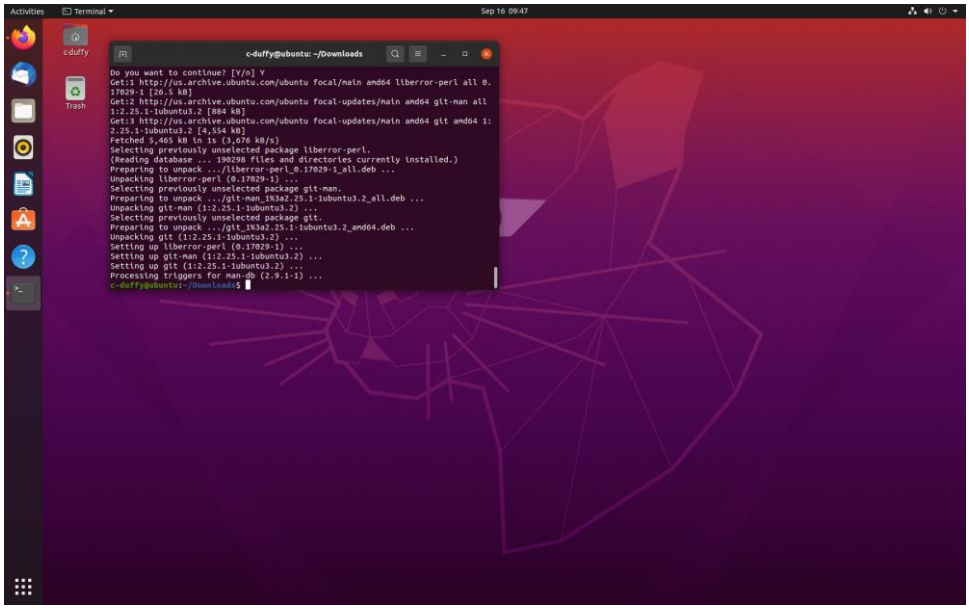
Title: Getting Started with the IoTBoard and Linux
Author: Craig Duffy 16/09/21
Module: Embedded Systems Programming
Awards: BSc Computer Science
Prerequisites: Some Linux skills, and basic computer architecture

Overview

The purpose of this worksheet is to get you started connecting to the IoTBoard (referred to as board from now on), to use git to download the files need to connect to the board from the Linux command line. As well as Linux skills you will learn a little about git, openocd, telnet and minicom.

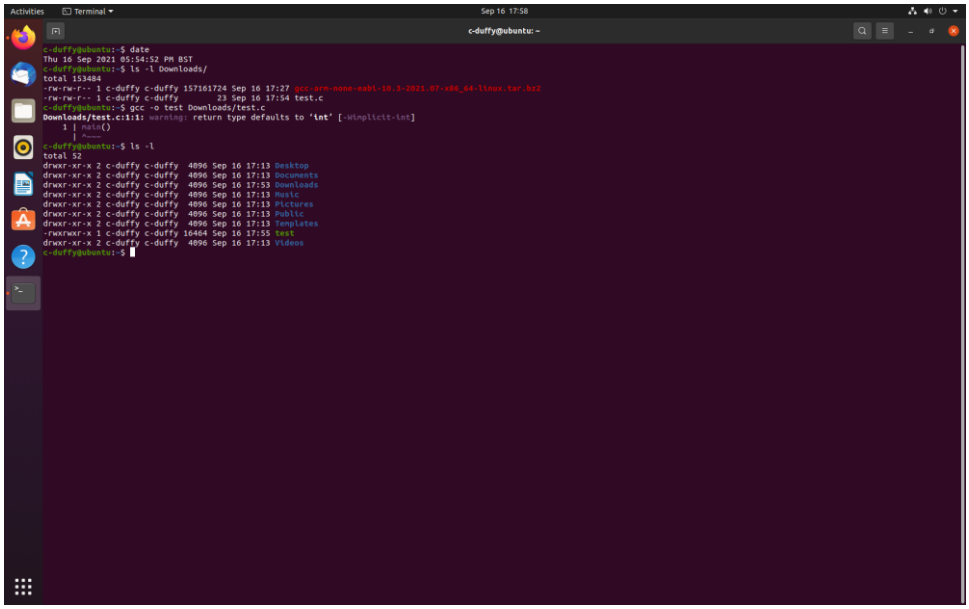
Using Linux

In the lab you need to log in to your account using your standard UWE credentials. Once in you will need to access a terminal.



