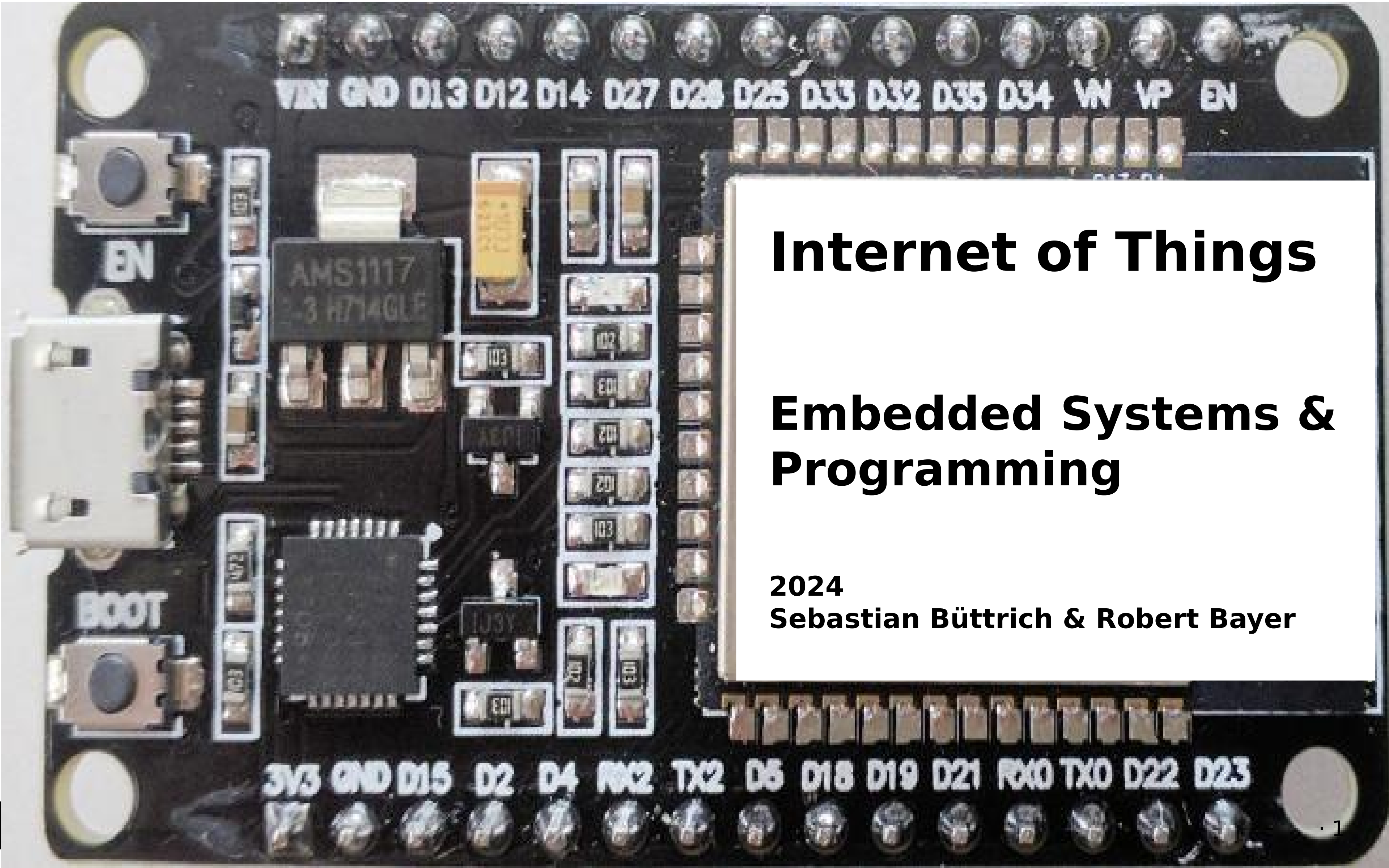


Internet of Things

Embedded Systems & Programming

2024
Sebastian Büttrich & Robert Bayer

Agenda

- Embedded system - term, definition, history
    - A look at what it is NOT: a full Operating System
- Constraints
- Elements of embedded hardware
- Examples of embedded systems
    - Arduino
    - FreeRTOS
- How to work with embedded systems

Embedded systems - definition

An embedded system is a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform <u>a specific function.</u>
A good example is the microwave oven. Almost every household has one, and tens of millions of them are used every day, but very few people realize that a processor and software are involved in the preparation of their lunch or dinner.

Barr, M., & Massa, A. (2006). Programming embedded systems: with C and GNU development tools. " O'Reilly Media, Inc.".

Embedded systems - definition

An embedded system is a computer system—a combination of a **computer processor, computer memory, and input/output peripheral devices**—that has a **dedicated function within a larger mechanical or electrical system**.[1][2] It is embedded as part of a complete device often including electrical or electronic hardware and mechanical parts. Because an embedded system typically controls **physical operations** of the machine that it is embedded within, it often has real-time computing **constraints.**

[Barr]

[Heath, S. (2002). Embedded systems design. Elsevier]

(Networked) Embedded System

”An embedded system is a **special-purpose** system in which the computer is completely **encapsulated** by the device it controls. Unlike a general-purpose computer, such as a personal computer, an embedded system performs pre-defined tasks, usually with very specific requirements. Since the system is dedicated to a **specific task**, design engineers can optimize it, reducing the size and cost of the product. Embedded systems are often mass-produced, so the cost savings may be multiplied by millions of items.”

source: NC State University, ECE

Embedded systems - definition

What is an embedded system?

There are many definitions for this but the best way to define it is to describe it in terms of what it is not and with examples of how it is used.

An embedded system is a microprocessor-based system that is built to control a function or range of functions and is not designed to be programmed by the end user in the same way that a PC is. Yes, a user can make choices concerning functionality but cannot change the functionality of the system by adding/replacing software. With a PC, this is exactly what a user can do: one minute the PC is a word processor and the next it's a games machine simply by changing the software. An embedded system is designed to perform one particular task albeit with choices and different options. The last point is important because it differentiates itself from the world of the PC where the end user does reprogram it whenever a different software package is bought and run. However, PCs have provided an easily accessible source of hardware and software for embedded systems and it should be no surprise that they form the basis of many embedded systems. To reflect this, a very detailed design example is included at the end of this book that uses a PC in this way to build a sophisticated data logging system for a race car.

Heath, S. (2002). Embedded systems design. Elsevier

Embedded systems – main distinction

A
specialized
&
hardware-specific function

Not a
general purpose computer

The embedded quality can take many forms – *embedded* in what? - and specifically in IoT actually disappear – **paradoxally, the embedded system in IoT tends to be stand-alone / autonomous**

Terms are not absolutes

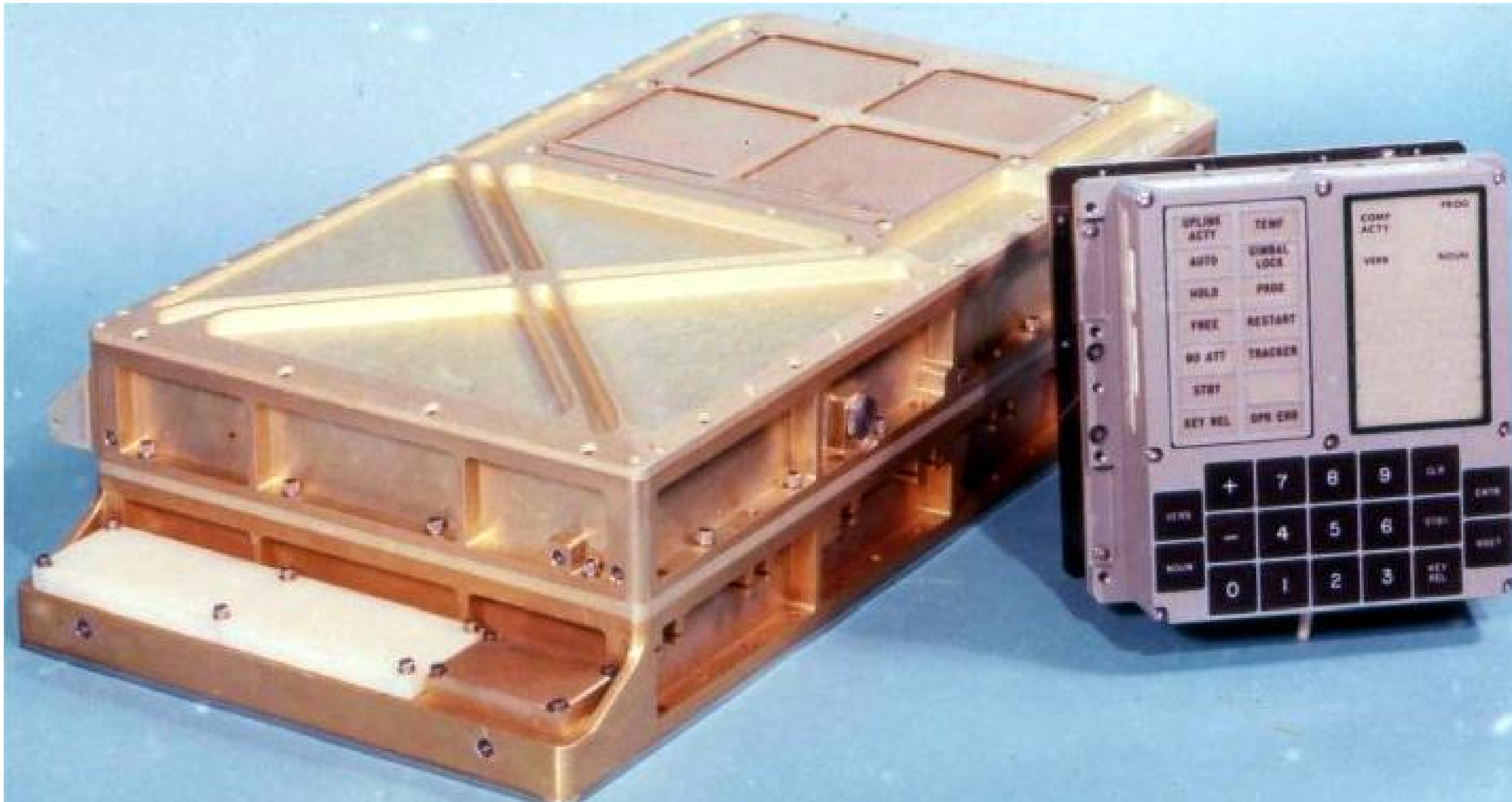
As for the term Internet of Things,
we here have another example of

computing terms not being absolute,
but historically conditioned,
and often leading to somewhat contradictory language.

The Things of the Internet are often
not on the Internet,

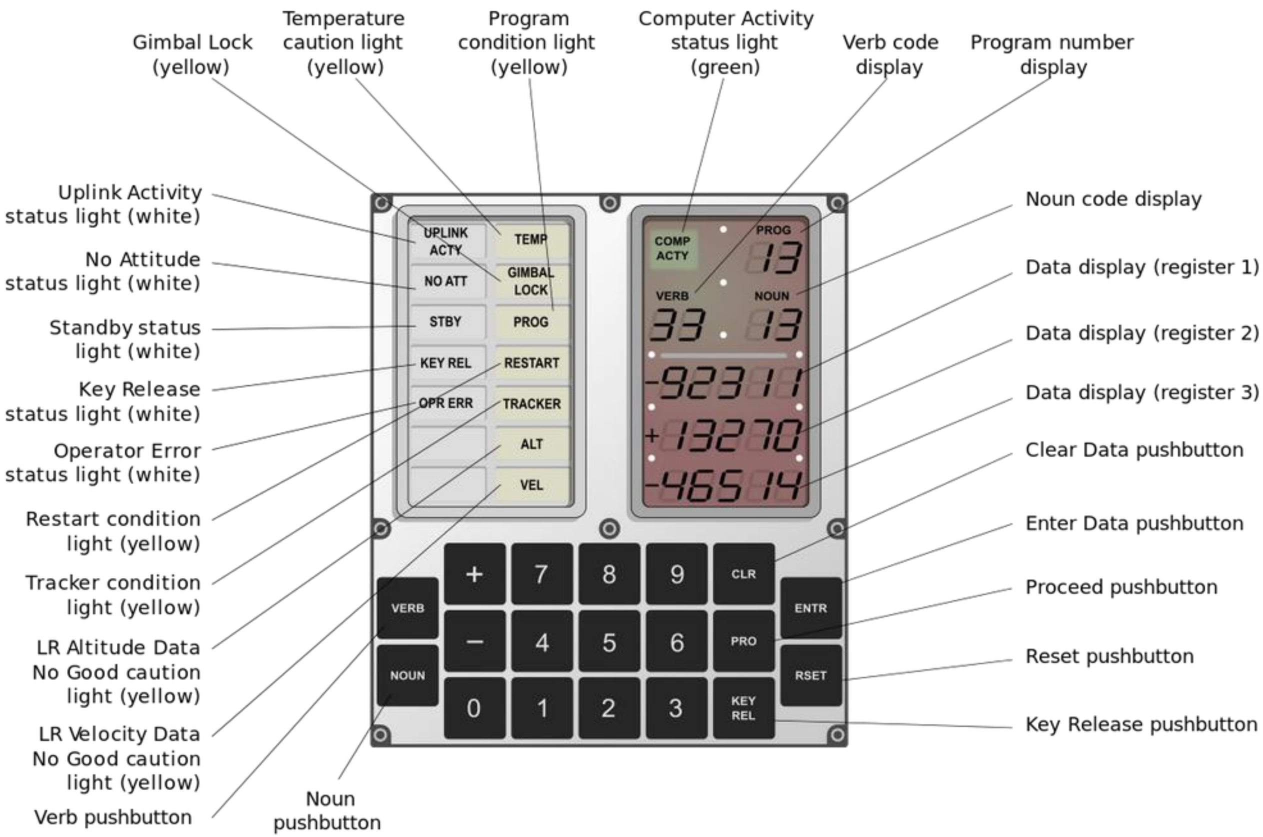
the Embedded Things are often not embedded,
but isolated.

