Riot-Os Setup Guide For Windows And Ubuntu

Ville Hiltunen

27.05.2018

Table of Contents

1. Introduction

1.1 Purpose of this document

This document will help you through the process of setting up RIOT for your computer, and how to compile your first test project. The process is detailed for both Windows and Linux users, specifically for Windows 10 and Ubuntu. Users of different operating systems might need to adjust parts of the guide.

1.2 Order of operation

We will first guide the reader through downloading the proper utilities. Then, we will show how to perform the Ubuntu virtualization, after which we will prepare the virtualized Ubuntu for RIOT. Finally, we will compile and upload a simple LED test onto a board.

The last part of the guide will contain general hints for programming with RIOT and some useful links to read.

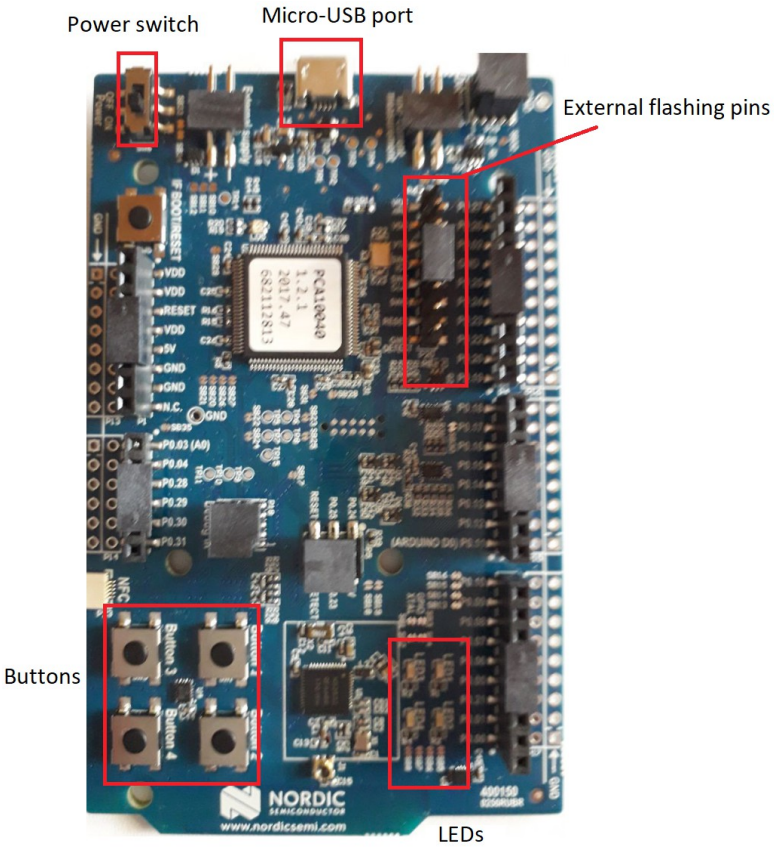
1.3 About RIOT

RIOT-OS is an operating system aimed at IoT devices. It accepts C/C++ and uses GCC to compile. and supports many different boards and CPUs. More information at their website.

1.4 Assumptions

It is assumed that the target build platform is nRF52 DK. The user also needs a micro USB cable to flash the device.

1.5 About nRF52 DK



NRF52 DK is one of Nordic Semiconductor's single board developement kits. It supports Bluetooth low energy, ANT and various other technologies. More information can be found on their website and their infocenter.

In the above picture, the most relevant features for getting started are shown with red rectangles. In this guide, the external flashing is not used, but it is a useful feature to understand. Flashing means the act of uploading program code to the flash memory of a microcontroller, which will then execute the code upon powering on. The DK can be used to flash another board by connecting the external flashing pins to the target board. An example can be seen here.

2. Setting up a virtual Ubuntu on Windows

There are two common ways of using RIOT on Windows. The first way is by utilizing a Cygwin terminal emulator, as is detailed on RIOT github. The guide was verified on 11.05.2018, but it is still quite old and might deprecate eventually. If you choose to use this method, note that flashing the target board might need additional utilities, which are not covered in the guide. In the case of nRF52 DK, it can be done manually by copy pasting the compiled .bin file to the DK.

The second way to use RIOT is to run it on a virtualized Ubuntu. This requires some setup, but this is the more versatile and future-proof choice. Virtualization also normalizes the experience between windows and Ubuntu, so this is the route we choose in this guide. If you already have a working Ubuntu platform, skip to the part 3 of this guide.

2.1 Getting the required utilities

Virtual box is used to simulate Ubuntu. PuTTY is used to connect to the virtualized Ubuntu, and it also doubles as a serial reader when needed. Finally WinSCP is used to make file transfer easier between the host computer and the virtual Ubuntu.

2.1.1 Virtual box

Virtual box is an open source virtualization program, and it can be downloaded from the website here. Follow the installation instructions once the download is complete.

2.1.2 PuTTY

Download PuTTY here. It is probably already familiar to you, but it is a SSH client commonly used to access remote servers. Once again, follow the installation instructions.

2.1.3 WinSCP

WinSCP is a file transfer program that is very good at synchronizing filesystems between Windows and Unix platforms. Grab it here, and install it.

