Lab 0 - Personal Lab Setup

Goals

- Get a build environment configured for the future labs and project
- Run C code on the Microbit
- Simple debugging in an embedded context

Equipment

- Computer that you will use for labs
    - If Windows: needs at least 20 GB of space
    - USB ports
- Micro:bit and cables (you can do all of the setup except testing without this)

This lab is required for Fall 2023. If you run into problems, please reach out on Piazza or during Office Hours and I will provide help!!

Fair warning: I'm certain there are some bugs in here. Probably some sentences that just trail off part-way through too. Let me know and I'll fix it.

There's no submission here to prove you've completed this. Just be sure to do it before we have our first lab session!!

Index:

MacOS Instructions

The good news here is that MacOS natively supports all of the software we need. Since it's not a clean installation though (presumably you've been using your Mac for other programming tasks) some of these steps might fail, or react differently. Power through as best you can and ask questions whenever you need!

To my knowledge this will all work great on both Intel and ARM Macs. I have a personal Intel Macbook that can program these boards (although I installed all the stuff years and years ago). I tested all of these instructions on an ARM M1 Macbook Air.

- Open a terminal window
    - We'll run all of the following `green` commands in the terminal window

- Install MacOS command line developer tools by running `xcode-select --install`
    - You may have to agree to some terms and conditions
    - This might complete immediately if it's already installed. If not, it'll take a few minutes to finish (and it's just *terrible* at estimating the time remaining for some reason)

- Install Homebrew by running: `/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"` (that's all one command)
    - If that's too much to type, you can go here and copy the "Install Homebrew" command, it's the same thing: https://brew.sh/
    - You should also be able to copy-paste from here into terminal. You'll have to use Cmd+Shift+V to paste, or right click and choose "Paste".
    - You'll have to type your password in to run it. Note that it won't show you any feedback when you type, just type the password anyways and hit enter
    - This shouldn't take all that long to install

- Homebrew might say "Run these two commands in your terminal to add Homebrew to your PATH:". If it does, run both of those commands.
    - You can highlight, right click, copy, and then paste it to run it
    - They add things to PATH so you can find executables later. We can always fix after-the-fact if we need to. You'll know things are broken if it can't find the `arm-none-eabi-gcc` executable even after you install it.

- Install our compiler with `brew tap ARMmbed/homebrew-formulae && brew update && brew install ARMmbed/homebrew-formulae/arm-none-eabi-gcc`
    - Again, that's one long command

- Install our JTAG tools with `brew install open-ocd`

- Install our serial console with `pip3 install pyserial`
    - You might get a warning that “blah blah is not on PATH”. You’ll need to add it to your PATH. If you don’t know how to do this, the following will work:
      ```
      echo "export PATH=\"`python3 -m site --user-base`/bin:\$PATH\"" >> ~/.zshrc
      ```
        - All one command
        - If you’re using Bash, that should either be ~/.bashrc or ~/.profile instead of ~/.zshrc
        - Then you’ll need to close your terminal and open a new one

- Clone the class github repo with `git clone https://github.com/nu-ce346/nu-microbit-base.git`

- `cd` into the repo and then run `git submodule update --init --recursive`
    - This is the magic command that fixes all git submodule issues
    - This will take a hot minute to run. There’s lots of stuff to download

- `cd` into “software/apps/blink” and run `make`
    - This should compile the code if everything is working!!
    - You’re done! There’s some bonus stuff below though.

- IF COMPILING DOESN’T WORK:
    - It could be an issue with things not being in your PATH. That would result in it not finding certain executables you installed, like `arm-none-eabi-gcc`. Try figuring out where they installed and add them to your PATH.

    - It could be an issue with a Space character in the file path. None of the directories including the code may have a space character (or other special character like plus or colon). They break Makefiles hard. A common issue here is if your username has a space in it. The solution is to move the directory to somewhere without any spaces in the folder names.

- If you have a microbit already, you can run `make flash` and that will actually load the program onto the board. The LED should start blinking

- Also make sure you have some editor installed that you're happy with. You'll use terminal to compile and flash the board, but you don't have to edit files in terminal if you don't want to.

Windows Instructions

We don't support native Windows, so you really need to have Linux instead. You've got two options here: you could install a virtual machine (these instructions cover that) or you could use Windows Subsystem for Linux (WSL) if you're feeling really bold. Instructions for WSL are at the end.

This is sort of a pain. These are the longest instructions of any part of this writeup. Most of that time was spent trying to bludgeon Virtual Box into being usable, which is pretty disappointing in the year 2023…

1. Install a Virtual Machine

A virtual machine (VM) allows you to run an operating system inside a windowed environment while running windows. Basically, it creates a "virtual" computer inside your computer. This has some consequences about speed and security (see CS343), but works great for our needs.

