The University Of Sheffield

Small but perfectly formed... digging into the ESP32

COM3505, Lecture 3

Prof Hamish Cunningham

Recap (1): the story so far...

- created **github repo**: pull weekly updates; push code assignments
- sent data to the **cloud server** (com3505.gate.ac.uk)
- set up the **Arduino IDE** and started writing C++ Sketches
- learned about **LiPo hazards**
- coded: inbuilt-**LED blink**; print **MAC on serial**, monitor in IDE;
- (optionally) investigated **memory fragmentation** and the String class

LTSP **Linux issues**:

- /dev/ttyUSB… not writeable
- login fails; X window manager doesn't appear

Apparently one or both are symptoms of a full local home… (Ruby gems? Arduino unpacked to the wrong location?)

Solution: move stuff to u-drive

Next up in the labs: **week 3** notes **now published**
connecting and programming external devices; debugging; time slicing

Recap (2): IoT probably implies a WiFi MCU

Constraints of power and cost mean the IoT is in the MCU territory.

Speed of development and ease of use mean the easiest option at present is WiFi (or sometimes Bluetooth).

Until recently devices would frequently use both an MCU (e.g. Atmel) with a separate WiFi chip (e.g. Mediatek).

The ESP8266 innovated in combining the two in a powerful but cheap package.

The ESP32 adds extra goodies inc. another core, BLE, more I/O.

If you're **not constrained** by power or cost, then a single-board computer (SBC) like the Raspberry Pi makes sense.

If you've serious processing to do, then your IoT device will pass its data to the cloud and crunch it there (perhaps via a *gateway*).

COM3505: developing with the ESP32 and connecting to the net (first half); ESP32-based applications (second half); some cloud-side on the way.

ESP32 vs. other hardware

- cheap! ($2.50/unit for the chip, $4.50/unit for a bare module, $20 for a posh development board)
- good compromise on power vs. cost
- flexible: doesn't tie you into a proprietary vision of the IoT (or anything else!)
- WiFi, Bluetooth Low Energy: ubiquitous and easy (though not IoT specific: WiFi is power hungry for example)
- open source SDK and Arduino core on GitHub; closed hardware
- layers on top of Arduino APIs and therefore a very large codebase of sensor / actuator libraries
- vibrant and extensive contributor community (see GitHub issues and commits, ESP8266 and ESP32 forums, Hackaday projects, Kickstarters…)

Tensilica Xtensa vs. ARM, MIPS, x86...

- Xtensa: generate everything from server CPU to tiny MCU from one core definition
- customisable ISA (instruction set architecture)
- this means less optimisation but more generality, potentially lower cost, better support for ASICs (application-specific integrated circuits)
- ARM: dominant architecture by volume
- MIPS: original RISC, still significant
- what about Intel?! Still chasing the MCU market...
  - ATOM-based: Quark, Joule, Edison: discontinued, summer 2017!
  - still available, the Curie… x86 ISA!
  - Arduino 101: based on the Curie; but note Adafruit's comment: "The Arduino 101 is very new, and uses a completely new core chipset! Many libraries won't work with it"...

A WiFi MCU for 2018, 2019…?

Who knows!

