|  |
| --- |
|  |
| Multimedia  Draft |
| |  |  |  | | --- | --- | --- | | Nico De Witte | 2014-2015 | nico.dewitte@vives.be | |

Contents

[2 The Raspberry Pi 3](#_Toc400646194)

[2.1 What’s the Raspberry Pi Made off 4](#_Toc400646195)

[2.1.1 Specifications 6](#_Toc400646196)

[3 An Operating System 7](#_Toc400646197)

[3.1 Installing Raspbian 7](#_Toc400646198)

[3.2 Booting the Raspberry Pi 8](#_Toc400646199)

[3.2.1 Graphical Interface 8](#_Toc400646200)

[3.2.2 SSH connection 8](#_Toc400646201)

[3.2.3 RS232 Connection 10](#_Toc400646202)

[3.2.4 Checking The Kernel Messages 12](#_Toc400646203)

[3.3 Initial Configuration 13](#_Toc400646204)

[3.3.1 Expanding the Filesystem 13](#_Toc400646205)

[3.3.2 Change User Password 13](#_Toc400646206)

[3.3.3 Enable Boot to Desktop / Scratch 13](#_Toc400646207)

[3.3.4 Internationalization Options 13](#_Toc400646208)

[3.3.5 Enable Camera 14](#_Toc400646209)

[3.3.6 Add to RasTrack 14](#_Toc400646210)

[3.3.7 Overclock 14](#_Toc400646211)

[3.3.8 Advanced Options 14](#_Toc400646212)

[3.4 Checking the configuration 16](#_Toc400646213)

[4 Linux Basics 17](#_Toc400646214)

[4.1 The MAN-Pages 17](#_Toc400646215)

[4.2 The Linux Filesystem 18](#_Toc400646216)

[4.2.1 Some Brief Notes on the History of the Linux Filesystem Layout 18](#_Toc400646217)

[4.2.2 Traversing the Filesystem 18](#_Toc400646218)

[4.2.3 An Overview of the Linux Filesystem Layout 20](#_Toc400646219)

[4.2.4 overview of Basic Filesystem commands 24](#_Toc400646220)

[4.3 Debian and it’s Packages 25](#_Toc400646221)

[5 Running a Virtual Machine 27](#_Toc400646222)

[5.1 Installing Oracle VirtualBox 27](#_Toc400646223)

[5.2 Creating a Virtual Machine 29](#_Toc400646224)

[5.3 Configuring the Virtual Machine 32](#_Toc400646225)

[5.4 Installing the Ubuntu Operating System 33](#_Toc400646226)

[5.4.1 Installing Guest Additions 35](#_Toc400646227)

[6 Buildroot: making embedded Linux easy 36](#_Toc400646228)

[6.1 The Linux kernel 36](#_Toc400646229)

[6.2 The cross-compilation toolchain 38](#_Toc400646230)

[6.3 Buildroot 40](#_Toc400646231)

[6.3.1 Downloading the latest Buildroot 40](#_Toc400646232)

[6.3.2 Host System Requirements 40](#_Toc400646233)

[6.3.3 Configuring your Buildroot System 41](#_Toc400646234)

[6.3.4 Building it all 42](#_Toc400646235)

[6.3.5 Creating a Bootable SD Card 42](#_Toc400646236)

[7 The Raspberry Pi Boot Process 45](#_Toc400646237)

[8 Important Commands Overview 47](#_Toc400646238)

[9 References 47](#_Toc400646239)

# The Raspberry Pi

The Raspberry Pi is a credit card-sized single-board computer developed in the UK by the Raspberry Pi Foundation with the intention of promoting the teaching of basic computer science in schools.

The Raspberry Pi has a Broadcom BCM2835[[1]](#footnote-1) system on a chip (SoC[[2]](#footnote-2)), which includes an ARM1176JZF-S 700 MHz processor, a VideoCore IV GPU, and was originally shipped with 256 megabytes of RAM, later upgraded (Model B & Model B+) to 512 MB. It does not include a built-in hard disk or solid-state drive, but it uses an SD card for booting and persistent storage, with the Model B+ using a MicroSD.

ARM is a family of instruction set architectures for computer processors based on a reduced instruction set computing (RISC[[3]](#footnote-3)) architecture developed by British company ARM Holdings.

The Broadcom SoC used in the Raspberry Pi is equivalent to a chip used in an old smartphone (Android or iPhone). While operating at 700 MHz by default, the Raspberry Pi provides a real world performance roughly equivalent to the 0.041 GFLOPS. On the CPU level the performance is similar to a 300 MHz Pentium II of 1997-1999, but the GPU, however, provides 1 Gpixel/s, 1.5 Gtexel/s or 24 GFLOPS of general purpose compute and the graphics capabilities of the Raspberry Pi are roughly equivalent to the level of performance of the Xbox of 2001. The Raspberry Pi chip operating at 700 MHz by default, will not become hot enough to need a heatsink or special cooling.

The Raspberry Pi became popular very fast as the system was cheap but powerful. Some projects include:

* **Use a Raspberry Pi to Automate Time-Lapse Photos** (<https://sites.google.com/site/raspilapse/home>)



* **Raspberry Pi Supercomputer** (<http://www.southampton.ac.uk/~sjc/raspberrypi/>)



* **Automatic DeviantArt Picture Frame** (<http://theswitchtolinux.blogspot.be/2012/12/raspberry-pi-daily-deviations-picture.html>)



And the list keeps growing every day.

## What’s the Raspberry Pi Made off

The Raspberry has everything onboard you need for a small media center, simple NAS or home automation system. Figure 1 shows the placement of the different components on the Raspberry Pi model B. While the Pi is at its third major board revision, we will be focusing on model B in this course.

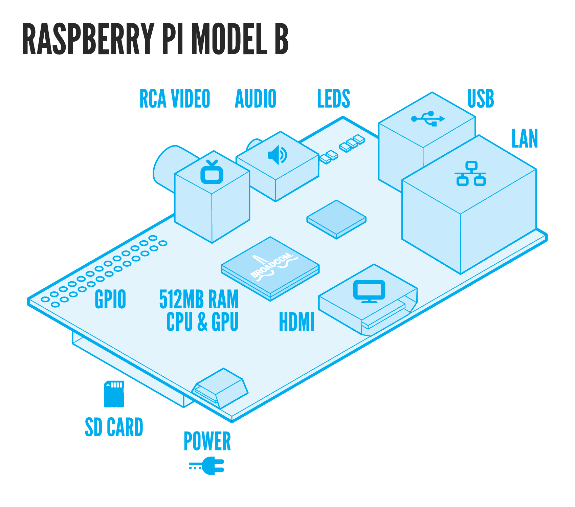
 

Figure 1 Raspberry Pi model B - Component placements (left) – Real Board (right)

The Pi model B offers:

* Full size SD card interface
* HDMI output port
* Composite video output
* 26 pin expansion header exposing GPIO, I2C, etc
* 3.5mm audio jack
* Camera interface port (CSI-2)
* LCD display interface port (DSI)
* One microUSB power connector for powering the device
* One Ethernet port
* Two USB ports

Check out Section 2.1.1 for more detailed information on the Pi’s onboard hardware.

If you're wondering why you can't see the Broadcom SoC anywhere on the Raspberry Pi, it's because you're looking in the wrong place. The processor is hidden beneath the Hynix memory chip at the centre of the board, which uses a package-on-package (PoP) (example shown in Figure 2) mounting process to create a RAM-SoC sandwich.

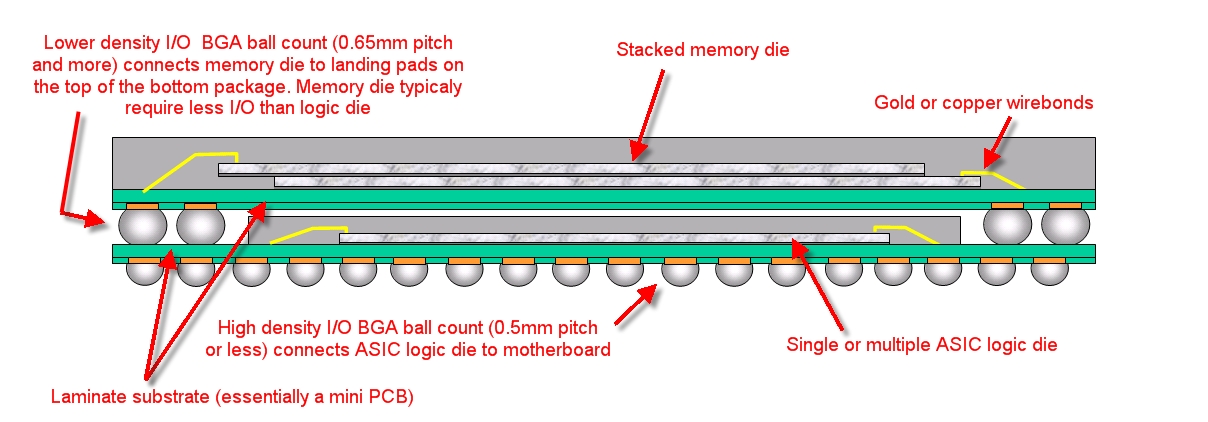


Figure 2 Package-on-Package

Important to know is that the Ethernet controller (shown in Figure 2) on the Pi has a build-in USB hub. The upstream port of this hub is connected to the Broadcom processor while the two downstream ports are made available using a dual-stacked USB port.

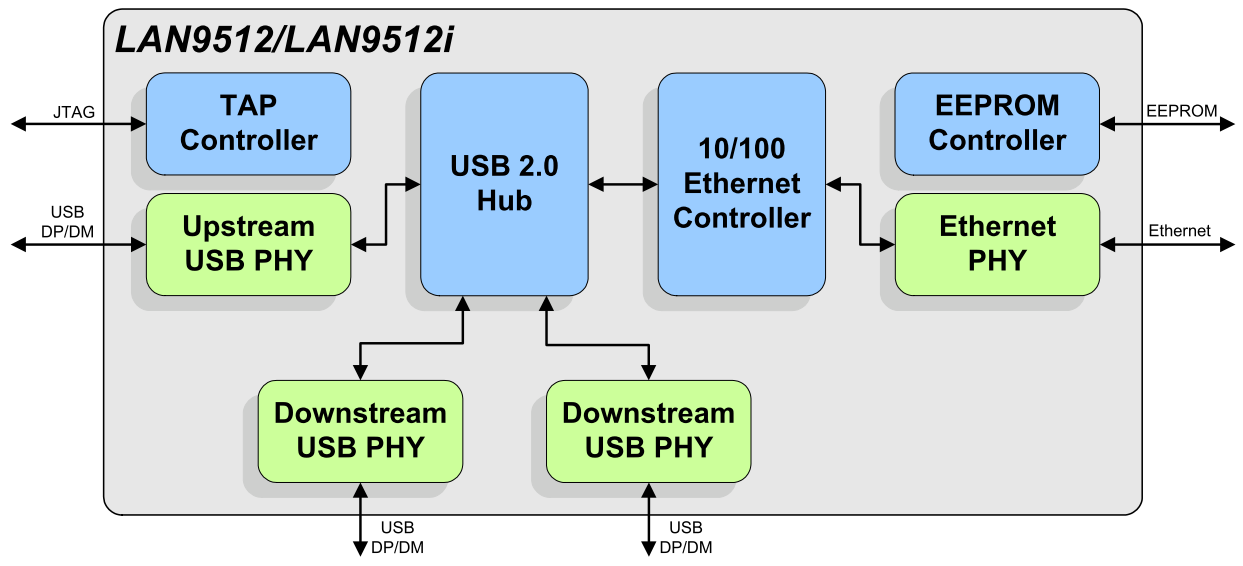


Figure 3 The Pi's Ethernet Controller with Integrated USB HUB

Detailed schematics of the Pi can be found at <http://www.raspberrypi.org/documentation/>.

### Specifications

Table 1 shows an overview of the differences between the different major board revisions of the Pi.

Table 1 RPi Major Board Revisions

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Model A** | **Model B** | **Model B+** |
| **Target price** | US$25 Ext tax | US$35 Ext tax | US$35 Ext tax |
| **System-on-a-chip (SoC)** | Broadcom BCM2835 (CPU + GPU. SDRAM is a separate chip stacked on top) | | |
| **CPU** | 700 MHz ARM11 ARM1176JZF-S core | | |
| **GPU** | Broadcom VideoCore IV, OpenGL ES 2.0, OpenVG 1080p30 H.264 high-profile encode/decode | | |
| **Memory (SDRAM)iB** | 256 MiB (planned with 128 MiB, upgraded to 256 MiB on 29 Feb 2012) | 256 MiB (until 15 Oct 2012)  512 MiB (since 15 Oct 2012) | 512 MiB |
| **USB 2.0 ports** | 1 (provided by the BCM2835) | 2 (via integrated USB hub) | 4 (via integrated USB hub) |
| **Video outputs** | Composite video | Composite RCA, HDMI (not at the same time) | | Composite video requires 4 Pole Adapter |
| **Audio outputs** | TRS connector | 3.5 mm jack, HDMI | | |
| **Audio inputs** | None, but a USB mic or sound-card could be added | |  |
| **Onboard Storage** | Secure Digital|SD / MMC / SDIO card slot | | Micro Secure Digital / MicroSD slot |
| **Onboard Network** | None | 10/100 wired Ethernet RJ45 | |
| **Low-level peripherals** | 26 General Purpose Input/Output (GPIO) pins, Serial Peripheral Interface Bus (SPI), I²C, I²S, Universal asynchronous receiver/transmitter (UART) | | 40 General Purpose Input/Output (GPIO) pins, Serial Peripheral Interface Bus (SPI), I²C, I²S, I2C IDC Pins, Universal asynchronous receiver/transmitter (UART) |
| **Real-time clock** | None | | |
| **Power ratings** | 300 mA, (1.5 W) | 700 mA, (3.5 W) | ~650 mA, (3.0 W) |
| **Power source** | 5 V (DC) via Micro USB type B or GPIO header | | |
| **Size** | 85.0 x 56.0 mm x 15mm | 85.0 x 56.0 mm x 17mm | 85.0 x 56.0 mm x 17mm |
| **Weight** | 31g | 40g | 40g |

While the specs recommend a 700mA power supply, it’s best to use a 1000mA power supply when connecting the Pi to an Ethernet network. Also make sure not to connect external HDD drives, which are powered by USB, directly to the Pi. For these setups always use an externally powered USB hub.

# An Operating System

The Raspberry Pi foundation provides several ready to use operating system images for the Pi. At the moment of this writing the following are available:

* RASPBIAN - Debian Wheezy
* PIDORA - Fedora Remix
* OPENELEC - An XBMC Media Centre
* RASPBMC - An XBMC Media Centre
* RISC OS - A non-Linux distribution
* ARCH LINUX - A lightweight Linux distribution

For the start of this course we will be using the Raspbian image. Later we will create our own distribution.

## Installing Raspbian

You can download the latest image from the Raspberry Pi website (<http://www.raspberrypi.org/downloads/>). Make sure to pick the “Download ZIP” option. Extract the zip file on your local disk. You should get an image file (.img extension).

The current version at the moment of this writing is of June 2014 with a Linux kernel version of 3.12. You can always check out the release notes on <http://downloads.raspberrypi.org/raspbian/release_notes.txt>.

To boot this Linux distribution we will need to write the image file to an SD card of at least 4GB. A popular tool to write the image to an SD card is “Win32 Disk Imager”[[4]](#footnote-4) which can be downloaded at <http://sourceforge.net/projects/win32diskimager/>

Select the correct device letter and load the Raspbian image from your local drive as shown in Figure 1. If you’re ready, hit the write button and grab a cup of coffee. You can also create a backup of your current SD card by reading from the SD card to an image file. Just make sure to select a new image file name.

