



# Revolutionary code: from MIT printers to the Arduino

COM3505, Lecture 2  
Prof Hamish Cunningham



# Do you own a phone?



You are carrying a phone, but do you **\*\*own\*\*** it?!

Call me an old stickler, but I think if I own something then:

- I can take it apart and see what it contains
- I can modify it
- I can repair it

Do those things apply to your phone? How about your laptop? Your tablet? You **did** read the End User Agreement, didn't you?!

We buy stuff, the people who sell them claim we own them, but we have few rights over them. Think that's a bummer? You're not the first....

- *The End of Ownership: Personal Property in the Digital Economy*  
Aaron Perzanowski, Jason Schultz, MIT Press 2016 <https://mitpress.mit.edu/end>
- Open Rights Group UK: <https://www.openrightsgroup.org/>
- Doctorow, Cory. Pirate Cinema. 2012 <http://craphound.com/pc/download/>



# Return with me to Boston in the 1970s...



The **PDP 10** was a cool computer:



The first machine to make time sharing (multitasking) common (via the ITS OS)

Max memory... a whole megabyte!

The operating system code (assembler) was routinely shared and improved by a community of programmers ("hackers"\*) at MIT, including one **Richard Stallman**.

But then....

\* 'The use of "hacker" to mean "security breaker" is a confusion on the part of the mass media. We hackers refuse to recognize that meaning, and continue using the word to mean someone who loves to program, someone who enjoys playful cleverness, or the combination of the two.' Stallman, Richard. 2002. *Free Software, Free Society: Selected Essays of Richard M. Stallman*.



# Whaddya mean, I can't fix it?!



The 1980s: no more source, and sign an NDA even to get access to a binary.

Stallman went on to suffer the consequences of an aversion to NDAs when he was refused access to the source for a printer control program, even though his intention was to improve that program.

*'This meant that the first step in using a computer was to promise not to help your neighbor. A cooperating community was forbidden. The rule made by the owners of proprietary software was, "If you share with your neighbor, you are a pirate. If you want any changes, beg us to make them."' [Ibid.]*



# What to do?



*'So I looked for a way that a programmer could do something for the good. I asked myself, was there a program or programs that I could write, so as to make a community possible once again?*

*The answer was clear: what was needed first was an operating system. That is the crucial software for starting to use a computer. With an operating system, you can do many things; without one, you cannot run the computer at all. With a free operating system, we could again have a community of cooperating hackers—and invite anyone to join. And anyone would be able to use a computer without starting out by conspiring to deprive his or her friends.'* (Ibid.)

And this is what lead to the kernel code that runs your Android phone, the GNU/Linux operating system that runs most servers in the cloud (even on Azure!), and the compiler code you'll use to create firmware to run on the ESP32 in this course....



# Why does this matter? (1)



If...

- you want to build quickly you need to stand on the shoulders of giants
- you want to build well, you want building blocks that have been tested, tested, and tested again
- you want to make money from software, you need to be popular! sharing code is often a prerequisite
- there's no code, there's no code audit (and there's no security without code audit...)



# Why does this matter (2): closed = insecure



Anything that isn't open source must now be assumed to be compromised by the state, by definition. Everything else (Microsoft Windows, Apple iOS, Google's Android services...) is subject to this scenario:

Man (almost always) in dark suit knocks on door of software corp. Sincere and nice person answers. "Yes?"

"Sign here: I promise to open all my data to The Secret Policeman whenever he so desires or spend the rest of my natural life behind bars. And I promise not to tell anyone."

And off they go, hand in hand (willing or otherwise) to stuff a bunch of back door tricks into the products of said software corp so that Mr. Pervy Spy and his mates can get their fill of other people's naughtiness in service of whatever monstrous threat they claim to be saving us from this week.

(Yes: there *are* monstrous threats out there; treating every citizen as a criminal doesn't make them any easier to avoid!)





# Open source software

## Open source:

- if you can't modify it, you don't own it
- if you can't see the source, you're insecure
- Copyleft licences protect user rights (free use on condition of preserving freedom)

## Open source and the IoT:

- first IoT device, the CMU coke machine, could only be built because they had the source code of finger...





# Open hardware



Open hardware; how could that involve making a living?!

- become the authority on your niche
- sell services
- Arduino: not the fastest, not the cheapest, but the best at sharing and supporting!

# Computing for the IoT (1): constraints



Embedded computation has to

- fit the parent device size
- fit the cost envelope

This means that hardware constraints dominate:

- power — frequently battery driven, perhaps off-grid
- compute — what is the smallest microcontroller that can do the job?
- memory — can we fit the binary into a tiny space?
- connectivity — how will we talk to the net?