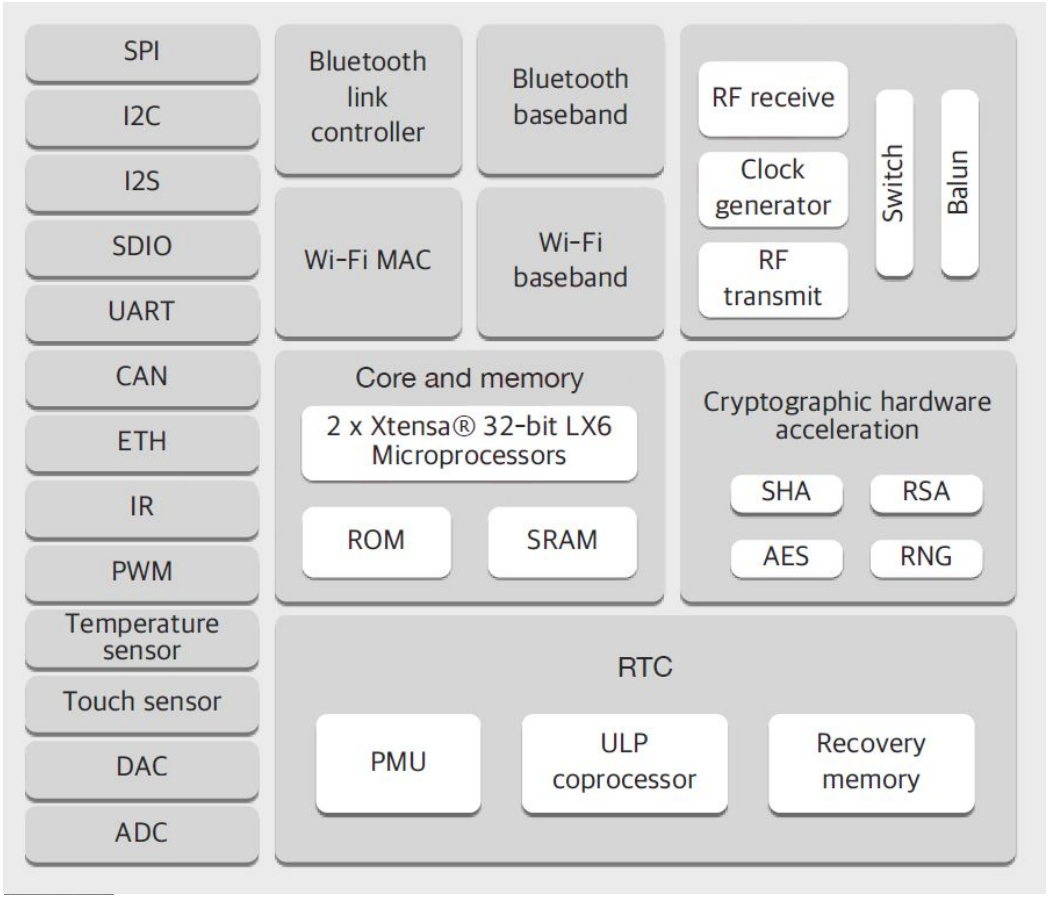
ESP SoCs are not the only game in town. Arduino LLC's MKR1000 board uses an Atmel ATSAMW25H18 for WiFi. Pine64, known for its Pine-A64 Linux hacker board, recently jumped in with PADI IoT Stamp COM based on Realtek's RTL8710AF, a WiFi-enabled ESP8266 competitor that runs FreeRTOS on a Cortex-M3. Like ESP-based modules, the PADI IoT Stamp is tiny (24x16mm), power efficient, and dirt cheap, selling for only $2 at volume.

Intel's second generation, MCU-like Quark D2000 and Quark SE processors are also aimed at IoT nodes, in this case running lightweight RTOSes such as Zephyr. The Quark SE drives Intel's BLE-enabled Curie modules for wearables, as well as Intel's Arduino 101 board.
https://www.linux.com/news/linux-and-open-source-hardware-iot

Why there's no top-ten list of IoT processors:
http://www.eetimes.com/author.asp?section_id=36&doc_id=1329732
(market fragmentation, mergers and aquisitions, definitional confusion, ...)

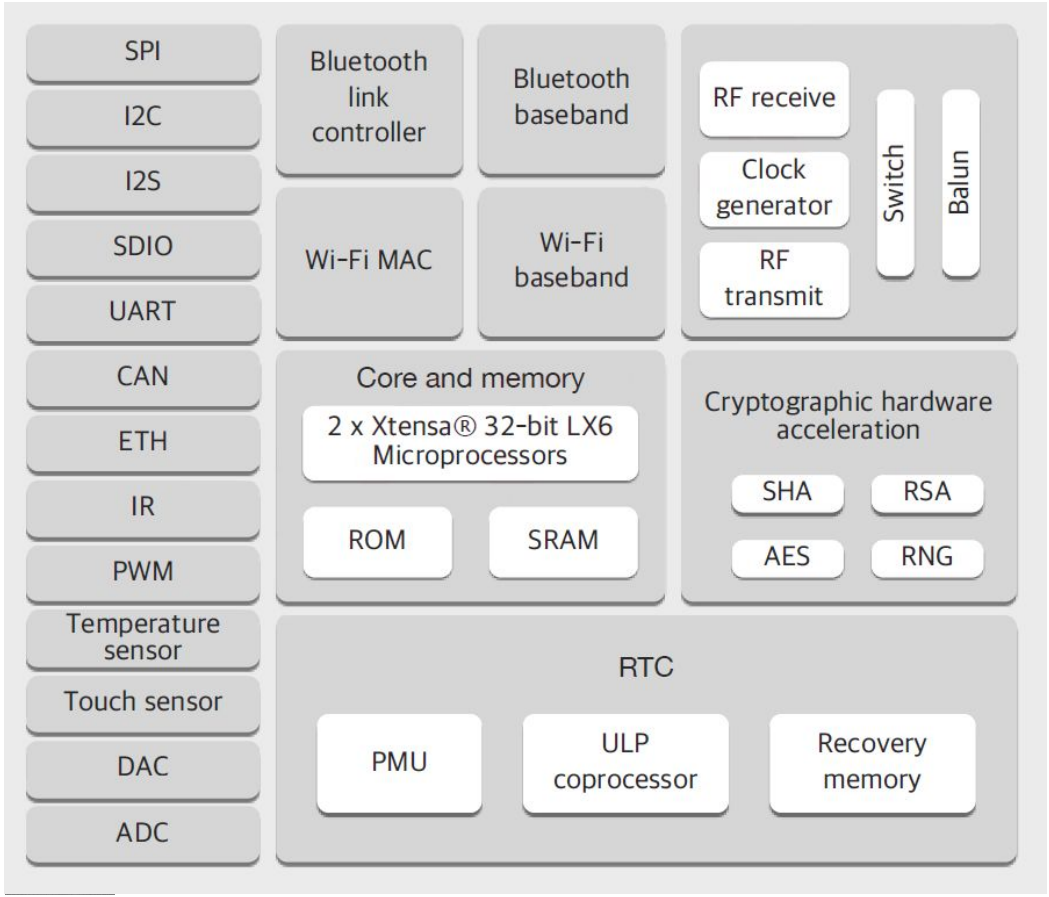
Which is another reason to use Arduino C++ and the Arduino IDE...

