Task Oriented Programming for the Internet of Things

Mart Lubbers Pieter Koopman Rinus Plasmeijer

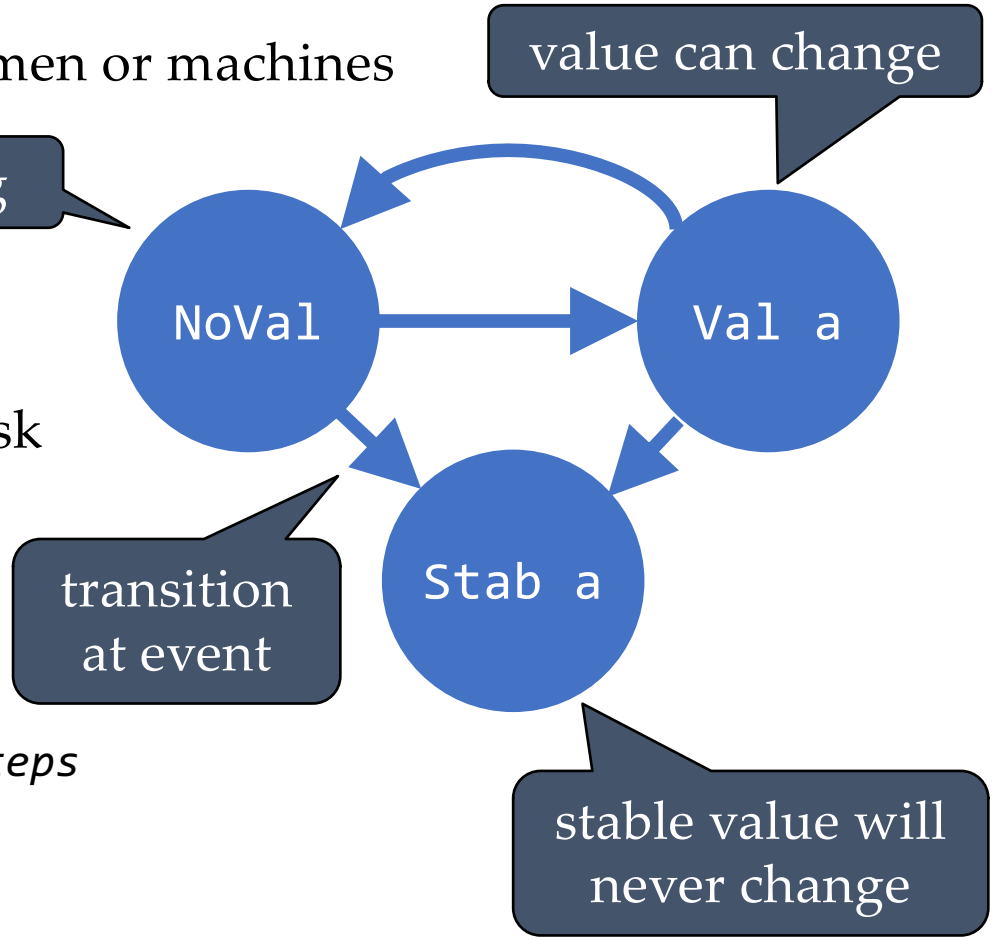
3COWS/CEFP summer school Budapest, June 2019