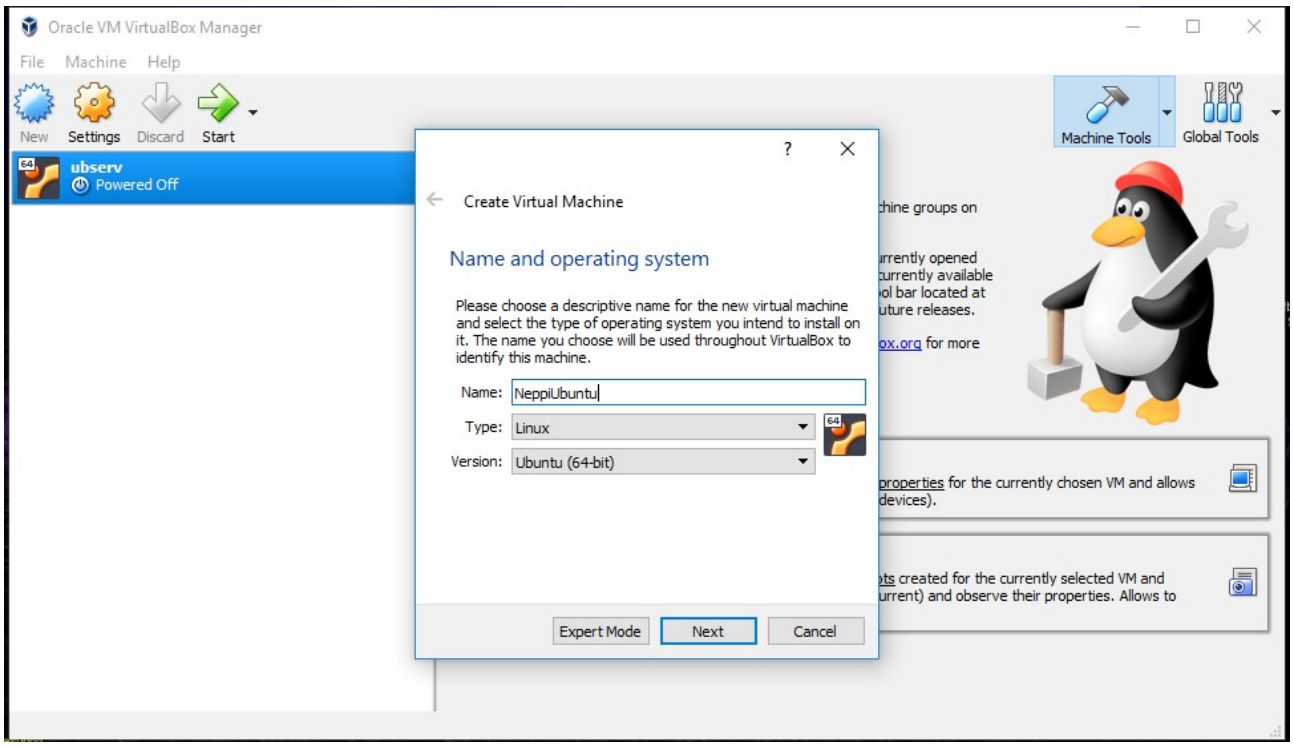
2.1.4 Ubuntu

Finally, Ubuntu is the operating system we need to virtualize. Since we don't need the (heavy) graphical user interface, we choose to use the Ubuntu server version. The latest ISO can be downloaded here.

2.2 Setting up the virtual environment

2.2.1 Creating a new virtual machine

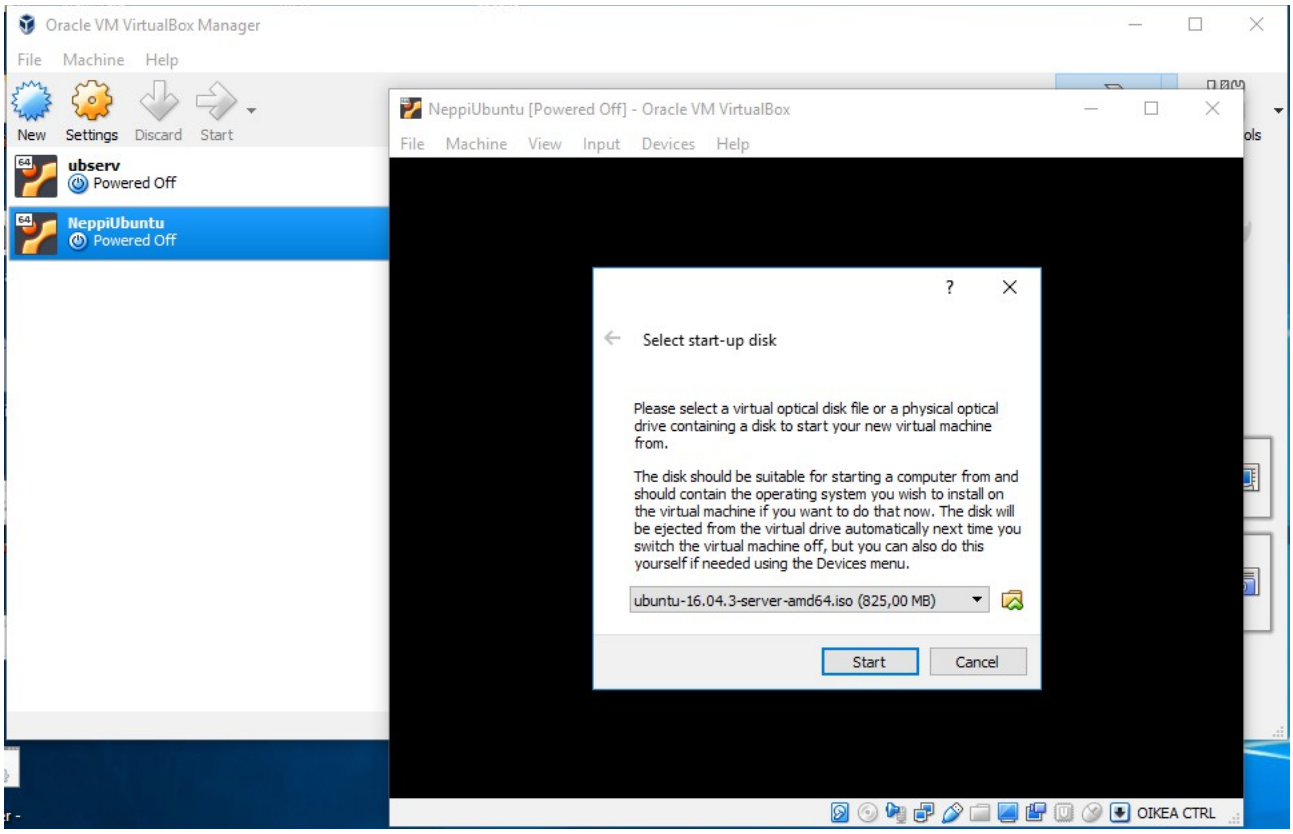
Open VirtualBox, and click on New. Choose Linux and Ubuntu from the drop down menu. The dialog will guide you through the process. The default settings have been verified to work.



NOTE: If your operating system is 64-bit, but the dropdown menu only shows 32-bit versions, virtualization needs to be enabled on your computer. This has to be done from the BIOS menu, and is usually labeled "virtualization technology" or "core virtualization." Find out what the computer's or laptop's BIOS key is and restart the computer to access BIOS. Enable virtualization. If only 32-bit is supported, the guide should still work, but this has not been tested.

2.2.2 Mounting the ISO on the machine

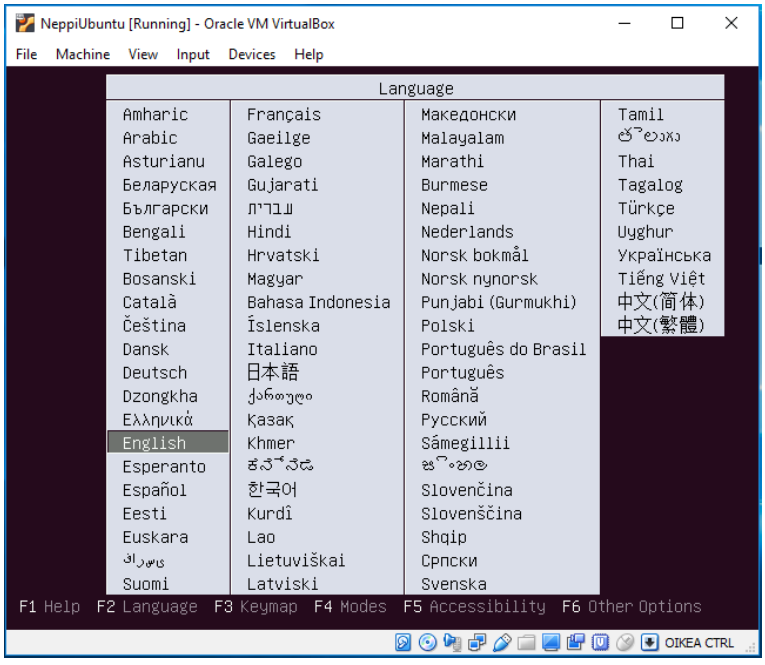
Select the newly created machine, and click start. The machine will now need the Ubuntu ISO that was downloaded earlier. Find it, and press start.

