# mruby/c
# The smallest Ruby implementation for microcontrollers

HASUMI Hitoshi @hasumikin

*May 6, 2019 in Warszawa*
*May 14, 2019 in Kraków*

# Cześć!

# about me

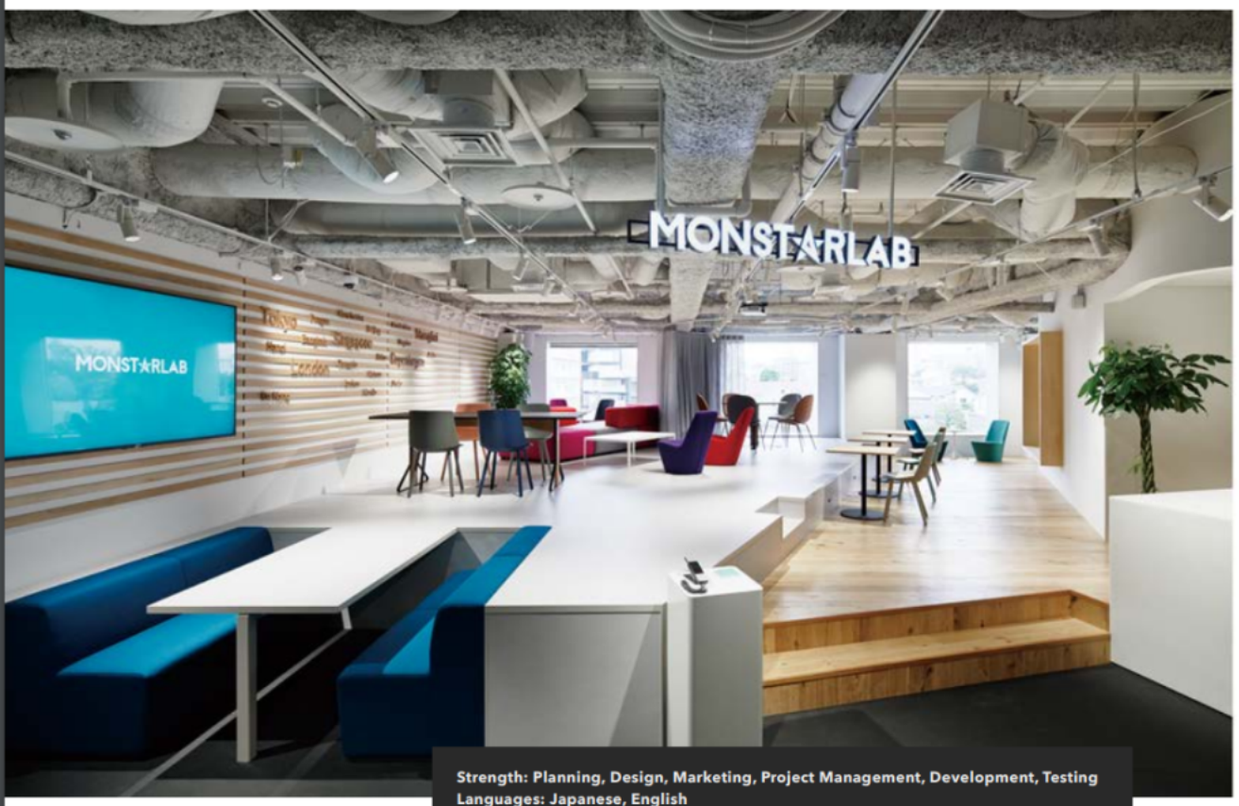
- HASUMI Hitoshi
  @hasumikin
- live in Matsue city,
  a holy place of Ruby
- Sake
  Soba
  Coffee

# about me

MONSTARLAB
GROUP

# about me

## Global Sourcing




Strength: Planning, Design, Marketing, Project Management, Development, Testing
Languages: Japanese, English
Development Languages: Ruby, Java, PHP, HTML/CSS, JavaScript, Swift, Objective C

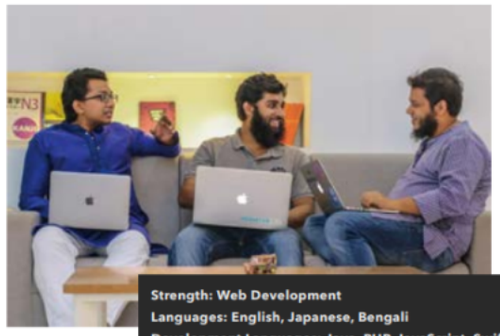
## JAPAN

TOKYO/OSAKA/MATSUE/FUKUOKA

## VIETNAM

DANANG/HANOI

Strength: Mobile App Development (iOS,Android)
Languages: Japanese, English, Vietnamese
Development Languages: PHP, Java, Ruby, iOS,Android, Swift



## BANGLADESH

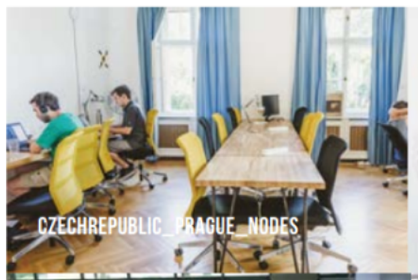
DHAKA

Strength: Web Development
Languages: English, Japanese, Bengali
Development Languages: Java, PHP, JavaScript, Swift, Objective C

## CHINA

BEIJING/SHANGHAI
QINGDAO/CHENGDU

Strength: Game Development
Languages: Japanese, English,Chinese
Development Languages: Java, JavaScript, PHP, iOS, Android






# Matsue.rb

# message from Matz

```
# # video
# # src = images/video.mp4
```

# agenda

- terminology
- about my IoT project
- mruby and mruby/c
- how does mruby/c work
- actual source code of my project
- development environment

# terminology

# terminology

- mruby/c
  - tha language I will talk about today
  - I say mrubyc as /c is hard to pronounce
- microcontroller
  - small computer contains CPU, memory and programmable I/O peripherals
  - in this talk, microcontroller is distinguished from single board computer like Raspberry Pi

# terminology

- RTOS
  - Real-time OS.  usually used for microcontroller
- task
  - almost equivalent to `Thread` in linux. we say `task` in microcontroller world

# terminology

- 旭日酒造(Aasahi-shuzo)
  - shuzo means Sake brewery
  - one of the best Japanese Sake brewery
  - Asahi-beer(famous for SUPER DRY) has no concern with Asahi-shuzo
- Aasahi-shuzo and I make IoT system using mruby/c

# why microcontroller?

- I never use single board computer like Raspberry Pi for production environment.