1965 – Apollo Guidance Computer AGC

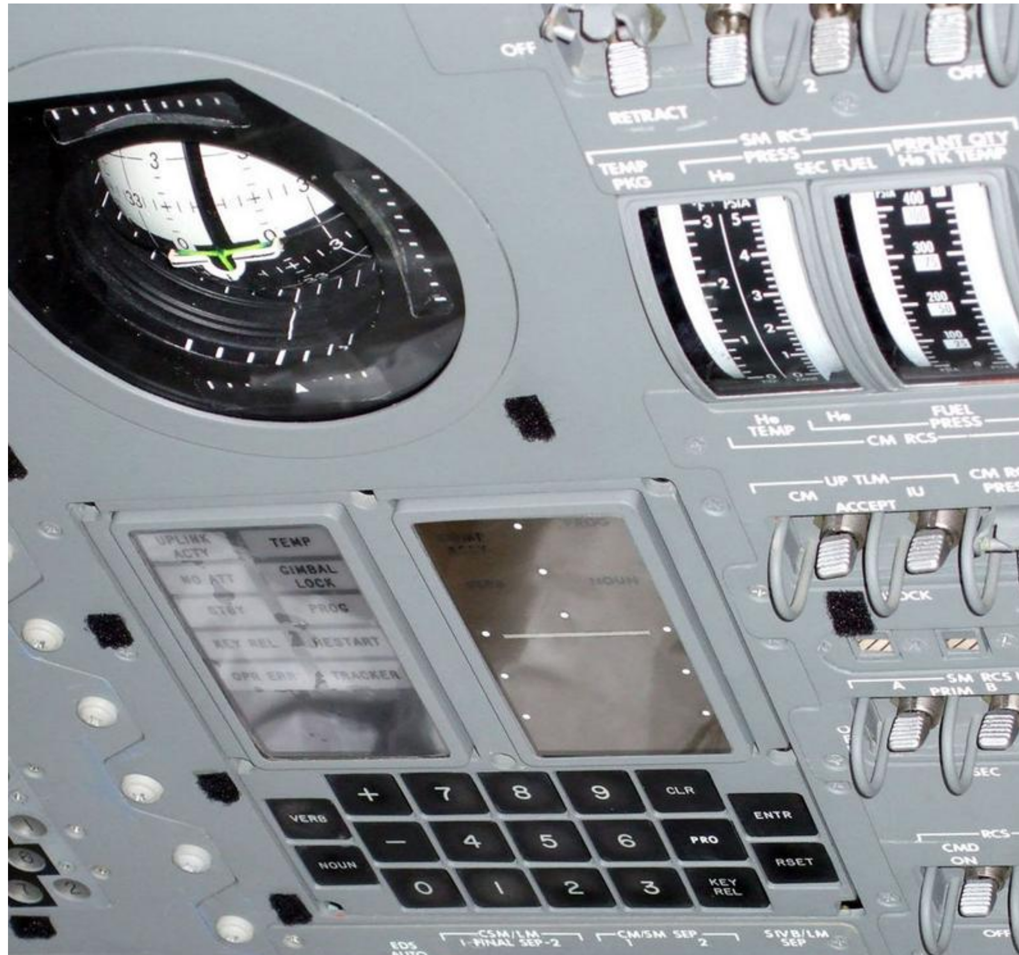


Embedded systems - history

1965 – Apollo Guidance Computer AGC



1965 – Apollo Guidance Computer AGC



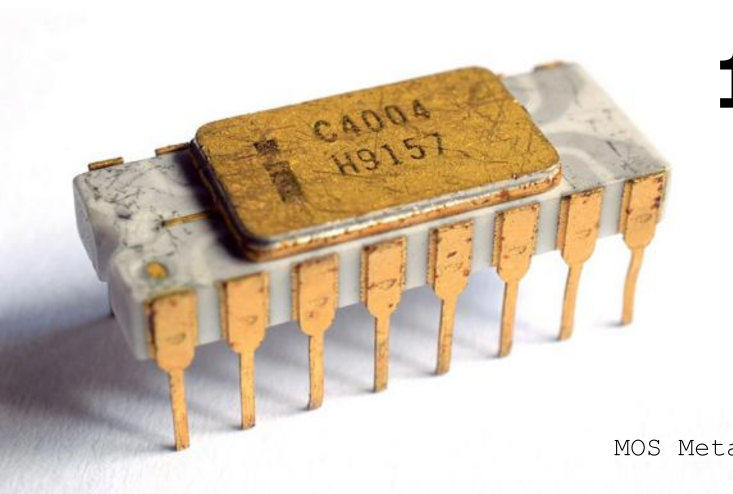
Embedded systems - history

1969 - MOS ICs

1969 - Four Phase AL1

1970 - Garrett AiResearch MP944

1971 - Intel 4004



MOS Metal Oxide Semiconductor

IoT and embedded systems

General embedded systems and the IoT share some basic **concerns & interests:**

- Size: minimize
- Power: optimize
- Networking: different from general purpose computer, often very specialized
- Cost
- Environmental robustness
- Computing power

specialized Concerns: Mars Rover 2020

RAD750 (2005)

what is the one very special concern for such a chip?

Constraints

We can contrast concerns & interests with our possibilities – and speak of **constraints.**

We know General Trade-offs from everyday life, e.g.

Quality – speed - price

Constraints of embedded systems

Very specific and multi-dimensional:

Power

Size

Cost

Network – speed/latency, bandwidth, synchronicity

Compute power

Temperature

Other Environmental (remember Mars ...)

Constraints analysis

Precise description of constraints

is essential for the design of good embedded system,

and lack thereof a key mistake one can make.

We often optimize things we do not need to optimize,

and forget those we should optimize.

What is an OS and what is not?

Context:

Course

C programming and Operating Systems

An operating system (OS) is a program that manages computer hardware.

|||

An operating system (OS) is system software that manages computer hardware, software resources, and provides common services for computer programs.

source: https://en.wikipedia.org/wiki/Operating_system

What is an OS and what is not?

An embedded system typically does NOT provide what a full operating system provides.

OS vs embedded systems

Operating system	**Embedded system**
Multi user	Single user
User privilges system	Usually only one privileged user
Multi task	Single task
Complex memory management	Limited memory management
Hardware independent	Tightly coupled to hardware
General purpose	Specialized

Processes / Linux

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
```

USER = user owning the process
PID = process ID of the process
%CPU = CPU time used divided by the time the process has been running.
%MEM = ratio of the process's RSS to the physical memory on the machine
VSZ = virtual memory usage of entire process (in KiB)
RSS = resident set size, the non-swapped physical memory that a task has used (in KiB)
TTY = controlling tty (terminal)
STAT = multi-character process state
START = starting time or date of the process
TIME = cumulative CPU time
COMMAND = command with all its arguments

Processes / Linux

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0 225740  9428 ?        Ss    2021   2:18 /lib/systemd/systemd --system --deserialize 38
root         2  0.0  0.0      0     0 ?        S     2021   0:01 [kthreadd]
root         4  0.0  0.0      0     0 ?        I<    2021   0:00 [kworker/0:0H]
root        11  0.0  0.0      0     0 ?        S     2021   0:28 [watchdog/0]
sebasti+   966  0.0  0.0 108364  5436 ?        R    15:55   0:00 sshd: sebastian@pts/0
sebasti+   969  0.2  0.0  24624  8308 pts/0    Ss   15:55   0:00 -bash
root      1390  0.0  0.0  71552  6128 ?        Ss    2021   0:16 /lib/systemd/systemd-logind
influxdb  1399 15.9  6.4 4328816 1066248 ?     Ssl   2021 38109:08 /usr/bin/influxd -config /etc/influxdb/influxdb.conf
root      1425  0.0  0.0 110560  3544 ?        Ssl   2021  15:13 /usr/sbin/irqbalance --foreground
prometh+  1426  0.1  0.4 1914676 75524 ?       Ssl   2021 427:54 /usr/bin/prometheus
daemon    1486  0.0  0.0  28340  2220 ?        Ss    2021   0:00 /usr/sbin/atd -f
prometh+  1505  0.1  0.0 1656768 14520 ?       Ssl   2021 338:12 /usr/bin/prometheus-node-exporter --collector.diskstats.ignored-devices=^(ram|loop|fd|(h|s|v|xv)d[a-z]|
nvmed+nd+p)d+$ --collector.filesystem
Debian-+  1509  0.0  0.0  64416 11316 ?        Ss    2021  55:48 /usr/sbin/snmpd -Lsd -Lf /dev/null -u Debian-snmp -g Debian-snmp -I -smux mteTrigger mteTriggerConf -f
mosquit+  1678  0.0  0.0  48032  5308 ?        S     2021  58:27 /usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf
redis     1824  0.0  0.0  50160  3620 ?        Ssl   2021 154:11 /usr/bin/redis-server 127.0.0.1:6379
mysql     2010  0.0  0.4 699740 78164 ?        Ssl   2021 105:12 /usr/sbin/mysqld
postfix   2525  0.0  0.0  73944  5016 ?        S     2021   0:12 qmgr -l -t unix -u
niec      2613  0.0  0.4 1295696 81548 ?       Sl   Feb06   0:03 /usr/bin/python3 /home/niec/.local/bin/gunicorn -b 0.0.0.0:5000 -w 4 mlflow.server:app
systemd+  3738  0.0  0.0  71864  3980 ?        Ss   Jan28   0:00 /lib/systemd/systemd-networkd
niec      4160  0.0  0.6 1369636 108628 ?      S     2021   0:01 /usr/bin/python3 /home/niec/.local/bin/mlflow server -h 0.0.0.0 --backend-store-uri
postgresql://mlflow:whatwouldyoudo@localhost:5432/mlflow
postfix   4527  0.0  0.0  87724  7864 ?        S     2021   0:03 tlsmgr -l -t unix -u -c
niec      4550  0.0  0.1  61000 23236 ?        S     2021   9:06 /usr/bin/python3 /home/niec/.local/bin/gunicorn -b 0.0.0.0:5000 -w 4 mlflow.server:app
appserv+ 11471  0.0  0.0 747832 14020 ?        Ssl  Jan28   9:13 /usr/bin/chirpstack-application-server
postgres 11513  0.0  0.1 4405068 19256 ?       Ss   Jan28   0:00 postgres: 11/main: chirpstack_as chirpstack_as ::1(53394) idle
root     11956  0.0  0.0      0     0 ?        I    Jan28   1:12 [kworker/4:3]
gateway+ 12160  0.0  0.0 715612  4312 ?        Ssl  Jan28  14:19 /usr/bin/chirpstack-gateway-bridge
postgres 12662  0.0  0.1 4405956 23208 ?       Ss   Jan30   0:00 postgres: 11/main: mlflow mlflow ::1(39654) idle
network+ 14127  0.1  0.0 728508 10616 ?        Ssl  Jan28  19:11 /usr/bin/chirpstack-network-server
postgres 14164  0.0  0.1 4405068 18952 ?       Ss   Jan28   0:00 postgres: 11/main: chirpstack_ns chirpstack_ns ::1(54848) idle
niec     14763  0.0  0.4 1295692 81564 ?       Sl   Feb05   0:04 /usr/bin/python3 /home/niec/.local/bin/gunicorn -b 0.0.0.0:5000 -w 4 mlflow.server:app
grafana  14984  0.1  0.4 2061756 67664 ?       Ssl  Jan28  18:26 /usr/sbin/grafana-server --config=/etc/grafana/grafana.ini --pidfile=/run/grafana/grafana-server.pid --
packaging=deb cfg:default.paths.logs=
postgres 15239  0.0  0.7 4404116 121192 ?      S     2021  27:11 /usr/lib/postgresql/11/bin/postgres -D /var/lib/postgresql/11/main -c
config_file=/etc/postgresql/11/main/postgresql.conf
postgres 15290  0.0  0.3 4404216 53560 ?       Ss    2021   0:02 postgres: 11/main: checkpointer
postgres 15292  0.0  0.2 4404116 40156 ?       Ss    2021   0:33 postgres: 11/main: background writer
niec     19976  0.0  0.6 1397052 112652 ?      Sl   Feb03   0:06 /usr/bin/python3 /home/niec/.local/bin/gunicorn -b 0.0.0.0:5000 -w 4 mlflow.server:app
syslog   20060  0.8  0.0 267272  4804 ?        Ssl   2021 750:08 /usr/sbin/rsyslogd -n
www-data 20906  0.0  0.0 183556 11320 ?        S    06:26   0:00 /usr/sbin/apache2 -k start
postgres 24321  0.0  0.2 4406276 39160 ?       Ss   Feb04   0:00 postgres: 11/main: mlflow mlflow ::1(40202) idle
postfix  29490  0.0  0.0  73816  5112 ?        S    15:28   0:00 pickup -l -t unix -u -c
root     30338  0.0  0.0 107996  7268 ?        Ss   15:55   0:00 sshd: sebastian [priv]
www-data 31290  0.0  0.1 534156 29580 ?        S    09:30   0:00 /usr/sbin/apache2 -k start
www-data 31848  0.0  0.1 534152 29776 ?        S    06:27   0:00 /usr/sbin/apache2 -k start
```

Common OS expectations

- **Multi user** –
  user space strictly separated from
  system / kernel space, separated between users
  (permission system)
- **Privileged / non-privileged space**
- Runlevels
- Multi tasking (processes, threads, forks, ..)
- Memory management,
- General system resources, ulimits

If not an OS, what then?

Embedded Systems, lacking a full Operating System, typically have something we call

Firmware
Scheduler
Operating Environment
Kernel

and yes, sometimes Operating System :(

source: Opler, Ascher (January 1967). "Fourth-Generation Software". Datamation. 13 (1): 22–24.

Firmware

In computing, **firmware**[a] is a specific class of computer software that provides the low-level control for a device's specific hardware. Firmware can either provide a standardized operating environment for more complex device software (allowing more hardware-independence), or, for less complex devices, act as the device's complete operating system, performing all control, monitoring and data manipulation functions. Typical examples of devices containing firmware are embedded systems, consumer appliances, computers, computer peripherals, and others. Almost all electronic devices beyond the simplest contain some firmware.

source: Opler, Ascher (January 1967). "Fourth-Generation Software". Datamation. 13 (1): 22–24.