An ubuntu terminal

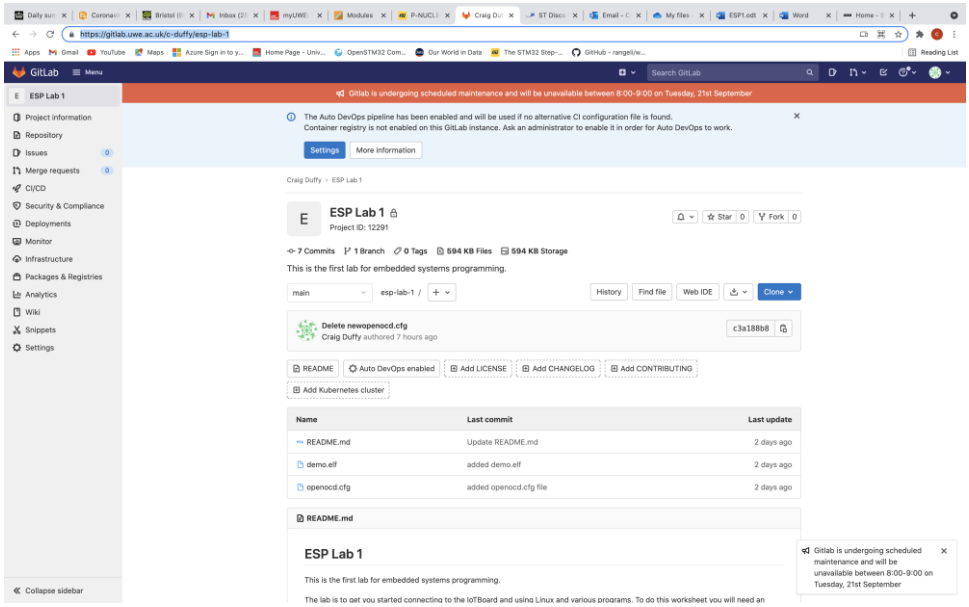
In the terminal you can type commands, for example you can print the date and time, list the files in a directory, compile a program and so forth:



git

In order to connect to our boards we will first need to download some files and we will be using **git** for this. git is going to be really useful for you in many later modules and also probably in your professional life, so learning to use it will be useful. For this worksheet we are doing the most simple things with it, but later on you will use more and more of its features.

In order to use git you will need to access the UWE gitlab server. The files, called a repo in git speak, are given in this link - https://gitlab.uwe.ac.uk/c-duffy/esp-lab-1. To access this repo you will need to login to gitlab using your UWE credentials. You should see something like this once logged in.