# why microcontroller?

- it starts immediately right after plugged in
  - end users, brewery workers in my case, can use it simply like home electical things
- you can narrow security issue list
  - many a malware aims at Linux or Windows platform as a target
  - you don't need to consider unnecessary deamon
  - you don't need to do `apt upgrade`

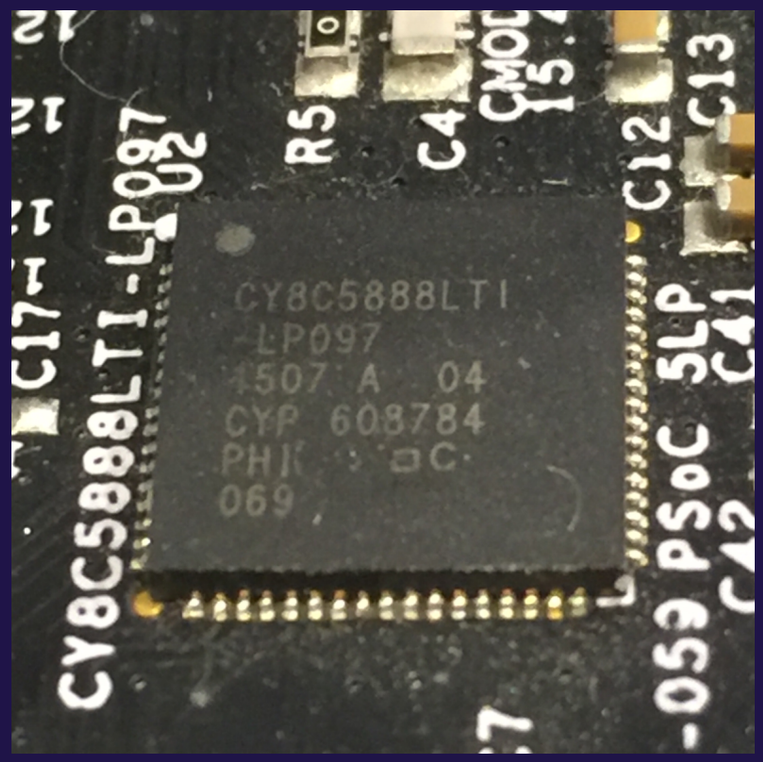
# why microcontroller?

- low energy
  - rarely overheated
  - many choices of power supply
- mass production
  - you can choose appropriate chipset (number of GPIOs, memory size, etc.) for your application
  - cost advantage for parts supply and subcontractor manufacturing

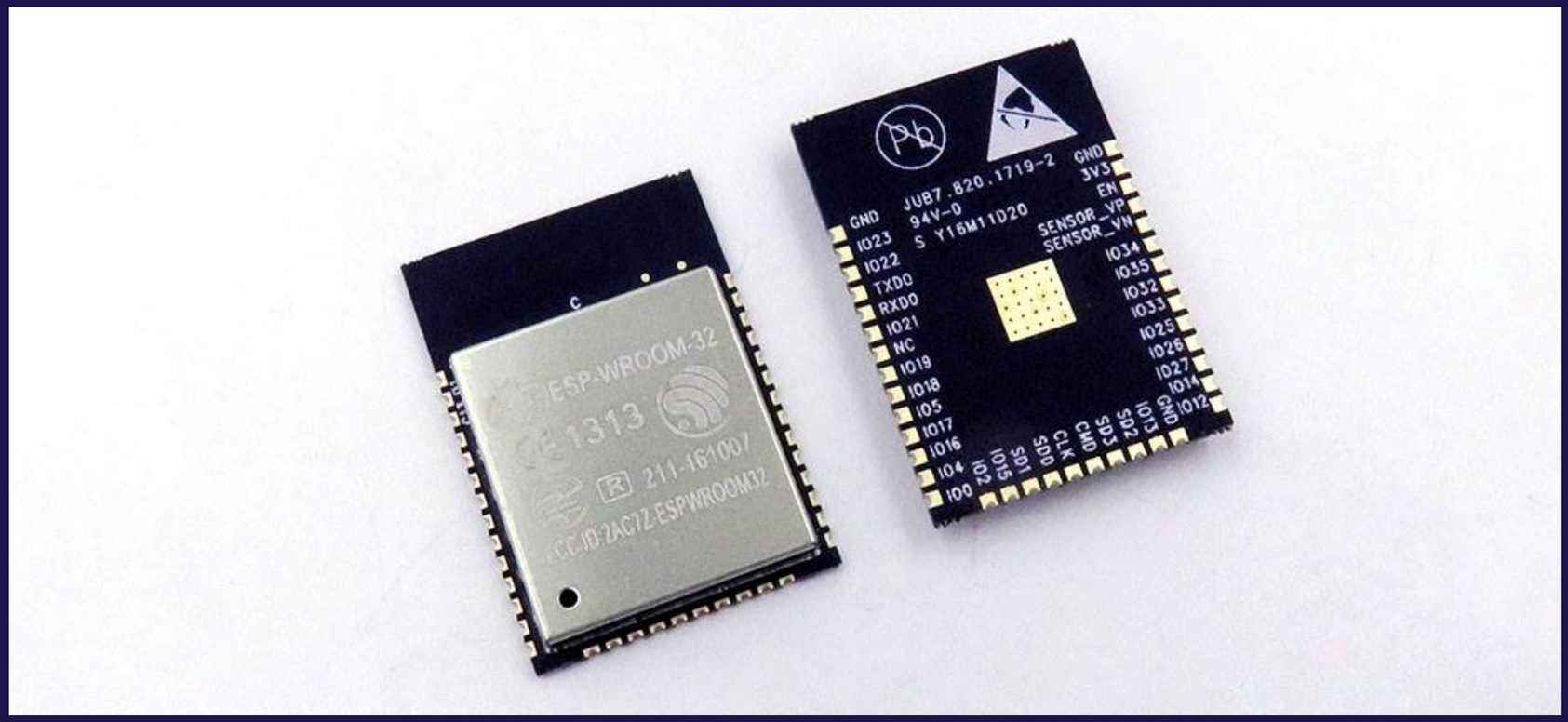
# which microcontroller?