Radboud University

Task Oriented Programming for the Internet of Things

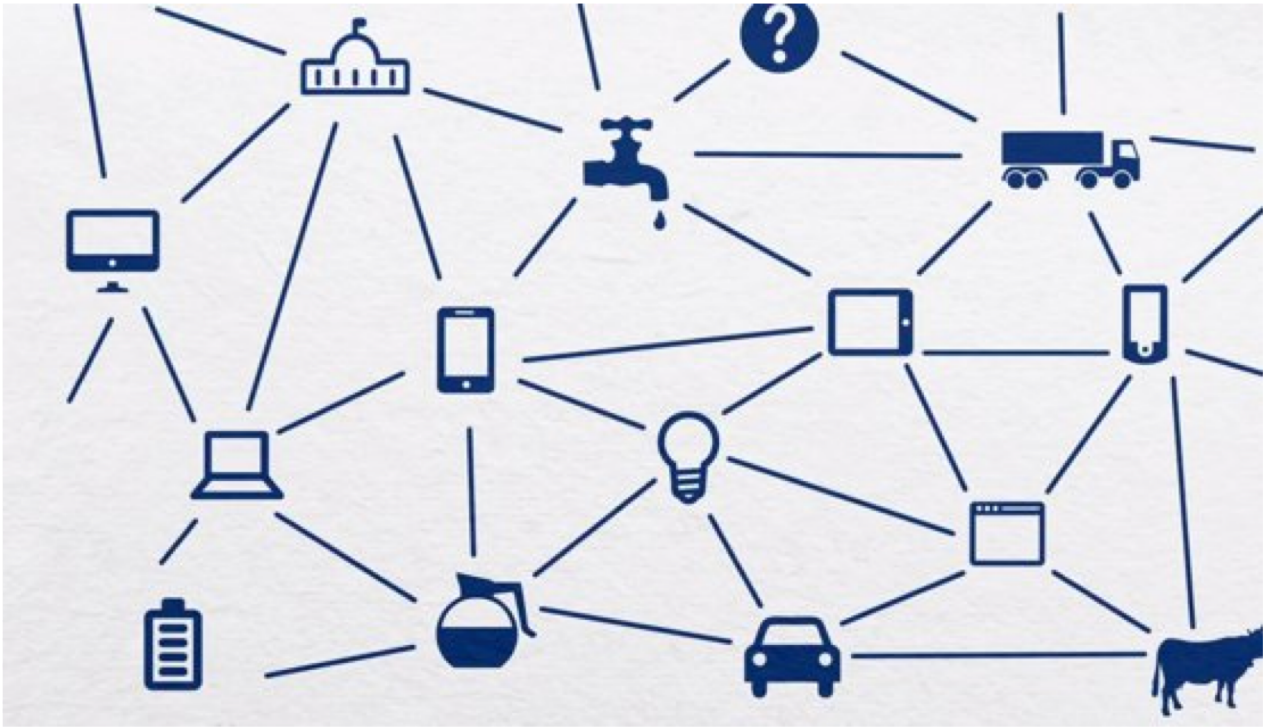
- task are pieces of work to be executed by men or machines