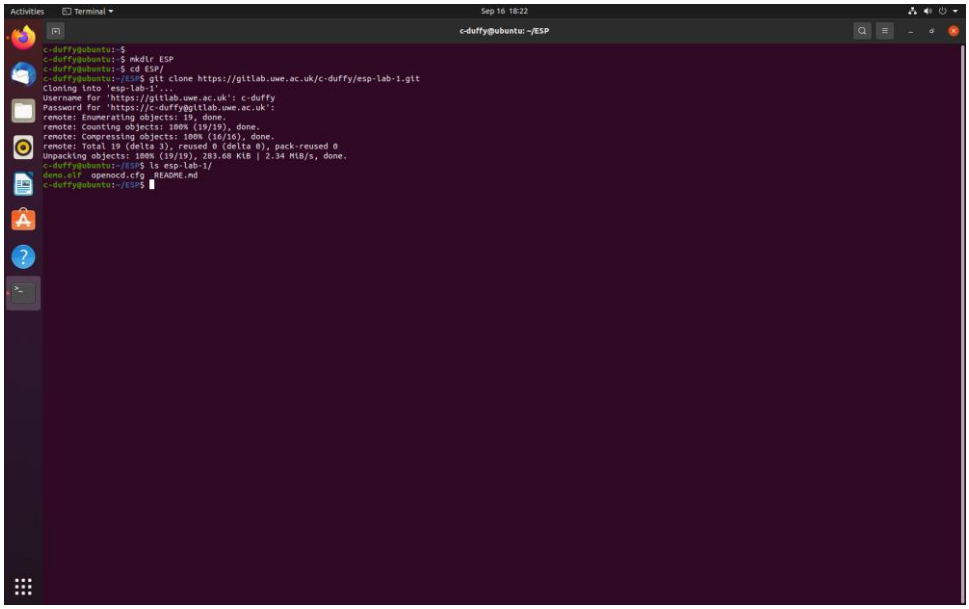


The ESP Lab 1 git repo

You will notice that the repo is pretty small – 3 files: a README, a binary (demo.elf) and a configuration file (openocd.cfg). These are needed for us to connect to our board. The process of copying a repo is called cloning – you will have probably noticed by now that computing people can't

help themselves in renaming every single command they use! In order to clone the repo click on the Clone button. You will get a choice of methods – ssh and https – for some reason UWE ITS doesn't allow ssh, so we will have to use https. Click on the clone https copy icon to grab the repo path name.

You can now go to your terminal. It would be a good idea to create a directory (the **mkdir** command) for your work now so that you can put your repo into it, I have called mine ESP. Change into the ESP directory (**cd** command) and issue the **git clone** command followed by the pasted pathname - *git clone https://gitlab.uwe.ac.uk/c-duffy/esp-lab-1.git*. Hit return and you should see something like the following.



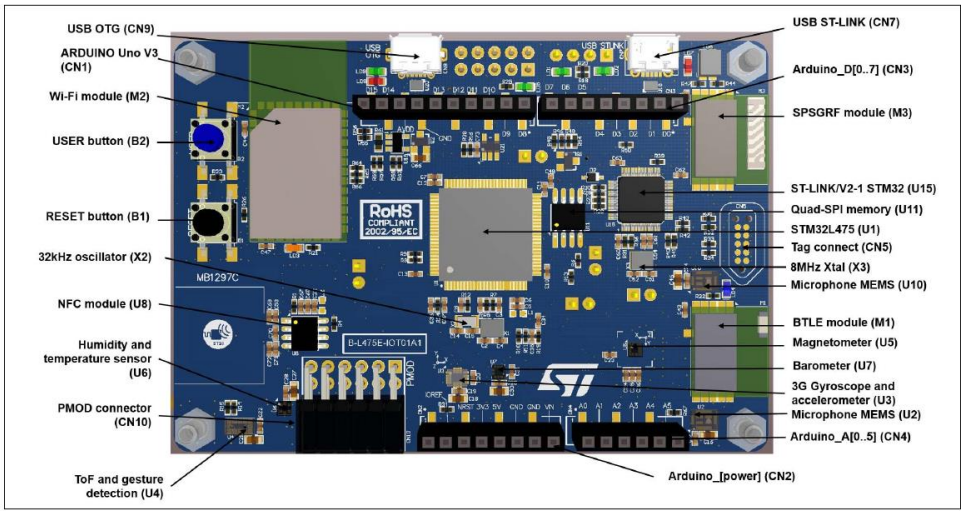
Cloning the gitlab esp-lab1 repo

When prompted for your username and password use your standard UWE login details – the same ones you used to login to gitlab earlier. Once the repo has been downloaded you should be able to change into the directory – cd esp-lab-. If you list the files using ls then you should see the same files that you saw on the gitlab repo page. If you use **ls –la** (**l**ong listing **a**ll files) you should see a .git directory which contains all of the git metadata for the repo.

We are now in a position to connect to the board and flash the test program (demo.elf) onto the board. This will involve creating some new terminal tabs, connecting the board and then running the **openocd, telnet** and **minicom** programs.