Firmware

The word "software" suggests that there is a single entity, separate from the computer's hardware, that works with the hardware to solve a problem.
In fact, there is no such single entity. A computer system is like an onion, with many distinct layers of software over a hardware core.
Even at the center—the level of the central processor—there is no clear distinction: computer chips carrying "microcode" direct other chips to perform the processor's most basic operations. Engineers call these codes "firmware," a term that suggests the blurred distinction.

source: Ceruzzi, P. E., & Paul, E. (2003). A history of modern computing. MIT press.

Firmware

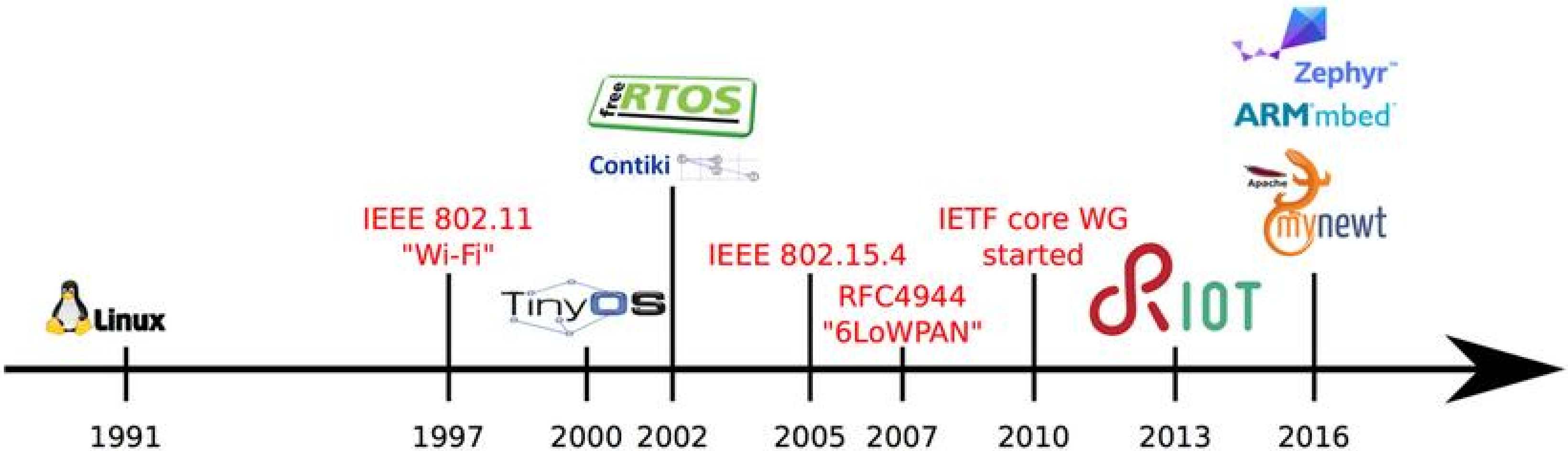
In every day use (Denmark, 2020), the work firmware is often used to denote any code that runs on an embedded node, i.e.

including the program/sketch running on the device.

“upload the new firmware”

This seems inconsistent.

Embedded operating systems



Hahm, O., Baccelli, E., Petersen, H., & Tsiftes, N. (2015). Operating systems for low-end devices in the internet of things: a survey. IEEE Internet of Things Journal, 3(5), 720-734.

Embedded Systems: TinyOS

TinyOS is an open source, BSD-licensed operating system designed for low-power wireless devices, such as those used in sensor networks, ubiquitous computing, personal area networks, smart buildings, and smart meters. A worldwide community from academia and industry use, develop, and support the operating system as well as its associated tools, averaging 35,000 downloads a year.

Latest News

January, 2013: The transition to hosting at GitHub is now complete. Part of this transition includes slowly retiring TinyOS development mailing lists for bug tracking and issues to using the GitHub trackers. Thanks to all of the developers who are now improving TinyOS and requesting pulls!

August 20, 2012: TinyOS 2.1.2 is now officially released; you can download it from the debian packages on tinyos.stanford.edu. Manual installation with RPMs with the instructions on docs.tinyos.net will be forthcoming. TinyOS 2.1.2 includes:

- Support for updated msp430-gcc (4.6.3) and avr-gcc (4.1.2).
- A complete 6lowpan/RPL IPv6 stack.
- Support for the ucmini platform and ATmega128RFA1 chip.
- Numerous bug fixes and improvements.

Embedded Systems: TinyOS

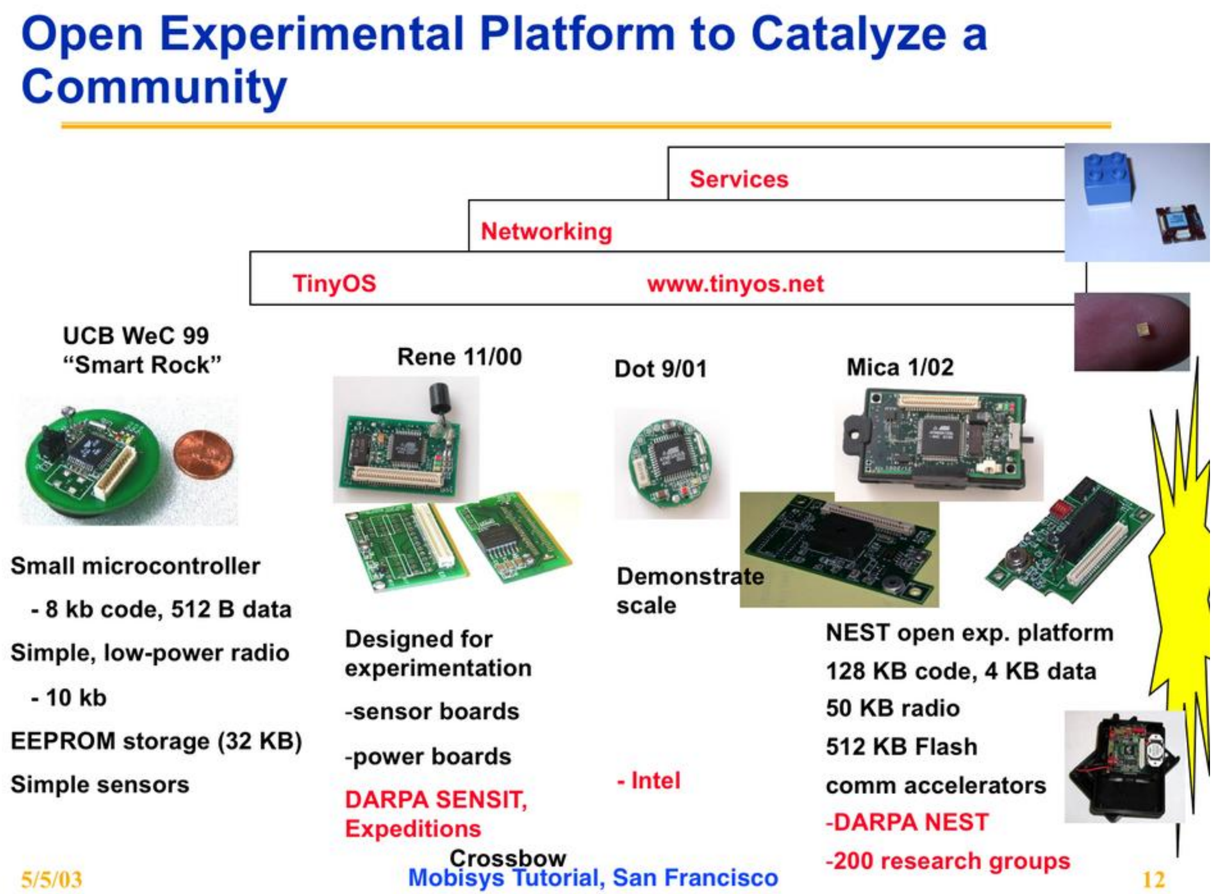
Developed at Stanford,
“TinyOS is an embedded, component-based operating system and platform for low-power wireless devices, such as those used in wireless sensor networks (WSNs), smartdust, ubiquitous computing, personal area networks, building automation, and smart meters.”

It is interchangeably called an
operating system or an *operating environment*.
It s developed in its own C-derived language, NesC.

Latest Release: 2012 ...

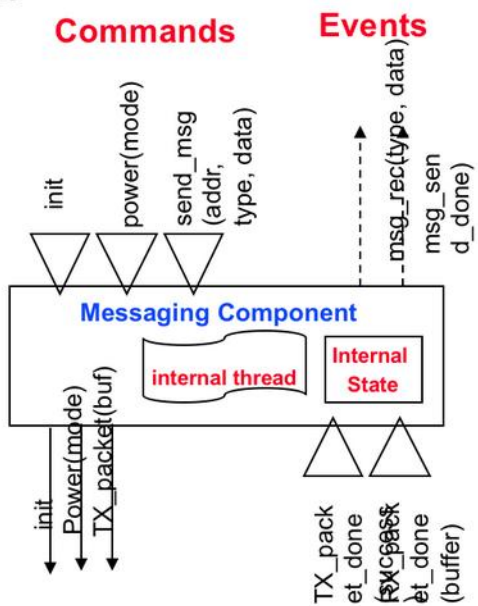
TinyOS

http://www.tinyos.net/ < 2013!!!

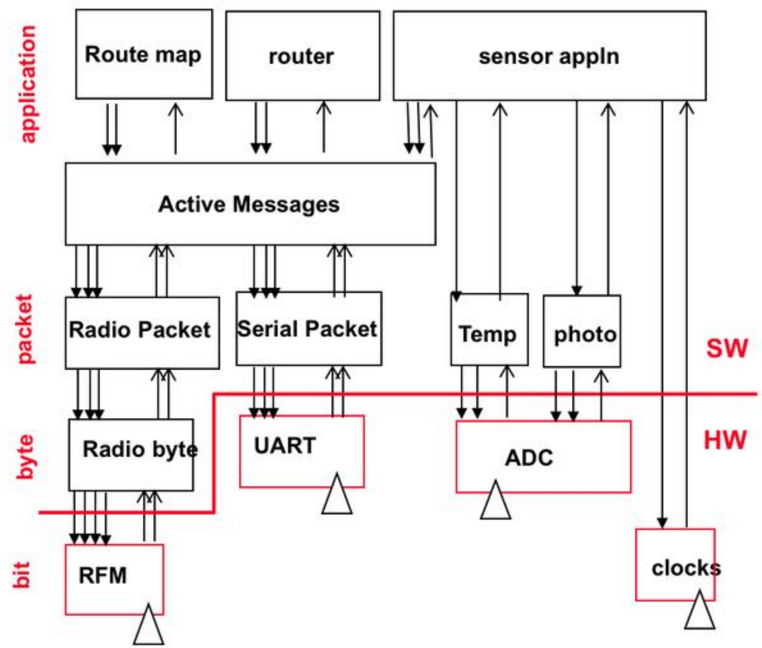


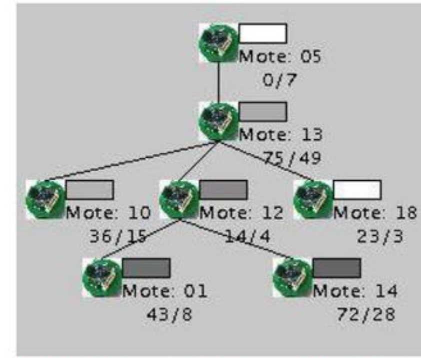
Tiny OS Concepts

- **Scheduler + Graph of Components**
  - constrained two-level scheduling model: threads + events
- **Component:**
  - Commands,
  - Event Handlers
  - Frame (storage)
  - Tasks (concurrency)
- **Constrained Storage Model**
  - frame per component, shared stack, no heap
- **Very lean multithreading**
- **Efficient Layering**



Application = Graph of Components





Example: ad hoc, multi-hop routing of photo sensor readings

3450 B code
226 B data

Graph of cooperating state machines on shared stack

NesC language

Embedded Systems: Contiki

Contiki was created by Adam Dunkels in 2002[2] and has been further developed by a worldwide team of developers from Texas Instruments, Atmel, Cisco, ENEA, ETH Zurich, Redwire, RWTH Aachen University, Oxford University, SAP, Sensinode, Swedish Institute of Computer Science, ST Microelectronics, Zolertia, and many others.

It seems somewhat active, in its new form contiki-ng - https://github.com/contiki-ng/contiki-ng/wiki

Contiki-NG is an operating system for resource-constrained devices in the Internet of Things. Contiki-NG contains an RFC-compliant, low-power IPv6 communication stack, enabling Internet connectivity. The system runs on a variety of platforms based on energy-efficient architectures such as the ARM Cortex-M3/M4 and the Texas Instruments MSP430. The code footprint is on the order of a 100 kB, and the memory usage can be configured to be as low as 10 kB. The source code is available as open source with a 3-clause BSD license.