- elementary tasks
  - edit a value in web editor, ..
  - read a digital pin, temperature sensor, ..
- combinators: compose tasks to a bigger task
  - parallel, choice: `t1 -||- t2`
  - parallel, both: `t1 -&&- t2`
  - sequential, no result: `t1 >>| t2`
  - sequential, transfer result: `t1 >>= f`
  - step `t >>* list_of_steps`

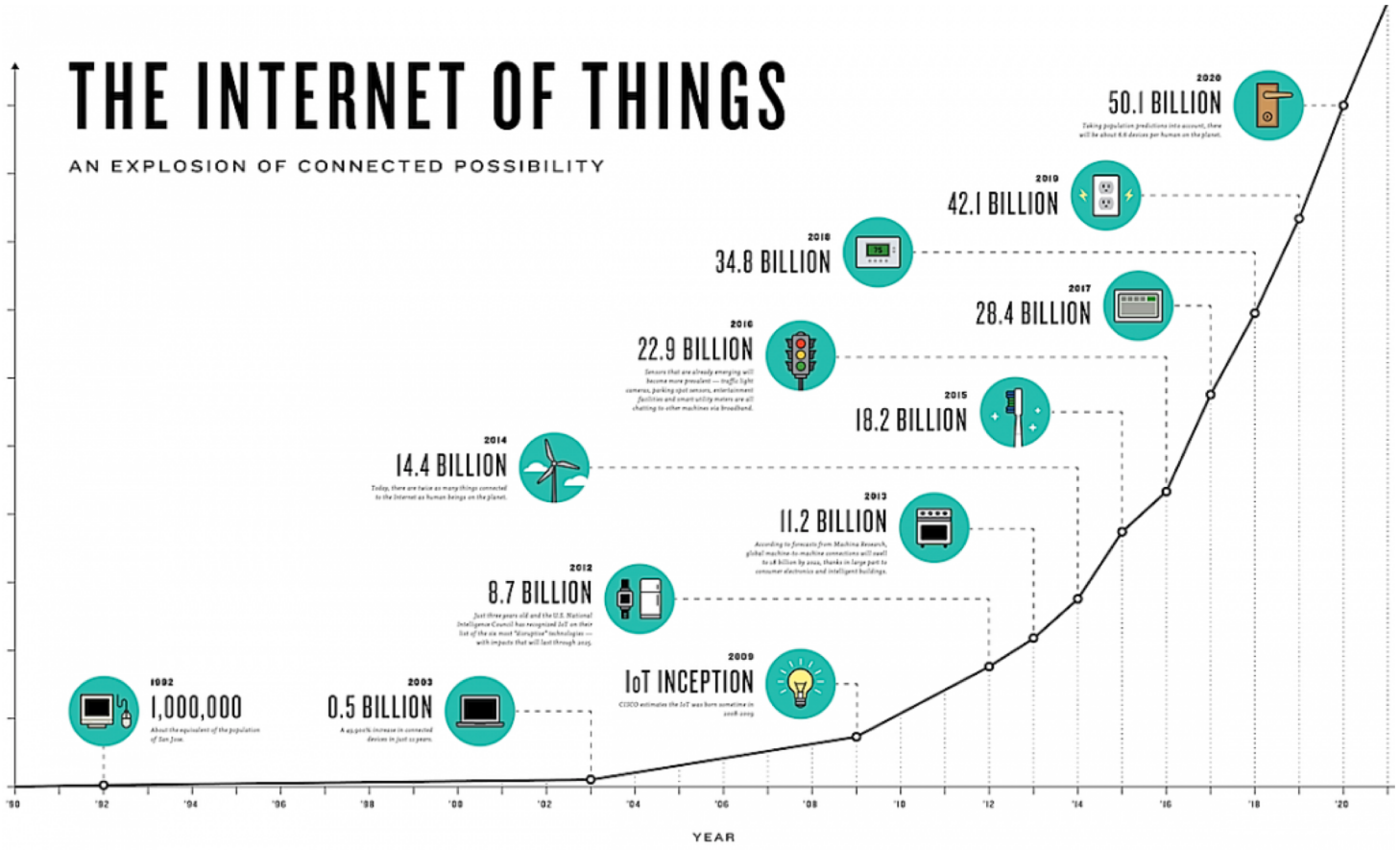


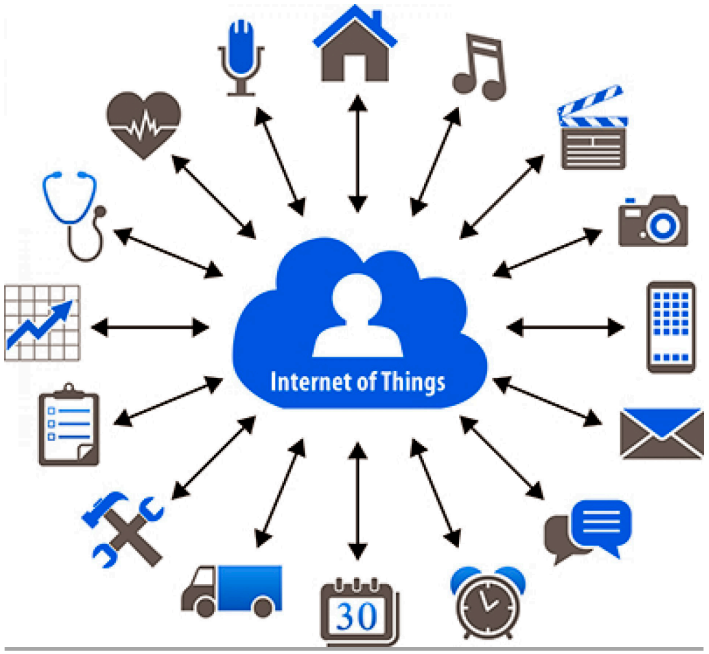
Task Oriented Programming for the Internet of Things

- the omnipresent network of connected `things'
  - computers
  - smartphones
  - smartwatches
  - thermostats
  - lightbulbs
  - cars
  - cows
  - doors
  - fridges
  - smart rooms
  - webcams
  - health sensors, …



the IoT is booming

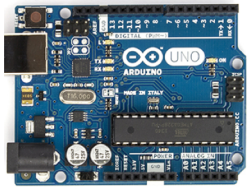




typical `things'