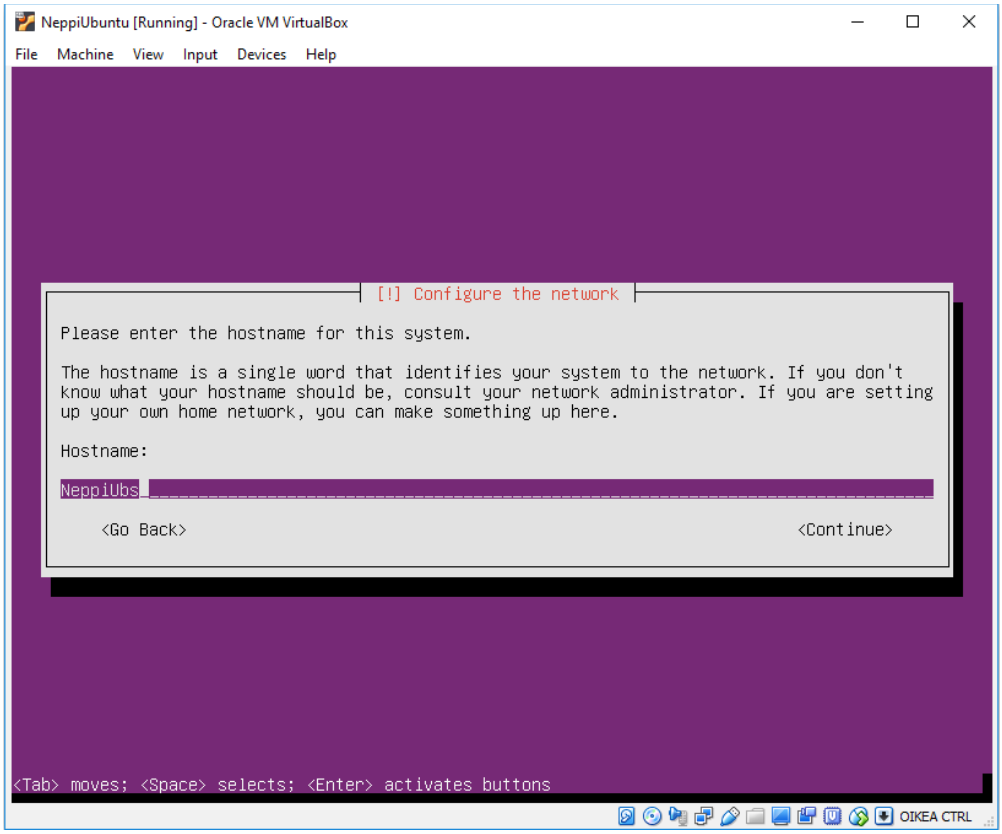


Ubuntu should now begin it's installation process.

2.2.3 Installing Ubuntu

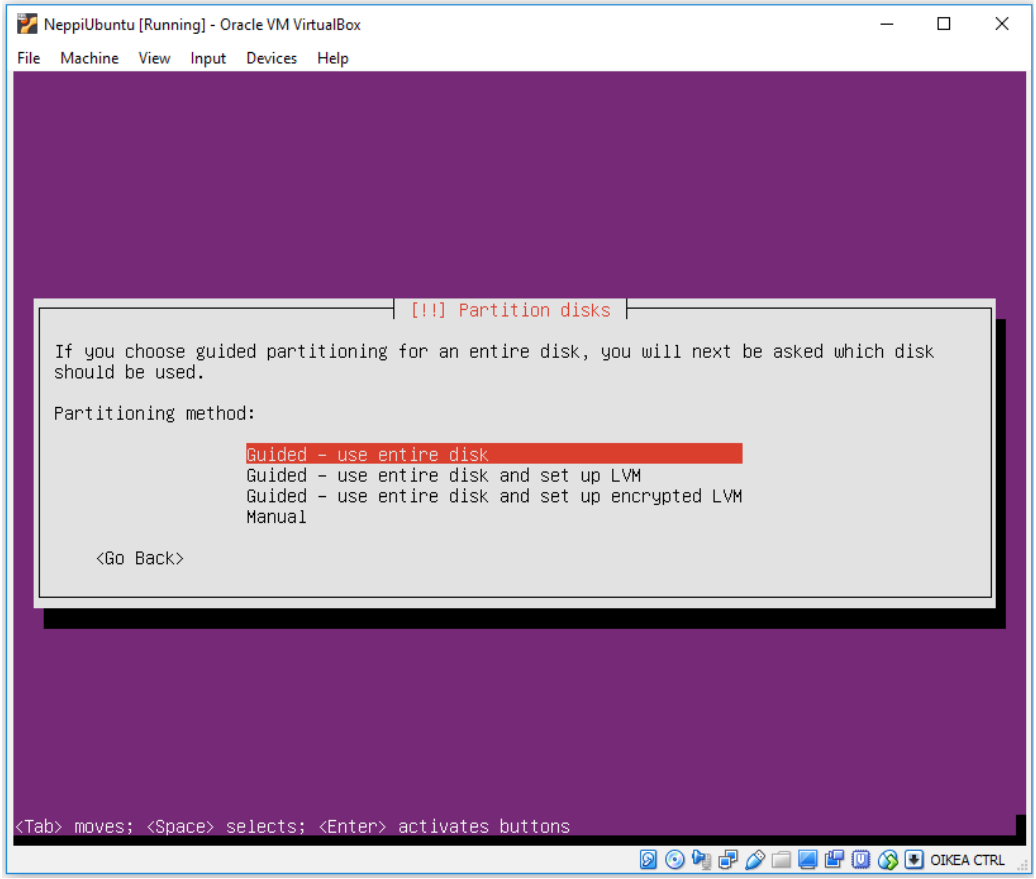
Installation of Ubuntu is simple. Follow the dialogue to select language and keyboard layout.





When the installation asks for a host name, give it something short and readable.

Similarly, a username and password are requested. Again, try to keep the password simple and easy to remember. If the password is forgotten, there are no recovery options.