Embedded Systems: Contiki

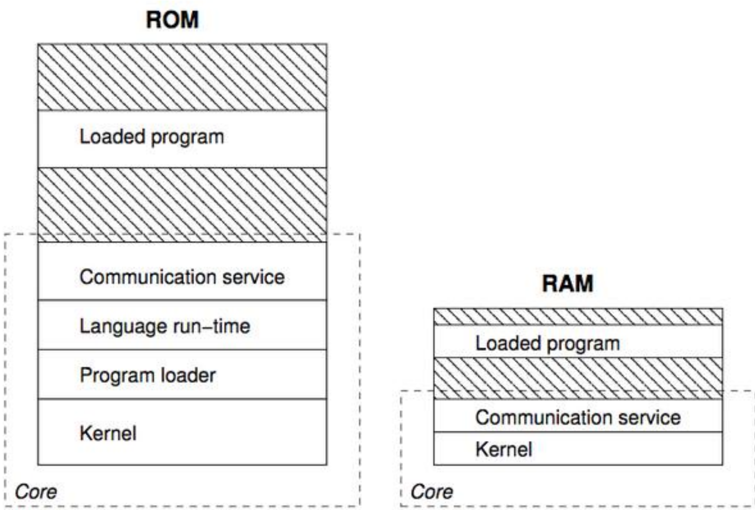
Contiki's big headline was

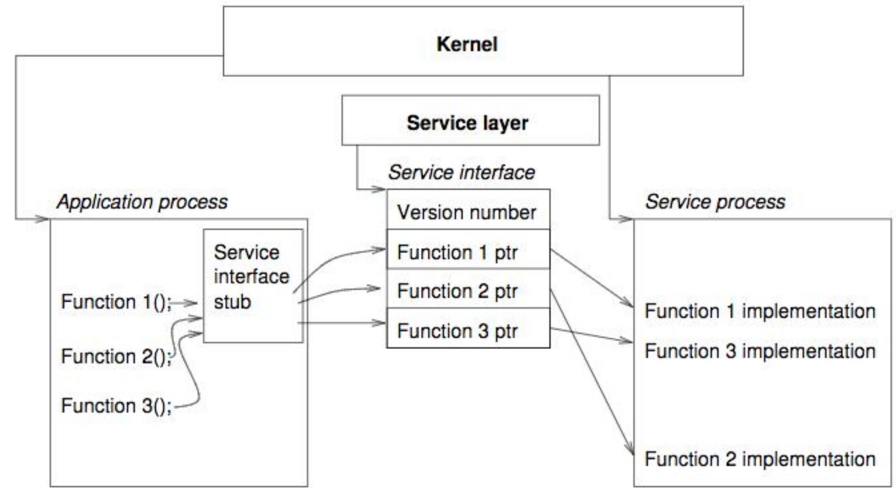
TCP/IP over IPv6 -

offering end-to-end connectivity to IoT devices – a concept which has since lost most of its momentum in the IoT marketplace, with security being one factor, and the overall delayed adoption curve for IPv6 as another.

Contiki

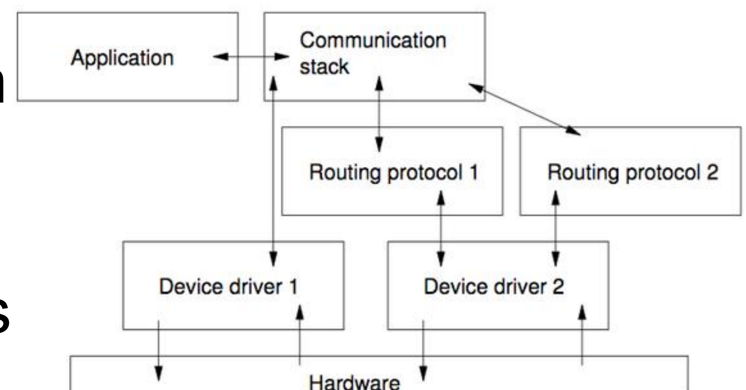
http://www.contiki-os.org/





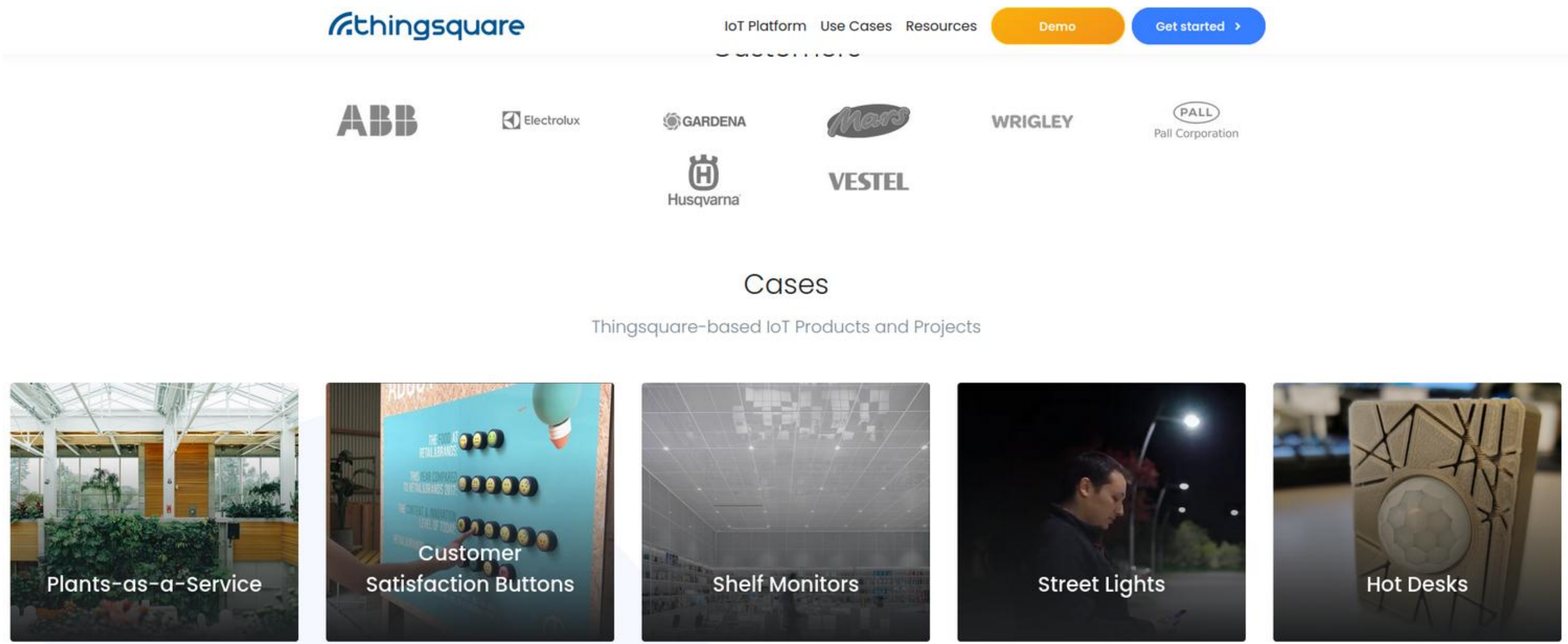
Protothreads
Dynamic Memory Allocation
IP Stack (6lowpan)

Range of Atmel/TI platforms



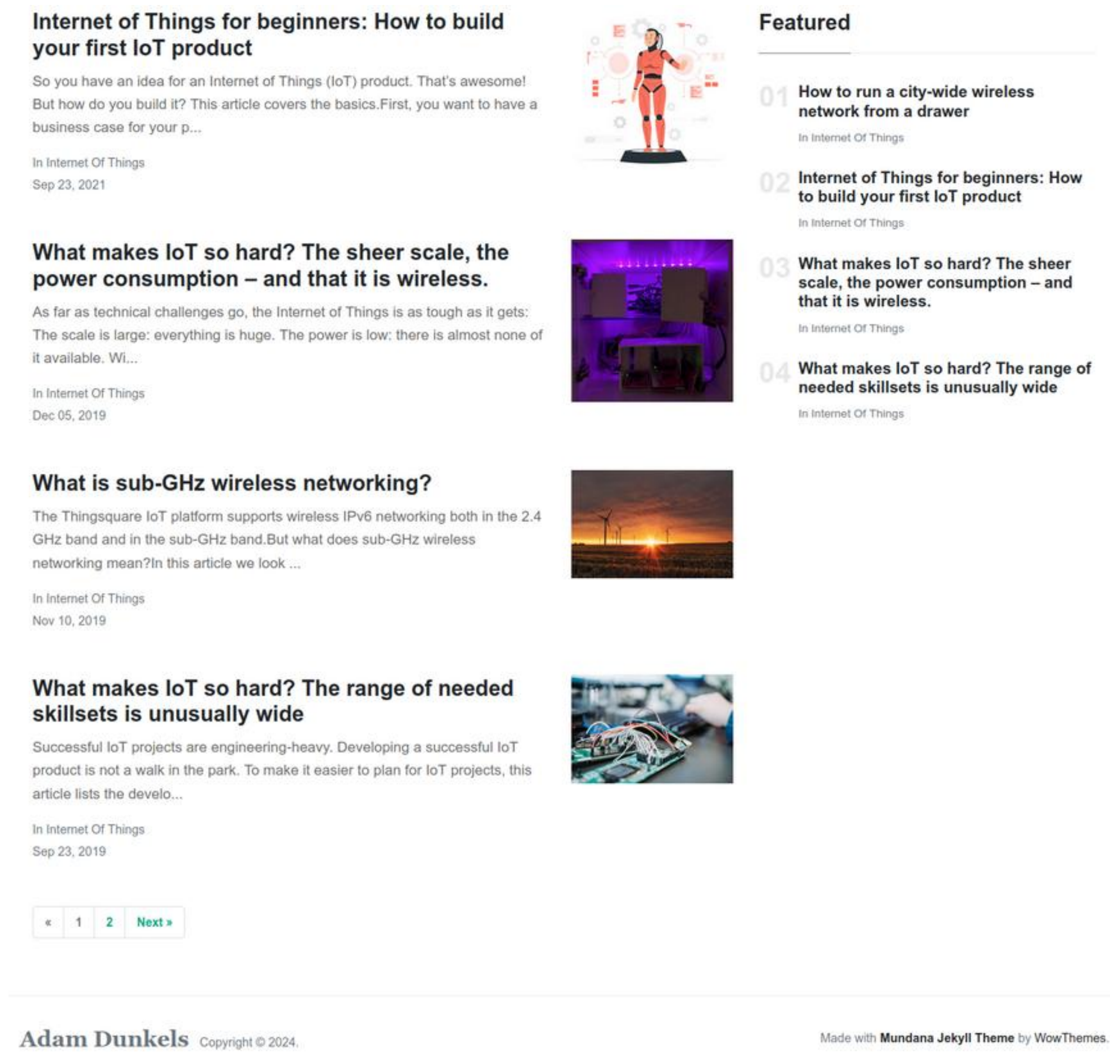
Embedded Systems: Contiki

Commercialized as **Thingsquare – all-IP mesh networks**



Embedded Systems: Contiki

Adam Dunkel’s page still worth a visit



Internet of Things for beginners: How to build your first IoT product

So you have an idea for an Internet of Things (IoT) product. That's awesome! But how do you build it? This article covers the basics.First, you want to have a business case for your p...

In Internet Of Things
Sep 23, 2021

What makes IoT so hard? The sheer scale, the power consumption – and that it is wireless.

As far as technical challenges go, the Internet of Things is as tough as it gets: The scale is large: everything is huge. The power is low: there is almost none of it available. Wi...

In Internet Of Things
Dec 05, 2019

What is sub-GHz wireless networking?

The Thingsquare IoT platform supports wireless IPv6 networking both in the 2.4 GHz band and in the sub-GHz band.But what does sub-GHz wireless networking mean?In this article we look ...

In Internet Of Things
Nov 10, 2019

What makes IoT so hard? The range of needed skillsets is unusually wide

Successful IoT projects are engineering-heavy. Developing a successful IoT product is not a walk in the park. To make it easier to plan for IoT projects, this article lists the develo...

In Internet Of Things
Sep 23, 2019

« 1 2 Next »

Featured

01 How to run a city-wide wireless network from a drawer
In Internet Of Things

02 Internet of Things for beginners: How to build your first IoT product
In Internet Of Things

03 What makes IoT so hard? The sheer scale, the power consumption – and that it is wireless.
In Internet Of Things

04 What makes IoT so hard? The range of needed skillsets is unusually wide
In Internet Of Things

Adam Dunkels

Made with **Mundana Jekyll Theme** by WowThemes.

Embedded Systems: RIOT OS

Another one of the older contenders, hailing fom Berlin, a.o.

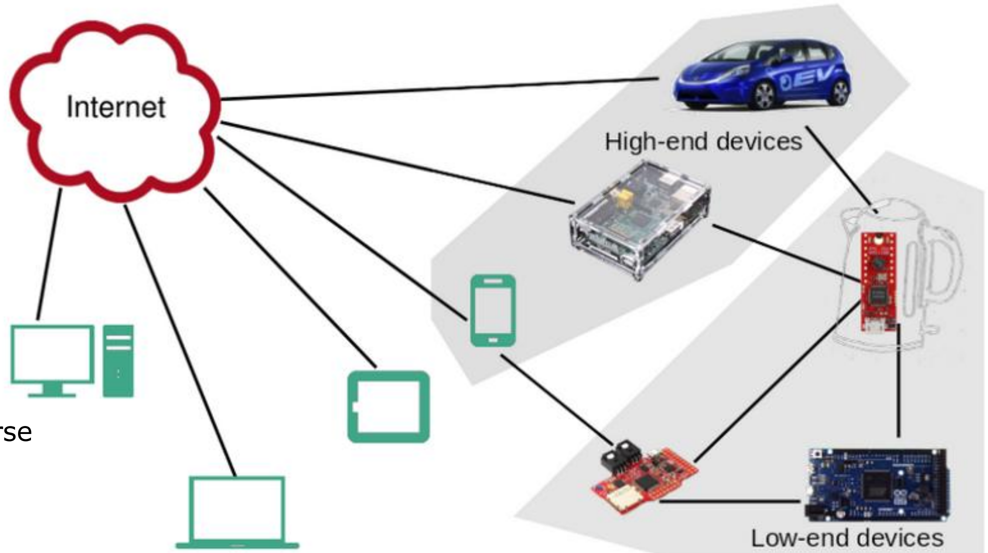
https://www.riot-os.org/

RIOT has recently modernized itself, adopting LoRa, offering an **online course**

https://github.com/riot-os/riot-course#content-of-the-course

Supports wide choice of boards

RIOT in the IoT world



⇒ RIOT is designed for low-end devices
(kB RAM, MHz, mW)

