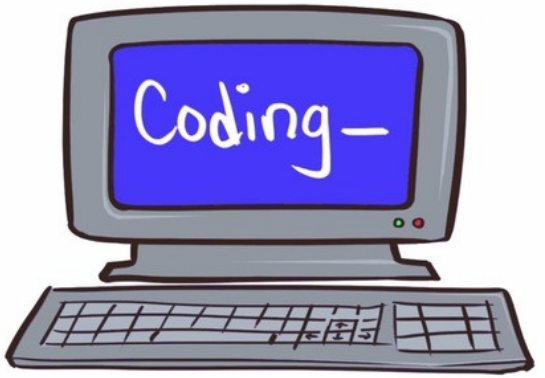
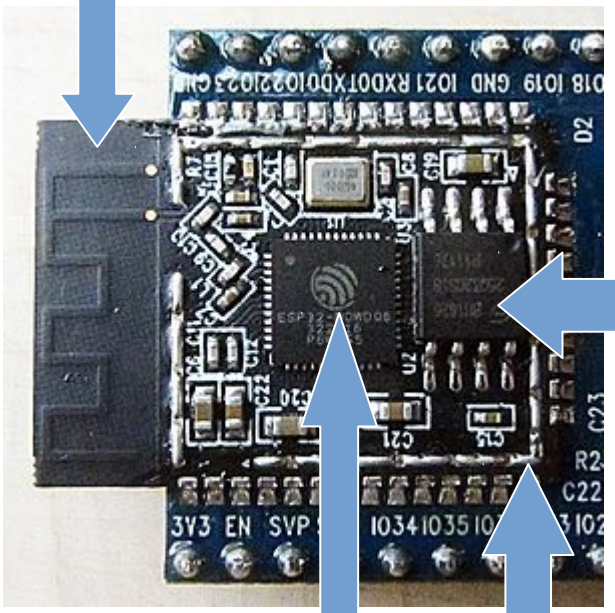
Class 44



ESP32

From Espressif

2.4GHz Antenna

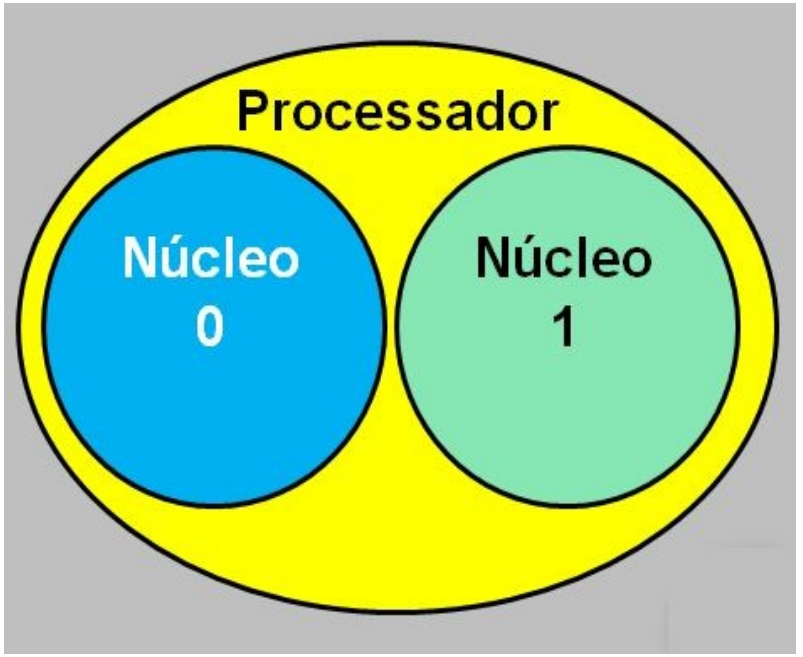


Flash memory

Tensilica Xtensa
LX6 uprocessor
32-bit
160-320MHz

Edge of metal
RF shield
(removed)

2 Core LX6 processor
(Portuguese anyone?)



Core 0 and Core 1

ESP32/Arduino platform - single core, single “task”

- Wifi & other inbuilt functions operate on Core 0
- Arduino code operates (concurrently) on Core 1

- By default ESP32/arduino code is single task/thread. (Just like any other arduino hardware)

