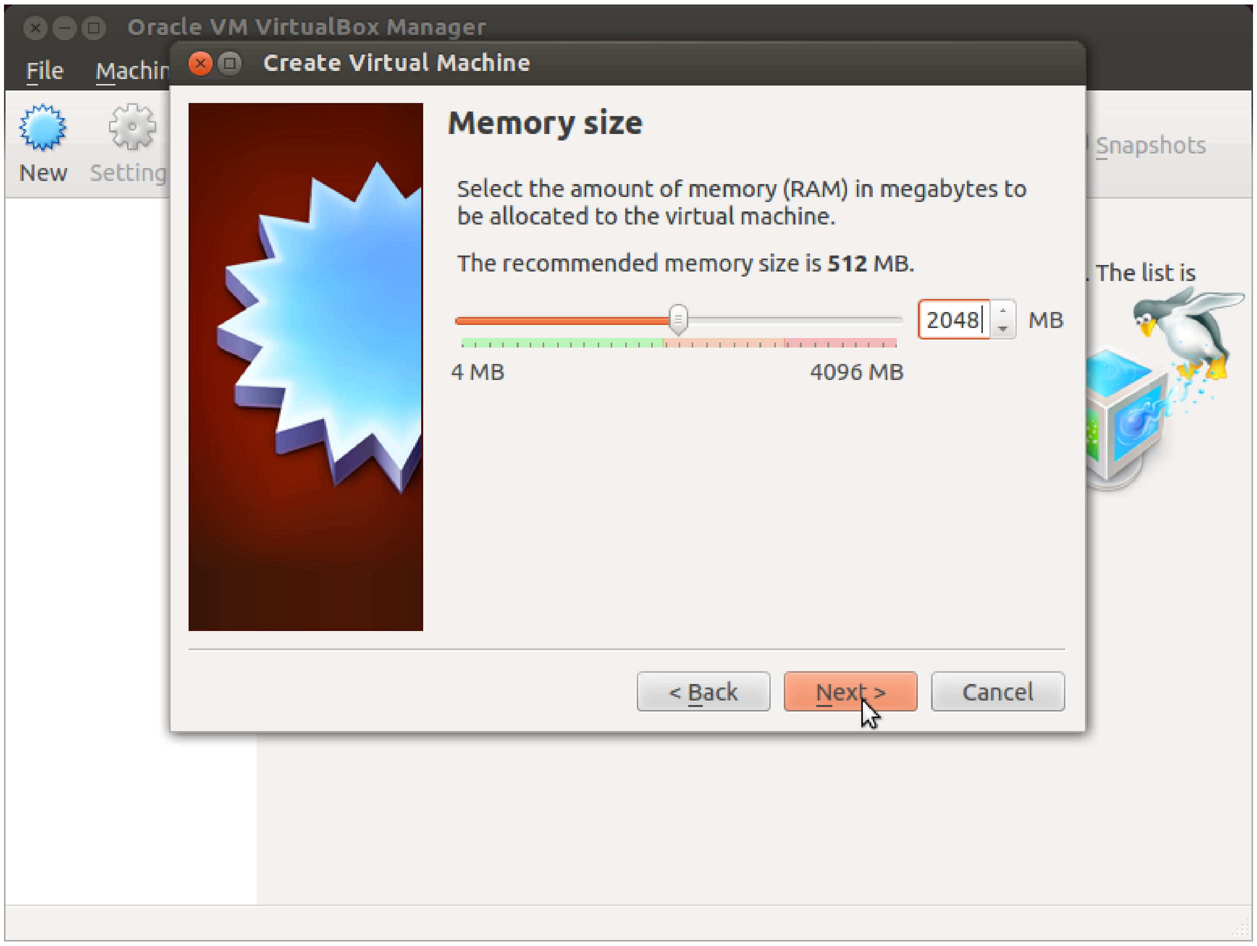
Please choose a descriptive name for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name: architech

Type: Linux

Version: Ubuntu (32 bit)

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



Memory size

Select the amount of memory (RAM) in megabytes to be allocated to the virtual machine.

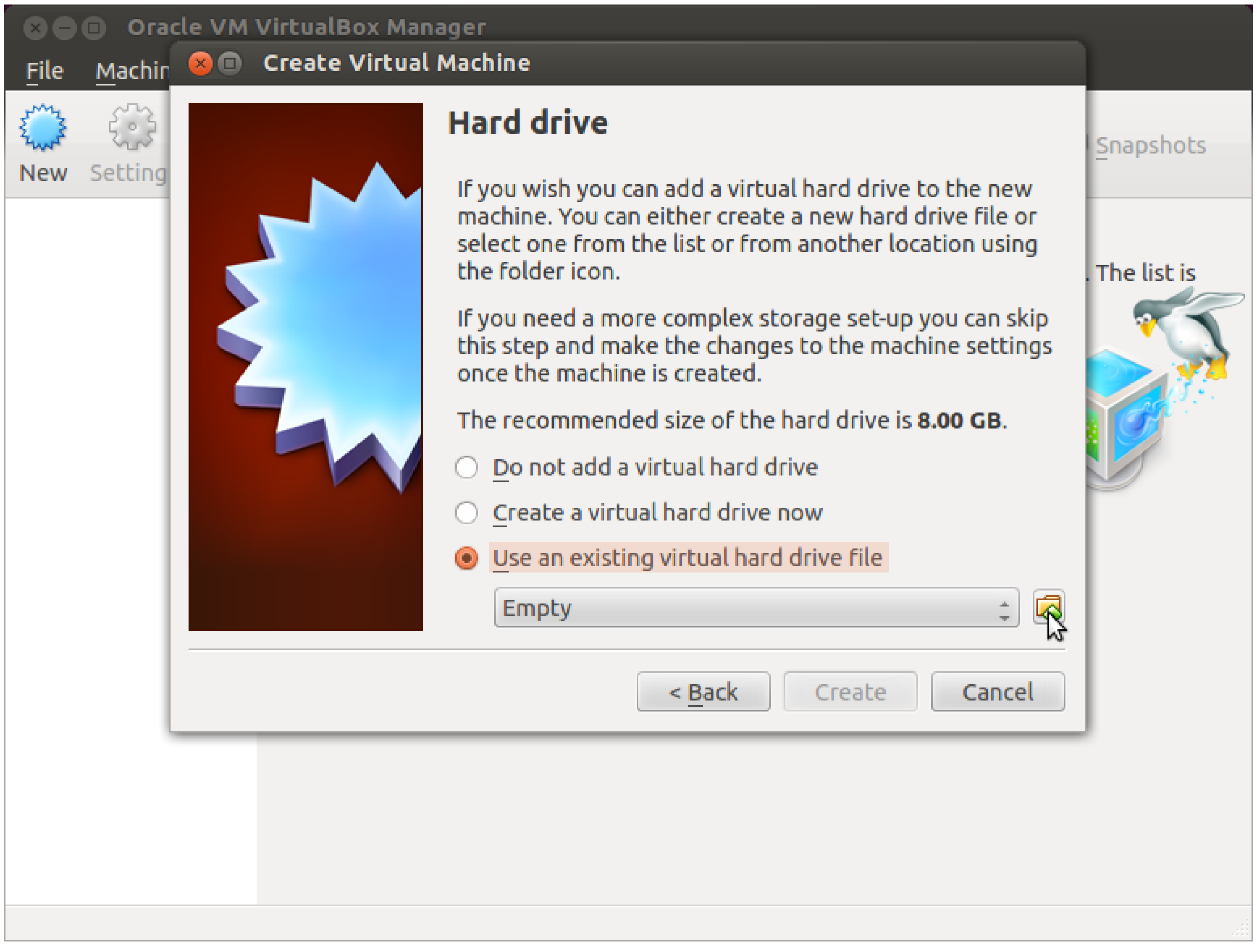
The recommended memory size is **512** MB.

2048 MB

4 MB 4096 MB

. The list is

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



Hard drive

If you wish you can add a virtual hard drive to the new machine. You can either create a new hard drive file or select one from the list or from another location using the folder icon.

If you need a more complex storage set-up you can skip this step and make the changes to the machine settings once the machine is created.

The recommended size of the hard drive is **8.00 GB**.

- Do not add a virtual hard drive
- Create a virtual hard drive now
- [✓ Use an existing virtual hard drive file]

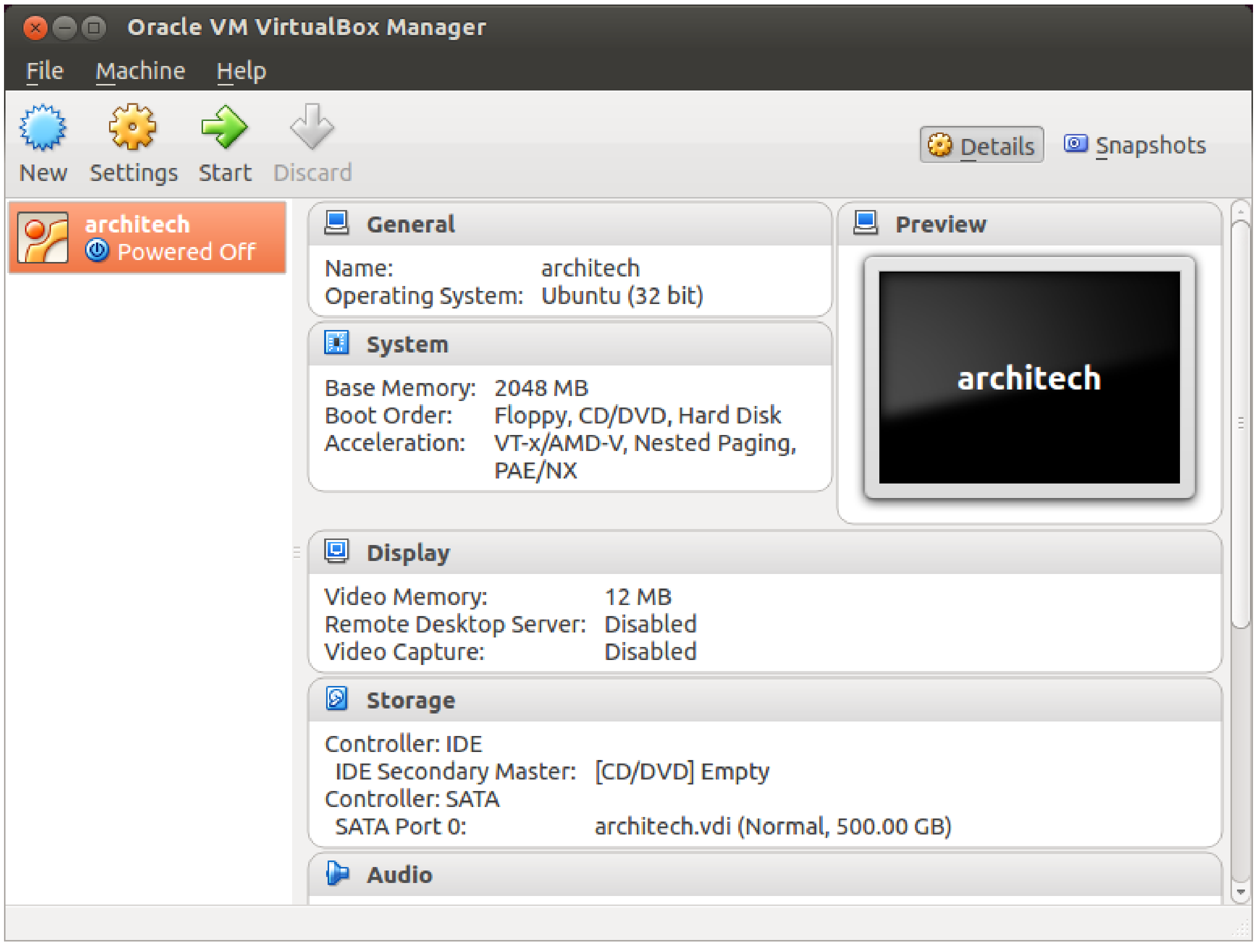
Empty

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



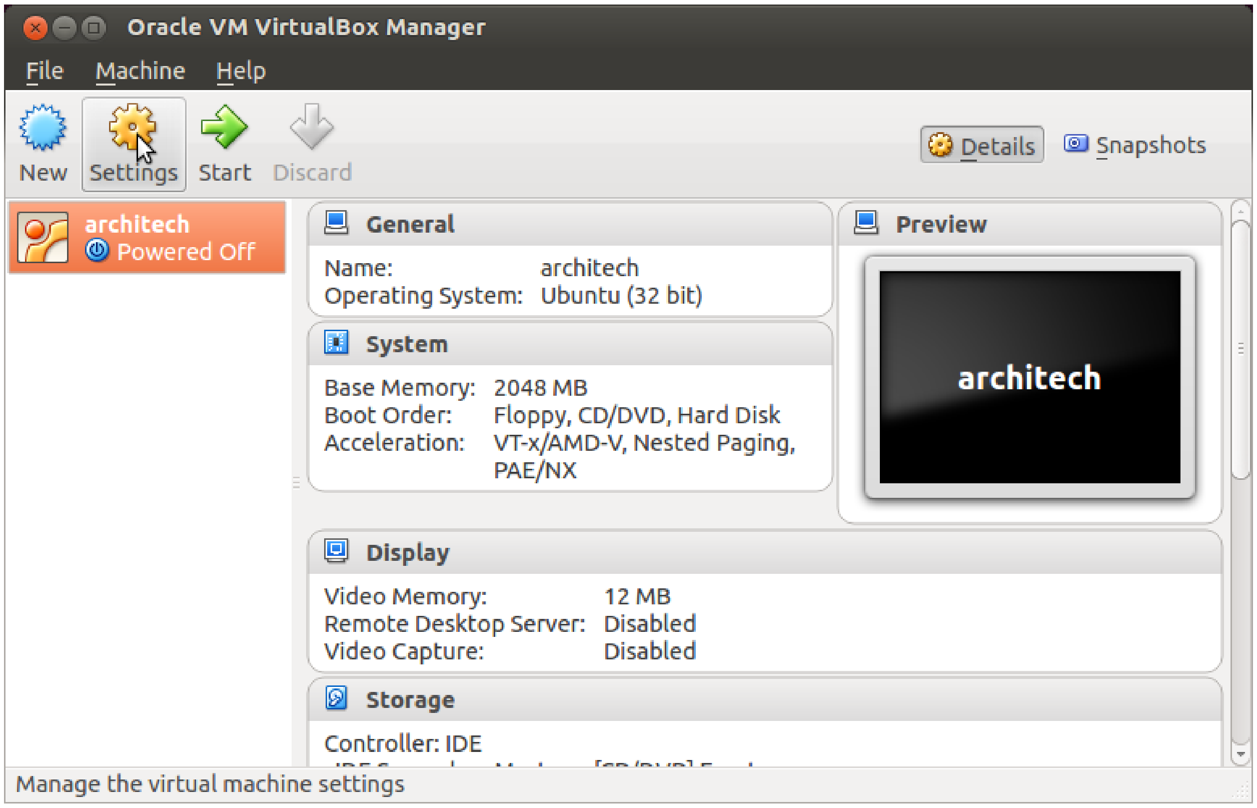
Name: architech
Operating System: Ubuntu (32 bit)

Base Memory: 2048 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX

Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Controller: IDE
IDE Secondary Master: [CD/DVD] Empty
Controller: SATA
SATA Port 0: architech.vdi (Normal, 500.00 GB)

2. Click on *Settings*



General
Name: architech
Operating System: Ubuntu (32 bit)

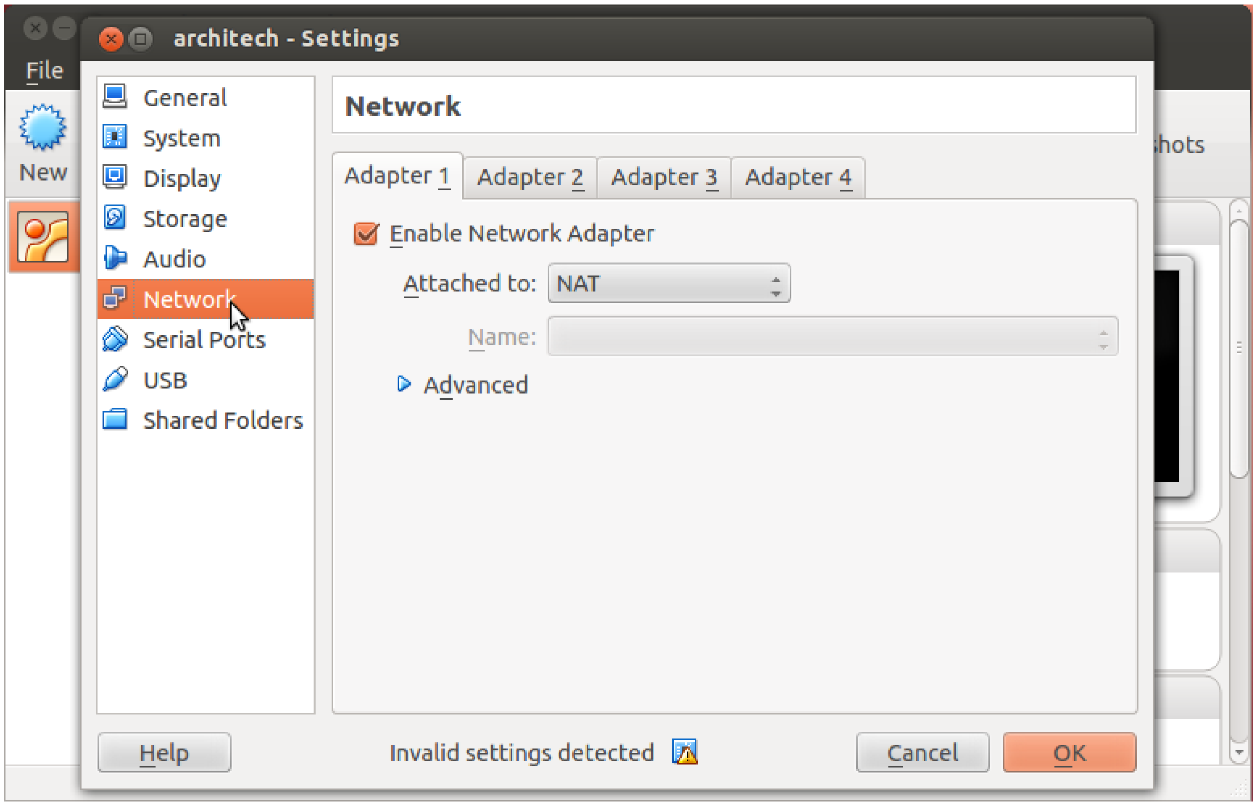
System
Base Memory: 2048 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX

Display
Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Storage
Controller: IDE

Manage the virtual machine settings

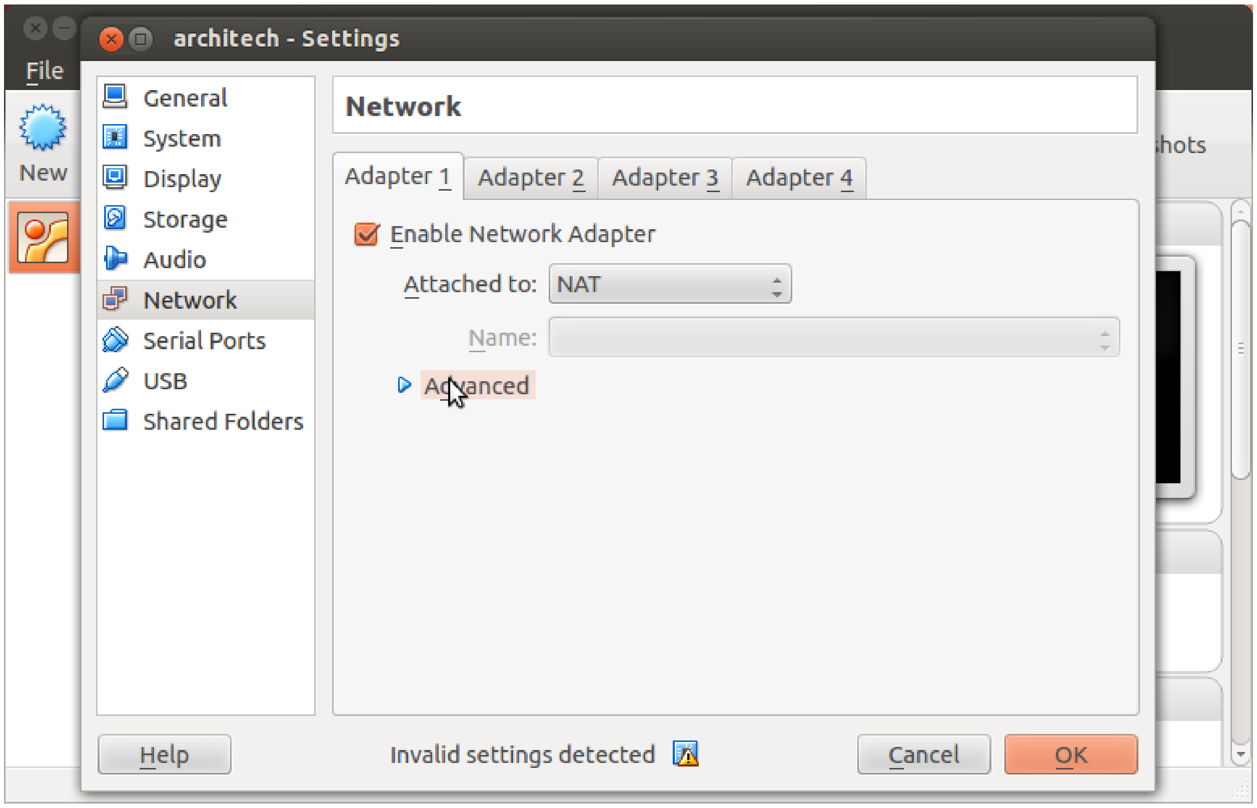
3. Select *Network*



☑ Enable Network Adapter
Attached to: NAT
Name:

Invalid settings detected

4. Expand *Advanced* of *Adapter 1*



Network

Enable Network Adapter: ✓

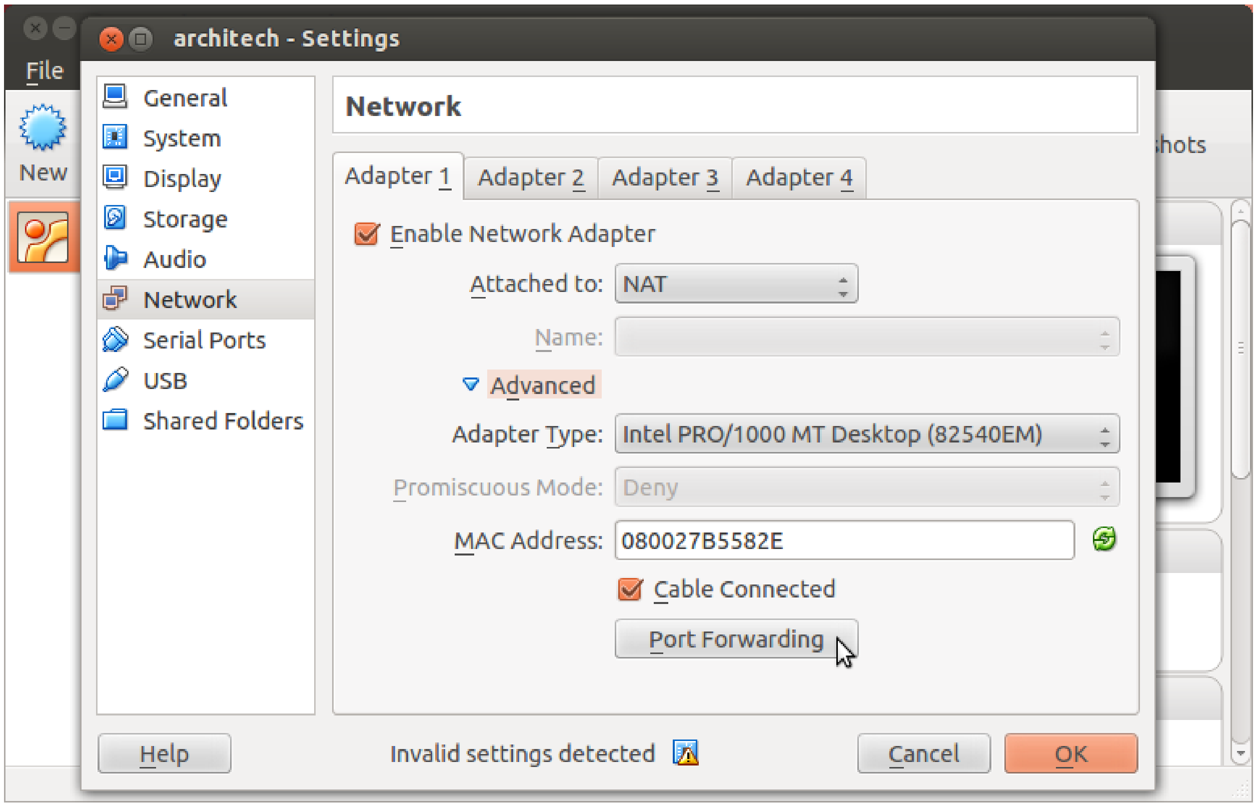
Attached to: NAT

Name:

Advanced

Invalid settings detected

5. Click on *Port Forwarding*



Network

Enable Network Adapter: ✓

Attached to: NAT

Name:

Advanced

Adapter Type: Intel PRO/1000 MT Desktop (82540EM)

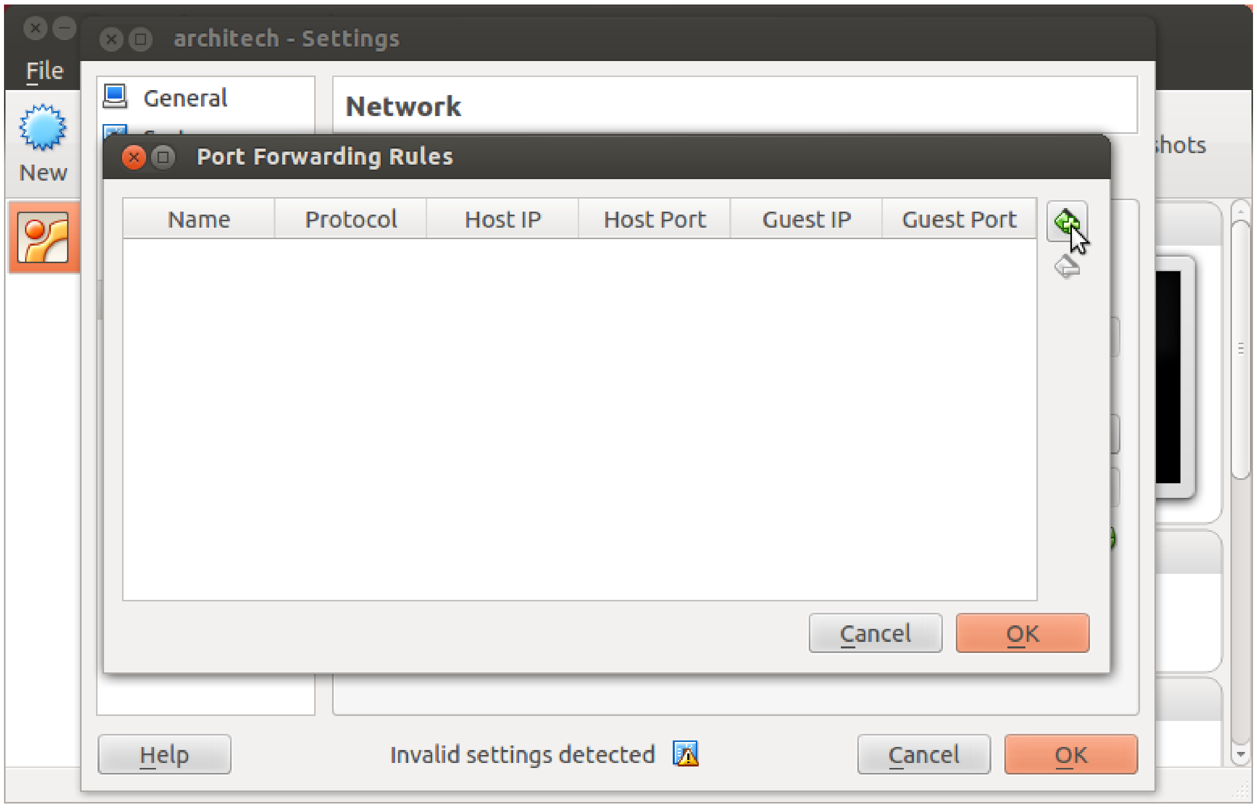
Promiscuous Mode: Deny

MAC Address: 080027B5582E

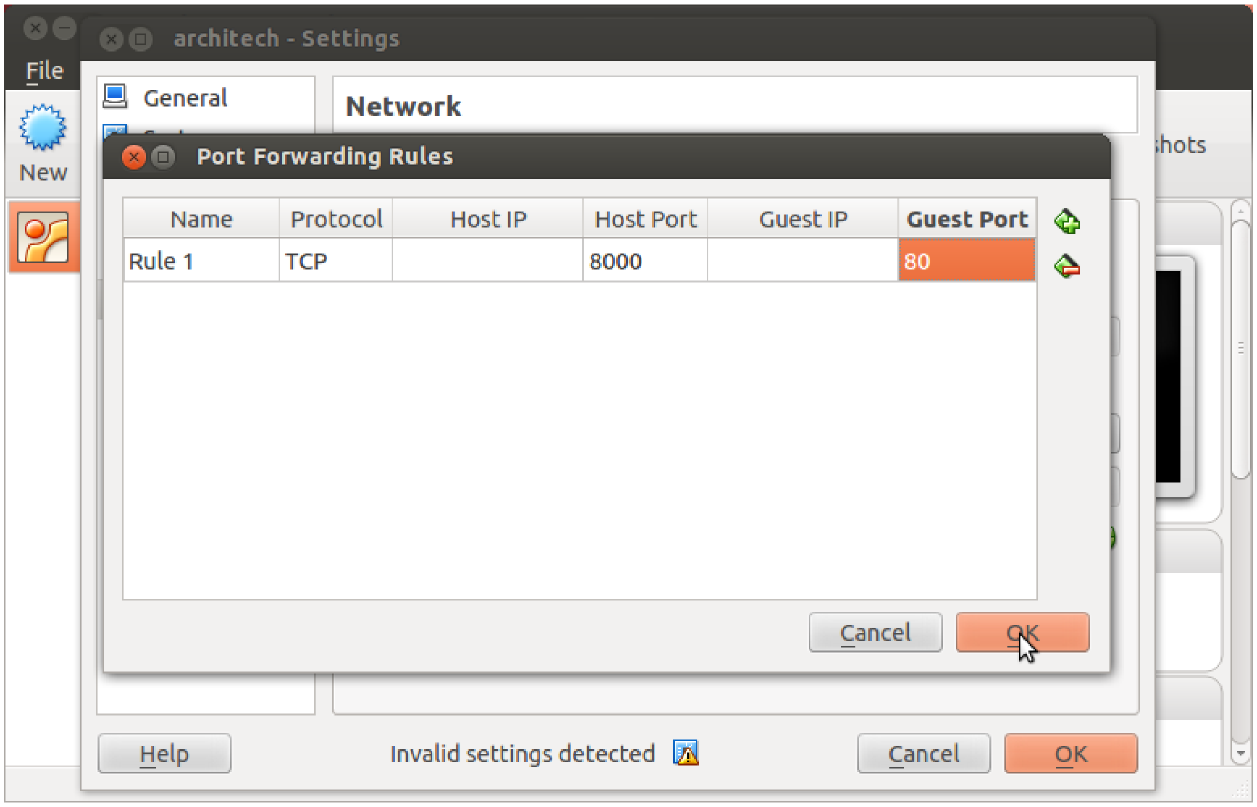
Cable Connected: ✓

Invalid settings detected

6. Add a new *rule*



7. Configure the *rule*



Name	Protocol	Host IP	Host Port	Guest IP	Guest Port
Rule 1	TCP		8000		80

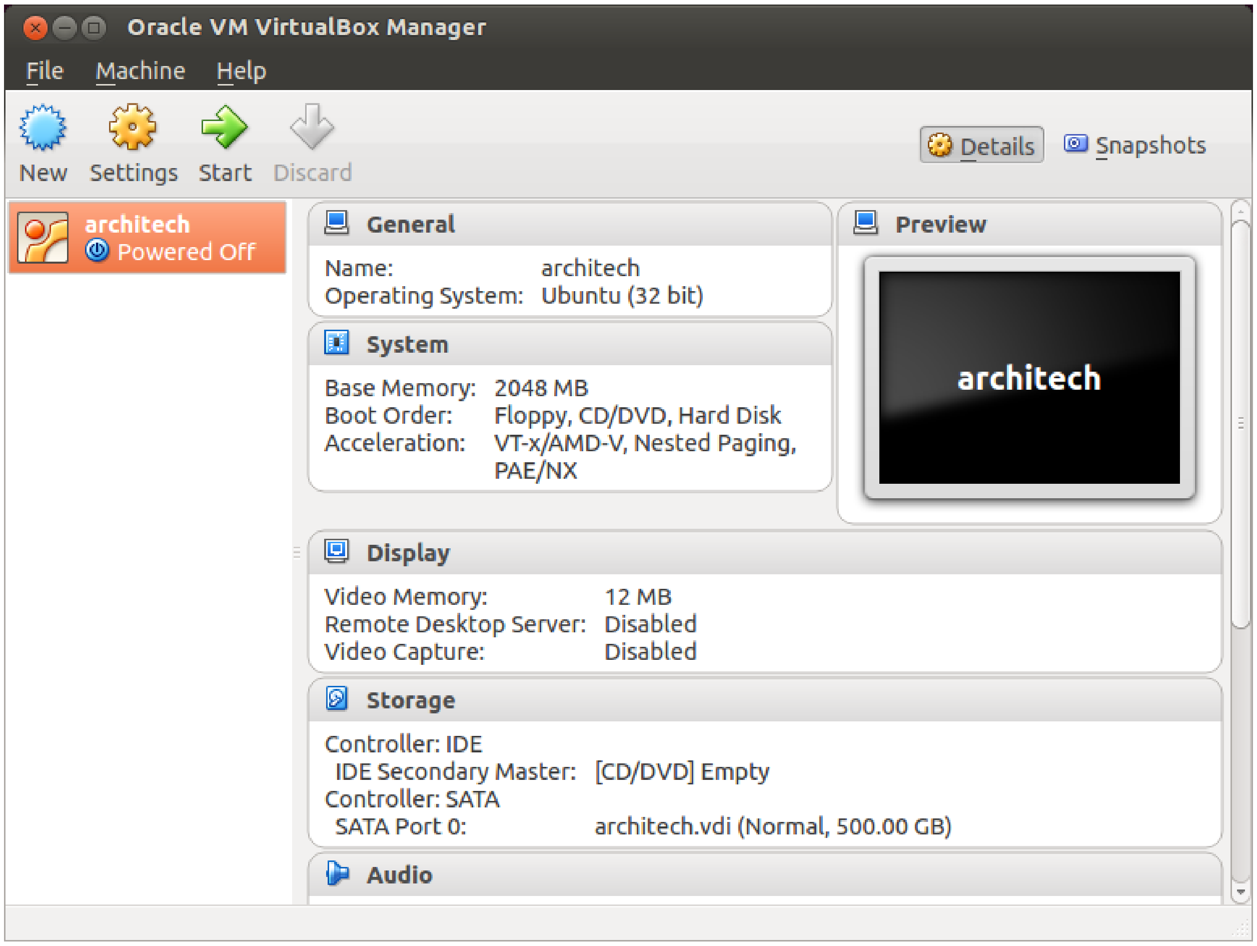
8. Click on *Ok*

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



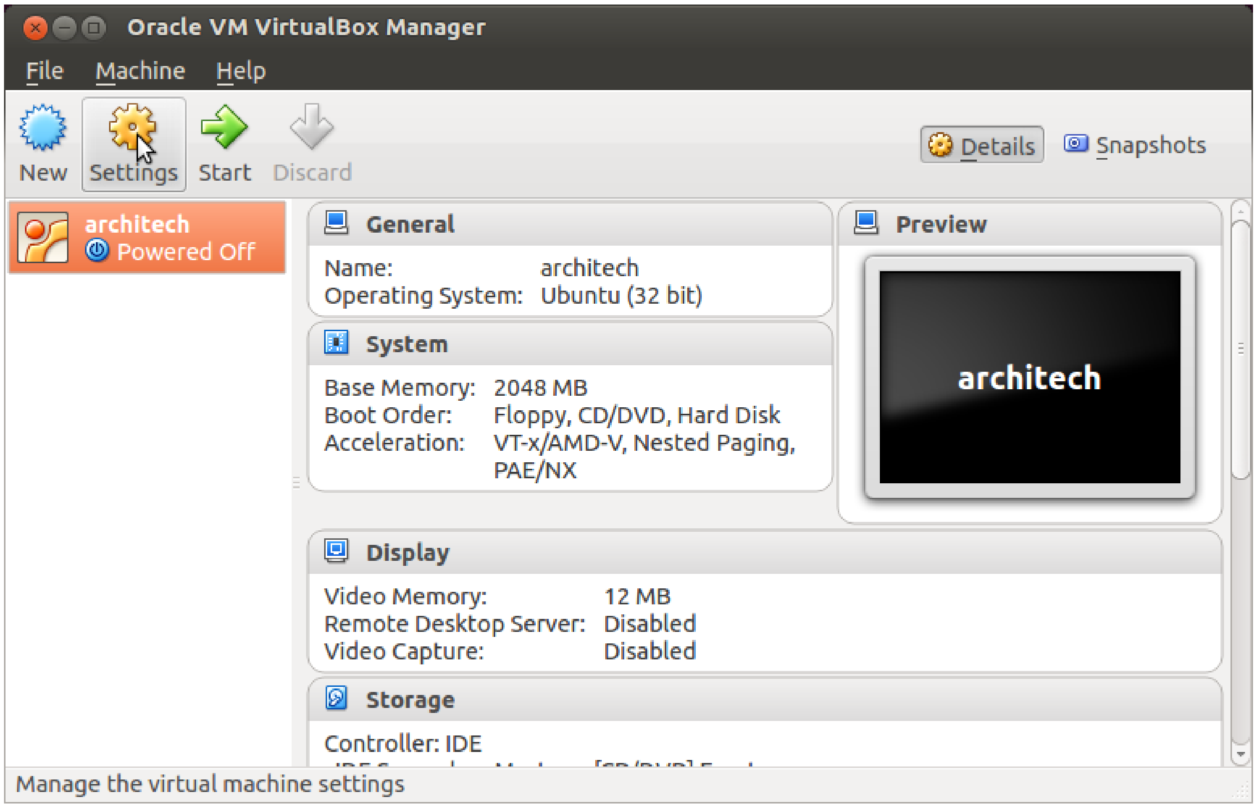
Name: architech
Operating System: Ubuntu (32 bit)

Base Memory: 2048 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX

Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Controller: IDE
IDE Secondary Master: [CD/DVD] Empty
Controller: SATA
SATA Port 0: architech.vdi (Normal, 500.00 GB)