ESP32: Main Elements (1: core)



- 240 MHz dual core Tensilica LX6 microcontroller with 600 DMIPS
- integrated 520 KB SRAM
- integrated 802.11 b/g/n Wi-Fi transceiver, baseband, stack and LWIP
- Bluetooth (classic and BLE)
- up to 16 MB memory-mapped flash
- 2.3V to 3.6V operating voltage
- -40°C to +125°C operating temperature
- on-board PCB antenna / IPEX connector for external antenna

ESP32: Main Elements (2: GPIO)



34 GPIO pins assignable to:

- Analog-to-Digital Converter (ADC)
- Digital-to-Analog Converter (DAC)
- Pulse-Width Modulation (PWM) – dimming LEDs or running motors
- Touch Sensor – ten capacitive sensing pins
- UARTs
- I2C, SPI, I2S – two I2C and three SPI interfaces to hook up all sorts of sensors and peripherals, plus two I2S sound interfaces

(more details of these in L4 next week)

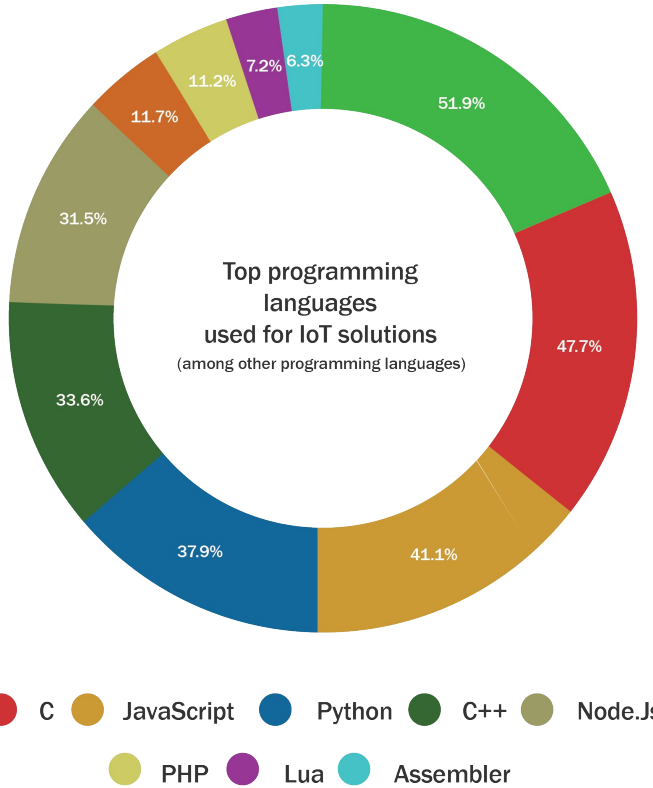
ESP32: light as a feather

What else do we need to learn? Which programming language(s) for IoT? (1)

According to one survey:

- between them C and C++ are used by >80% of IoT programmers
- Node.js plus Javascript are also a good bet, used by 70%.
- Java: >50%
- Python: nearly 40%

COM3505: both C and C++ (as implemented in the Arduino compiler toolchain)