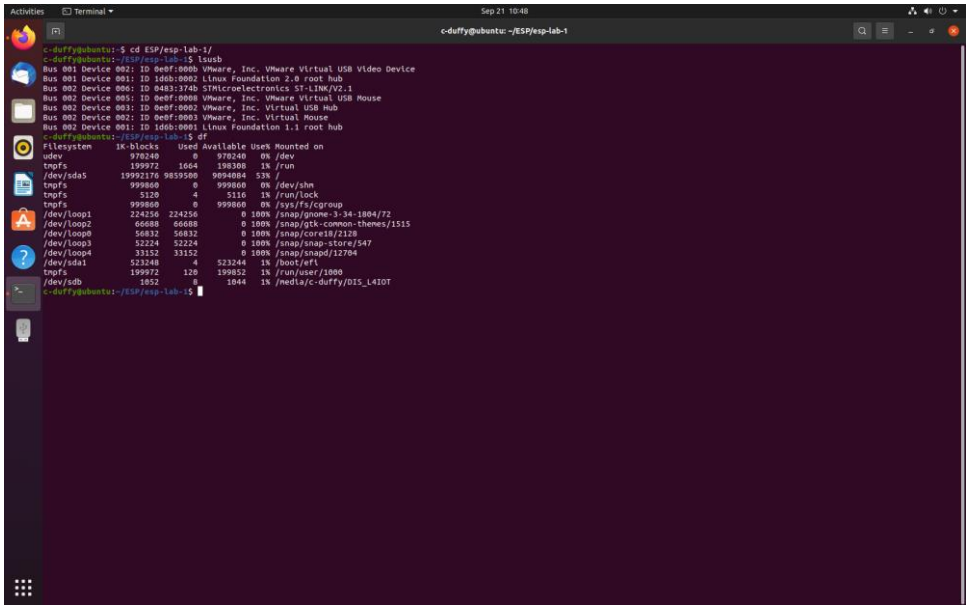
Connecting the board

The board connects to the Linux host PC via a male USB A to USB micro B cable. The board has 2 USB connectors so select the STLINK one. When you plug the board into the Linux machine, the red power light on the board should go on – other things may happen dependent upon what code was loaded onto the board before you used it. You should get a message from Linux telling you a file system has been mounted – called DIS_L4IOT. This is not the standard behaviour for a board plugged into a USB slot but the board is running arm's mbed operating system which causes this to happen. We won't be using the file system interface to program the board but will talk to the board directly using openocd.



The stm32l475 IoTBoard

It is possible that you may get a board from which mbed has been removed – in which case there will be no file system evident. In order to check that the board has connected you can use the **lsusb** command – this lists the connected USB devices. You should see something like this - the output from the **df** command is also given to show the mounted file system device:



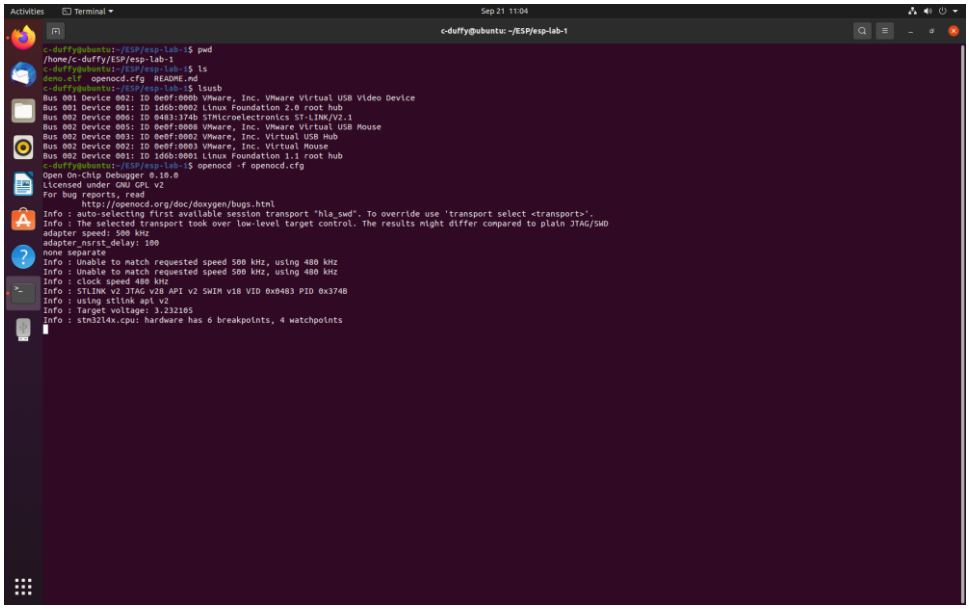
Output from lsusb and df

If the board doesn't connect and doesn't show up on lsusb then discuss this with the lab supervisor. If you are doing this on your own machine or on a vm then there are a number of things that can go wrong and will need sorting out, you can either discuss with me or look up the solutions online.