2. Click on *Settings*



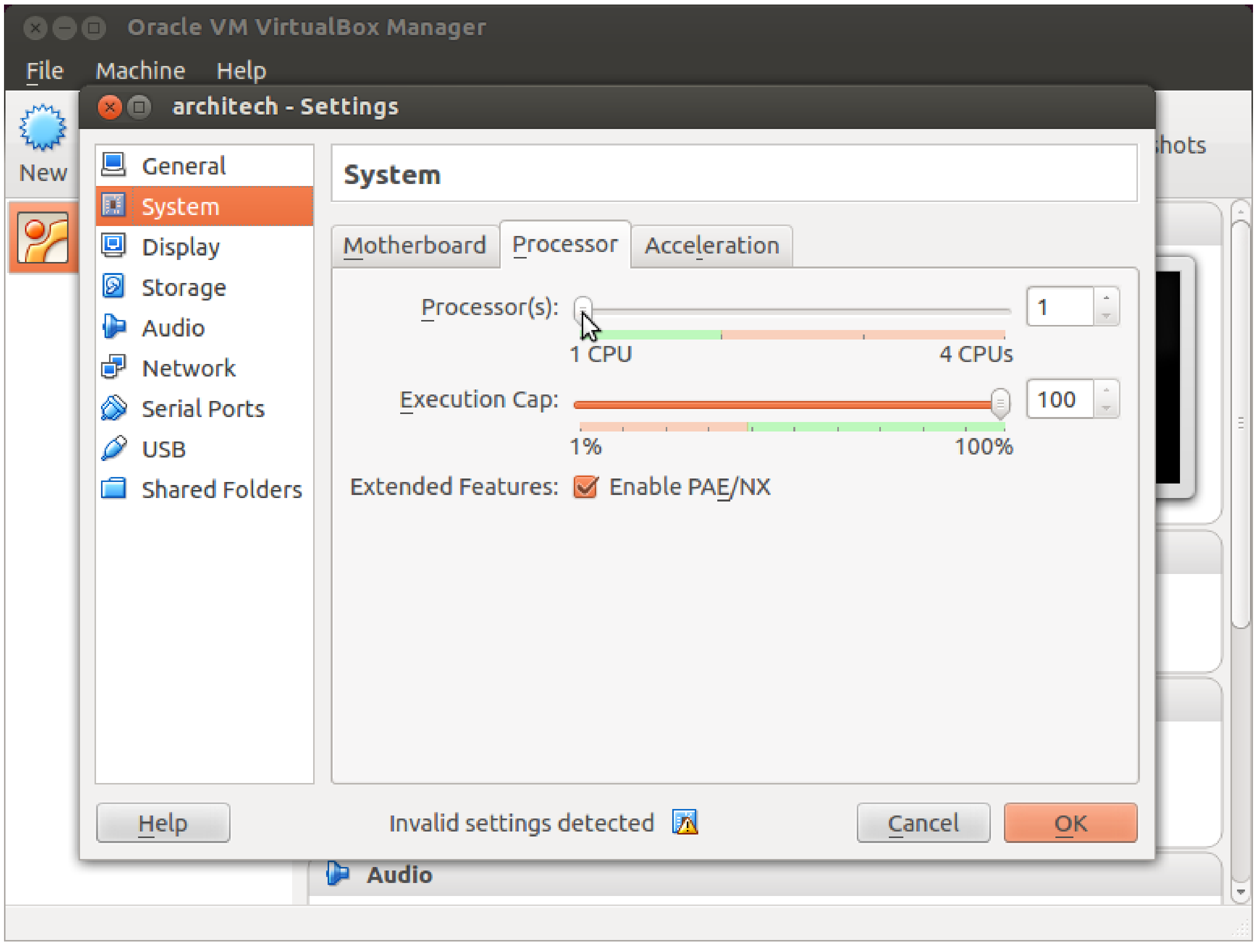
Name: architech
Operating System: Ubuntu (32 bit)

Base Memory: 2048 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX

Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Controller: IDE

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



System

Processor(s): 1 (1 CPU – 4 CPUs)

Execution Cap: 100 (1% – 100%)

Extended Features: ✓ Enable PAE/NX

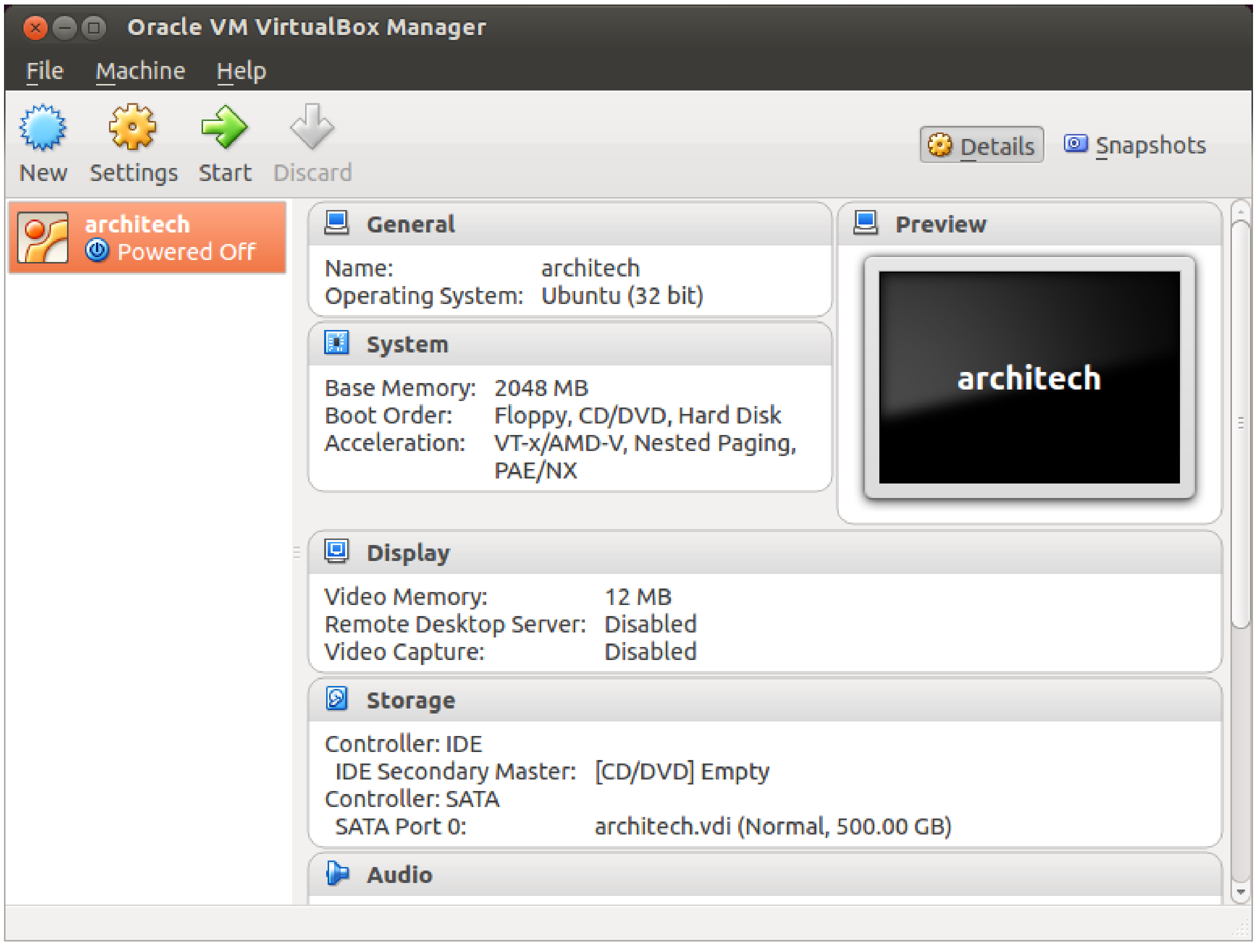
Invalid settings detected

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



General

Name: architech
Operating System: Ubuntu (32 bit)

System

Base Memory: 2048 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX

Display

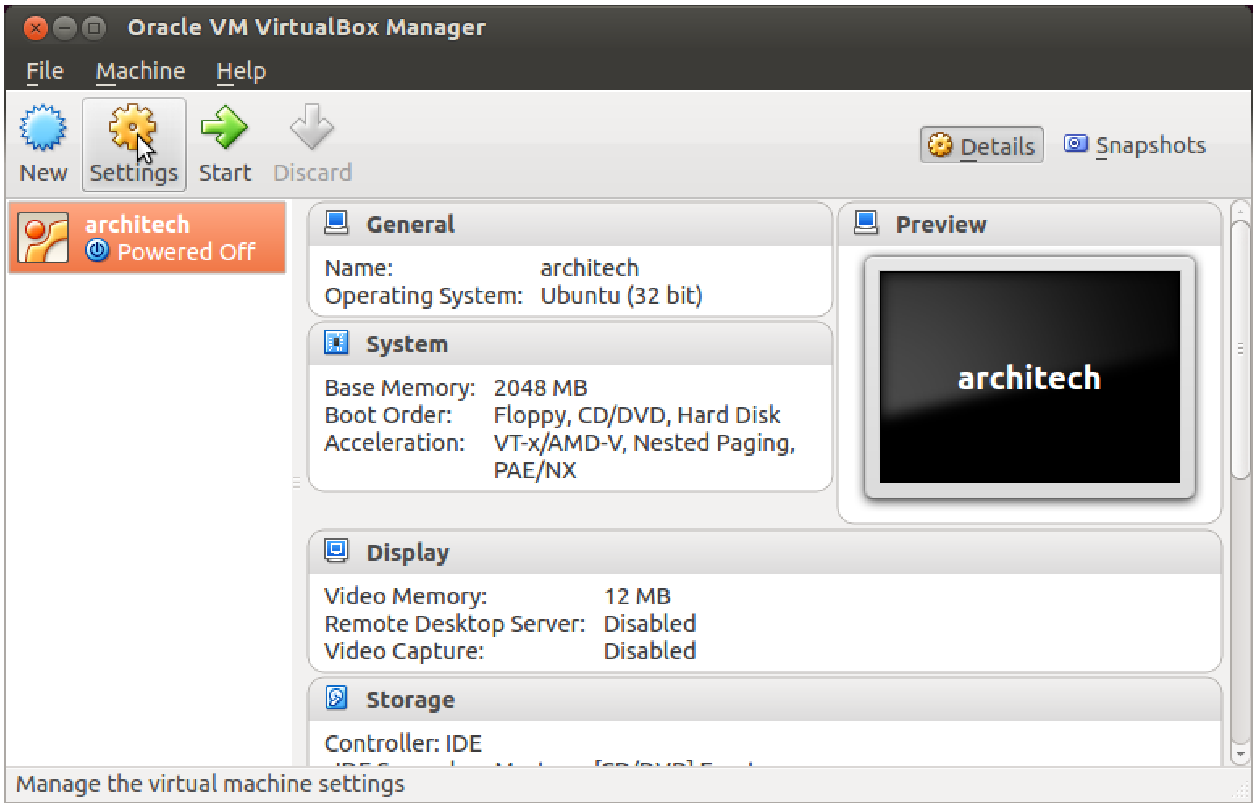
Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Storage

Controller: IDE
IDE Secondary Master: [CD/DVD] Empty
Controller: SATA
SATA Port 0: architech.vdi (Normal, 500.00 GB)

Audio

2. Click on *Settings*



General

Name: architech
Operating System: Ubuntu (32 bit)

System

Base Memory: 2048 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX

Display

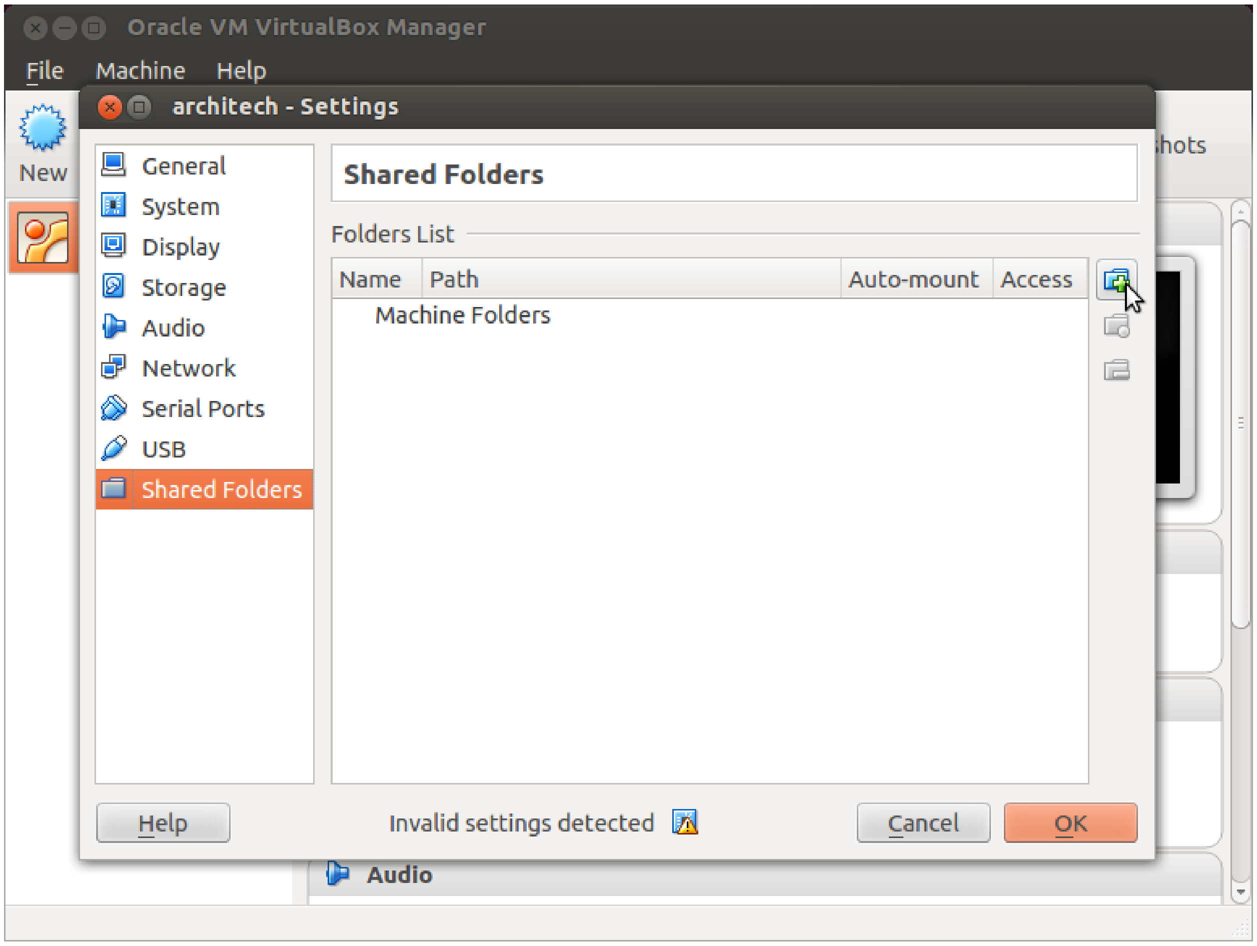
Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Storage

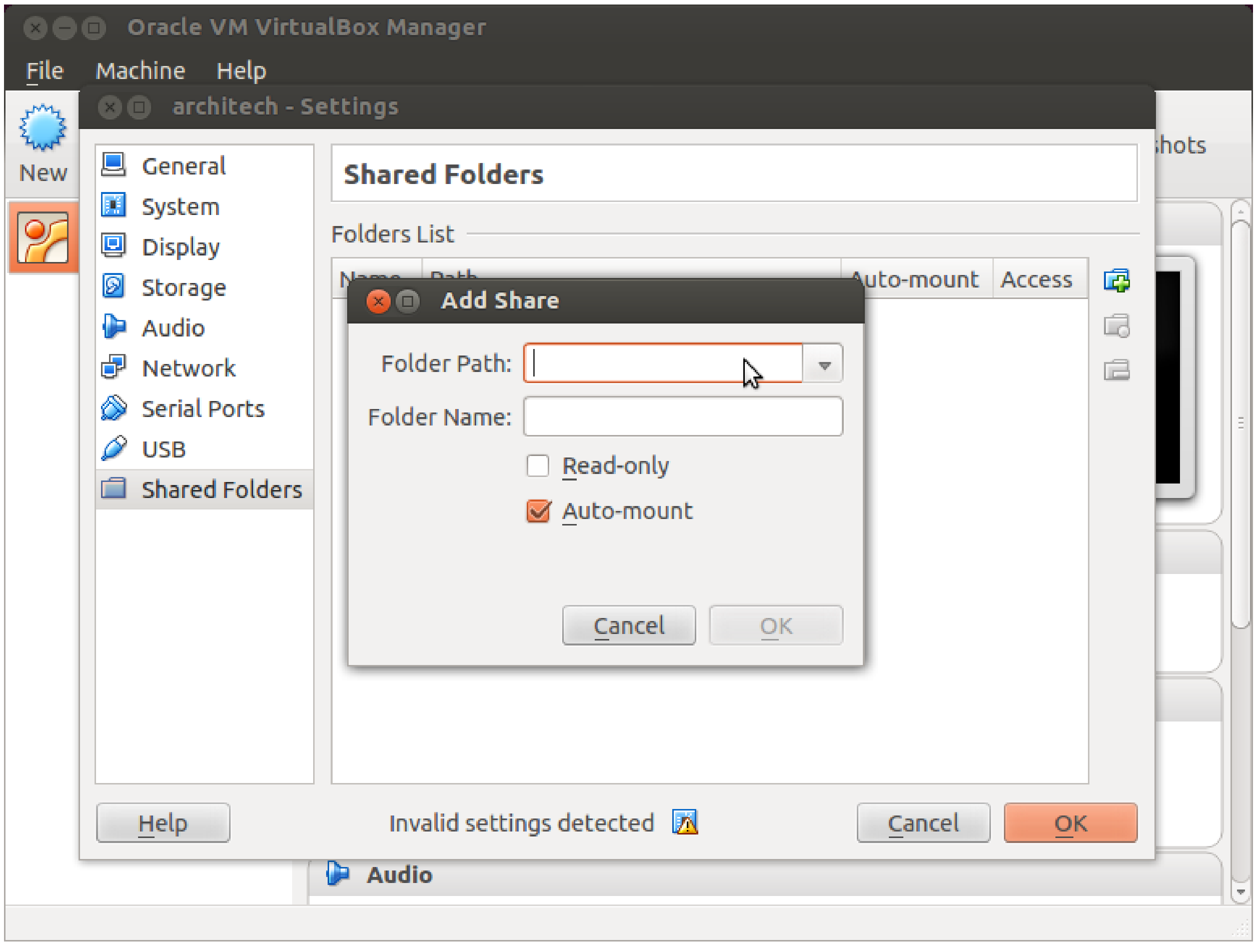
Controller: IDE

Manage the virtual machine 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.

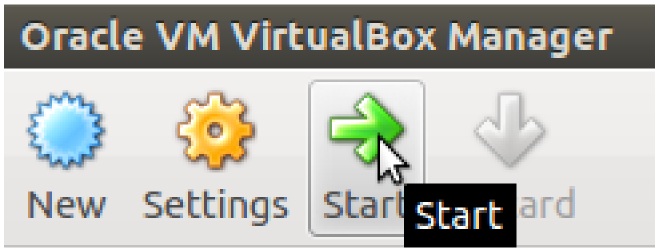


Once the virtual machine has been booted, the shared folder will be mounted under */media/* directory inside the virtual machine.

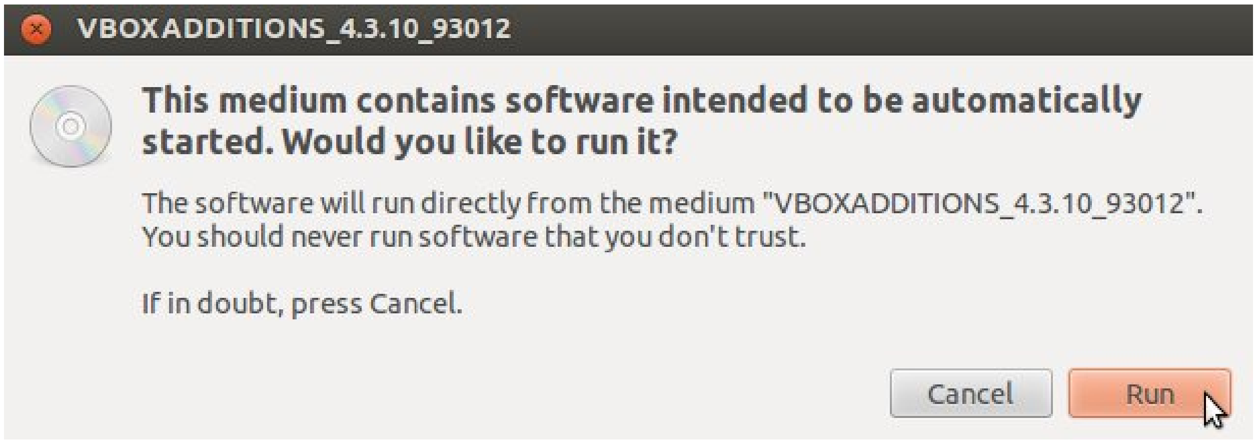
Install VBox Additions

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

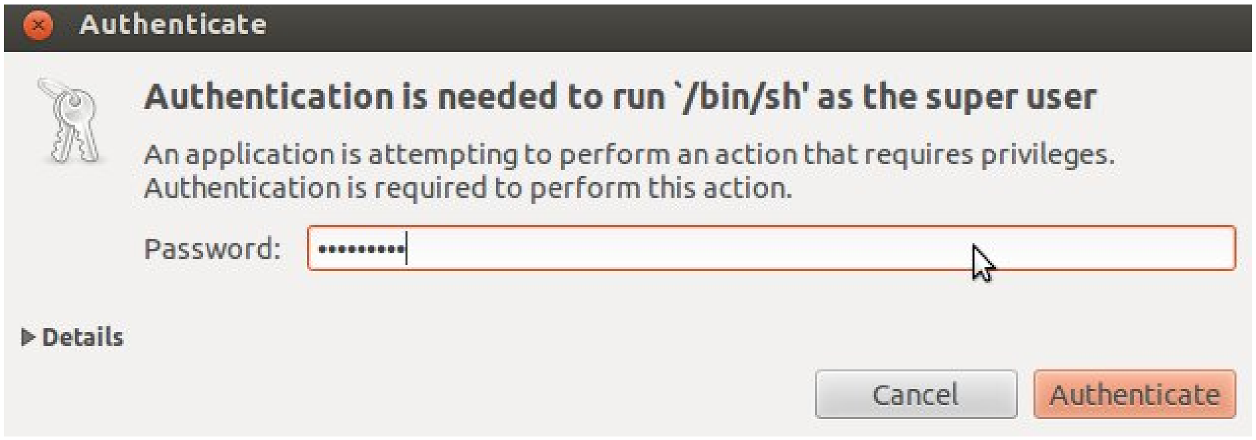

VBOXADDITIONS_4.3.10_93012

This medium contains software intended to be automatically started. Would you like to run it?

The software will run directly from the medium "VBOXADDITIONS_4.3.10_93012". You should never run software that you don't trust.

If in doubt, press Cancel.

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

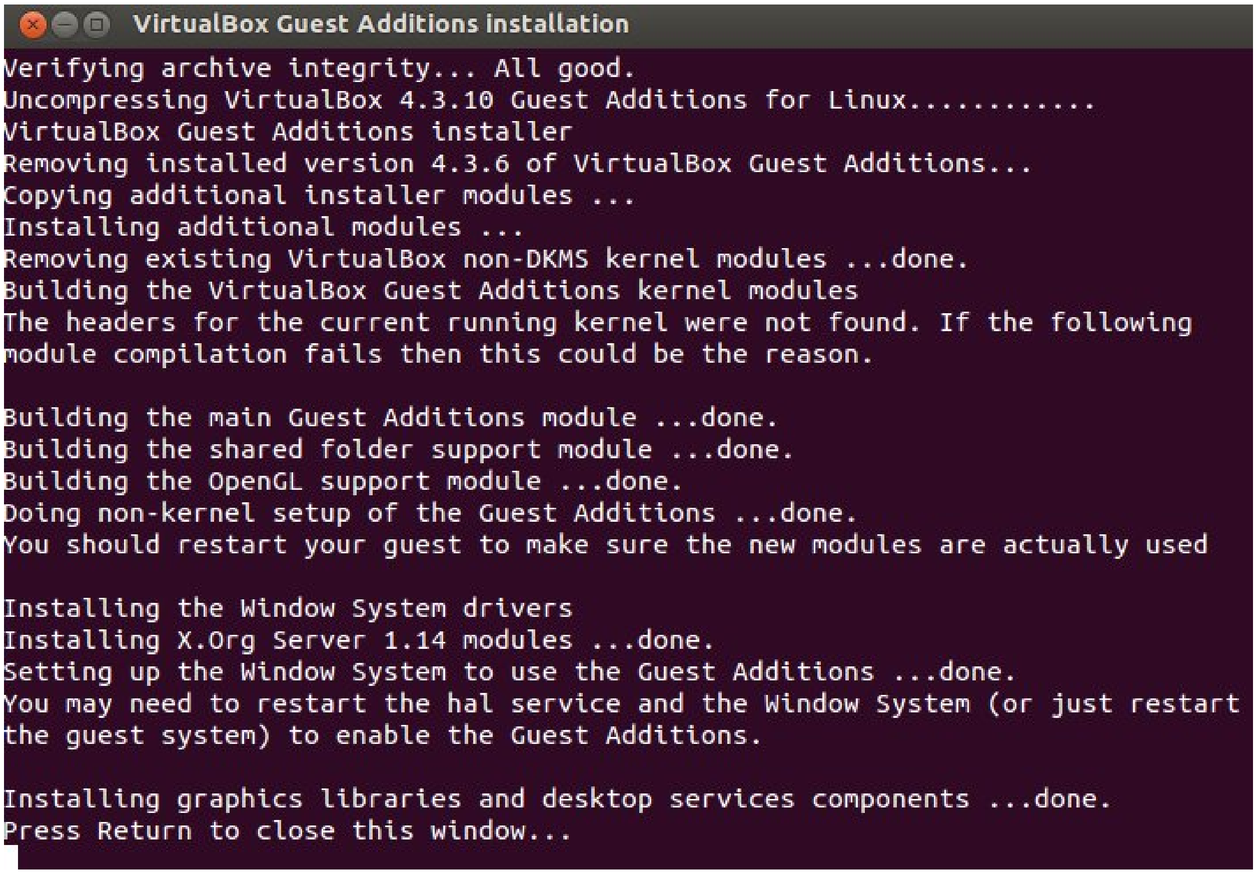

Authenticate

Authentication is needed to run `/bin/sh' as the super user

An application is attempting to perform an action that requires privileges. Authentication is required to perform this action.

Password: ••••••••••

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



VirtualBox Guest Additions installation

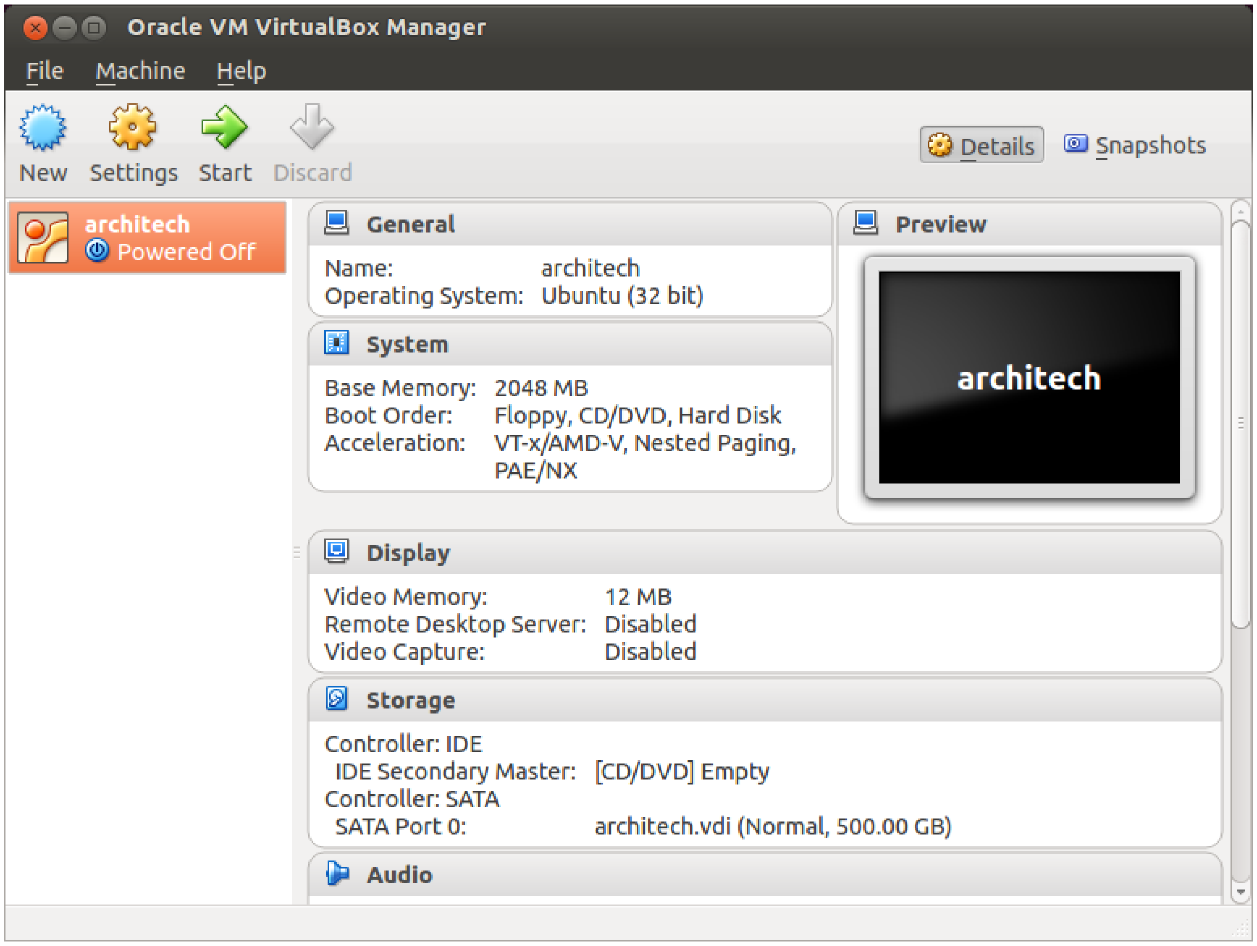
```
Verifying archive integrity... All good.
Uncompressing VirtualBox 4.3.10 Guest Additions for Linux...........
VirtualBox Guest Additions installer
Removing installed version 4.3.6 of VirtualBox Guest Additions...
Copying additional installer modules ...
Installing additional modules ...
Removing existing VirtualBox non-DKMS kernel modules ...done.
Building the VirtualBox Guest Additions kernel modules
The headers for the current running kernel were not found. If the following
module compilation fails then this could be the reason.

Building the main Guest Additions module ...done.
Building the shared folder support module ...done.
Building the OpenGL support module ...done.
Doing non-kernel setup of the Guest Additions ...done.
You should restart your guest to make sure the new modules are actually used

Installing the Window System drivers
Installing X.Org Server 1.14 modules ...done.
Setting up the Window System to use the Guest Additions ...done.
You may need to restart the hal service and the Window System (or just restart
the guest system) to enable the Guest Additions.

Installing graphics libraries and desktop services components ...done.
Press Return to close this window...
```

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



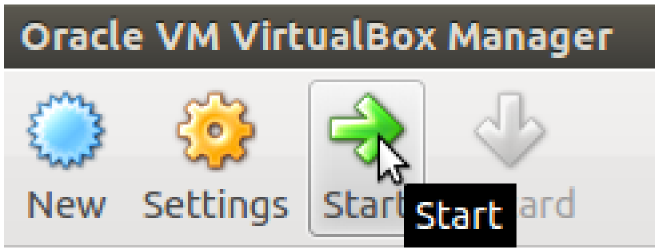
Name: architech
Operating System: Ubuntu (32 bit)

Base Memory: 2048 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX

Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

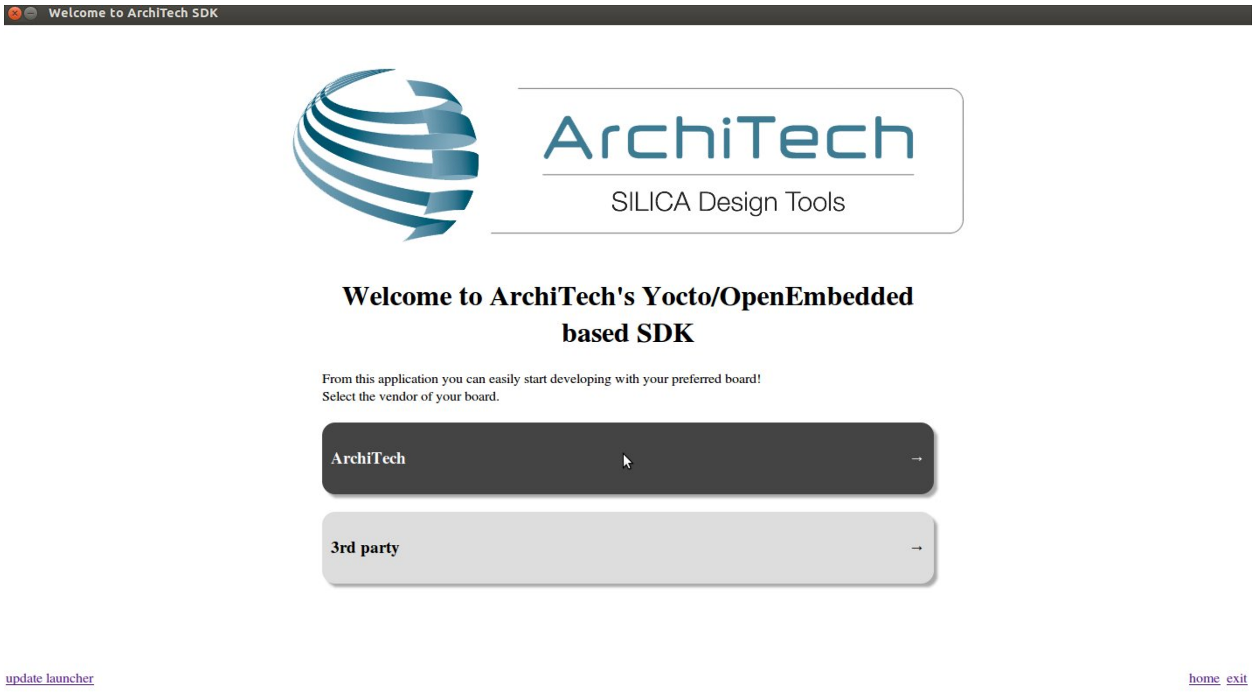
Controller: IDE
IDE Secondary Master: [CD/DVD] Empty
Controller: SATA
SATA Port 0: architech.vdi (Normal, 500.00 GB)

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.

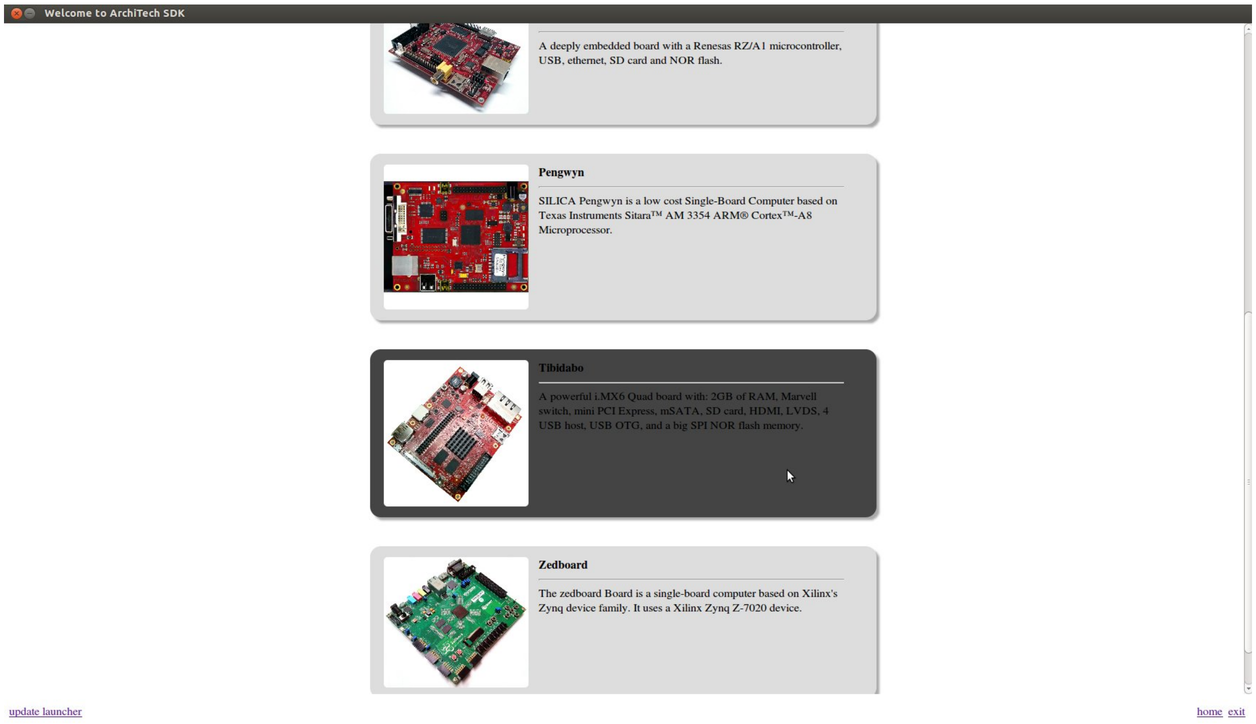
4. The first screen gives you two choices: *ArchiTech* and *3rd Party*. Choose *ArchiTech*.



Welcome to ArchiTech's Yocto/OpenEmbedded based SDK

From this application you can easily start developing with your preferred board!
Select the vendor of your board.

5. Select Tibidabo as board you want develop on.



A deeply embedded board with a Renesas RZ/A1 microcontroller, USB, ethernet, SD card and NOR flash.

Pengwyn

SILICA Pengwyn is a low cost Single-Board Computer based on Texas Instruments Sitara™ AM 3354 ARM® Cortex™-A8 Microprocessor.