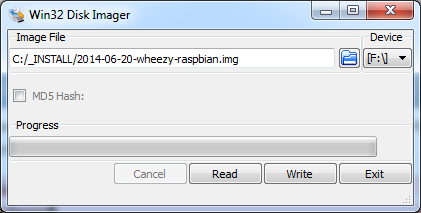


Figure 4 Win32 Disk Imager

Once the write process is finished you can remove the SD card and plug it in the Raspberry Pi. Just make sure to disconnect the power before inserting the SD card.

If you want your Pi to be connected to your local area network (LAN) you will have to plug in the Ethernet cable before booting the Pi. The Pi is default configured to acquire an IP address using DHCP.

## Booting the Raspberry Pi

Booting the Raspberry Pi is really simple. All you have to do is plug in the supply adapter and it automatically boots from the SD card. Interacting with the Linux operating system from that point on can be a bit harder in certain situations.

### Graphical Interface

If you deployed an OS such as Raspbian than you can attach an HDMI display or RCA Video compatible device (yellow connector on the board, cfr. Figure 1). You will also have to connect a USB keyboard to the Pi to be able to navigate through the configuration menu you will be presented with on the initial boot.

Jump to Section 3.3 to configure the Pi for initial use by means of the configuration menu.

### SSH connection

Raspbian comes default with the SSH[[5]](#footnote-5) daemon enabled. This allows us to connect to the Pi from a remote computer using the SSH protocol. Before we can do this we will have to determine the IP address of the Pi. In case of a home network you can log on to your router and look for the last IP address that was given by your DHCP server running on the router.

Another option can be a network scan tool such as SoftPerfect Network Scanner[[6]](#footnote-6) which allows you to scan a range of IP addresses and display some basic information about them such as the MAC (Media Access Control) address and the hostname.

This would not be an option in a LAB if there are 12 Pi’s connected to the same subnet all with the default configuration of Raspbian. However for your convenience we added labels on the Pi’s with their respective MAC addresses so you can identify which Pi is yours.

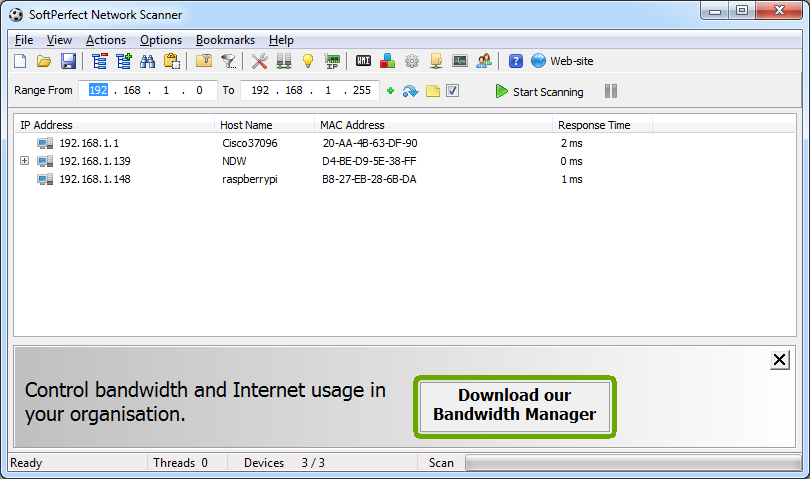


Figure 5 Network scan using SoftPerfect Network Scanner

Another option is using WireShark[[7]](#footnote-7) and watch the communication on the network. Especially the DHCP traffic which distributes IP addresses to the connected client devices. This way you can also identify what IP address is given to your device (if you know the MAC address of your device).

TODO

Connecting to a device using the SSH protocol can be easily achieved using a terminal tool such as Putty[[8]](#footnote-8). All you have to do is start Putty and select the SSH connection option and specify the IP address of the device as shown in Figure 3. Once the connection is configured you can open it.

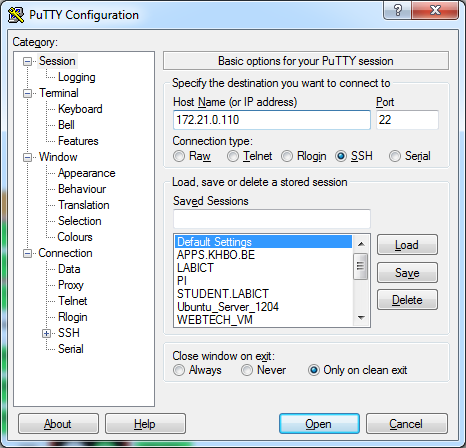


Figure 6 Opening an SSH connection using Putty

You will be presented with the command line interface (CLI) of the Linux operating system running on your device. The first thing you will see is a login screen similar to the one shown in Figure 4.



Figure 7 The login screen of the Raspbian distribution running on the Pi

The default username and password can be found on the Raspberry Pi website. For Raspbian it is “pi” as username and “raspberry” as password. Once you login with these credentials you are presented with the command line interface as shown in Figure 5. From this point on you can start to execute commands on the Pi.

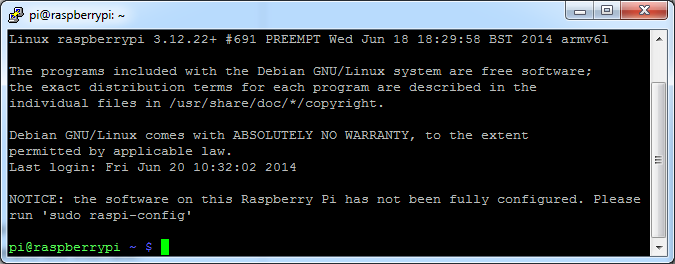


Figure 8 The command line interface after logging in

One of the most useful commands you should remember is the “ifconfig” command which displays the current network interfaces and their configuration parameters. If you execute the command you should get a similar output to the one shown in Figure 6. Try to identify the IP address and MAC address of the primary Ethernet interface (eth0).

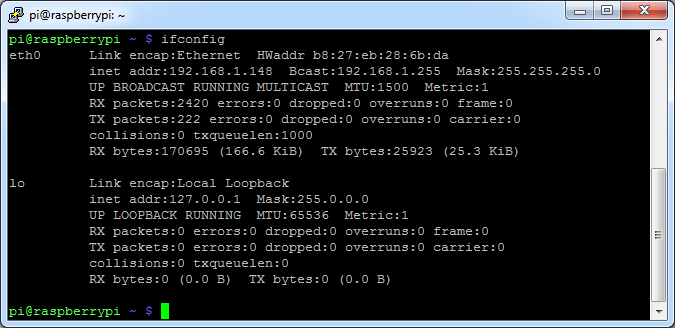


Figure 9 Output of the ifconfig command

Jump to Section 3.3 to configure the Pi for initial use using the configuration menu.

### RS232 Connection

A last option we can use to connect to the Raspberry Pi is by using a serial connection. This is often used for debugging embedded systems because it is a very basic connection type. Because of this the kernel will also output its kernel messages (debugging information and errors) to this connection. Since most computers these days lack the serial interface we can use a simple RS232 to USB converter such as the PL-2303HX[[9]](#footnote-9).

To attach the converter we do have to take a look at the pinout of the GPIO connector on the Raspberry Pi board, shown in Figure 7.

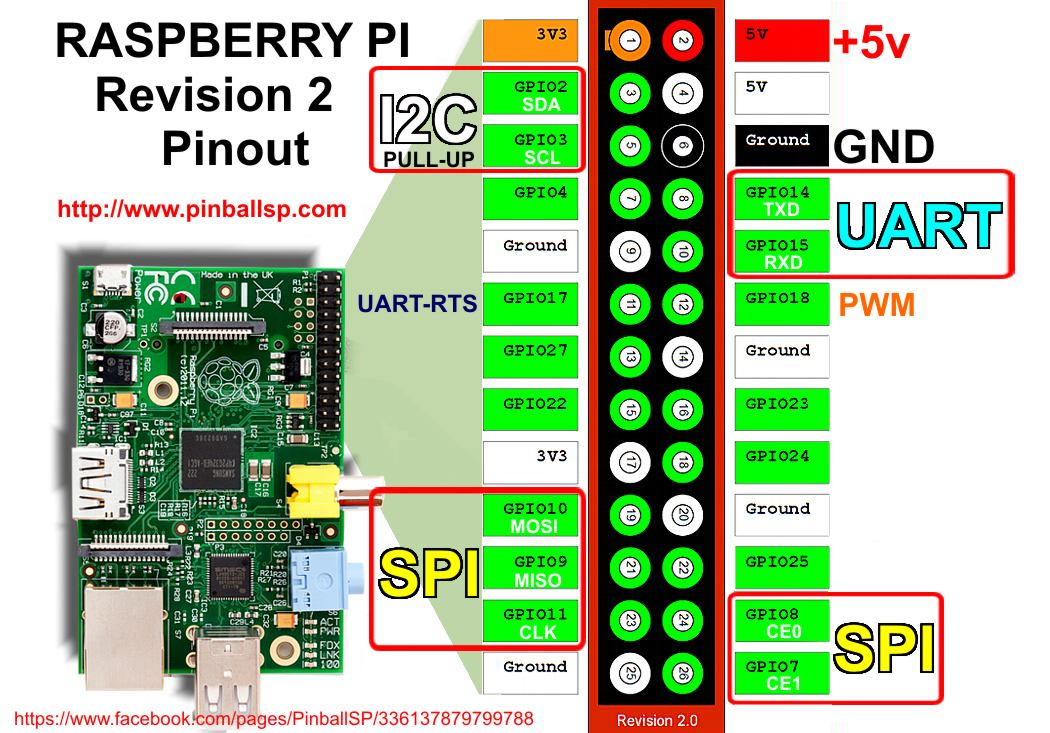


Figure 10 Raspberry Pi GPIO pinout

The serial-to-USB converter has three pins that need to be connected to the UART of the Pi. A Tx (transmit), an Rx (receive) and a GND (ground) pin. To allow communication with the Raspberry Pi the Tx of the Pi has to be connected with the Rx of the converter, while the Rx of the Pi has to be connected with the Tx of the converter. The GND pin of the converter needs to be connected to a GND pin of the Pi. Do NOT connect the 3V3 or 5V pin of the connector to the Pi! The necessary connections are shown in Figure 8.

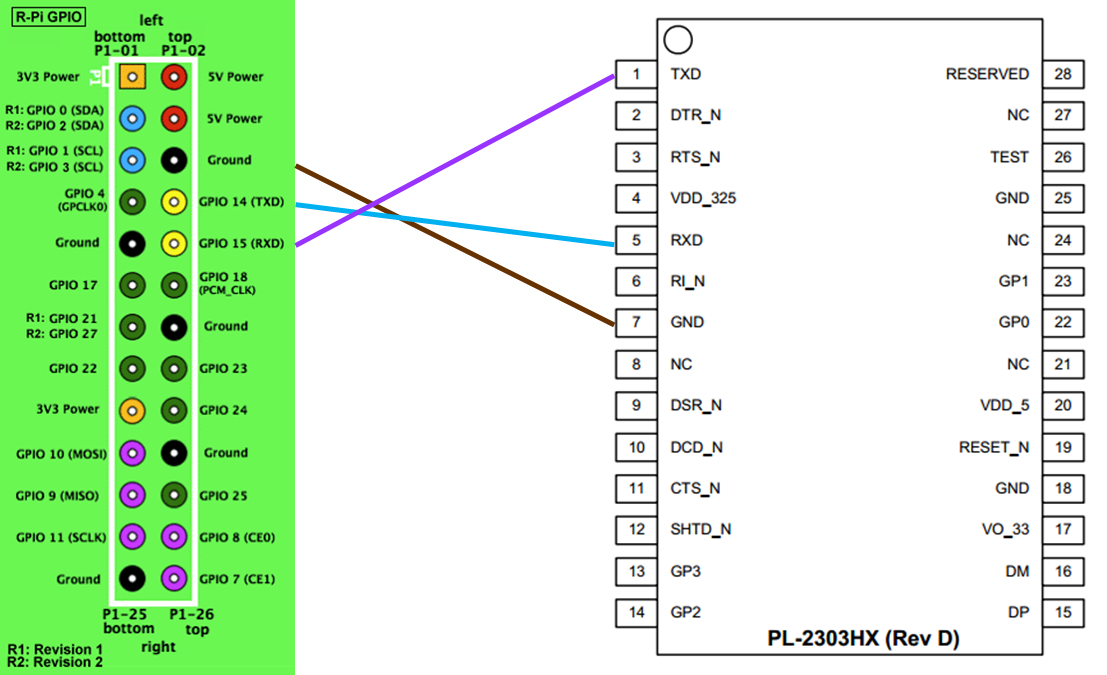


Figure 11 Connecting the PL2303HX to the UART of the Raspberry Pi

Just as with SSH, you can use Putty for the serial terminal. Just select “serial” as connection type, “COMx”[[10]](#footnote-10) (where x is an integer number) as serial line and “115200” as speed. An example is shown in Figure 9. Choose open and you will a get a command line interface similar to the one of SSH.

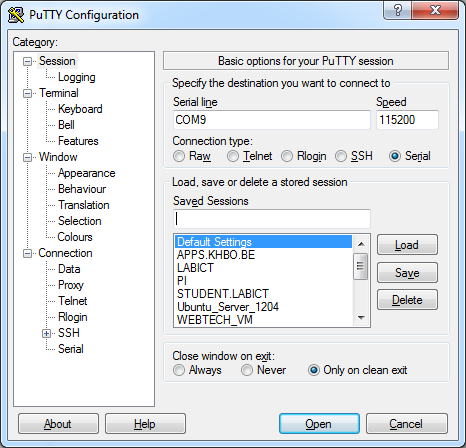


Figure 12 Serial line connection parameters

If you reboot your Raspberry Pi at this moment you will see the kernel messages shown in Figure 10 mentioned earlier.

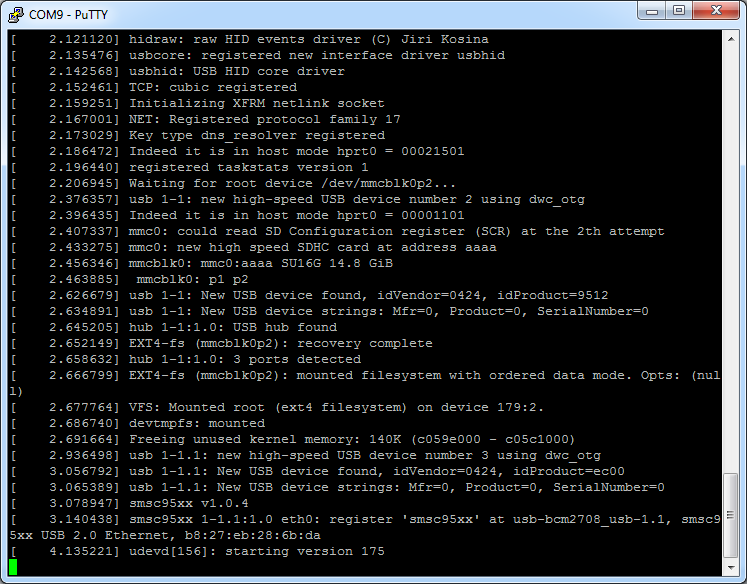


Figure 13 Kernel messages on serial line

### Checking The Kernel Messages

While the RS232 connection will automatically show the kernel messages while booting, you can also retrieve the output from an SSH connection by using the command below:

$ dmesg

which outputs the kernel messages to your current terminal.

## Initial Configuration

If you choose to attach an external display to the Pi, then you will already have seen the menu with the initial configuration options shown in Figure 11.

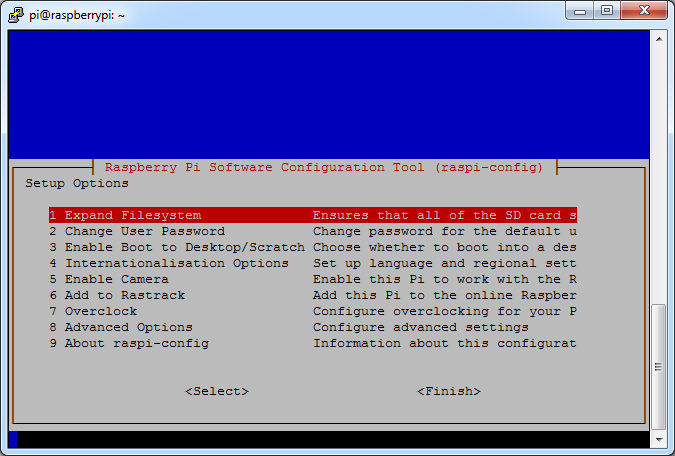


Figure 14 Initial configuration tool

If you choose to use the SSH or serial method you will need to run the command

$ sudo raspi-config

to get this configuration menu. Don’t worry too much about how this works. This will be explained in a later chapter.