Once the board is connected then we can start 'talking' to it properly. To do this we will need some software which can speak directly to the board. There are a number of different programs and indeed protocols for talking to such boards. Many of these programs are proprietorial and can be very expensive. We are using a free, open source program called openocd. The ocd part stands for on chip debugger – not obsessive compulsive disorder, which you might imagine has something to do with it! This program allows us to talk directly to the board using the JTAG debug protocol. In later lectures and worksheets we will go into more details about JTAG, but for the moment all we

need to know is that it is the protocol which allows the board and our Linux machine to send commands and messages to each other.

To launch the openocd program we need be in the esp-lab1 directory as we need access to the files openocd.cfg and demo.elf. The openocd.cfg file has all the configuration details that we need to speak to our board – openocd can talk to loads of boards so has lots of different configurations. So, if you run the command – *openocd –f openocd.cfg* - you should see something like the following.

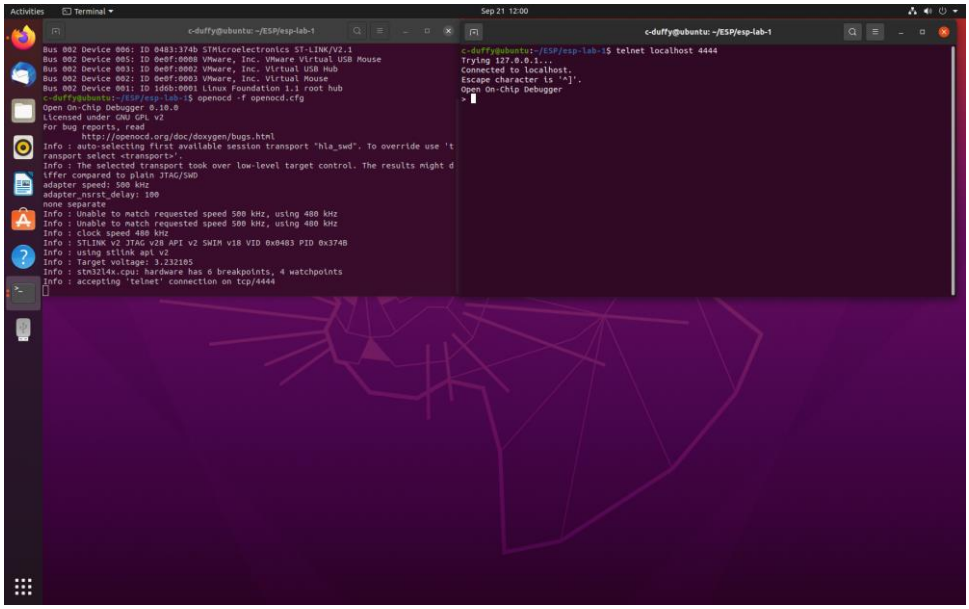


Connecting to the board with openocd

The board may well then start flashing lights and doing various things dependent upon current state of the code last loaded onto it.

openocd works as monitor program – it talks to the board but you can't talk directly to openocd. In order to talk to openocd you need another program and telnet is commonly used. telnet used to be the most common way of connecting to networked devices for the first 30 years or so of the Internet, until ssh became standard around 2006. telnet communicates with hosts over ports – a large number of the ports are standardised, port 23 is specifically for telnet, 20/21 are for ftp, 80 is for http and so on. Port numbers after 1024 are available for other protocols - so openocd's telnet port is 4444. This is set up in the openocd.cfg file and can be change. telnet isn't the only way of talking to openocd, as we will see later we can use the GNU debugger – gdb to do this too.