# Computing for the IoT (2): SoC or MCU?



In the stone age days of computers a mainframe's CPU occupied multiple cabinets. Later... one or a few PCBs. The next step... a CPU on a single chip... a **microprocessor** (uP) [Wouter van Ooijen](#); [then:]

- the CPU-on-a-chip is made more powerful... a cache is added, more CPU's (cores) are combined in one chip, etc. This results in the (mainly Intel) super-microprocessors of today's desktops and laptops.
- CPU is combined with memory and I/O on the same chip, creating a complete computer on a single chip. This is called a **microcontroller (MCU)**
- a moderately powerful CPU.. is combined with a small boot ROM and a set of complex peripherals, like a video/lcd subsystem, mpeg decoder, wired or wireless ethernet interface, USB interfaces, etc. to create a 'system-on-a-chip' [**SoC**]... (Beaglebone, Raspberry Pi, etc.)



# Computing for the IoT (3): connected MCUs



The core component for building IoT devices is a net-connected MCU

What are the options?

Many! [www.postscapes.com](http://www.postscapes.com) lists 167!

Criteria for choosing:

- fit to constraints as above
- ecosystem: code, forum posts, code issues and solutions, development contributors, stackexchange questions, ....

I.e. *community*... and the biggest embedded computation community is around a device called the *Arduino*





# The Arduino



Multiple personalities:

- a family of microcontroller-based PCBs
- a company making those PCBs
- an IDE for programming the microcontrollers

And: a massive community of open source developers contributing to the ecosystem of code and documentation and forum posts and github repos and etc...

See also: [Arduino: the Documentary](#)



# Arduino History



- arts and design students in Ivrea, Italy, early 2000s
- using microcontrollers to create interactive exhibits
- Arduino developed hardware cheaper than the alternatives (based on Atmel's AVR chips)
- added an IDE derived from work on *Processing* and *Wiring*
- some millions (or possibly 10s of millions) of official and unofficial boards now in existence
- the most popular platform for embedded electronics experimentation

# The IDE, libraries, example code



The Arduino IDE is a Java Swing desktop app.

It runs the GNU C++ toolchain (**gcc**) to convert the Arduino C dialect into executable binaries.

These are then uploaded (“burned”) to the board using various other tools (in our case a Python tool called esptool —

<https://github.com/espressif/esptool>).

The IDE then allows monitoring of serial comms from the board, and provides access to libraries and example code.

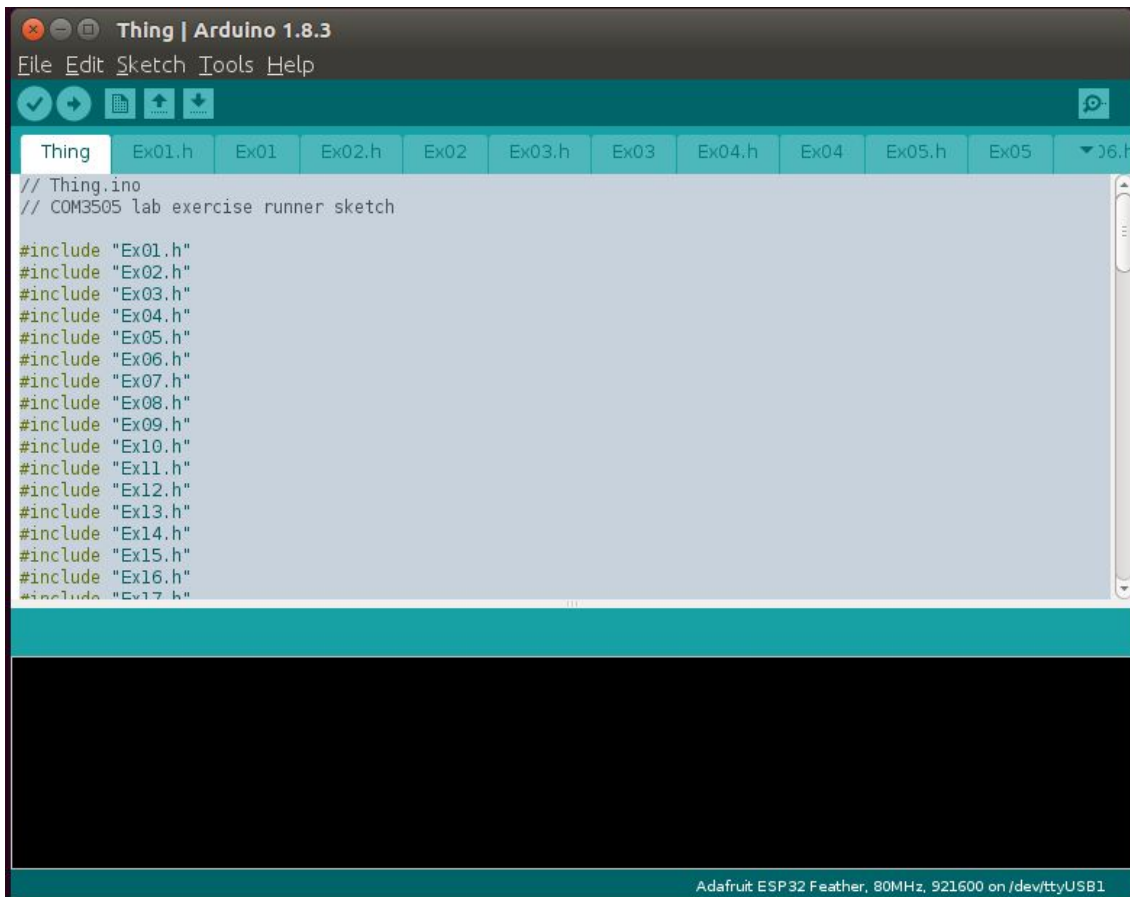
The IDE is relatively basic, without code completion or sophisticated error handling. You may want to use an external editor, or experiment with alternative IDEs. For COM3505 we can help best when you’re using the Arduino IDE itself.