Once you’re done configuring the Pi, choose the Finish option and let the Pi reboot.

### Expanding the Filesystem

The first thing we need to do is expand the filesystem. Currently we are using SD cards of 8GB or 16GB but the root file system only takes up about 3GB (with more than 80% is used). So to expand the root filesystem to the full SD card we can use the “Expand Filesystem” configuration script. You will need to reboot the Raspberry Pi to make this available.

### Change User Password

The “Change User Password” tool allows you to change the default password of the pi user. Make sure to do this before continuing. Watch out if you do this using an external keyboard as the keyboard layout may be configured to qwerty. In this case configure the keyboard layout first by selecting the internationalization menu option.

### Enable Boot to Desktop / Scratch

You can change what happens when your Pi boots. Use this option to change your boot preference to command line, desktop, or straight to Scratch. In our case we will be using the command line interface.

### Internationalization Options

This will open up a sub menu with internationalization options to configure.

#### Change Locale

Locales are a framework to switch between multiple languages and allow users to use their language, country, characters, collation order, etc.

In the first screen you will be asked which locales to generate. UTF-8 locales should be chosen by default, particularly for new installations. Other character sets may be useful for backwards compatibility with older systems and software.

Select both “en\_GB.UTF-8 UTF-8” and “nl\_BE.UTF-8 UTF-8”. On the next screen you can pick any of these two to be the default locale.

#### Change Timezone

The time zone should be changed to “Europe – Brussels” to reflect our own time zone.

#### Change Keyboard Layout

This option opens another menu which allows you to select your keyboard layout. It will take a long time to display while it reads all the keyboard types. Changes usually take effect immediately, but may require a reboot.

This option however does not seem to work with the current version of the configuration tool.

You can however change the keyboard layout to a typical Belgian azerty layout by executing the following command the next time you get to the command line interface:

$ sudo setxkbmap be

### Enable Camera

In order to use the Raspberry Pi camera module, you must enable it here. This option will also make sure at least 128MB of RAM is dedicated to the GPU.

We will come back to this later.

### Add to RasTrack

Rastrack is a user-contributed Google Map to which Pi users in the community have added their location; it shows a heat map of where Pi users are known to be around the world. This was set up by young Pi enthusiast Ryan Walmsley in 2012. Rastrack is located at rastrack.co.uk.

Skip this option for the LAB.

### Overclock

It is possible to overclock your Raspberry Pi's CPU. The default is 700MHz but it can be set up to 1000MHz. The overclocking you can achieve will vary; overclocking too high may result in instability.

Selecting this option shows the following warning:

Be aware that overclocking may reduce the lifetime of your Raspberry Pi. If overclocking at a certain level causes system instability, try a more modest overclock. Hold down `shift` during boot to temporarily disable overclock.

For our LABs we will leave the Pi with its default settings.

### Advanced Options

The advanced options allow the configuration of the SSH daemon, the hostname, the division of the memory with the GPU and so on.

#### Overscan

Old TV sets had a significant variation in the size of the picture they produced; some had cabinets that overlapped the screen. TV pictures were therefore given a black border so that none of the picture was lost; this is called overscan. Modern TVs and monitors don't need the border, and the signal doesn't allow for it. If the initial text shown on the screen disappears off the edge, you need to enable overscan to bring the border back.

On some displays, particularly monitors, disabling overscan will make the picture fill the whole screen and correct the resolution. For other displays, it may be necessary to leave overscan enabled and adjust its values.

Any changes will take effect after a reboot.

#### Hostname

Set the visible name for this Pi on a network. Pick a unique name, different from the other students.

#### Memory Split

Change the amount of memory made available to the GPU (Graphics Processing Unit). For the initial LAB exercises you can change this to 16MB.

#### SSH

Enable/disable remote command line access to your Pi using SSH.

SSH allows you to remotely access the command line of the Raspberry Pi from another computer. Disabling this ensures the SSH service does not start on boot, freeing up processing resources. Note that SSH is enabled by default. If connecting your Pi directly to a public network, you should disable SSH unless you have set up secure passwords for all users.

Make sure to leave this enabled or you will be required to reconfigure your Pi using an external display and keyboard or by using a serial connection.

#### SPI (Serial Peripheral Interface)

Enable/disable automatic loading of SPI kernel module, needed for products such as PiFace.

#### Audio

Force audio out through HDMI or a 3.5mm jack.

#### Update

Update this tool to the latest version. This requires an active internet connection.

## Checking the configuration

At this point you should be able to login to the Pi using the pi user with the new password. If you execute the disk free command you should get a similar output to the one shown in Figure 12.

$ df –h

# df stands for disk free and the –h option requests a human readable output formatting.

Notice how the disk size has increased and the used disk space percentage has dropped significantly.

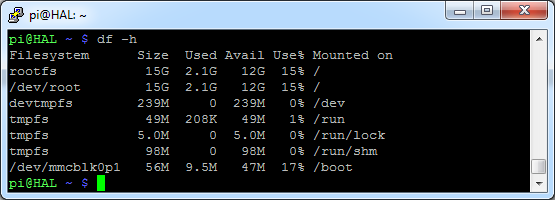


Figure 15 Output of the disk free command after initial configuration

Also notice how the hostname of the device changed (HAL[[11]](#footnote-11) in this case). Rescanning the subnet using SoftPerfect Network Scanner should result in systems with all different hostnames.

# Linux Basics

While the information and examples presented here will be valid for most Linux distro’s, some information and commands may only work for Debian based distro’s.

## The MAN-Pages

The most important command you need to know is the “man” command, which provides an interface to the online reference manuals. By adding a command after the man command you can consult the man-pages for the particular command. These pages provide all the information you need to know to use the command such as general information, a detailed description of the arguments and usage examples. Let’s see an example:

$ man ls

Which gives the output shown in Figure 16. We can for example see if we add “-a” after the “ls” command it will also display hidden files[[12]](#footnote-12). You can scroll through the man-pages using the arrow keys.

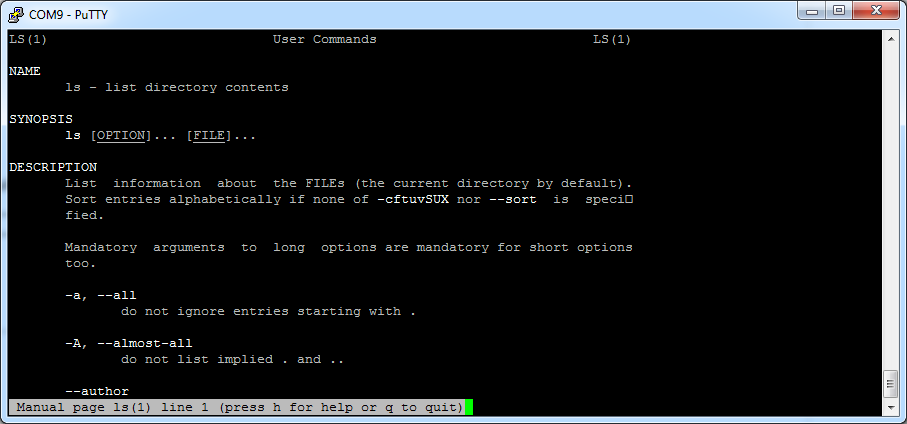


Figure 16 Output of "man ls" command

Searching the current man-page can be done by first typing a slash (“/”), followed by your search term. Jumping to the next hit can be done by hitting the “n” key, while jumping back is done with “SHIFT-n”.

Exiting the man-pages is achieved using the “CTRL-c” combination.

## The Linux Filesystem[[13]](#footnote-13)

### Some Brief Notes on the History of the Linux Filesystem Layout

Linux inherits many of its concepts of filesystem organization from its Unix predecessors. As far back as 1979, Unix was establishing standards to control how compliant systems would organize their files.

The Linux Filesystem Hierarchy Standard[[14]](#footnote-14), or FHS for short, is a prescriptive standard maintained by the Linux Foundation that establishes the organizational layout that Linux distributions should uphold for interoperability, ease of administration, and the ability to implement cross-distro applications reliably.

One important thing to mention when dealing with these systems is that Linux implements just about everything as a file. This means that a text file is a file, a directory is a file (simply a list of other files), a printer is represented by a file (the device drivers can send anything written to the printer file to the physical printer), etc.

Although this is in some cases an oversimplification, it informs us of the approach that the designers of the system encouraged: passing text and bytes back and forth and being able to apply similar strategies for editing and accessing diverse components.

### Traversing the Filesystem

Before actually delving into the filesystem layout, you need to know a few basics about how to navigate a filesystem from the command line. We will cover the bare minimum here to get you on your feet.

The first thing you need to do is orient yourself in the filesystem. There are a few ways to do this, but one of the most basic is with the pwd command, which stands for "print working directory":

pi@HAL:~$ pwd

/home/pi

This simply returns the directory you are currently located in.

To see what files are in the current directory, you can issue the *ls* command, which stands for "list":

pi@HAL:/$ ls

bin dev home lost+found mnt proc run selinux sys usr

boot etc lib media opt root sbin srv tmp var

This will tell you all directories and files in your current directory.

The *ls* command can take some optional flags. Flags modify the commands default behavior to either process or display the data in a different way.

The two most common flags are probable *-l* and *-a*. The first flag forces the command to output information in long-form:

pi@HAL:/$ ls -l

total 88

drwxr-xr-x 2 root root 4096 Jun 20 10:55 bin

drwxr-xr-x 2 root root 16384 Jan 1 1970 boot

drwxr-xr-x 12 root root 3060 Sep 24 13:31 dev

drwxr-xr-x 99 root root 4096 Sep 24 13:31 etc

drwxr-xr-x 3 root root 4096 Jun 20 07:48 home

drwxr-xr-x 12 root root 4096 Jun 20 10:42 lib

-rw-r--r-- 1 root root 0 Sep 24 13:37 log.txt

drwx------ 2 root root 16384 Jun 20 07:34 lost+found

drwxr-xr-x 2 root root 4096 Jun 20 07:36 media

...

This produces output with one line for each file or directory (the name is on the far right). This has a lot of information that we are not interested in right now. One part we are interested in though is the very first character, which tells us what kind of file it is. The three most common types are:

* -: Regular file
* d: Directory (a file of a specific format that lists other files)
* l: A hard or soft link (basically a shortcut to another file on the system)

The -a flag lists all files, including hidden files. In Linux, files are hidden automatically if they begin with a dot:

pi@HAL:/$ ls -a

. bin dev home log.txt media opt root sbin srv tmp var

.. boot etc lib lost+found mnt proc run selinux sys usr

The first two entries, . and .. are special. The . directory is a shortcut that means "the current directory". The .. directory is a shortcut that means "the current directory's parent directory".

Now that you can find out where you are in the filesystem and see what is around you, it is time to learn how to move throughout the filesystem.

To change to a different directory, you issue the *cd* command, which stands for "change directory":

pi@HAL:/$ cd /home

You can follow the command with either an absolute or a relative pathname.

* An **absolute path** is a file path that specifies the location of a directory from at the top of the directory tree. Absolute paths begin with a "/", as you see above.
* A **relative path** is a file path that is relative to the current working directory. This means that instead of defining a location from the top of the directory structure, it defines the location in relation to where you currently are.

For instance, if you want to move to a directory within the current directory called documents, you can issue this command:

pi@HAL:/home$ cd pi

The lack of the "/" from the beginning tells to use the current directory as the base for looking for the path.

This is where the .. directory comes in handy. To move to the parent directory of your current directory, you can type:

pi@HAL:~$ cd ..

There is also a shortcut to specify your own homedir, and that is by using the tilde “~”. You can immediately jump to your homedir by for example executing the following command:

$ pi@HAL:/$ cd ~

### An Overview of the Linux Filesystem Layout

The first thing you need to know when viewing a Linux filesystem is that the filesystem is contained within a single tree, regardless of how many devices are incorporated.

What this means is that all components accessible to the operating system are represented somewhere in the main filesystem. If you use Windows as your primary operating system, this is different from what you are used to. In Windows, each hard drive or storage space is represented as its own filesystem, which are labeled with letter designations (C: being the standard top-level directory of the system file hierarchy and additional drives or storage spaces being given other letter labels).

In Linux, every file and device on the system resides under the "root" directory, which is denoted by a starting "/".

Thus, if we want to go to the top-level directory of the entire operating system and see what is there, we can type:

pi@HAL:/home$ cd /

Every file, device, directory, or application is located under this one directory. Under this, we can see the beginnings of the rest of the directory structure.

One of the principles guiding the organization of the file system is to allow it to be split across multiple disk partitions (or multiple disks) in a rational manner, and to allow appropriate pieces of it to be shared between machines. Key to this is the notion of the root partition (/, the parent of the entire filesystem).

When Linux boots, the kernel attaches a single file system partition all by itself. This is known as the root partition. Any other partitions that need to be attached are mounted by the mount command, usually under control of entries in the file “/etc/fstab”. Because in the early stages of startup, only the root file system is available, it must contain everything needed for the system to function and attach the other pieces of the file system.

Tools on the root partition include the init program (which starts all the other processes), a shell, mount and the “/etc/fstab” file. The File System Hierarchy standard specifies a number of directories that must lie within the root partition.

Figure 17 shows a typical Linux file system hierarchy.

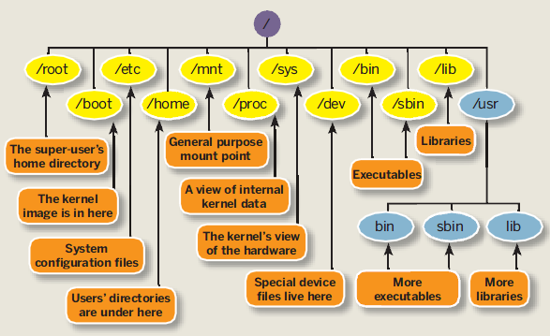


Figure 17 A typical Linux file system hierarchy

Let’s take a closer look at the different directories which can be found under root.

**/bin** - This directory contains basic commands and programs that are needed to achieve a minimal working environment upon booting. These are kept separate from some of the other programs on the system to allow you to boot the system for maintenance even if other parts of the filesystem may be damaged or unavailable. If you search this directory, you will find that both *ls* and *pwd* reside here. The *cd* command is actually built into the shell we are using (bash), which is in this directory too.

**/boot** - This directory contains the actual files, images, and kernels necessary to boot the system. While /bin contains basic, essential utilities, /boot contains the core components that actually allow the system to boot. If you need to modify the bootloader on your system, or if you would like to see the actual kernel files and initial ramdisk (initrd), you can find them here. This directory must be accessible to the system very early on.

**/dev** - This directory houses the files that represent devices on your system. Every hard drive, terminal device, input or output device available to the system is represented by a file here. Depending on the device, you can operate on the devices in different ways. For instance, for a device that represents a hard drive, like /dev/sda, you can mount it to the filesystem to access it. On the other hand, if you have a file that represents a line printer like /dev/lpr, you can write directly to it to send the information to the printer.

**/etc[[15]](#footnote-15)** - This is one area of the filesystem where you will spend a lot of time if you are working as a system administrator. This directory is basically a configuration directory for various system-wide services. By default, this directory contains many files and subdirectories. It contains the configuration files for most of the activities on the system, regardless of their function. In cases where multiple configuration files are needed, many times an application-specific subdirectory is created to hold these files. If you are attempting to configure a service or program for the entire system, this is a great place to look.