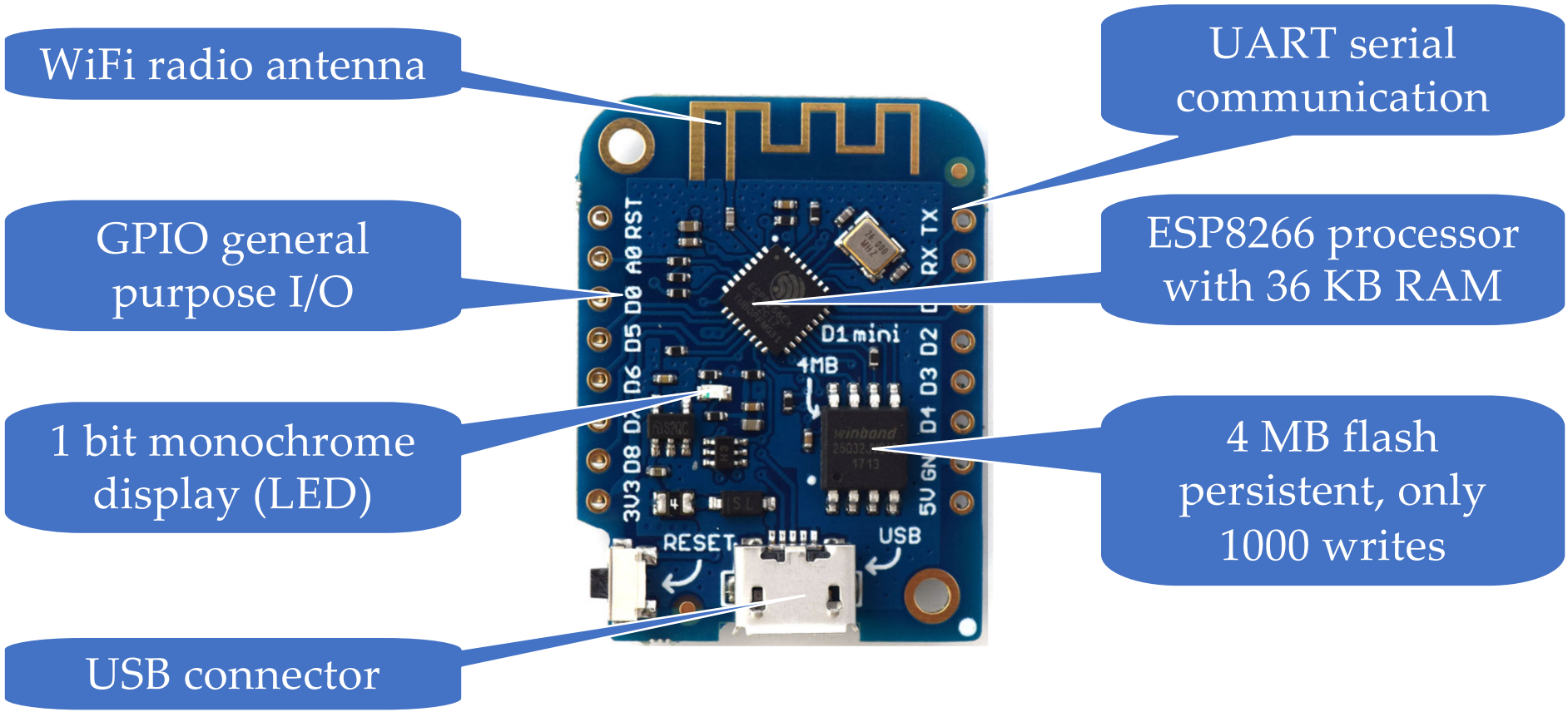
- requirements: small, cheap, energy efficient
- use a **microprocessor**, system on a chip