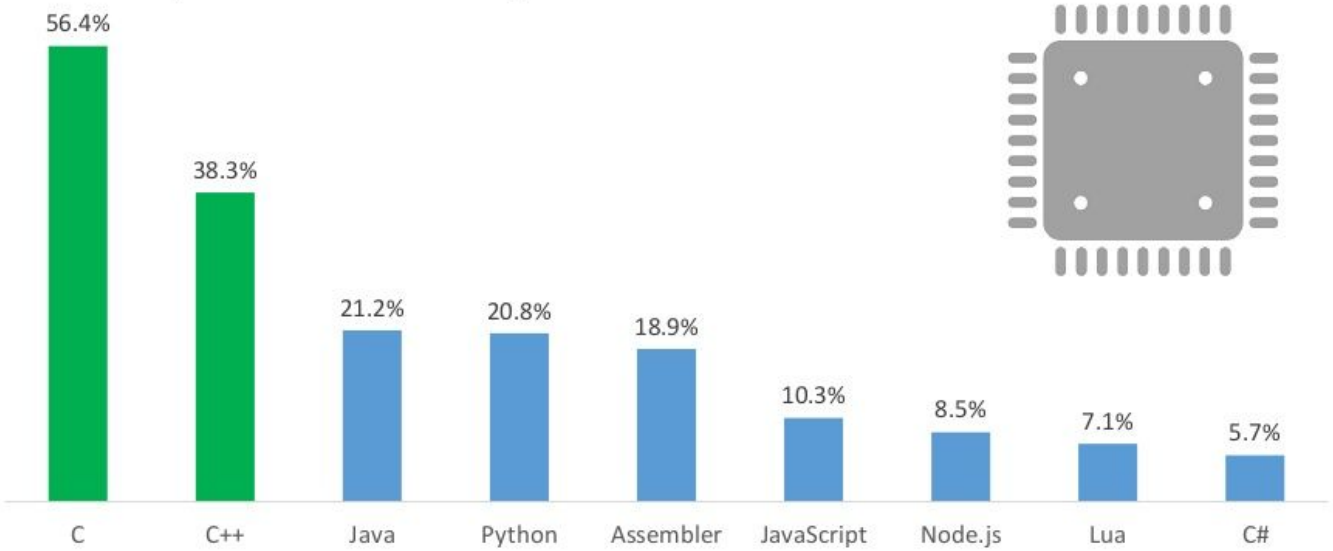


(*Statistics - Eclipse IoT Working Group. IEEE IoT & AGILE IoT)

Which programming language(s)? (2)

PROGRAMMING LANGUAGES – CONSTRAINED DEVICES

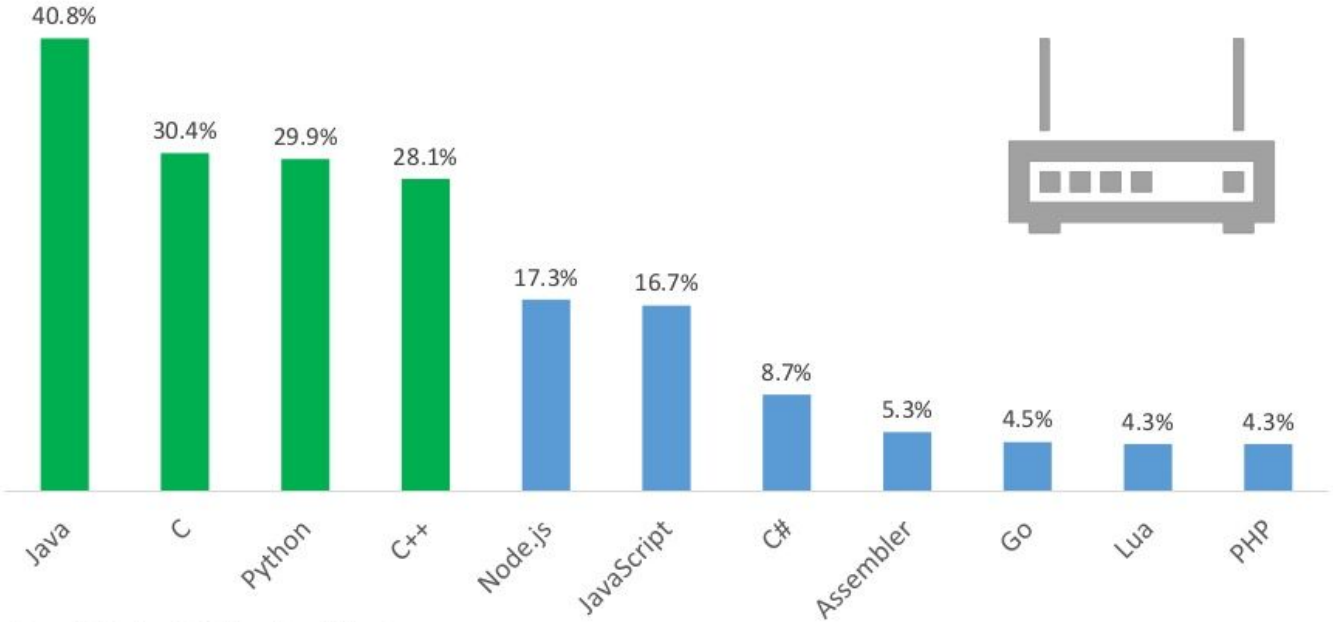
Which of the following programming languages, if any, do you use to build IoT solutions? (Constrained Devices)