**/home** - This location contains the home directories of all of the users on the system (except for the administrative user, root). If you have created other users, a directory matching their username will typically be created under this directory. Inside each home directory, the associated user has write access. Typically, regular users only have write access to their own home directory. This helps keep the filesystem clean and ensures that not just anyone can change important configuration files.

Within the home directory, that are often hidden files and directories (represented by a starting dot) that allow for user-specific configuration of tools. You can often set system defaults in the /etc directory, and then each user can override them as necessary in their own home directory.

**/lib** - This directory is used for all of the shared system libraries that are required by the /bin and /sbin directories. These files basically provide functionality to the other programs on the system. This is one of the directories that you will not have to access often.

**/lost+found** - This is a special directory that contains files recovered by /fsck, the Linux filesystem repair program. If the filesystem is damaged and recovery is undertaken, sometimes files are found but the reference to their location is lost. In this case, the system will place them in this directory.

In most cases, this directory will remain empty. If you experience corruption or any similar problems and are forced to perform recovery operations, it's always a good idea to check this location when you are finished.

**/media** - This directory is typically empty at boot. Its real purpose is simply to provide a location to mount removable media (like cd’s). If your Linux operating system ever mounts a media disk and you are unsure of where it placed it, this is a safe bet.

**/mnt -** This directory is similar to the /media directory in that it exists only to serve as an organization mount point for devices. In this case, this location is usually used to mount filesystems like external hard drives, etc.

This directory is often used in a VPS environment for mounting network accessible drives. If you have a filesystem on a remote system that you would like to mount on your server, this is a good place to do that.

**/opt** - This directory's usage is rather ambiguous. It is used by some distributions, but ignored by others. Typically, it is used to store optional packages. In the Linux distribution world, this usually means packages and applications that were not installed from the repositories.

For instance, if your distribution typically provides the packages through a package manager, but you installed program X from source, then this directory would be a good location for that software. Another popular option for software of this nature is in the /usr/local directory.

**/proc -** The /proc directory is actually more than just a regular directory. It is actually a pseudo-filesystem of its own that is mounted to that directory. The proc filesystem does not contain real files, but is instead dynamically generated to reflect the internal state of the Linux kernel.

This means that we can check and modify different information from the kernel itself in real time. For instance, you can get detailed information about the memory usage by typing

pi@HAL:/$ cat /proc/cpuinfo

processor : 0

model name : ARMv6-compatible processor rev 7 (v6l)

Features : swp half thumb fastmult vfp edsp java tls

CPU implementer : 0x41

CPU architecture: 7

CPU variant : 0x0

CPU part : 0xb76

CPU revision : 7

Hardware : BCM2708

Revision : 000e

Serial : 000000008d79b8e3

**/root -** This is the home directory of the administrative user (called "root"). It functions exactly like the normal home directories, but is housed here instead.

**/run -** This directory is for the operating system to write temporary runtime information during the early stages of the boot process.

**/sbin -** This directory is much like the /bin directory in that it contains programs deemed essential for using the operating system. The distinction is usually that /sbin contains commands that are available to the system administrator, while the other directory contains programs for all of the users of the system.

**/selinux -** This directory contains information involving security enhanced Linux. This is a kernel module that is used to provide access control to the operating system.

**/srv** - This directory is used to contain data files for services provided by the system. In most cases, this directory is not used too much because its functionality can be implemented elsewhere in the filesystem.

**/tmp -** This is a directory that is used to store temporary files on the system. It is writable by anyone on the computer and does not persist upon reboot. This means that any files that you need just for a little bit can be put here. They will be automatically deleted once the system shuts down.

**/usr[[16]](#footnote-16)** **-** This directory is one of the largest directories on the system. It basically includes a set of folders that look similar to those in the root / directory, such as /usr/bin and /usr/lib. This location is basically used to store all non-essential programs, their documentation, libraries, and other data that is not required for the most minimal usage of the system.

This is where most of the files on the system will be stored. Some important subdirectories are /usr/local, which is an alternative to the /opt directory for storing locally compiled programs. Another interesting thing to check out is the /usr/share directory, which contains documentation, configuration files, and other useful files.

**/var -** This directory is supposed to contain variable data. In practice, this means it is used to contain information or directories that you expect to grow as the system is used.

For example, system logs and backups are housed here. Another popular use of this directory is to store web content if you are operating a web server.

### overview of Basic Filesystem commands

The most used commands to traverse and manipulate the file system of a Linux system are listed in Table 2. You can always use the man-command to get a detailed description.

Table 2 Basic Linux commands

|  |  |
| --- | --- |
| Command | Description |
| ls | List files |
| cp | Copy files |
| rm | Remove files |
| mv | Move files |
| cd | Change working dir |
| chmod | Change file permission mode |
| chown | Change owner of file |
| cat | Concatenate files and output to terminal |
| touch | Create an empty file |
| mkdir | Make directory |

## Debian and it’s Packages

Packages generally contain all of the files necessary to implement a set of related commands or features. There are two types of Debian packages:

* **Binary packages**, which contain executables, configuration files, man/info pages, copyright information, and other documentation. These packages are distributed in a Debian-specific archive format; they are usually distinguished by having a '.deb' file extension. Binary packages can be unpacked using the Debian utility *dpkg* (possibly via a frontend like aptitude).
* **Source packages**, which consist of a .dsc file describing the source package (including the names of the following files), a .orig.tar.gz file that contains the original unmodified source in gzip-compressed tar format and usually a .diff.gz file that contains the Debian-specific changes to the original source. The utility *dpkg-source* packs and unpacks Debian source archives. (The program apt-get can be used as a frontend for dpkg-source.)

Installation of software by the package system uses "dependencies" which are carefully designed by the package maintainers. These dependencies are documented in the control file associated with each package.

For example, the package containing the GNU C compiler (gcc) "depends" on the package binutils which includes the linker and assembler. If a user attempts to install gcc without having first installed binutils, the package management system (dpkg) will send an error message that it also needs binutils, and stop installing gcc.

There are multiple tools that are used to manage Debian packages, from graphic or text-based interfaces to the low level tools used to install packages. All the available tools rely on the lower level tools to properly work.

It is important to understand that the higher level package management tools such as *aptitude* or *dselect* rely on *apt* which, itself, relies on *dpkg* to manage the packages in the system.

**dpkg** is the main package management program.

**APT** is the Advanced Package Tool and provides the *apt-get* program. apt-get provides a simple way to retrieve and install packages from multiple sources using the command line. Unlike dpkg, apt-get does not understand .deb files, it works with the packages proper name and can only install .deb archives from a source specified in /etc/apt/sources.list. apt-get will call dpkg directly after downloading the .deb archives from the configured sources.

Some common ways to use apt-get are:

* To update the list of package known by your system, you can run:

$ apt-get update

(you should execute this regularly to update your package lists)

* To upgrade all the packages on your system (without installing extra packages or removing packages), run:

$ apt-get upgrade

* To install the foo package and all its dependencies, run:

$ apt-get install foo

* To remove the foo package from your system, run:

$ apt-get remove foo

* To remove the foo package and its configuration files from your system, run:

$ apt-get --purge remove foo

* To upgrade all the packages on your system, and, if needed for a package upgrade, installing extra packages or removing packages, run:

$ apt-get dist-upgrade

**aptitude** is a package manager for Debian GNU/Linux systems that provides a frontend to the apt package management infrastructure. aptitude is a text-based interface using the curses library, it can be used to perform management tasks in a fast and easy way.

* + aptitude offers easy access to all versions of a package.
  + aptitude makes it easy to keep track of obsolete software by listing it under "Obsolete and Locally Created Packages".
  + aptitude includes a fairly powerful system for searching particular packages and limiting the package display. Users familiar with mutt will pick up quickly, as mutt was the inspiration for the expression syntax.
  + aptitude can be used to install the predefined tasks[[17]](#footnote-17) available.
  + aptitude in full screen mode has su functionality embedded and can be run by a normal user. It will call su (and ask for the root password, if any) when you really need administrative privileges

Aptitude is most commonly used for searching for packages. You can use the following command for this:

pi@HAL:/$ aptitude search foobar

**Assignment**

Install the git package and create a local clone inside your homedir (currently the user pi) of the repository <https://github.com/BioBoost/multimediatechnieken>

**Assignment**

Install the apache package and find out where the webpages are stored. Make sure you can view your PI’s website from your host machine. Change the index.html page (you can use the *nano* editor for this) and add some cool things to it.

# Running a Virtual Machine

A virtual machine (VM) is an emulation of a particular computer system. This system can be based on an existing or hypothetical machine. As a user we can create such virtual machines and install an operating system of choice on them. This allows us to run a Linux distribution while working on a Windows machine and vice versa.

Several software packages are available to create and run virtual machines. Examples are VMware, Hyper-V[[18]](#footnote-18) which comes with Windows 8, Oracle VirtualBox, … For our labs we will be using VirtualBox as this is free, lightweight, easy to use and available for Windows, Linux, Mac and Solaris.

## Installing Oracle VirtualBox

Start by going to the download section of the website of VirtualBox (<https://www.virtualbox.org/>). Download the VirtualBox platform package for your system. At the moment of this writing the current version of VirtualBox is 4.3.16.

When running the installer package make sure to install VirtualBox with all features enabled as shown in Figure 18.



Figure 18 Installing VirtualBox with all features enabled

Next we also need to install the extension pack which introduces USB2.0 support and some other extra features. You can download the extension pack on the same page as you downloaded the installer for VirtualBox. Just make sure to pick the correct version for your current VirtualBox version.

The installer of VirtualBox should also have created a virtual network adapter (such as can be seen in xxxxxxxx) which is used for private networking between the host machine and the virtual machine.

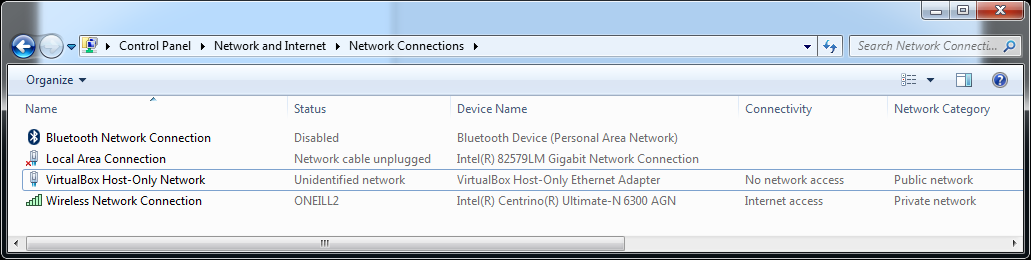


Figure 19 VirtualBox virtual network adapter

Once you’re finished you can start the VirtualBox client and you should get the interface presented in Figure 19.

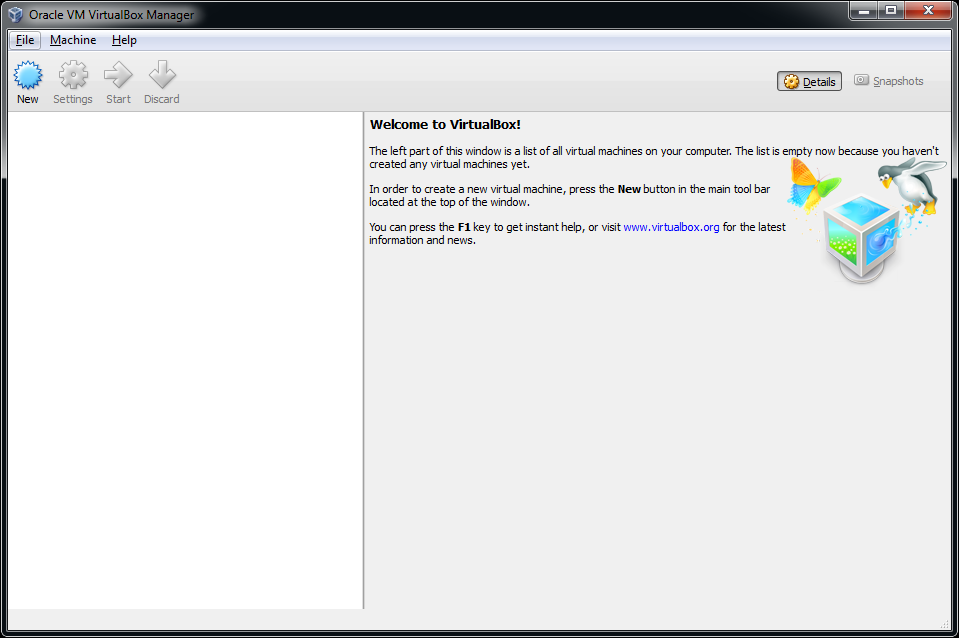


Figure 20 Launching VirtualBox after installation

Under *Files => Preferences => General* you can change the default path for your virtual machines. Do take note that you will need about 25GB of free space for each VM. For these labs you will most likely only need 1 VM.

Under *Files => Preferences => Language* you can also change the default interface language if you wish.

## Creating a Virtual Machine

Creating a virtual machine is very simple as it just following the steps presented to you by the wizard. To start the process of creating a VM hit the *New* button on the main interface of VirtualBox.

The first step consist of giving your VM a name and selecting the operating system you will be running on the VM as shown in Figure 21. In our case we will use Ubuntu 12.04 LTS[[19]](#footnote-19) 64 bit Desktop edition. More on this later.

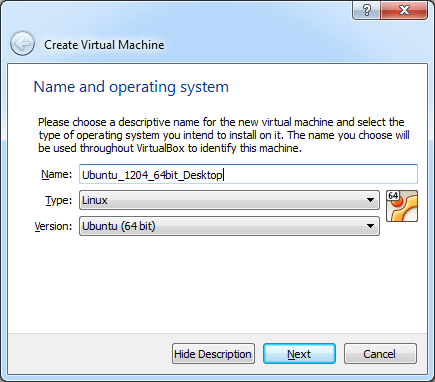


Figure 21 Creating a VM - Step 1 - The name and OS

In step 2 we need to select the amount of memory we want to assign to the virtual machine (as seen in xxxxxxxxx). The recommended amount is 512MB. However if you have more than 4GB, select 1024MB or even 2048MB, which will improve the responsiveness and performance of the VM.

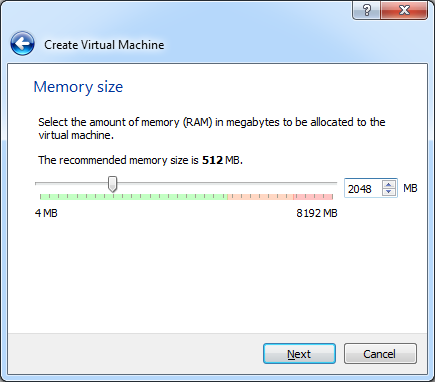


Figure 22 Creating a VM - Step 2 - Amount of memory

Next we need to choose what if we want to create a new or use an existing virtual hard drive. Pick the option to create one now as depicted in Figure 23.This will launch another wizard which will lead us through the creation process of a virtual drive.

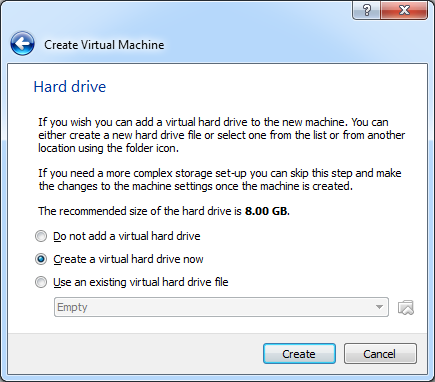


Figure 23 Creating a VM - Step 3 - Virtual hard drive

The first screen will allow us to select what type of virtual drive file we want to create. Just leave the default option (VDI – Virtual Disk Image) as shown in Figure 24.