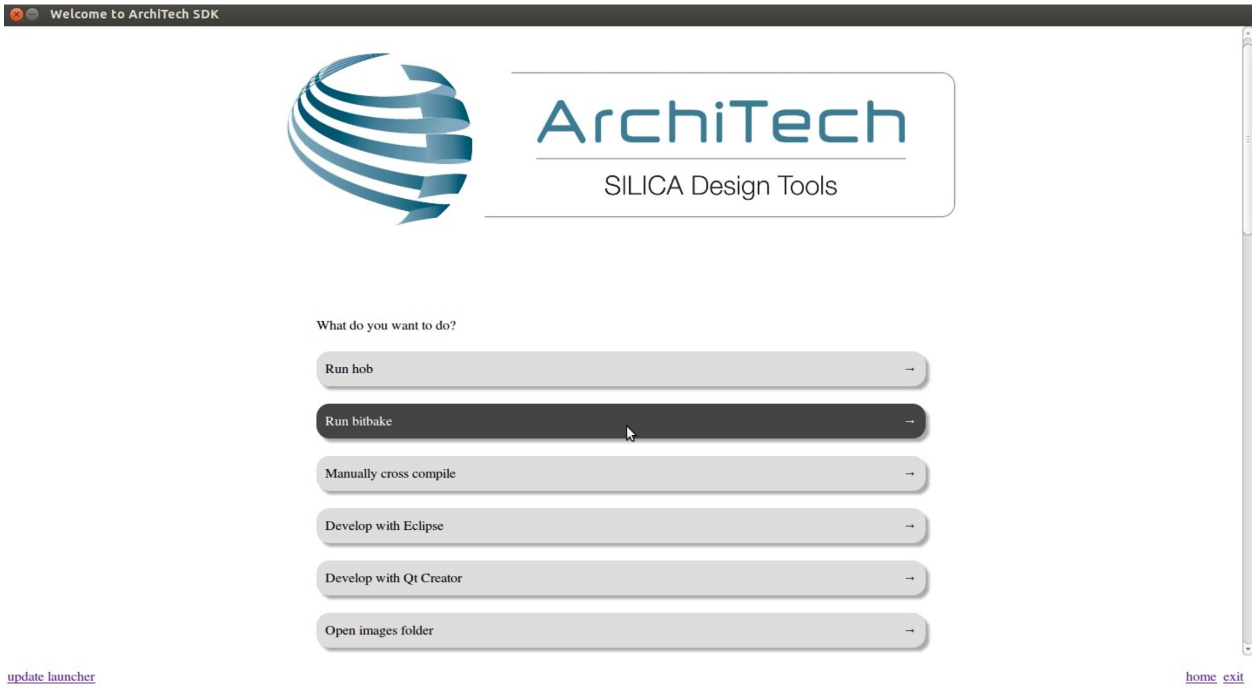
Tibidabo

A powerful i.MX6 Quad board with: 2GB of RAM, Marvell switch, mini PCI Express, mSATA, SD card, HDMI, LVDS, 4 USB host, USB OTG, and a big SPI NOR flash memory.

Zedboard

The zedboard Board is a single-board computer based on Xilinx's Zynq device family. It uses a Xilinx Zynq Z-7020 device.

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



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

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

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

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

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

Note: **sudo** password is: “**architech**“

Deploy

To deploy the root file system, you are going to need a micro SD card.

1. Copy the root file system to your SD card

Warning: Be very careful when you use *dd* to write to a device to pick up the right device, otherwise you can mess up another disk you have on your machine, destroying its content forever!

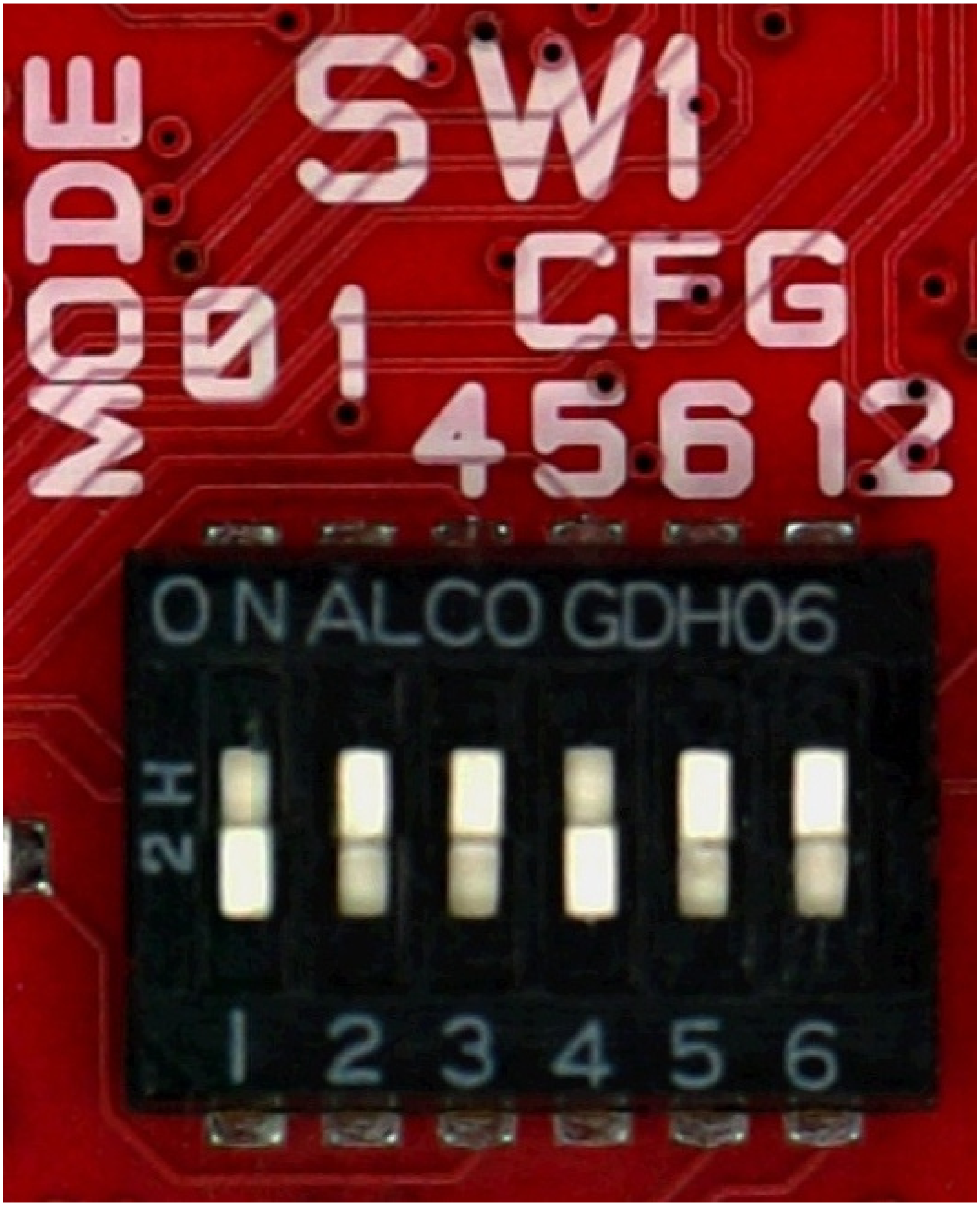
Warning: The content of the SD card will be lost forever!

Important: Be sure you **unmount the device** from the filesystem before using **dd** program, you sure don't want to have the operating system interfere during the write process.

2. After *dd* completes, make sure everything has been really written to the SD card:
3. Unmount the micro SD card from your computer
4. Plug the micro SD in the board socket.

Boot

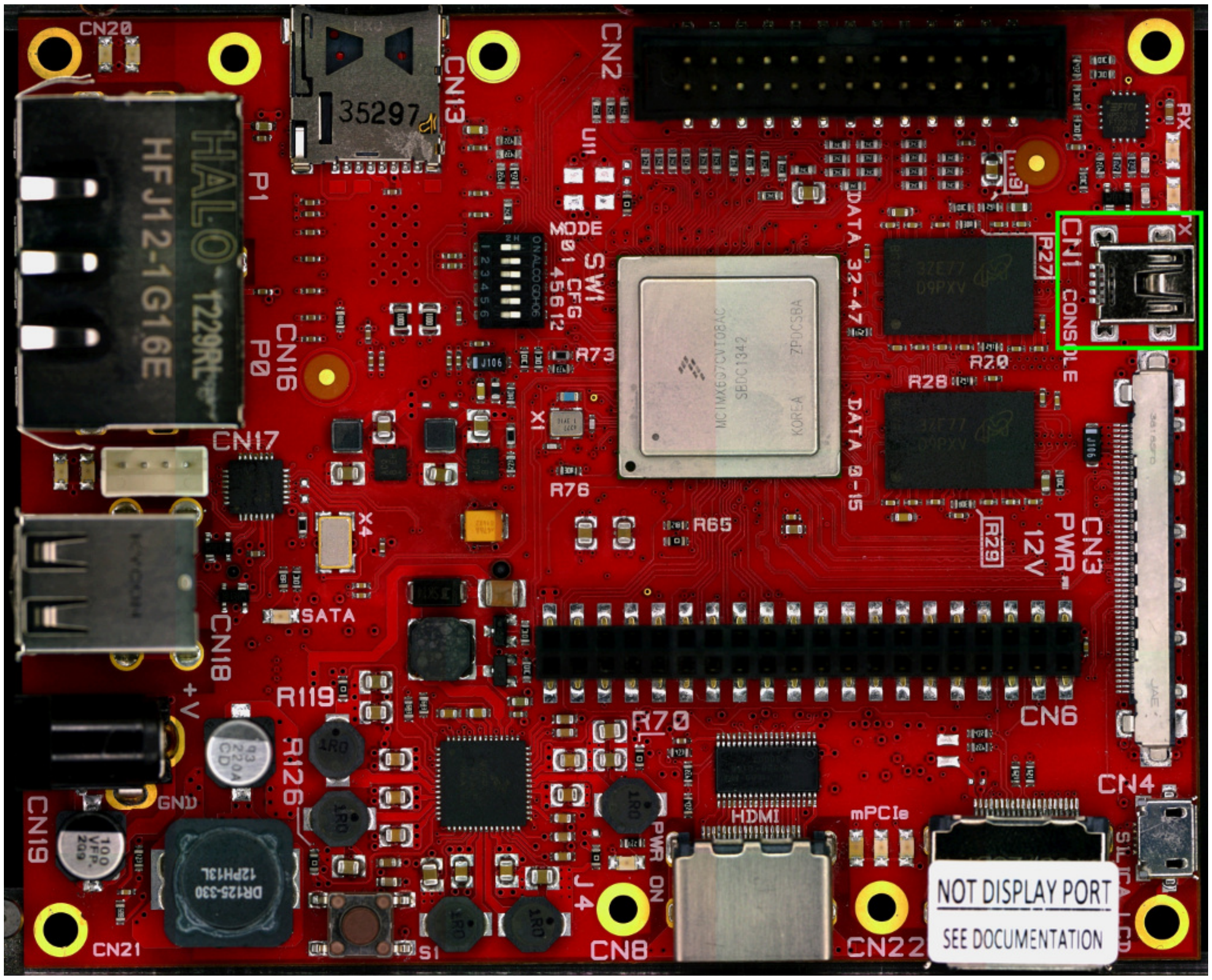
First of all, make sure the board can boot entirely from the micro SD card by setting *SW1* with this configuration



Take the power socket adapter compatible with your country, plug it in the power adapter. When in position, you should hear a slight *click*. Power on the board connecting the external power adapter to Tibidabo connector **CN19**.

Now it's time to start the serial console.

On Tibidabo there is the dedicated serial console connector **CN1**



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

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

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

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

then you can setup your port with these parameters:

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

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

Give *root* to the login prompt:

Board

```
tibidabo login: root
```

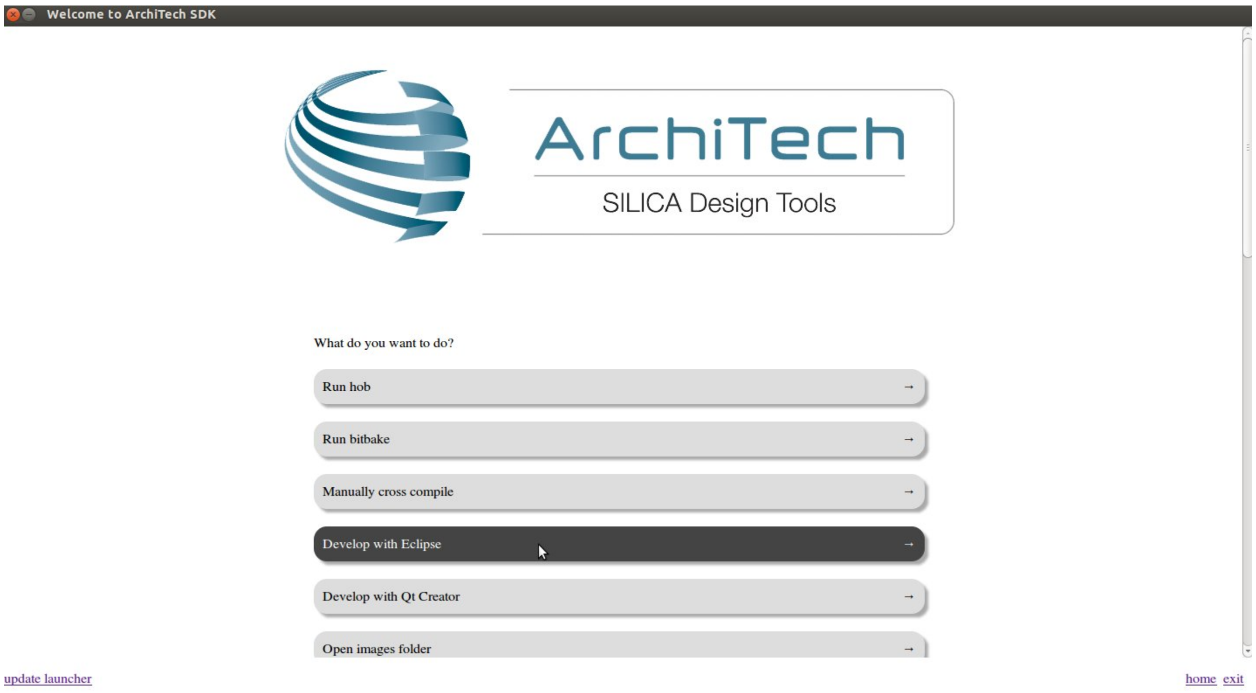
and press *Enter*.

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.

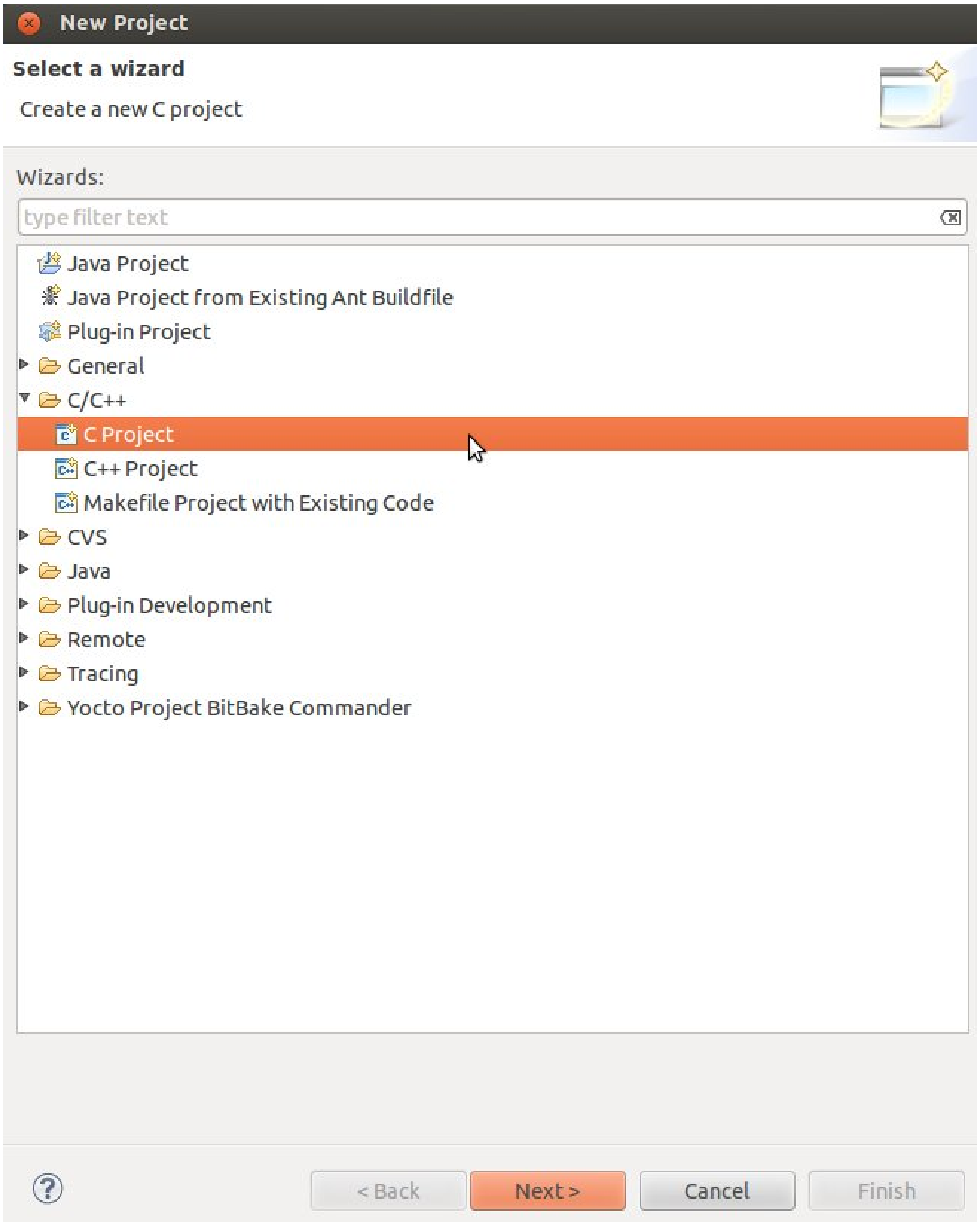
Code

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

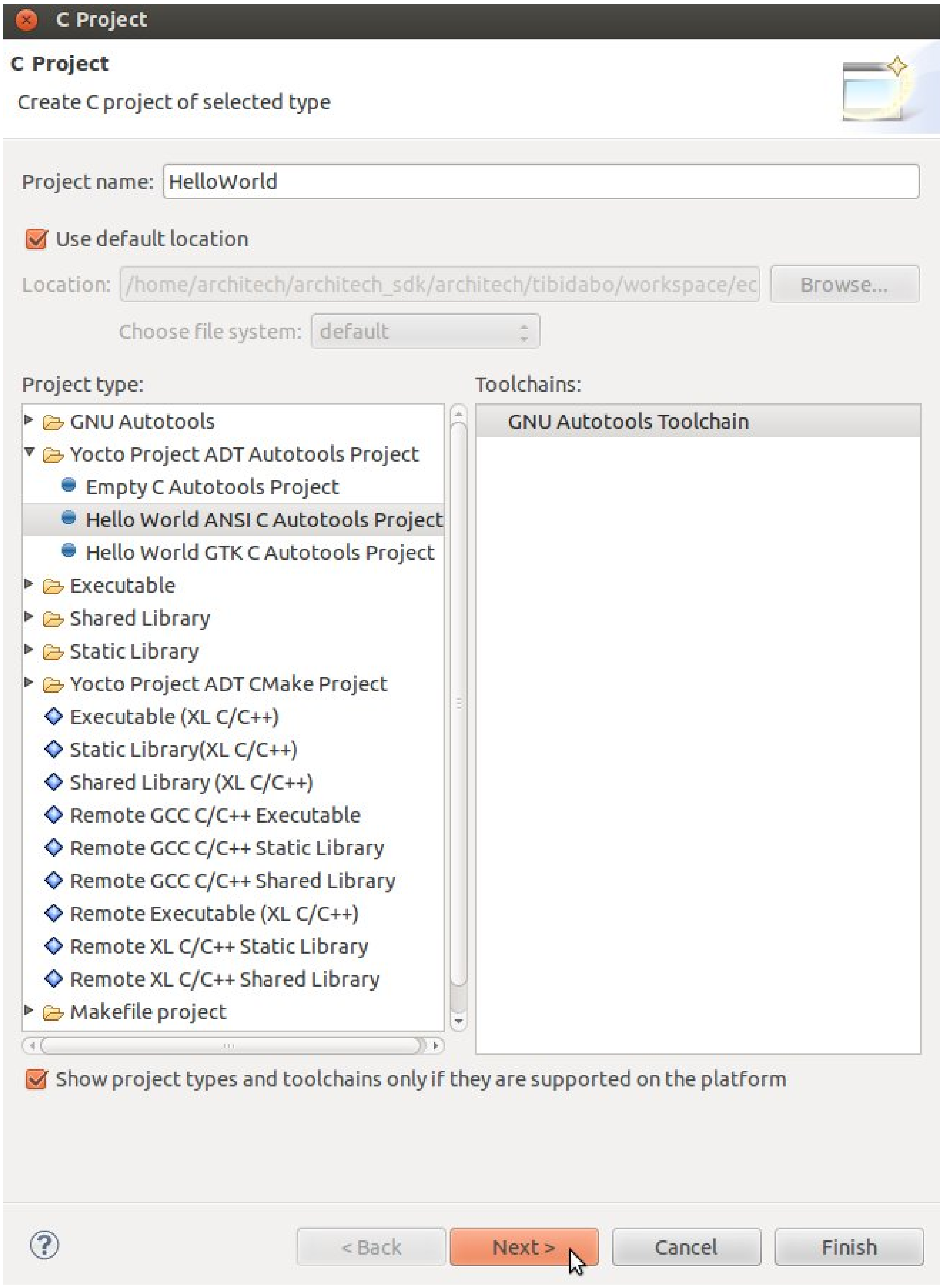
1. Return to the **Splashscreen**, which we left on Tibidabo board screen, and click on *Develop with Eclipse*.



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/tibidabo/workspace/ec

Choose file system: default

Project type:

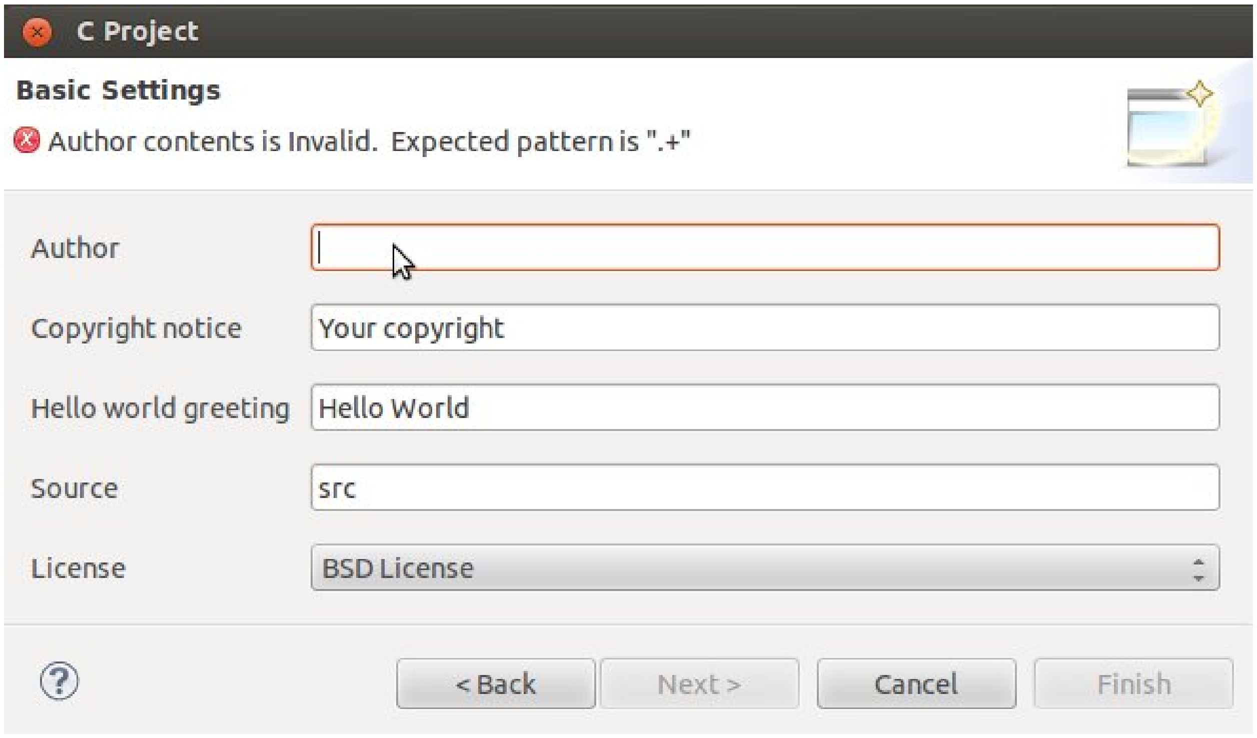
- GNU Autotools
- Yocto Project ADT Autotools Project
  - Empty C Autotools Project
  - Hello World ANSI C Autotools Project
  - Hello World GTK C Autotools Project
- Executable
- Shared Library
- Static Library
- Yocto Project ADT CMake Project
- Executable (XL C/C++)
- Static Library(XL C/C++)
- Shared 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

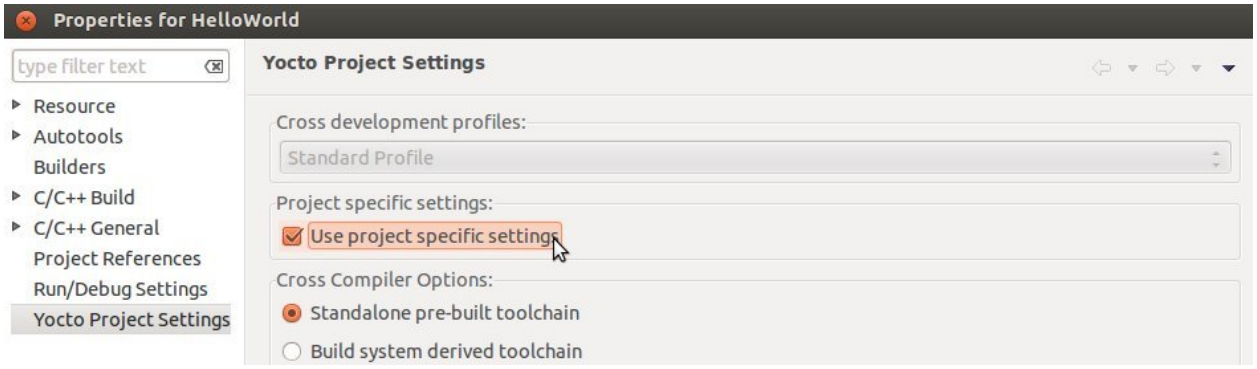
4. Insert *Author* field and click on *Finish* button. Select *Yes* on the *Open Associated Perspective?* question.



Author contents is Invalid. Expected pattern is ".+"

Author:
Copyright notice: Your copyright
Hello world greeting: Hello World
Source: src
License: BSD License

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.



Yocto Project Settings

Cross development profiles: Standard Profile

Project specific settings: ✓ Use project specific settings

Cross Compiler Options: ● Standalone pre-built toolchain ○ Build system derived 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 CN16 Port P0) to your PC. Configure your workstation ip address as 192.168.0.100. Make sure the board can be seen by your host machine:

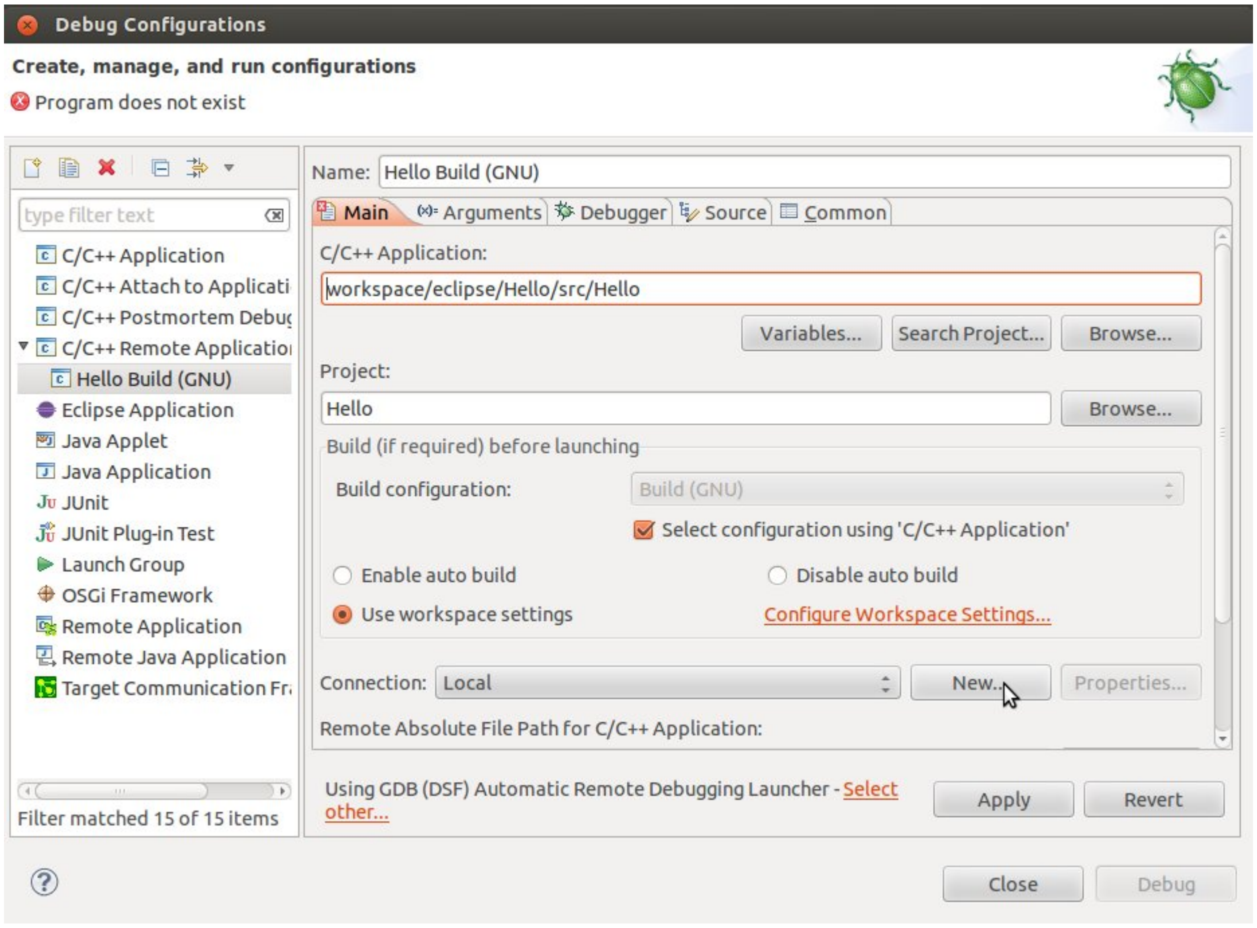
If the output is similar to this one:

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



Name: Hello Build (GNU)

C/C++ Application: workspace/eclipse/Hello/src/Hello

Project: Hello

Build (if required) before launching

Build configuration: Build (GNU)

[✓] Select configuration using 'C/C++ Application'

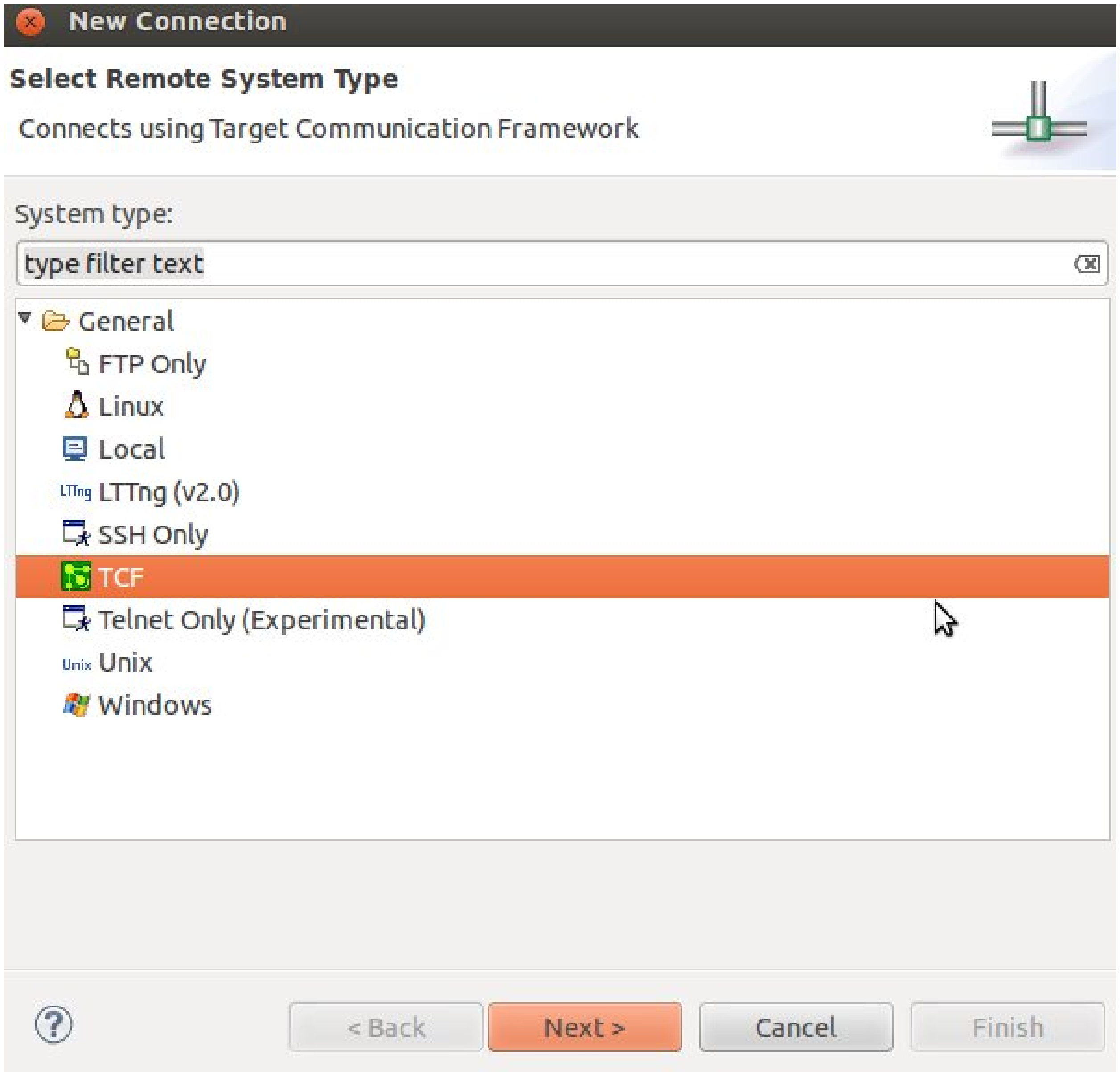
Enable auto build | Disable auto build | [✓ Use workspace settings] | Configure Workspace Settings...

Connection: Local

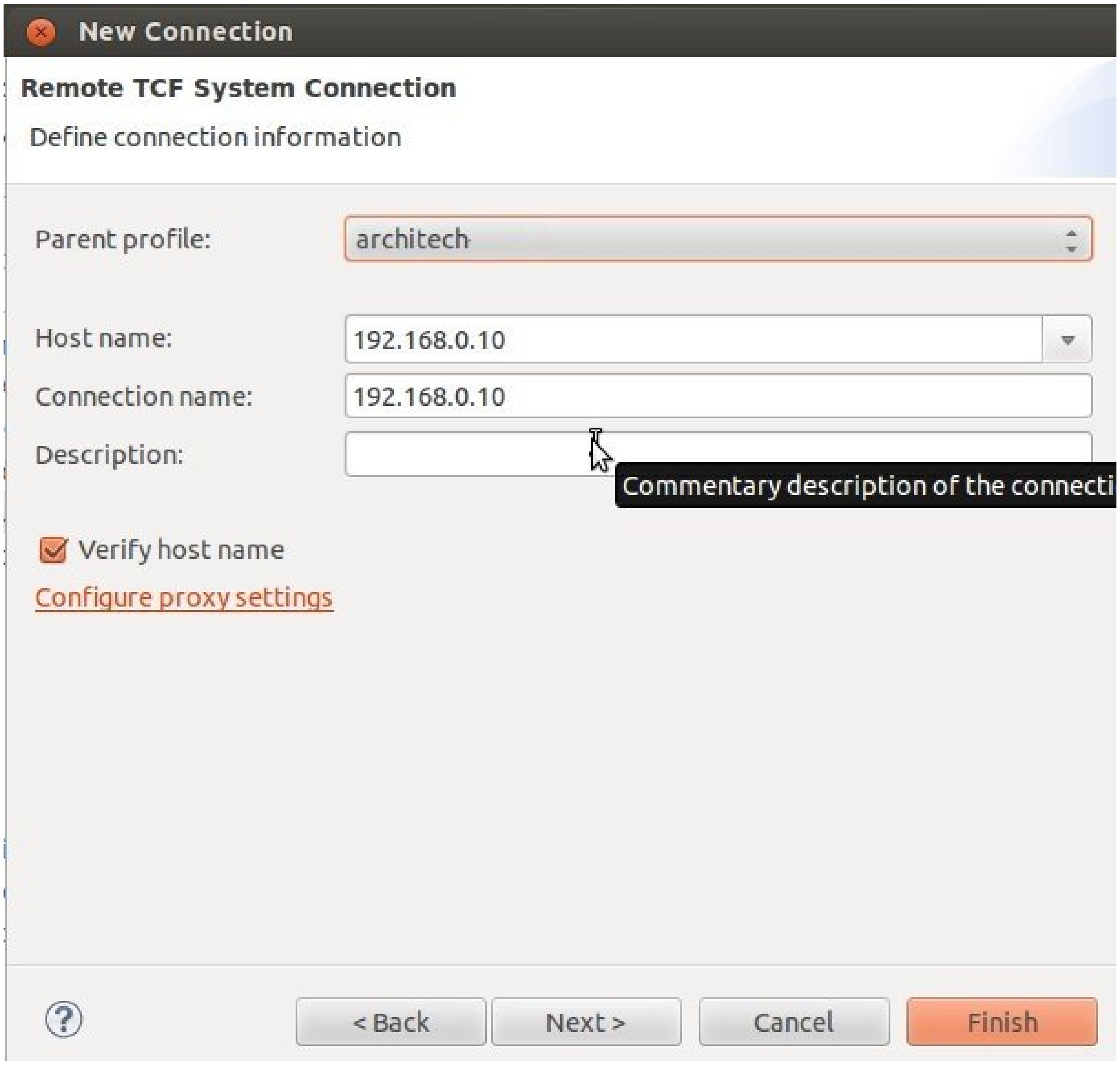
Remote Absolute File Path for C/C++ Application:

Using GDB (DSF) Automatic Remote Debugging Launcher - Select other...