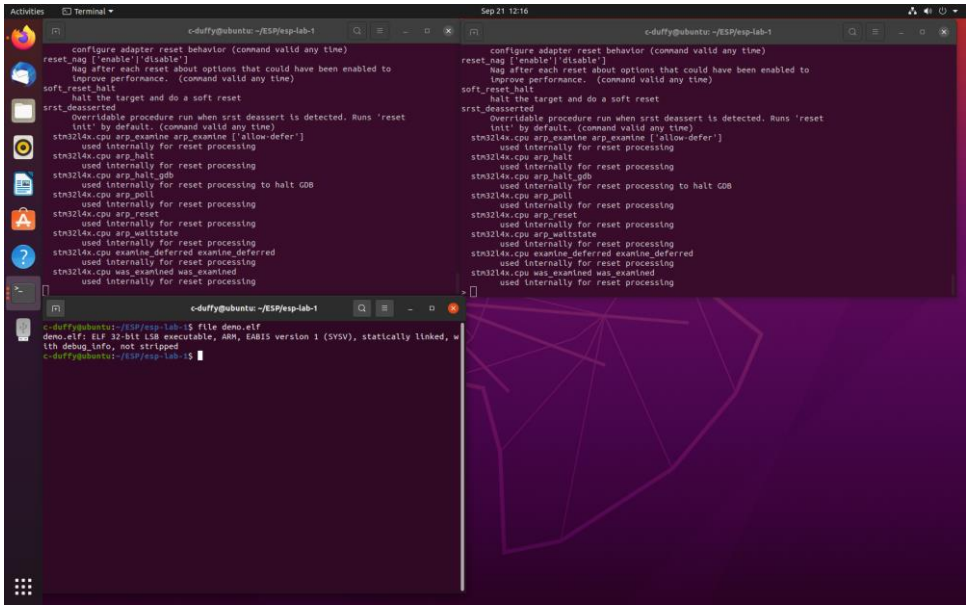
In order to telnet to openocd, and this to the board, you will require another terminal. I find creating tabs quite handy – the first tab is openocd, 2nd telnet and so on. Once your tab/terminal is open you can then connect to the running openocd using the *telnet localhost 4444* command. localhost means we are talking to the current host rather than over a network – nowadays telnet ports are blocked so remotely loging in is, sadly a thing of the past.



Starting a telnet session

You will notice that the telnet command comes back with the Open On-Chip Debugger message and the openocd terminal shows a message logging the connection. We are now, via openocd, talking to the board. Any command we issue on the telnet window will go directly to the board. If you type in help you should see quite a bit of help output from the board giving details of commands and warnings. As is common with software most of the commands will probably remain quite mysterious until you really need them. The commands we are going to be using in the session are **reset** and **flash**. reset allows us to set the machine into a known state – normally halted or running. Whenever we need to flash (write to flash memory) programs or data to the board it needs to be in a halted state. Try typing reset halt and then reset run. The board may well react especially if it has software that does something visible to the user. You will also note that the openocd window logs everything that goes on.

We are now in a position to write some code to the board. Before we do this we need to make sure that the board is halted by running reset halt in the telnet window. Next, we need to upload some code, that has been compiled into a 32 bit arm executable onto the board. This is our demo.elf program. If you open a new tab/window you can use the Linux **file** command to check the demo.elf program. You will see the output below – you will notice that is indeed reports that is an ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, with debug_info, not stripped. You can safely ignore the other information for the moment.



The output from file demo.elf

So now we can transfer the file up to the board. This is done via the telnet window – using the command *flash write_image erase demo.elf*. This lets the board know that we are writing an executable image and that we need to erase the existing memory first. Flash memory always needs to be reset to a known state before it can be written to – again we will cover this in later labs and lectures. You should receive a message like this back form the board:

```
>flash write_image erase demo.elf
auto erase enabled
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x20000050 msp: 0x20018000
block write succeeded
wrote 145408 bytes from file demo.elf in 7.725911s (18.380 KiB/s)
>
```