back to 1975

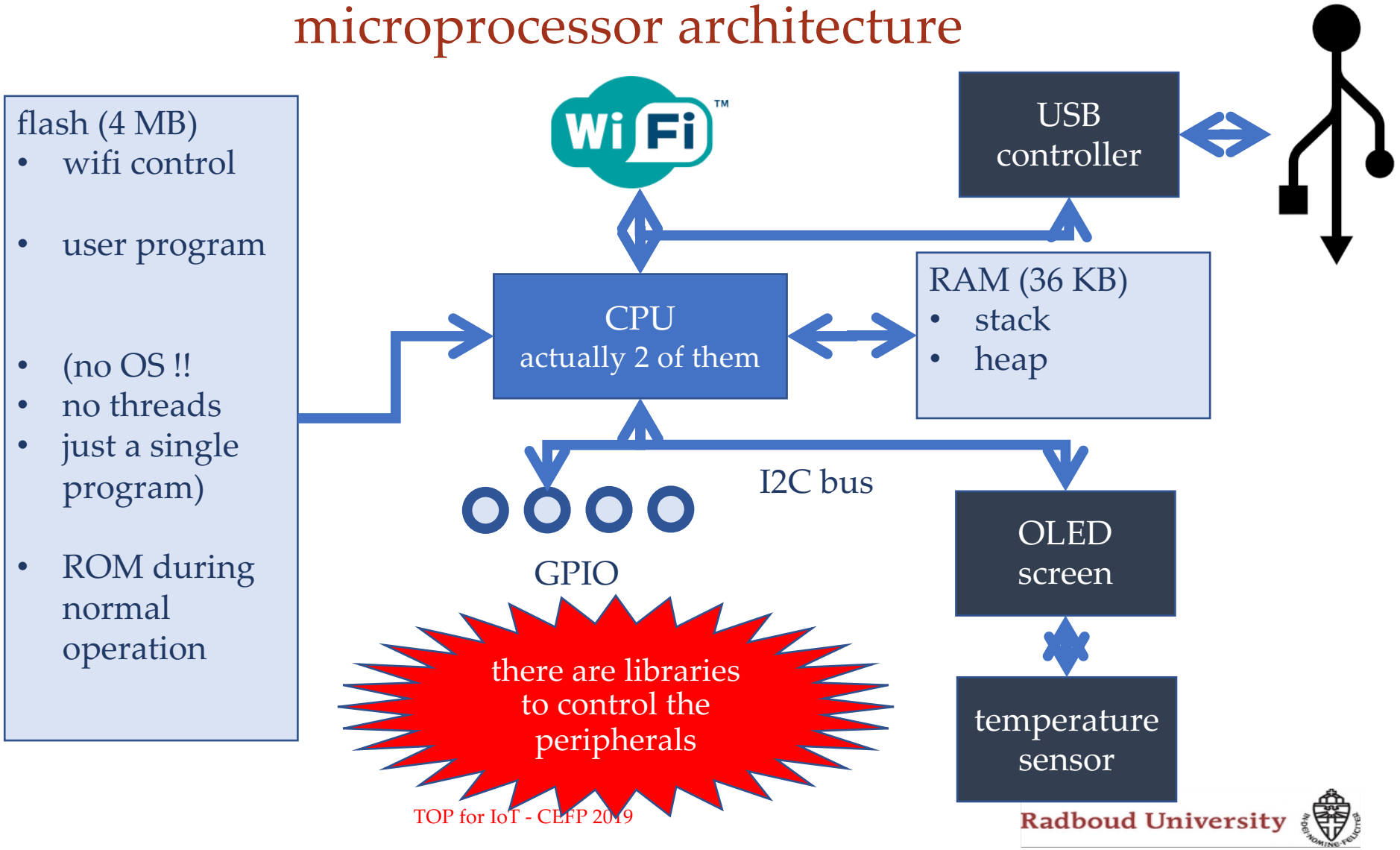


name	Arduino Uno R3	ESP8266	2017 Macbook
processor	8-bit ATMega328	32-bit LX106	64-bit iCore 7
clock speed	16 MHz	80 MHz	4 GHz, duel-core
RAM	2 KB	36 KB	8 GB
Flash memory	32 KB	4 MB	512 GB SSD
USB	✓	✗	✓
WiFi	✗	✓	✓
cost	10 €	3 €	2000 €

example microprocessor: D1 mini (ESP8266)



microprocessor architecture



hello world on the D1 mini

```
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);       //  use D2 as output pin
}

void loop() {                         //  repeated forever
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);                        //  1000 ms = 1 second
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

displaying movement with PIR (Passive InfraRed sensor)


```
#define PIR D3

void setup() {
  pinMode(PIR, INPUT);
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  digitalWrite(LED_BUILTIN, ! digitalRead( PIR ));
  delay(100);
}
```

we cannot execute any other task during this sleep

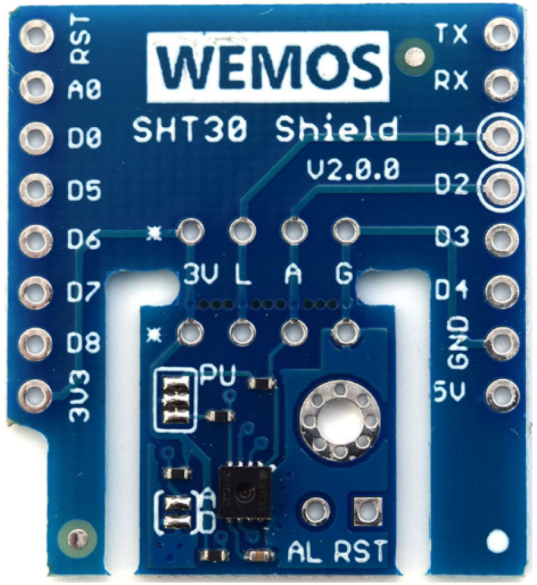
we need task composition

measuring the temperature

```
#include <WEMOS_SHT3X.h>

SHT3X sht30(0x45);