- 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)



New Connection

Remote TCF System Connection

Define connection information

Parent profile: architech

Host name: 192.168.0.10

Connection name: 192.168.0.10

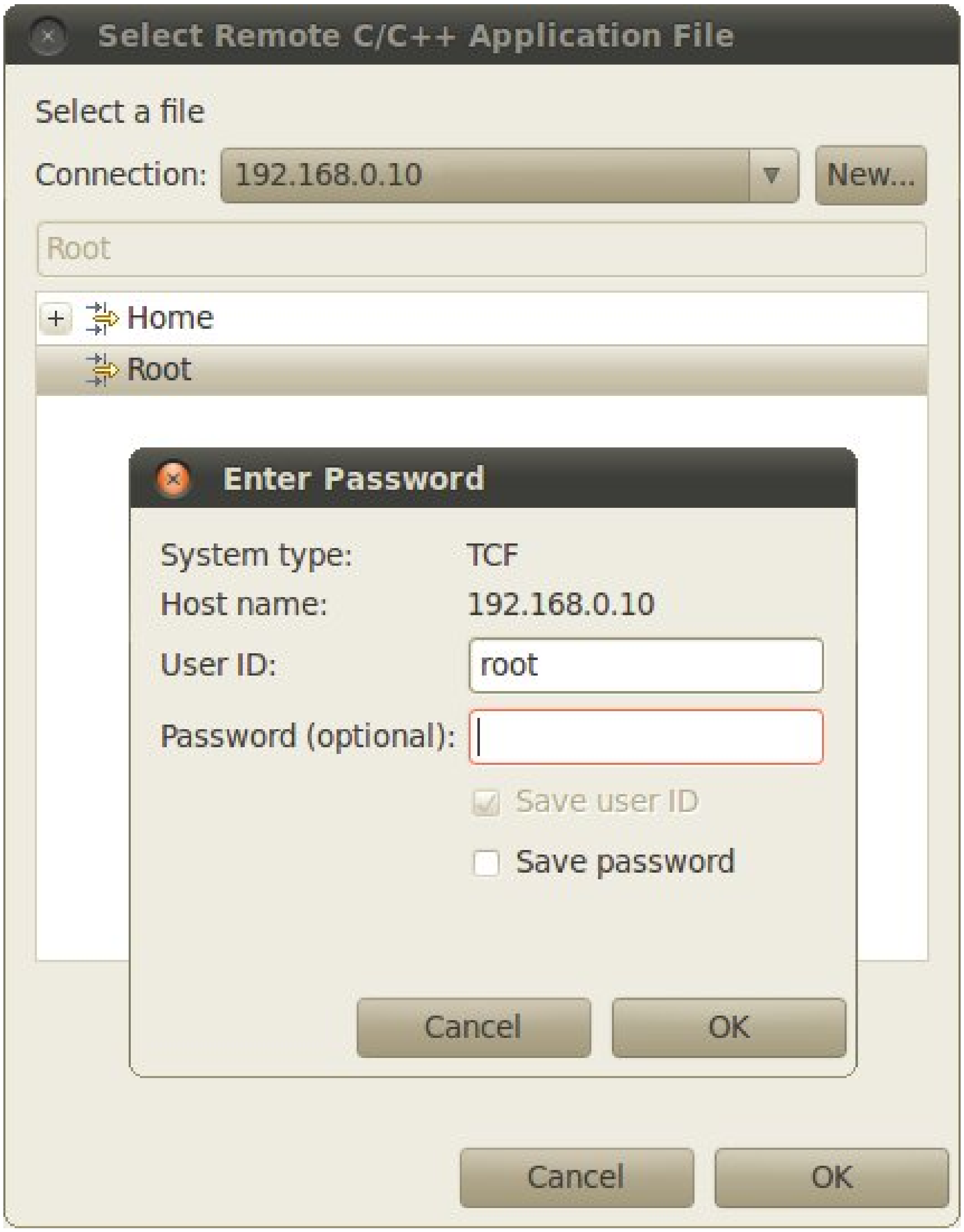
Description:

Commentary description of the connecti

[✓] Verify host name

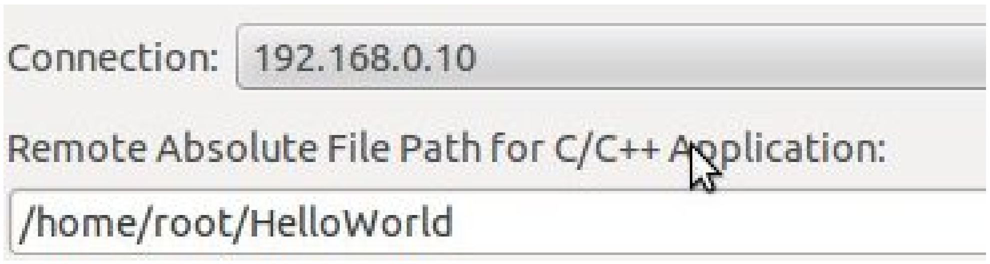
Configure proxy settings

- 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.



System type: TCF
Host name: 192.168.0.10
User ID: root
Password (optional):
[✓ Save user ID]
Save password

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

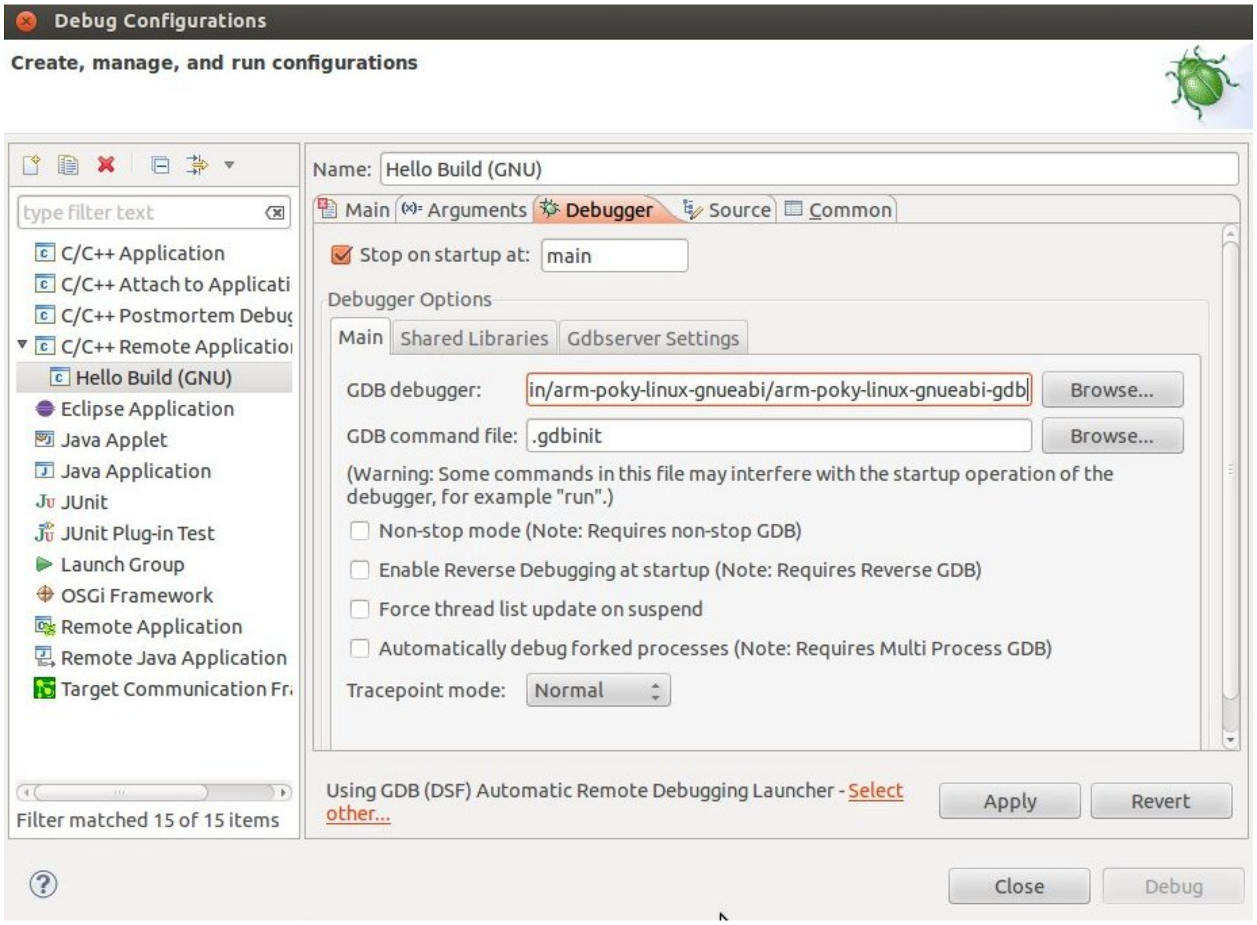


Connection: 192.168.0.10

Remote Absolute File Path for C/C++ Application:

/home/root/HelloWorld

- Select *Debugger* tab



Debug Configurations

Create, manage, and run configurations

Name: Hello Build (GNU)

Stop on startup at: main

Debugger Options

GDB debugger: in/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gdb

GDB command file: .gdbinit

(Warning: Some commands in this file may interfere with the startup operation of the debugger, for example "run".)

☐ Non-stop mode (Note: Requires non-stop GDB)

☐ Enable Reverse Debugging at startup (Note: Requires Reverse GDB)

☐ Force thread list update on suspend

☐ Automatically debug forked processes (Note: Requires Multi Process GDB)

Tracepoint mode: Normal

Using GDB (DSF) Automatic Remote Debugging Launcher - Select other...

- 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 Tibidabo is composed by several components, the most important of which are:

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

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

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

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

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

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

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, Tibidabo 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

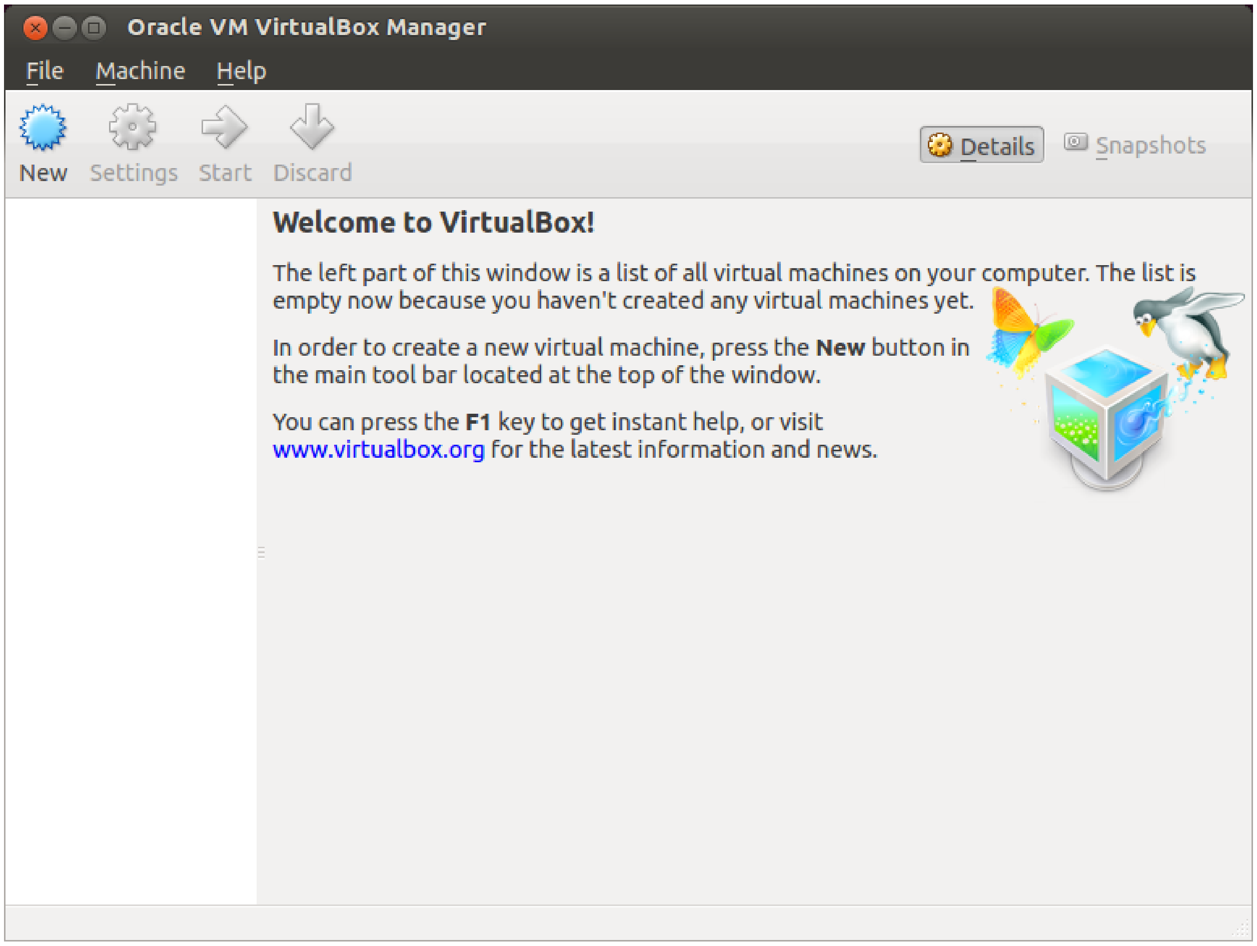
Download the version that suits your host operating system. You need to download and install the **Extension Pack** as well.

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



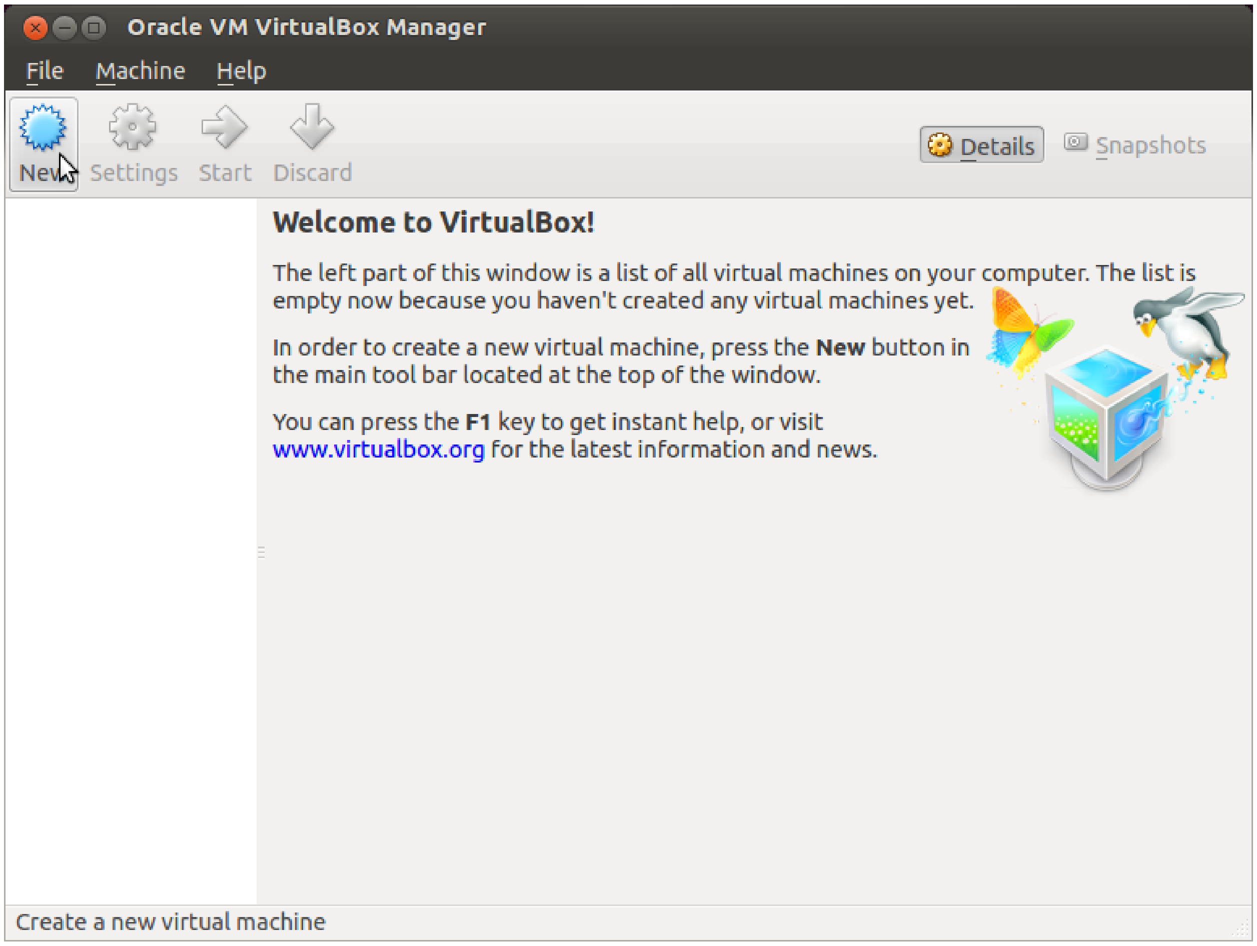
Welcome to VirtualBox!

The left part of this window is a list of all virtual machines on your computer. The list is empty now because you haven't created any virtual machines yet.

In order to create a new virtual machine, press the **New** button in the main tool bar located at the top of the window.

You can press the **F1** key to get instant help, or visit www.virtualbox.org for the latest information and news.

2. Click on *New* button



Welcome to VirtualBox!

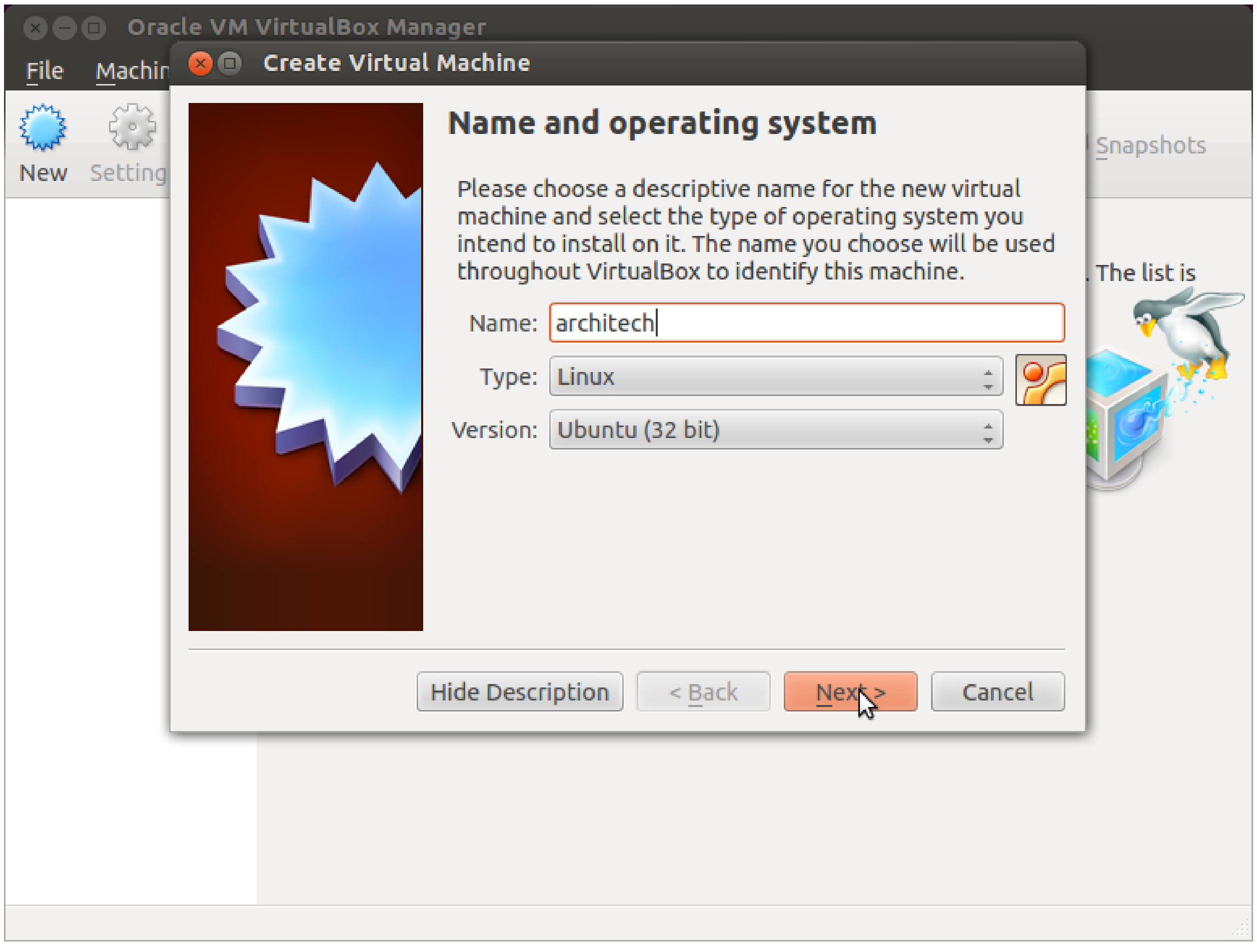
The left part of this window is a list of all virtual machines on your computer. The list is empty now because you haven't created any virtual machines yet.

In order to create a new virtual machine, press the **New** button in the main tool bar located at the top of the window.

You can press the **F1** key to get instant help, or visit www.virtualbox.org for the latest information and news.

Create a new virtual machine

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



Name and operating system

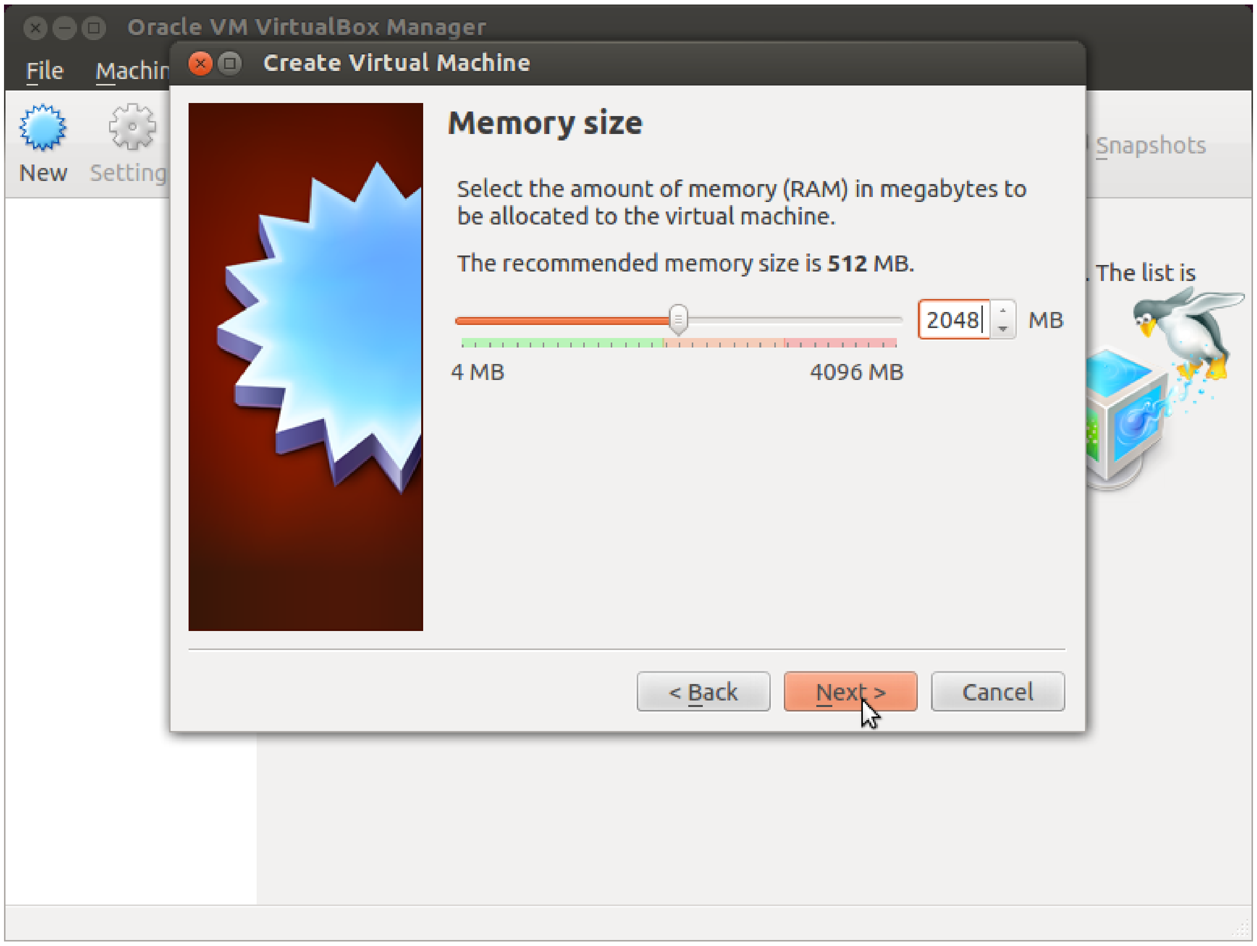
Please choose a descriptive name for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name: architech

Type: Linux

Version: Ubuntu (32 bit)

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



Memory size

Select the amount of memory (RAM) in megabytes to be allocated to the virtual machine.

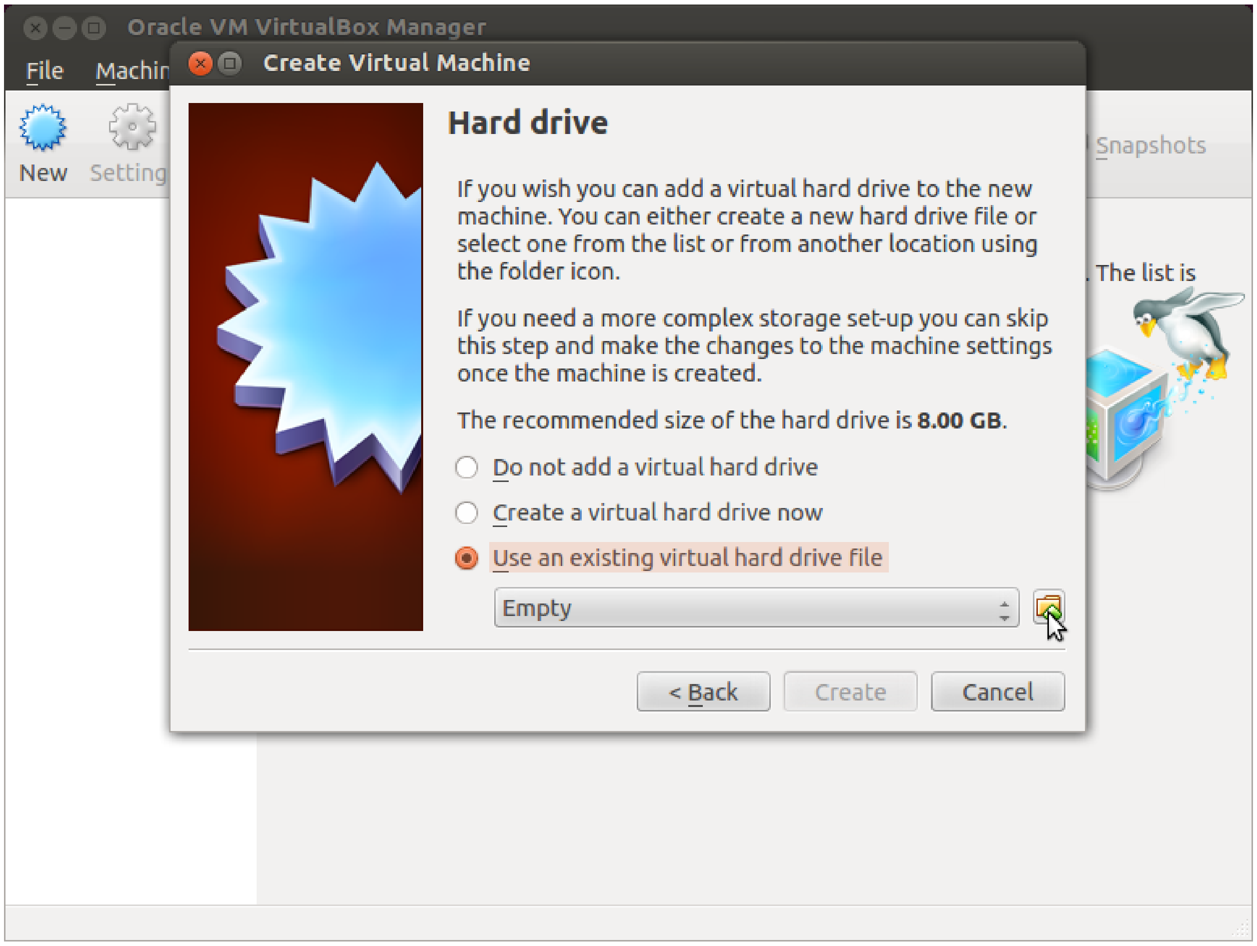
The recommended memory size is **512** MB.

2048 MB

4 MB 4096 MB

The list is

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



Hard drive

If you wish you can add a virtual hard drive to the new machine. You can either create a new hard drive file or select one from the list or from another location using the folder icon.

If you need a more complex storage set-up you can skip this step and make the changes to the machine settings once the machine is created.

The recommended size of the hard drive is **8.00 GB**.

Do not add a virtual hard drive
Create a virtual hard drive now
[✓ Use an existing virtual hard drive file]

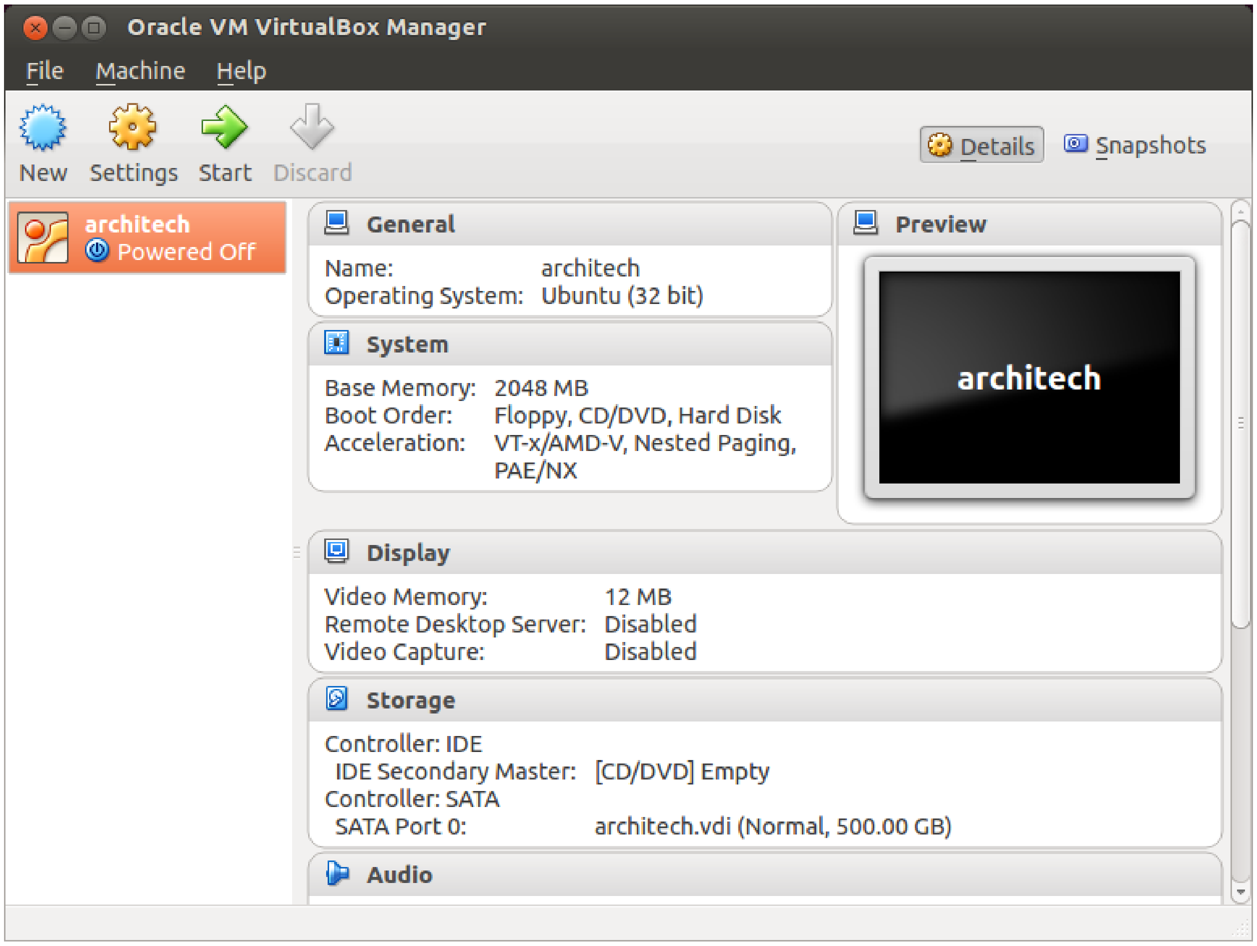
Empty

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



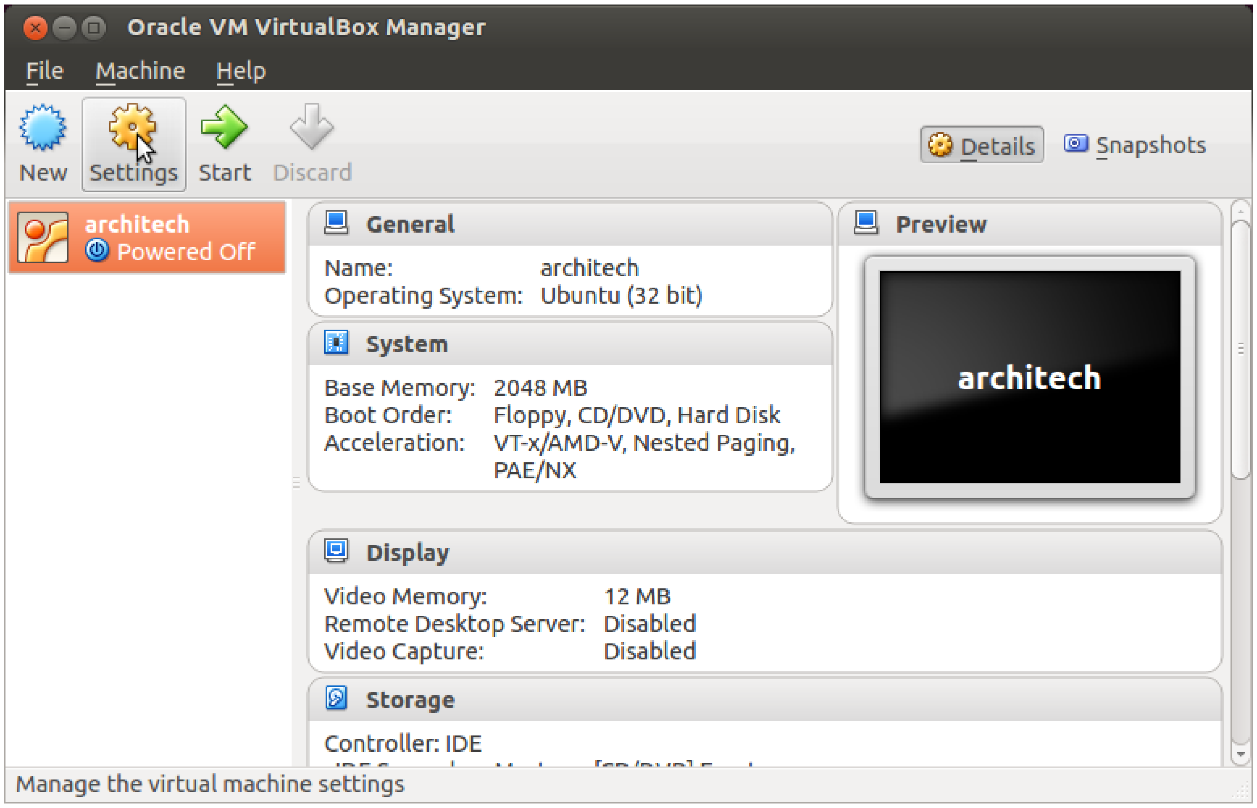
Name: architech
Operating System: Ubuntu (32 bit)

Base Memory: 2048 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX

Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Controller: IDE
 IDE Secondary Master: [CD/DVD] Empty
Controller: SATA
 SATA Port 0: architech.vdi (Normal, 500.00 GB)

2. Click on *Settings*


Oracle VM VirtualBox Manager
File Machine Help
New Settings Start Discard
Details Snapshots
architech
Powered Off
Preview
architech
Manage the virtual machine settings

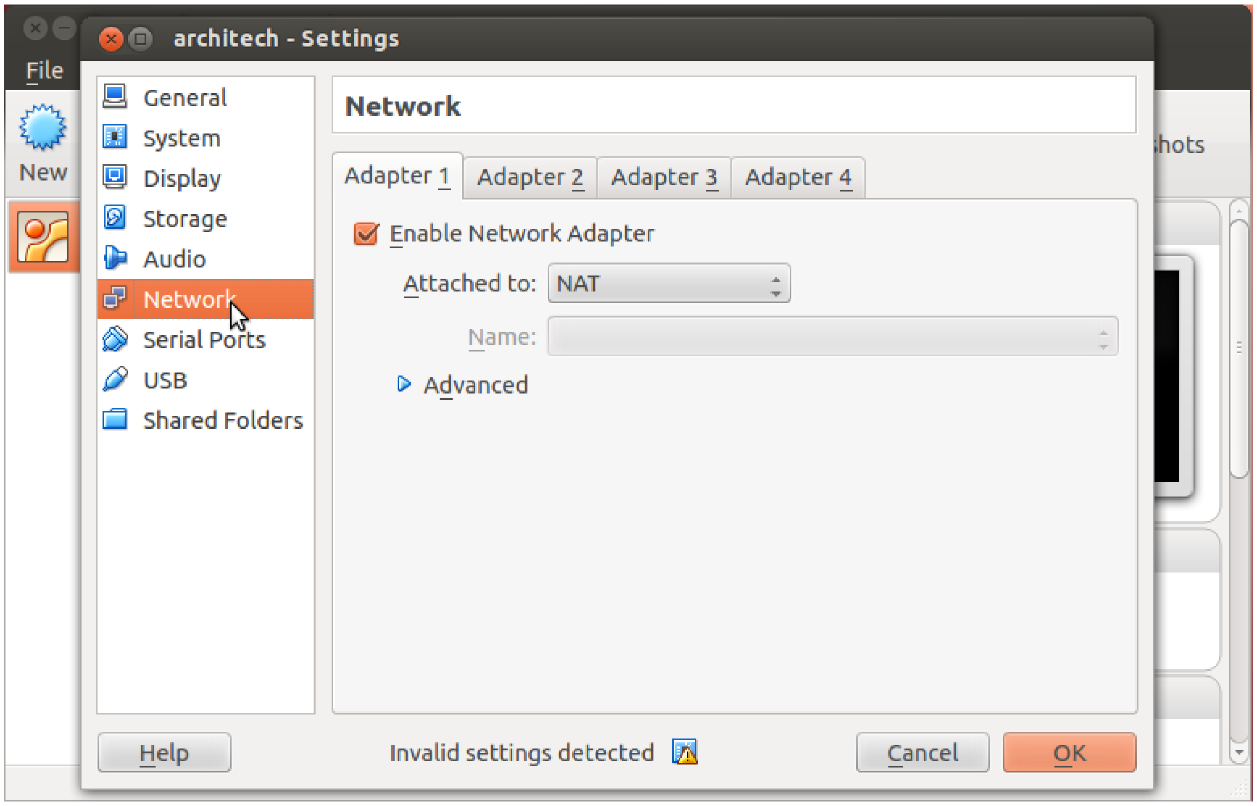
General
Name: architech
Operating System: Ubuntu (32 bit)

System
Base Memory: 2048 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX

Display
Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Storage
Controller: IDE

