# Lab 0 - Personal Lab Setup (OPTIONAL)

**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
    - Needs at least 20 GB of space
    - USB ports
- Micro:bit and cables (you can do all of the setup except testing without this)

**Note:** This lab is optional. If you want a lab environment set up on your personal computer, follow these steps. Otherwise, the lab environment will be set up on the computers in CG50 for you.

# Lab Steps

## 1. Create a virtual machine (optional)

The most straightforward approach to getting a build environment configured is to install Ubuntu on a virtual machine. That will be a clean start to add packages too that also won't affect anything on your computer.

You are welcome to not follow this advice. I have a Macbook that I'm capable of programming boards on. Be aware that if you have an ARM Macbook (M1/M2), you will not be able to install a virtual machine. I also have many friends who use other distributions than Ubuntu. In any of those cases, skip to step number 2.

I have also had students get WSL (Windows Subsystem for Linux) working! See Bonus - WSL Instructions at the end of this writeup. After completing that, continue to step 2.

Assuming you want to install an Ubuntu virtual machine:
Download the following:

- Virtualbox download link: https://www.virtualbox.org/wiki/Downloads
    - VirtualBox 6.x.xx platform packages (for whatever X is these days)
    - Note: VMWare Player also works instead of Virtualbox
- Ubuntu download link: https://releases.ubuntu.com/20.04/
    - ubuntu-20.04.x-desktop-amd64.iso
    - Note: I'm pretty sure this will all work on Ubuntu 22.04, but I haven't tried it

Creating the machine:

- Open Virtualbox. Click "New"
- Type: "Linux", Version: "Ubuntu (64-bit)"
- Memory size: At most half of your machine's memory, hopefully 4096 MB.
- Hard disk: keep hitting defaults. Make sure it is "Dynamically allocated". Size of the virtual disk should be at least 100 GB (not the default of 10 GB, which is crazy small). Since it's dynamically allocated, it'll only actually use what it needs, but resizing the disk later is a huge pain.
- After creating it, you may want to change some settings (the "Settings" gear):
    - General/Advanced/Shared Clipboard: bidirectional
    - System/Processor/Processors: half of what's available for your system (usually 4)
    - Settings/Display: Video Memory increase to 128 MB and enable 3D Acceleration
- If you're on MacOS
    - You may have to "Allow" VirtualBox inside "System Preferences"/"Security & Privacy". This actually required a restart on my Macbook to get working.

Start the virtual machine (the "Start" arrow):

- Start-up disk should be the Ubuntu iso that you downloaded
- Click "Install Ubuntu" once that finally loads.

- Choose “Minimal installation” (unless you want OpenOffice, games, etc.)
- The default “Erase disk and Install Ubuntu” is correct. Click “Install now” (Don’t worry, this will only erase the virtual disk you created for the VM)
- Choose “Continue” on the pop-up warning you about disk sectors
- Choose “Log in automatically” when creating your account to make your life easier
- It’ll take a few minutes to do the installation
- You’ll eventually get to a screen that says “Please remove the installation medium, then press ENTER:”. Just hit enter.
- Click next through a bunch of setup windows.
- You should now have your own Ubuntu machine!

Update and install guest additions (which makes the VM resize and stuff):

- CTRL+ALT+T to open a terminal
- sudo apt update
- sudo apt upgrade
    - This will take a while
    - While it’s going, open Settings/Privacy/Screen Lock and disable/set-to-never everything on that page
- sudo apt install build-essential dkms linux-headers-$(uname -r)
    - Unfortunately you won’t be able to copy-paste yet. That and window resizing is what we’re fixing right now.
- In the virtualbox menubar, go to Devices/Insert Guest Additions…
- A pop-up should say “blah blah Would you like to run it?” Click Run and then type your password
    - That will eventually say “Press Return to close this window…” when it’s done
- sudo reboot
- This should now allow the window to be resized
    - I had problems where the screen would go black when it got too big. I powered off the VM, went to Settings/Display and increased Video Memory to 128 MB and enabled 3D Acceleration, which fixed it. (And I then added those instructions above, so hopefully you won’t have the problem.)

## 2. Install requirements

If you're using your own system, a bunch of these will be redundant, but won't hurt. I'll show the Ubuntu package name and sometimes MacOS instructions. Translate to whatever your own system is.

- sudo apt install build-essential python3 python3-pip python3-serial git vim emacs meld screen

- Install the gcc cross compiler for ARM microcontrollers
    - For the Linux version, you have to do this super manually. The following is one long line of code you can copy-paste into terminal to do it. (Ctrl-Shift-V)

        - `cd /tmp  && wget -c https://developer.arm.com/-/media/Files/downloads/gnu-rm/9-2020q2/gcc-arm-none-eabi-9-2020-q2-update-x86_64-linux.tar.bz2 && tar xjf gcc-arm-none-eabi-9-2020-q2-update-x86_64-linux.tar.bz2 && sudo mv gcc-arm-none-eabi-9-2020-q2-update /opt/gcc-arm-none-eabi-9-2020-q2-update && rm gcc-arm-none-eabi-9-2020-q2-update-x86_64-linux.tar.bz2 && sudo ln -s /opt/gcc-arm-none-eabi-9-2020-q2-update/bin/* /usr/local/bin/.`

    - For the MacOS version:
        - `brew tap ArmMbed/homebrew-formulae`
        - `brew install arm-none-eabi-gcc`