Embedded Systems: RIOT OS

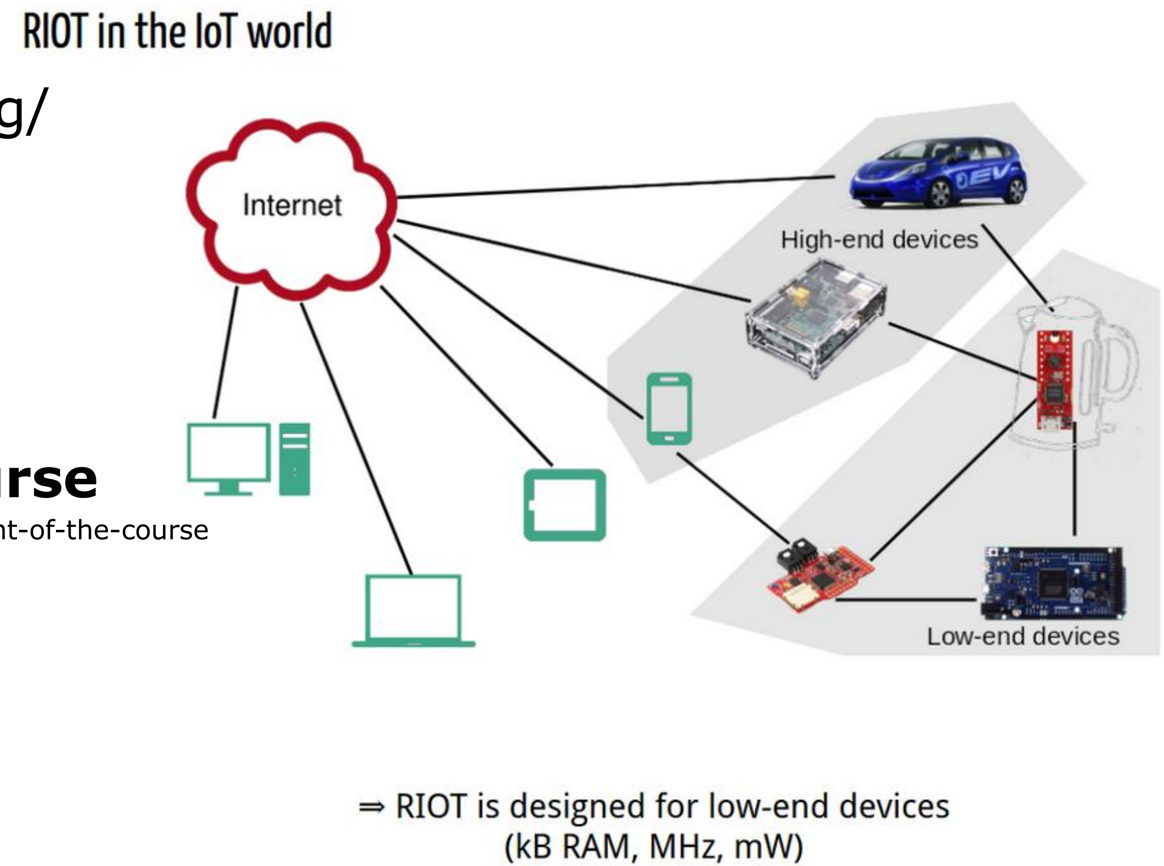
Another one of the older contenders, hailing fom Berlin, a.o.

https://www.riot-os.org/

RIOT has recently modernized itself, adopting LoRa, offering an **online course**

https://github.com/riot-os/riot-course#content-of-the-course

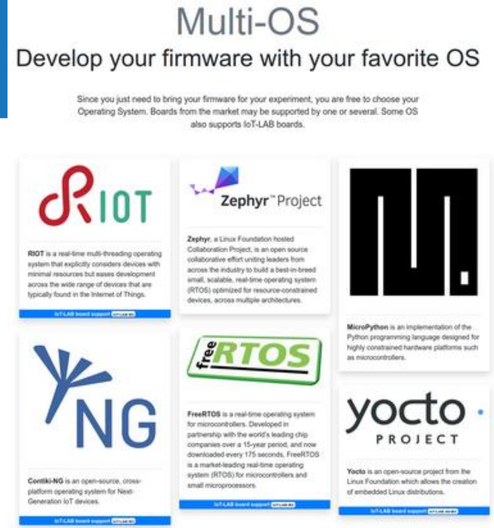
Supports wide choice of boards

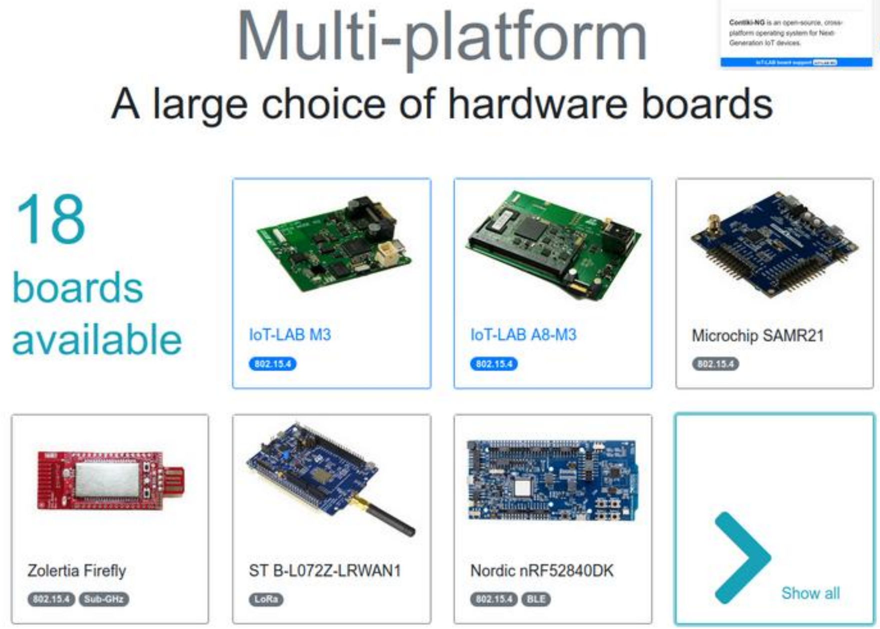


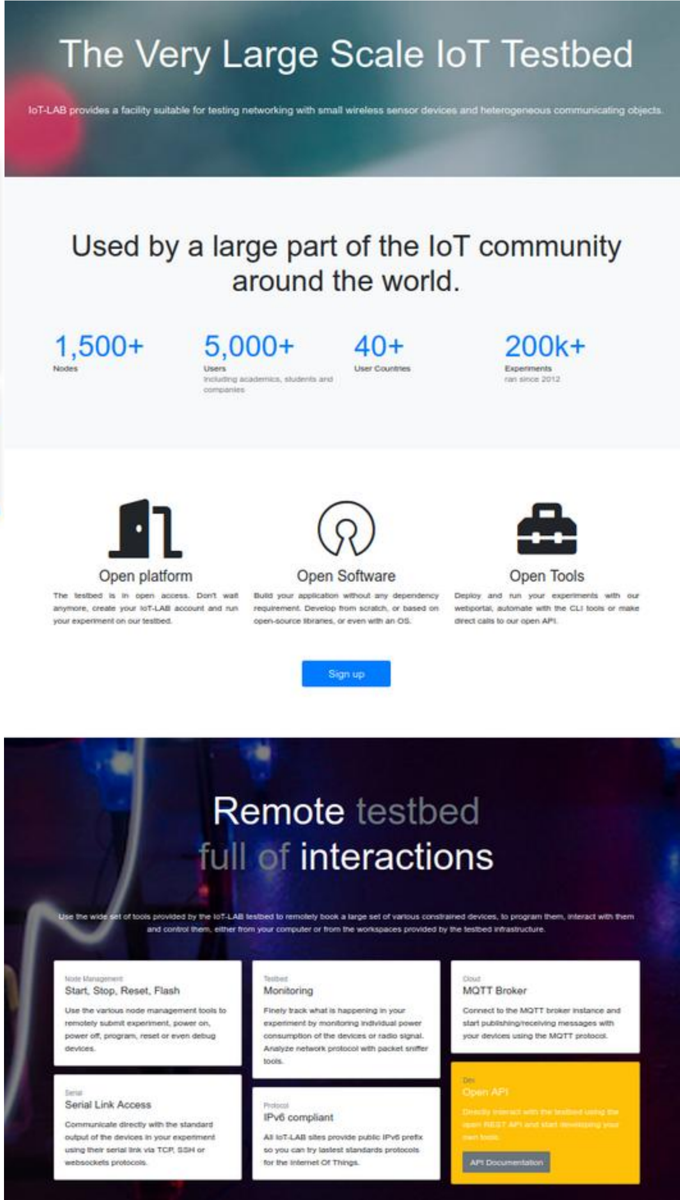
RIOT

Side comment:
RIOT uses for courses

the fit iot-lab
https://www.iot-lab.info/







RIOT

https://riot-os.org/

Linux-inspired
Micro-kernel based architecture
Multi-threading + real-time
On-demand wake-up from deep sleep mode

- ✓ Robustness & code-footprint flexibility
- ✓ Enabling maximum energy-efficiency
- ✓ Real-time capability due to ultra-low interrupt latency (~50 clock cycles) and priority-based scheduling
- ✓ Multi-threading with ultra-low threading overhead (<25 bytes per thread)

- ✓ 6LoWPAN, IPv6, RPL, and UDP
- ✓ CoAP and CBOR
- ✓ Static and dynamic memory allocation
- ✓ High resolution and long-term timers
- ✓ Tools and utilities (System shell, SHA-256, Bloom filters, ...)

http://riot-os.org/files/2018-IEEE-IoT-Journal-RIOT-Paper.pdf

Arduino

"Arduino is an open-source electronics platform based on easy-to-use hardware and software" (website)

Arduino is a good example of the merging of hardware, software, cultural and social aspects.
It s part business,
part movement,
It s hardware, IDE, education,
… and commercial use.

Hardware:Components of an embedded board

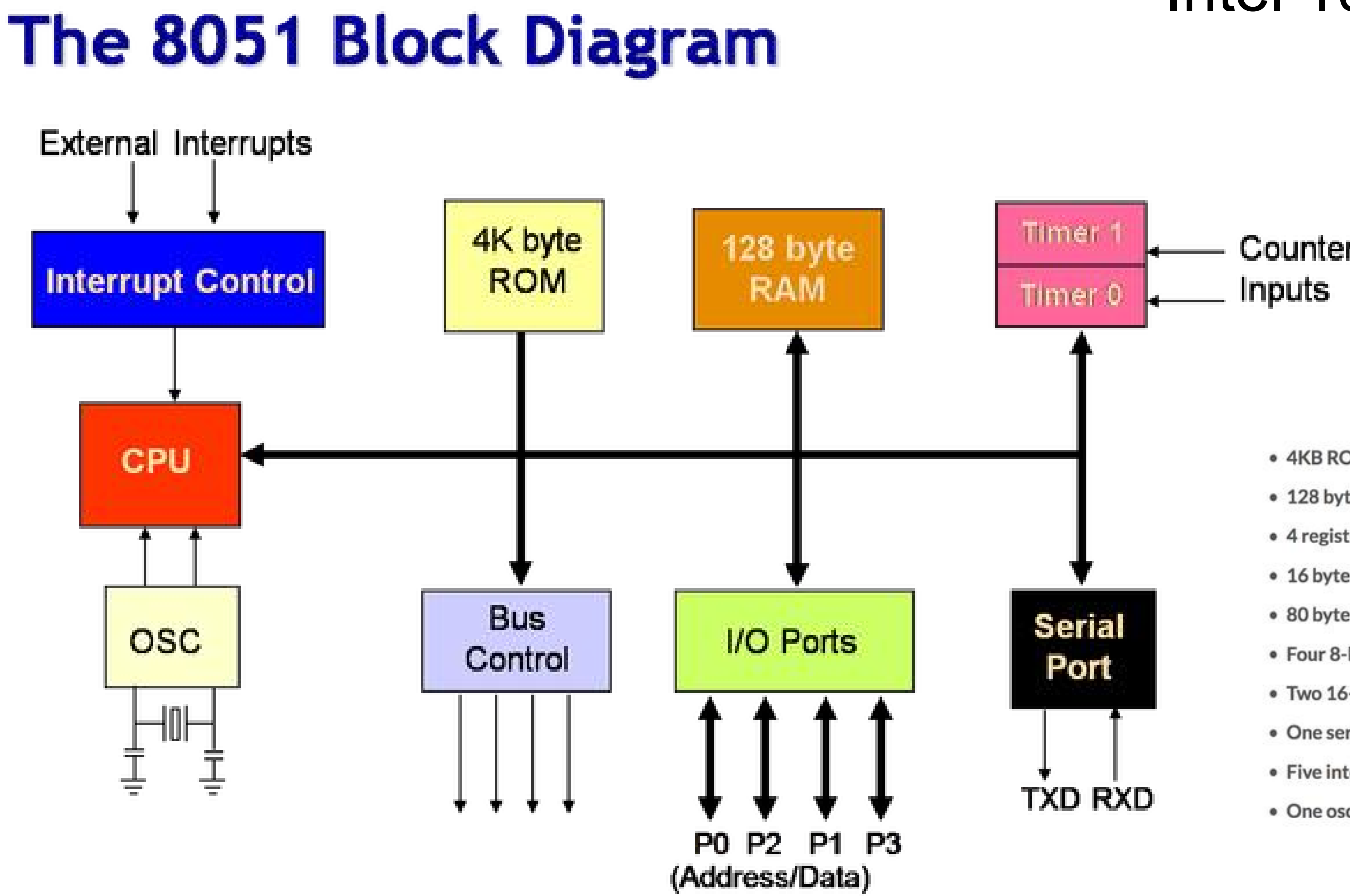
- power
- processor
- ram
- storage, volatile and non-volatile
- timer/clock
- on-board comms, buses
- network
- i/o of various types
- (led/sound)

Embedded boards: Examples

- Intel 8051 (1980!) as historical reference
- Espressif ESP32 / LoPy4 (no longer ..) / TTGO ESP32
- Arduinos
- Heltec ARM Cortex
- timer/clock
- on-board comms, buses
- network
- i/o of various types
- (led/sound)