Manufacturer's code

Manufacturer Espressif (China) provide

1)“bios” code for all chip functions,

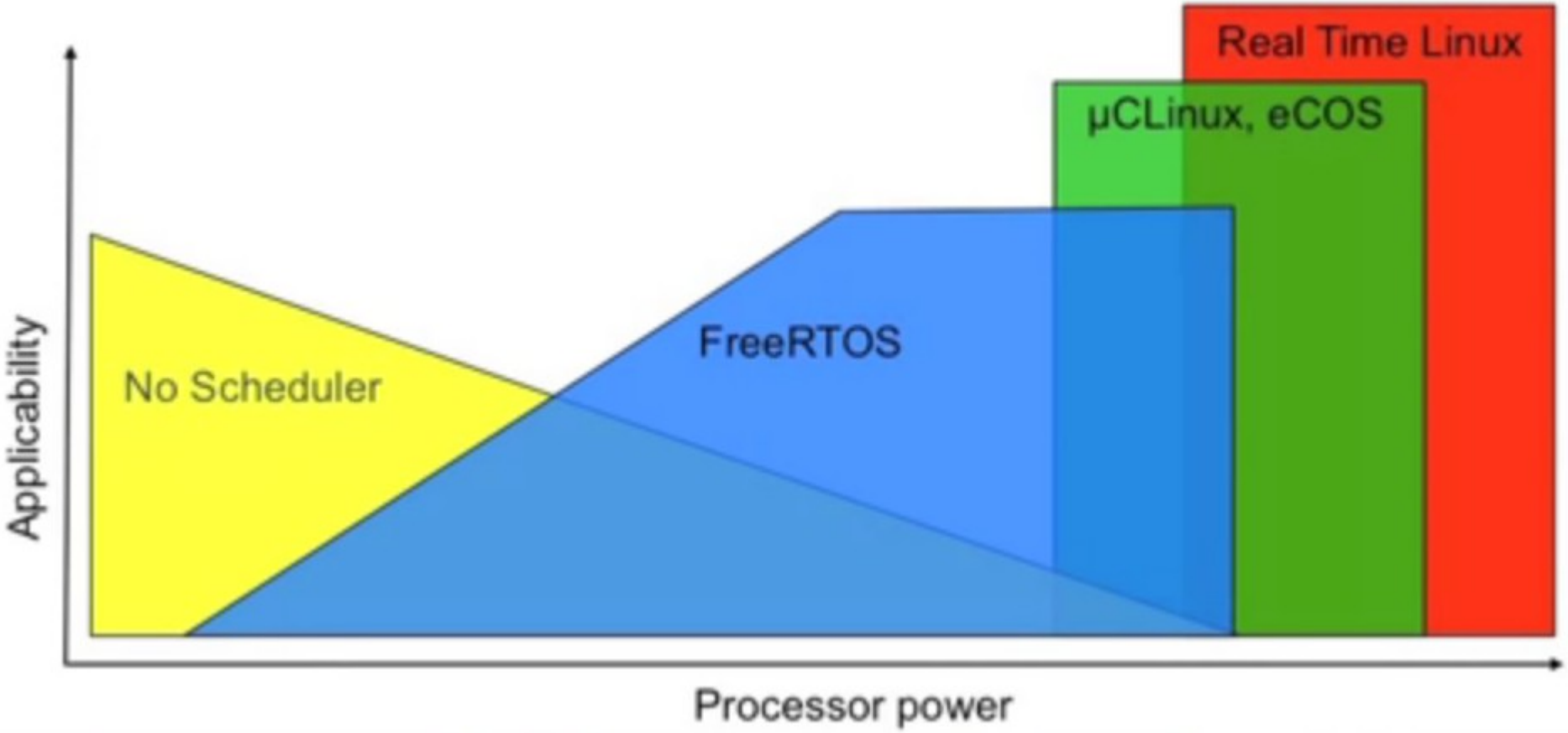
2)Well documented API to that code

3)Customised open-source of FreeRTOS for ESP32

4)Arduino IDE code

FreeRTOS & other rtos

- 33 architectures and 18 tool chains



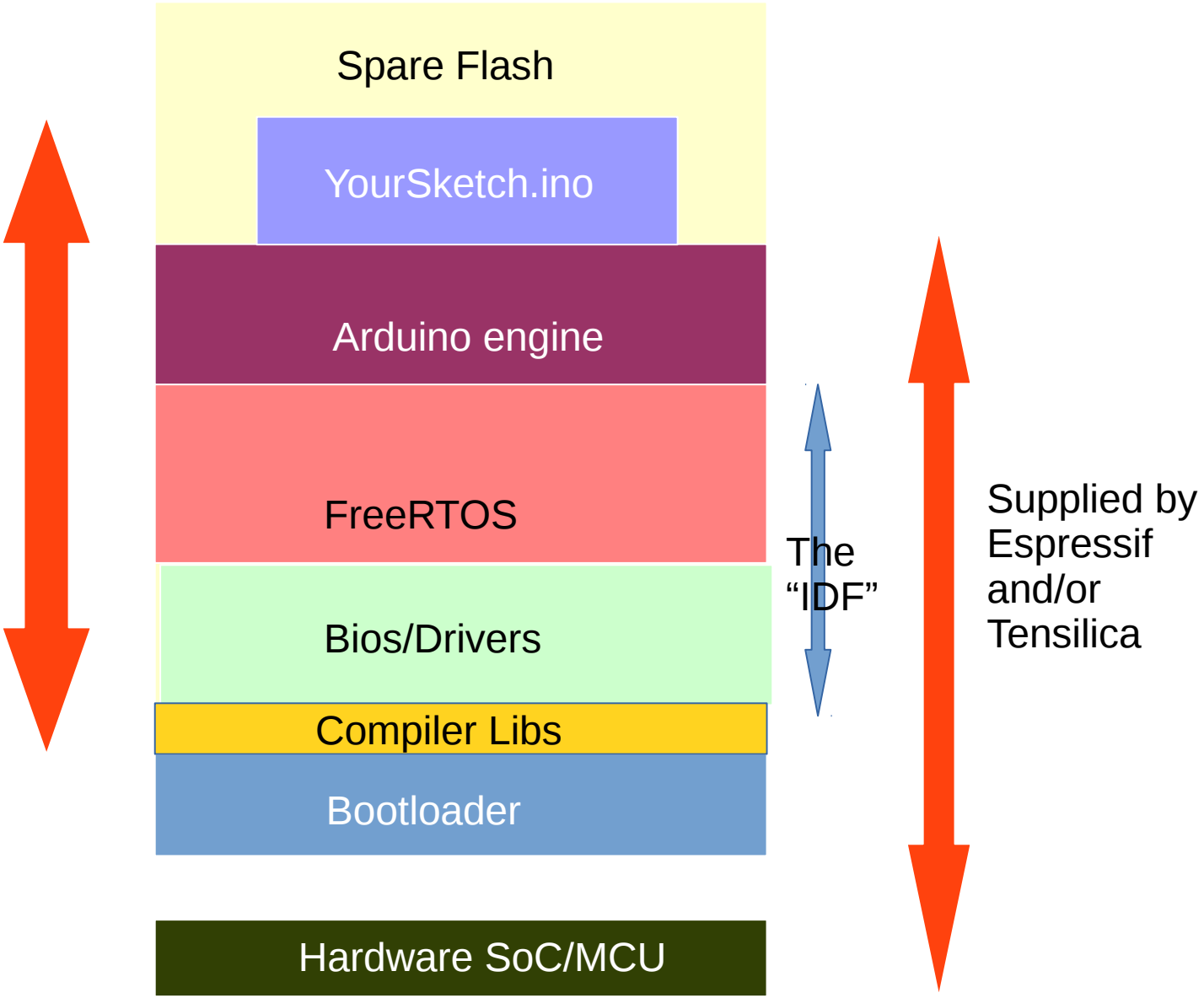
Some of the many others:
ctask22 (dos era) uC/os (by Micrium) chibios (on arduino) uCLinux windows CE

“Compiling takes a long time ..”

Every compile rebuilds all this as a download image.

The “compiler” comes from Tensilica.

Functions in Bios or FreeRTOS or Arduino not needed by your sketch are omitted during compiling.