Which programming language(s)? (3)

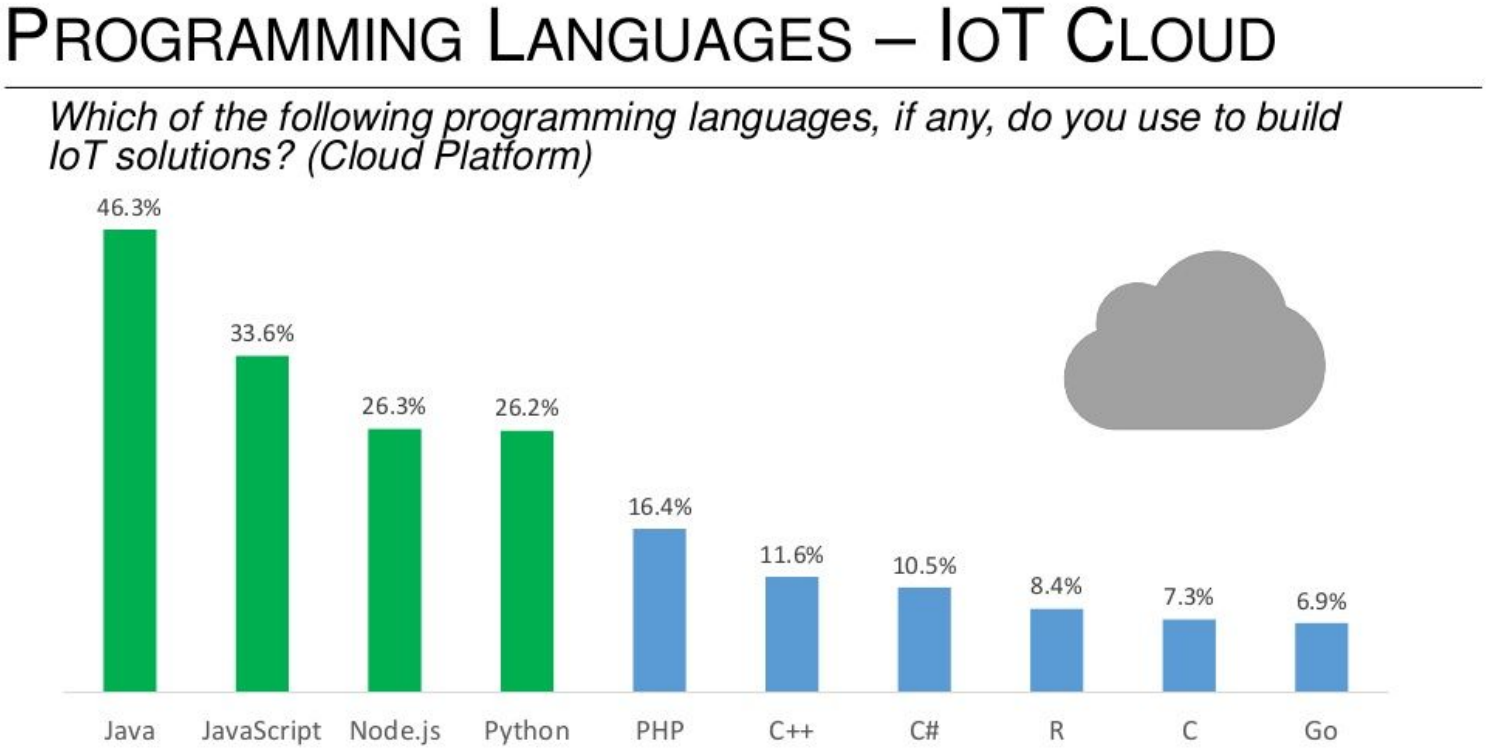
PROGRAMMING LANGUAGES – IOT GATEWAYS

Which of the following programming languages, if any, do you use to build IoT solutions? (Gateways)



IoT Developer Survey 2017 - Copyright Eclipse Foundation, Inc.