Another option:

- ESP IDF: Espressif IoT Development Framework
- <https://github.com/espressif/esp-idf>
- deploys libraries that the Arduino IDE APIs are partly layered on
- more mature in some respects?
- harder to use? (though can use Eclipse)
- **not** supported by this course, though you’re welcome to try it!



# Editing a “Sketch”



The screenshot shows the Arduino IDE interface. The title bar reads "Thing | Arduino 1.8.3". The menu bar includes "File", "Edit", "Sketch", "Tools", and "Help". The toolbar contains icons for checking, running, uploading, and downloading. The tab bar shows "Thing" as the active tab, followed by "Ex01.h", "Ex01", "Ex02.h", "Ex02", "Ex03.h", "Ex03", "Ex04.h", "Ex04", "Ex05.h", "Ex05", and "Ex06.h". The main text area contains the following code:

```
// Thing.ino
// COM3505 lab exercise runner sketch

#include "Ex01.h"
#include "Ex02.h"
#include "Ex03.h"
#include "Ex04.h"
#include "Ex05.h"
#include "Ex06.h"
#include "Ex07.h"
#include "Ex08.h"
#include "Ex09.h"
#include "Ex10.h"
#include "Ex11.h"
#include "Ex12.h"
#include "Ex13.h"
#include "Ex14.h"
#include "Ex15.h"
#include "Ex16.h"
#include "Ex17.h"
```

The status bar at the bottom indicates the hardware: "Adafruit ESP32 Feather, 80MHz, 921600 on /dev/ttyUSB1".