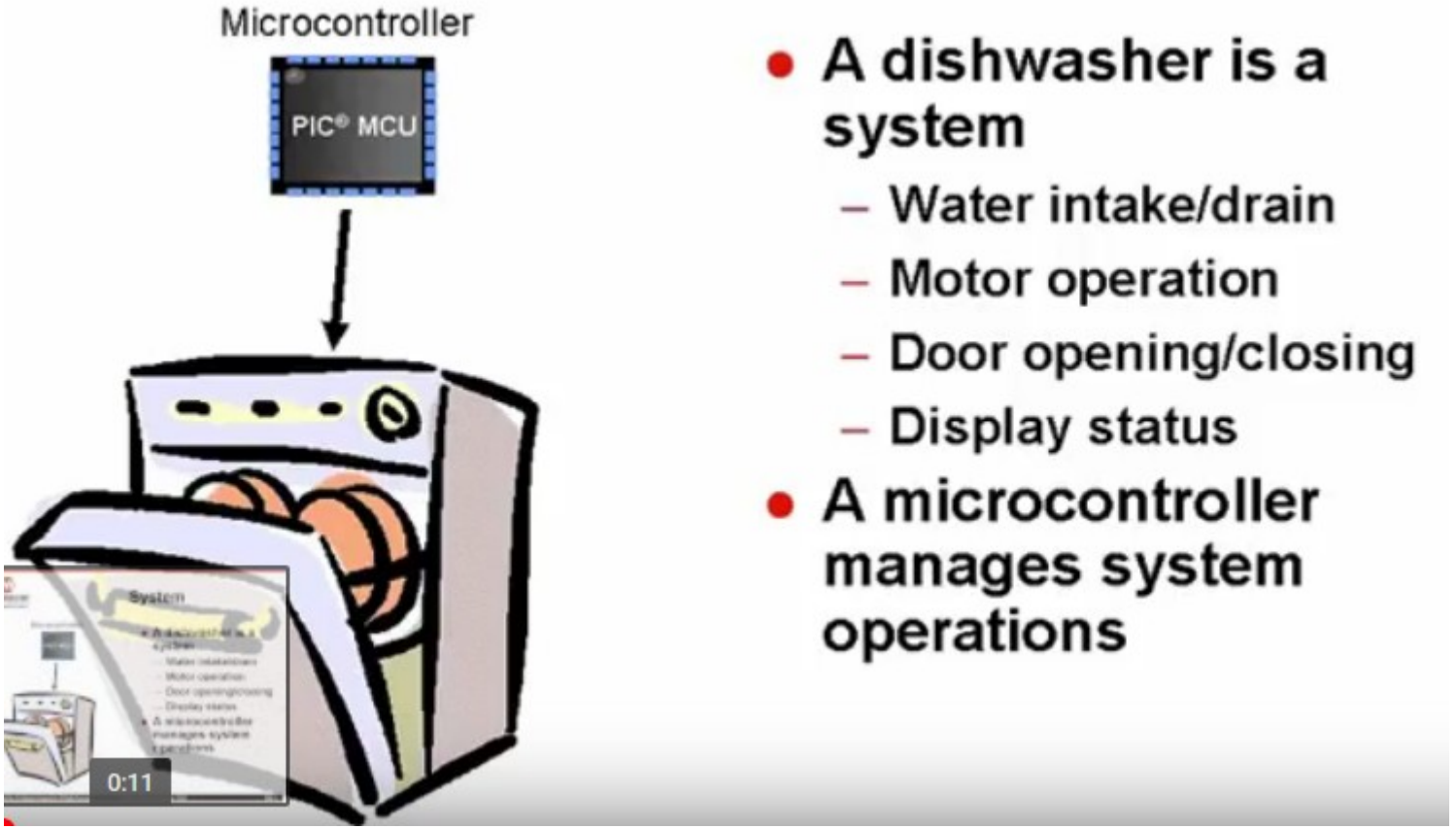


FreeRTOS

- RTOS = Real-time Operating System
- Used on mid-size microcontrollers
- Every interrupt/event has a guaranteed worst-time for being serviced.

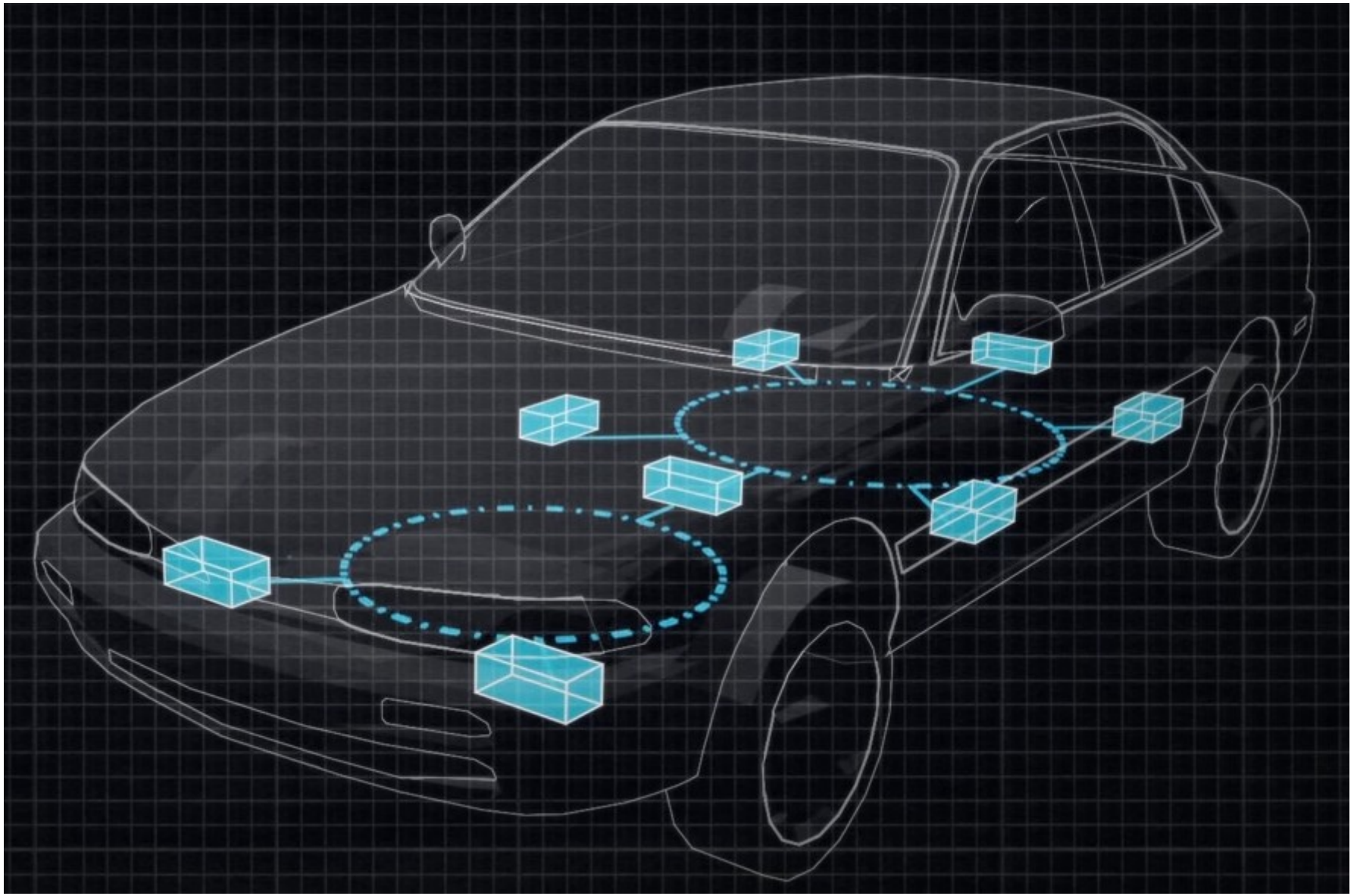
- Linux & Windows are NOT “real-time”
  even if a fast machine might “feel” snappy.