# which microcontroller?

- CYPRESS PSoC5LP
  - 32 bit Arm Cortex-M3 CPU
  - Flash: 256KB
  - SRAM: 64KB
    - 64KB is the target size of mruby/c



# which microcontroller?



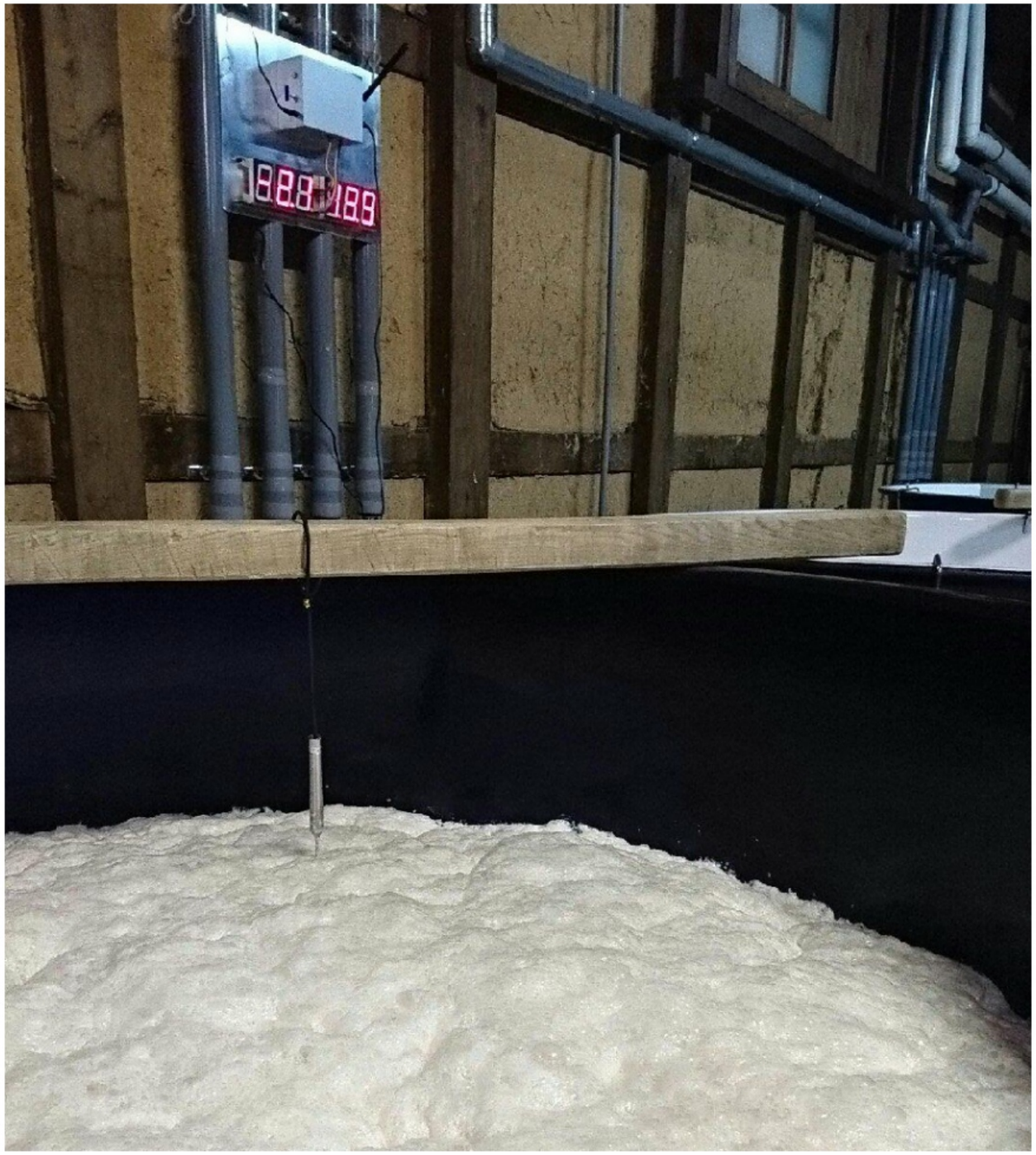

- Espressif ESP-WROOM-32 (ESP32)
  - 32 bit dual core LX6 CPU
  - Flash: 4MB
  - SRAM: 520KB