Micro-controller Example#1: 8051

Intel 1980

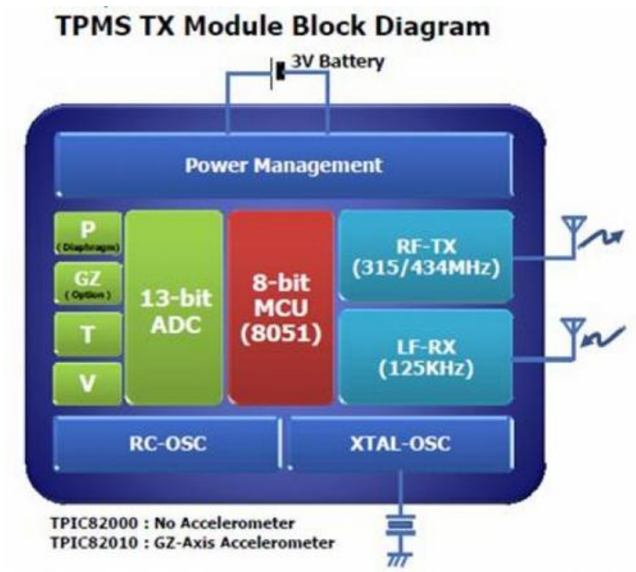


- 4KB ROM
- 128 bytes internal RA
- 4 register banks of 8 bytes each (R0-R7)
- 16 bytes of bit-addressable area
- 80 bytes of general purpose memory
- Four 8-bit I/O ports (P0-P3)
- Two 16-bit timers (Timer0 & Timer1)
- One serial receiver-transmitter interface
- Five interrupt sources (2 external & 3 internal)
- One oscillator (generates clock signal)

source: https://aninditadhikary.wordpress.com/tag/intel-8051/

8051 and IoT?

https://www.technobyte.org/2016/10/8051-not-old-still-popular/



Tire Pressure
Monitor System (TI)

1. **Energy Management:** Competent measuring device systems aid in calculating energy consumption in domestic and industrialized applications. These meter systems are prepared competent by integrating microcontrollers.
2. **Touch screens:** A high degree of microcontroller suppliers integrate touch sensing abilities in their designs. Transportable devices such as media players, gaming devices & cell phones are some illustrations of micro-controller integrated with touch sensing screens.
3. **Automobiles:** The microcontroller 8051 discovers broad recognition in supplying automobile solutions. They are extensively utilized in hybrid motor vehicles to control engine variations. In addition, works such as cruise power and anti-brake mechanism has created it more capable with the amalgamation of micro-controllers.
4. **Medical Devices:** Handy medicinal gadgets such as glucose & blood pressure monitors bring into play micro-controllers, to put on view the measurements, as a result, offering higher dependability in giving correct medical results.
5. **Medical Devices:** Handy medicinal gadgets such as glucose & blood pressure monitors bring into play micro-controllers, to put on view the measurements, as a result, offering higher dependability in giving correct medical results.

8051 IP cores are free to use

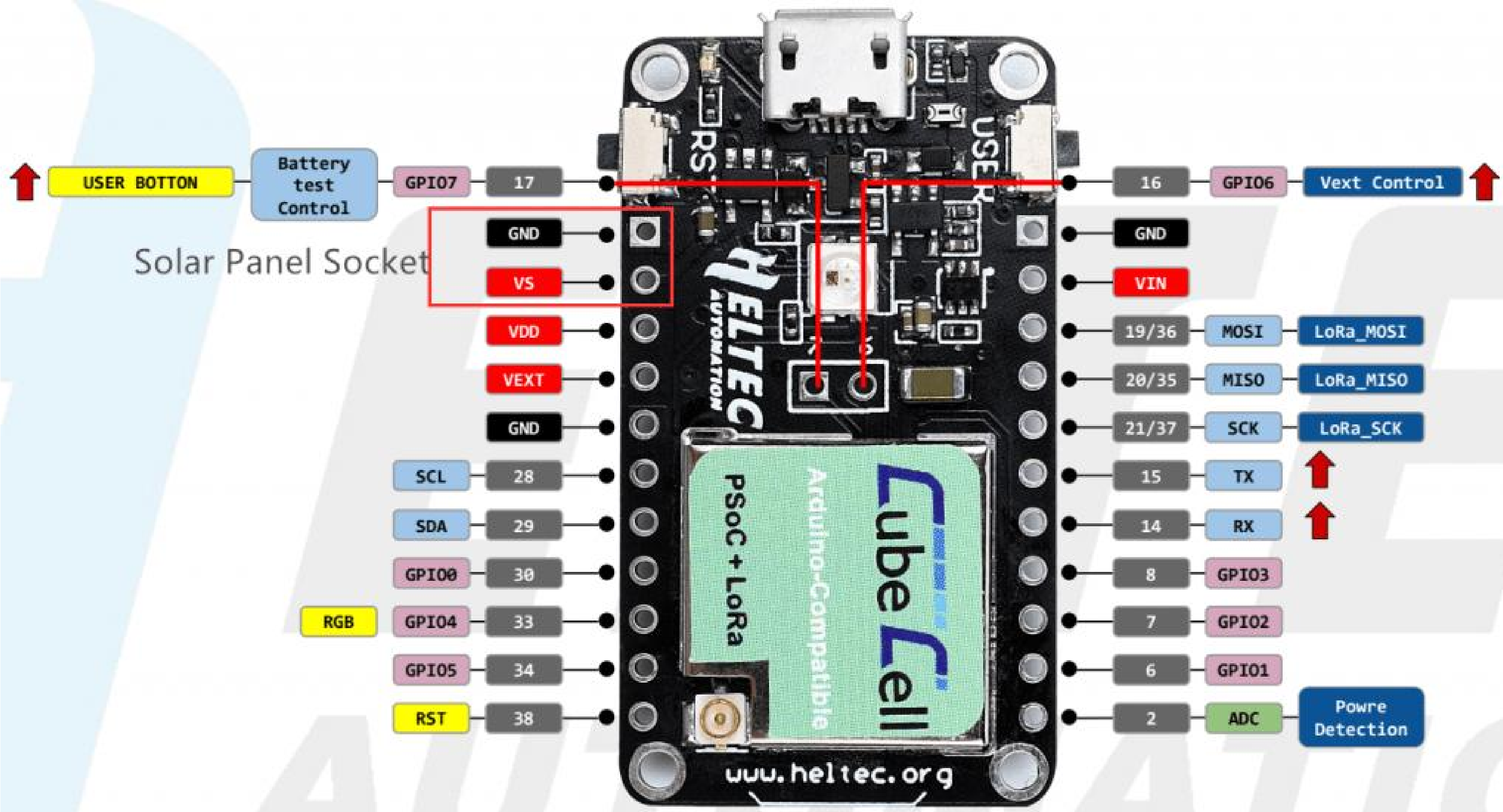
8051 vs ESP32

	8051	ESP32
RO	4KB	
RAM	128 byte	4 MB, 8 MB flash
I/O	4 8-bit GPIO	GPIO: Up to 24, 8 x 12-bit ADCs
Serial	1	2 x UART, SPI, 2 x I2C, I2S
Network	-	Wi-Fi, Sigfox, LTE, LoRa

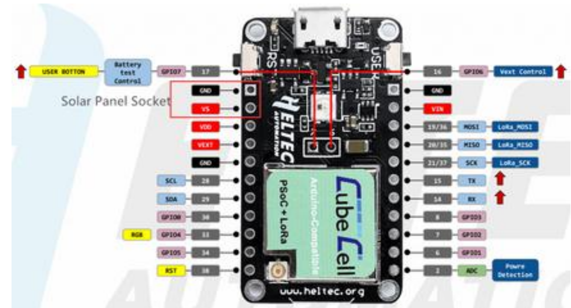
8051 vs Arduino Pro Mini

	8051	Pro Mini ATmega328P
RAM	128 byte	32 kB (2 kB f bootloader)
I/O	4 8-bit GPIO	14 I/O 8 x 12-bit ADCs
Serial	1	UART, SPI, I2C
Network	-	-

8051 vs Heltec Cubecell



8051 vs Heltec Cubecell



	8051	Cubecell
		ASR605x, integrated the PSoC® 4000 series MCU (ARM® Cortex® M0+ Core) and SX1262;
RAM	128 byte	128 kB flash
I/O	4 8-bit GPIO	6 I/O, 8 x 12-bit ADCs
Serial	1	UART, SPI, I2C
Network	-	LoRa

DeepSleep 3.5 uA !!!!!

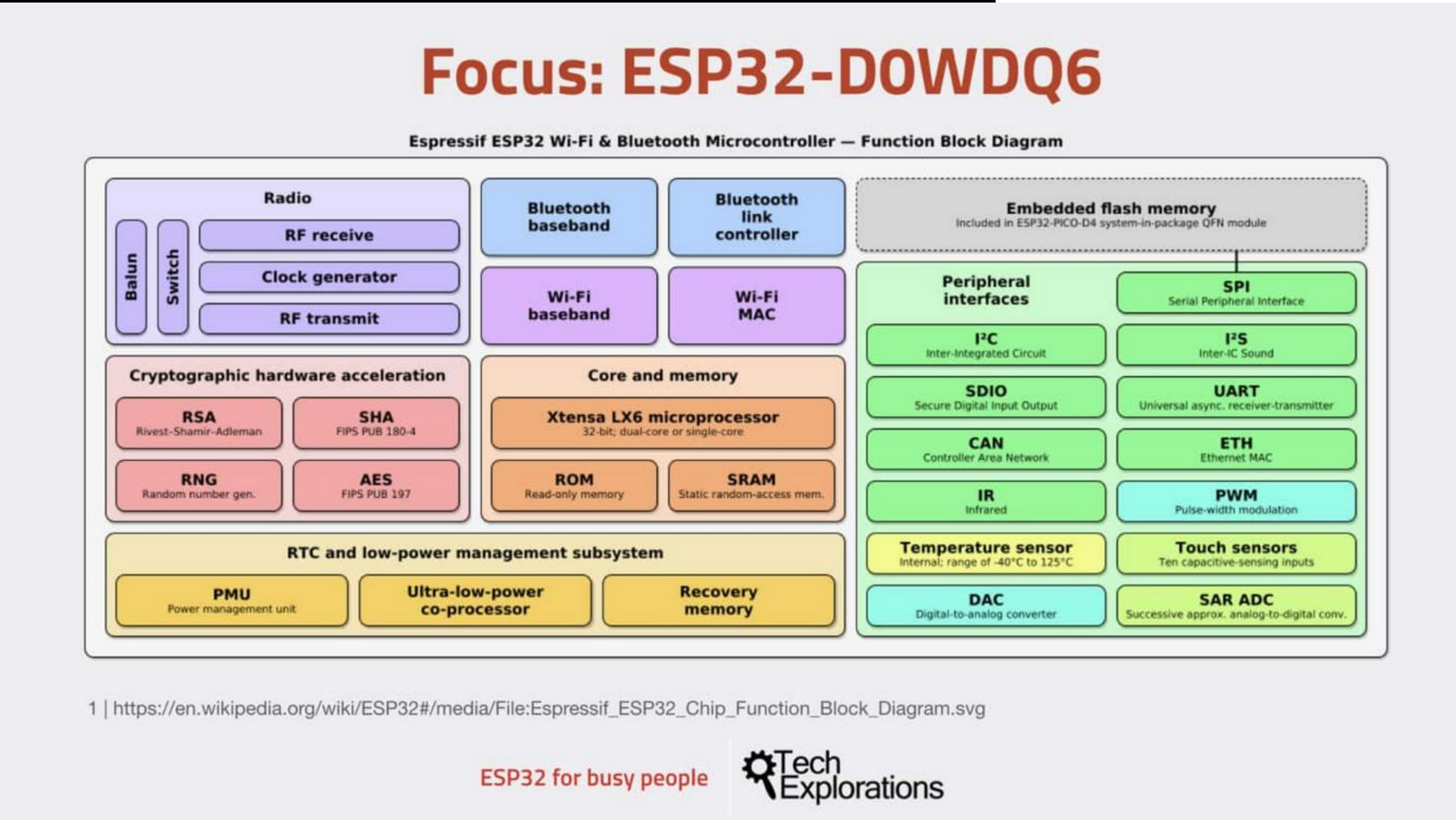
8051 vs ESP32

what is new **2020 vs 1980 (and even 2010)?**

Networking	multi-network as standard IP exception rather than norm
Hash/Encryption Security	SHA, MD5, DES, AES SSL
Power optimization	(deep) sleep modes
Cost	simple boards for ~$