What is an RTOS?



- A dishwasher is a system
  - Water intake/drain
  - Motor operation
  - Door opening/closing
  - Display status
- A microcontroller manages system operations

Car Computer = RTOS



FreeRTOS on our esp32

The code from manufacturer (Espressif) is **inherently** built using FreeRTOS.

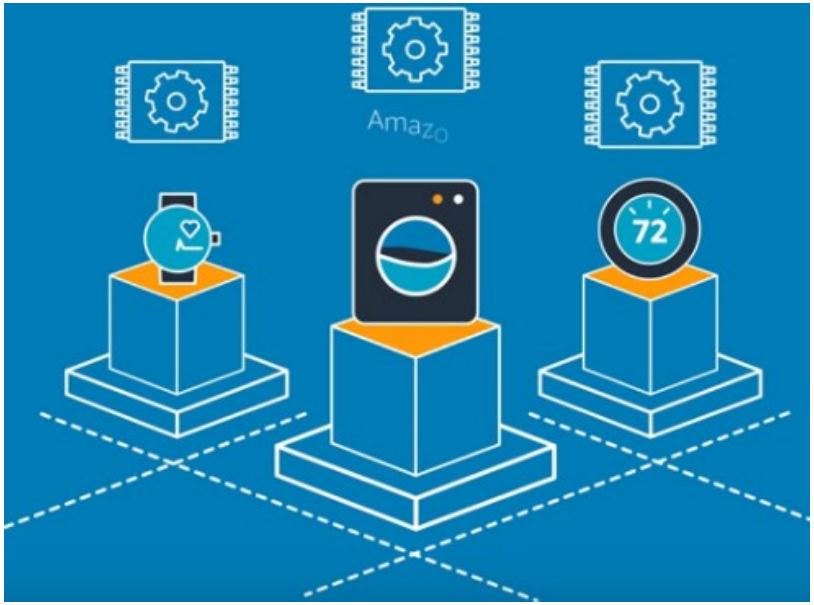
However you compile some code for the ESP32,

- Arduino?
- Micropython?
- Raw C/C++?

… it still is running FreeRTOS engine.

AWS

- FreeRTOS is recently taken over by Amazon Web Services
- It remains open source.



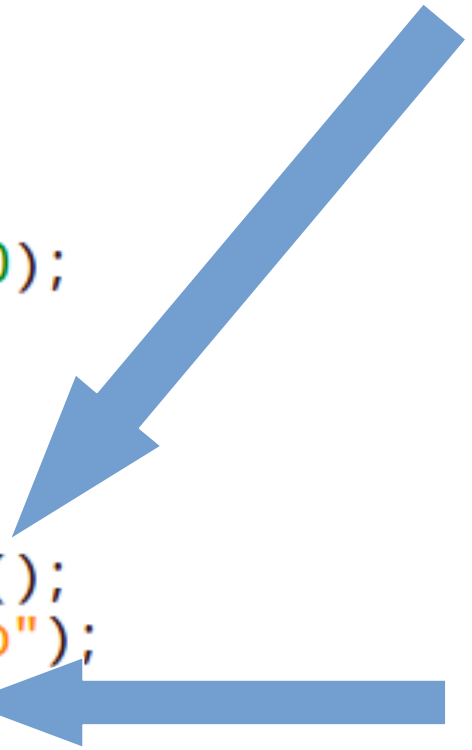
The arduino model - setup() & loop()

```
#include "lib2.h"


void setup() {
    Serial.begin(9600);
}

void loop() {
    readKeyboard();
    quadencoder1.run();
    Serial.print("boo");
    delay(1000);
}
```

“Round Robin”
or “Executive Loop”
controls repeating
sequence of function calls.

The arduino model – the “delay” problem

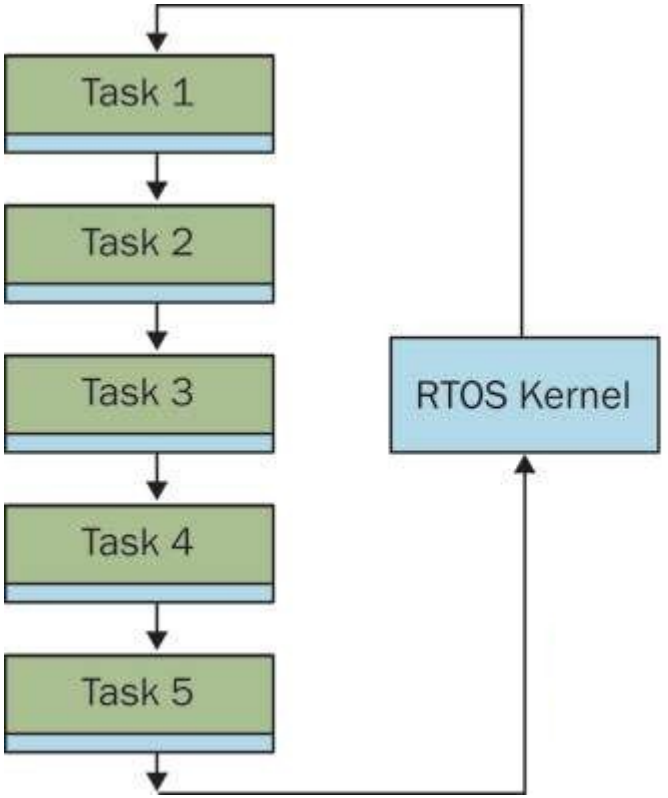
```
#include "lib2.h"


void setup() {
    Serial.begin(9600);
}

void loop() {
    readKeyboard();
    quadencoder1.run();
    Serial.print("boo");
    delay(1000);
}
```



Very fast “.run()” calls are one attempt to avoid delays in any area of code.

Any delay() in loop() or in any called function, will suspend all other processing.