- Install VirtualBox. You can download it here: https://www.virtualbox.org/wiki/Downloads
    - You want the "Windows Hosts" option. At time of writing this was version 7.0.10
    - VirtualBox is most well-known VM software and should suffice
    - I personally use VmWare Workstation Player which is a bit fancier and would also work. Other VM software would be fine too. These instructions will be VirtualBox based though.

- Follow the installation instructions to install it
    - The defaults are all fine. Just keep hitting "yes" for a while.

2. Install Linux on the Virtual Machine

Many flavors of Linux would work just fine, but we're going to use Ubuntu. It's very popular and widely used with lots of support on the internet.

- Download Ubuntu: https://ubuntu.com/download/desktop
    - Download the most recent LTS (Long Term Support) version. At time of writing this was 22.04.3 LTS
    - This is about 5 GB in size, so it's going to take a while to download.
    - Be sure to delete it after you're done with this setup if you're short on space!

- Open VirtualBox and hit the "New" button on the homepage
    - Add a name for the machine
    - Choose the Ubuntu `.iso` file as the "ISO Image"

        - VirtualBox should automatically figure out that it is an Ubuntu 64-bit image and will say that it can install it “unattendedly”. This is great and we want this.
    - Hit next

- You have reached the “Unattended Guest OS Install Step”
    - Enter a username and password. You’ll need to remember the password to log in again in the future!!
    - Select the “Guest Additions” checkbox
        - This installs extra things that make Ubuntu aware it’s in a virtual machine and lets it adapt to do that better. For example, it’ll automatically adjust its “monitor size” to whatever the size of your VirtualBox window is. We want this.
    - Hit next

- You have reached the “Hardware” step
    - Set the “Base Memory” to roughly half of your computer’s RAM, but not less than 2048 MB.
        - 4-8 GB (4096-8192 MB) is the sweet spot here
    - “Processors” can stay at 1
    - Do not check the EFI checkbox
    - Hit next

- You have reached the “Virtual Hard disk” step
    - Choose “Create a Virtual Hard Disk Now” (should be selected by default)
    - Set the “Disk Size” to 100 GB
        - Do **NOT** check the “pre-allocate full size” checkbox
        - The hard disk will not take up 100 GB. It will only take up the space it needs and can grow up to 100 GB if you really needed it
    - Hit next

- You have reached the “Summary” step
    - Hit Finish
    - VirtualBox will automatically start up a window and will install Ubuntu for you with the correct settings. Let it go on its own and don’t disturb it. This will take a few minutes at least.
        - I wouldn’t click inside the window while it’s running in case that disrupts things. Probably not, but why risk it?
        - It’ll eventually finish installing and restart itself. When it stops on the login page, it’s done

- Log in to Ubuntu using the password you created a bit ago
    - You remember it, right? 🙃

- You'll have to click through a "Welcome to Ubuntu" dialogue for a bit. Nothing important there.

- Open a terminal in Ubuntu
    - You can click the desktop and hit "Ctrl+Alt+T"
    - Or you can hit the 9-dot box pattern in the bottom left and then type "terminal"
    - You'll type the `green` commands into terminal and run them

- Add yourself as a sudoer (no idea why this isn't enabled by default)
    - Run `su -`
    - Type your password
    - Then run `sudo adduser [your username] sudo`
    - Then reboot the computer again

- Open a terminal again and update Ubuntu
    - Run `sudo apt update`
    - Run `sudo apt upgrade`

- Install Virtualbox guest extensions
    - Run `sudo apt install build-essential dkms linux-headers-$(uname -r) virtualbox-ext-pack virtualbox-dkms virtualbox-guest-utils virtualbox-guest-x11`
    - That's one long command
    - You should also be able to copy-paste from here into terminal. You'll have to use Cmd+Shift+V to paste, or right click and choose "Paste".
    - Actually power off Ubuntu this time

- Go to the Virtual Box settings for your machine (the yellow gear)
    - Choose the Display page on the left
    - For "Video Memory" increase it to 128 MB
    - Check the "Enable 3D Acceleration" checkbox
    - Now you can boot up Ubuntu again and log in

- You might be able to drag the bottom-right corner of the VM window to increase the size of the Ubuntu desktop now. See if that works acceptably.
    - If it doesn't, in Ubuntu, right-click the desktop and choose "Display Settings"
        - Set the Resolution to something reasonably large. The Virtual Machine window should get larger.
        - Play around and find something that is acceptable for using Ubuntu

- You should now have a "usable" Ubuntu installation. Continue on to the "Linux" instructions to actually get the stuff installed that you need for lab.

Linux Instructions

This part is pretty easy. If you've got Linux you're only a few quick steps away from having a working setup.