void setup() {
  Serial.begin(115200);
}

void loop() {
  if ( sht30.get() == 0) {
    Serial.print("Temperature in Celsius: ");
    Serial.println(sht30.cTemp);
  } else {
    Serial.println("Error!");
  }
  delay(2000);
}
```

a task result is safer

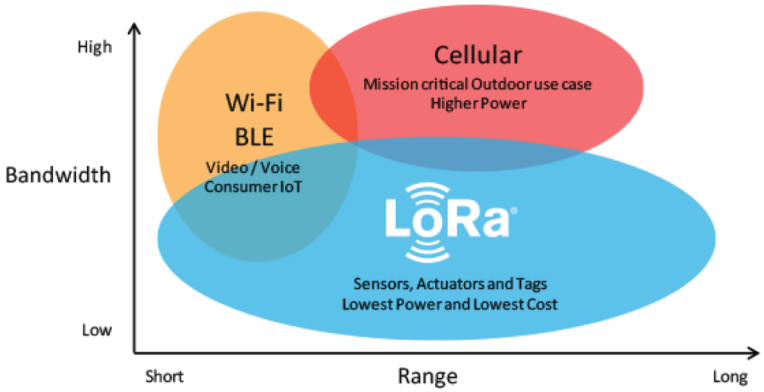
hard to mix with the PIR program

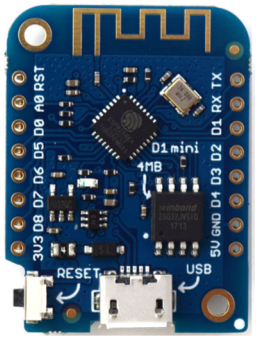


IoT architecture

mix of languages and technologies, hard to develop and maintain

iTasks to the rescue





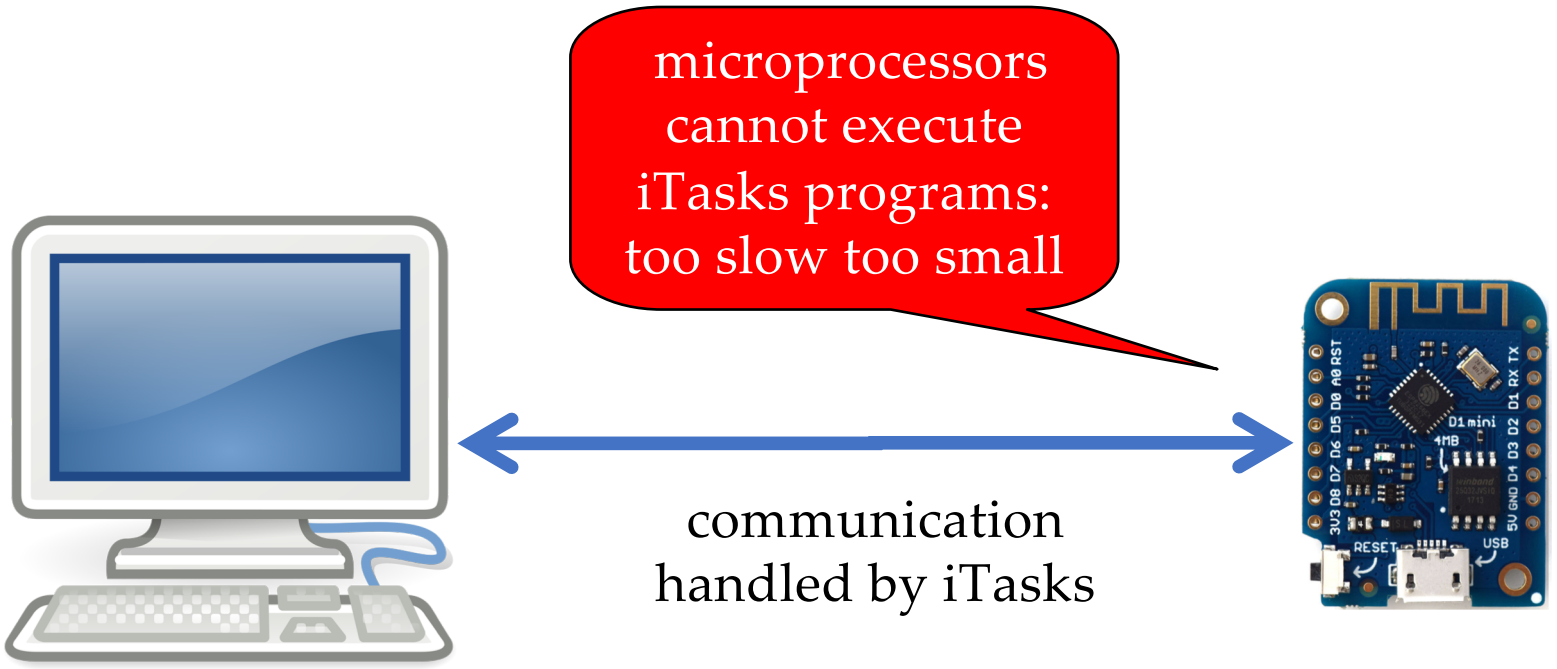
communication via HTTP, TCP, WiFi, GSM, LORA, BLE, MQTT, ..