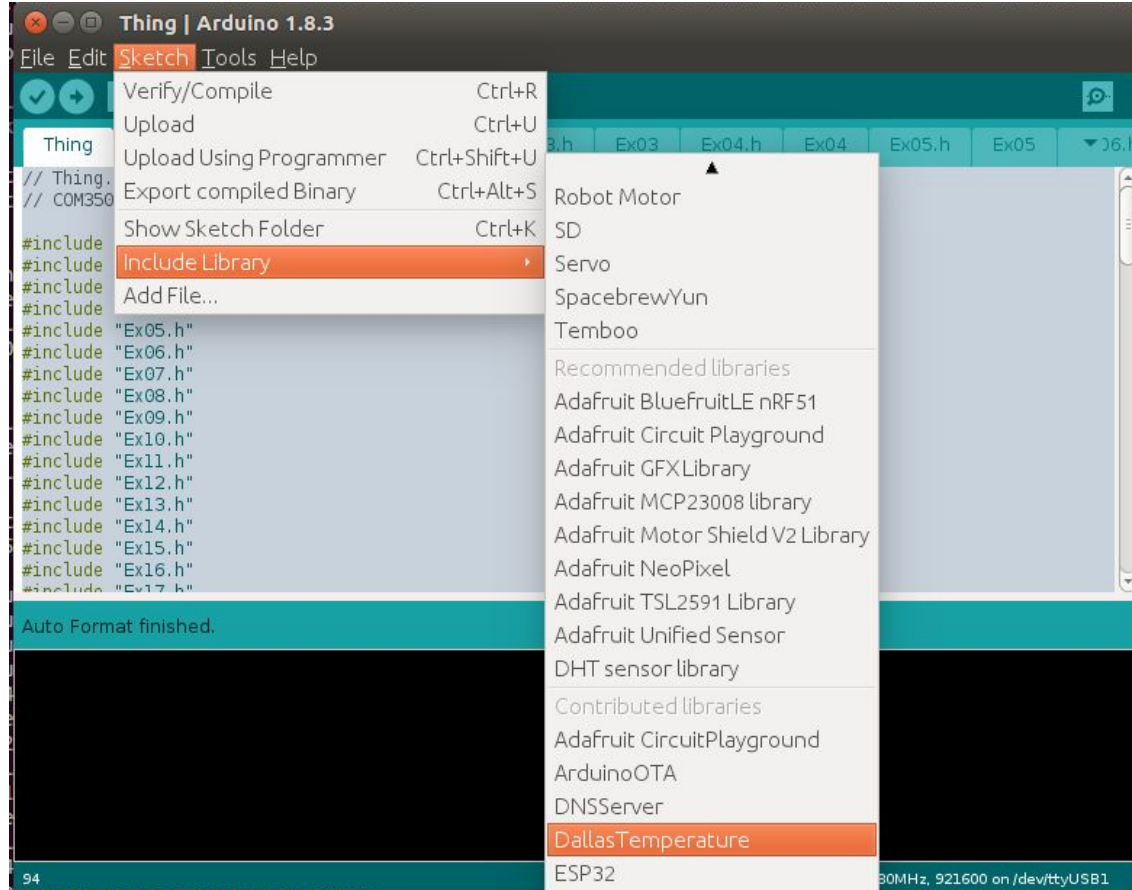




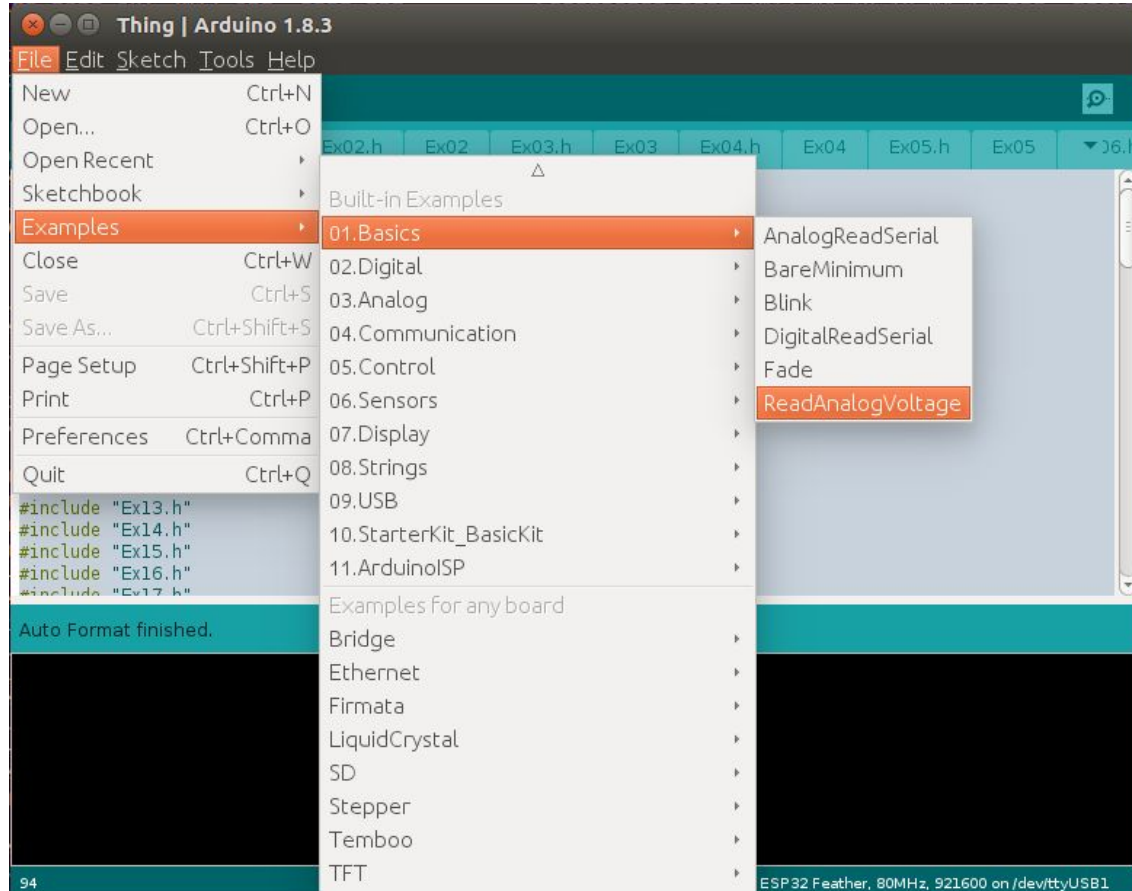
# Including a Library



But: most or all  
of the libraries  
we will be using  
*should* already  
be in your  
filesystem....



# Example Code



# C[++] on the Arduino Platform



Arduino C (or C++) is C++ with lots of added libraries and a bit of code rewriting.