- Install various pre-requisites: `sudo apt install build-essential python3-pip python3-serial git vim emacs meld screen`
- Install our compiler: `sudo apt install gcc-arm-none-eabi`
- Install our JTAG tools with: `sudo apt install openocd`
- Clone the class github repo with `git clone https://github.com/nu-ce346/nu-microbit-base.git`
- `cd` into the repo and then run `git submodule update --init --recursive`
    - This is the magic command that fixes all git submodule issues
    - This will take a hot minute to run. There's lots of stuff to download
- `cd` into "software/apps/blink" and run `make`
    - This should compile the code if everything is working!!
    - You're done! There's some bonus stuff below though.
- IF COMPILING DOESN'T WORK:
    - It could be an issue with things not being in your PATH. That would result in it not finding certain executables you installed, like `arm-none-eabi-gcc`. Try figuring out where they installed and add them to your PATH.
    - It could be an issue with a Space character in the file path. None of the directories including the code may have a space character (or other special character like plus or colon). They break Makefiles hard. A common issue here is if your username has a space in it. The solution is to move the directory to somewhere without any spaces in the folder names.
- If you have a microbit already, you can run `make flash` and that will actually load the program onto the board. The LED should start blinking
    - If you're using a VM, before flashing when you first plug in the Microbit, you'll need to go in the VM to "Devices->USB" and check the box next to "Arm BBC micro:bit" to attach it to the VM

- Also make sure you have some editor installed that you're happy with. You'll use terminal to compile and flash the board, but you don't have to edit files in terminal if you don't want to.

Bonus - WSL Instructions

Courtesy of Joshua Fiest

This is still fairly experimental. You may run into weird issues here. This is not recommended, but you're welcome to try it if you want to! Be sure the read the whole thing first, as there are some updates at the end.

I was able to get the required software to program the Microbit working on my computer using WSL (Windows Subsystem for Linux). This worked for me on Windows 10 21H1. Here's how I did it.

1. Install WSL
    1. Using the search bar on your computer, go to Turn Windows features on or off, then click the check boxes to enable Windows Subsystem for Linux and Hyper-V Platform (If you have Windows Pro, you can also enable Hyper-V Management Tools to make virtual machines, but you don't have to)
    2. In an administrator command prompt, run the command "wsl --update" followed by "wsl --shutdown"
    3. If you already had WSL installed and set up, make sure you are running WSL2 by running the command "wsl -l -v." It should list the installed Linux distros and their versions, make sure the version is 2.
    4. In the Microsoft Store, search for and install Ubuntu
    5. If you are running Windows 10, you will also have to install an XServer like GWSL so that WSL can use a GUI (graphical user interface)
        1. To install GWSL, search for it on the Microsoft Store, then install it. Once it's installed, run it, and click on GWSL Distro Tools -> Display/Audio Auto-Exporting to configure WSL to use GWSL for graphics.
        2. Note: configuring GWSL will prevent WSLg from working properly. If you upgrade to Windows 11, be sure to revert the changes made by GWSL in the Linux installation by opening GWSL in Windows and clicking on GWSL Distro Tools -> More Shells and Options -> ~Clean GWSL Additions. You can then uninstall GWSL since it isn't needed to get WSL graphics to work on Windows 11.
2. Install the 64-bit Windows version of OpenOCD
3. Start Ubuntu by typing it in the search bar and set up your account. It may take a couple of minutes to start the first time.
4. Follow the instructions in the Getting Started lab to set up your Ubuntu installation for programming the Microbit
5. ~~When programming the Microbit, run the J-link Remote Server that was installed with J-link on your windows computer. It should give you an IP address that you can type (or copy and paste) when asked for it while running "make flash." If you are using Windows 10 with GWSL, make sure to start GWSL before running "make flash" or you won't get the graphical pop-up asking for the IP address.~~

6. ~~Because Ubuntu doesn't have direct access to your USB port, you will need a Windows serial console application instead of miniterm. I used the one built into the Arduino IDE, but PUTTY or something else should work too.~~

~~On Windows 11 21H2, there are two very annoying bugs: the J-link remote server may not show the IP address (you can get it by running "ipconfig" in the Windows command prompt) and if you hit the "Enter" key on your keyboard in the J-link emulator selection screen from Ubuntu, it will continue to think Enter is pressed the next time that window opens, which will prevent you from using it. To get around this, click "Yes" instead of using the Enter key, and if you accidentally hit the Enter key, you can restart Ubuntu by running "wsl --shutdown" in the Windows command prompt.~~

Update 11/9/2021: WSL2 now has compatibility with USB devices. If set up properly, this removes the need for using a separate Windows serial monitor and the J-link remote server. https://devblogs.microsoft.com/commandline/connecting-usb-devices-to-wsl/

For some reason, when using the USB port in WSL2, access will be denied unless you run the command (miniterm or make flash) as sudo.