# about my IoT project

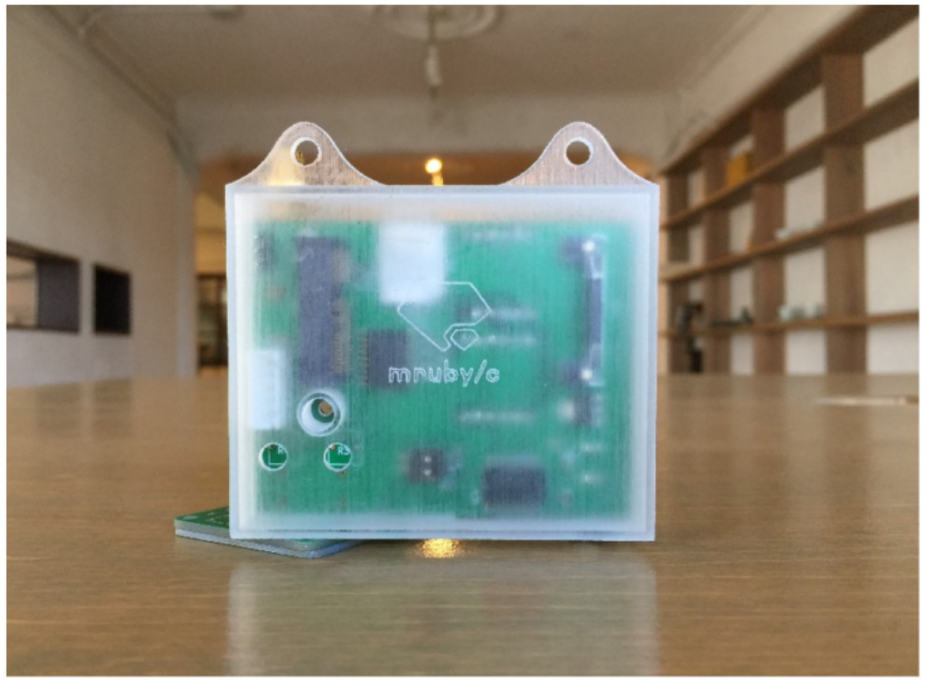
# about my IoT project

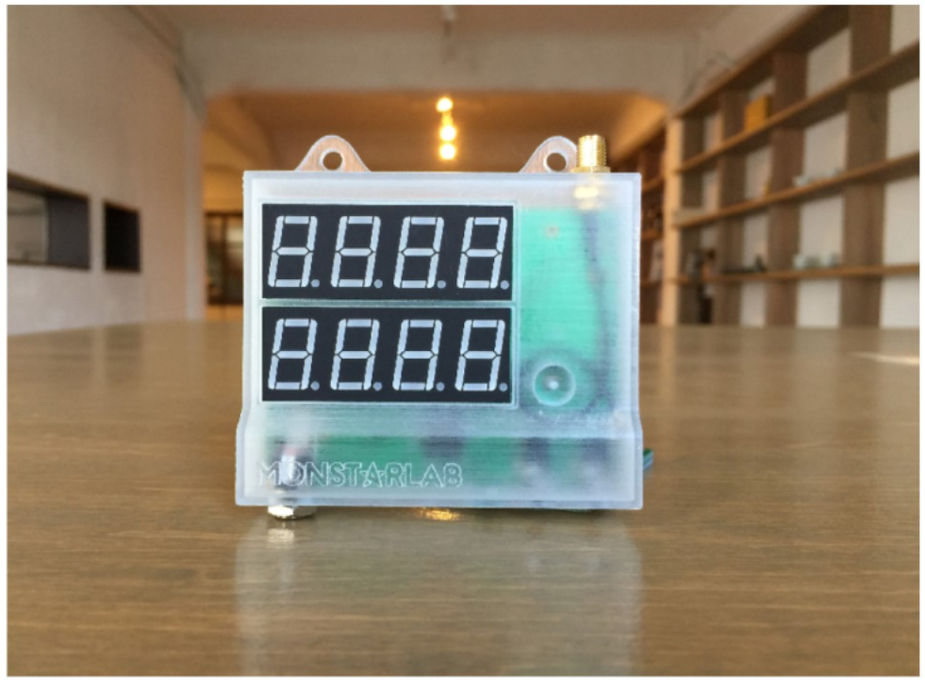
- IoT system for Asahi-shuzo
- delivered to actual brew work in January 2018
- devices post temperature of Sake materials in brewing, surrounding temperature and humidity to server
- data is displayed on smartphone app

# about my IoT project



# about my IoT project




# about my IoT project

## what were difficult about mruby/c?

- we can neither do step execution nor look into appropriate memory address of mruby/c's variables
- so many troubles in IoT
  - hard to find why the application doesn't work well
- mruby/c was growing
  - bugs, lack of features, docs and examples

# about my IoT project

## so, was mruby/c bad?

# about my IoT project

## so, was mruby/c bad? - NO

- IoT at work makes you hurry
  - you have to go back and forth between dark 10℃ storage cellar and humid 35℃ manufacturing room
  - brewery workers run around
  - you have to amend your firmware with your small laptop in 10 minutes
  - you will thank Ruby's descriptiveness and agility

# today's demo

- $CO_2$ concentration
  - 400ppm : atmospheric
  - 1000ppm : your programming speed decreases
  - 1500ppm : tomatoes 🍅 may grow well
  - > 2000ppm : sleepy, headache
  - > 40000ppm : 💀

# today's demo

- my device keeps taking $CO_2$ concentration
- I will prove that it is due to $CO_2$ if someone slept while I speaking

# so many troubles in IoT

# so many troubles in IoT

- peripheral equipments (☆)
- circuit and wiring design
- printed circuit board = PCB
- soldering (☆)
- C, mruby and mruby/c (☆)
- network