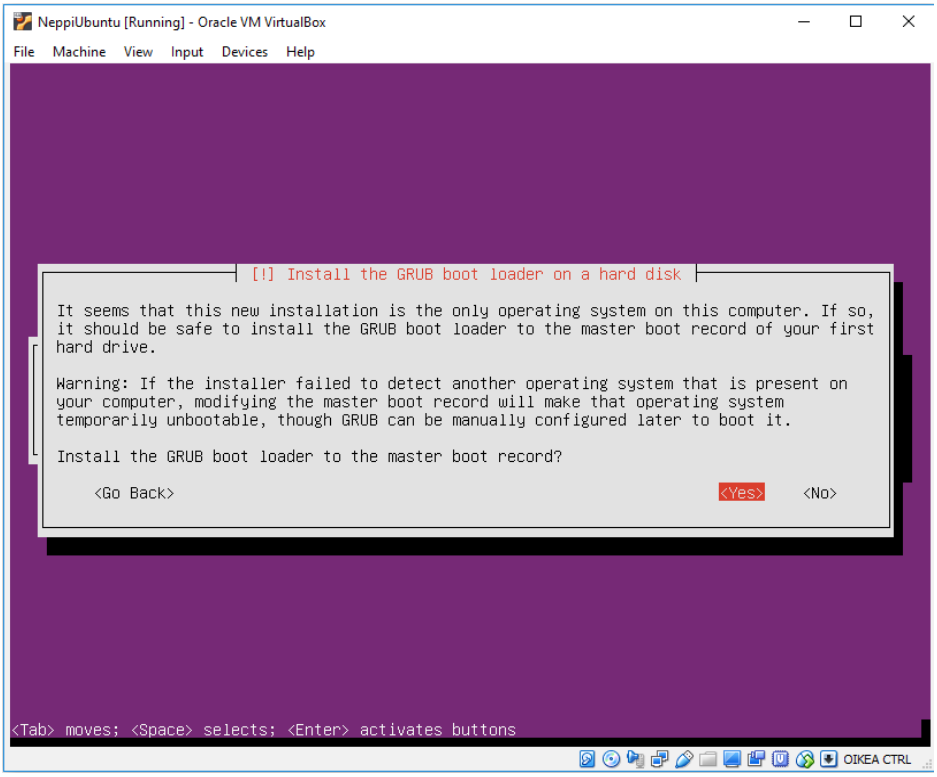
Since we are using a virtual hardrive of around 10 GB, partitioning is not necessary. When the dialogue asks for it, choose guided partitioning, and then to use the entire disk.

Accept the partitioning, and then move on. Next the dialogue should ask for a http proxy. Unless you know you need one, leave it blank.

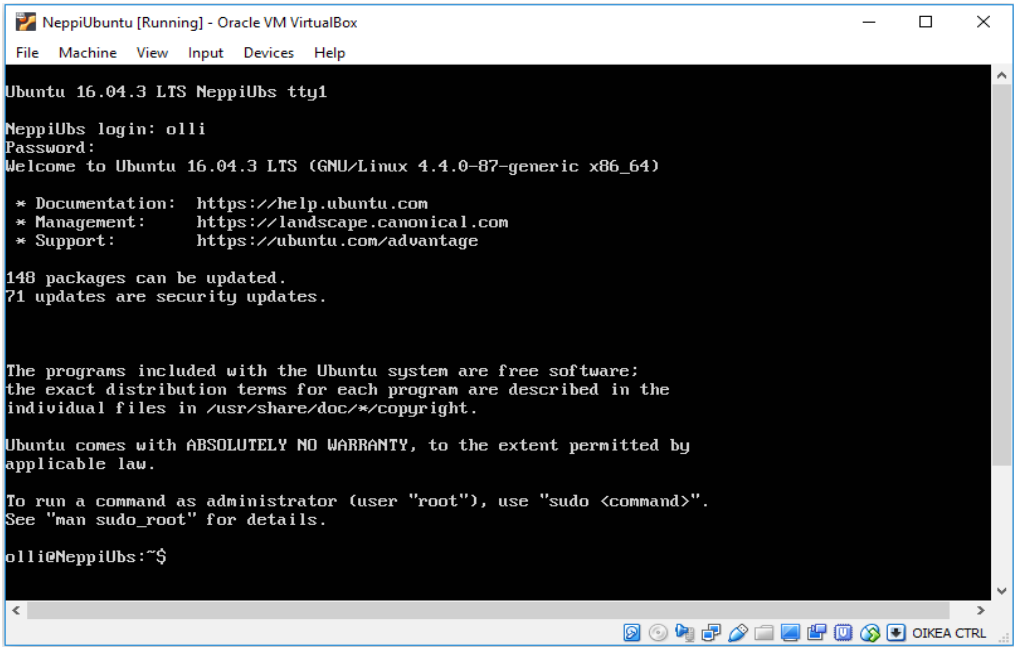
When prompted on how to update Ubuntu, choose manual updates. For the scope of the course a single update at the start should be enough.

Next, we have the option of preinstalling packages. We will install the ones needed manually, so select the first option.

Accept the GRUB boot loader, though the warning is scary.



Accept the final prompts, and the installation should be complete. VirtualBox should unmount the ISO automatically. The virtual machine restarts, and will after a brief boot sequence ask for login information. Log in with the username and password used during installation, and we are ready for the next phase.



2.3 Configuring the virtual Ubuntu

2.3.1 Updating

First, we need to make sure our Ubuntu is up to date. VirtualBox should by default enable your machine to access the internet.

Input the following commands one after another. You will be prompted for your password.

```
sudo apt update
```

```
sudo apt upgrade
```

Update fetches the newest patches, while upgrade actually implements them.

Sudo in this instance runs the command with root permissions (and asks for the password) and apt is the tool we use to update parts of the Ubuntu. If a recent version of Ubuntu was used, not much should be updated.

```
olli@NeppiUbs:~$ sudo apt update
Hit:1 http://fi.archive.ubuntu.com/ubuntu xenial InRelease
Get:2 http://fi.archive.ubuntu.com/ubuntu xenial-updates InRelease [109 kB]
Get:3 http://security.ubuntu.com/ubuntu xenial-security InRelease [107 kB]
Get:4 http://fi.archive.ubuntu.com/ubuntu xenial-backports InRelease [107 kB]
Fetched 323 kB in 0s (424 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
All packages are up to date.
olli@NeppiUbs:~$ sudo apt upgrade
Reading package lists... Done
Building dependency tree
Reading state information... Done
Calculating upgrade... Done
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
olli@NeppiUbs:~$
```

Now we need to get some development tools. Input:

```
sudo apt install build-essential
```

This grants us many important tools such as GCC, which is the compiler we will use.