3. Select *Network*



Network

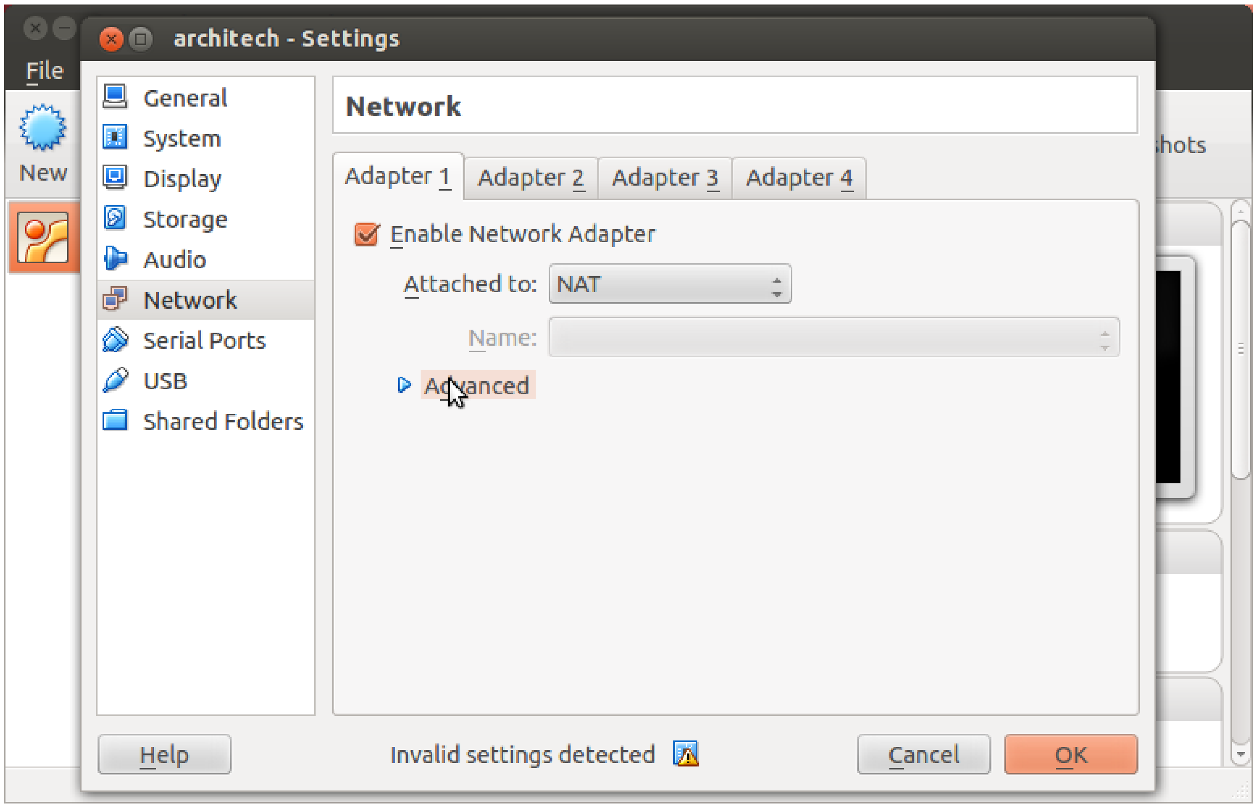
Enable Network Adapter

Attached to: NAT

Name:

Invalid settings detected

4. Expand *Advanced* of *Adapter 1*



Network

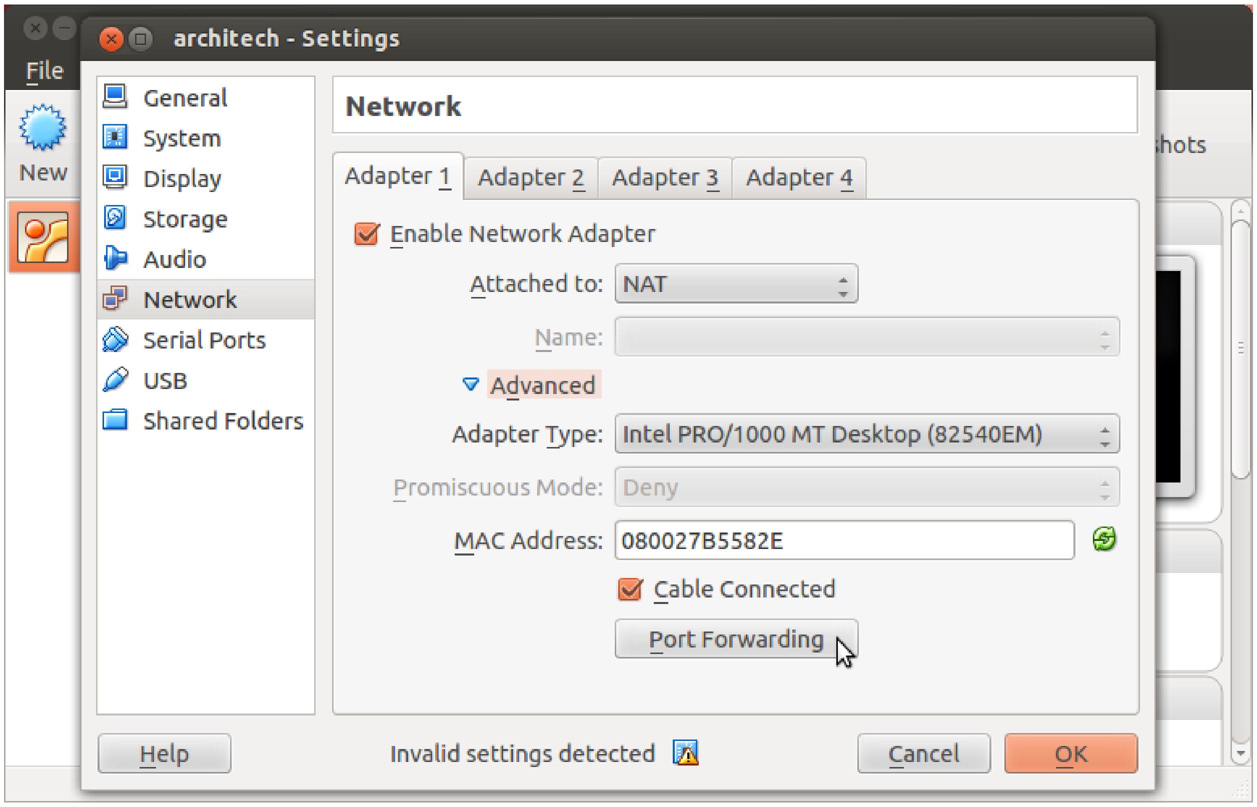
Enable Network Adapter

Attached to: NAT

Name:

Invalid settings detected

5. Click on *Port Forwarding*



Network

Enable Network Adapter

Attached to: NAT

Name:

Adapter Type: Intel PRO/1000 MT Desktop (82540EM)

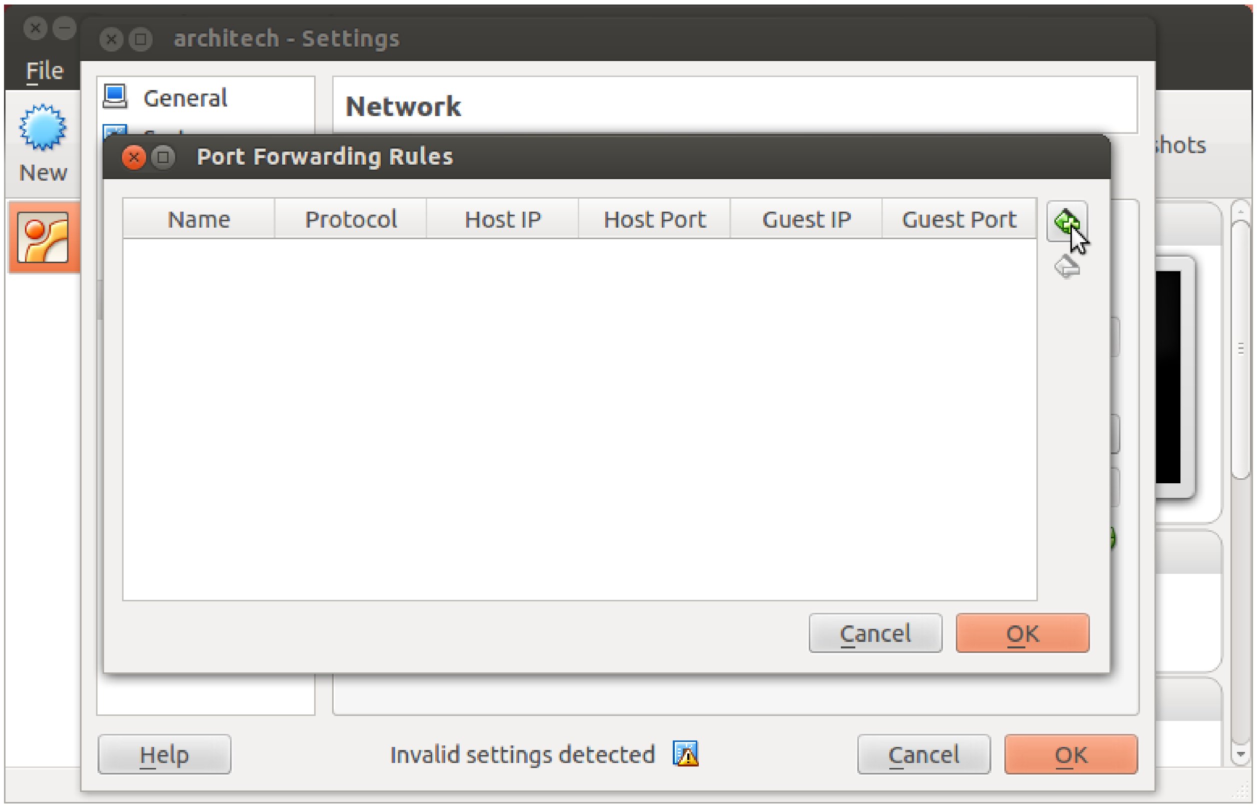
Promiscuous Mode: Deny

MAC Address: 080027B5582E

Cable Connected

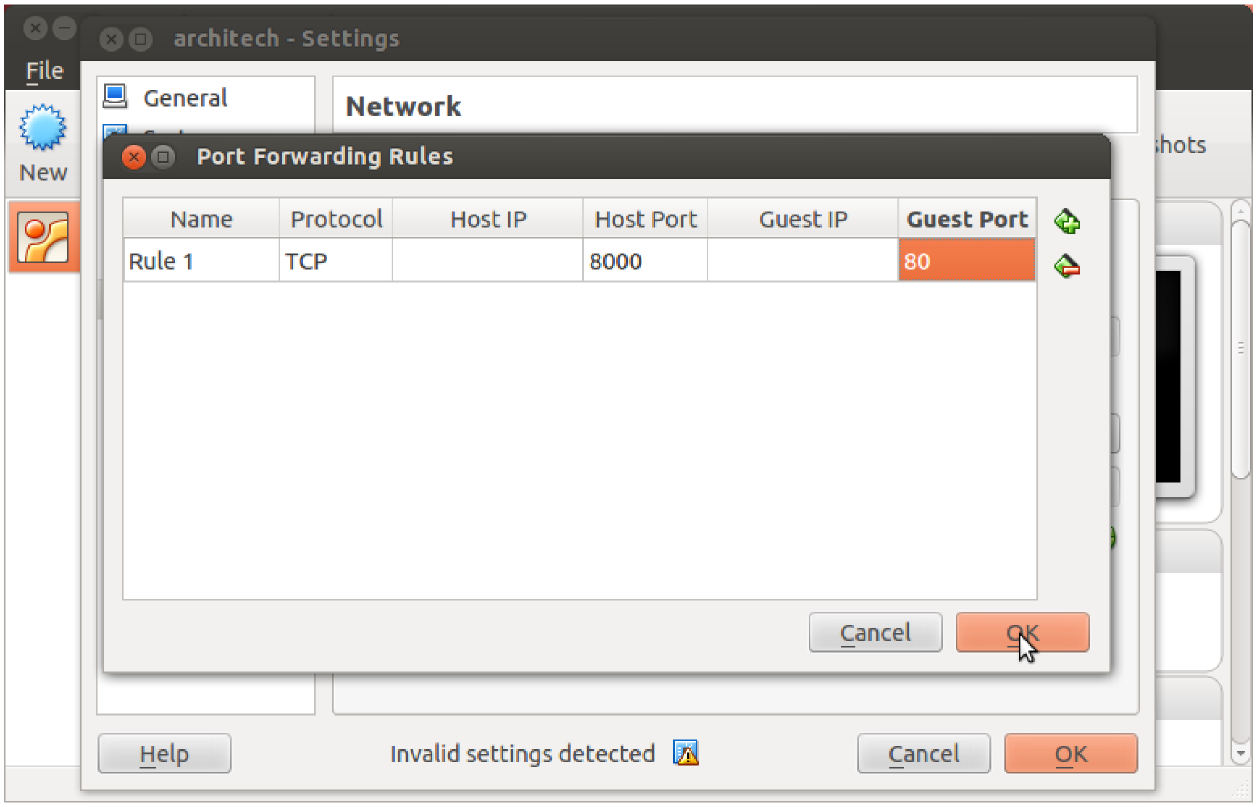
Invalid settings detected

6. Add a new *rule*



Name	Protocol	Host IP	Host Port	Guest IP	Guest Port

7. Configure the *rule*



Name	Protocol	Host IP	Host Port	Guest IP	Guest Port
Rule 1	TCP		8000		80

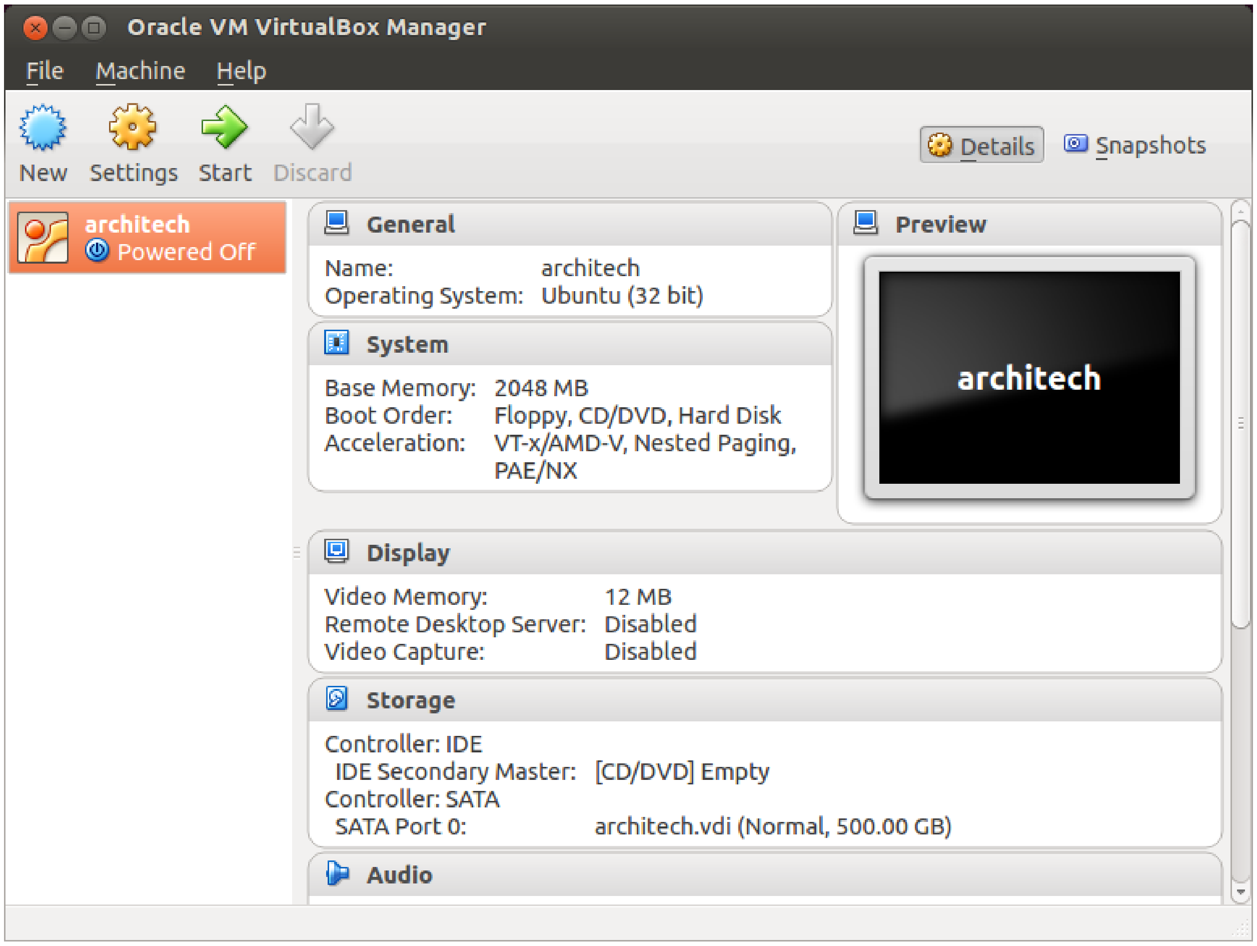
8. Click on *Ok*

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



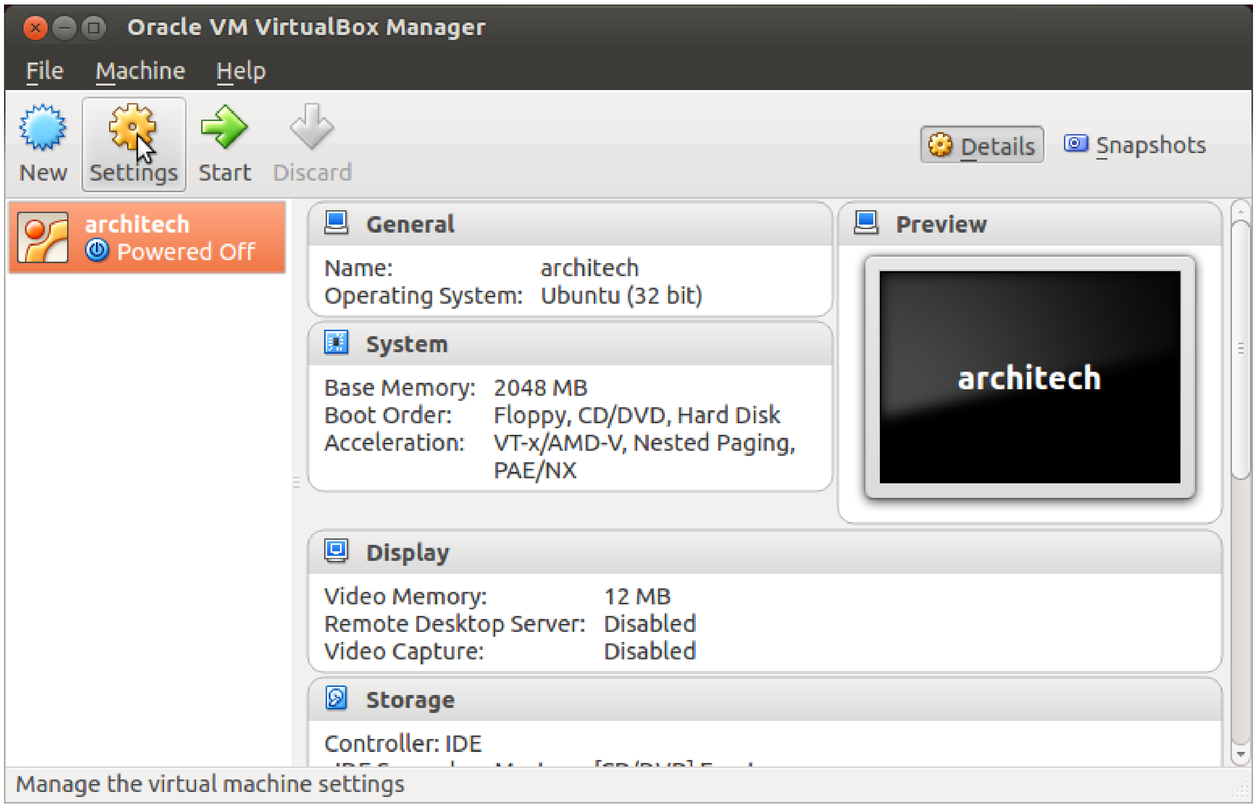
Name: architech
Operating System: Ubuntu (32 bit)

Base Memory: 2048 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX

Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Controller: IDE
 IDE Secondary Master: [CD/DVD] Empty
Controller: SATA
 SATA Port 0: architech.vdi (Normal, 500.00 GB)

2. Click on *Settings*



Name: architech
Operating System: Ubuntu (32 bit)

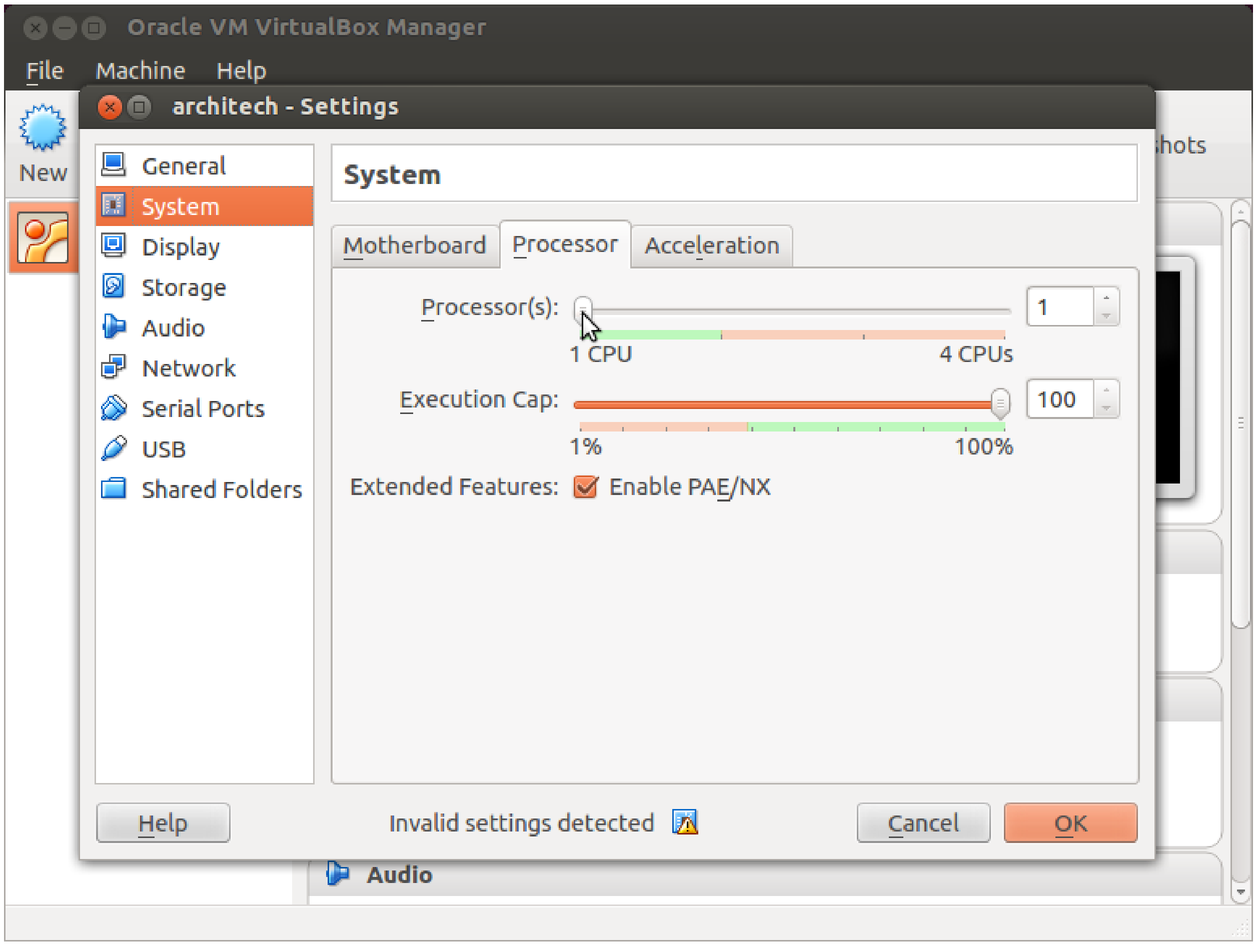
Base Memory: 2048 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX

Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Controller: IDE

Manage the virtual machine settings

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



System

Processor(s): 1 (1 CPU – 4 CPUs)

Execution Cap: 100 (1% – 100%)

Extended Features: ✓ Enable PAE/NX

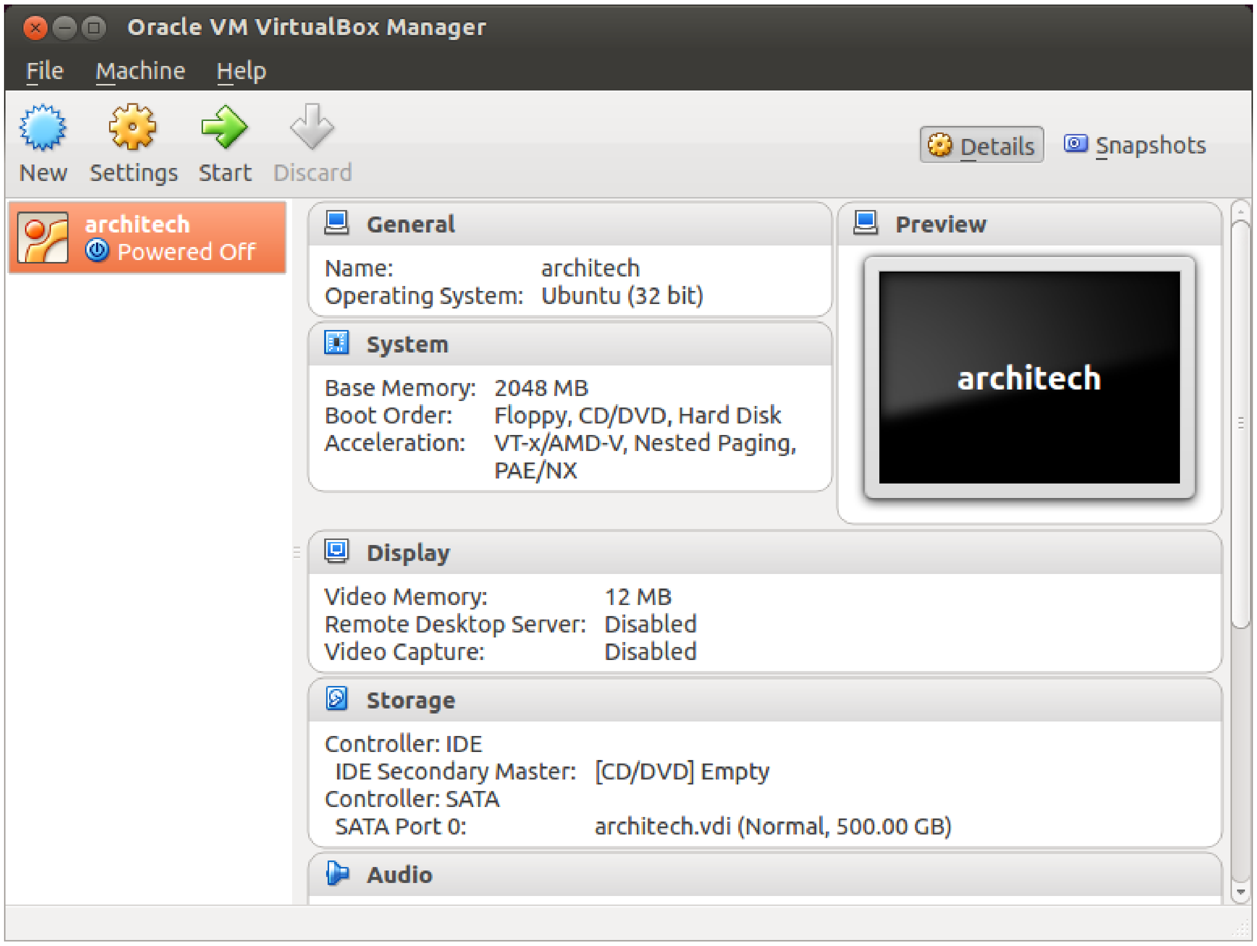
Invalid settings detected

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



General

Name: architech
Operating System: Ubuntu (32 bit)

System

Base Memory: 2048 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX

Display

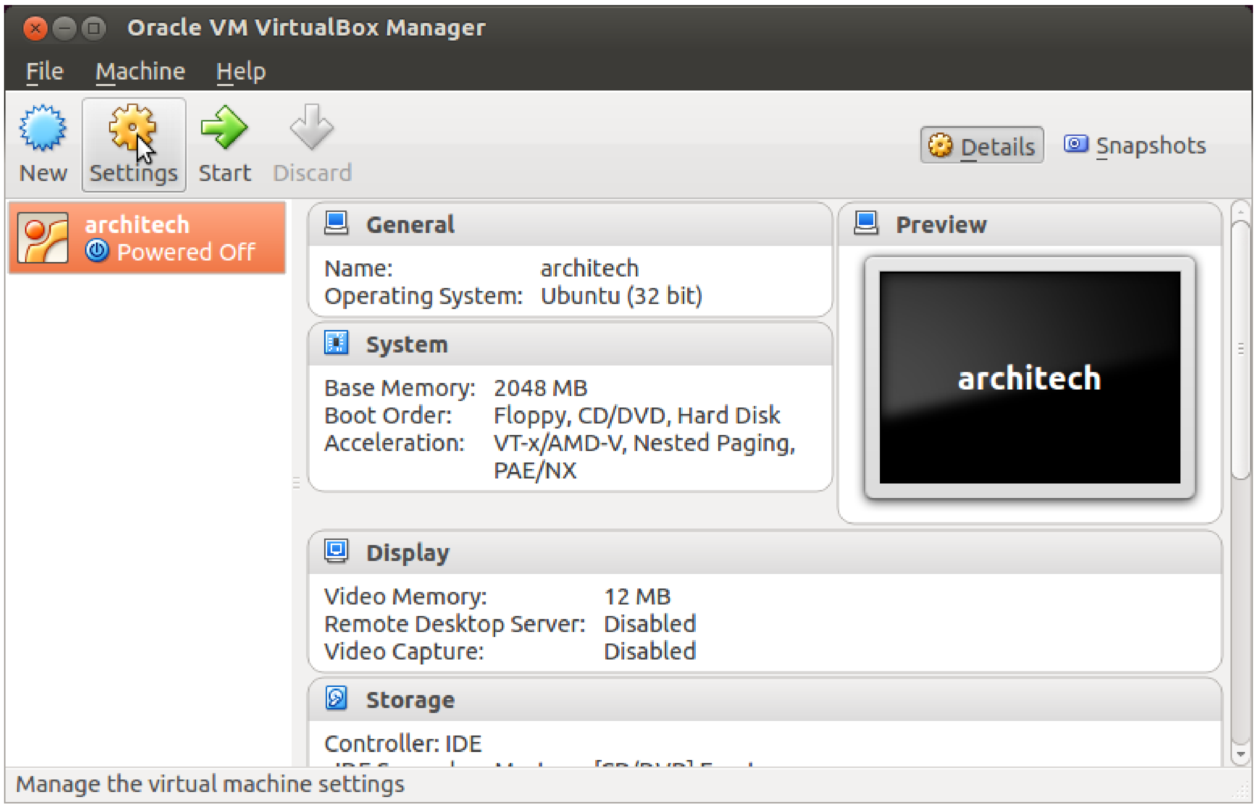
Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Storage

Controller: IDE
IDE Secondary Master: [CD/DVD] Empty
Controller: SATA
SATA Port 0: architech.vdi (Normal, 500.00 GB)

Audio

2. Click on *Settings*



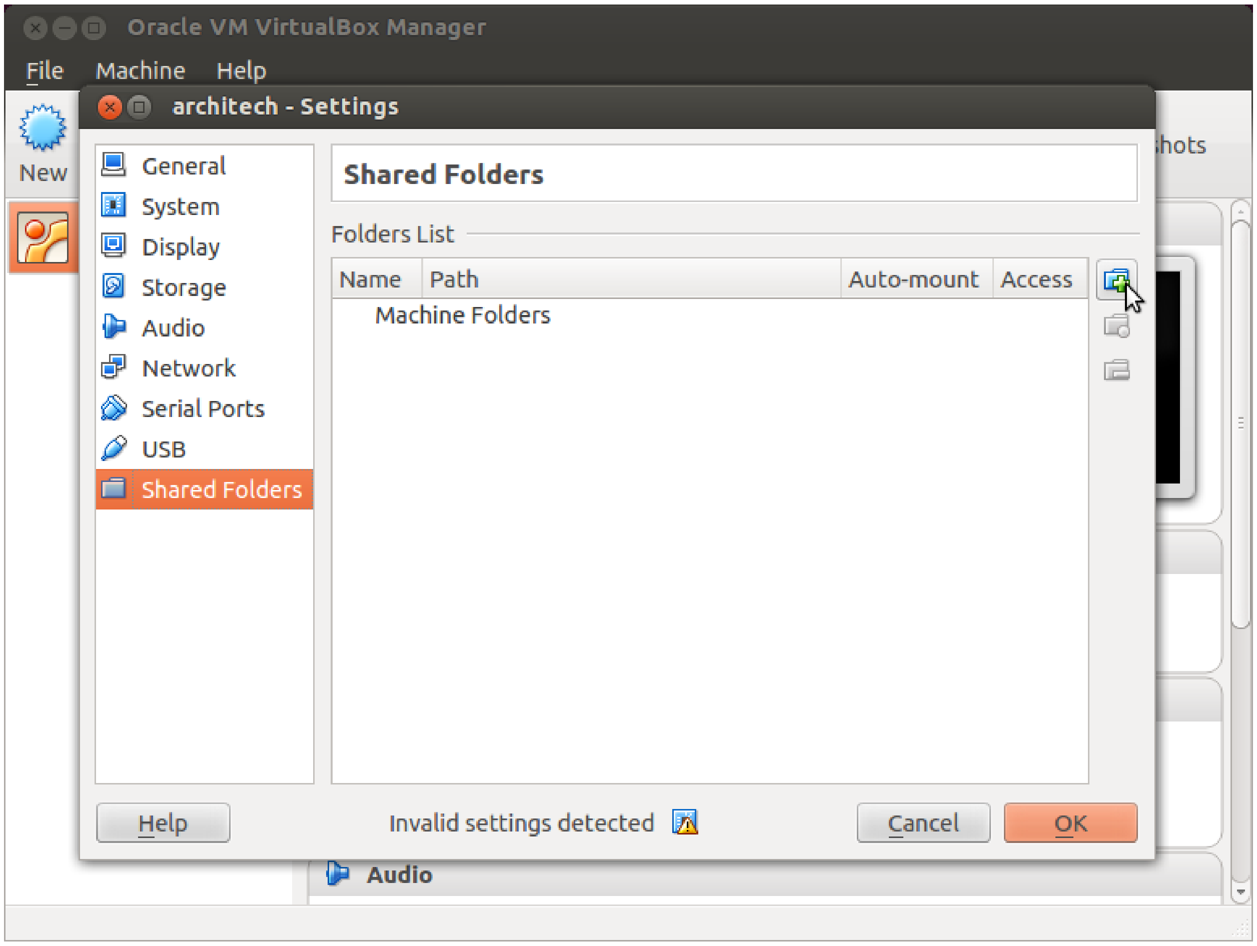
Name: architech
Operating System: Ubuntu (32 bit)

Base Memory: 2048 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX

Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Controller: IDE

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



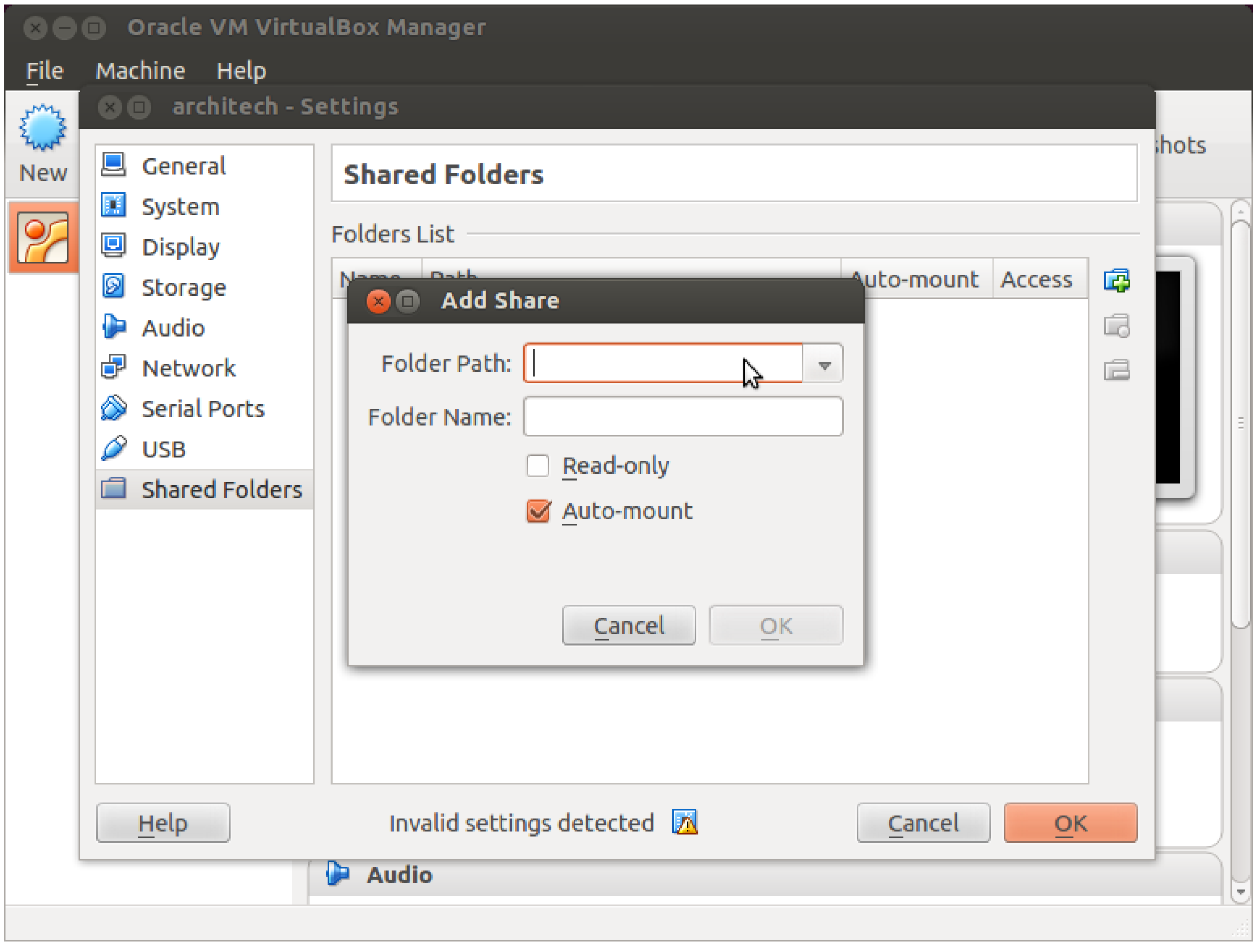
Shared Folders

Folders List

Name	Path	Auto-mount	Access
Machine Folders			

Invalid settings detected

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

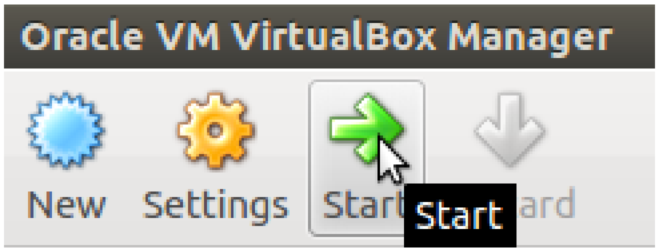


Once the virtual machine has been booted, the shared folder will be mounted under */media/* directory inside the virtual machine.