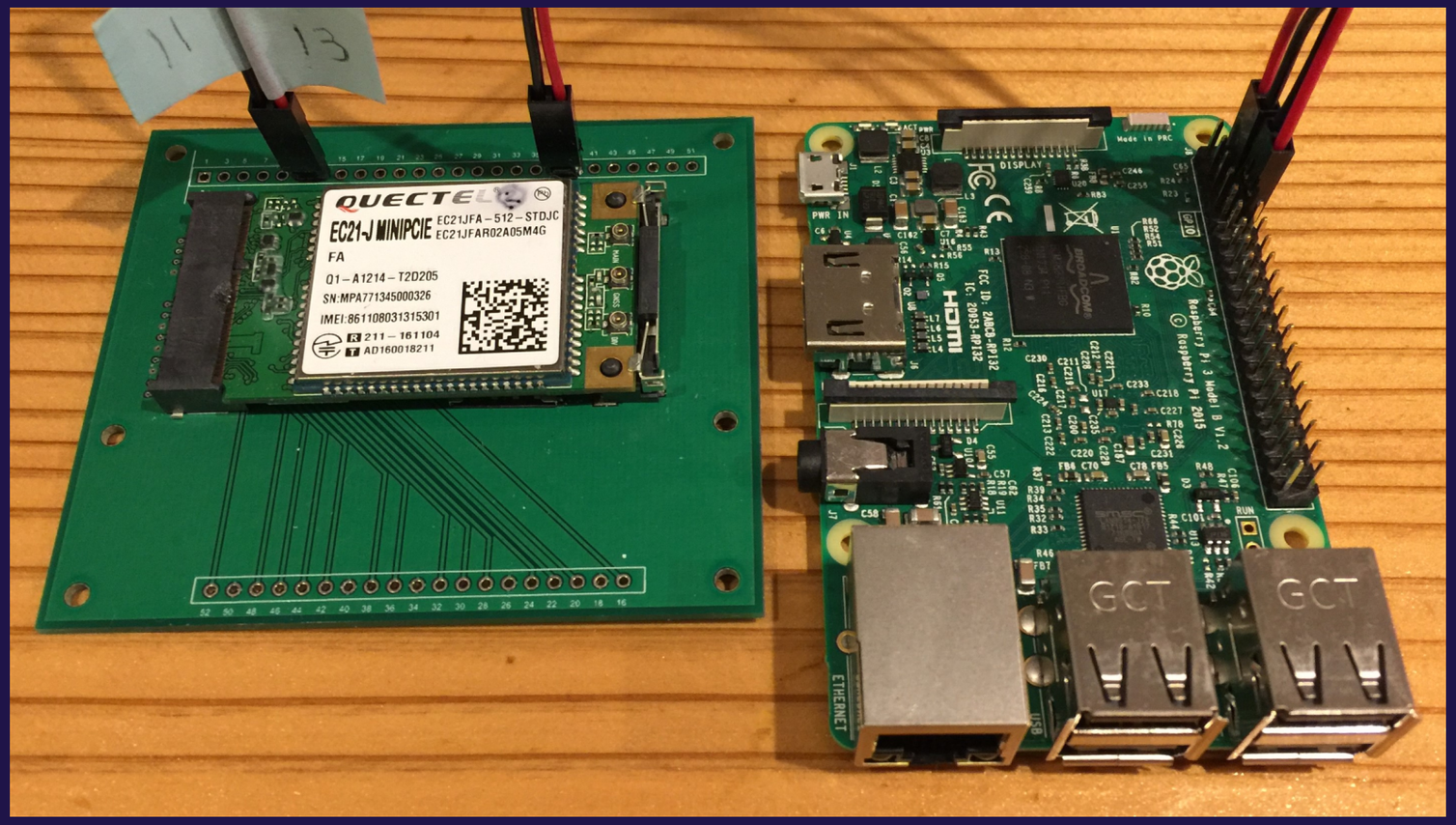
☆...I will explain these topics today

# peripheral equipments

# peripheral equipments

- it is very important to check the following before writing application code
  - equipment like sensor or communication module works as its spec sheet
  - whether the equipment is not broken (sometimes broken by soldering 😭)
- unless you will regret
  - so...

# peripheral equipments




- Raspberry Pi & CRuby are great for pre-prototyping
  - use breadboard or make PCB for test like this photo

# peripheral equipments

## ex) CRuby for serial communication test

```ruby
# notice this is CRuby for RasPi
require 'rubyserial'
require 'timeout'
BAUÐRATE = 9600 # match with your instrument
sp = Serial.new '/dev/serial0', BAUÐRATE, 8
loop do
  puts '[command]'
  command = gets
  sp.write command.sub("\n", "\r") # replace LF if needed
  sleep 0.1
  result = ''
  begin
    Timeout.timeout(10) do
      loop do
        line = sp.read(128)
        break if line == '' && result != ''
        result << line
        sleep 0.1
      puts '=> ' + result
  rescue Timeout::Error
    puts 'timeout!'
```

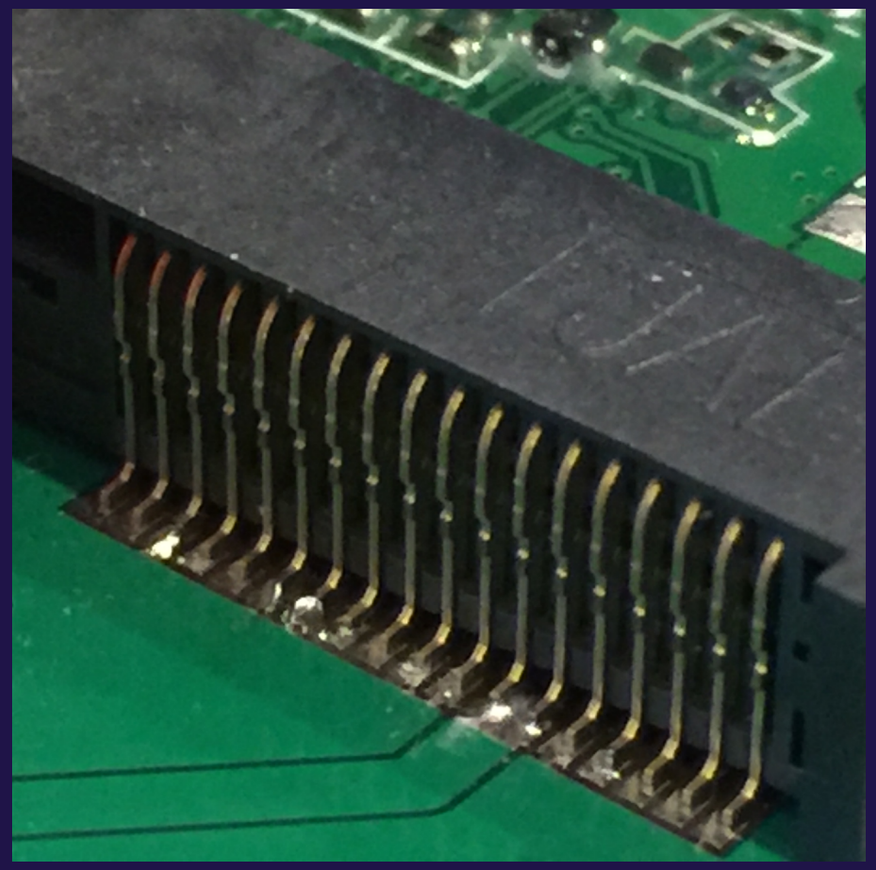
# peripheral equipments

## ex) CRuby for serial communication test

```
$ serial_communication_test.rb
[command]
AT                      # command
=> OK                   # response
[command]
AT+CIMI                 # command
=> 123456789012 # response
[command]
AT+XXX                  # command
=> error                # response
```