Which programming language(s)? (4)

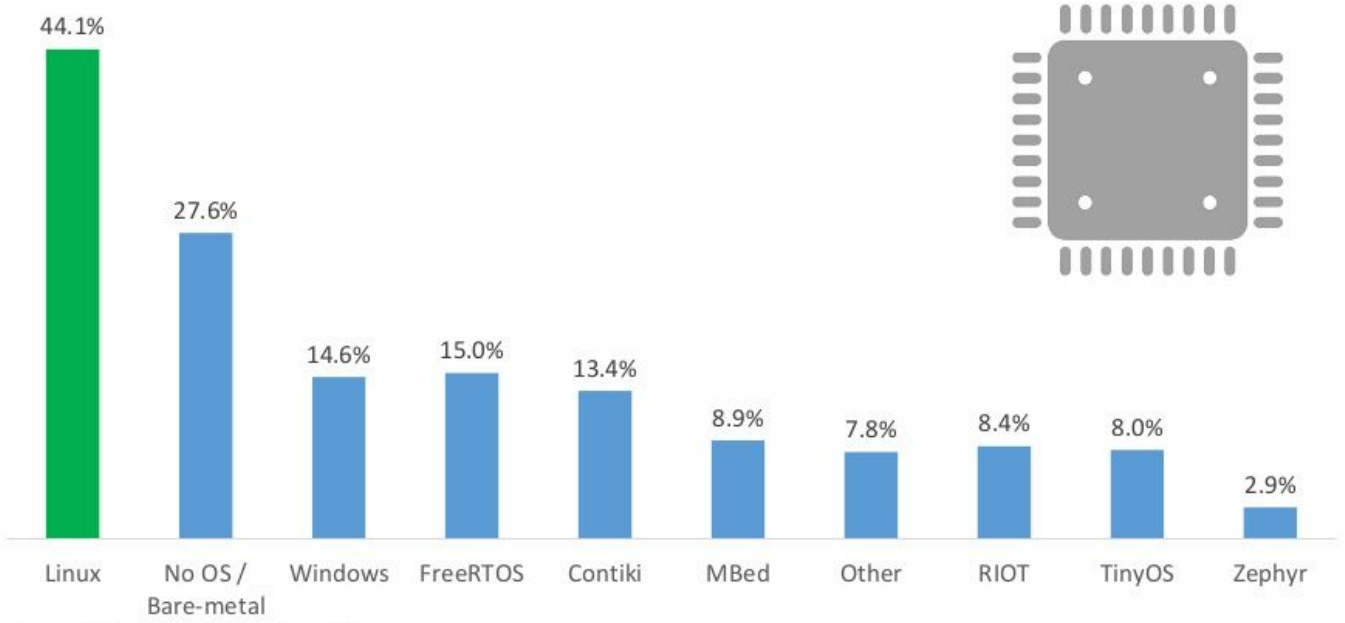


IoT Developer Survey 2017 - Copyright Eclipse Foundation, Inc.

Which programming language(s)? (5: the OS)

IoT Operating Systems – Constrained Devices

Which operating system(s) do you use for your IoT devices? (Devices)



Appropriate tech: less waste!

IoT: small cheap local... Low power!

Less cloud, more microcontrollers!

This is also, in theory, a potential benefit of *fog computing* (aka *edge computing*), which implies:

- reduced data migration to the cloud
- increased reliance on edge device computation
- resilience and (partial) autonomy in face of network or cloud outages

Theoretically, IoT will make business more efficient, a claim that is backed up by some early anecdotal evidence. Farmers are more closely monitoring crops with the help of sensor networks to ensure a better yield, and factory owners are monitoring operations to spot maintenance issues without requiring costly shutdowns. Electric utilities have been early IoT users, using sensors to monitor equipment and help customers reduce energy bills. If and when we start taking energy consumption and climate change seriously, IoT will be one of the key tools for documenting the problem and helping to solve it.