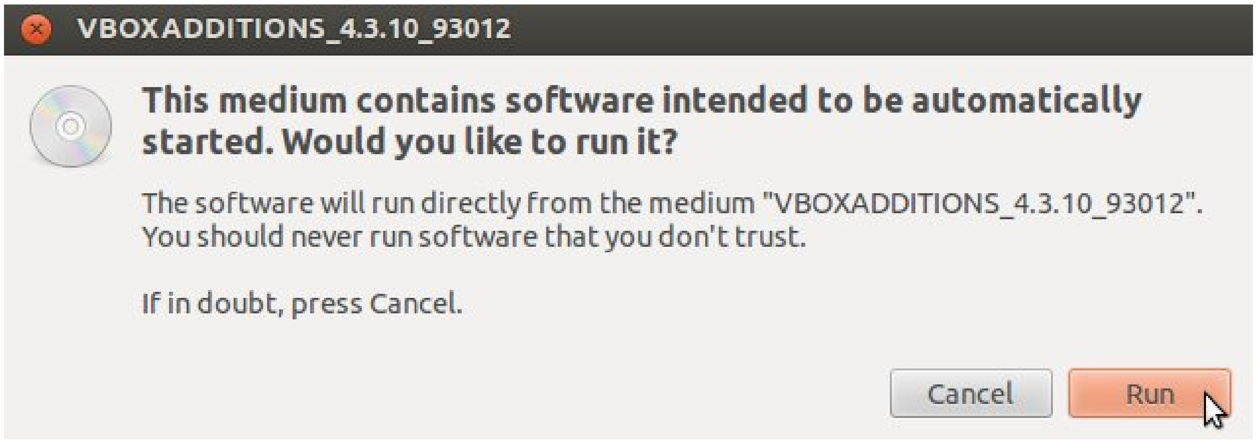
Install VBox Additions

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



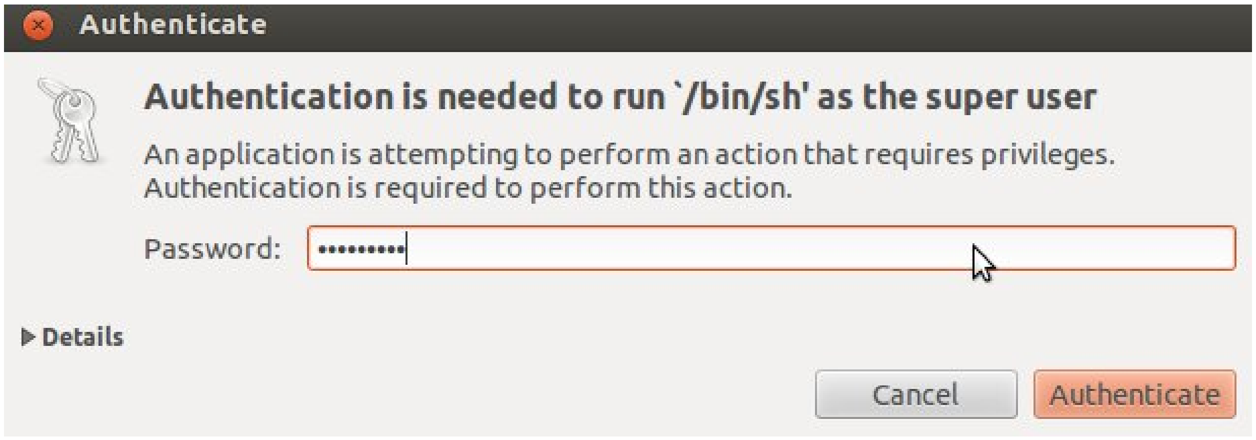
VBOXADDITIONS_4.3.10_93012

This medium contains software intended to be automatically started. Would you like to run it?

The software will run directly from the medium "VBOXADDITIONS_4.3.10_93012". You should never run software that you don't trust.

If in doubt, press Cancel.

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



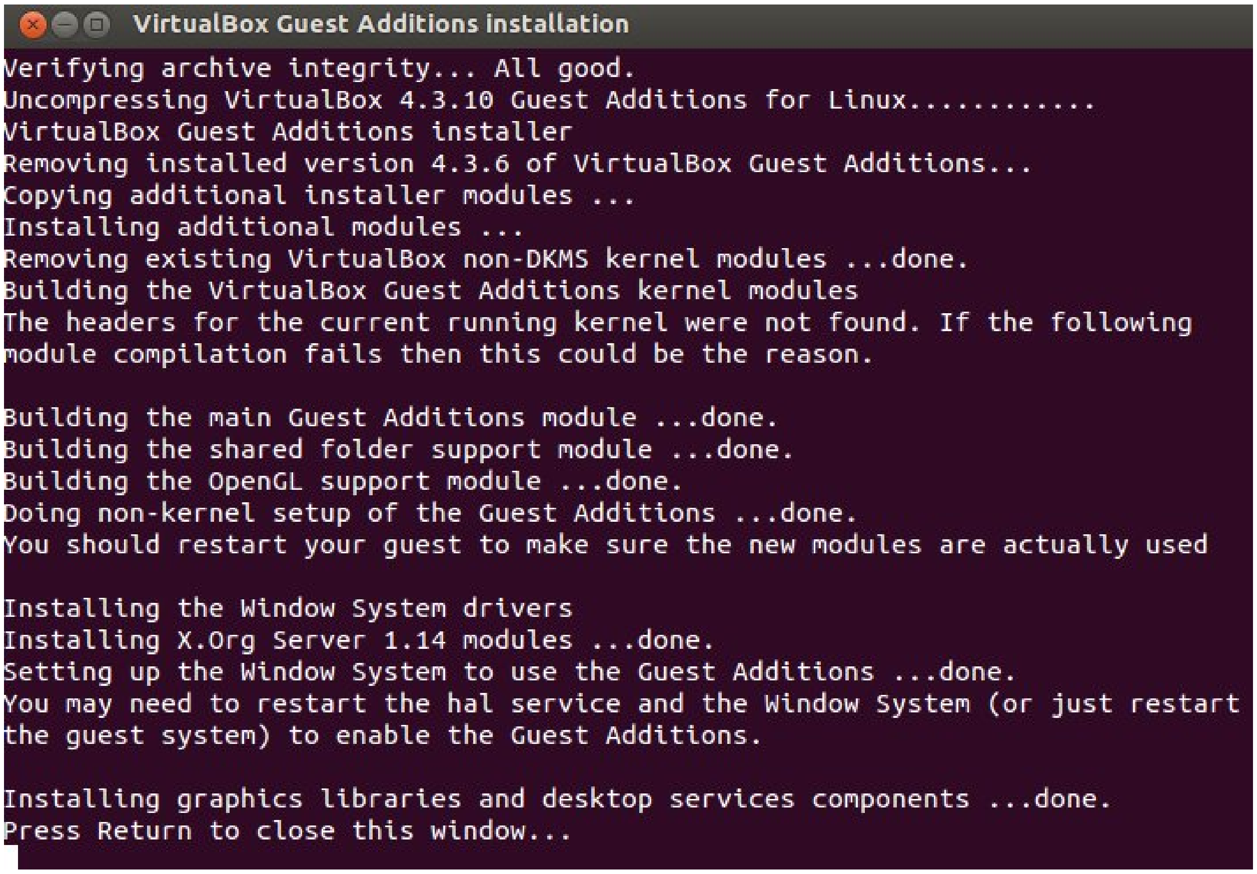
Authenticate

Authentication is needed to run `/bin/sh' as the super user

An application is attempting to perform an action that requires privileges. Authentication is required to perform this action.

Password: ••••••••••

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



```
Verifying archive integrity... All good.
Uncompressing VirtualBox 4.3.10 Guest Additions for Linux...........
VirtualBox Guest Additions installer
Removing installed version 4.3.6 of VirtualBox Guest Additions...
Copying additional installer modules ...
Installing additional modules ...
Removing existing VirtualBox non-DKMS kernel modules ...done.
Building the VirtualBox Guest Additions kernel modules
The headers for the current running kernel were not found. If the following
module compilation fails then this could be the reason.

Building the main Guest Additions module ...done.
Building the shared folder support module ...done.
Building the OpenGL support module ...done.
Doing non-kernel setup of the Guest Additions ...done.
You should restart your guest to make sure the new modules are actually used

Installing the Window System drivers
Installing X.Org Server 1.14 modules ...done.
Setting up the Window System to use the Guest Additions ...done.
You may need to restart the hal service and the Window System (or just restart
the guest system) to enable the Guest Additions.

Installing graphics libraries and desktop services components ...done.
Press Return to close this window...
```

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:

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

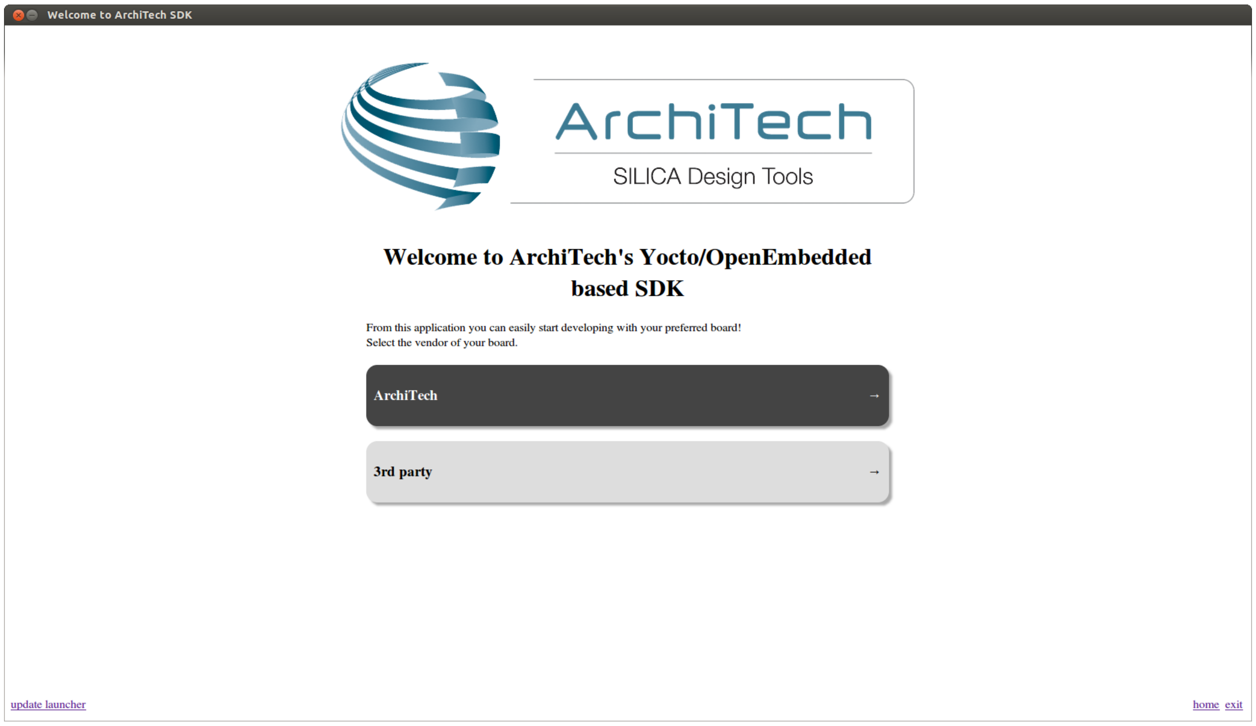
- *eclipse* directory is where Eclipse IDE has been installed
- *java* directory is where the Java Virtual Machine has been installed (needed by Eclipse)
- *qtcreator* contains the installation of Qt Creator IDE
- *splashscreen* directory contains information and scripts used by the splash screen application,
- *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 Tibidabo 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 Tibidabo, choose **ArchiTech**.

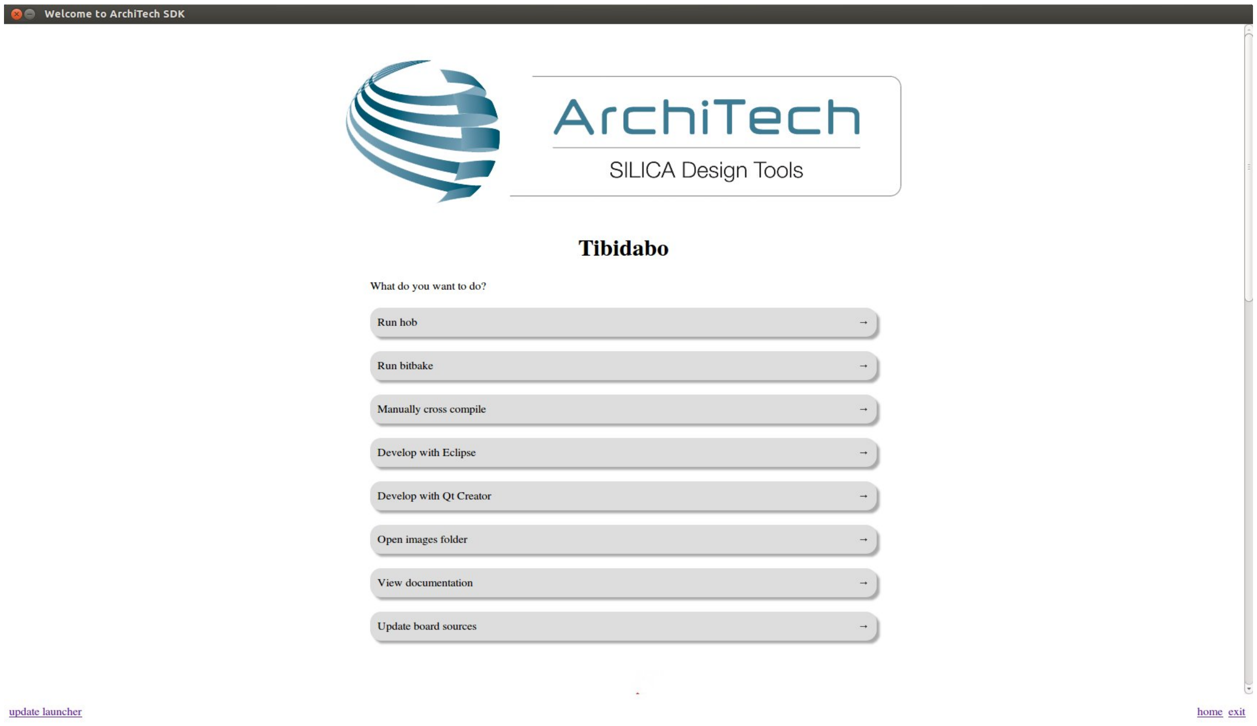


Welcome to ArchiTech's Yocto/OpenEmbedded based SDK

From this application you can easily start developing with your preferred board!
Select the vendor of your board.

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

A list of actions related to Tibidabo 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 Tibidabo 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 Tibidabo:

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 tibidabo 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-tibidabo*:

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

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

Bitbake will place *u-boot* sources under:

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

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

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

and by properly patching the sources:

Now that you have the sources, you can start browsing the code from the following files:

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

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

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

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

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

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

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

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

Once the build process is complete, you will find **u-boot.imx** file in your sources directory, that's the file you need to boot the board.

Linux Kernel

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

Bitbake will place the sources under directory:

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

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

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

and by properly patching the sources:

Now that you have the sources, you can start browsing the code from the following files:

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

and to compile it:

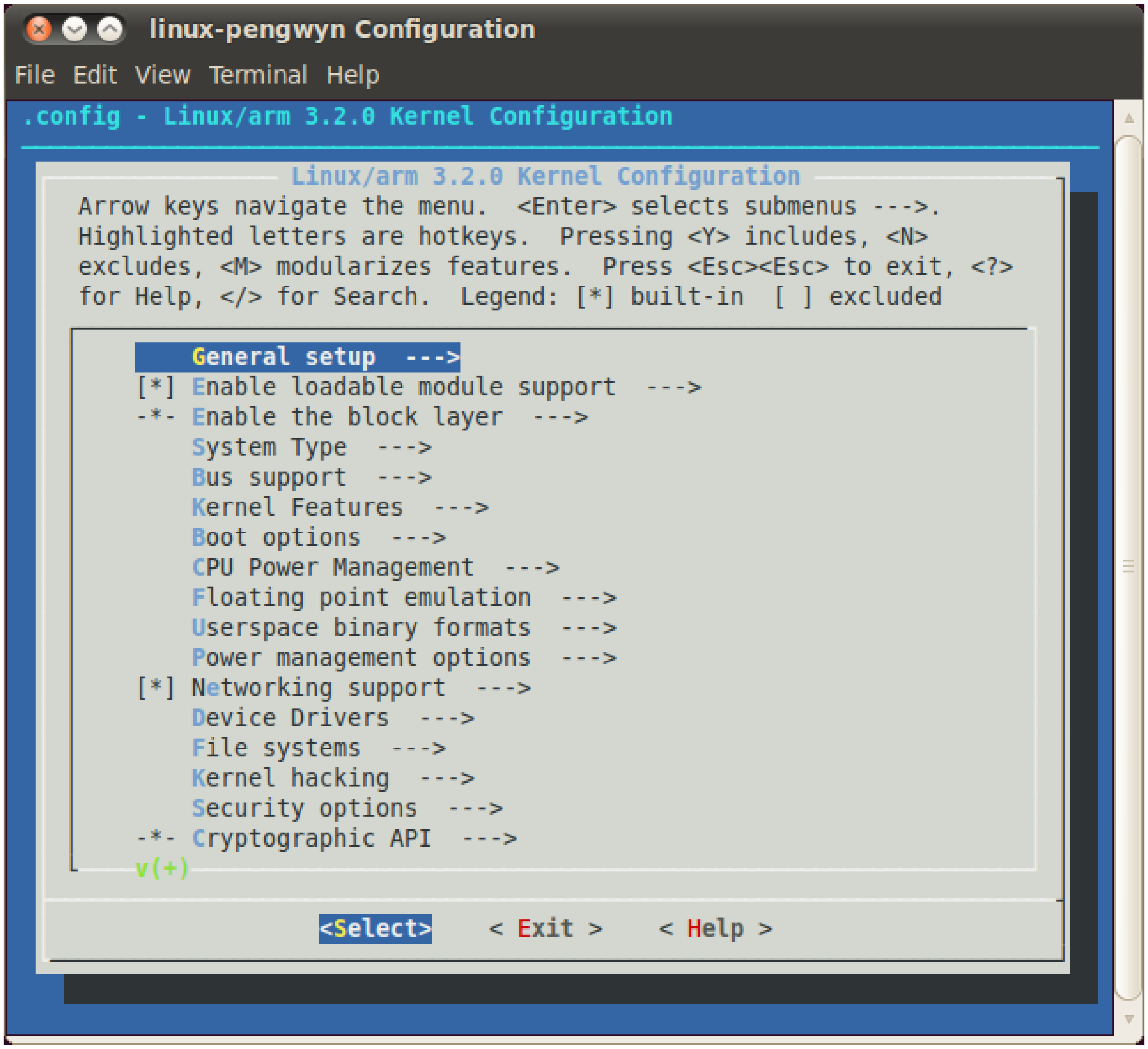
If you omit *-j* parameter, *make* will run one task after the other, if you specify it *make* will parallelize the tasks execution while respecting the dependencies between them. Generally, you will place a value for *-j* parameter corresponding to the double of your processor's cores number, for example,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

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

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:



```
.config - Linux/arm 3.2.0 Kernel Configuration

                  Linux/arm 3.2.0 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N>
  excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?>
  for Help, </> for Search.  Legend: [*] built-in  [ ] excluded

        General setup  --->
    [*] Enable loadable module support  --->
    -*- Enable the block layer  --->
        System Type  --->
        Bus support  --->
        Kernel Features  --->
        Boot options  --->
        CPU Power Management  --->
        Floating point emulation  --->
        Userspace binary formats  --->
        Power management options  --->
    [*] Networking support  --->
        Device Drivers  --->
        File systems  --->
        Kernel hacking  --->
        Security options  --->
    -*- Cryptographic API  --->
    v(+)

              <Select>    < Exit >    < Help >
```

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/tibidabo/* 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/tibidabo/yocto/build/tmp/deploy/images/tibidabo/*.

Meta Layer

A Yocto/OpenEmbedded meta-layer is a directory that contains recipes, configuration files, patches, etc., all needed by *Bitbake* to properly "see" and build a BSP, a distribution, a (set of) package(s), whatever. **meta-tibidabo** is a meta-layer which defines the customizations to make to Freescale's i.MX6 BSP and Yocto/OpenEmbedded in order to get a working system, tailor made of Tibidabo.

You can get it with *git*:

The machine name for Tibidabo is **tibidabo**.

The strictly BSP related recipes are located under:

The other recipes are there just to customize other aspects of the system or to offer some facility to help you easily manage some task, for example, working with flash memory or partitions.

Tibidabo is powered by a big serial NOR memory, big enough to place a full featured root file system inside of it. However, you might not be interested in how to place the file system inside of it from the beginning and how to mount and unmount it inside your file system. There is a recipe inside meta-tibidabo, **tibidabo-flash-utils**, that will install three scripts inside the target file system to make the aforementioned tasks easy:

- *tibidabo_fs2flash*
- *tibidabo_mount_flash*
- *tibidabo_umount_flash*

tibidabo_fs2flash takes as input a *.tar.bz2* file, cleans and formats the flash memory, and finally takes the file you gave him to setup the root file system. For more information just run:

from Tibidabo shell.

tibidabo_mount_flash lets you mount the flash memory partition inside your filesystem (under */mnt/flash*) without any effort and, likewise, *tibidabo_umount_flash* helps you unmounting the partition.

Remember that to install those scripts inside the target, you need to add **meta-openmbedded/meta-oe** meta layer to your *bblayers.conf* file. If you are working with Architech virtual machine, you don't have to worry about that, everything is already in place.

tibidabo-flash-utils won't be placed by default inside your file system, if you want it you need to add a line like this one to your *local.conf* file

Probably the most comfortable way, at least at the beginning, to build a valid SD card or SATA disk is to use file *.sdcard* that *Bitbake* emits when builds an image. However, *Bitbake* prepares a final iso image to write to the medium without any knowledge of its size. If you write the image on an SD card, for example, the first thing you notice is that the file system does not fit the card. How do you resize partitions and file systems to get the best out of your device? You have two possibilities:

1. put your SD card into your computer and use some tool, however, this option is available only on a Linux machines, or
2. resize the file system directly on the target board.

meta-tibidabo has a recipe, **tibidabo-resize-partition**, that puts a script inside the target file system that does **online resizing of the last partition** on the medium (that must be a *primary partition*), which can be an SD card, an mSATA hard disk, or an USB memory stick. The script name is **tibidabo_resize_partition**, to see the help just type:

on Tibidabo's console.

An example for resizing the SD card iso image generated by *Bitbake*, can be:

then follow the instructions, if any.

Even *tibidabo-resize-partition* won't be placed by default inside the final root file system, unless you asks *Bitbake* for it, by adding the following line to your build directory *local.conf* file:

Root FS

By default, Tibidabo's Yocto/OpenEmbedded SDK will generate three different types of files when you build an image:

- *.ext3*,
- *.tar.bz2*, and
- *.sdcard*.

.ext3 is meant to be used by *QEMU* and won't be discussed here. The *.tar.bz2* file can be flattened out in your final medium partition (on SD card, flash memory, mSATA disk or USB stick) or on your host development system and used for build purposes with the Yocto Project. File *.sdcard* can be written out "as is" on the final medium with, for example, *dd* program:

Where, the path to the image *.sdcard* file inside the SDK virtual machine is:

Warning: Be very careful when you use *dd* to write to a device to pick up the right device, otherwise you can mess up another disk you have on your machine, destroying its content forever!

Warning: The content of the media will be lost forever!

Important: Be sure you **unmount the device** from the filesystem before using **dd** program, you sure don't want to have the operating system interfere during the write process.

After *dd* completes, run:

Generally, especially at the beginning, when you build an image for Tibidabo is more comfortable to create an SD card using the *.sdcard* file, because you need almost zero effort to get everything running. However, if you need to develop for a while on the board this solution turns out to be inefficient, and you will want a faster solution. Assuming you already built an SD card out of a *.sdcard* file, you have an SD card with two partitions on it. The first one is supposed to contain the kernel image (*uImage* file) and the *bootscript* file, the second partition is supposed to contain the root file system. When you build a new file system you can delete everything contained on the second partition and you can untar file *.tar.bz2* to the second partition on the SD card. If you have built a new kernel just overwrite the old one on the first partition. In case you have built a new bootloader take a look at *Bootloader deploy*.

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 Tibidabo is located under:

and the splash screen has a facility (a button located under Tibidabo'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 Tibidabo is tibidabo:

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

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

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

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

Command line

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

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

for example:

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

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

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

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

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

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

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

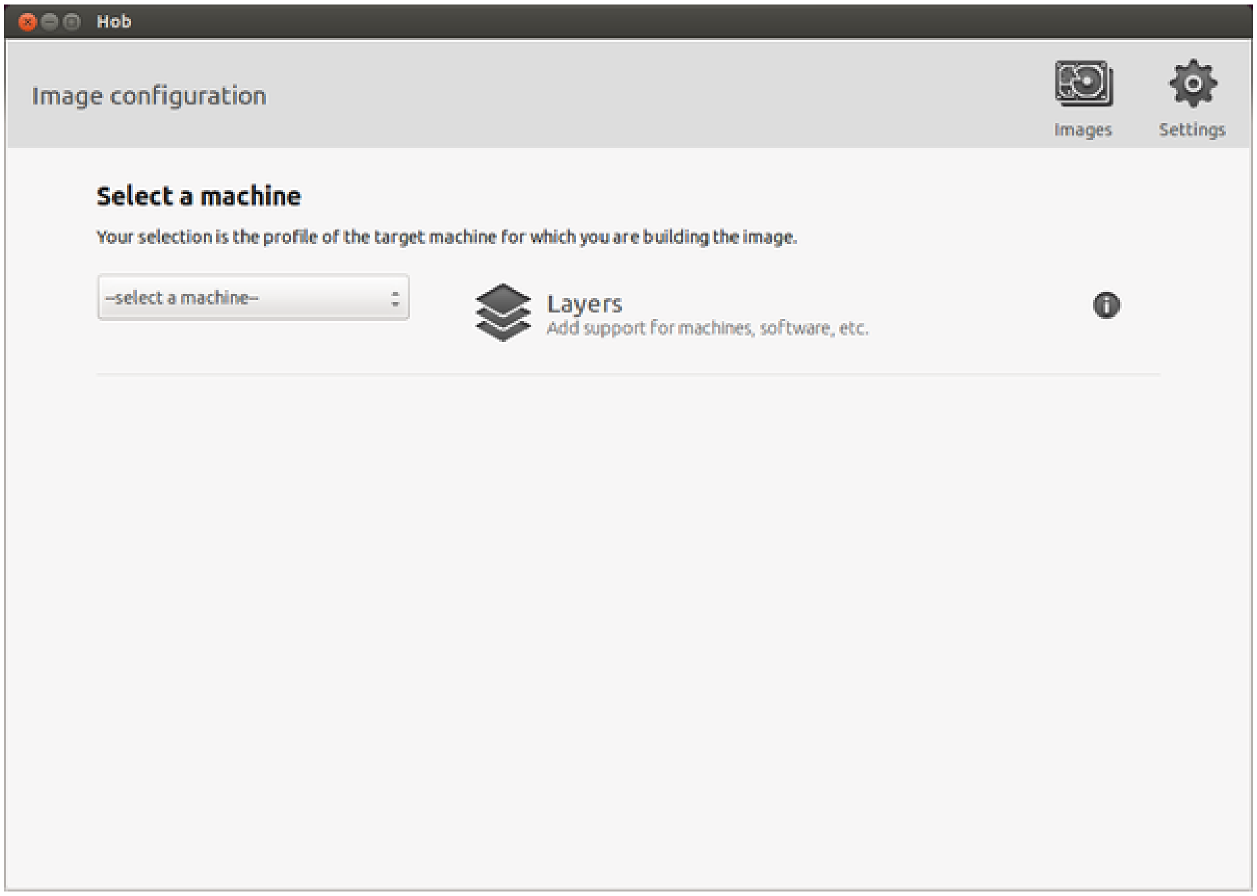
Hob

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



Select a machine

Your selection is the profile of the target machine for which you are building the image.

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 Tibidabo. 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.

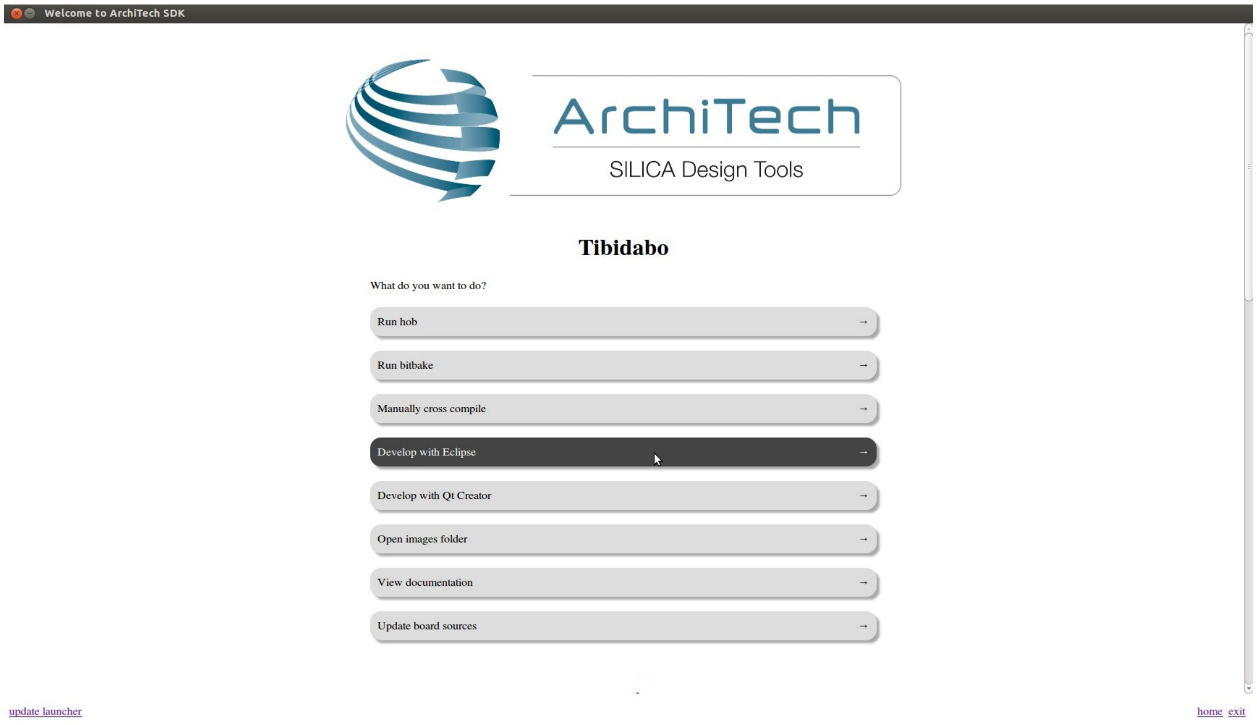
To write your application you need:

- 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
- Tibidabo *powered up* with the aforementioned root file system
- a working *serial console* terminal
- a working *network* connection between your workstation and the board (connector *CN16 Port P0*), so, be sure that:

1. your board has ip address 192.168.0.10 on interface pt0, and
2. your PC has an ip address in the same family of addresses, e.g. 192.168.0.100.

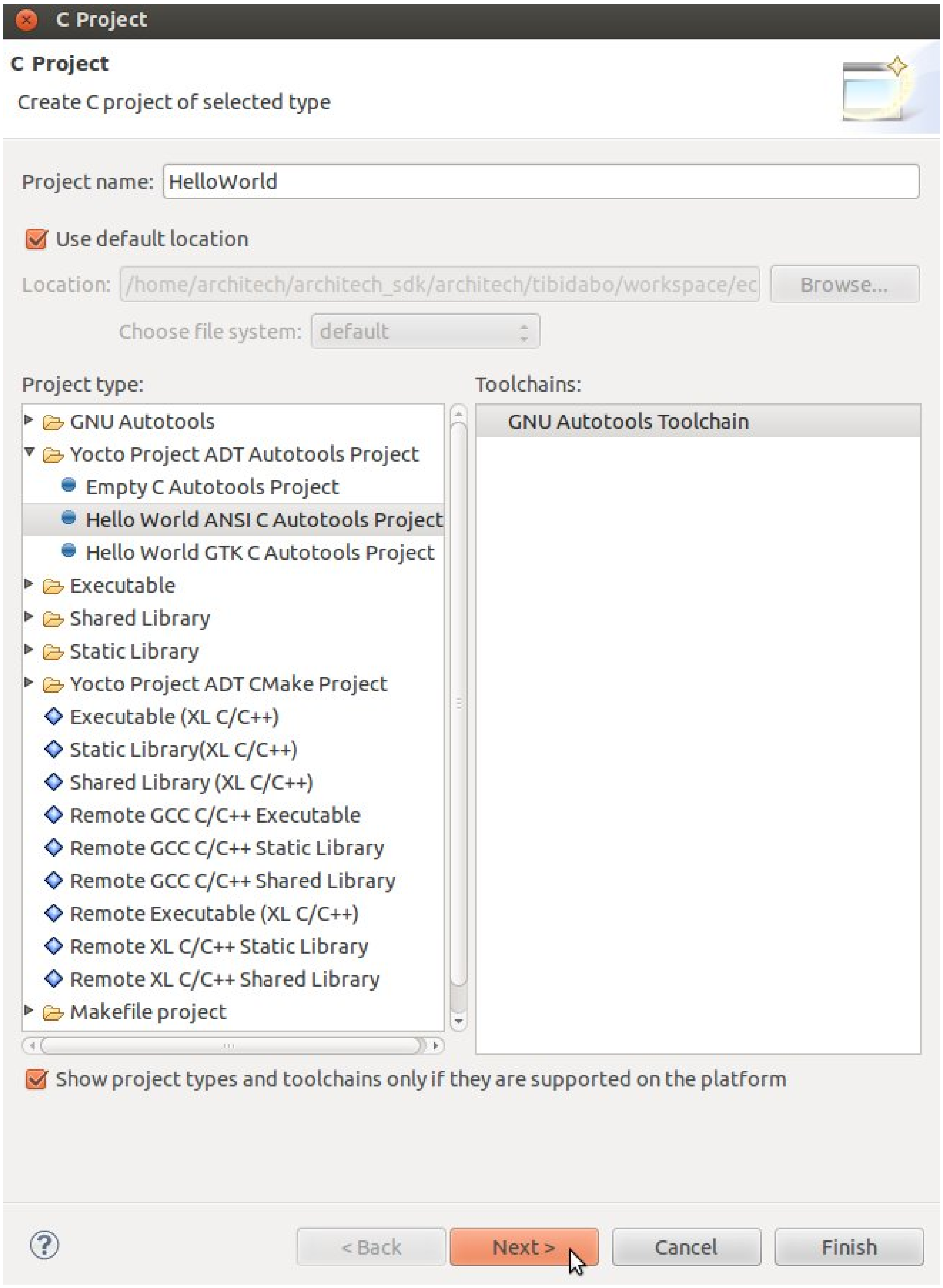
Creating the Project

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/tibidabo/workspace/ec

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

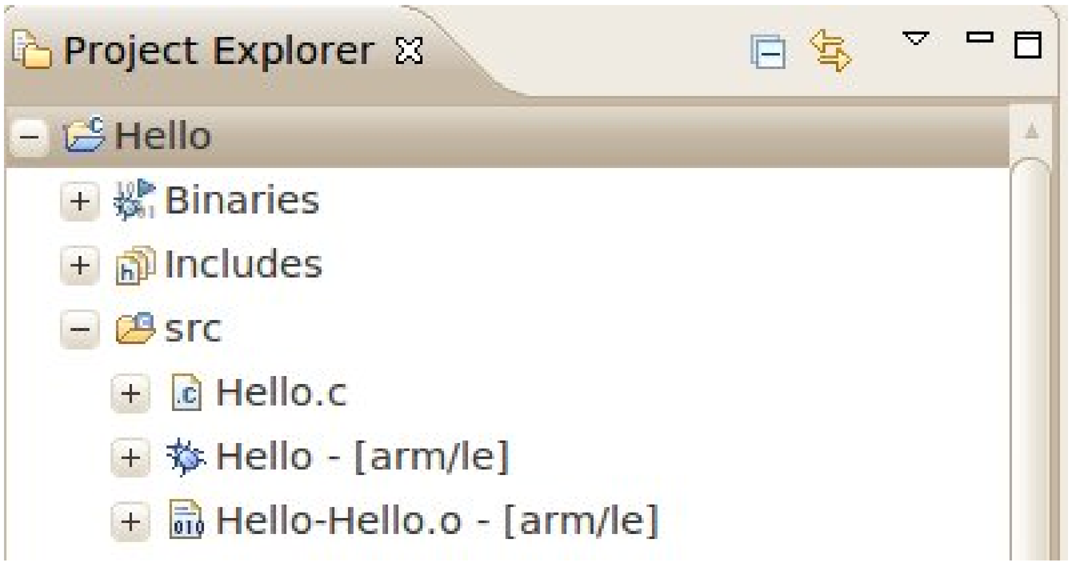
Toolchains:

GNU Autotools Toolchain

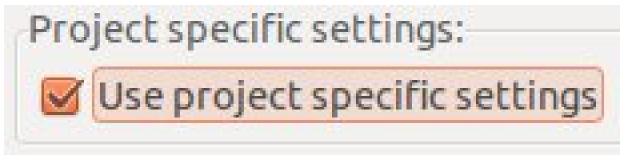
[✓] Show project types and toolchains only if they are supported on the platform

- 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*


Project specific settings:

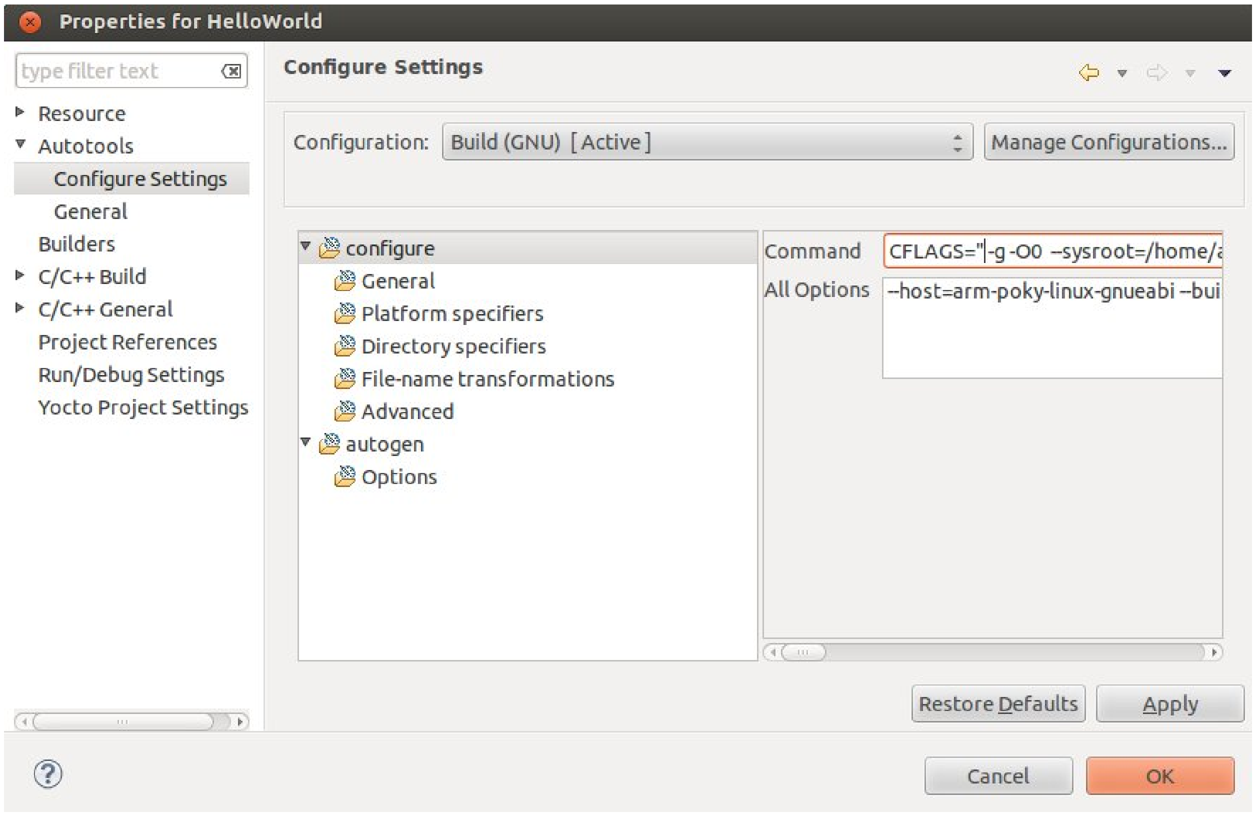
☑ Use project specific settings

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/tibidabo/sysroot* subdirectories.