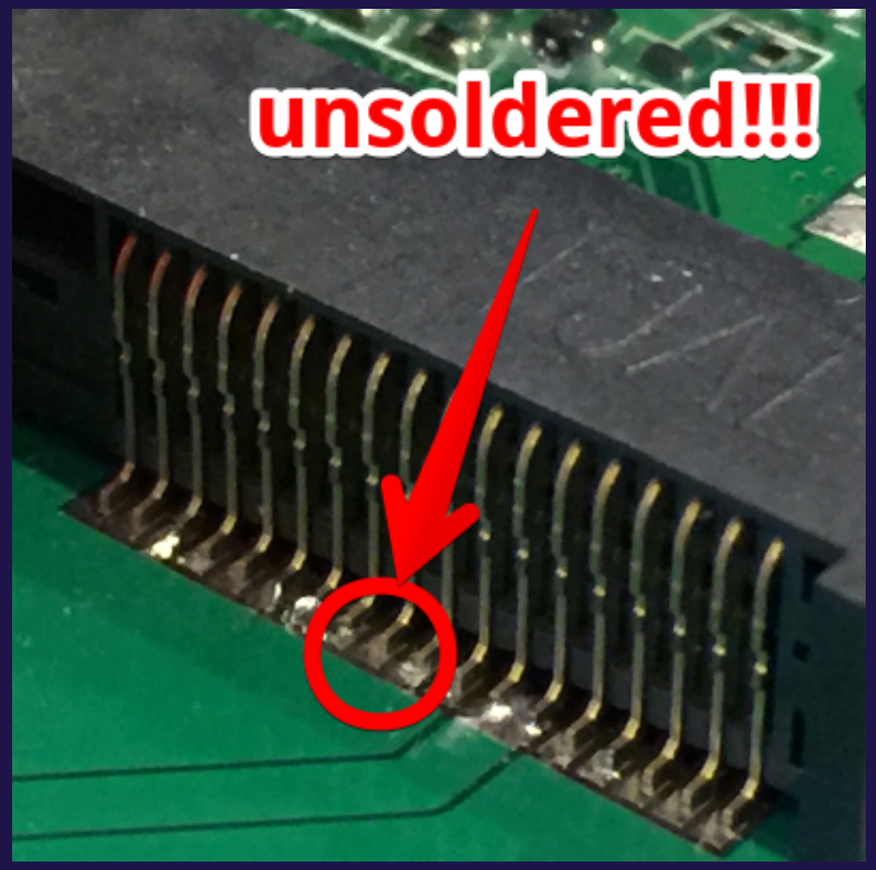
# soldering

# soldering



- it may work even if you leave a pin unsoldered on surface mounting
  - because the pin touches circuit's surface
  - then, it will not work one day

# soldering




- discovering this kind of bug is much more difficult than software bug
- my teacher said
  "all the cause of failure, it is **impatience**"

# what is mruby?

# what is mruby?

- github.com/mruby/mruby
- another implemantation of Ruby for embedded usage
- easily being called from C/C++
- ngx_mruby is a popular one
- good for command line tools as one-binary executable

# what is mruby/c?

# what is mruby/c?

- github.com/mrubyc/mrubyc
- yet another implementation of mruby
- `/c` symbolizes compact, concurrent and capability
- especially dedicated to one-chip microcontroller