2.3.2 Initiating SSH connection to our remote.

Now let us connect to our virtual machine through ssh. This might seem pointless at first, but it brings us some useful additions. Firstly, copy pasting seems to be buggy in VirtualBox Ubuntu for many computers, which we can fix by using a terminal program such as PuTTY. Additionally, file transfer can be a bit of a pain, which WinSCP will fix for us.

Adding ssh server capabilities is simple, just type in the command

```
sudo apt install openssh-server
```

This sets everything up for us, now we just need to connect to our machine.

Type in the command

```
ifconfig
```

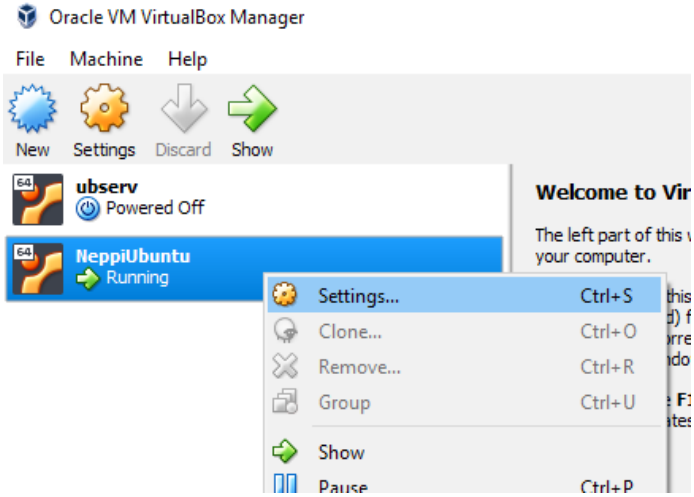
The output should look something like this. The relevant information we are after is within the red box:

```
olli@NeppiUbs:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:c9:5a:93
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fec9:5a93/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:295276 errors:0 dropped:0 overruns:0 frame:0
          TX packets:136539 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:247356948 (247.3 MB)  TX bytes:8231976 (8.2 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:160 errors:0 dropped:0 overruns:0 frame:0
          TX packets:160 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:11840 (11.8 KB)  TX bytes:11840 (11.8 KB)

olli@NeppiUbs:~$
```

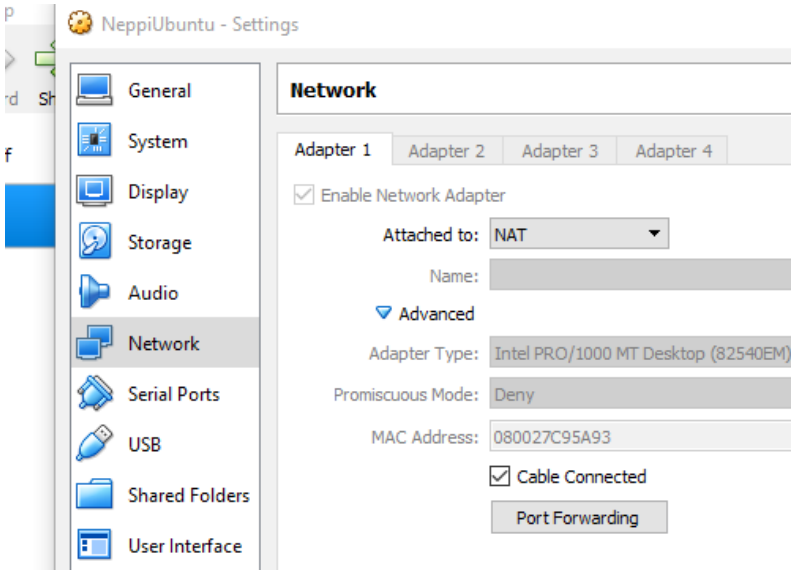
This is the ip address from the perspective of our virtual Ubuntu. This is also the address we want to connect to. In this case it is '10.0.2.15', but this might not be so for everyone.

To enable connection to our machine, VirtualBox can add a port forwarding rule for this address.

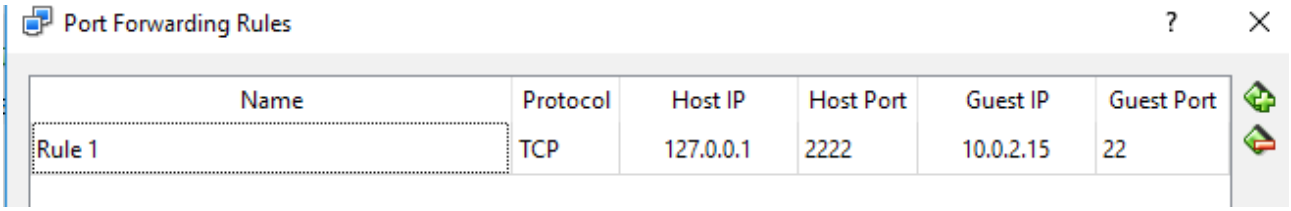
Open the machine settings in VirtualBox:



Under Network, click on the Advanced arrow:

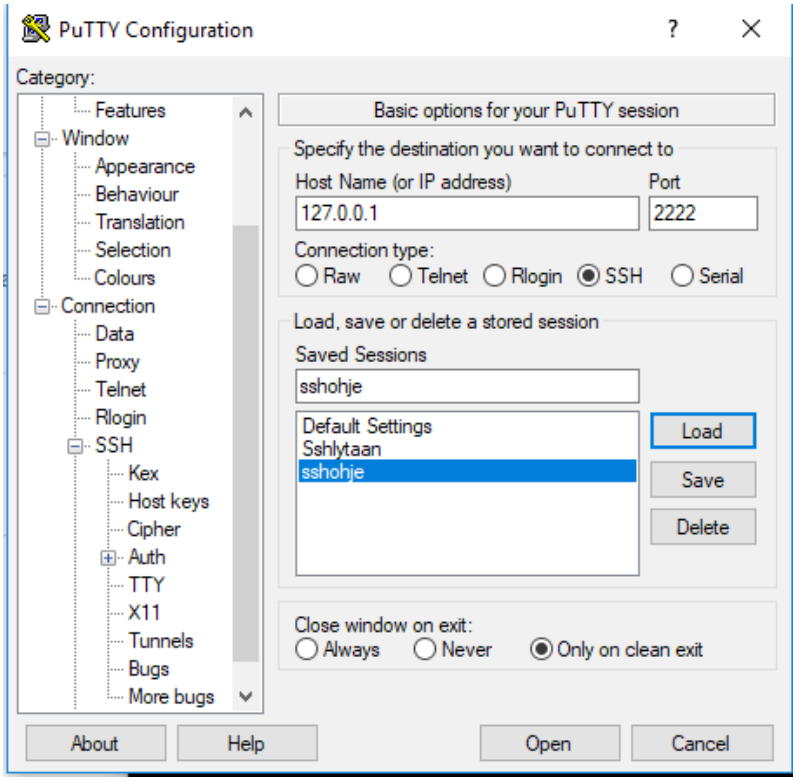


Open Port Forwarding, and add the following rule:



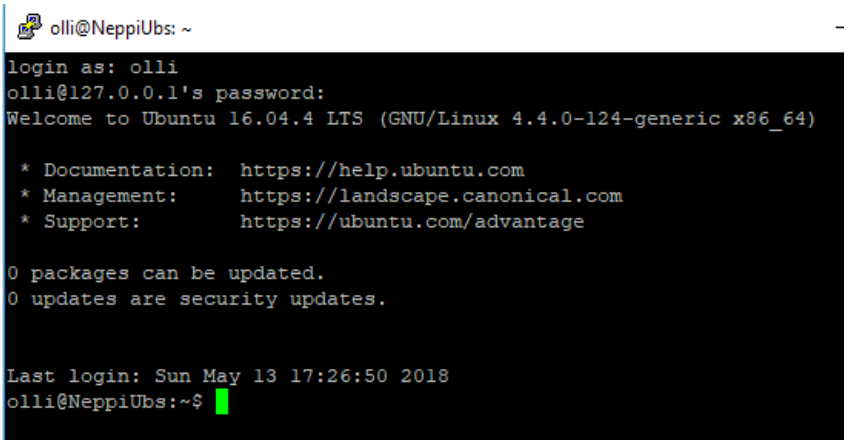
This means that if we are sending packets to our loopback address (127.0.0.1) to port 2222, they are redirected to our virtual machine's address (10.0.2.15 in our case) port 22, which is the default ssh port.

Open up PuTTY. Enter the loopback address as the target address, and port to 2222. Give this session a name, and click save. This saves time in the future.



Click open. If the process was succesful, PuTTY should give you a warning about connecting to an unknown server. Accept the prompt, and you should now have SSH access.

Now, as long as the virtual machine is running, it can be operated through PuTTY just as it could be in the VirtualBox screen.



Ubuntu should now be fully operational. Next up is RIOT setup.

3. Setting up RIOT in Ubuntu

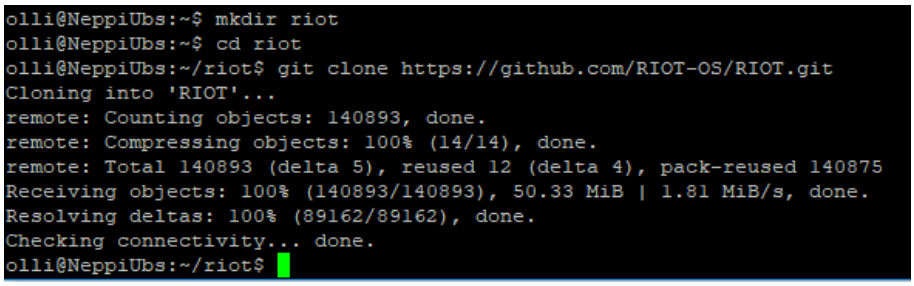
3.1 Initial setup and installing dependencies

First, we need to download RIOT. RIOT is freely available on github, which makes things easy.

Create a directory for RIOT, and then download it by inputting the following commands:

```
mkdir riot

cd riot

git clone https://github.com/RIOT-OS/RIOT.git
```

Mkdir makes a new directory, while cd makes it the current directory. Git clone copies a repository from the internet. HINT: Pasting text into a PuTTY terminal works by right clicking the screen.



RIOT would be usable as such, but we lack couple important tools to compile code for our DK. We need a way to translate our code to ARM cpu code, and a software to flash it.

Get back to your home directory by entering either "cd .." or "cd ~", and start the next phase:

```
cd ..
```

3.1.1 GNU ARM toolchain

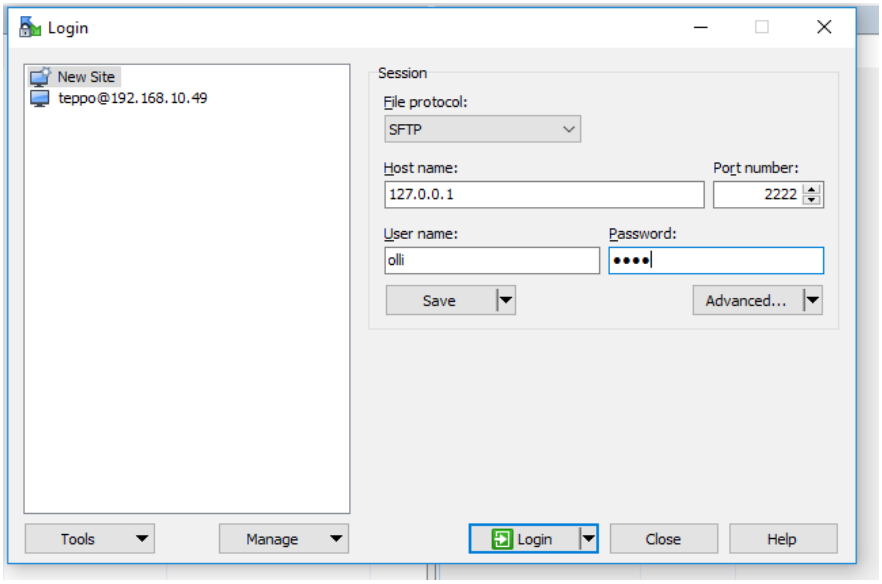
Grab the Linux 64-bit package from here.

While the package is downloading, make a directory for it

```
mkdir gnuarm
```

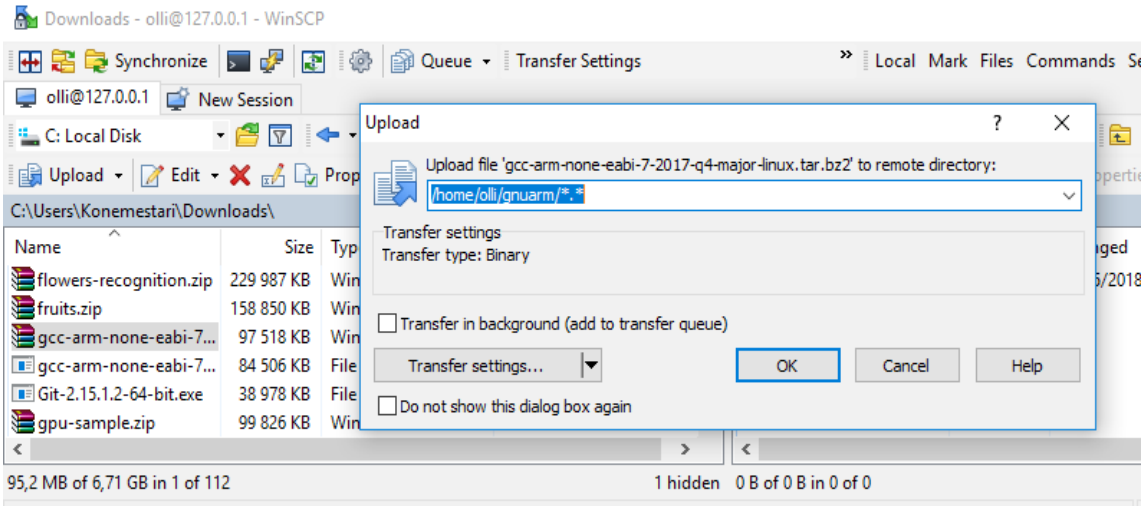
and transfer the package there. For Ubuntu users this is easy, but for Windows users we now need to utlize WinSCP.