Configuration: Build (GNU) [Active]

Command CFLAGS="-g -O0 --sysroot=/home/a

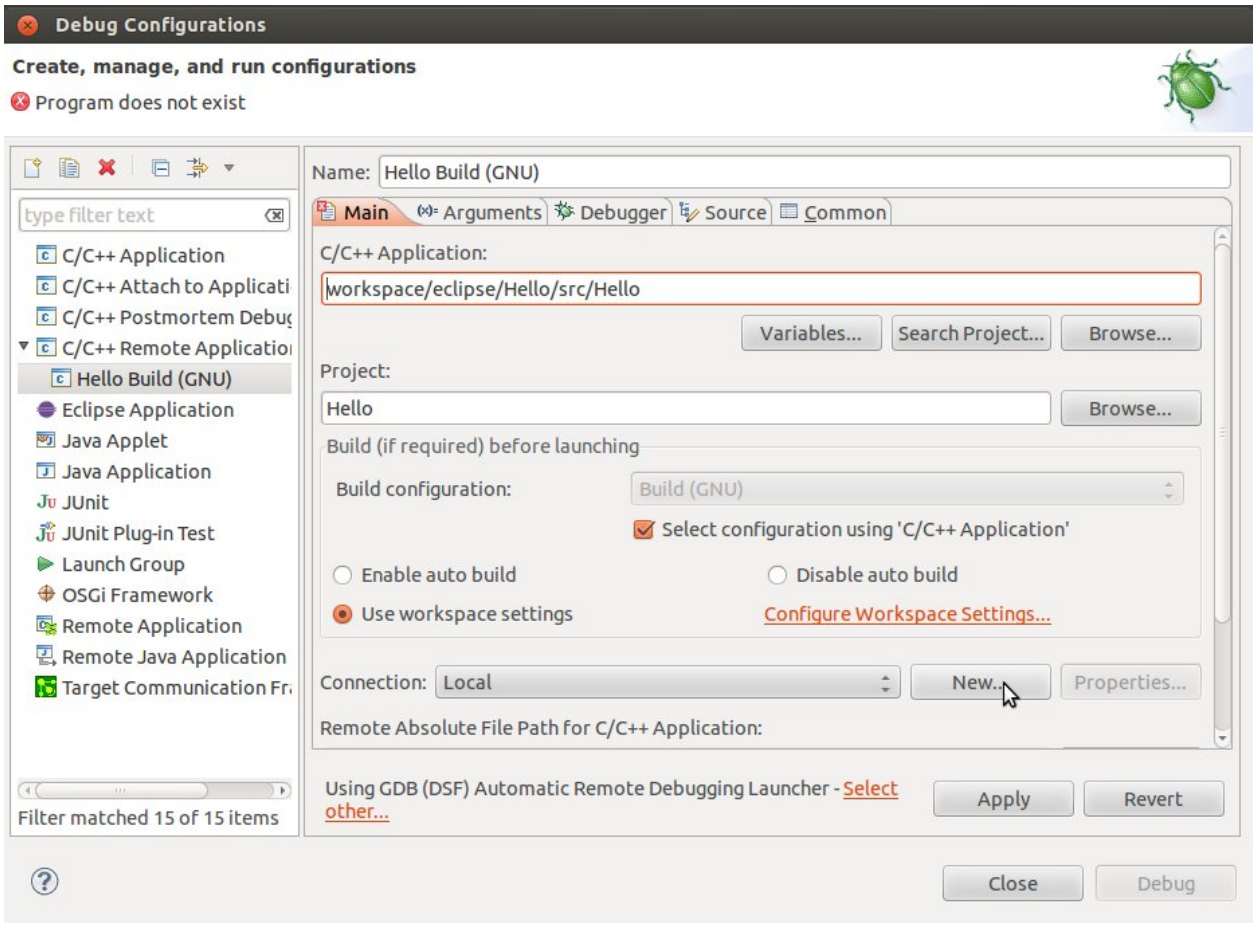
All Options --host=arm-poky-linux-gnueabi --bui

Deploying and Debugging the Application

Connect Tibidabo 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.



Name: Hello Build (GNU)

C/C++ Application: workspace/eclipse/Hello/src/Hello

Project: Hello

Build (if required) before launching

Build configuration: Build (GNU)

[✓] Select configuration using 'C/C++ Application'

Enable auto build · Disable auto build

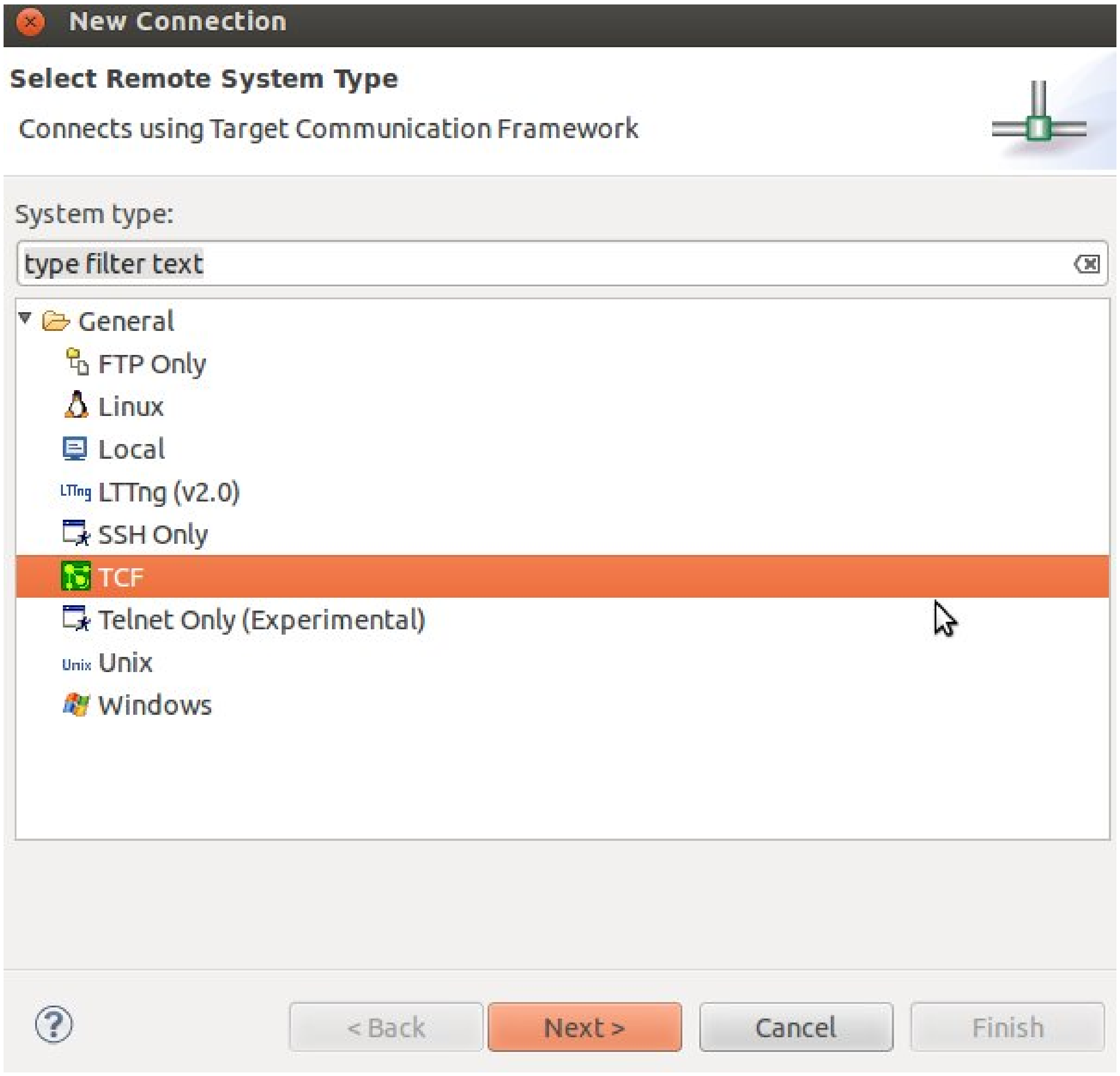
[● Use workspace settings] · Configure Workspace Settings...

Connection: Local

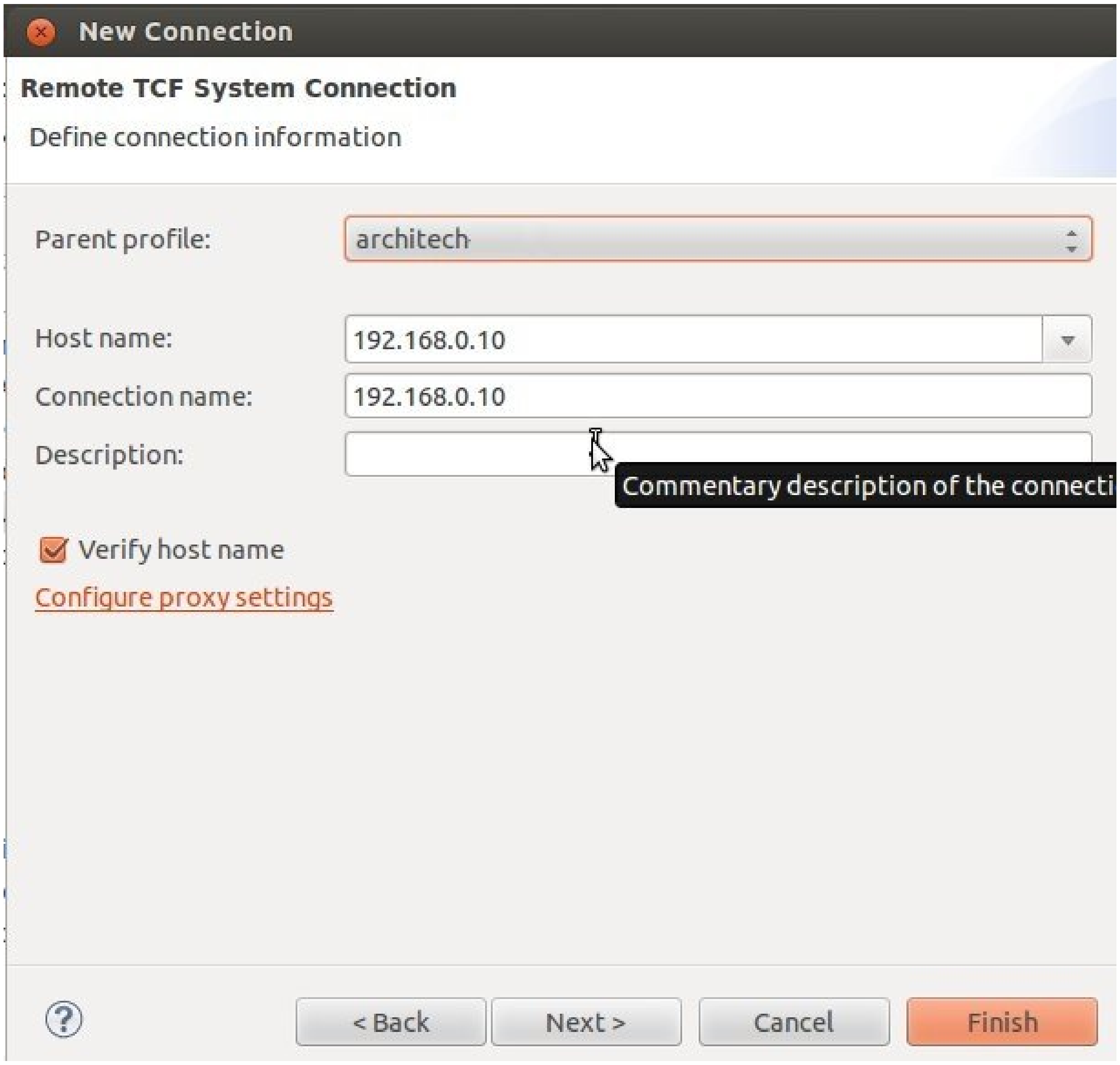
Remote Absolute File Path for C/C++ Application:

Using GDB (DSF) Automatic Remote Debugging Launcher - Select other...

- 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)



Remote TCF System Connection

Define connection information

Parent profile: architech

Host name: 192.168.0.10

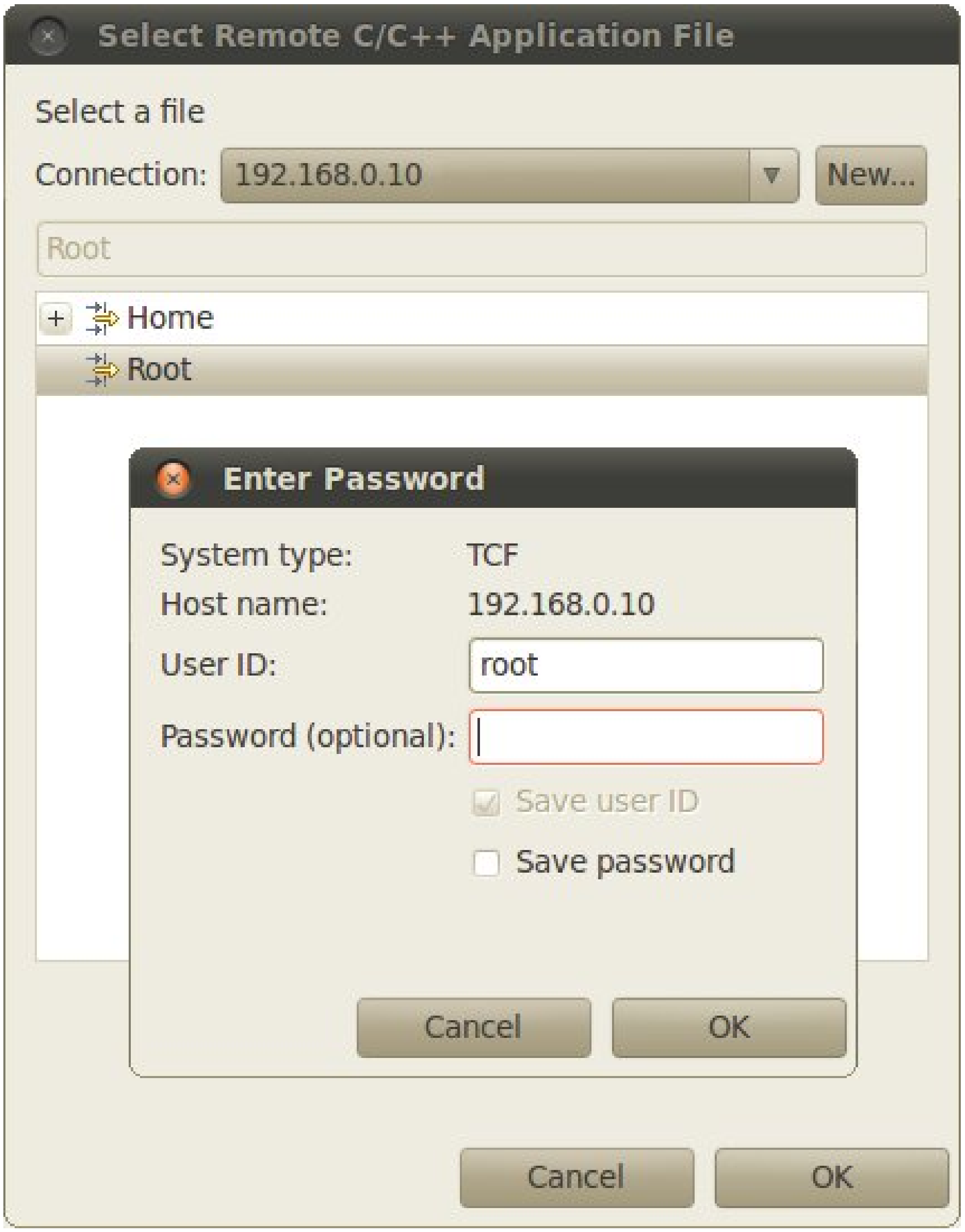
Connection name: 192.168.0.10

Description:

Commentary description of the connecti

✓ Verify host name

- 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.



Connection: 192.168.0.10

Root

System type: TCF

Host name: 192.168.0.10

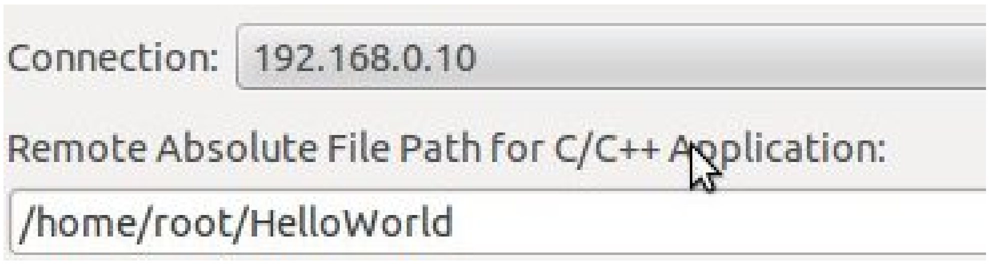
User ID: root

Password (optional):

[✓ Save user ID]

Save password

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

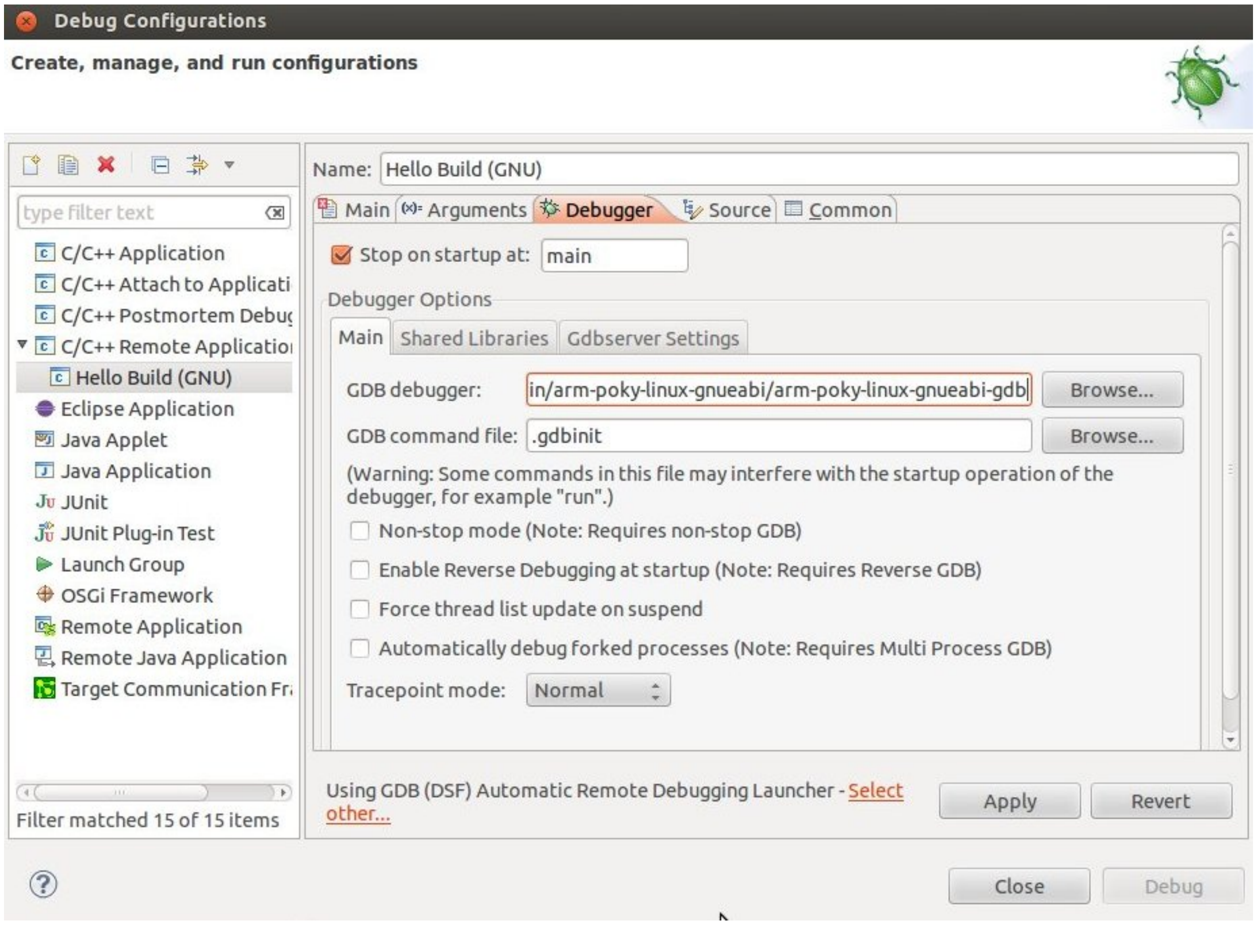


Connection: 192.168.0.10

Remote Absolute File Path for C/C++ Application:

/home/root/HelloWorld

- Select *Debugger* tab



Name: Hello Build (GNU)

[✓] Stop on startup at: main

GDB debugger: in/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gdb

GDB command file: .gdbinit

(Warning: Some commands in this file may interfere with the startup operation of the debugger, for example "run".)

[ ] Non-stop mode (Note: Requires non-stop GDB)

[ ] Enable Reverse Debugging at startup (Note: Requires Reverse GDB)

[ ] Force thread list update on suspend

[ ] Automatically debug forked processes (Note: Requires Multi Process GDB)

Tracepoint mode: Normal

Using GDB (DSF) Automatic Remote Debugging Launcher - Select other...

- 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.

Qt Framework

The Qt Framework used by this SDK is composed of libraries for your host machine and your target. To compile the libraries for *x86* you only need your distribution toolchain, while to compile the libraries for Tibidabo board you need the proper cross-toolchain (see Chapter *Cross compiler* for further information on how to get it).

This section just wants to show you how the framework has been generated.

Before to begin, keep in mind you might need to install the following package to compile yourself the libraries under Ubuntu

:.. host:

```
| sudo apt-get install libxrender-dev
```

So, to install *qt-everywhere* for *x86* from sources, the usual drill of download, uncompress, *configure*, *make* and *make install* is required:

The installation of the libraries for Tibidabo from sources is sligthly more complicated. Once you downloaded and uncompressed the sources

you need to customize *qmake* configuration

save the file and exit from gedit, then *configure*, *make* and *make install*

A comfortable tool to get your job done with Qt is *Qt Creator*, which its use will be introduced in Section *Qt Creator*. You can download it from here:

Tip: http://sourceforge.net/projects/qtcreator.mirror/files/Qt%20Creator%202.8.1/qt-creator-linux-x86-opensource-2.8.1.run/download

Qt Creator

Qt is a cross-platform application framework that is used to build applications. One of the best features of Qt is its capability of generating Graphical User Interfaces (GUIs).
Qt Creator is a cross-platform C++ IDE which includes a visual debugger, an integrated GUI layout and form designer. It makes possible to compile and debug applications on both **x86** (host) and **ARM** (target) machines.
This SDK relies on **version 4.8.5** of Qt and **version 2.8.1** of Qt Creator.

Before getting our hands dirty, make sure all these steps have been followed:

1. Use *Hob* or *Bitbake* to build an image which includes: *openssh*, support for C++, *tcf-agent* and *gdbserver*.

Note: To follow this guide build *qt4e-demo-image* image. Remember to complete its file system with *tcf-agent*, *gdbserver* and *openssh*.

2. Deploy the *root file system* just generated on the final media used to boot the board
3. Replicate the same root file system into directory
4. Copy the Qt Libraries to the board media used to boot
5. Copy the Qt Libraries and cpp libraries to your sdk sysroot directory
6. Unmount the media used to boot the board from your computer and insert it into the board
7. *Power-On* the board
8. Open up the *serial console*.

If you based your root file system on *qt4e-demo-image*, be sure you execute this command

to stop the execution of the demo application.

9. Provide a working *network* connection between your workstation and the board (connector *CN16 Port P0*), so, be sure that:

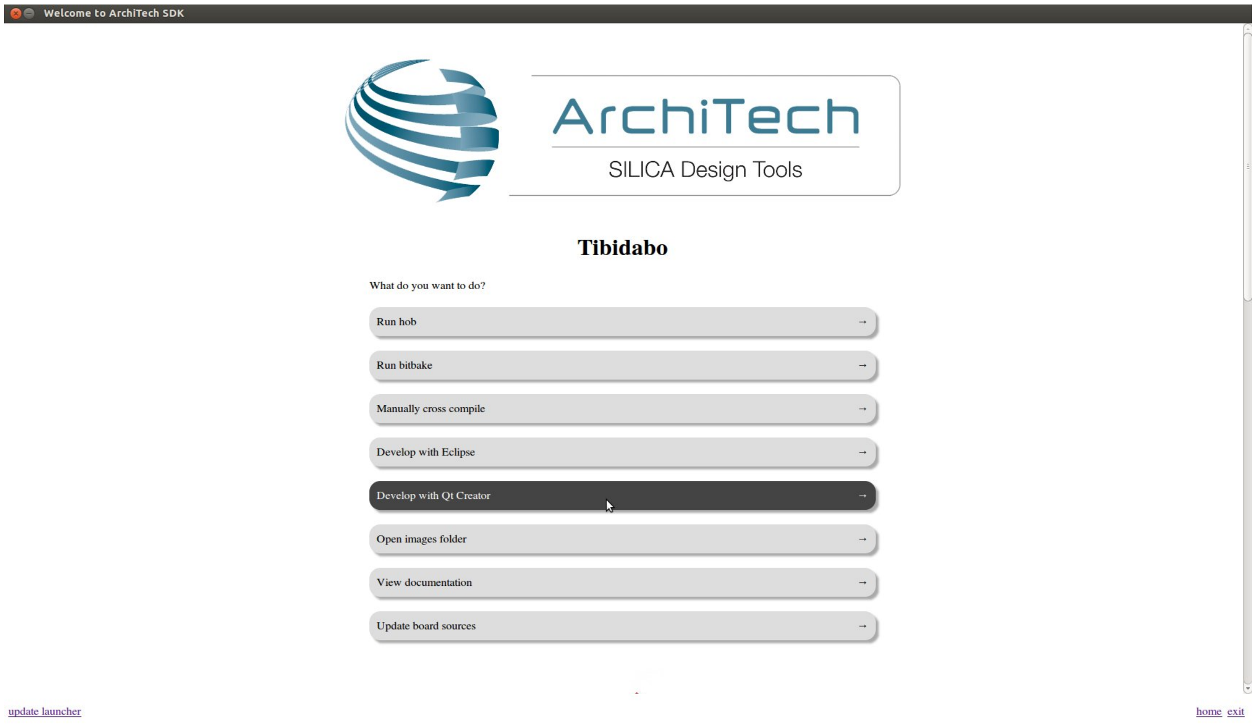
1. your board has ip address 192.168.0.10 on interface pt0, and
2. your PC has an ip address in the same family of addresses, e.g. 192.168.0.100.

Hello World!

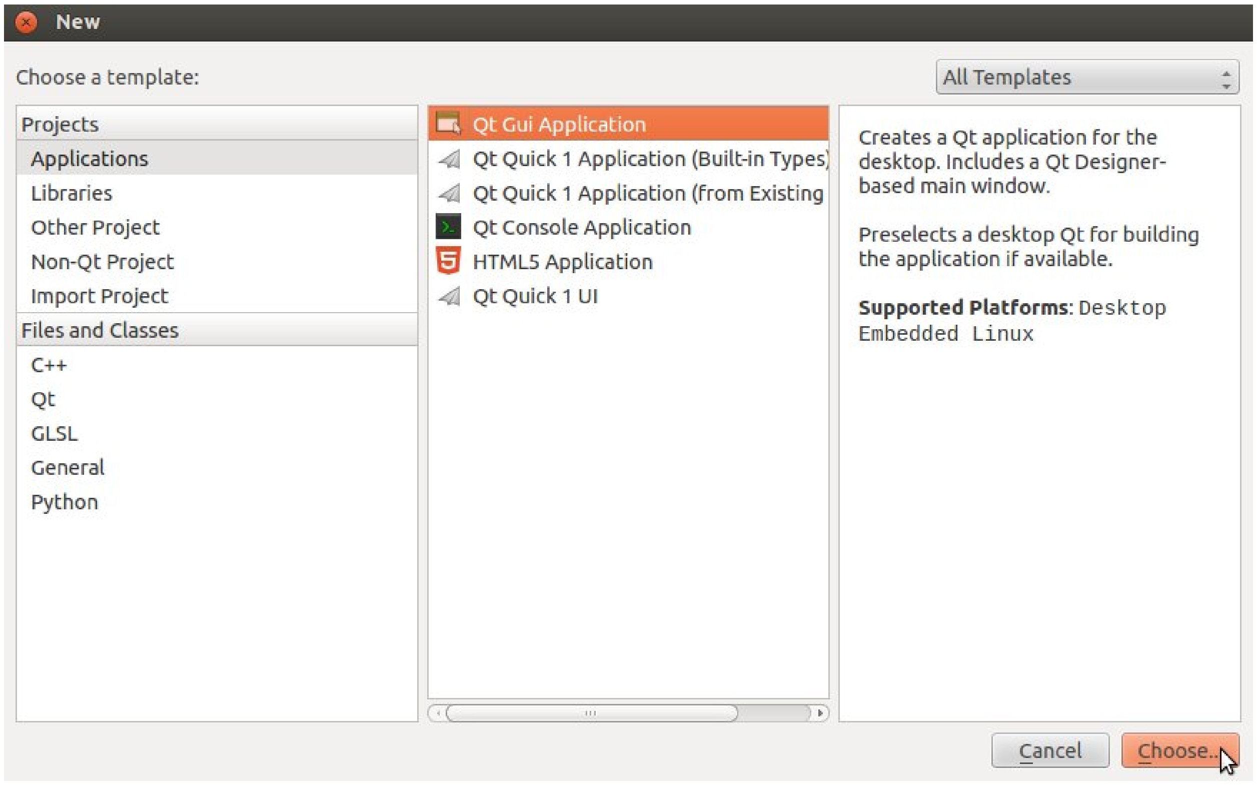
The purpose of this example project is to generate a form with an "Hello World" label in it, at the beginning on the x86 virtual machine and than on Tibidabo board.

To create the project follow these steps:

1. Use the **Welcome Screen** to run Qt Creator by selecting *Architech→Tibidabo→Develop with Qt Creator*



2. Go to *File -> New File or Project*. In the new window select *Applications* as project and *Qt Gui Application*. Click on *Choose...* button.

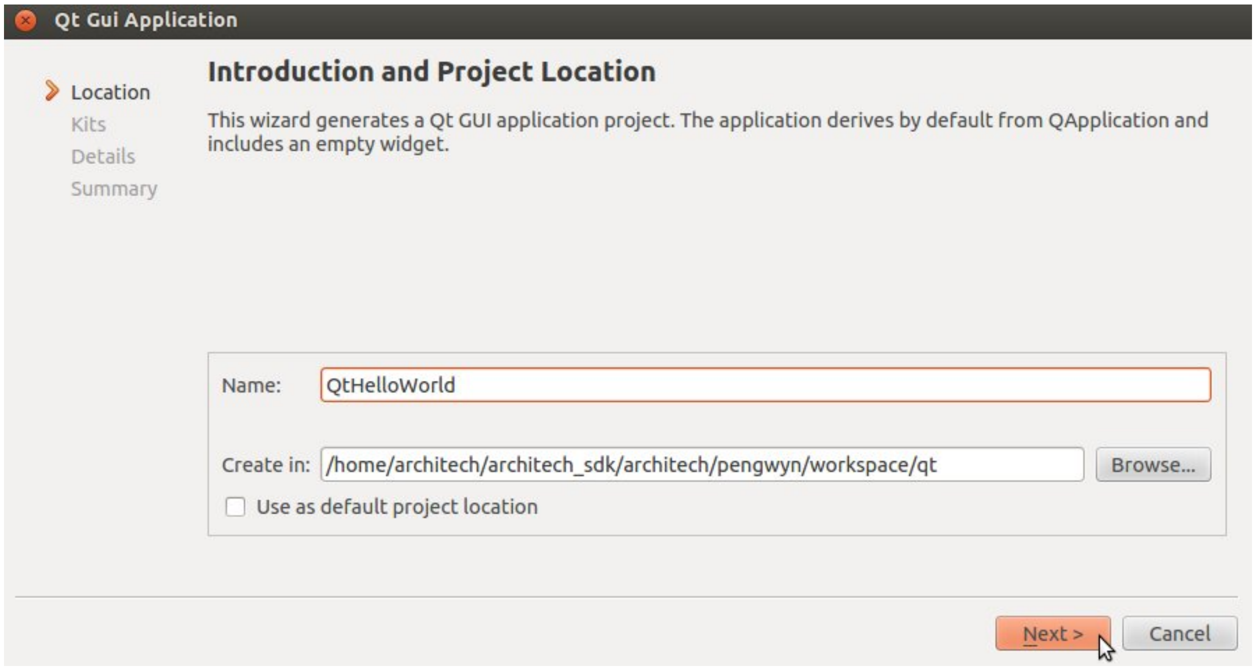


Creates a Qt application for the desktop. Includes a Qt Designer-based main window.

Preselects a desktop Qt for building the application if available.

Supported Platforms: Desktop Embedded Linux

3. Select a name for your project for example *QtHelloWorld* and press *next* button.



Introduction and Project Location

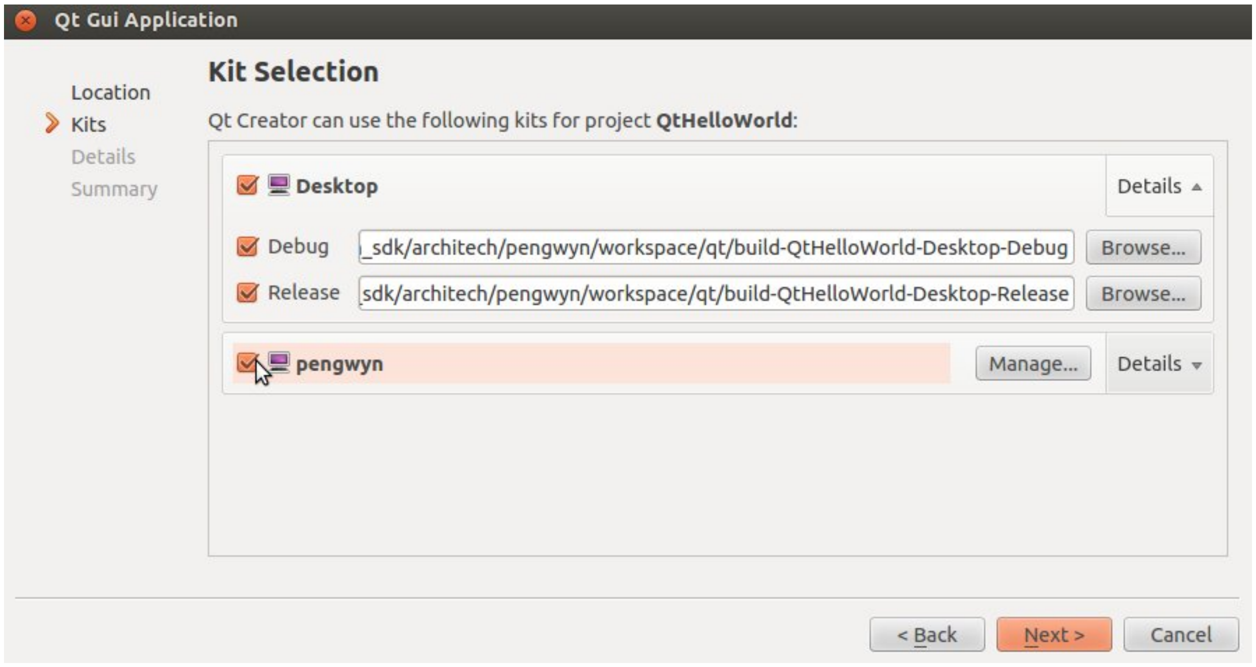
This wizard generates a Qt GUI application project. The application derives by default from QApplication and includes an empty widget.

Name: QtHelloWorld

Create in: /home/architech/architech_sdk/architech/pengwyn/workspace/qt

☐ Use as default project location

3. Check also *pengwyn* kit and continue to press *next* button to finish the creation of the project.



Kit Selection

Qt Creator can use the following kits for project **QtHelloWorld**:

✓ **Desktop**

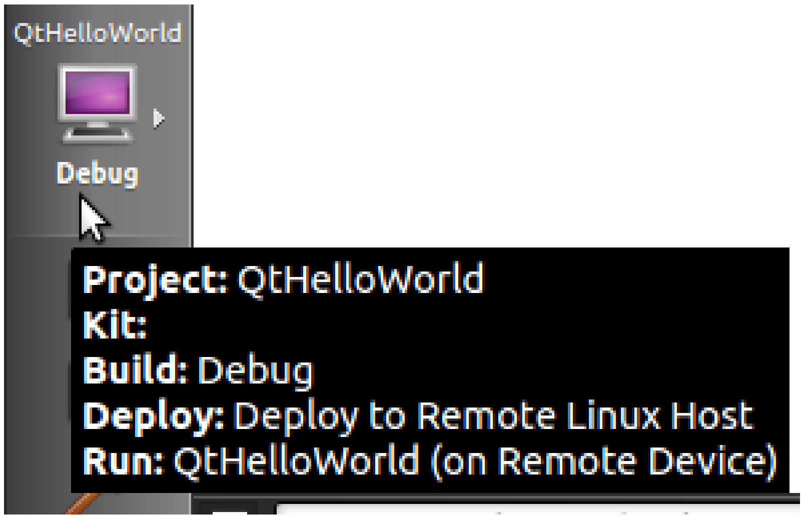
✓ Debug _sdk/architech/pengwyn/workspace/qt/build-QtHelloWorld-Desktop-Debug

✓ Release sdk/architech/pengwyn/workspace/qt/build-QtHelloWorld-Desktop-Release

✓ **pengwyn**

Note: Now you can edit your application adding labels and more, how to do this is not the purpose of this guide.