    - To check if this works run: arm-none-eabi-gcc --version
      It should autocomplete, work, and return 9.3.1 (or similar. At least version 9)

- Install the Segger JLink tools: https://www.segger.com/downloads/jlink/#J-LinkSoftwareAndDocumentationPack (the "J-Link Software and Documentation Pack" is what you want)
    - For the Linux version:
        - Open that website in your VM. And find "J-Link Software and Documentation pack, Linux, 64-bit DEB Installer". Click Download. Then accept terms and download software. Choose "Save File" which will put it in ~/Downloads/
        - sudo apt install ~/Downloads/<NAME OF DEB FILE HERE>
    - For the MacOS version:
        - Open that website in your VM. And find "J-Link Software and Documentation pack, macOS, Universal Installer". Click Download. Then accept terms and download software. Then install it as normal.

## 3. Download the repository

I've put together a git repo with some basic programs that can be loaded on the Micro:bit. It's based on nrf52x-base, which is a library I and my colleagues put together at Berkeley for programming our nRF52 based devices.

**IMPORTANT:** the directory you put the repo in, and *all of its parent directories* MUST NOT have spaces in their names. Otherwise, the build system won't work. So if you have a "CE 346" folder or something like that, rename it to remove the space.

- The base repo for this class is located at https://github.com/nu-ce346/nu-microbit-base
- Clone the repo locally and cd into the repo
    - Make sure the directory path where you clone it doesn't have any parent directories with special characters or spaces. The tools are sometimes brittle to that kind of thing.
- git submodule update --init --recursive
    - This will take a minute or two to run as it loads everything
    - The nrf52x-base submodule contains all the libraries needed for the nrf52840DK
    - You'll need to run this command again when things in that submodule change
- Check that code builds
    - cd nu-microbit-base/software/apps/blink/
    - make
        - Should create _build/blink_sdk16_blank.elf
- Check that JLink tools work
    - make flash
    - Should pop up a "J-Link VX.XXx Emulation selection" window. Which you should click no to. This is what happens when you try to program a board, but no board is attached.
    - You'll have to click No like four times. Sorry.
        - Or better, click in the terminal and just "Ctrl-C"

That's everything you can do without a Microbit. If you've got your own Microbit you want to set up, the next steps apply to that.

## 4. Change the Microbit's programmer firmware

If you have your own Microbit, updating the firmware will make JTAG easier to use with our class repo, and it's simple to do and reversible, so we're going to do it.

NOTE: this will not work for Microbit v2.2 (talk to Branden)

- Download the Segger Micro:bit v2 JLink Firmware
    - https://www.segger.com/downloads/jlink/#BBC_microbit
    - Note: Be sure to get the Micro:bit v2 version
- Follow instructions here: https://www.segger.com/products/debug-probes/j-link/models/other-j-links/bbc-microbit-j-link-upgrade/
    - Hold reset button while plugging in microbit to start "maintenance mode"
        - Microbit will appear as a storage device called "MAINTENANCE"
    - Drag and drop the new firmware into that USB storage device
        - Do not unplug until the LEDs stop blinking
    - Unplug device and replug (without holding reset button)
    - Device should now appear with JLink in its name
        - Sometimes the name doesn't change on MacOS. Not sure why. As long as flashing code to it works in the next step, everything is fine.
- NOTE: if you later (after this class) want to get back to the original JTAG firmware, you can follow the instructions here: https://microbit.org/get-started/user-guide/firmware/

## 5. Program a board

- Plug the board into the computer
    - WARNING: if you haven't loaded code on it before, the default app makes noise
        - And is rather annoying
    - You plug into the USB on the top of the board
- Attach the board to the VM
    - In the menubar, click Devices/USB/Segger-JLink (out of your USB devices)
    - If you hover over Devices/USB/ again, it should now have a check mark
    - You'll have to check this button each time you plug in a board. There will be a separate one for each board you have attached to the computer.
- In the blink app
    - `make flash`
    - It should pop up a window with a loading bar that uploads the code
    - Things like "Downloading file [_build/blink_sdk16_blank.hex]..." and "O.K." are good
    - Things like "J-Link connection not established yet but required for command" and "Connecting to J-Link via USB...FAILED: Failed to open DLL" are bad
    - Also, the board should start blinking the red microphone LED if it works

### 6. Get some apps working

- There are three good starter apps:
    - blink - blinks the microphone LED
    - printf - periodically prints a message from the board
    - error - demonstrates a hardfault and error messages on the board
- Commands to control them
    - make flash
        - To build code and load it onto the board over JTAG
    - miniterm /dev/ttyACM0 38400
        - To listen to serial output
        - (Any other serial console would work too)
        - If you're on MacOS and can't find miniterm:
            - python3 -m serial.tools.miniterm /dev/ttyACM0 38400
            - Or install minicom, a different serial console: brew install minicom
        - Note: it doesn't buffer output. Anything that happened before you opened it won't appear. Hit the "Reset" button at the top of the Microbit to start the currently loaded program again.
        - Also note: you don't have to close this when programming a board. Just leave it open in another terminal window. It should only stop working if you unplug your Microbit.
- Take a look at each of the starter apps and try out modifying board behavior

## Lab 0 Submission

- None. This lab is optional.

# Bonus - WSL Instructions

## Courtesy of Joshua Fiest

This is still fairly experimental. You may run into weird issues here.

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 J-link from the Getting Started lab
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.