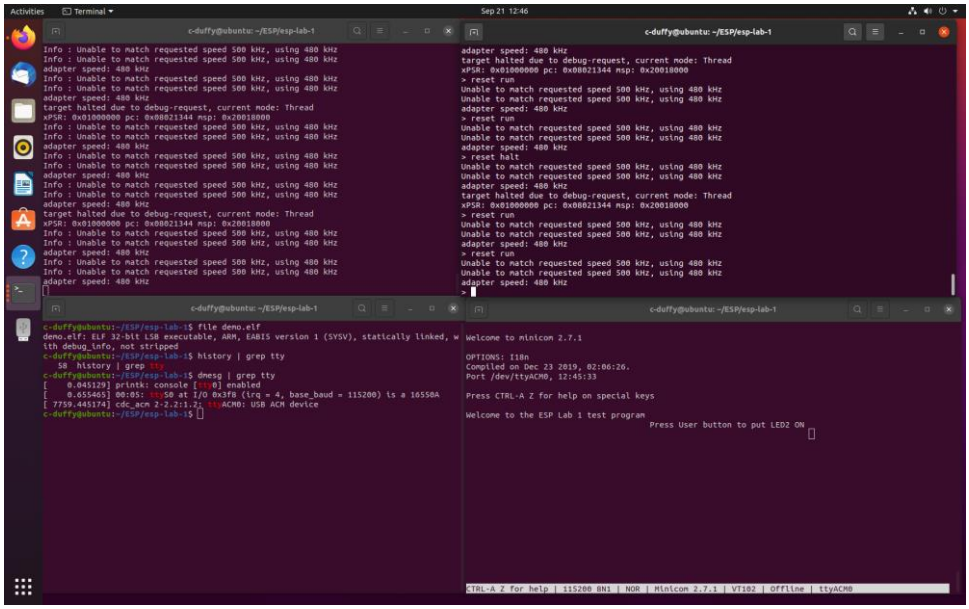
This means we have succeeded in writing our test program to the board. If the board complains either you didn't do a reset halt or openocd is not in your esp-lab1 directory so it can't find the file.

We are now almost ready to run the program, however we need one more piece of information before doing so. The demo program runs through testing most of the board's sensors and memory, giving some output messages and requires some input from a keyboard to run. In order to achieve this we need terminal emulator program to capture and send the input/output. The program we are using is called **minicom**.

minicom terminal emulator

There are a number of different terminal emulators and we are using minicom – we could have used a number of other ones. If we were doing this on Windows we would probably use putty. Terminal emulators are quite historic pieces of software and traditionally they were used to connect computers to the Internet via devices such as modems. Our terminal device is going to be using the USB cable for the serial/terminal data as well as it being used for the JTAG data and incidentally the board's power. To connect to the board's serial interface via a terminal, first open a new tab/terminal, then

type in the command *minicom -o IoTBoard*. The -o argument stops minicom sending modem control characters, which it does by default, the IoTBoard argument means it picks up its settings from a pre-built-in configuration file called IoTBoard. We are now ready to run the code on the board by typing reset run on the telnet connection, and we should see the below.



minicom talking to the board

If you notice that minicom keeps scrolling to the right – it does a new line but no carriage return. To fix this type in **ctrl a z** followed by **u** to turn the carriage return on. In the manual page for minicom it is described as a 'friendly serial communications program', I would hate to think what an unfriendly one would be like. You will also notice on the above output on the bottom left-hand command window I did a **dmesg | grep tty** command. This was to check the kernel messages, **(dmesg)**, pipe the output **(|)** and then search for anything with the string tty in it (**grep tty**). It is possible that some other USB or serial devices have been plugged in and therefore the standard setting of /dev/ttyACM0 won't work. If this is the case then in the minicom window typing **ctrl a o** to go to settings you can reset the minicom's serial port configuration.

Finally, we get the actual program! The program firstly requires you to press the user button on the board – the blue button on the right hand side. Once done it then runs through a series of checks – the first 2 being on the quad serial peripheral interface (QSPI) memory controls. Each test can be exited – by typing q, after the first 2 tests, you can do temperature tests, magneto and gyro tests. At any point pressing the black button, the reset button, will restart the entire program from the beginning.