Open WinSCP, and enter our target IP and port similarily to PuTTY. Add also your Ubuntu username and password.



Press login, and you should receive a similar warning as PuTTY did. Accept connection.

WinSCP lets the user browse files on both the home and the target computer at the same time and freely transfer them back and forth. Find the empty gnuarm folder on the ubuntu side, and find the package that was just downloaded. Click Upload to send it over.



Now back on the Ubuntu, enter the "gnuarm" directory like before:

```
cd gnuarm
```

We now need a tool to unzip the package. Install one by entering:

```
sudo apt install unzip
```

Unzip the package by calling the following. The filename might change in the future:

```
tar xf gcc-arm-none-eabi-7-2017-q4-major-linux.tar.bz2
```

Hint: Certain commands can be autocompleted by pressing tab. After typing in "tar xf g", by pressing tab Ubuntu should autocomplete the rest.

Back out of the directory to our home again:

```
cd ..
```

3.1.2 Segger JLink

This is the program RIOT uses to communicate with our DK. Download it from here.

Agree to the license agreement. Sadly this is third party software.

Make a directory for the file:

```
mkdir segger
```

Transfer the downloaded file to this directory just like before, and install it with the command:

```
cd segger
```

```
sudo dpkg -i JLink_Linux_V632c_x86_64.deb
```

We are now close to done.

3.1.3 Permissions and cleanup

RIOT requires the gcc-multilib package. It can be downloaded with the command:

```
sudo apt install gcc-multilib
```

RIOT also needs to know where our ARM toolchain exists. The location needs to be added to our path, and can be done as follows:

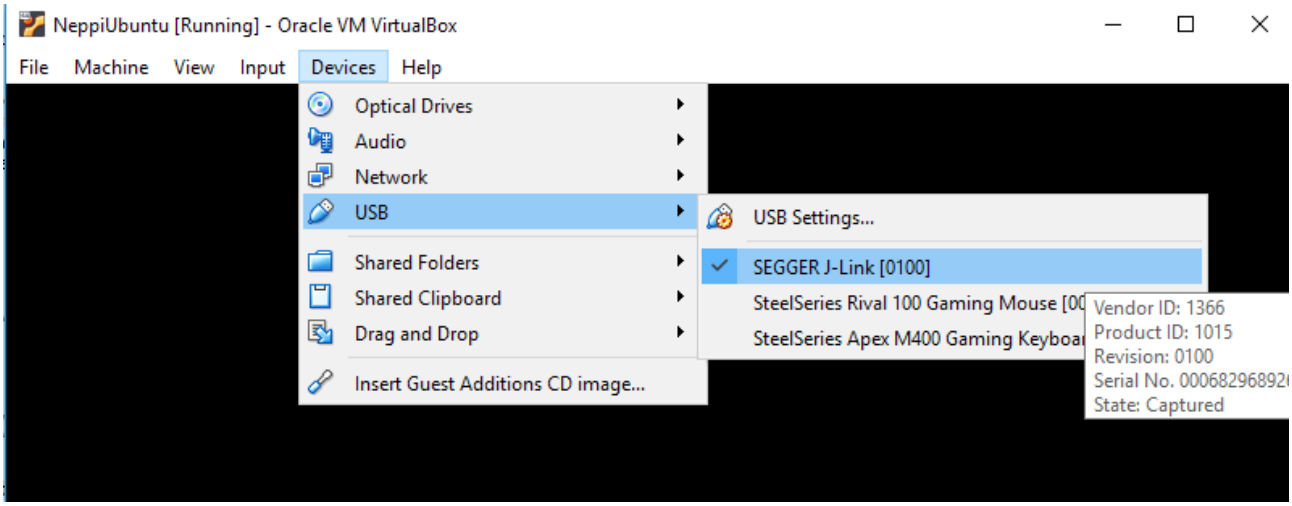
```
export PATH=${PATH}:~/gnuarm/gcc-arm-none-eabi-7-2017-q4-major/bin
```

Note: This command only adds the path for the remainder of the session and must be reinputted after every reboot of the machine. It is not recommended to permanently add the toolchain to the path due to compiler conflicts. However, for the scope of this course this a conflict is unlikely, so the lazy can find whatever way to automate this command.

Now, time to plug in our device. First get the list of devices currently plugged:

```
ls /dev > 1
```

After connecting the DK to an USB port with micro USB, open the virtual machine Device settings, and tick the JLink field.



This links the device to our virtual machine instead of the host computer.

Call the list of devices again, and list the differences:

```
ls /dev > 2
```

```
diff 1 2
```

Our DK is connected to the USB port that appeared on the list. Usually this is ttyACM0.

Because we will be reading this port, we need to give it access (replace the 0 with whatever you have if the port is different):

```
sudo chmod 666 /dev/ttyACM0
```

To read said port, RIOT uses pyterm. For this, we need python pyserial:

```
sudo apt install python-pip
```

```
pip install pyserial
```

All the pieces should be in place!

4. Making the LEDs blink and more

4.1 LED test

For the first example head over to the RIOT tests folder. This is where the unit tests for drivers and modules are kept. In particular we want to test the LEDs on our board.

```
cd ~/riot/RIOT/tests/leds/
```

```
ls
```

The directory should contain 3 files. The main.c code file, the makefile for the gnu make and a readme. This is the basic structure of a RIOT project. The default text editor on server Ubuntu is nano, which can be used to open files by calling "nano" and then the filename:

```
nano Makefile
```

There is nothing much to see in this one, back out by pressing control + x.