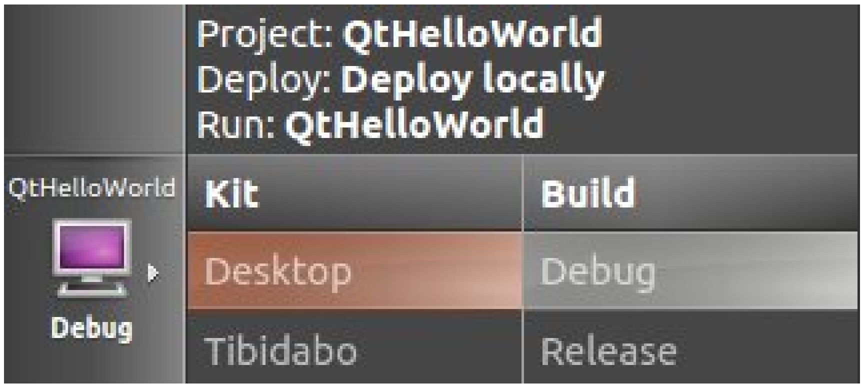
4. To compile the project click on "QtHelloWorld" icon to open project menu.


QtHelloWorld

Debug

Project: QtHelloWorld
Kit:
Build: Debug
Deploy: Deploy to Remote Linux Host
Run: QtHelloWorld (on Remote Device)

5. Select the build configuration: **Desktop - Debug**.

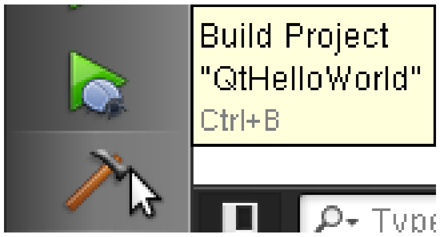

Project: **QtHelloWorld**
Deploy: **Deploy locally**
Run: **QtHelloWorld**

QtHelloWorld

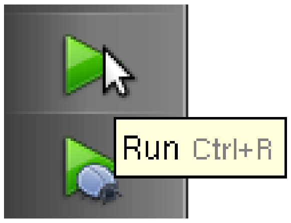
Debug

Kit	Build
Desktop	Debug
Tibidabo	Release

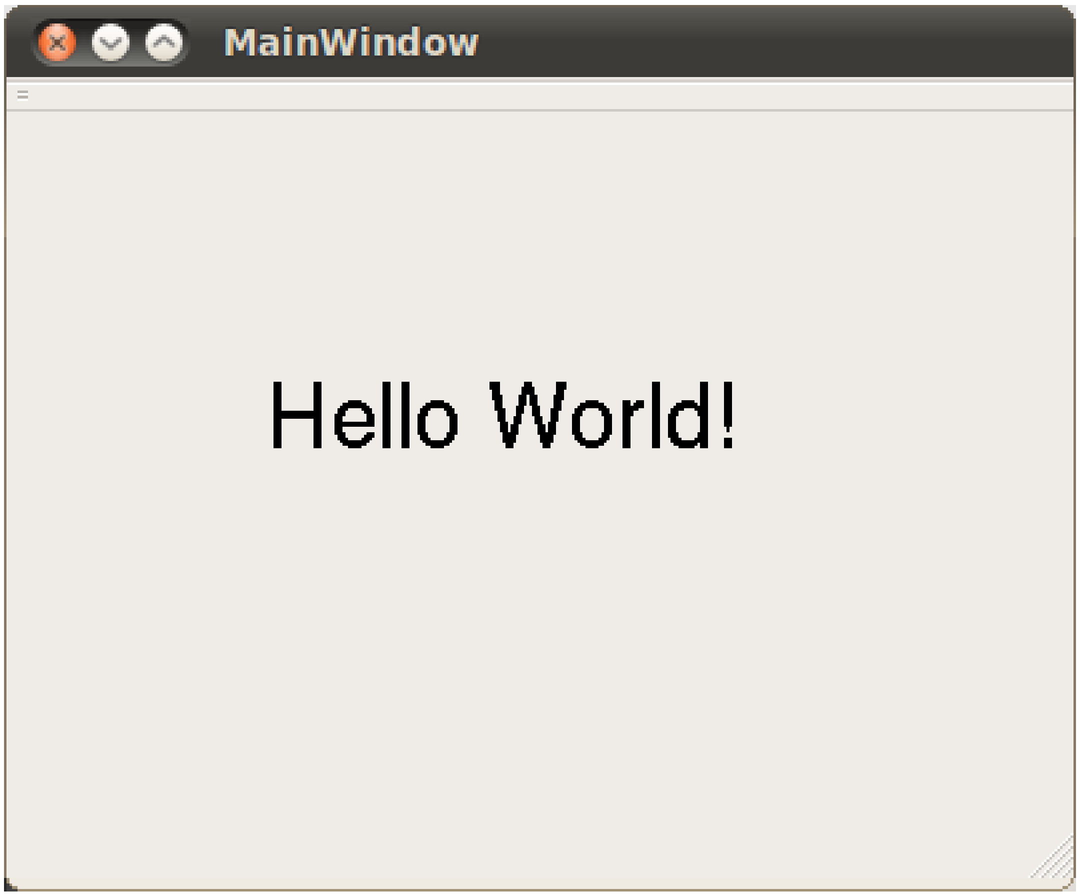
6. To build the project, click on the bottom-left icon.


Build Project "QtHelloWorld" Ctrl+B

7. Once you built the project, click on the green triangle to run it.



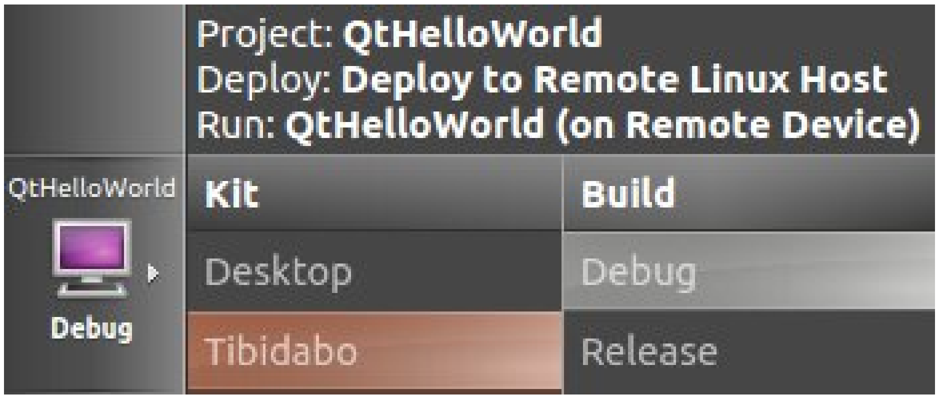
8. Congratulations! You just built your first Qt application for x86.



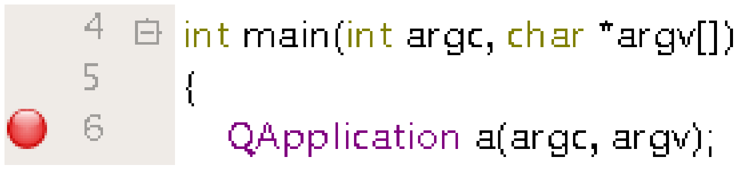
In the next section we will debug our Hello World! application directly on Tibidabo.

Debug Hello World project

1. Select build configuration: **tibidabo - Debug** and build the project.

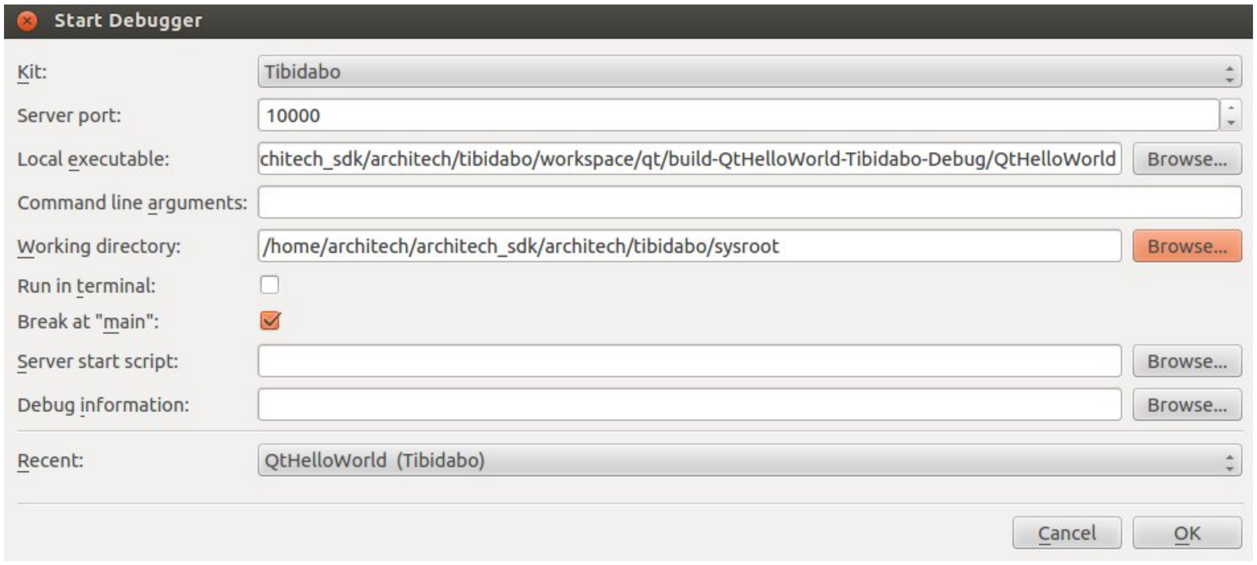


2. Copy the generated executable to the target board (e.g /home/root/).
3. Use minicom to launch gdbserver application on the target board:
4. In Qt Creator, open the source file main.cpp and set a breakpoint at line 6. | To do this go with the mouse at line 6 and click with the right button to open the menu, select **Set brackpoint at line 6**



```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
```

5. Go to *Debug→Start Debugging→Attach To Remote Debug Server*, a form named "Start Debugger" will appear, insert the following data:

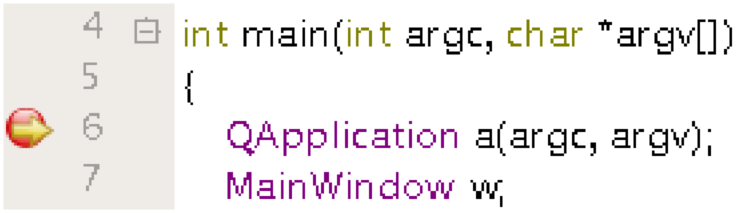


Start Debugger

Kit:	Tibidabo
Server port:	10000
Local executable:	chitech_sdk/architech/tibidabo/workspace/qt/build-QtHelloWorld-Tibidabo-Debug/QtHelloWorld
Command line arguments:	
Working directory:	/home/architech/architech_sdk/architech/tibidabo/sysroot
Run in terminal:	☐
Break at "main":	✓
Server start script:	
Debug information:	
Recent:	QtHelloWorld (Tibidabo)

- Kit: **tibidabo**
- Local executable:

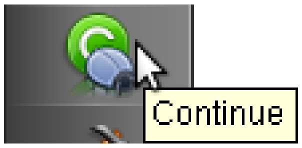
Press **OK** button to start the debug.



```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
```

6. The hotkeys to debug the application are:

- **F10**: Step over
- **F11**: Step into
- **Shift + F11**: Step out
- **F5**: Continue, or press this icon:



7. To successfully exit from the debug it is better to close the graphical application from the target board with the mouse by clicking on the 'X' symbol.

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

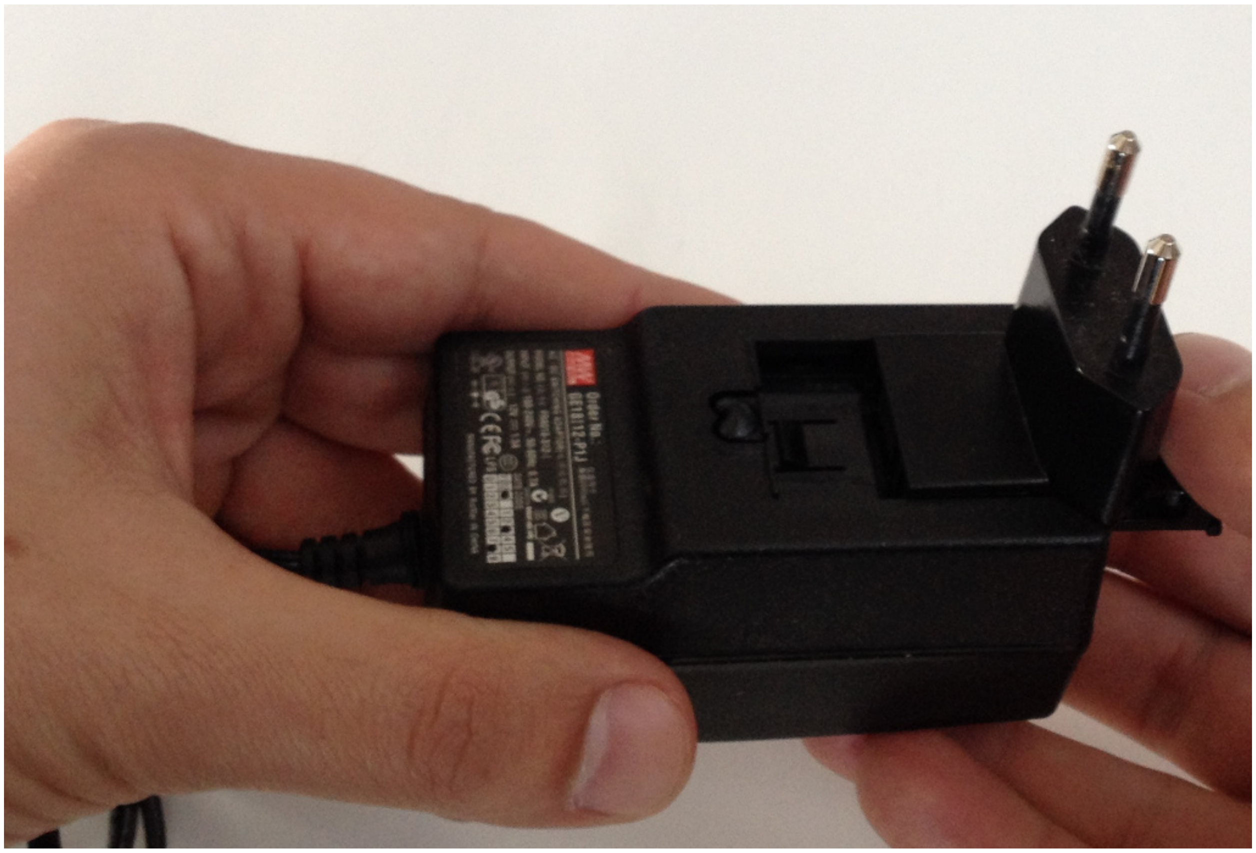
The hardware documentation of Tibidabo can be found here:

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

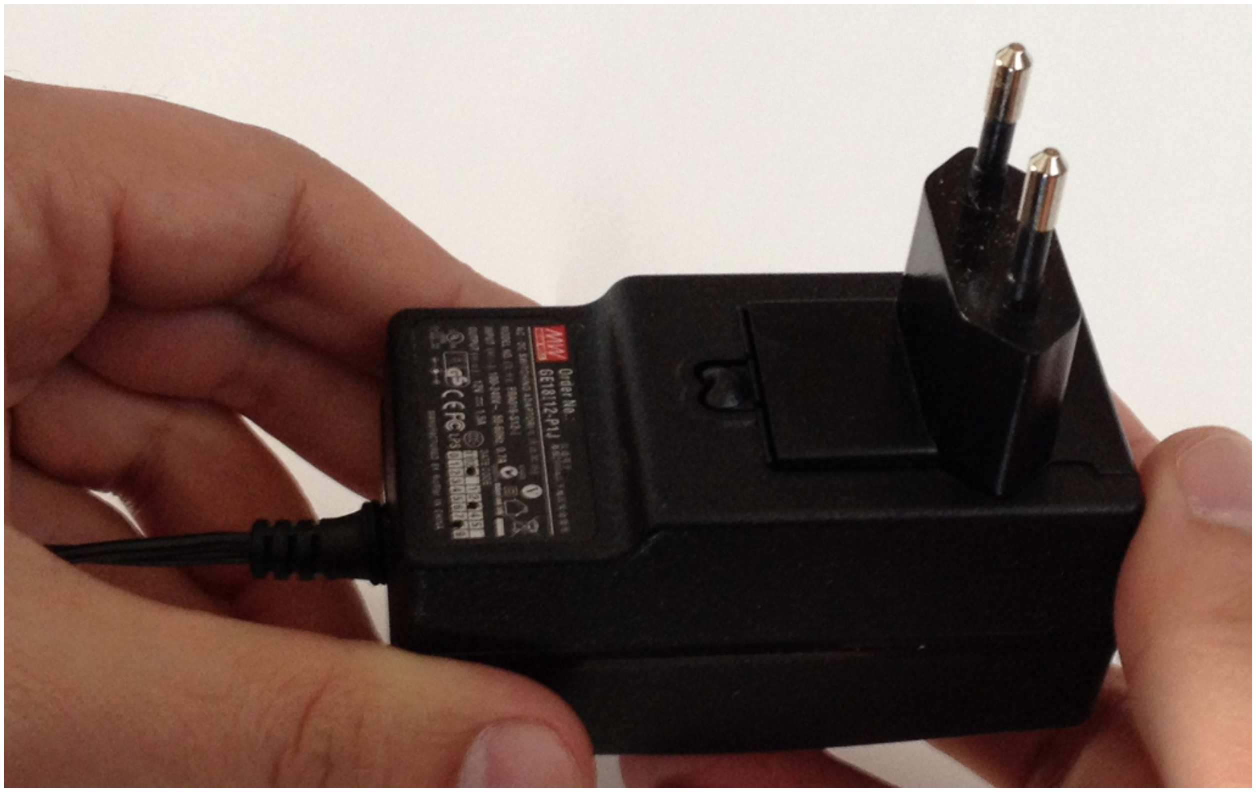
Power-On

Tibidabo takes the power from connector **CN19**. The board is shipped with an external power adapter.

To assemble it, take the power socket adapter compatible with your country, plug it in the power adapter.



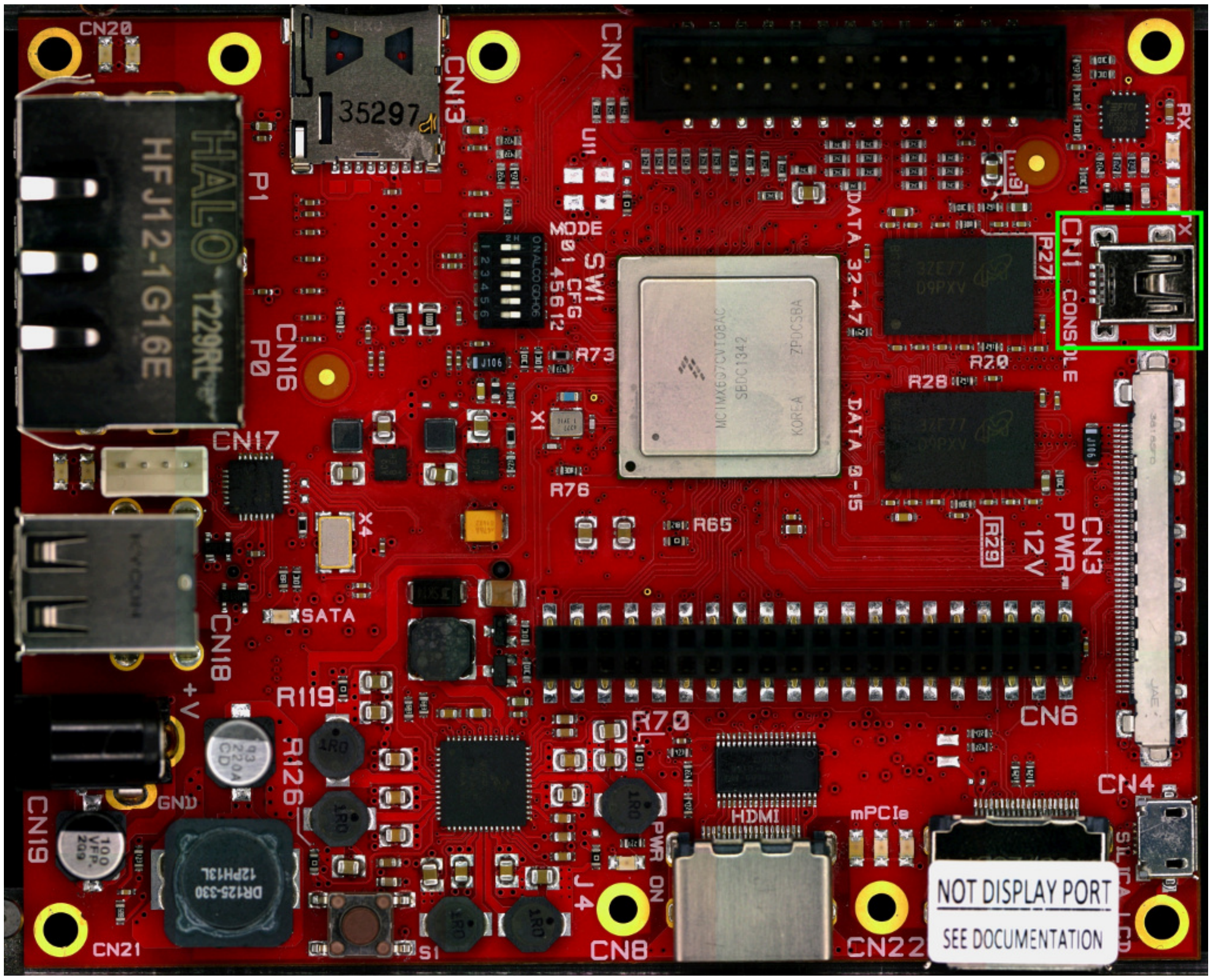
When in position, you should hear a slight *click*.



To power-on the board, just connect the external power adapter to Tibidabo connector *CN19*.

Serial Console

On Tibidabo there is the dedicated serial console connector **CN1**



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

3. read the output

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

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

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

then you can setup your port with these parameters:

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

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

Let's boot

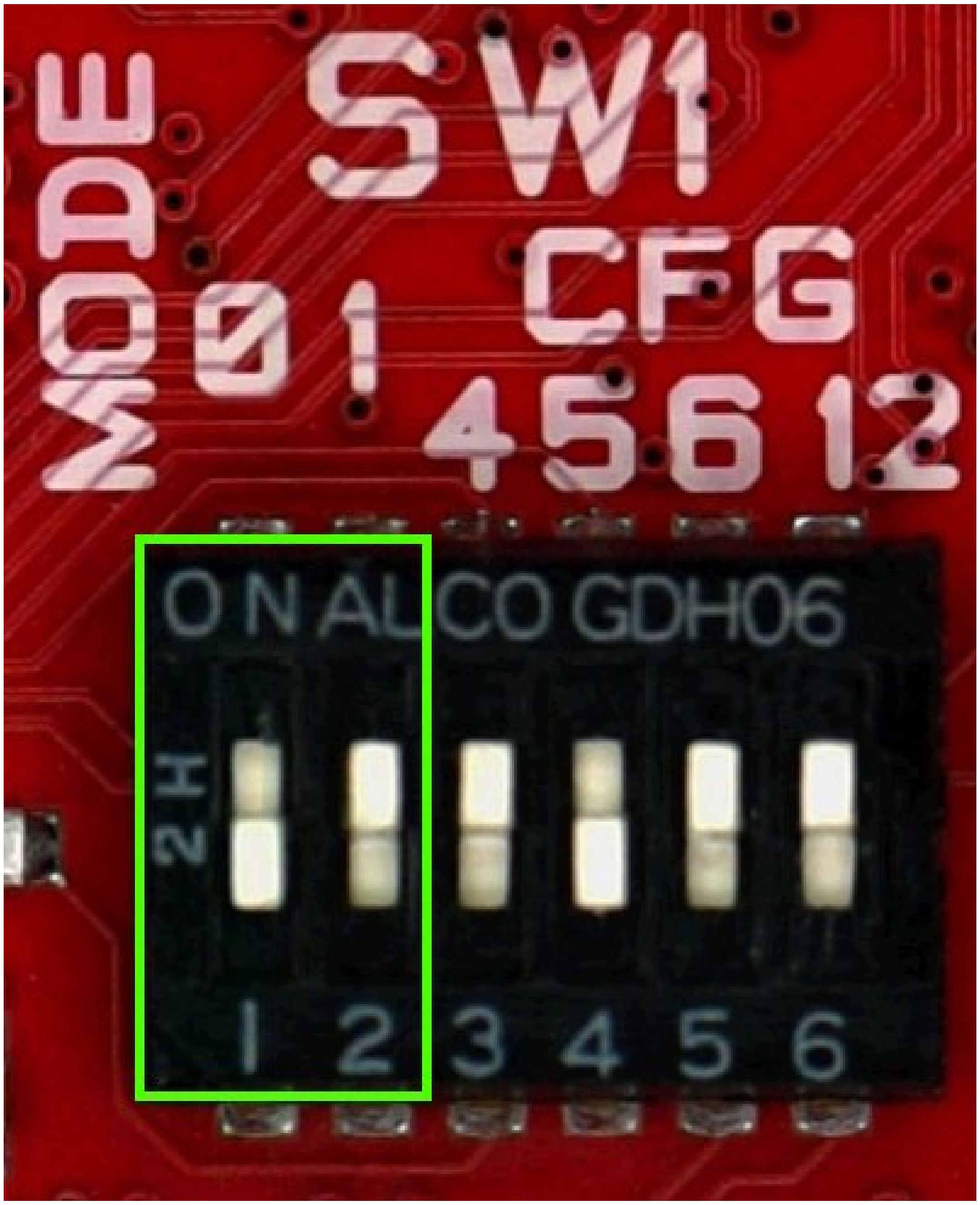
The boot process of an i.MX6 processor is quite complex. After a Power On Reset (POR) the processor starts executing the internal ROM program. The boot mode is based on the binary value stored in the internal **BOOT_MODE** register:

BOOT_MODE[1:0]	Boot Type
00	Boot from fuses
01	Serial downloader
10	Internal boot
11	Reserved

BOOT_MODE[1] is read from **SRC_BOOT_MODE1** pin (F12). **BOOT_MODE[0]** is read from **SRC_BOOT_MODE0** pin (C12).

On Tibidabo, switches 1 and 2 of **SW1** let you define the values for **BOOT_MODE** register:

- SW1 switch 1 controls BOOT_MODE[0]
- SW1 switch 2 controls BOOT_MODE[1]



in the image BOOT_MODE[1:0] = 10 (*Internal boot*).

The other switches of SW1 are used for *Internal boot* mode and will be explained later in this chapter.

eFUSEs

eFUSEs are One Time Programmable (OTP) devices. The On-Chip OTP Controller (OCOTP_CTRL) manages reads/writes from/to eFUSEs and memory mapping of the values by means of shadow registers. You can blow the fuses by means of **u-boot** fuse command, be very careful because **fuses are one time programmable only**, a mistake will last forever! However, even if you manage to brik the board, you can always use it with the Serial downloader boot mode.

Boot from FUSEs

In *boot from fuses mode* the boot ROM uses the fuses values to decide how to boot. The boot flow is controlled by **BT_FUSE_SEL** eFUSE:

- if 1 the boot ROM will load the bootloader according to the state of eFUSEs,
- if 0 (the device has not yet been programmed) the boot ROM will jump to *serial downloader* mode.

Tibidabo is shipped with no fuse blown so you can blow the fuses when you think you are ready.

For example, to instruct the processor to boot from **SD card** you can blow the following fuses with *u-boot* **fuse** command:

where, the first command setup the boot from sd card, while the second command sets **BT_FUSE_SEL = 1**.

Again, if you want to instruct the processor to boot from **SPI NOR** you can blow the following fuses:

where the first command setup the boot from serial ROM, and the second command sets **BT_FUSE_SEL = 1**.

Serial Downloader

Serial downloader boot mode tells the processor's boot ROM to load registers configuration and bootloader from **USB**. To work with this boot mode you need a micro USB cable to connect the board (connector *CN4*) to your Personal Computer and a software installed on your PC, speaking of which, if you have a Microsoft Windows operating system you need Freescale's i.MX6 *Manufacturing Tool* that can be downloaded from:

```
http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=IMX6_SW
```

If you have a Linux operating system instead, you need Boundary Devices *imx_usb_loader* tool that can be obtained from their git repository:

```
git://github.com/boundarydevices/imx_usb_loader
```

To compile *imx_usb_loader* project you need *libusb* installed on your distribution. This is the set of commands needed on an *Ubuntu* machine to setup the tool:

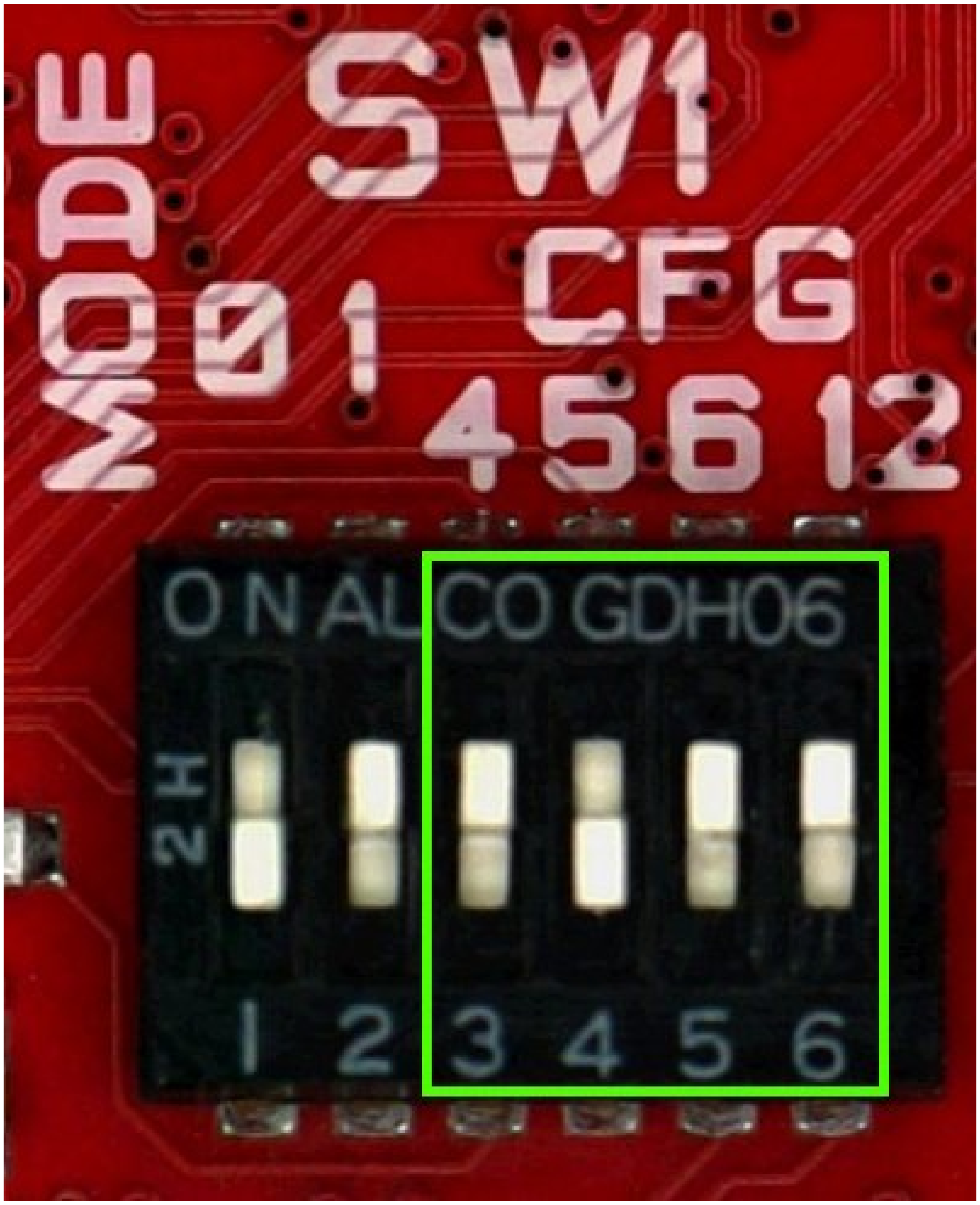
Once the tool is ready, power up the board, then you can download your *u-boot.imx* on the board with this command:

Internal Boot

If **BT_FUSE_SEL = 1** then all boot options are controlled by the eFUSEs, otherwise, if **BT_FUSE_SEL = 0** then specific boot configuration parameters may be set using GPIO pins rather than eFUSEs. The use of GPIOs is intended for **development only**. If an error occurs, the boot ROM jumps to serial downloader boot mode. On Tibidabo, **SW1** switches 3, 4, 5, 6 (along with a set of jumpers available on the bottom side of the board) can define a custom boot mode so you can simulate your configuration before blowing fuses.

SW1[6:3] = BOOT_CFG[24]-BOOT_CFG1[6:4]	Boot Device
1100	SD regular boot
1101	SD fast boot
0011	Serial NOR
0010	SATA

For example, this is the selection of the boot from SD card (fast boot)



Bootloader deploy

When you boot with *serial downloader*, you just do:

but when you *boot from fuses* or you want to use the *internal boot* you need to understand where the processor looks for the bootloader binary. If you want to boot from SPI NOR, you need to write the bootloader binary (*u-boot.imx*) to the flash memory. You can do it with from *u-boot* or from *Linux* as well. To do it from *u-boot*, you first need to read into memory a valid bootloader binary (from ethernet, SD card, mSATA or USB), then:

where *loadaddr* is an environment variable where the memory load address is defined, and *filesize* is the size of file *u-boot.imx* that has been previously loaded to memory. Be careful, by default the bootloader is configured to save the environment inside the SD card, not in the flash itself. If you prefer to save the environment inside the SPI NOR, open u-boot file:

define macro **CONFIG_ENV_IS_IN_SPI_FLASH** by uncommenting it, comment **CONFIG_ENV_IS_IN_MMC** definition, and recompile the bootloader.

In case you want to boot from SD card, you need to write the bootloader starting at address 1024 on the medium, just inside the MBR gap. The first partition on the medium must start at an address that leaves enough room for then bootloader and its environment variables, block 8192 (with block size of 512) will be more then enough (the environment gets written/read on the SD card with an offset of 384KB and will be 8KB large). Good, but how do you write your u-boot binary on the SD card? If you do not care to customize the bootloader, and you built an image with Yocto/OpenEmbedded, you may have noticed that under the directory where Yocto/OpenEmbedded puts all the built images there is a file with extension *.sdcard*. Well, such a file is an iso and can be written *as is* to the SD card device, just:

Once the iso has been written, the SD card will have all you need to make it boot from it (it will have bootloader, kernel image, file system and kernel modules). Ok, but what if you want to rewrite just the bootload and not the all image? You can overwrite the bootloader on the SD card always with *dd*:

Bootscript

Once the bootloader has been properly deployed (see *Bootloader deploy*), you turn on the board, the bootloader gets loaded and starts running until it gets to the boot command. What happens next? Well, since the board have a lot of options from where to load the kernel and with which options run the kernel, where is the root file system, which video mode, etc..., you get the best result if you have a simple facility to customize the system boot process yourself instead of having a milion combinations script that doesn't do exactly what you want it to do. The facility we are talking about is a simple *u-boot* script that the default boot command tries to load from, in order, mSATA, SD and tftp. When u-boot finds it, the script gets executed. That's it. Here is an example of an u-boot script that tries to load the Linux kernel binary from the SD card first partition (the partition can be FAT, EXT2, EXT3 or EXT4), and tells the kernel to use the second partition of the SD card as root partition:

But that is an u-boot script, not the *bootscript*, to make it suitable as a bootscript you need to give it **mkimage** as input first. If you are not that comfortable with *mkimage*, you can have a simplified interface offered by create-bootscript.sh script. The usage is very simple, just run it like this:

where parameter *-i* stands for source file to take as input and *-o* stands for "binary" file to emit as output.

Copy the output file to where you want it to be found, that is:

- SD card, first or second partition in the root director
- mSATA, first or second partition in the root directory, or
- TFTP directory on your computer.

Important: Name the script exactly **bootscript**

Video modes

Tibidabo has three possible video outputs:

- HDMI via connector *CN8*
- LVDS via connector *CN3*, thought for SAMSUNG's MODEL LTI460HN08 (connector pad numeration is reversed with respect to SAMSUNG monitor datasheet to direct use of a flat ribbon lvds cable)
- LVDS via display port connector *CN22*, meant for SILICA lvds display

Warning: Do not connect CN22 to DISPLAY PORT devices, CN22 uses just the connector of a DISPLAY PORT but the signals are meant to work just with Silica's LCD (LVDS) displays.

If you want to boot using SILICA's lcd as the only video output device you need to add to the kernel command line something like:

```
video=mxcfb0:dev=ldb,LDB-WVGA,if=RGB666 ldb=dul0
```

If you want to boot using SAMSUNG's display as the only video output device you need to add to the kernel command line something like:

```
video=mxcfb0:dev=ldb,LDB-1080P60,if=RGB666 ldb=spl0
```

If you want to boot using a full HD HDMI display as the only video output device you need to add to the kernel command line something like:

```
video=mxcfb0:dev=hdmi,1920x1080M@60,if=RGB24
```

You can have a video output on more than one device and the resolutions stated before are not the only resolutions available. Keep also into account that the LVDS output has several working modes, like: *spl, dul, sin, sep* (please, have a look at */drivers/video/mxc/ldb.c*).

Network

Tibidabo networking is powered by MARVELL Gigabit switch MV88E6123. On the board there is a dual ethernet connector, each connector has a name that is printed on the PCB (*P0* and *P1*). The switch is supported both by *u-boot* and *Linux kernel*, however, *u-boot* support is limited so, if you need u-boot to load files from the network use just one of the two ports. Under Linux, instead, the default network configuration is:

but if you take a closer look, you discover that there are more interfaces available:

where **pt0** is the network inteface corresponding to connector **P0**, while **pt1** is the network interface corresponding to connector **P1**.

eth0 has a random MAC address assigned and, as you can see, *pt0* and *pt1* have the same address. To properly use the network you need to be sure that *pt0* and *pt1* have unique MAC addresses. You can change the MAC address of a specific network interface by means of this command:

substitute <*port*> with *pt0* or *pt1*, and <*new mac address*> with the MAC address you decided to assign.

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 *pt0* 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:

You can, of course, define the default configuration for *pt1* as well.

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.

Tibidabo

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