LoPy4 – ESP32 – processor

ESP32 – block diagram

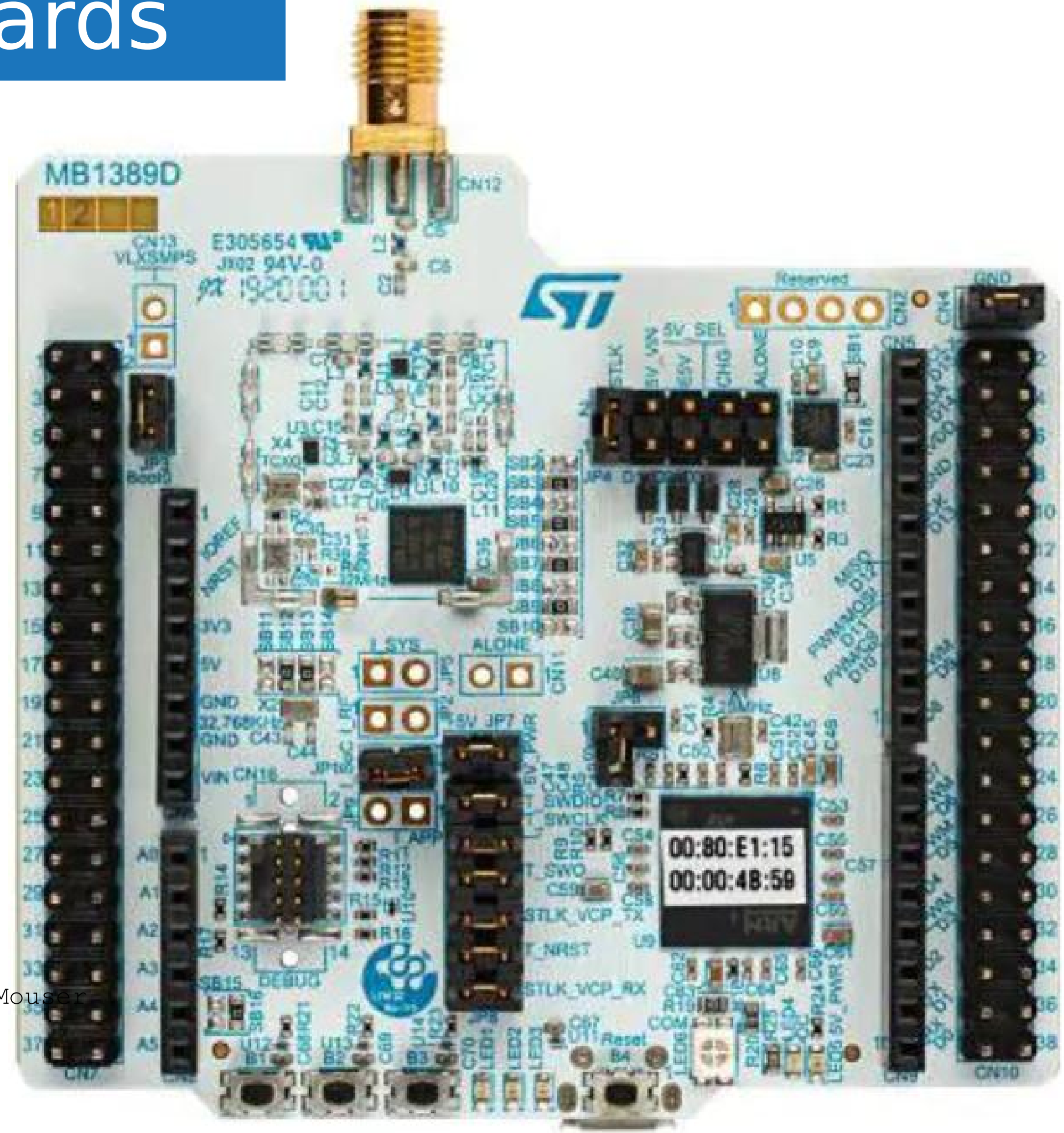


STM32WL55JC boards

<=> Generic Node:
same SoC

New board with
Tight integration of
LoRa chip SX 1262
(**LR-FHSS**!)
(=> Networking)

==> interest from
satellite perspective



source: ST / Mouser

STM32WL55JC boards

- STM32WL55JC microcontroller multiprotocol LPWAN dual-core 32-bit (Arm® Cortex®-M4/M0+ at 48 MHz) in UFBGA73 package featuring:
  - Ultra-low-power MCU
  - RF transceiver (150 MHz to 960 MHz frequency range) supporting LoRa®, (G)FSK, (G)MSK, and BPSK modulations
  - 256-Kbyte Flash memory and 64-Kbyte SRAM
- 3 user LEDs
- 3 user buttons and 1 reset push-button
- 32.768 kHz LSE crystal oscillator
- 32 MHz HSE on-board oscillator
- Board connectors:
  - USB with Micro-B
  - MIPI debug connector
  - ARDUINO Uno V3 expansion connector
  - ST morpho extension pin headers for full access to all STM32WL I/Os
- Delivered with SMA antenna
- Flexible power-supply options: ST-LINK, USB V_{BUS}, or external sources
- On-board STLINK-V3 debugger/programmer with USB re-enumeration capability: mass storage, Virtual COM port, and debug port
- Comprehensive free software libraries and examples available with the STM32CubeWL MCU Package
- Support of a wide choice of Integrated Development Environments (IDEs) including IAR Embedded Workbench®, MDK-ARM, and STM32CubeIDE
- Suitable for rapid prototyping of end nodes based on LoRaWAN, Sigfox, wM-Bus, and many other proprietary protocols
- Fully open hardware platform

source: ST / Mouser

The biggest successes in the IoT space

Seen from the angle of numbers sold and presence in the "maker" space, the most successful platforms are:

- Arduino (and compatible)
- Raspberry Pi (GNU/Linux)
- Android?

C. Other Software

For the sake of completeness, we also summarize in this section a collection of other pieces of software that are sometimes mentioned as potential contenders, but in fact are not full-fledged OSs, or are not applicable on Class 1 devices.

1) Arduino [88]: Originating from a university project, Arduino is an open source hardware and software company. Bundled with an IDE targeting people unfamiliar with programming, it enables easy prototyping. Good support for hardware features is achieved by the fact that Arduino provides both platforms and software. Arduino does not, however, provide a real scheduler, support for threading, or any higher layer functionality, thus making it suitable primarily for simpler applications.

Hahm, O., Baccelli, E., Petersen, H., & Tsiftes, N. (2015).
Operating systems for low-end devices in the internet of things: a survey.
IEEE Internet of Things Journal, 3(5), 720-734.

Sensors / Communication in embedded devices I

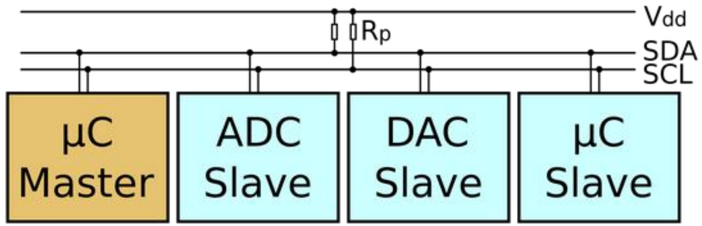
- **Short distance (intra board)**
- **Moderate data rates (kBps)**

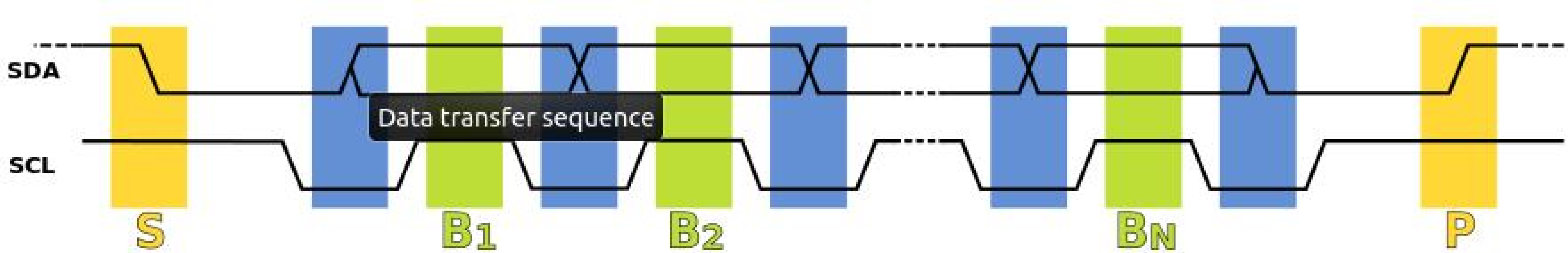
Three/Four **most popular standards:**

- **I²C** (Inter-Integrated Circuit)
- **SPI** (Serial Peripheral Interface)
- **UART** - RS232
- 1-Wire

- **I²C** (Inter-Integrated Circuit), pronounced I-squared-C, is a synchronous, multi-master, multi-slave, packet switched, single-ended, serial computer bus (1982 Philips Semiconductor, now NXP Semiconductors). Two bidirectional wires: Serial Data Line (SDA) and Serial Clock Line (SCL)

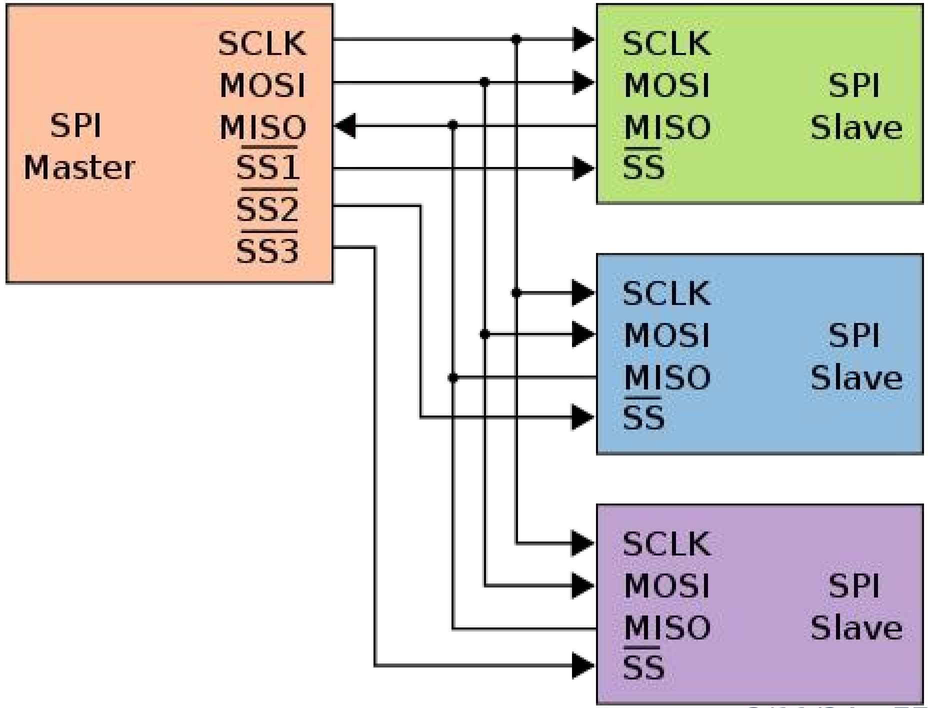
```python
from machine import I2C
# configure the I2C bus
i2c = I2C(0, I2C.MASTER, baudrate=100000)
i2c.scan() # returns list of slave addresses
i2c.writeto(0x42, 'hello') # send 5 bytes to slave with address 0x42
i2c.readfrom(0x42, 5) # receive 5 bytes from slave
i2c.readfrom_mem(0x42, 0x10, 2) # read 2 bytes from slave 0x42, slave memory 0x10
i2c.writeto_mem(0x42, 0x10, 'xy') # write 2 bytes to slave 0x42, slave memory 0x10
```



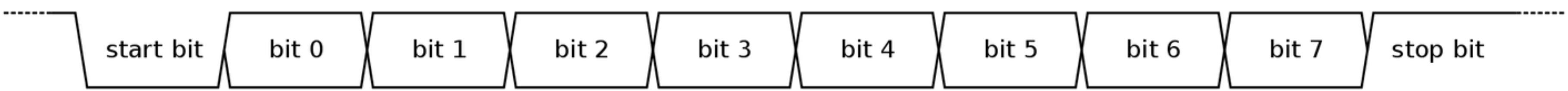


Sensors / Communication in embedded devices / SPI

- The Serial Peripheral Interface (**SPI**) is a synchronous serial communication interface specification used for short distance communication, primarily in embedded systems (Motorola, 1980s). 4 wires, full duplex.

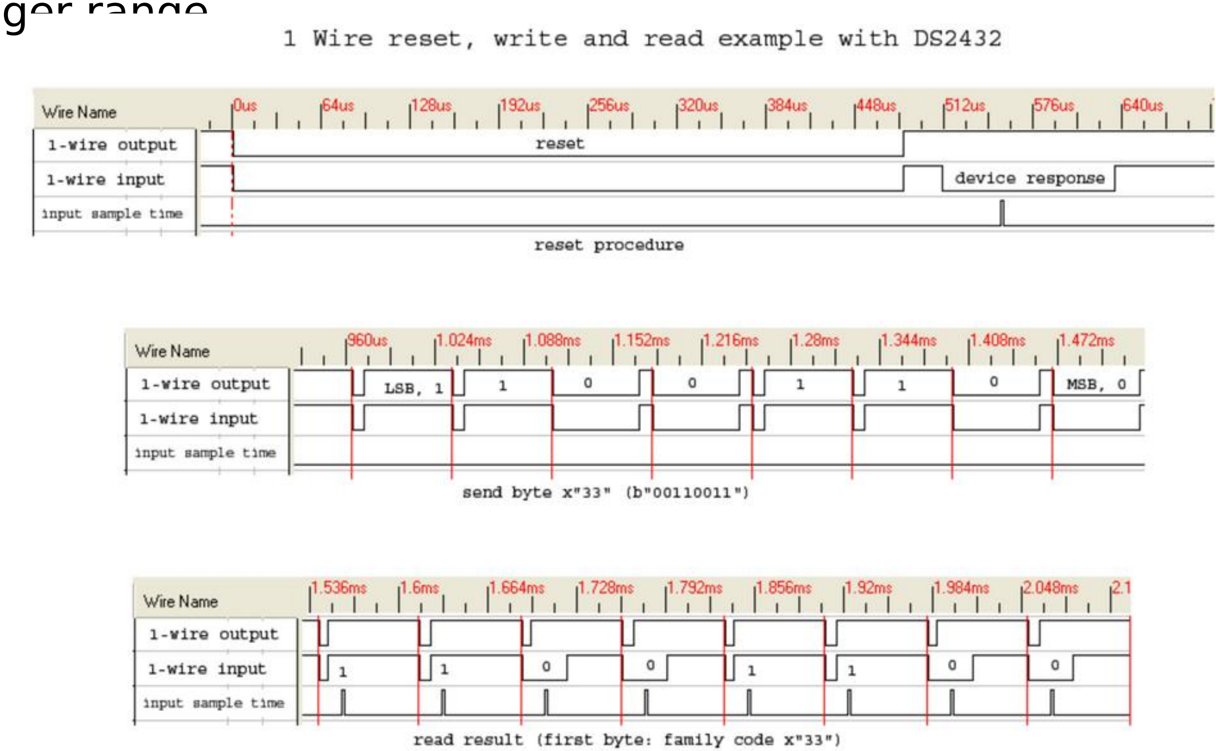


- A **universal asynchronous receiver-transmitter (UART**) is a computer hardware device for asynchronous serial communication in which the data format and transmission speeds are configurable. It sends data bits one by one, from the least significant to the most significant, framed by start and stop bits so that precise timing is handled by the communication channel. It was one of the earliest computer communication devices, used to attach teletypewriters for an operator console. It was also an early hardware system for the Internet. The electric signaling levels are handled by a driver circuit external to the UART. Two common signal levels are RS-232, a 12-volt system, and RS-485, a 5-volt system. .