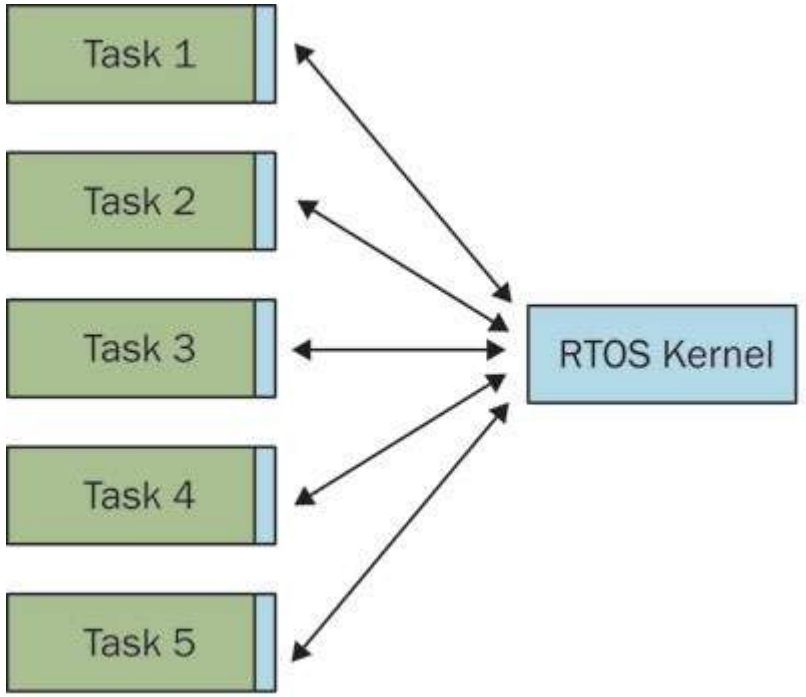
The “kernel” - sequencing the jobs to be done.



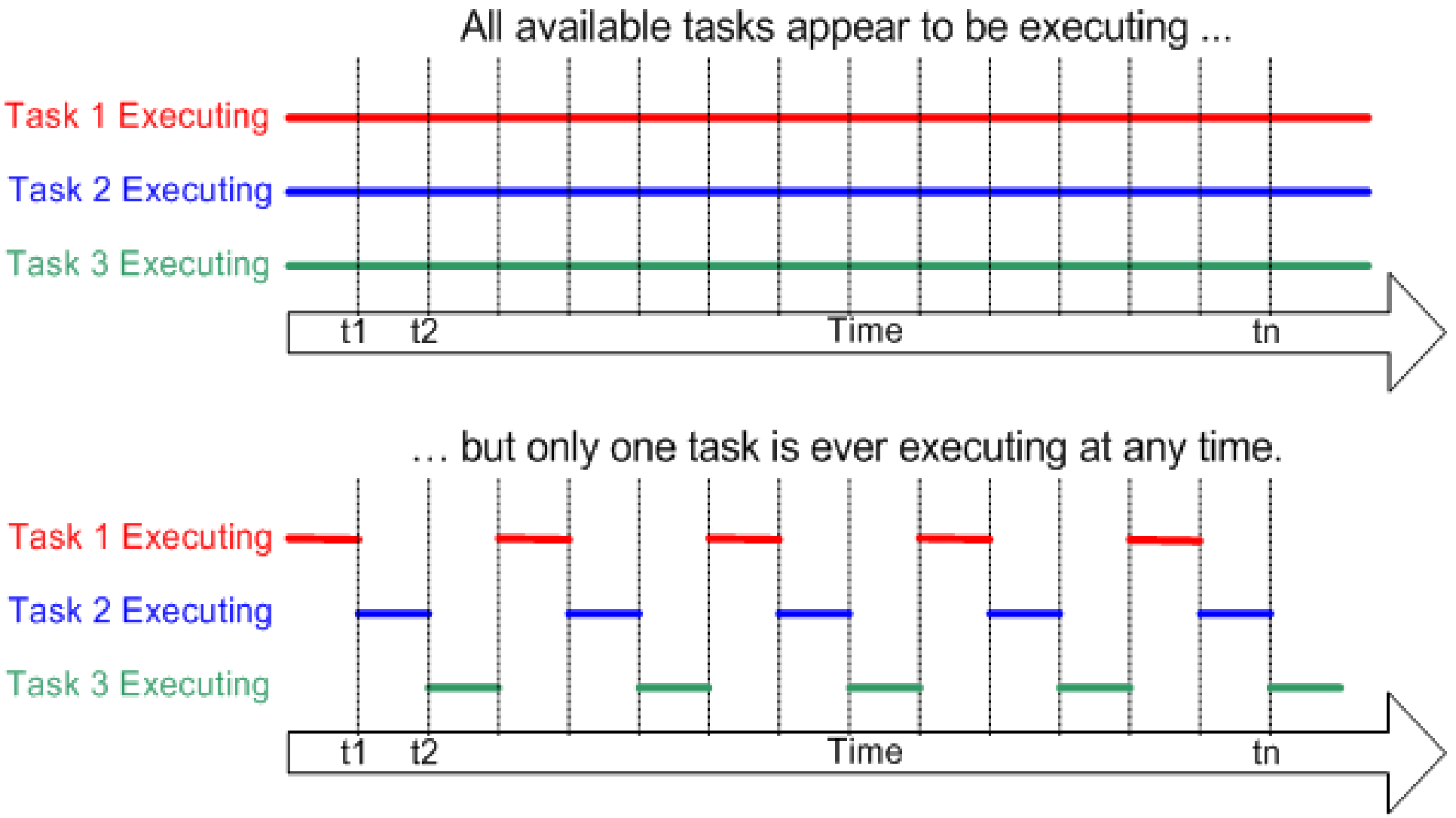
For arduino, our “kernel” is trivialised to a function call list in loop().

In our arduino single thread model, all those calls are really the one “task” In the sense used by RTOS people (the “embedded software engineers”).