To compile a project, simply call "make" in the directory of the project.

```
make
```

```
olli@NeppiUbs:~/riot/RIOT/tests/leds$ make
Building application "tests_leds" for "native" with MCU "native".

"make" -C /home/olli/riot/RIOT/boards/native
"make" -C /home/olli/riot/RIOT/boards/native/drivers
"make" -C /home/olli/riot/RIOT/core
"make" -C /home/olli/riot/RIOT/cpu/native
"make" -C /home/olli/riot/RIOT/cpu/native/periph
"make" -C /home/olli/riot/RIOT/cpu/native/vfs
"make" -C /home/olli/riot/RIOT/drivers
"make" -C /home/olli/riot/RIOT/drivers/periph_common
"make" -C /home/olli/riot/RIOT/sys
"make" -C /home/olli/riot/RIOT/sys/auto_init
   text    data     bss     dec     hex filename
  21365     372   47684   69421   10f2d /home/olli/riot/RIOT/tests/
ve/tests_leds.elf
olli@NeppiUbs:~/riot/RIOT/tests/leds$
```

Without additions, we compile the code for native. This basically emulates the code without hardware. We are interested in compiling code for out DK. RIOT checks if a BOARD variable exists, and uses it to determine the target platform. This can be set in the Makefile or in the terminal by adding the line "BOARD=nRF52dk." For now, we will make the board a session wide variable:

```
export BOARD=nRF52dk

make
```

```
olli@NeppiUbs:~/riot/RIOT/tests/leds$ make
Building application "tests_leds" for "nrf52dk" with MCU "nrf52".
```

The target is now right. Without further ado, let's make sure that RIOT knows which port our DK is connected to and then flash:

```
export PORT=/dev/ttyACM0

make flash
```

If everything was configured properly, the LEDS should now be flashing! Note that the port is defined to be ACM0 by default in hte makefile, making the PORT variable declaration redundant. However, this must be done in the case the DK uses a different port. Adapt the command accordingly.

4.2 LED with terminal

Now we need to test if our terminal connection works. Peek inside the "main.c" file:

```
nano main.c
```

Explore the code so you get an idea what is happening. Find where the loop starts, and add a statement that prints something:

```
    while (1) {
        puts("woo it works");
#ifdef LED0_ON
        LED0_ON;
        dumb_delay(DELAY_LONG);
        LED0_OFF;
```

Back out and save with control + x.

Now to read the output of the DK, add "term" to the make command

```
make flash term
```

After a while, lines of text should be visible on the terminal.

```
Script processing completed.

/home/olli/riot/RIOT/dist/tools/pyterm/pyterm -p "/dev/ttyACM0" -b "115200"
No handlers could be found for logger "root"
2018-05-14 01:41:44,798 - INFO # Connect to serial port /dev/ttyACM0
Welcome to pyterm!
Type '/exit' to exit.
2018-05-14 01:41:57,832 - INFO # woo it works
2018-05-14 01:42:11,038 - INFO # woo it works
2018-05-14 01:42:24,241 - INFO # woo it works
```

If everything worked, congratulations! If not, it is time to troubleshoot. This ends the RIOT setup portion.

5. Brief introduction to RIOT

At first glance, RIOT can be confusing. Because RIOT aims to be platform independent, much of the code needs to be decentralized. However, understanding the structure becomes easier if the analysis is started from the boards. The supported boards are most easily read from RIOT's github.

In our case, the board is nRF52dk. Taking a peek inside the nrf52dk directory, a common pattern can be observed. An include folder and a bunch of makefiles. The Makefile.dep lists the software dependencies the board needs. Makefile.features lists all the hardware features that the board provides for modules to use, such as i2c or bluetooth. The makefile and makefile.include make sure that everything is included properly. In our case all these files point to the common nRF files, meaning they share features with other "nrf52xxxdk" models.

The include directory contains header files that contain information about the board. In nRF52dk's case, 4 leds and 4 buttons have been defined. The macros used to toggle LEDs in the LED test were also defined here.

In the common files at /boards/common/nrf52xxxdk/include more hardware information is defined. Periph_conf.h defines how hardware peripherals such as i2c and spi are wired to the physical pins. If a project needs to change wirings or add new peripherals, it is done here. Also of note are the timer and clock definitions here.

The makefile.include in the nrf52xxxdk files exports CPU = nrf52. This means that now that RIOT knows the board, RIOT can now build the corresponding CPU code. This leads us to the CPU directory, which is organized in the same way as the board directory. Code here defines how the cpu is initialized, and how the peripherals are defined using the manufacturers code. The cpu code needs to be seldom modified, unless there is a bug that needs to be fixed. However, many important pieces of information can be found in the CPU code, such as what clock speeds are used and ADC resolutions.

Everything so far has been achieved by simply defining BOARD=nrf52dk. If more bells and whistles need to be added to a project, this happens with modules. For example, compare the hello-world and timer_periodic_wakeup examples. The hello world merely outputs text to the screen, and does not need a module. The periodic wakeup example however has the line "USEMODULE += xtimer", which indicates that the program uses the "xtimer" module. If the timer project would wish to include a temperature sensor to it, then for example "USEMODULE += bme280" could be added. List of supported drivers lies in the drivers directory.

Finally, there are the parts a regular user probably doesn't have to worry about. The core directory holds the RIOT kernel, which contains the scheduler and threading code. Dist and pkg contain libraries and tools that come with RIOT. The template makefile in most cases includes everything needed as long as the board is defined properly.

Getting a new project going starts with the following steps:

- Make a new project directory in the main directory.
- Copy the template makefile and modify it to suit the target platform.
- Start from a simple example such as flipping a LED and expand out.
- Use the existing modules to add features to the program.
- If a new module is needed, copy from the existing modules as much as possible. (Code structure, makefiles, header files.)

As a final note, if a hardware debugger is needed, the arm toolchain includes one. RIOT needs to know where to find the debugger, and if the guide was followed it can be added as follows:

```
export DBG=~/gnuarm/gcc-arm-none-eabi-7-2017-q4-major/bin/arm-none-eabi-gdb
```

Then to start the debugger type the following in the directory of the project which needs debugging:

```
make debug
```

The code can be stepped through by typing "step", or continue until next breakpoint with "c". The debugger can be quit with "q" and a backtrace can be printed with "bt". There are many commands, which can be explored with "help". Best of luck debugging.

6. Cheat sheet

At the start of a new session, copy paste (or scroll past commands with arrow keys) these:

```
export PATH=${PATH}:~/gnuarm/gcc-arm-none-eabi-7-2017-q4-major/bin
```

```
export DBG=~/gnuarm/gcc-arm-none-eabi-7-2017-q4-major/bin/arm-none-eabi-gdb
```

make sure that the DK is plugged in and checked in the VirtualBox devices. Make sure the port is correct.

```
sudo chmod 666 /dev/ttyACM0
```

```
export PORT=/dev/ttyACM0
```

7. Useful links

RIOT github - https://github.com/RIOT-OS/RIOT

RIOT tutorials - https://github.com/RIOT-OS/Tutorials

RIOT documentation - https://riot-os.org/api/

RIOT GPIO documentation - https://riot-os.org/api/group__drivers__periph__gpio.html

RIOT multi-threading documentation - https://riot-os.org/api/group__core__thread.html

RIOT network stack documentation - https://riot-os.org/api/group__net__gnrc.html

Nordic semiconductor infocenter - http://infocenter.nordicsemi.com/index.jsp

Nordic's Bluetooth low energy tutorial - https://devzone.nordicsemi.com/tutorials/b/bluetooth-low-energy/posts/ble-advertising-a-beginners-tutorial