programmed in Java, C#, Haskell, JavaScript, Clean, C++, Erlang, Python ..

System: Windows, OSX, Linux

programmed in Arduino C, LUA, microPython, ..

IoT architecture with iTasks ☺

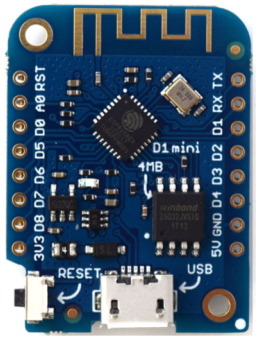


programmed in iTasks

programmed in iTasks

IoT architecture with iTasks ☺

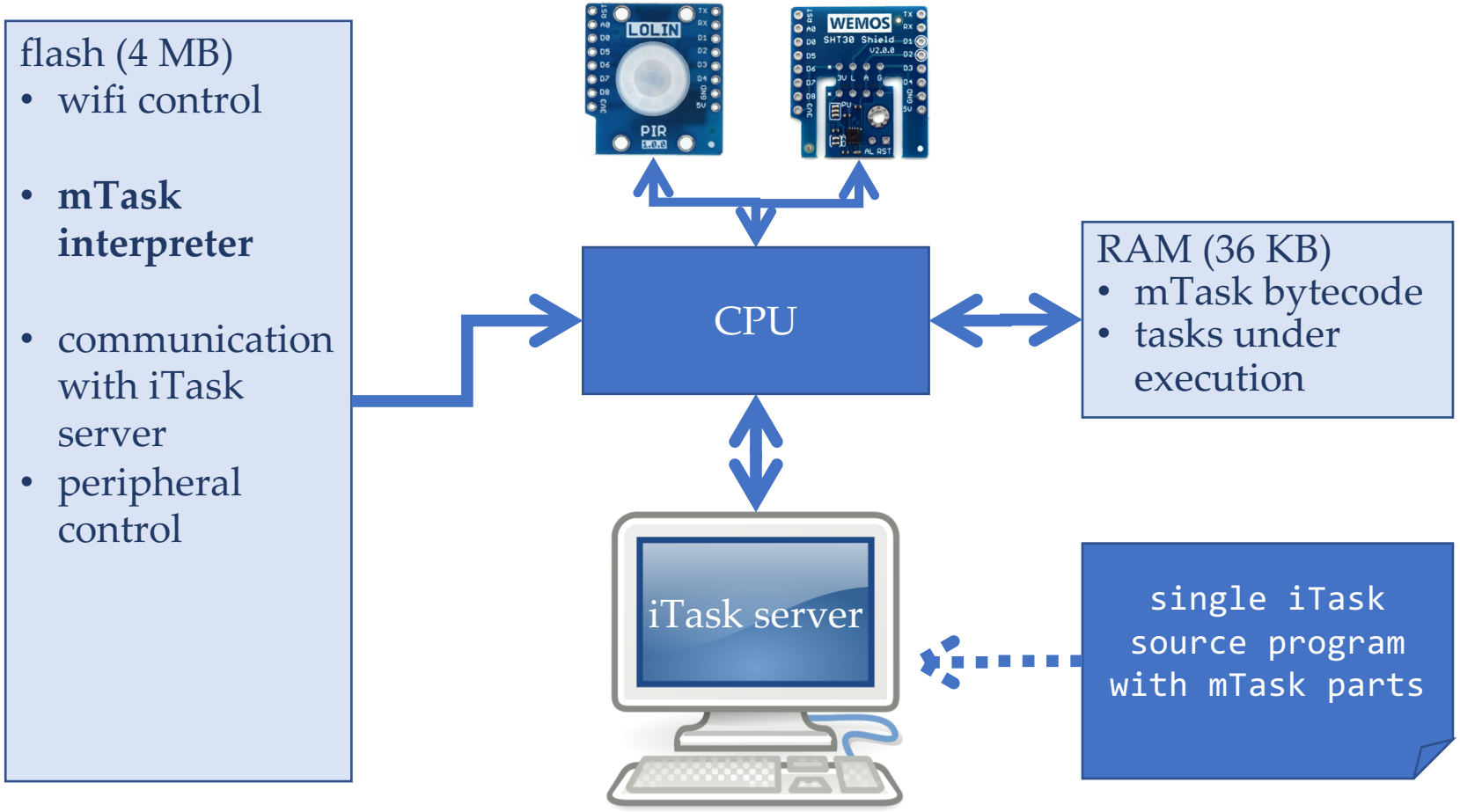
communication
handled by iTasks



programmed in iTasks

programmed in mTasks
special DSL for TOP on IoT

mTask architecture



Domain Specific Language design

• *class based shallow embedding*: DSL is set of **type classes**