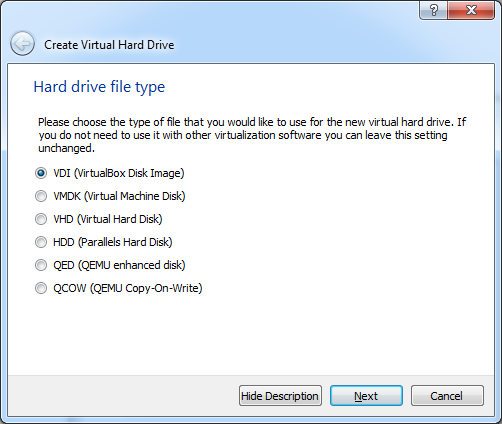


Figure 24 Creating a VM - Step 3a - Hard drive file type

In the next step we get the option to create a dynamically allocated image or a fixed size image as can be seen in Figure 25. A fixed size image is faster but will take up the full space we select for the size of the virtual drive. A dynamically allocated image is faster but will only grow in size when needed. You will need to decide this for yourself based on the free space available on your host system.

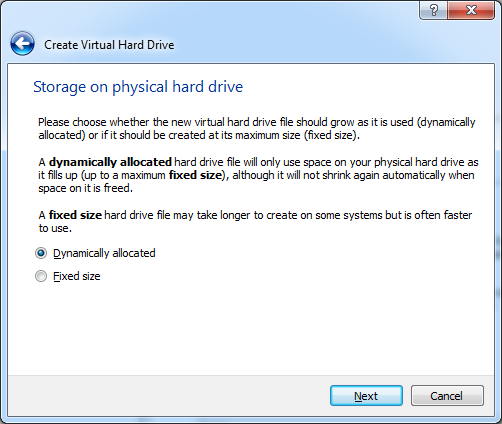


Figure 25 Creating a VM - Step 3b - Virtual drive allocation method

Now we need to select the hard drive file location (leave it as is) and size of the drive. Make sure to select at least 25GB for the size as shown in Figure 26. Hitting create will finish the process of creating a VM.

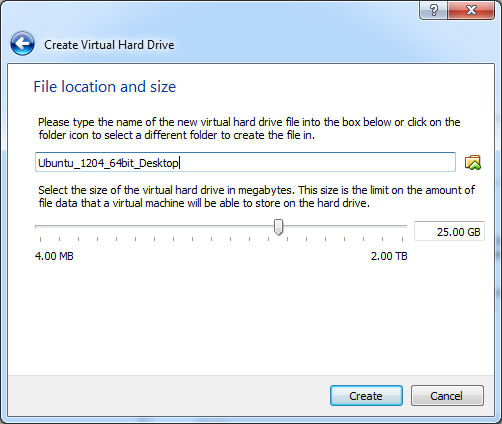


Figure 26 Creating a VM - Step 3c - Location and size of the virtual drive

Your new VM should now appear in the list of VM’s on the left side of the VirtualBox main interface as seen in Figure 27. Selecting a VM in the list also displays some basic information about the VM.

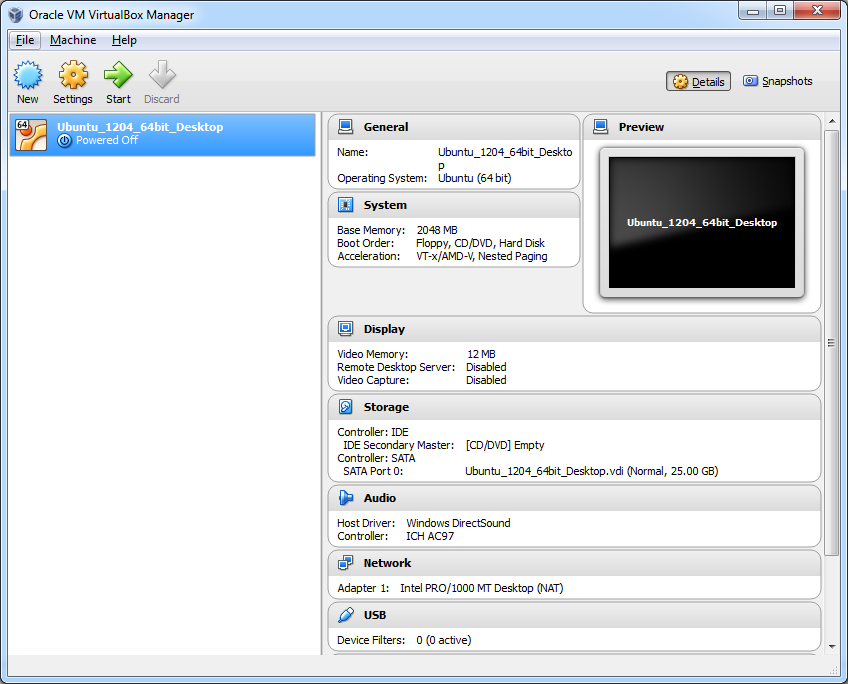


Figure 27 A new VM is added to your current list of VM’s

## Configuring the Virtual Machine

Before installing an operating system on the newly created VM it is necessary to make a few configuration changes. Select the new VM and hit the *Settings* button on the main interface. You will be presented with the configuration settings of your VM.

Start by going to General => Advanced and enabling the bidirectional shared clipboard. This allows text to be copied to the clipboard in the VM and pasted in your host OS and vice versa. Also enable bidirectional Drag’n Drop. The resulting configuration is shown in Figure 28.

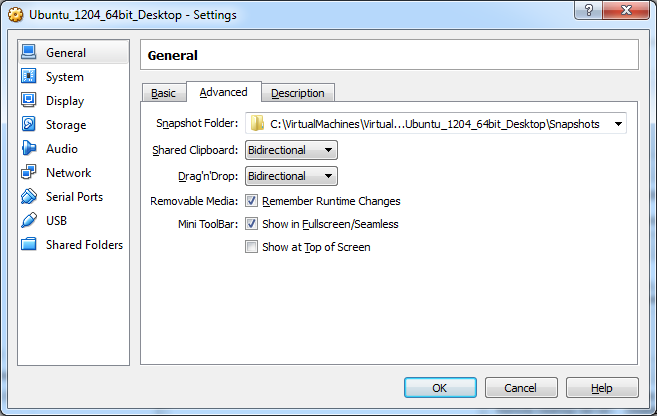


Figure 28 Configure VM to allow bidirectional clipboard and drag'n drop

In some cases it is also necessary to change the networking configuration of the VM. Default the VM is configured with a single network adapter with NAT (Network Address Translation) enabled as shown in Figure 29. This means that the VM has access to the network and also has access to the Internet. However because of NAT we will not be able to connect to the VM from another machine using SSH without configuring port forwarding. However since this will not be needed for these LAB’s we can leave the standard behavior.

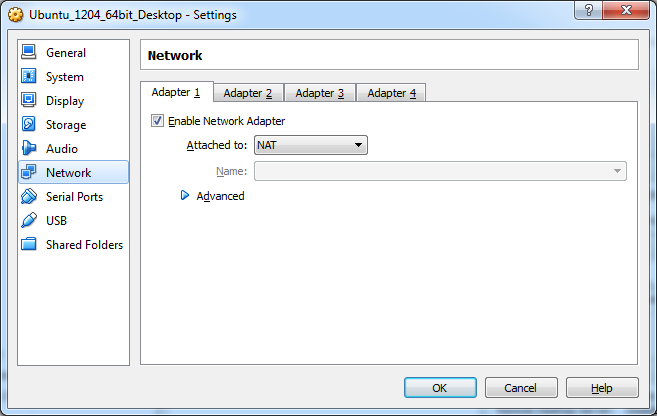


Figure 29 Configure VM to use NAT with the network adapter

Another much used option is a bridged adapter (shown in Figure 30). This will basically create a network bridge between the VM’s network adapter and your physical host adapter making your VM’s directly available on your network. This may be a security issue but can also simplify working with your VM. This option also implies that your VM will get its IP address from the same DHCP (Dynamic Host Configuration Protocol) server as your host machine if you have a DHCP enabled network.

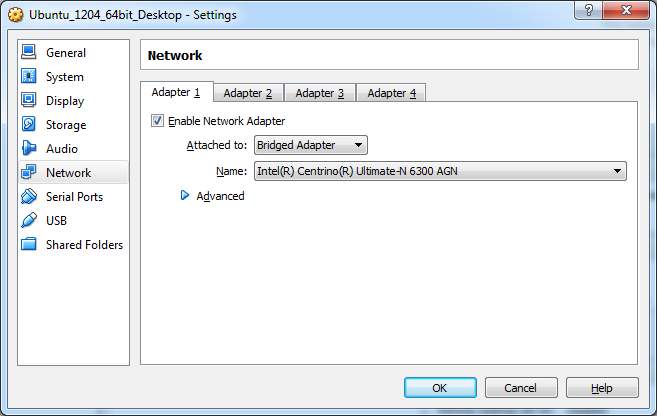


Figure 30 Configuring a VM to use a bridged network adapter

Skim through the rest of the configuration options and check out some of the available options.

## Installing the Ubuntu Operating System

Before we can start the install procedure of the operating system, we will have to download an installer image. This image can then be mounted on our VM allowing us to boot from it.

For our LABs we will be using Ubuntu 12.04 LTS (64 bit) (Precise Pangolin) which can be downloaded here <http://releases.ubuntu.com/precise/>. Make sure to select the 64-bit Desktop edition.

Once downloaded start VirtualBox and open the setting of your VM. Next open the *storage* settings. Now select the virtual CD/DVD drive below the IDE controller as shown in step 1 of Figure 31.

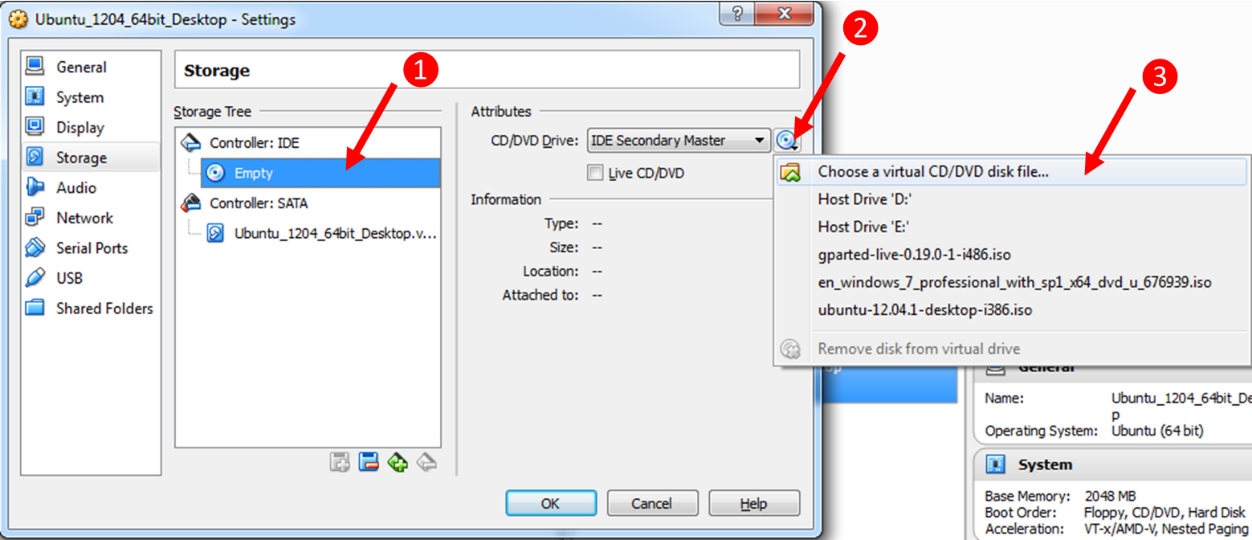


Figure 31 Steps for mounting an image in VirtualBox

Hit the small CD/DVD icon next to the *IDE Secondary Master* dropdown (step 2 in Figure 31) and select *Choose a virtual CD/DVD disk file…*. A browse window will open; select the image file you downloaded from the Ubuntu website and hit ok. Hit the ok button of the setting panel to close it.

Ready ? Then hit the start button of the VM and follow the steps for installing the Ubuntu operating system. Do make sure once you see the output shown in Figure 32 you hit a key on the keyboard or the Live DVD mode will boot. If you missed it, you can always reset the VM from the *Machine* menu in VirtualBox.

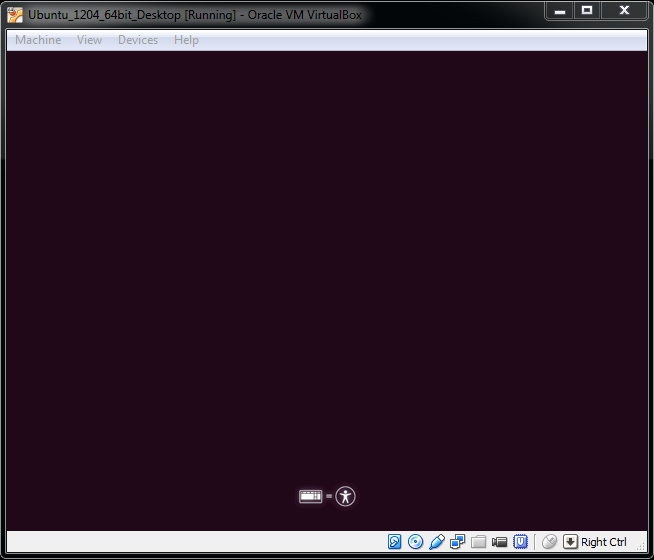


Figure 32 Live DVD or boot menu option

From this point on all steps are self-explanatory.

If you click inside the VM window your mouse cursor will automatically be captured. Releasing your cursor can be achieved using the right CTRL key.

Once the installation procedure is finished you will be asked to reboot the VM.

Login to the desktop. You should be presented with a popup window from the Update Manager after a few seconds. If not just launch the Update Manager from the menu. It’s always a good idea to keep your distribution up-to-date. So if you have some time and an active internet connection, you can hit the *Install Updates* button.

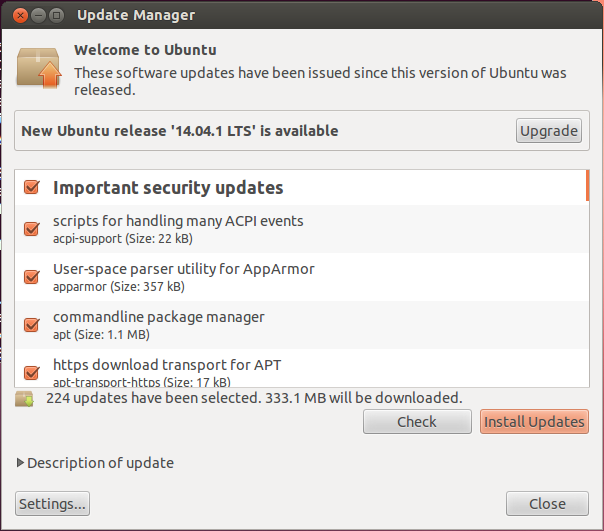


Figure 33 Ubuntu Update Manager

### Installing Guest Additions

You may or may not have noticed that your mouse movement is a bit sluggish within the VM. That’s because the guest additions haven’t been installed yet.

Open the Devices menu which can be found at the top of the VM window (not in Ubuntu). Next select *Insert Guest Additions CD image…* as shown in Figure 33. A window in Ubuntu will open asking if you’d wish to run the package. Hit run and follow the instructions.

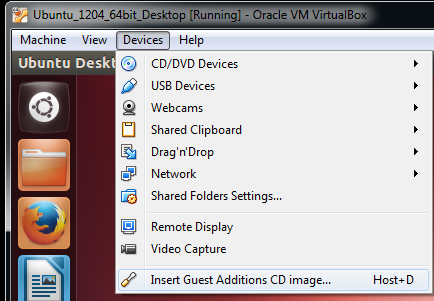


Figure 34 Inserting the Guest Additions for Ubuntu

Once finished remove the image from the virtual drive (through settings or through the small CD/DVD icon in the bottom right corner of your VM window).

If you update your machine it may be necessary to repeat this procedure.

## Creating a Shared Directory

Some files can be dragged and dropped between your host machine and the VM. However this does not seem to be possible for all file types. For these instances it is more easy to create a shared directory which can be accessed from the host and the VM.