It uses GCC to create binaries to burn to compatible devices (including the ESP32).

The IDE includes a compilation handler that converts your .ino files into a conventional .cpp file (poke around in /tmp or ~/.arduino15 to see).

The binaries created then:

- call setup once at boot
- call loop repeatedly

For example (from

<https://www.arduino.cc/en/Tutorial/HelloWorld>):

```
// include the library code:
#include <LiquidCrystal.h>

// initialize the library by associating any needed LCD interface pin
// with the arduino pin number it is connected to
const int rs = 12, en = 11, d4 = 5, d5 = 4, d6 = 3, d7 = 2;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);

void setup() {
  // set up the LCD's number of columns and rows:
  lcd.begin(16, 2);
  // Print a message to the LCD.
  lcd.print("hello, world!");
}

void loop() {
  // set the cursor to column 0, line 1
  // (note: line 1 is the second row, since counting begins with 0):
  lcd.setCursor(0, 1);
  // print the number of seconds since reset:
  lcd.print(millis() / 1000);
}
```

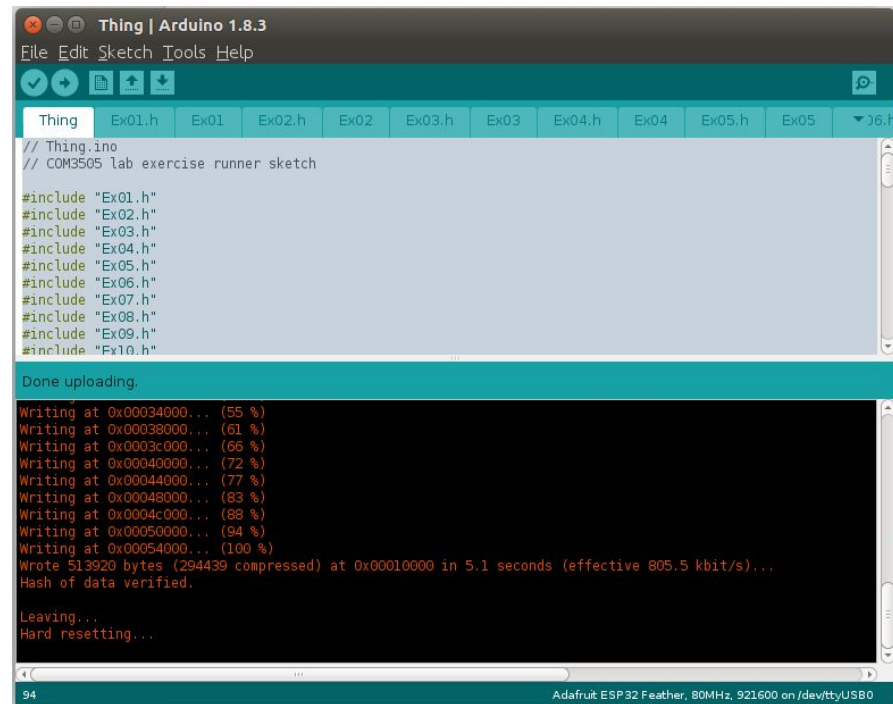


# Code, burn, test, repeat...



The development cycle goes like this:

- create a “sketch”: a .ino file in a directory that shares its name, e.g. Thing/Thing.ino
- optionally, other .ino and/or .h files
- compile: Cntrl&R (or the tick on the menu bar)
- burn to device: Cntrl&U (or the right arrow on the menu bar)
- monitor on serial: Cntrl&Shift&M
- test
- repeat



The screenshot shows the Arduino IDE window titled "Thing | Arduino 1.8.3". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu bar is a toolbar with icons for checking, uploading, and monitoring. The main editor area shows the "Thing.ino" file with the following code:

```
// Thing.ino
// COM3505 lab exercise runner sketch

#include "Ex01.h"
#include "Ex02.h"
#include "Ex03.h"
#include "Ex04.h"
#include "Ex05.h"
#include "Ex06.h"
#include "Ex07.h"
#include "Ex08.h"
#include "Ex09.h"
#include "Ex10.h"
```

Below the code editor, a status bar indicates "Done uploading." and a progress bar shows the upload progress. The serial monitor window at the bottom displays the following output:

```
Writing at 0x00034000... (55 %)
Writing at 0x00038000... (61 %)
Writing at 0x0003c000... (66 %)
Writing at 0x00040000... (72 %)
Writing at 0x00044000... (77 %)
Writing at 0x00048000... (83 %)
Writing at 0x0004c000... (88 %)
Writing at 0x00050000... (94 %)
Writing at 0x00054000... (100 %)
Wrote 513920 bytes (294439 compressed) at 0x00010000 in 5.1 seconds (effective 805.5 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting...
```

The status bar at the bottom of the IDE window shows "94" and "Adafruit ESP32 Feather, 80MHz, 921600 on /dev/ttyUSB0".