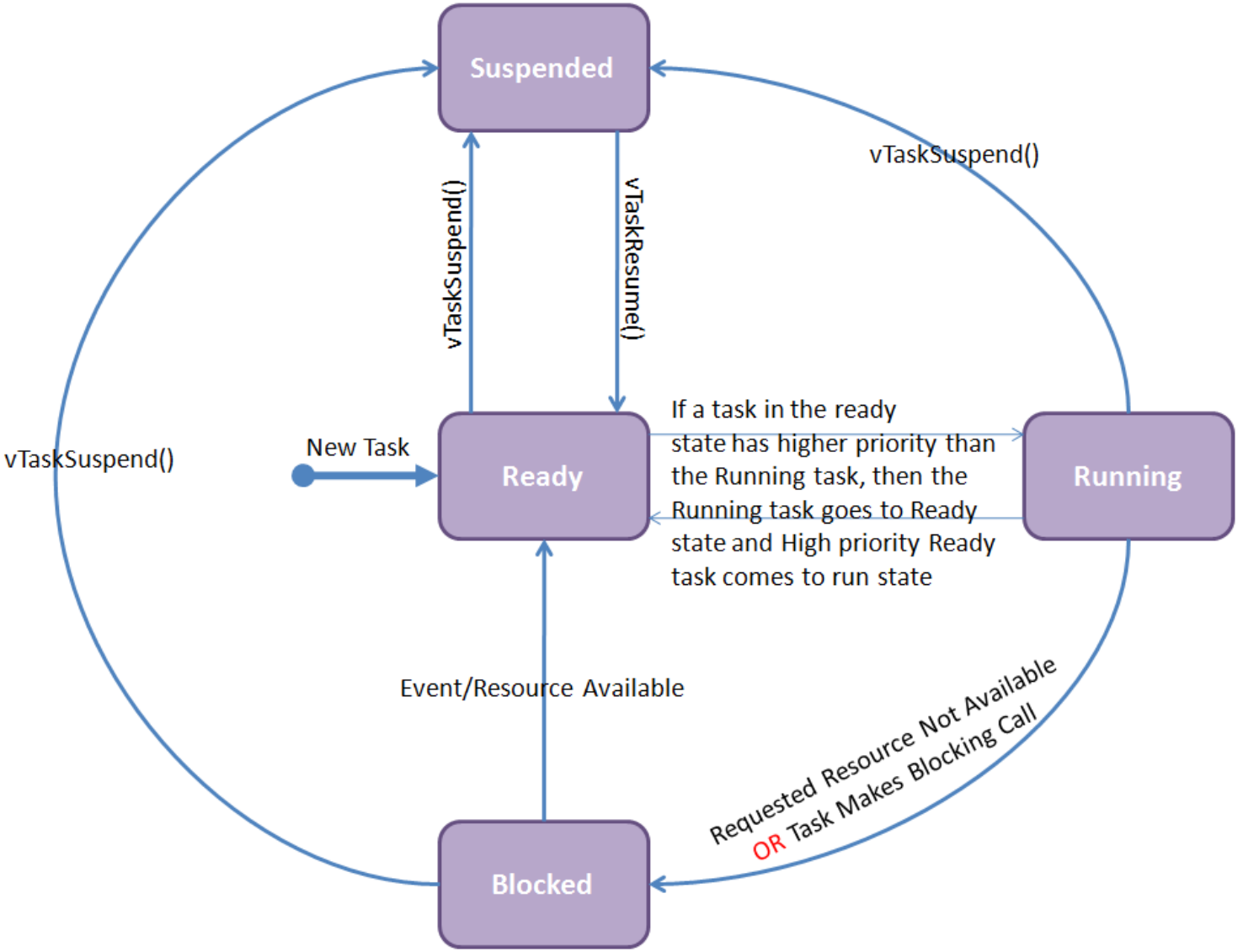
A more general “kernel” has smarter ways to decide what job gets cpu time



“time-slicing” for “concurrent” tasks



Any task may be running or paused



FreeRTOS allows concurrent tasks

Each “task” is an isolated specific-job function.
Tasks are frequently created to run “forever”.

COOPERATIVE MULTITASKING

Each task once started is expected to “yield()” (very) frequently, Permitting the kernel to assign the CPU to the task with highest priority just now.

So yield() is a bit like delay() except it allows another task to run. (But you may be unsure when CPU is returned to you!)

delay() in esp32/arduino actually yields to allow other (properly created) tasks use the time. (Our simply loop() “round robin” is NOT a list of real “tasks”.)

FreeRTOS allows concurrent tasks

Each “task” is an isolated specific-job function.
Tasks are frequently created to run “forever”.

PRE-EMPTIVE MULTITASKING

The kernel assigns time-slices for each to “run”.

When kernel reassigns the CPU to a second task,
the first task is suspended just where it was,
and resumes (as though not interrupted) when it next gets CPU time.

Premptive!

- We can use freeRTOS calls in our sketches.
- **Pre-emptive** multitasking is used on Arduino/ESP32. You don't need to consider just how the kernel slices up the CPU to the tasks.

Sketch: Tasks.ino

Tasks need to communicate !

Scenario:

- Task 1: Continually re-writes a variable V1 that has several bytes of memory
- Task 2: Reads variable V1 and prints its content to serial window.

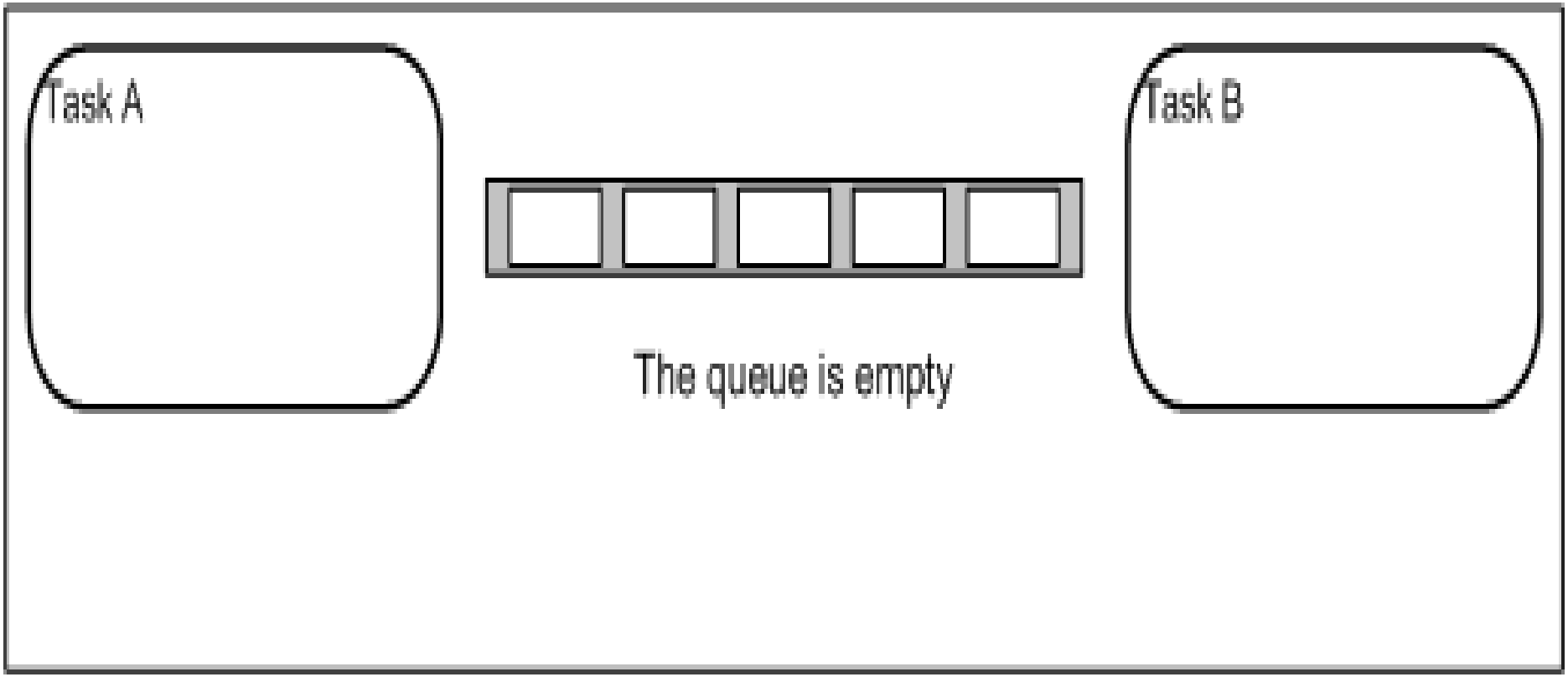
THE CLASSIC BIG ISSUE OF PREEMPTION:

- What does task 2 read if task1 was HALF-WAY THROUGH writing a new V1?

Tasks need to communicate !

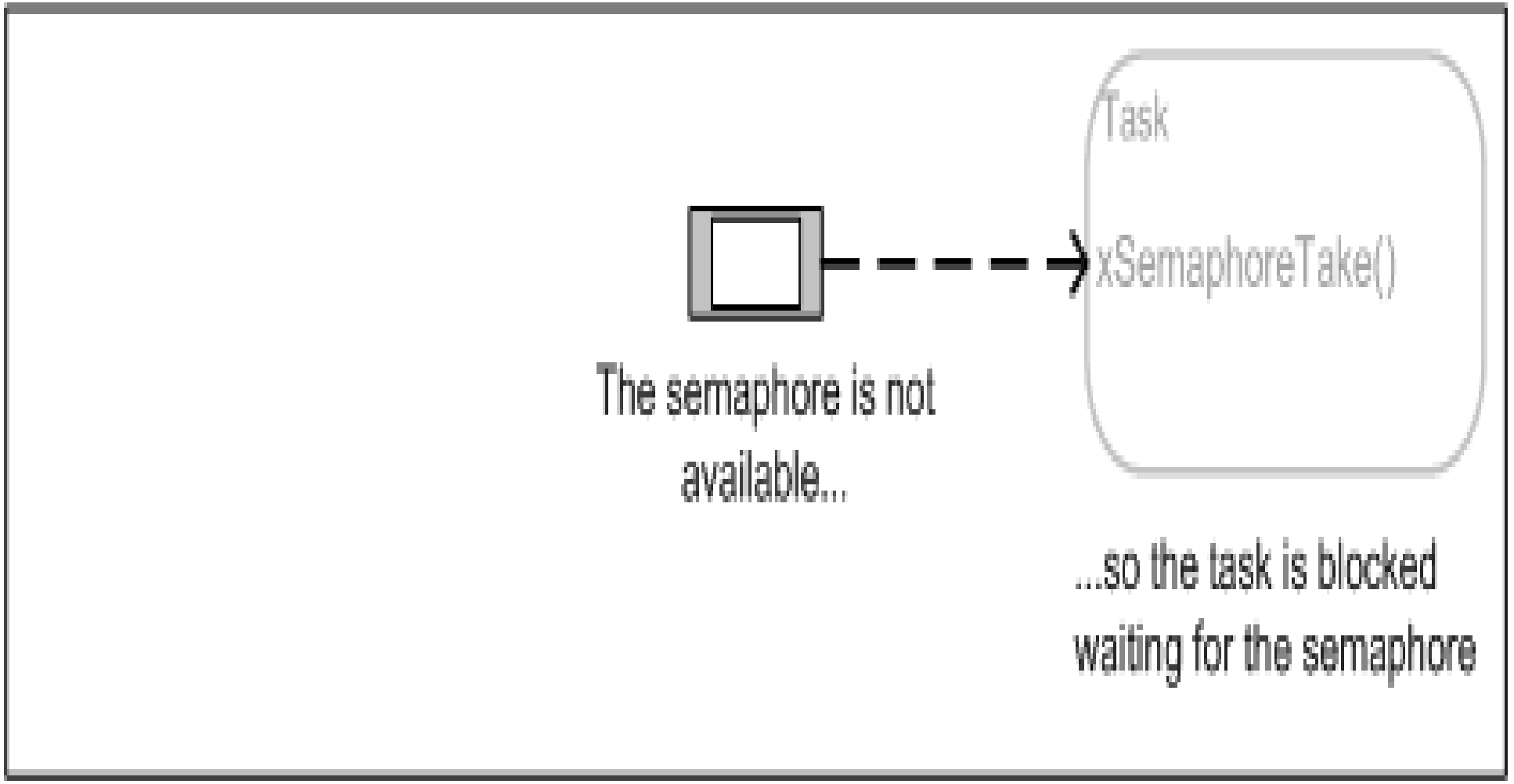
1. **Queue / message:** Task1 posts a package of data, and task2 collects it. (eg bit like email)
   The queue may have room for multiple messages still uncollected.
2. **Semaphore:** Transferred like a message, but it is just a single “flag”, ie no other data content.
   Good for synchronising actions in 2 tasks. (eg, I'm finished, are you? Or, An event just happened. Letting you know.)
3. **Mutex:** “Mutual Exclusion. A semaphore used to allow just one task to get access to something.
   (Ever requested the one toilet key at a restaurant?)