Sensors / Communication in embedded devices / 1-Wire

- **1-Wire** is a device communications bus system designed by Dallas Semiconductor Corp. that provides low-speed (16.3kbps) data, signaling, and power over a single conductor (+ ground). Similar in concept to I²C, but with lower data rates and longer range



Accessing Embedded Systems

Physical:
Serial connections / USB
Wi-Fi (don’t!)
Bluetooth
NFC

Generally via IDE

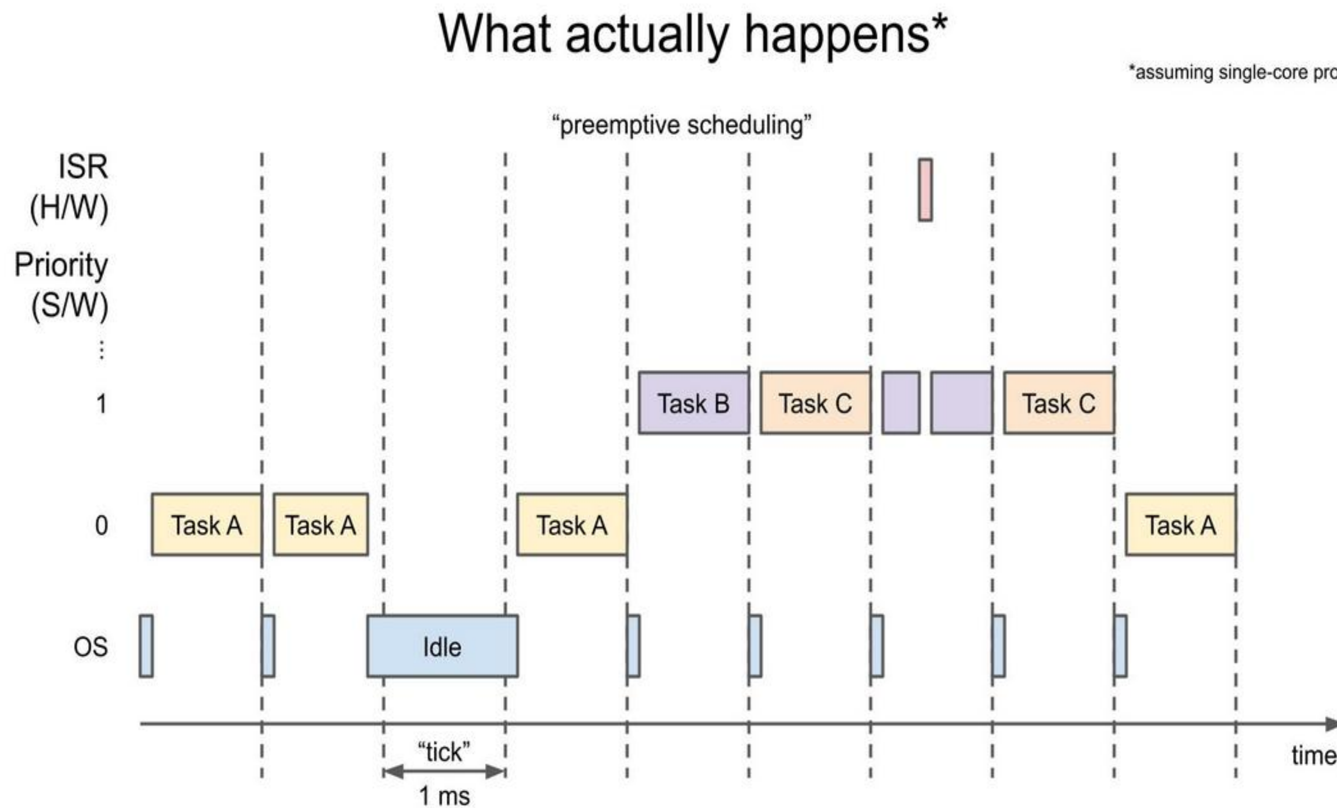
Hardware specific libs

(changeover to Robert Bayer)

Real-time Operating Systems (RTOS)

- Used to fullfil critical time constraints
- Main building block of operating system == Scheduler
- General-purpose OS
  - For example: Linux kernel
  - Time-sharing scheduler => switching between tasks in regular time intervals)
- RTOS
  - Tasks are assigned priorities
  - Event-driven scheduler => tasks switched only when task with higher priority needs to be executed

RTOS – Scheduling

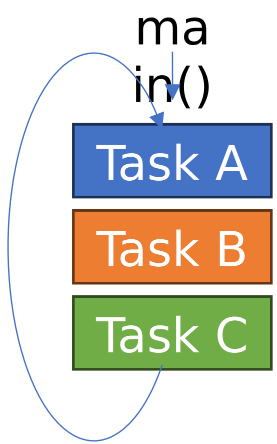


Source: Digikey – Introduction to RTOS (task scheduling)

Bare-metal vs RTOS

Bare-metal

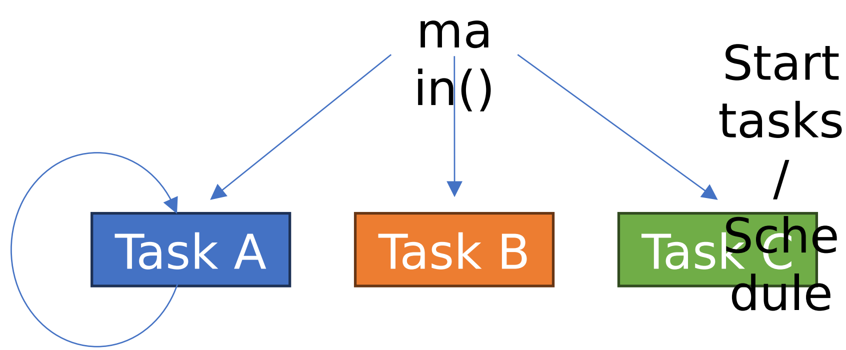
Super loop



The only way to jump between tasks out-of-order are interrupts

RTOS

Spin up tasks, execute them based on priority



Loop until interrupted

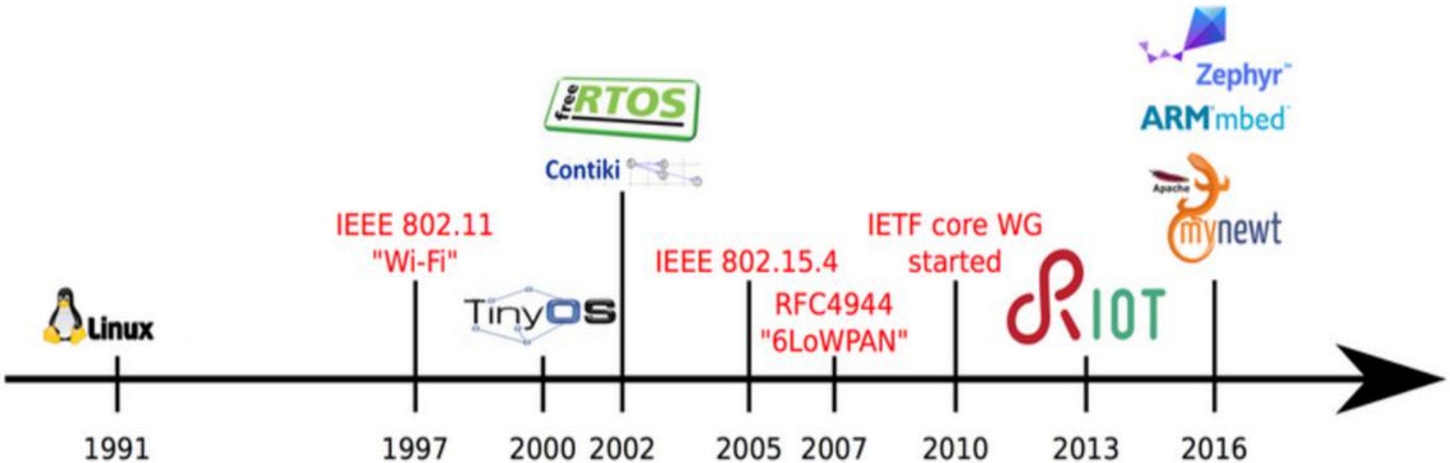
FreeRTOS

- Most widely used RTOS (downloaded every 170 seconds)
- Lightweight - 6-12 KB typically, core kernel contained in 3 C files
- Besides scheduler:
  - Intertask communication
    - Queues, mutexes and semaphores
    - Notifications
    - Streams and buffers
  - Opt-in libraries
    - Networking (TCP, HTTP, MQTT, SNTP, JSON, LoRaWAN, etc.)
    - Filesystem (FAT, I/O)
    - Many more community-driven libraries

FreeRTOS - example

```c
static const int led_pin = 10;

void toggleLED_1(void *parameter) {
  while(1) {
    digitalWrite(led_pin, HIGH);
    vTaskDelay(500 / portTICK_PERIOD_MS); // 500ms on
    digitalWrite(led_pin, LOW);
    vTaskDelay(500 / portTICK_PERIOD_MS); // 500ms off
  }
}

void setup() {

  // Configure pin
  pinMode(led_pin, OUTPUT);

  // Task to run forever
  xTaskCreatePinnedToCore(  // Use xTaskCreate() in vanilla FreeRTOS
              toggleLED_1,  // Function to be called
              "Toggle LED",   // Name of task
              1024,         // Stack size (bytes in ESP32, words in FreeRTOS)
              NULL,         // Parameter to pass to function
              1,            // Task priority (0 to configMAX_PRIORITIES - 1)
              NULL,         // Task handle
              0);     // Run on one core for demo purposes (ESP32 only)
}

void loop() {
    vTaskStartScheduler();
}
```

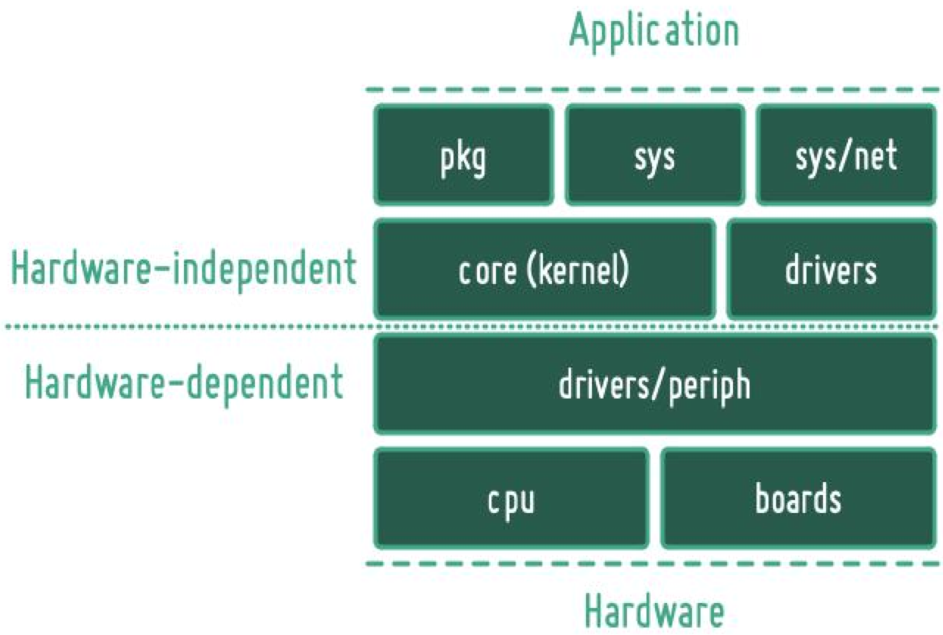
Alternatives



Hahm, O., Baccelli, E., Petersen, H., & Tsiftes, N. (2015). Operating systems for low-end devices in the internet of things: a survey. IEEE Internet of Things Journal, 3(5), 720- 734.

RIOT OS

- Inspired by Linux
- Micro-kernel architecture
- Supports multi-threading
- More complete OS in comparison to FreeRTOS
- Strong focus on networking / IoT
    - Out-of-the-box support for different networking stacks (UDP, TCP, LoRaWAN, MQTT)



Source: RIOT OS documentation

Embedded Linux

- Can make use of all of the software already written and used on desktops
- Highly customized
  - Stripped of everything non-essential to your application
- Examples
  - Yocto – toolchain for building custom Linux
  - OpenWrt – customizable Linux framework (built for routers)
- Has support for preemptive execution (still more relaxed than RTOS)
- Much larger footprint – only feasible on larger systems (10s of MB)

Python for microcontrollers

- MicroPython and CircuitPython
- Serves mainly for education purposes
- Based on Espressif-IDF (FreeRTOS)
- High level of abstraction
- Read evaluate print loop (repl)
  - Run python interactively in shell

Discussion

Pros / cons of using Python on embedded devices?

MicroPython example

```
import pycom
import time

pycom.heartbeat(False)

while True:
    pycom.rgbled(0xFF0000)  # Red
    time.sleep(1)
    pycom.rgbled(0x00FF00)  # Green
    time.sleep(1)
    pycom.rgbled(0x0000FF)  # Blue
    time.sleep(1)
```

MicroPython – Running scripts

- Deployment using 2 files
  - boot.py
    - Executes once at the start
    - Like setup() function
    - Set up pins
    - Initialize libraries
  - main.py
    - Main loop of your system

Discussion

Bare-metal vs RTOS vs embedded OS

Terms

ADC	Analog to Digital Converter
ASIC	Application specific Integrated Circuit
CPU	Central Processing Unit
CRC	Cyclic redundancy check
FPGA	Field-programmable Gate Array
I^2C	Inter-Integrated Circuit
MCU	Micro Controller Unit
MMU	Memory Management Unit
MOS	Metal Oxide Semiconductor
MPU	Memory Protection Unit
RAM	Random Access Memory, static SRAM, dynamic DRAM
REPL	Read-Eval-Print Loop
RF	Radio Frequency
SoC	System on chip
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver-Transmitter
ULP	Ultra Low Power
Pinout	Map of pin functions and connections

Take aways

- Embedded system - term, definition, history
    - Firmware vs OS
- Constraints
- Elements of embedded systems
- Examples of embedded systems
- *How to work with embedded systems*
    - *interfacing*
    - *programming*
    - *sleeping :)*