Go to your VM’s settings and locate the *Shared Folders* options depicted in Figure 35.

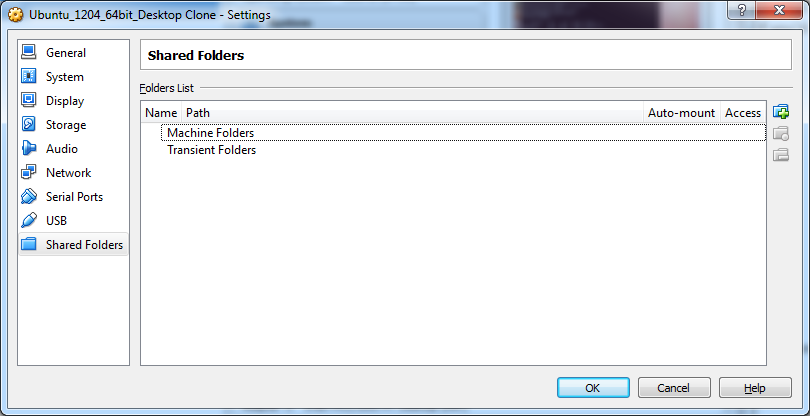


Figure 35 Shared folders settings of VM

Create a shared folder on your system and make sure to select the *Make Permanent* and *Auto-mount* options as shown in Figure 36.

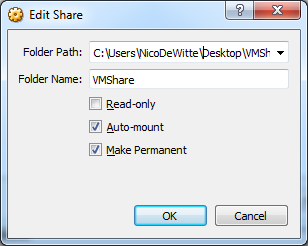


Figure 36 Creating a shared folder for your VM

The folder should now be automatically mounted under /media in your VM on your next reboot.

By adding your own user account to the user group *vboxsf* you get read and write access to the shared folder:

$ sudo usermod -a -G vboxsf nico

# nico should be replaced with your own user account

# Buildroot: making embedded Linux easy

Buildroot is a tool that simpliﬁes and automates the process of building a complete Linux system for an embedded system, using cross-compilation.

In order to achieve this, Buildroot is able to generate a cross-compilation toolchain, a root ﬁlesystem, a Linux kernel image and a bootloader for your target. Buildroot can be used for any combination of these options, independently (you can for example use an existing cross-compilation toolchain, and build only your root ﬁlesystem with Buildroot).

Buildroot is useful mainly for people working with embedded systems. Embedded systems often use processors that are not the regular x86 processors everyone is used to having in his PC. They can be PowerPC processors, MIPS processors, ARM processors, etc. Buildroot supports numerous processors and their variants; it also comes with default conﬁgurations for several boards available off-the-shelf.

Besides this, a number of third-party projects are based on, or develop their BSP[[20]](#footnote-20) or SDK[[21]](#footnote-21) on top of Buildroot.

The following sections will briefly discuss the different components required to build your own system.

## The Linux kernel

The Linux kernel provides the core system facilities required for any system based upon Linux to operate correctly. It has complete control over everything that occurs in the system. Application software relies upon specific features of the Linux kernel (as shown in Figure 35) such as its handling of hardware devices and its provision of many fundamental abstractions such as virtual memory, sockets, tasks (known as processes), files and many others.

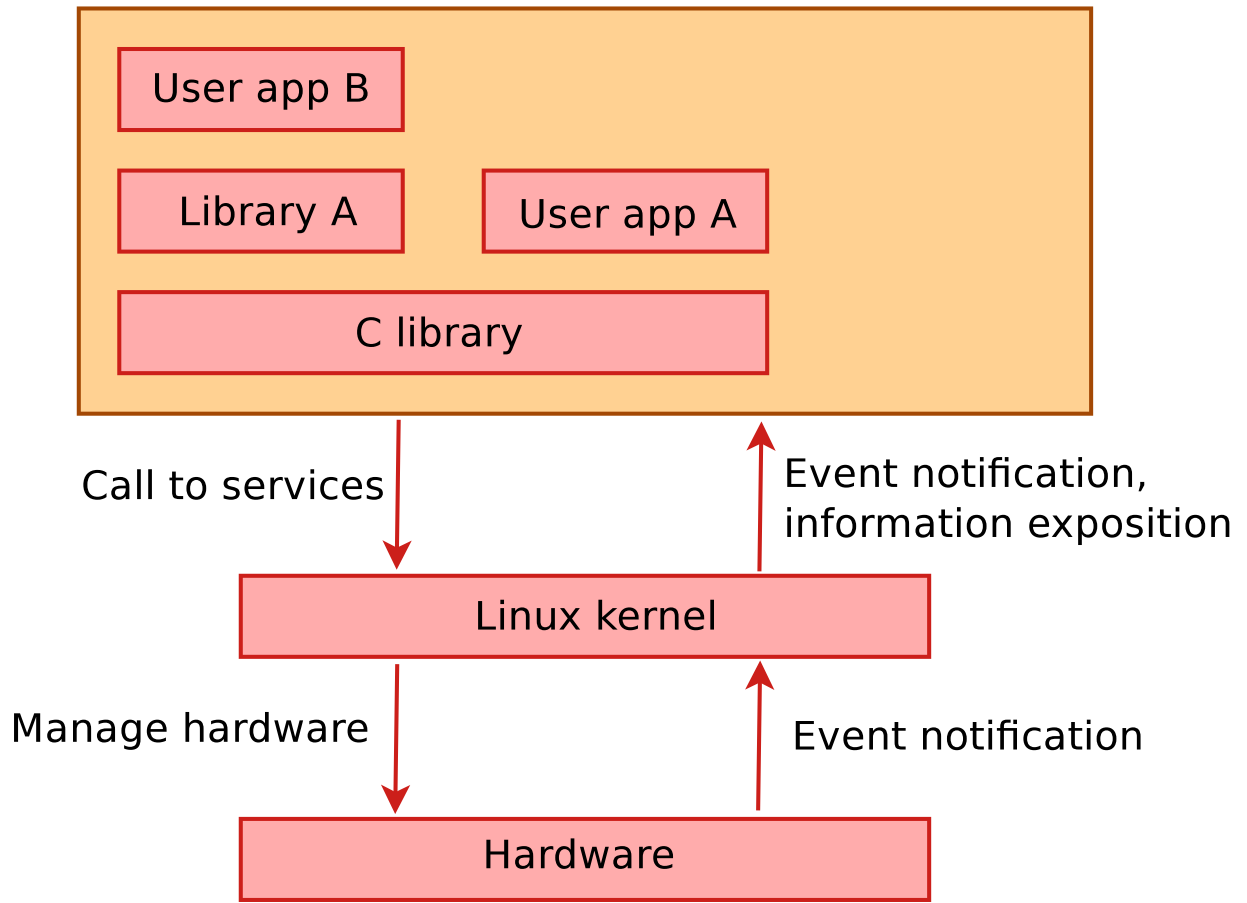


Figure 37 The Linux kernel provides services to user applications

The Linux kernel was originally created as a hobby project in 1991 by a Finnish student named Linus Torvalds. In the meantime the current version of the kernel is maintained and extended by thousands of users and developers.

When we talk about Linux we mostly mean a Linux distribution such as Red Hat, Suse, Ubuntu, … This is basically an installer which contains a Linux kernel packed together with a lot of applications.

When we are talking about embedded Linux we actually are talking about the same kernel code running on millions of other systems. There is no separate code base for embedded systems. When we however build a Linux system for an embedded target we do exclude features we won’t be using and we are also cross-compiling the kernel to binary code that can run on the target system.

Some key features of the kernel are:

* Portability and hardware support. It runs on most architectures.
* Scalability. Linux can run on super computers as well as on tiny devices (4 MB of RAM is enough).
* Compliance to standards and interoperability.
* Exhaustive networking support.
* Security. It can't hide its ﬂaws. Its code is reviewed by many experts.
* Stability and reliability.
* Modularity. Can include only what a system needs even at run time.
* Easy to program. You can learn from existing code. Many useful resources on the net.

The main roles of the kernel are:

* Manage all the hardware resources: CPU, memory, I/O.
* Provide a set of portable, architecture and hardware independent APIs to allow user space applications and libraries to use the hardware resources.
* Handle concurrent accesses and usage of hardware resources from diﬀerent applications.
* Example: a single network interface is used by multiple user space applications through various network connections. The kernel is responsible for "multiplexing" the hardware resource.

The main interface between the kernel and user space is the set of system calls that are provided by the kernel (about 300 system calls that provide the main kernel services) as shown in Figure 35.

These services include file and device operations, networking operations, inter-process communication, process management, memory mapping, timers, threads, synchronization primitives, etc (see Figure 36).

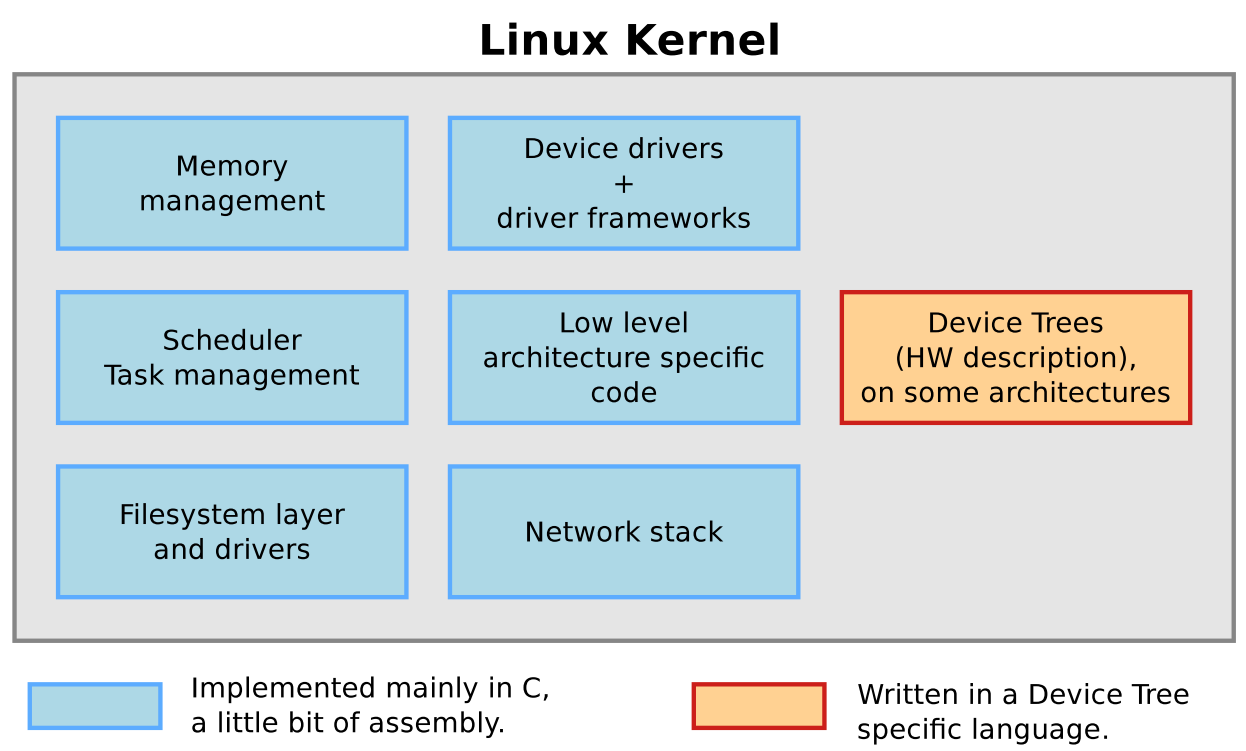


Figure 38 The Linux kernel services

Very important to know is that the kernel interface is stable over time. This basically means that only new system calls can be added by the kernel developers and no old calls can be removed. This means that applications running on an older kernel version should always work on a newer one.

The system call interface is actually wrapped by the C library and user space applications usually never make a system call directly but rather use the corresponding C library functions.

Check out <http://www.makelinux.net/kernel_map/> to see some prominent functions and structures used throughout the kernel.

Linux makes system and kernel information available in user space through pseudo ﬁlesystems, sometimes also called virtual ﬁlesystems. Pseudo ﬁlesystems allow applications to see directories and ﬁles that do not exist on any real storage: they are created and updated on the ﬂy by the kernel. The two most important pseudo ﬁlesystems are

* **proc**, usually mounted on /proc: Operating system related information (processes, memory management parameters...)
* **sysfs**, usually mounted on /sys: Representation of the system as a set of devices and buses. This virtual filesystem provides information about these devices.

## The cross-compilation toolchain

A compilation toolchain is the set of tools that allows you to compile code for your system. It consists of a compiler (in our case, gcc), binary utils like an assembler and a linker (in our case, binutils[[22]](#footnote-22)) and a C standard library (for example GNU Libc, uClibc).

The system installed on your development station (in our case this is the Ubuntu VM) certainly already has a compilation toolchain that you can use to compile an application that runs on your system. If you're using a PC, your compilation toolchain runs on an x86 processor and generates code for an x86 processor. Under most Linux systems, the compilation toolchain uses the GNU libc (glibc) as the C standard library. This compilation toolchain is called the "host compilation toolchain". The machine on which it is running, and on which you are working, is called the "host system".

The compilation toolchain is provided by your distribution, and Buildroot has nothing to do with it (other than using it to build a cross-compilation toolchain and other tools that are run on the development host).

As said above, the compilation toolchain that comes with your system runs on and generates code for the processor in your host system. As your embedded system has a different processor, you need a cross-compilation toolchain - a compilation toolchain that runs on your host system but generates code for your target system (and target processor). For example, if your host system uses x86 and your target system uses ARM, the regular compilation toolchain on your host runs on x86 and generates code for x86, while the cross-compilation toolchain runs on x86 and generates code for ARM.

Notice that the cross-compiler is being built twice. This is normal and required, because some languages supported by GCC (such as C++) require glibc support. Since this is not yet available at this point, a bootstrap compiler is built which only supports C. A full compiler is built after the glibc libraries are available.

Figure 37 through Figure 40 shows the steps necessary to build a cross-compilation toolchain. The last step (shown in Figure 41) is the compilation of the Linux kernel using our newly created cross-compilation toolchain.