```
class artih v where
  lit           :: t           → v t | type t
  (*.) infixl 5 :: (v t) (v t) → v t | type, * t
```

✓ kind of easy

✓ many views – ways to assign a meaning to the DSL;
each instance of the class is a view of the DSL

✓ can be task-based

✓ type safe, even for functions

✓ extendable

☹ we need a new * operator; always add a dot to existing operator names

mTask DSL

- mTask is a collection of type constructor classes

```
class artih v where
  lit           :: t           → v t | type t
  (*.) infixl 5 :: (v t) (v t) → v t | type, * t
```

- class `type` control types allowed in mTask

```
class type t | toString, toCode t
instance type Int, Real, Bool, Char
```

`t`: element type
`v`: view of DSL

overloaded

extendable DSL add new classes by need

example

```
lit 6 *. lit 7
```

type error

```
lit 1 *. lit True
```

making views

```
class arith v where
  lit           :: t → v t              | type t
  (*.) infixl 5 :: (v t) (v t) → v t    | type, * t

e = lit 2 *. lit 3 *. lit 7
```

```
:: Show a = Show String

instance arith Show where
   lit  x    = Show (toString x)
   (*.) x y = x + Show " * " + y
```

```
:: Eval a = Eval a

instance arith Eval where
   lit  x    = Eval x
   (*.) x y = x * y
```

```
Start :: (Show Int, Eval Int)
Start = (e, e)
```

```
((Show "2 * 3 * 7"),(Eval 42))
```

blinking the LED in mTask

```
blink :: Main (MTask v ()) | mtask v
blink =
  {main =
      rpeat
      (     writeD d2 (lit True)
      >>|. delay (lit 500)
      >>|. writeD d2 (lit False)
      >>|. delay (lit 500)
   )}
```

embedding in Clean

in contrast to Arduino C: `main` is **not** repeated

a function to beautify the code

```
blink :: Main (MTask v Bool) | mtask v
blink
  = fun \blink = (\b →
                  delay (lit 500)
          >>|. writeD d2 b
          >>=. blink o Not)
    In {main = blink (lit True)}
```

function named `blink`

function argument named `b`

`>>=.` use result of `writeD`

equivalent to
`\x . blink (Not x)`

do not forget the \ before the name in the definition of functions and argument

blinking 3 LEDs (hard in Arduino C)

```
blink :: Main (MTask v Bool) | mtask v
blink
   = fun \blink = (\(pin, b, d) →
          delay d
      >>|. writeD pin b
      >>|. blink (p, Not b, d))
   In {main = blink (d1, true, lit 500)
        .||. blink (d2, true, lit 300)
        .||. blink (d3, true, lit 800)}
```

blink has 3 arguments

.||. parallel task composition, first result stable is used

conditional tasks: step

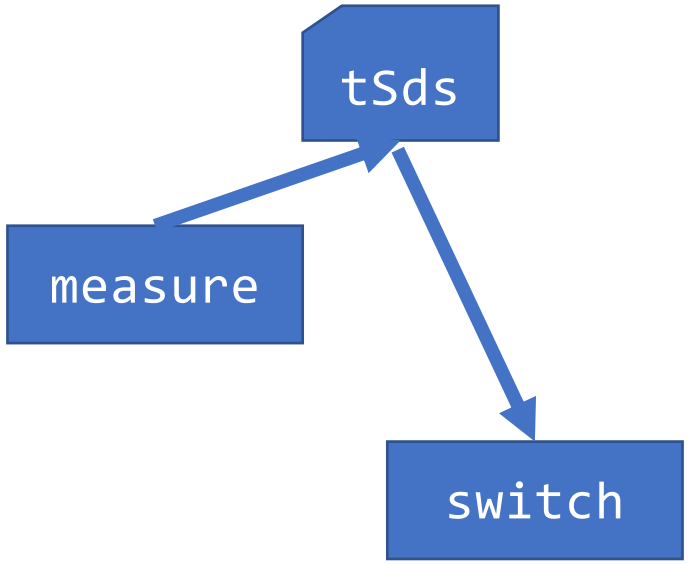
```
t137 :: Main (v (TaskValue Bool)) | mtask, dht v & fun () v
t137 =
   DHT D4 DHT22 \dht =
   fun \temp = (\lim.
      temperature dht >>*.
         [IfValue (\t.t >. lim)
                  (\t.writeD d2 false)
         ,Always (writeD d2 true) ] >>|.
      delay s1 >>|.
      temp lim) In
   {main = temp (lit 25)}
```

possible steps

```
class step v | arith v where
  (>>*.) infixl 1 :: (MTask v t) [Step v t u] -> MTask v u
                   | type u & type t


:: Step v t u
  = IfValue    ((v t) → v Bool) ((v t) → MTask v u)
  | IfStable   ((v t) → v Bool) ((v t) → MTask v u)
  | IfUnstable ((v t) → v Bool) ((v t) → MTask v u)
  | IfNoValue                   (MTask v u)
  | Always                      (MTask v u)
```

task communication: share

```
t42 :: Main (v (TaskValue (Int, Int))) | ...
t42 =
  DHT D4 DHT22 \dht =
  sds \tSds = 0 In
  fun \measure = (\().
    temperature dht >>~. \t.
    setSds tSds (t /. (lit 10)) >>|.
    delay s1 >>|.
    measure ()) In
  fun \switch = (\lim.
    delay s5 >>|.
    getSds tSds >>~. \t.
    writeD d2 (t >. lim) >>|.
    switch lim) In
  {main = measure () .&&. switch (lit 25)}
```



overview operator / combinators

task composition	mTasks	iTasks
parallel, one of them	`.||.`	`-||-`
parallel, both of them	`.&&.`	`-&&-`
sequence	`>>|.`	`>>|`
bind	`>>=.`	`>>=`
bind, proceed on unstable value	`>>~.`	`>>~`
step	`>>*.`	`>>*`

arithmetic operators	mTasks	iTasks
addition	`+.`	`+`
equality	`==.`	`==`

wrap-up

- programming the IoT is much simpler from one source
- Task Oriented Programming is very suited: TOP = top

- IoT devices cannot execute iTask programs
- hence, we build mTask: a TOP for IoT DSL
  - real TOP design, but small footprint

- we need collaboration iTask ⇹ mTask
- Mart will do this after a short break