# Labs: soldering, sig gen, measurement



EE lab, DIA 2.02, Friday 9-10.50:

- soldering
- breadboarding
- generating (an AC) signal
- measuring e.g. voltage
- other week 1 exercises

EE lab Monday 9-10.50:

- solder the headers on your ESP
- coding the week 2 exercises

Why learn to prototype circuits?

- the more we can build locally, the more resilient we are
- there's a new raft of opportunity around modern embedded computation
- a little knowledge goes a long way

Why learn to solder?

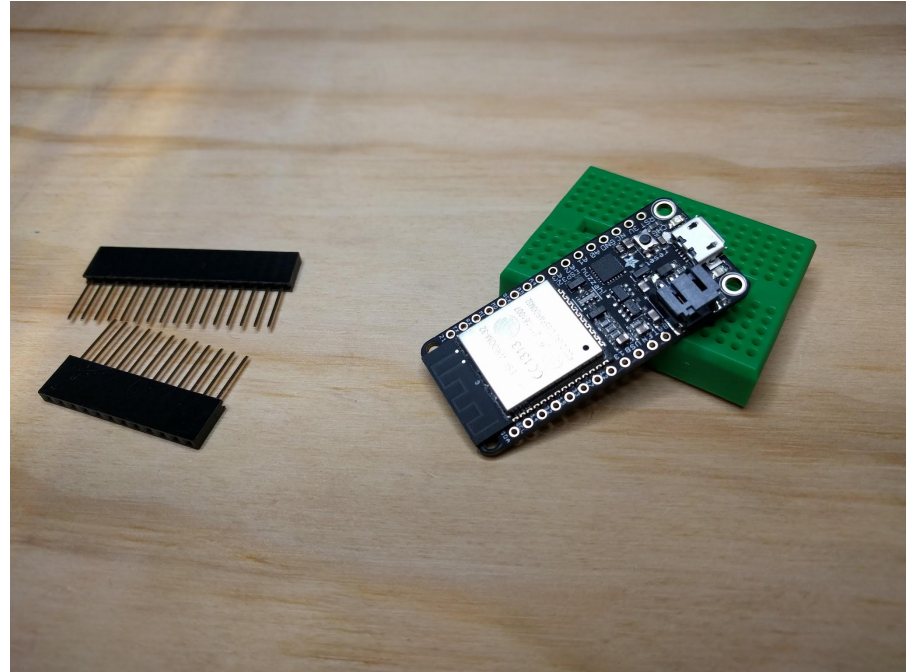
- more robust than breadboards
- promotes manual dexterity
- exercise another part of your brain! (avoid Alzheimers!)



# Soldering the headers on the ESP32



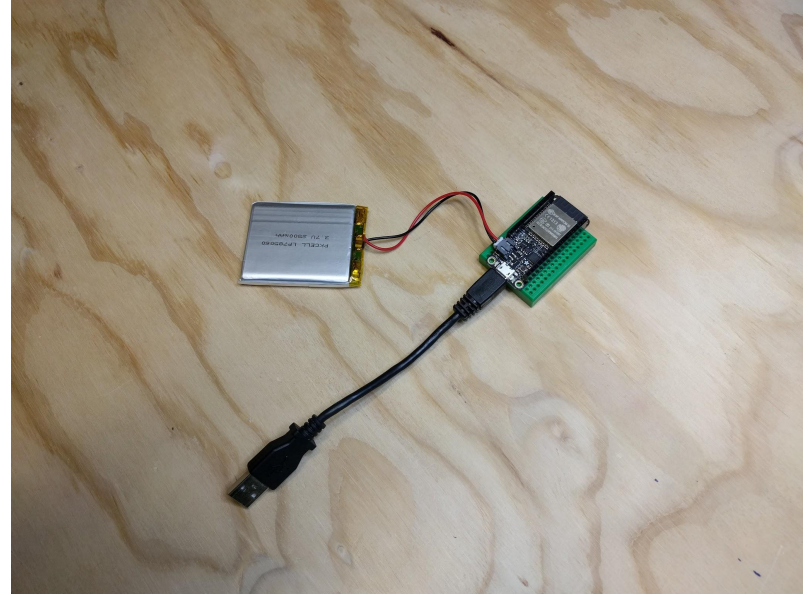
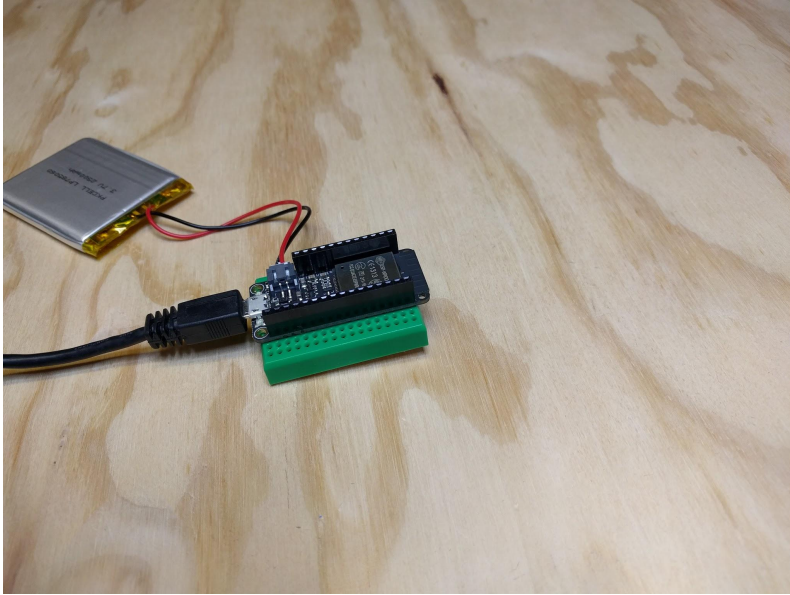
- your ESP32 Feather comes without headers attached
- we need headers that we can fit to a breadboard, for example
- the board is supplied with low-profile headers
- we've given you stacking headers so we can attach other Feather boards on top later on
- on Monday in the lab you will solder the stacking headers onto the ESP