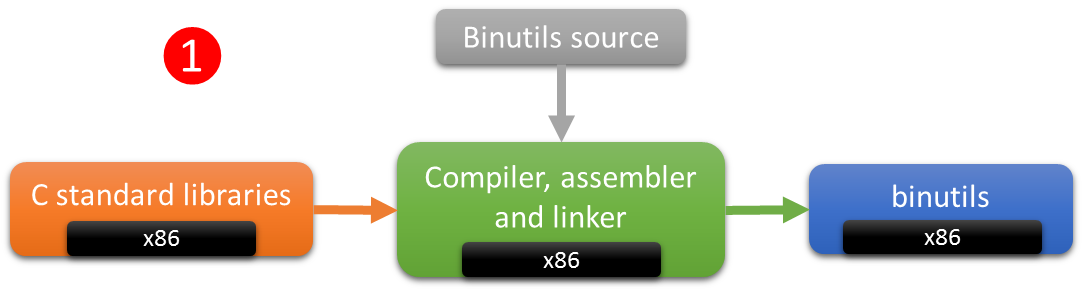


Figure 39 Building the toolchain - Step 1 - binutils

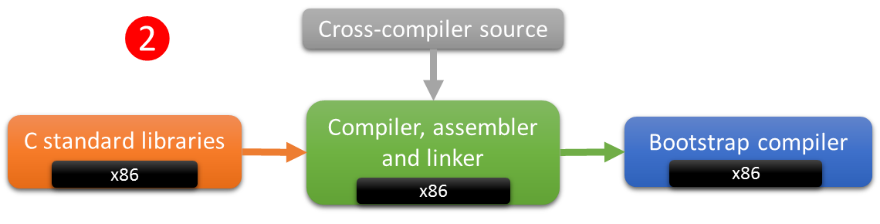


Figure 40 Building the toolchain - Step 2 - Bootstrap compiler

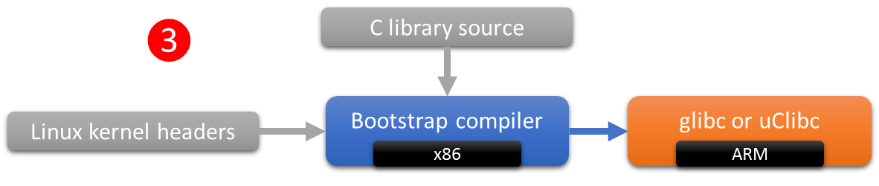


Figure 41 Building the toolchain - Step 3 - glibc or uClibc

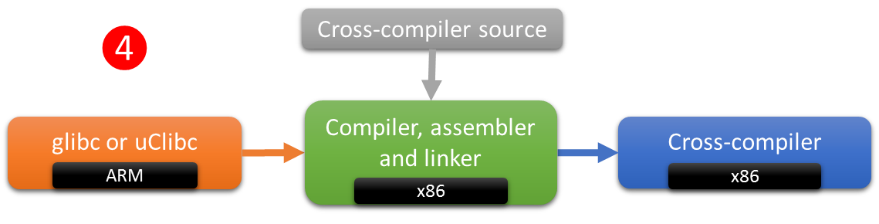


Figure 42 Building the toolchain - Step 4 - Full cross-compiler

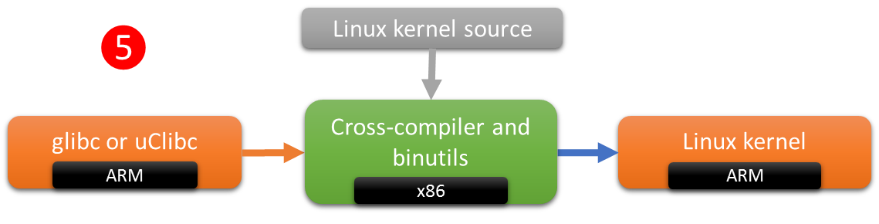


Figure 43 Compiling the Linux kernel

When building the cross-compilation toolchain we will have to make a choice on which standard C library we will be using. There are three popular ones, namely the GNU C library (glibc), uClibc and diet libc. The reason there are different options is because of embedded systems.

The GNU C library is a portable, high performance C library supporting all relevant standards (ISO C99, POSIX.1c, …). It also supports internationalization, sophisticated name resolution, time zone information and all sorts of authentication. However being rich and full-features, glibc is huge. To compound the problem for embedded systems, it is not very modular (removing features is a cumbersome and sometimes impossible job). Glibc designers and implementers have traditionally optimized for performance instead of resource use.

Glibc minimally takes up as much as 2MB of storage space. This does not seem that much but keep in mind that almost every application is compiled against the C library headers, meaning it will also affect the size of the application executable and libraries.

For this reason more compact and modular C library alternatives were developed. Of course the choice of C library effect the building process of the cross-compilation toolchain, our kernel and eventually every application compiled with the cross-compiler.

## Buildroot

You can actually build the cross-compilation toolchain all by yourself using the provided make scripts that come with each package and of course some extra configuration and path-setting. However this is quite an extensive job.

For this reason Buildroot was created. This allows us the easily configure all the different aspects of the toolchain, the kernel and even more. Once configured we can build everything with a single command.

Buildroot is made to be run on Linux so start up your newly created VM and let’s start hacking.

### Downloading the latest Buildroot

Before we can start we have to download the latest stable release of the Buildroot system. Just go to <http://buildroot.uclibc.org/download.html> (the current stable release is 2014.08). Download this archive to your home directory on your VM.

Open a terminal (CTRL – ALT – T or use the start menu). You can extract the Buildroot package using the following command:

$ tar -xvzf buildroot-2014.08.tar.gz

# x indicates an extraction

# v gives extra output (verbose)

# z is for a gz compressed archive

# f is a file specifier

This will extract all the files to a directory called buildroot.

If you are interested you can also get the latest release of the Buildroot system using git. However this repository is constantly being altered and it does not always contain a stable release. You can use the command below to get a clone of the latest repository:

$ git clone git://git.buildroot.net/buildroot

### Host System Requirements

Buildroot is designed to run on Linux systems.

While Buildroot itself will build most host packages it needs for the compilation, certain standard Linux utilities are expected to be already installed on the host system. Below you will ﬁnd an overview of the mandatory and optional packages (note that package names may vary between distributions).

Luckely Buildroot will generate warnings or errors if certain dependencies are not met.

To install the minimal tools and libraries required to get the system to build you will need to install the following dependencies:

$ sudo apt-get install git git-core libncurses5 libncurses5-dev build-essential

### Configuring your Buildroot System

Luckely Buildroot has minimal configuration files ready for the Raspberry Pi. The *raspberrypi\_defconfig* configuration is a minimal configuration with all that is required to bring the Raspberry Pi. We should base our work on this defconfig:

$ make raspberrypi\_defconfig

All the default configuration files can be found in the *configs* directory.

The default configuration file we will use as a start point is actually copied in the Buildroot top directory to a hidden file called “.config”.

You can now open the ncurses menuconfig to alter the configuration of your Buildroot system. Just execute the following command:

$ make menuconfig

If you select the *Target Options* menu item you should see that the *Target Architecture* is set to ARM.

This configuration allows you to manipulate all sort of options such as

* The standard C library
* Device tree support
* The Linux kernel version
* Busybox
* Type of root file system (rootfs)
* The bootloader (U-boot for example)
* …

One thing to keep in mind is that this configuration is an overall configuration of your Buildroot system. Specific configurations should be altered on a subsystem base. In other words we also have a config for the kernel, busybox, uboot, glibc, … Every aspect of the system can be tailored to your needs.

The configuration of these subsystems will be discussed later.

### Building it all

You will need to have access to the network, since Buildroot will download the packages' sources.

You may now build your system with:

$ make

(This may take a while; consider getting yourself a coffee). If you get an error about the version of aclocal then check out Section 8.1.

If you traverse to the *buildroot/output/images* directory and execute a *tree* command you should get the following output:

nico@NICOUBU:~/first\_br/buildroot/output/images$ tree

.

├── rootfs.tar

├── rpi-firmware

│   ├── bootcode.bin

│   ├── cmdline.txt

│   ├── config.txt

│   ├── fixup.dat

│   └── start.elf

└── zImage

1 directory, 7 files

These files need to be placed on the SD card in a certain way to be able for the bootloaders to find the files and boot the system correctly. See Section 7 for more information on the boot process of the Raspberry Pi.

### Creating a Bootable SD Card

A problem we have to overcome is the fact that we do not have direct (RAW) access to the SD card from our virtual machine. This is not supported by either VMware or VirtualBox. This can be solved by creating an image file of the SD card and using the Windows tool Win32 Disk Imager to write the image to the SD card.

From the mentality that everything is a file in Linux we can solve our problem by creating an image file in Linux, create partitions in it, format it, mount it and write files to it.

The kernel and root filesystem are quit small in size so we will not need a lot of space.

Start by creating an image file:

$ cd ~

$ dd if=/dev/zero of=minimal\_pi.img bs=1M count=150

This will create a file called *minimal\_pi.img* in your home directory with a size of 150MB. This file will be filled with all zeroes (coming from the virtual device /dev/zero).

As noted on <http://git.buildroot.net/buildroot/tree/board/raspberrypi/readme.txt> the SD card needs to be partitioned with two partitions. Our next step is to create these two partitions. The first is a FAT32 partition which should be marked as bootable and will need to contain the kernel image, boot loaders and some configuration files. The second partition will contain the root file system and can be formatted as ext3, ext4, ….

The partitions can be created using the system tool *fdisk*. Just issue the following command to operate on the image as it were a hard drive:

$ fdisk ~/minimal\_pi.img

Use m to get some more information. You are to create the following partition layout:

Disk minimal\_pi.img: 157 MB, 157286400 bytes

255 heads, 63 sectors/track, 19 cylinders, total 307200 sectors

Units = sectors of 1 \* 512 = 512 bytes

Sector size (logical/physical): 512 bytes / 512 bytes

I/O size (minimum/optimal): 512 bytes / 512 bytes

Disk identifier: 0xec0641d8

Device Boot Start End Blocks Id System

minimal\_pi.img1 \* 2048 206847 102400 c W95 FAT32 (LBA)

minimal\_pi.img2 206848 307199 50176 83 Linux

Note that you will need to set the type of the first partition to “W95 FAT32 (LBA)” yourself. Also make sure to make it bootable.

Now we need to be able to format these partitions. This would be quit complex if we did not have a tool such as *kpartx*, which you should install now. Using kpartx we can map a raw image file to a loop device. A loop device is a virtual device which allows a file to be used as a block device (an example of a block device is a hard drive).

By executing the following command we map both partitions to virtual devices using a loop device. The virtual devices can be found under */dev/mapper/*.

$ sudo kpartx –a ~/minimal\_pi.img

Now we can format the partitions using the mkfs[[23]](#footnote-23) system tool. Enter the following command to format the first partition as FAT32 with a label *boot*:

$ sudo mkfs.vfat –F32 –n boot /dev/mapper/loop0p1

Do note that the virtual device may be called different based on the used free loop device.

Next we create an ext4 Linux filesystem on the second partition:

$ sudo mkfs.ext4 –L rootfs /dev/mapper/loop0p2

Now we can access the newly created filesystems by mounting them:

$ sudo mkdir –p /mnt/boot

$ sudo mkdir –p /mnt/rootfs

$ sudo mount –t vfat -o rw,sync /dev/mapper/loop0p1 /mnt/boot

$ sudo mount –t ext4 -o rw,sync /dev/mapper/loop0p2 /mnt/rootfs

Now you can copy the kernel image, the configuration files and the bootloaders to the boot partition:

$ sudo cp ~/buildroot/output/images/zImage /mnt/boot

$ sudo cp ~/buildroot/output/images/rpi-firmware/\* /mnt/boot

We also need to extract the root filesystem to the second partition:

$ sudo tar xf ~/buildroot/output/images/rootfs.tar –C /mnt/rootfs

Once this is finished we can unmount the partitions and remove the virtual device mapping:

$ sudo umount /mnt/boot

$ sudo umount /mnt/rootfs

$ sudo kpartx –d ~/minimal\_pi.img

Now you can copy the image file to your shared directory so you can access it from your windows machine.

**Assignment**

Create a bash-script which automates the steps for creating the filesystems and copying the files to the partitions. Check out <http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html> on how to create a simple bash-script.

**Assignment**

Also add the creation of the partitions to your bash script. Check out <http://xmodulo.com/how-to-run-fdisk-in-non-interactive-batch-mode.html> for some guidelines (use the first method).

### Configuring the Linux Kernel

Todo

$ make linux-menuconfig

### Configuring BusyBox

Todo

$ make busybox-menuconfig

# The Raspberry Pi Boot Process[[24]](#footnote-24)

To reduce cost, the Raspberry Pi (Model A & B) omits any on-board non-volatile memory used to store the boot loaders, Linux kernels and file systems as seen in more traditional embedded systems. Rather, a SD/MMC card slot is provided for this purpose. (The Raspberry PI Compute Module has 4Gbyte eMMC Flash on-board)

The Raspberry Pi’s Broadcom BCM2835 system on a chip (SoC) powers up with its ARM1176JZF-S 700 MHz processor held in reset. The VideoCore IV GPU core is responsible for booting the system. It loads the first stage bootloader from a ROM embedded within the SoC. The first stage bootloader is designed to load the second stage bootloader (bootcode.bin) from a FAT32 or FAT16 filesystem on the SD Card.

The second stage bootloader - bootcode.bin - is executed on the VideoCore GPU and loads the third stage bootloader - start.elf. (Historically, yet another bootloader called loader.bin was loaded at this stage, but has since been phased out).

The third stage bootloader - start.elf - is where all the action happens. It starts by reading config.txt, a text file containing configuration parameters for both the VideoCore (Video/HDMI modes, memory, console frame buffers etc) and loading of the Linux kernel (load addresses, device tree, uart/console baud rates etc).

(Prior to the introduction of the 512MB Raspberry Pi, multiple start.elf files were provided (arm128\_start.elf, arm192\_start.elf, arm224\_start.elf & arm240\_start.elf). These files would partition the memory used between the ARM processor (Linux) and the VideoCore GPU. This has now been superseded with a gpu\_mem parameter in the config.txt file specifying the memory to be allocated to the GPU.)

Once the config.txt file has been loaded and parsed, the third stage bootloader will also loads cmdline.txt and kernel.img. cmdline.txt is a file containing the kernel command line parameters to be passed to the kernel at boot. It will load kernel.img into the shared memory allocated to the ARM processor, and release the ARM processor from reset. Your kernel should now start booting.

The third stage bootloader - start.elf, also passes extra parameters to the kernel than contained within the cmdline.txt. For example, my Linux Kernel receives the following parameters, specifying DMA channels, GPU parameters, MAC addresses, eMMC clock speeds and allowable Kernel Memory size.

dma.dmachans=0x7f35

bcm2708\_fb.fbwidth=1280

bcm2708\_fb.fbheight=1024

bcm2708.boardrev=0xe

bcm2708.serial=0xd9b35572

smsc95xx.macaddr=B8:27:EB:B3:55:72

sdhci-bcm2708.emmc\_clock\_freq=250000000

vc\_mem.mem\_base=0xec00000

vc\_mem.mem\_size=0x10000000

console=ttyAMA0,115200

kgdboc=ttyAMA0,115200

console=tty1

root=/dev/mmcblk0p2

rootfstype=ext4

rootwait

Mainline Linux Kernels may not parse these extra parameters.

# Compiling and Installing Packages on Your Host System

Sometimes you will be required to compile and install packages on your host system[[25]](#footnote-25) yourself. Let’s take *automake 1.14* as an example.

Surf to <http://packages.ubuntu.com/en/source/trusty/automake-1.14> and scroll down to the bottom of the page. As shown in Figure 44 you will see three files. automake-1.14\_1.14.1.orig.tar.xz is the original Debian source. automake-1.14\_1.14.1-2ubuntu1.debian.tar.gz is a patch required when compiling for an Ubuntu system. automake-1.14\_1.14.1-2ubuntu1.dsc is a digital signature file.

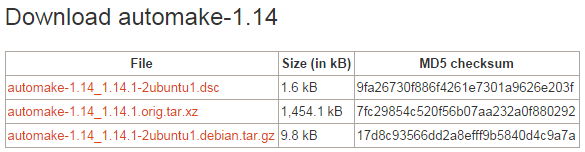


Figure 44 Automake sources

Download both the Debian package and the Ubuntu patch to your homedir[[26]](#footnote-26) using the following commands:

$ cd ~

$ mkdir automakesrc

$ cd automakesrc

$ wget http://archive.ubuntu.com/ubuntu/pool/main/a/automake-1.14/automake-1.14\_1.14.1.orig.tar.xz

$ wget http://archive.ubuntu.com/ubuntu/pool/main/a/automake-1.14/automake-1.14\_1.14.1-2ubuntu1.debian.tar.gz

Unpack both packages:

$ tar -xvf automake-1.14\_1.14.1.orig.tar.xz

$ tar -xzvf automake-1.14\_1.14.1-2ubuntu1.debian.tar.gz

Move the patches inside the original source directory:

$ mv debian automake-1.14.1

To apply the patches we can use the *quilt* tool which allows us to apply and remove patches from a source. If you haven’t installed quilt yet execute the following commands[[27]](#footnote-27):

$ sudo apt-get install quilt

$ echo "export QUILT\_PATCHES=debian/patches" >> ~/.bashrc

# Before working with Quilt you need to tell it where to find the patches.

$ . ~/.bashrc

# Source the file to apply the new export

Now you can apply the Ubuntu patch:

$ cd automake-1.14.1/

$ quilt push

To check which patches have been applied you can always use the command:

$ quilt applied

Next we need to execute the configure script that comes with the source package.