https://www.linux.com/news/who-needs-internet-things

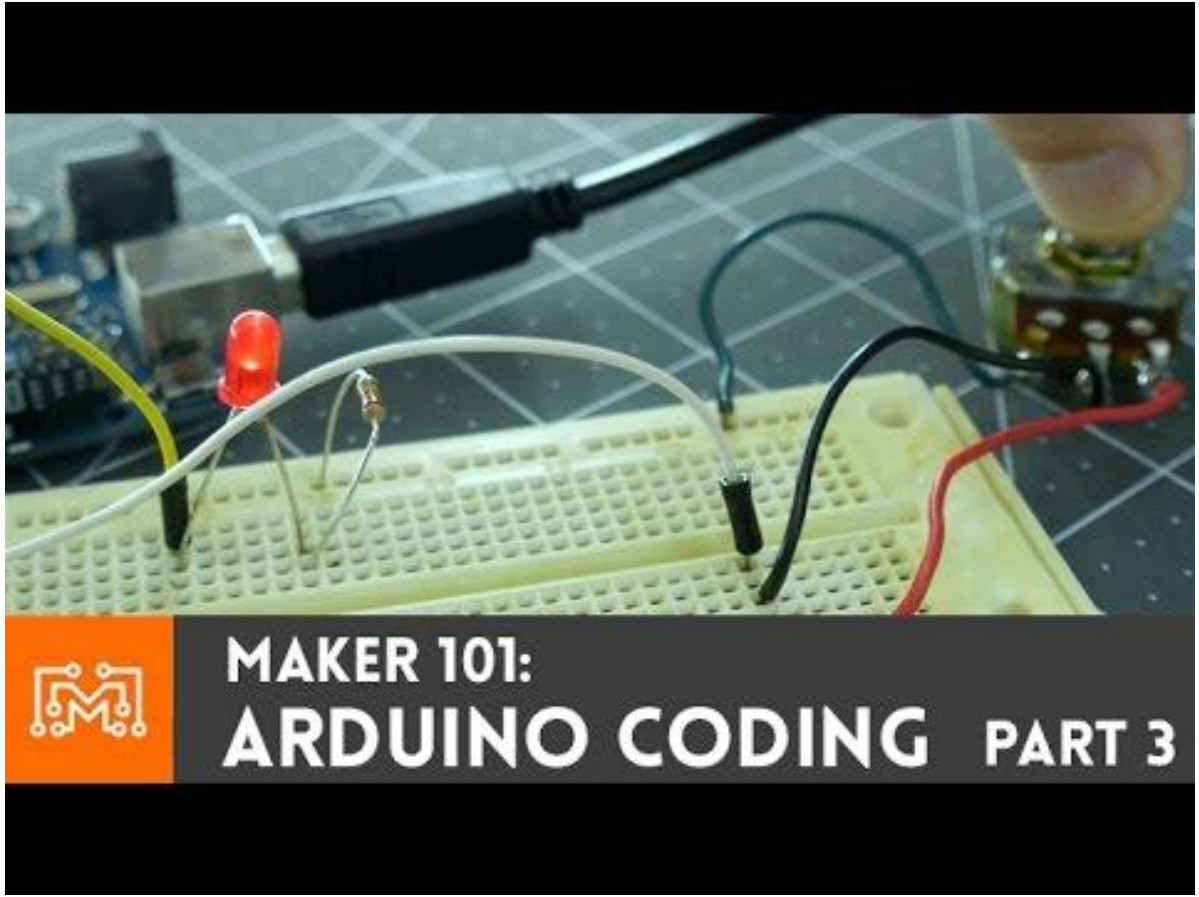
Peer-to-peer vs. cloud: liberation from the advertising surveillance industry! :-)

Enough chat already, let's code! (1)

(This example uses an Arduino board, **not** an ESP.)

Enough chat already, let's code! (2)

(This example uses an Arduino board, **not** an ESP.)



Need a slower presenter?! Try: https://youtu.be/d8_xXNcGYgo

Thing.ino, Ex01.h, Ex01.ino ...

- You now have Thing/Thing.ino, Ex01.h and Ex01.ino in your repo
- [**DON'T edit** these files; feel free to copy them, but do any edits or new code in MyThing/…]
- These are model solutions to last week's exercises, plus convenience code to choose and run multiple different exercises
  - set LABNUM to select Ex num

```
// Thing.ino
// DON'T edit these files, do your coding in MyThing!
#include "Ex01.h"
#include "Ex02.h"
...
int LABNUM = 2; // which lab exercise number are we doing?
void setup() { // initialisation entry point
  switch(LABNUM) {
    case  1: setup01(); break;
    case  2: setup02(); break;
...
    default: Serial.println("oops! invalid lab number");
  }
}
void loop() { // task loop entry point
  switch(LABNUM) {
    case  1: loop01(); break;
    case  2: loop02(); break;
...
    default: Serial.println("oops! invalid lab number");
  }
}
```

ExN.h, ExN.ino ...

```
// Thing.ino
// DON'T edit these files, do your coding in MyThing!
#include "Ex01.h"
#include "Ex02.h"
...
int LABNUM = 2; // which lab exercise number are we doing?
void setup() { // initialisation entry point
  switch(LABNUM) {
    case  1: setup01(); break;
    case  2: setup02(); break;
...
    default: Serial.println("oops! invalid lab number");
  }
}
void loop() { // task loop entry point
  switch(LABNUM) {
    case  1: loop01(); break;
    case  2: loop02(); break;
...
    default: Serial.println("oops! invalid lab number");
  }
}
```

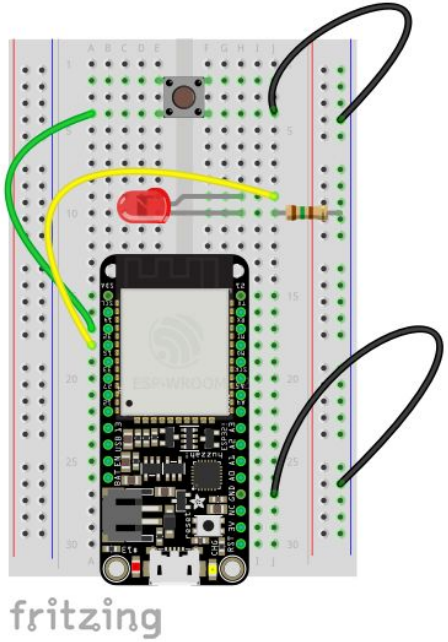
This is how I did it… that doesn't (necessarily) make it "right"!