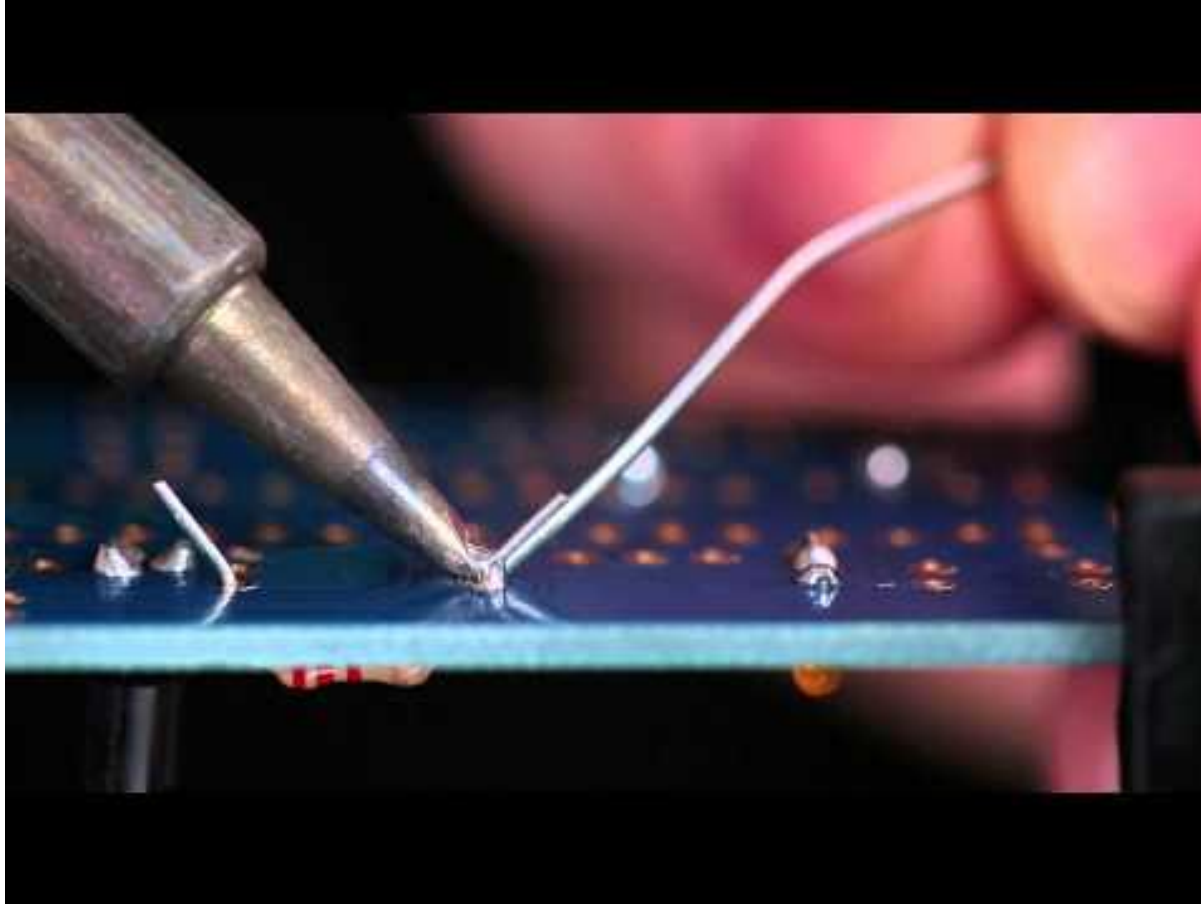




# Headers (2)



# Soldering basics



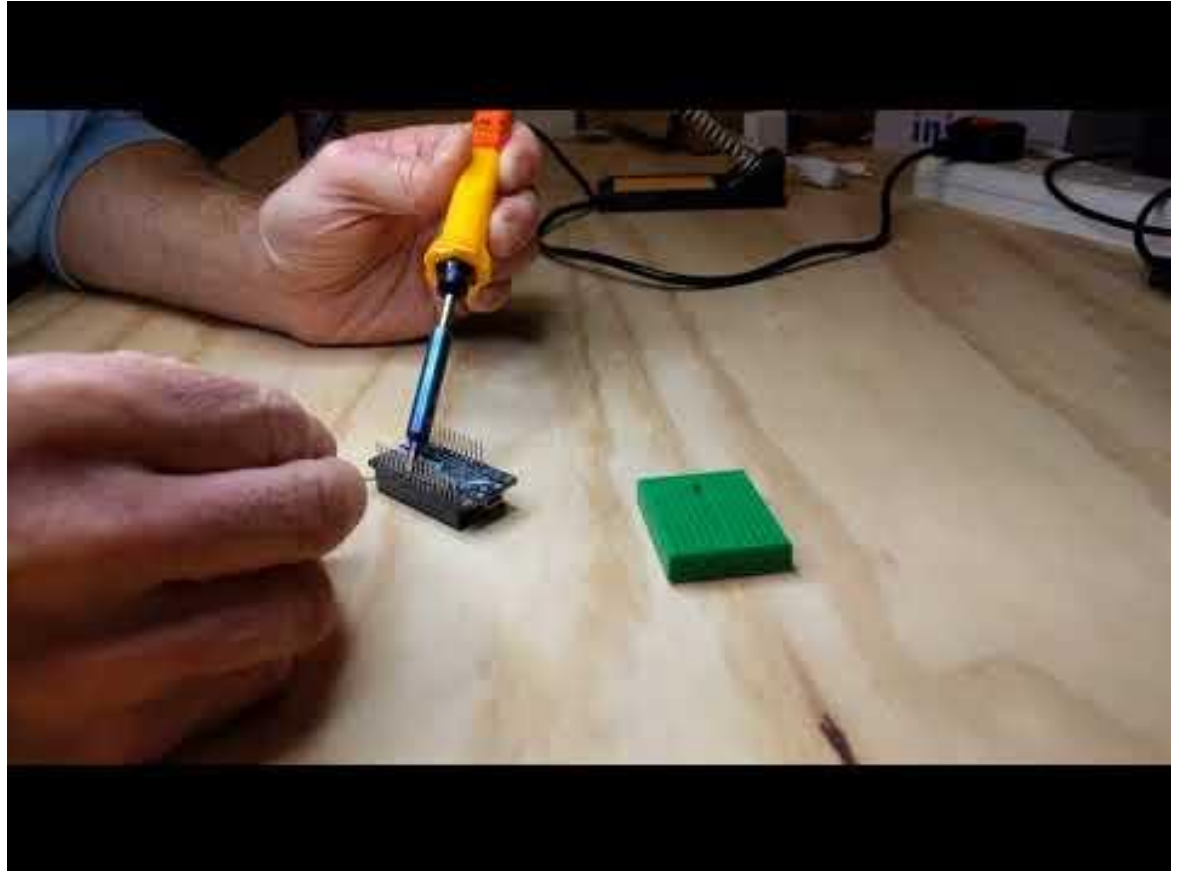


# Soldering the Feather headers



## Gotchas:

- board wrong way up
- dry joints (check conductivity)
- headers not seated properly (need to be at exactly 90°)
- leaving solder on the pins (5:00, 7:00)
- bending the pins when breadboarding





# Your **TODO** list for week 2:

- In the EE lab (DIA 2.02: Friday 9am; Monday 9am)
  - work in pairs on Friday
  - **practice soldering** (so you're prepared to do ESP headers Monday)
  - experiment with signal generation and measurement
  - coding as below
- Update your git repository clone
- Read and digest Notes/Week02.mkd
  - blink the ESP32 LED; print to serial
  - (optional) investigate String and provide example code
  - **DON'T** edit code in the Thing directory — use the MyThing tree
- Do the reading
- Make sure you understood the lecture, and review the slides if needed:

[tinyurl.com/com3505I2](https://tinyurl.com/com3505I2)