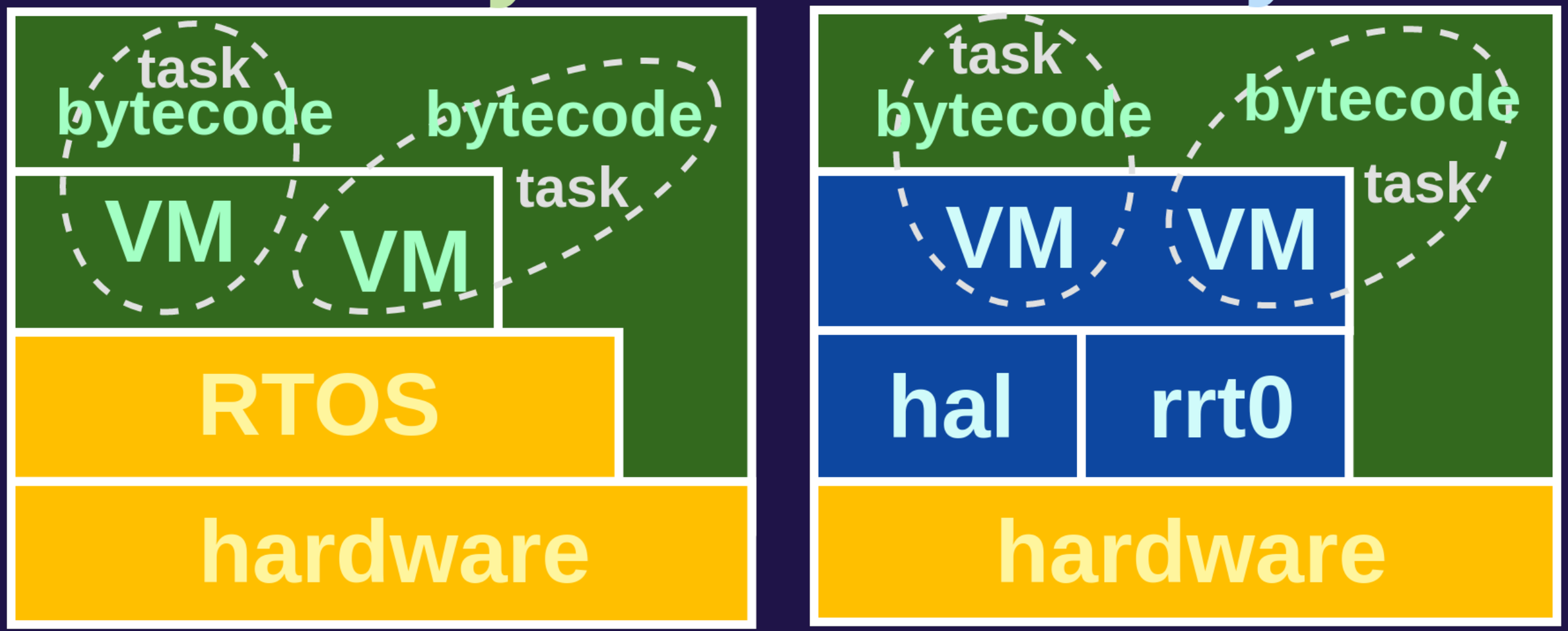
# mruby and mruby/c

| mruby | mruby/c |
|---|---|
| v1.0.0 in Jan 2014 | v1.0 in Jan 2017 |
| for general embedded software | for one-chip microcontroller |
| RAM < 400KB | RAM < 40KB |

- sometimes mruby is still too big to run on microcontroller

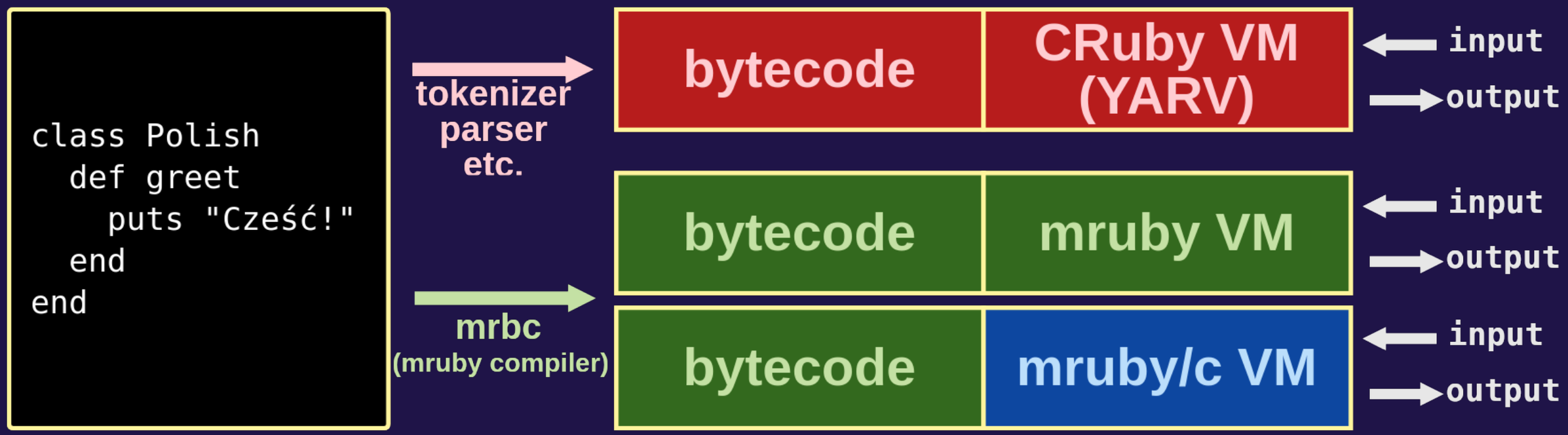
# **both** mruby and mruby/c

- bytecodes are compiled by `mrbc` and VM executes the bytecode



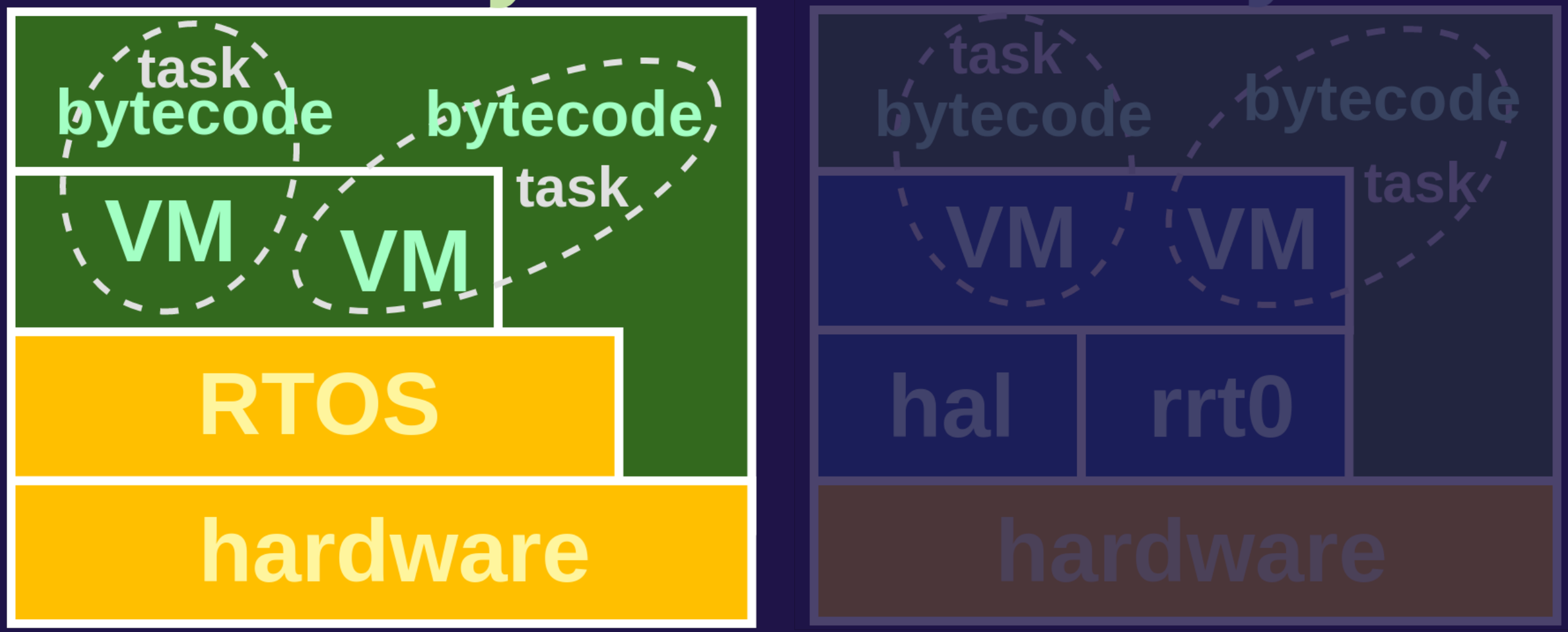
# bytecode

- a kind of intermediate representation
- virtual machine dynamically interprets the bytecode and processes the program