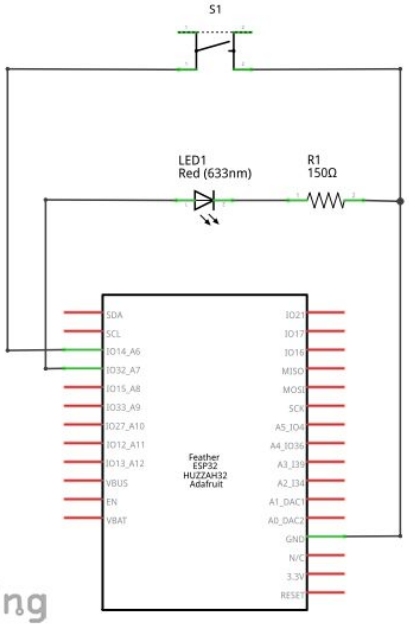
```
// Ex01.h
#ifndef LAB_01_H
#define LAB_01_H
char MAC_ADDRESS[13]; // MACs are 12 chars, plus the NULL terminator
void getMAC(char *);
void ledOn(); void ledOff(); void blink(int = 1, int = 300);
#endif

// Ex01.ino
void setup01() {
  Serial.begin(115200);           // initialise the serial line
  getMAC(MAC_ADDRESS);            // store the MAC address as chip id
  pinMode(BUILTIN_LED, OUTPUT);   // set up GPIO pin for built-in LED
  Serial.println("\nsetup01..."); // say hi
}
void loop01() {
  Serial.printf("ESP32 MAC = %s\n", MAC_ADDRESS); // print ESP's "ID"
  blink(3);   delay(1000);        // blink the on-board LED and pause
}
void getMAC(char *buf) { // MAC is 6 bytes so needs careful conversion
  uint64_t mac = ESP.getEfuseMac(); // ...to string (high 2, low 4):
  sprintf(buf, "%04X%08X", (uint16_t) (mac >> 32), (uint32_t) mac);
}
void ledOn()  { digitalWrite(BUILTIN_LED, HIGH); }
void ledOff() { digitalWrite(BUILTIN_LED, LOW); }
void blink(int times, int pause) {
  ledOff();
  for(int i=0; i<times; i++) {
    ledOn(); delay(pause); ledOff(); delay(pause);
  }
}
```

Exercises 2—5

Based on a new circuit that adds and external LED and switch to your ESP (build instructions are in Notes/ in your repo):





Exercises 2—5

Using the breadboard, code and test the following:

- **Ex02**: blink the external LED; read from the switch
- **Ex03** (optional):
  - add more two more LEDs to your board see here
  - run as traffic lights, triggered by the switch
- **Ex04** (optional): debugging infrastructure: experiment with macros to allow adding flexible debug code
- Exercise 5 (**Ex05**) is ...

Threads? Processes? Ha!

- The ESP is a small device — so in order to perform multiple tasks at once we need to adopt a multiprocessing model of one sort of another.
- Threads and processes are relatively heavyweight, and interrupt-driven callbacks relatively immature (on the ESP devices).
- A simpler model: time slicing in the main loop, using...
  - a loop counter (e.g. an int; it will overrun but we don't care)
  - a set of iteration points at which different tasks run, e.g.:
    - if(loopCounter == TICK_MONITOR) { // monitor levels, step valves, push data
  - and/or a slice size to modulo the loop counter:
    - if(loopIteration % sliceSize == 0) { // a slice every sliceSize iterations
- Exercise 5 (**Ex05**) is to explore models like this in your MyThing sketch. How many milliseconds does a action take? How does loop code relate to the ESP wifi stack? What other options are there?

Your **TODO** list for week 3:

- Update your git repository clone as usual
- Read and digest Notes/Week03.mkd
- More coding around LEDs, switches, debugging and time-slicing
- Do the reading
- Make sure you understood the lecture, review if needed: tinyurl.com/com3505l3
- **NOTE**: there are two lab sessions before the next lecture…
  - WED 1-1:50 Diamond 2.02 Week03 lab (part 1)
  - MON 9-10:50 Diamond 2.02 Week03 lab (part 2)