$ ./configure

The configure script checks some details about the machine on which the software is going to be installed. This script checks for lots of dependencies on your system. For the particular software to work properly, it may be requiring a lot of things to be existing on your machine already. If any of the major requirements are missing on your system, the configure script would exit and you cannot proceed with the installation, until you get those required things.

The configure will also generate a Makefile. A Makefile is a special file, containing shell commands, that you create and name Makefile. While in the directory containing this Makefile, you will type make and the commands in the Makefile will be executed.

$ make

The *make* command uses the Makefile to build (compiles) the source code. Compiler compiles the code, but, most of the times, the code cannot stand alone, it requires external libraries (usually provided by Ubuntu packages) to be installed. After this step the executable(s) of this specific application you are trying to install will be created.

Once finished you can execute the make install command which will copy the executables to the appropriate directories:

$ sudo make install

This command moves all the needed files for the application to the appropriate system directories. This has to be done after make because the executables of the application have been created and can be moved to the appropriate system directory (e.g. /usr/local/bin/).

By executing the *which* command following by the name of a tool you see where the binary executable is located:

$ which aclocal

/usr/local/bin/aclocal

By appending the ‘--version’ option to aclocal we can check the version:

$ aclocal --version

aclocal (GNU automake) 1.14.1

Copyright (C) 2013 Free Software Foundation, Inc.

License GPLv2+: GNU GPL version 2 or later <http://gnu.org/licenses/gpl-2.0.html>

This is free software: you are free to change and redistribute it.

There is NO WARRANTY, to the extent permitted by law.

Written by Tom Tromey <tromey@redhat.com>

and Alexandre Duret-Lutz <adl@gnu.org>.

# Solution Center

## Buildroot ac-local 1.14 is missing[[28]](#footnote-28)

If you get the following error while making the buildroot:

WARNING: 'aclocal-1.14' is missing on your system.  
         You should only need it if you modified 'acinclude.m4' or  
         'configure.ac' or m4 files included by 'configure.ac'.  
         The 'aclocal' program is part of the GNU Automake package:  
         <<http://www.gnu.org/software/automake>>  
         It also requires GNU Autoconf, GNU m4 and Perl in order to run:  
         <<http://www.gnu.org/software/autoconf>>  
         <<http://www.gnu.org/software/m4/>>  
         <<http://www.perl.org/>>  
make[1]: \*\*\* [aclocal.m4] Error 127

You can fix this by surfing to <http://packages.ubuntu.com/en/source/trusty/automake-1.14> and navigating to the binary package automake url indicated in Figure 44 at the top of the page.

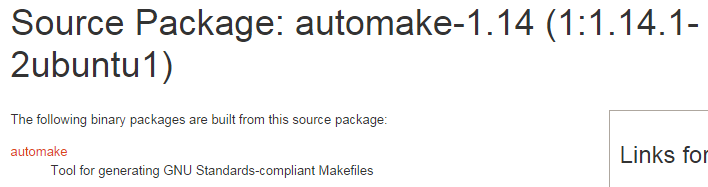


Figure 45 Binary automake package

Scroll down the page and select the download link at the bottom of the page. Select a mirror to download the Debian package and install it.

Running the following command should indicate that the aclocal tool is now upgraded to version 1.14:

$ aclocal --version

# Important Commands Overview

|  |  |
| --- | --- |
| Command | Description |
| ifconfig | Configure a network interface |
| dmesg | Print or control the kernel ring buffer |
| df | Disk free |
| ls | List files |
| cp | Copy files |
| rm | Remove files |
| mv | Move files |
| cd | Change working dir |
| chmod | Change file permission mode |
| chown | Change owner of file |
| cat | Concatenate files and output to terminal |
| touch | Create an empty file |
| mkdir | Make directory |
| tree | List contents of directories in a tree-like format |
| dd | Convert or copy a file |
| which | Locate a command |
| file | Determine file type |

# References

|  |  |
| --- | --- |
| Description | URL |
| Configuration options of the Pi | <http://www.raspberrypi.org/documentation/configuration/config-txt.md> |
| How To Understand the Filesystem Layout in a Linux VPS | <https://www.digitalocean.com/community/tutorials/how-to-understand-the-filesystem-layout-in-a-linux-vps> |
| Understanding the Raspberry Pi Boot Process | <http://wiki.beyondlogic.org/index.php?title=Understanding_RaspberryPi_Boot_Process> |
| Compiling things on Ubuntu the Easy Way | <https://help.ubuntu.com/community/CompilingEasyHowTo> |
| Patches to Packages | <http://packaging.ubuntu.com/html/patches-to-packages.html> |

# mBed I2C Memory Slave

The I2C Memory Slave emulates a really small I2C memory device. It allows you to store 255 integers (of 4 bytes) each and retrieve them.

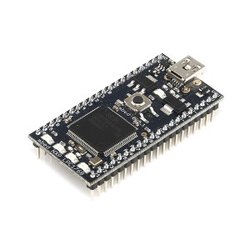
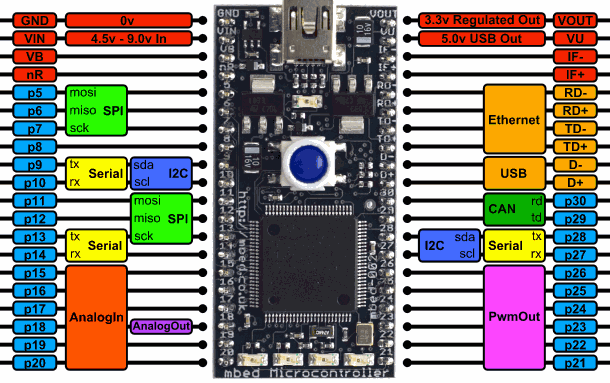
 

Figure 46 The mbed platform (left) and its pinout (right)

To connect to the device you will need to attach an I2C master device to pin 27 (SCL) and pin 28 (SDA). You will also need to add two pull-up resisters of 2k2. Figure 46 shows the pinout of the mbed.

To get some console output from the slave device you can use a terminal program such as Putty to connect to the serial interface over USB at a speed of 115200 baud.

The slave device also has an alive LED which will blink periodically as long as the device is operational and responsive.

The I2C bus operates at 100kHz and the slave device address is 0x84.

Table 3 shows an overview of the commands that can be send to slave device.

Table 3 mbed I2C memory slave commands

|  |  |  |  |
| --- | --- | --- | --- |
| Command value | Command name | Description | Example |
| 0x00 | PUSH | Write a value to the specified address | Write: [0x00] [1 byte address] [4 bytes data] |
| 0x01 | PULL | Set internal read pointer to specific address | Write: [0x01] [1 byte address] |
| 0x02 | CLEAR | Reset the memory to all zeroes | Write: [0x02] |
| 0x03 | PRINT | Print memory to slave console | Write: [0x03] |

With the PUSH command you can write a value to the memory device. All you need to do is send 6 bytes to the device. The command byte, followed by the address of where the value should be stored and last the actual integer value consisting of 4 bytes.

The PULL command allows you to set the internal read pointer of the slave. This pointer is used when reading from the device and should point to the memory location you want to read. Once set you can do a read from the slave device, which will send you 5 bytes. The first byte is the address of the internal read pointer. The following 4 bytes are the actual value stored in the memory at the given address.

The CLEAR and PRINT commands are simple 1 byte commands which can be send to the device to respectively clear the memory and to print the content of the memory to the slave’s console.

## I2C Slave Code

### Memory.h

#ifndef MEMORY\_HEADER

#define MEMORY\_HEADER

class Memory {

public:

const static int MEMORY\_SIZE = 32;

private:

int memory[MEMORY\_SIZE];

public:

Memory();

void reset();

void set(int address, int value);

int get(int address);

void print();

};

#endif

### Memory.cpp

#include "mbed.h"

#include "memory.h"

Memory::Memory()

{

reset();

}

/\*

\* Reset all memory locations to 0.

\*/

void Memory::reset()

{

for (int i = 0; i < Memory::MEMORY\_SIZE; i++) {

this->memory[i] = 0;

}

}

/\*

\* Store value in memory

\*/

void Memory::set(int address, int value)

{

if (address < Memory::MEMORY\_SIZE) {

this->memory[address] = value;

}

}

/\*

\* Retrieve value from memory

\*/

int Memory::get(int address)

{

if (address < Memory::MEMORY\_SIZE) {

return this->memory[address];

} else {

return 0;

}

}

/\*

\* Print current memory content to console

\*/

void Memory::print()

{

int i = 0, c = 0;

while (i < Memory::MEMORY\_SIZE) {

c = (c + 1) % 4;

printf("\t[%#04x]: %6d", i, this->memory[i]);

if (!c) {

printf("\r\n");

}

i++;

}

}

### main.c

#include "mbed.h"

#include "memory.h"

DigitalOut aLED(LED1);

Serial pc(USBTX, USBRX); // tx, rx

I2CSlave slave(p28, p27);

const int SLAVE\_ADDRESS = 0x84;

const int I2C\_FREQUENCY = 100000;

const int I2C\_BUFFER\_SIZE = 6;

enum COMMAND { PUSH, PULL, CLEAR, PRINT };

#define intToByte(pbytebuff,intval) (\*((int\*)(pbytebuff)) = intval)

#define byteToInt(pbytebuff,pintval) (\*(pintval) = \*((int\*)(pbytebuff)))

int main() {

pc.baud(115200);

pc.printf("Size of integer is %d bytes\r\n", sizeof(int));

// Alive LED

int cAlive = 0;

// Configure I2C

slave.frequency(I2C\_FREQUENCY);

pc.printf("Slave is working @ %dHz\r\n", I2C\_FREQUENCY);

slave.address(SLAVE\_ADDRESS);

pc.printf("Slave is working @ SLAVE\_ADDRESS = 0x%x\r\n", SLAVE\_ADDRESS);

// Setup memory

Memory memory;

pc.printf("Size of memory buffer is %d elements\r\n", Memory::MEMORY\_SIZE);

memory.print();

pc.printf("Awaiting commands from master ...\r\n");

// I2C buffer

char buffer[I2C\_BUFFER\_SIZE];

// Internal address pointer

int pointer = 0;

while (1) {

int rec = slave.receive();

switch (rec) {

case I2CSlave::ReadAddressed:

{

int value = memory.get(pointer);

buffer[0] = pointer;

intToByte(buffer+1, value);

if (!slave.write(buffer, 5)) {

pc.printf("Retrieving and sending to master %d@%d\r\n", value, pointer);

} else {

pc.printf("Failed to send to master %d@%d\r\n", value, pointer);

}

break;

}

case I2CSlave::WriteAddressed:

{

// First we read the command byte

int command = slave.read();

// Check the command

switch (command)

{

case PUSH:

// Expect 5 more bytes [address] [int value]

if(!slave.read(buffer, 5)) {

int address = buffer[0];

int value;

byteToInt(buffer+1, &value);

pc.printf("Storing %d@%d\r\n", value, address);

memory.set(address, value);

} else {

pc.printf("PUSH received with missing address/data\r\n");

}

break;

case PULL:

// Expect 1 more byte [address]

if(!slave.read(buffer, 1)) {

int address = buffer[0];

pc.printf("Setting pointer to %d\r\n", address);

if (address < Memory::MEMORY\_SIZE) {

pointer = address;

} else {

pc.printf("Address out of boundary\r\n");

}

} else {

pc.printf("PULL received with missing address\r\n");

}

break;

case CLEAR:

pc.printf("Clearing the memory\r\n");

memory.reset();

break;

case PRINT:

memory.print();

break;

default:

pc.printf("Unknown command byte\r\n");

}

}

}

// Clear buffer

for (int i = 0; i < I2C\_BUFFER\_SIZE; i++) {

buffer[i] = 0;

}

// Alive LED

cAlive = (cAlive + 1) % 100000;

if (!cAlive) {

aLED = !aLED;

}

}

}

### Some Notes

Important to know is that the ’int read(char \* data, int length)’ function of the mbed I2C and I2CSlave libraries will return 0 when the buffer contains the number of bytes given as the *length* argument. If the buffer is not completely filled then the function will return non-zero. This means you always need to check if you do a read if you received the correct number of bytes.

1. The BCM2835 is a High Definition 1080p Embedded Multimedia Applications Processor [↑](#footnote-ref-1)
2. SoC, or System on a Chip, is a method of placing all necessary electronics for running a computer on a single chip. Instead of having an individual chip for the CPU, GPU, USB controller, RAM, Northbridge, Southbridge, etc., everything is compressed down into one tidy package. [↑](#footnote-ref-2)
3. A RISC-based computer design approach means ARM processors require significantly fewer transistors than typical CISC x86 processors in most personal computers. This approach reduces costs, heat and power use. These are desirable traits for light, portable, battery-powered devices—​including smartphones, laptops, tablet and notepad computers, and other embedded systems. A simpler design facilitates more efficient multi-core CPUs and higher core counts at lower cost, providing improved energy efficiency for servers. [↑](#footnote-ref-3)
4. Check out <http://www.raspberrypi.org/documentation/installation/installing-images/README.md> for instructions for different host operating systems such as Linux or Mac. [↑](#footnote-ref-4)
5. SSH or Secure Shell is a secure way to connect to a device and execute commands from a distance. Default SSH daemon listen on port 22. See chapter xxxx for more information on SSH. [↑](#footnote-ref-5)
6. <http://www.softperfect.com/products/networkscanner/> [↑](#footnote-ref-6)
7. Wireshark, originally named Ethereal, is a free and open-source packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development, and education. It can be downloaded from <https://www.wireshark.org/> [↑](#footnote-ref-7)
8. PuTTY is a free implementation of Telnet and SSH for Windows and Unix platforms, along with an xterm terminal emulator. It can be downloaded from <http://www.chiark.greenend.org.uk/~sgtatham/putty/> [↑](#footnote-ref-8)
9. The datasheet can be downloaded from <https://www.adafruit.com/datasheets/PL2303HX.pdf> [↑](#footnote-ref-9)
10. You can find the COM port number using the device manager. Select the “Ports (COM & LPT)” category and look for a “USB-to-Serial Comm Port (COMx)” device. [↑](#footnote-ref-10)
11. <http://en.wikipedia.org/wiki/HAL_9000> [↑](#footnote-ref-11)
12. In Linux hidden files start with a dot, for example “.ssh”. [↑](#footnote-ref-12)
13. [https://www.digitalocean.com/community/tutorials/how-to-understand-the-filesystem-layout-in-a-linux-vps] [↑](#footnote-ref-13)
14. Check http://www.pathname.com/fhs/ [↑](#footnote-ref-14)
15. Stands for “Editable Text Configuration” [↑](#footnote-ref-15)
16. Stands for “Unix System Resources” [↑](#footnote-ref-16)
17. See “man tasksel” [↑](#footnote-ref-17)
18. Hyper-V, codenamed Viridian and formerly known as Windows Server Virtualization, is a native hypervisor; it can create virtual machines on x86-64 systems. Starting with Windows 8, Hyper-V supersedes Windows Virtual PC as the hardware virtualization component of the client editions of Windows NT. [↑](#footnote-ref-18)
19. Long-term support (LTS) is a term used to describe special versions or editions of software designed to be supported for a longer than normal period. It is particularly applicable to open-source software projects. [↑](#footnote-ref-19)
20. Board Support Package [↑](#footnote-ref-20)
21. Software Development Kit [↑](#footnote-ref-21)
22. The Binutils package contains a linker, an assembler, and other tools for handling object files. [↑](#footnote-ref-22)
23. mkfs allows us to create filesystems on a disk. [↑](#footnote-ref-23)
24. Source: <http://wiki.beyondlogic.org/index.php?title=Understanding_RaspberryPi_Boot_Process> [↑](#footnote-ref-24)
25. In our case this is our Ubuntu virtual machine [↑](#footnote-ref-25)
26. You can also use the directory /usr/local/src for this. [↑](#footnote-ref-26)
27. <http://packaging.ubuntu.com/html/patches-to-packages.html> [↑](#footnote-ref-27)
28. <http://askubuntu.com/questions/484237/how-do-i-get-the-latest-automake> [↑](#footnote-ref-28)