Message



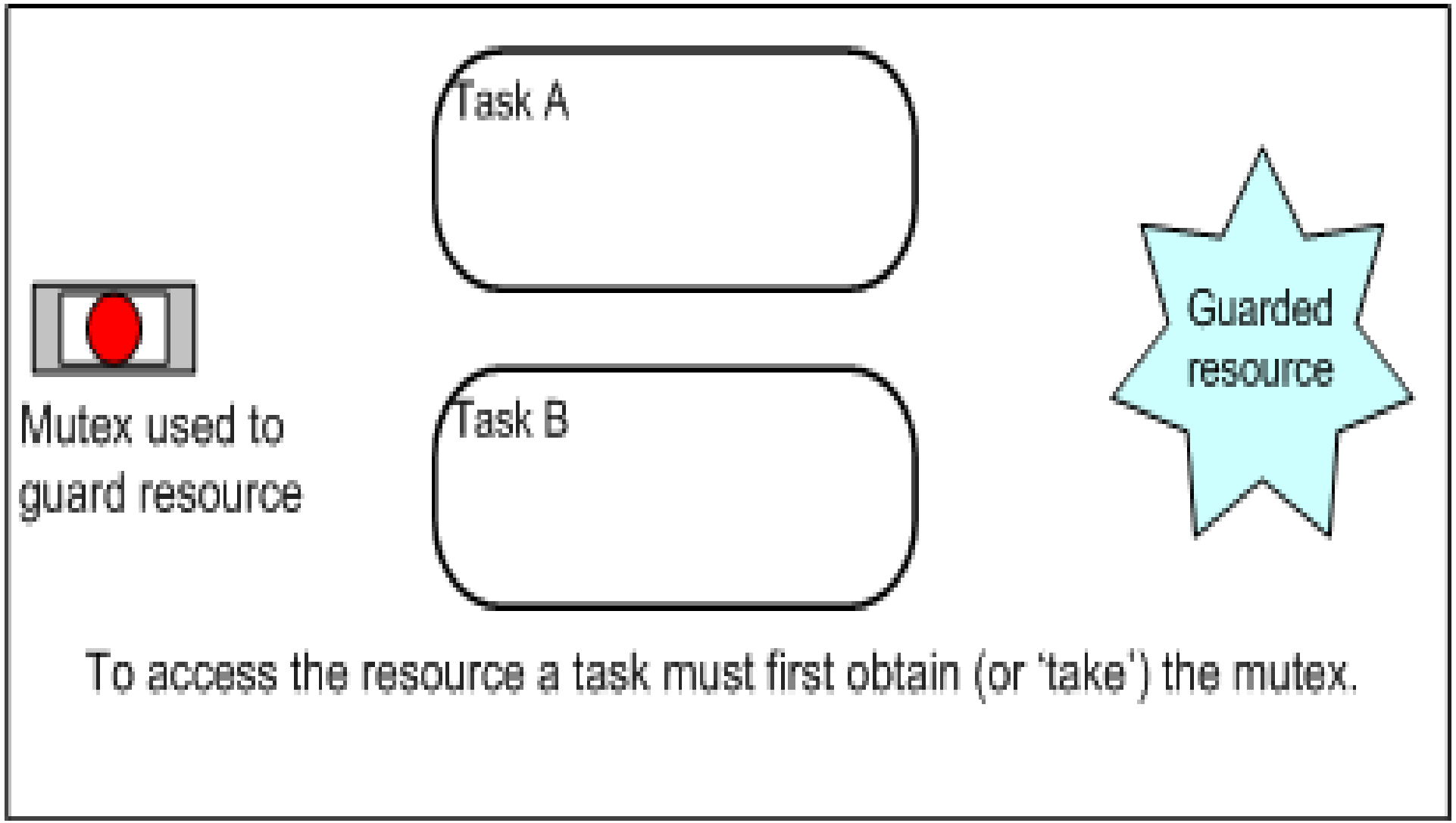
Sketch: TaskPassMsg.ino

Semaphore



Semaphores often have a max queue of 1, but not always

Mutual Exclusion



A demo of data corruption

Sketch: corruptionDemo.ino

Task2 repeatedly rewrites a string (“str”)

Task1 repeatedly reads str, and checks if it is a complete correct sequence. Or was it caught half-rewritten at read time?

Had the write been pre-empted in mid-write, and then read before the write was finished off?

Message passing between tasks

Sketch: taskPassMsg.ino

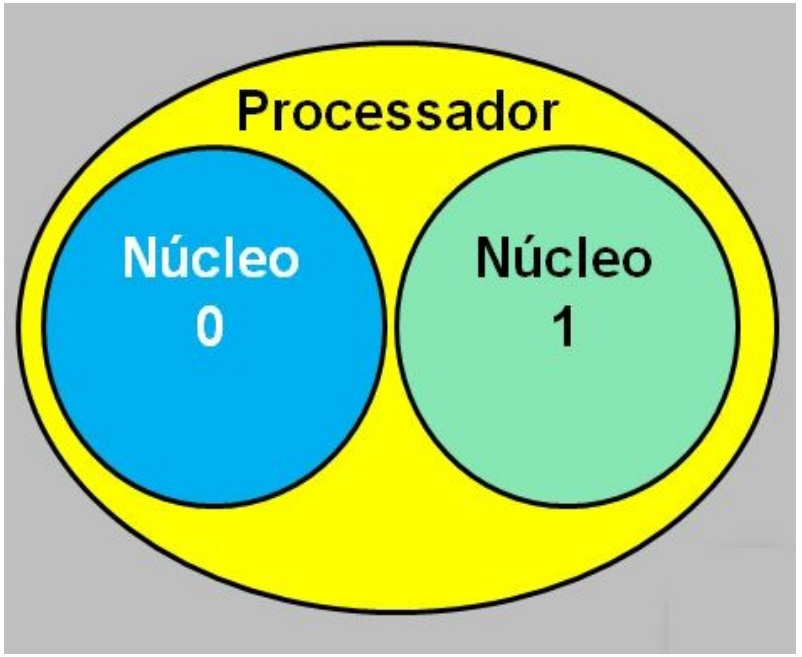
Task1 (“producer”) sends data to queue

Task2 (“consumer”) takes data from queue

Queue array needs pre-creating:

- Max number of messages?
- Max size each message?

Corruptionless transfer !



Cores

So far, this has all been happening on Core 1

But we know that there is also a Core 0,
lightly loaded with wifi and other inbuilt functions.

The FreeRTOS functions will let us put tasks on Core 0 as well.
Core 0 is NOT time-sliced with Core 1 – it runs full-time independently!

Two Cores in Use:

- **Sketch: TaskTwoCore.ino**

  - the main task (ie our familiar loop()) on Core1
  - and our new task on either Core0 or Core1
  (edit line 7 to select core 0 or 1)

  New task blinks the led.

SoftwareTimers – common on RTOS

Simple timer:

Set a time delay.

Start clock.

Interrogate periodically?

If time reached, do xxx()

Timer might be repeating?

Callback timer:

Set a time delay.

Nominate a task/function xxx to timer.

Start clock.

When time reached, xxx() starts automatically

Timer might be repeating?

Recall the ESP8266 Lua used callback timers extensively (in a single-threaded mode).

FreeRTOS functions

https://esp-idf.readthedocs.io/en/v3.1-rc1/api-reference/system/freertos.html