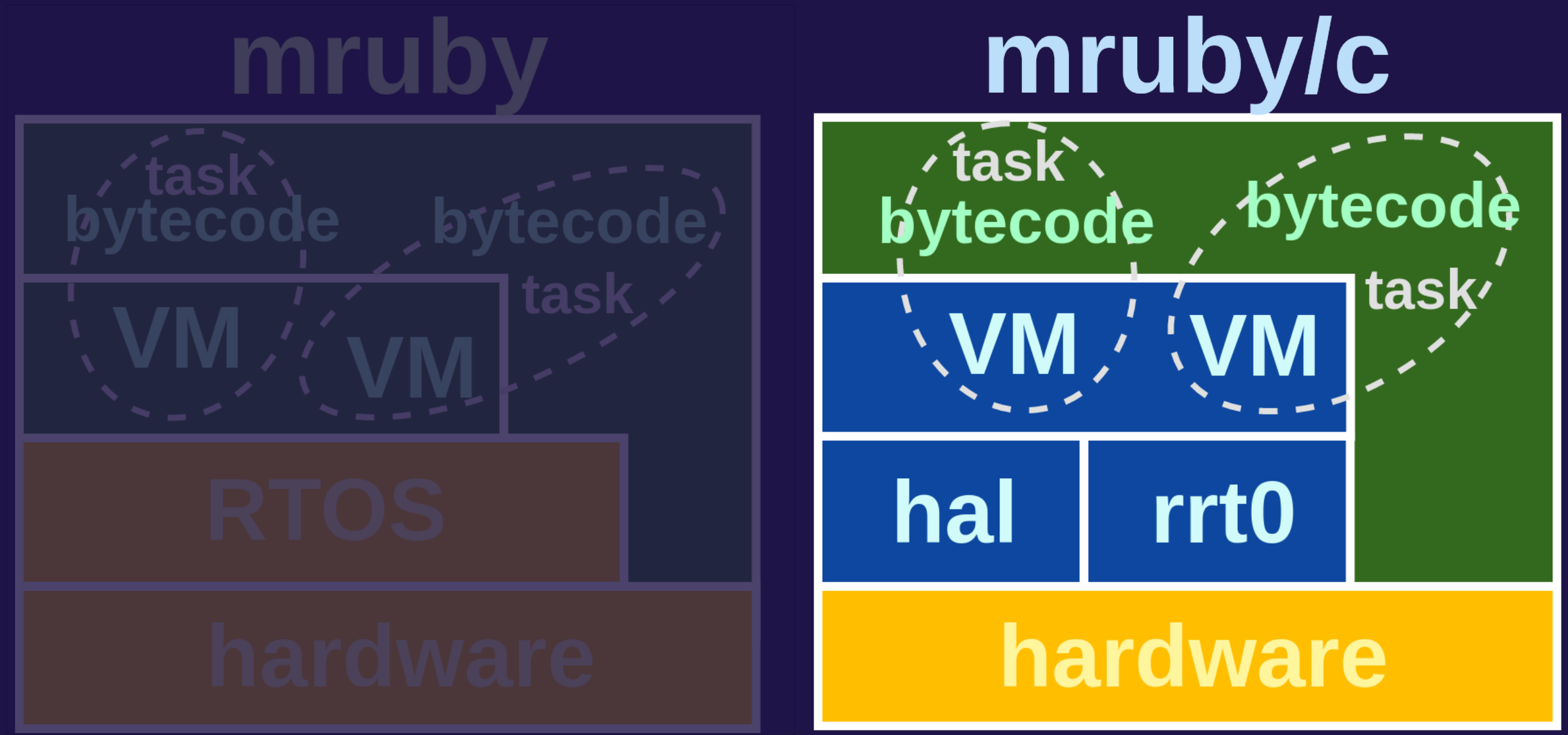
# mruby on microcontroller

- RTOS (Real-Time OS) manages mruby VMs. RTOS has features like multi tasking



# mruby/c on microcontroller

- mruby/c has its own mechanism to manage the runtime: **rrt0**



# mruby/c - virtual machine (VM)

- much smaller than mruby's one
  - that's why mruby/c runs on smaller RAM
- accordingly, mruby/c has **less** functionality than mruby

# how less?

# how **less**? - for example

- mruby/c doesn't have module, hence there is no Kernel module
- then you must wonder how can you `#puts`?
- in mruby/c, `#puts` is implemented in Object class

# how **less**? - for example

- mruby/c doesn't have #send, #eval, nor #method_missing
- moreover, mruby/c neither have your favorite features like TracePoint nor RubyVM::AST ☹

# how **less**? - actually

- the full list of mruby/c's classes
  - Array, FalseClass, Fixnum, Float, Hash, Math, Mutex, NilClass, Numeric, Object, Proc, Range, String, Symbol, TrueClass, VM

# despite the fact,

- no problem in practical use of microcontroller
- as far as IoT go, mruby/c is enough Ruby as I expect
- we can fully develop firmwares with features of mruby/c

# how does mruby/c work

# how does mruby/c work

```
~/sample_project
 ├──── main.c
 ├──── mrblib
 │      ├──── task_1.rb
 │      └──── task_2.rb
 └──── src
        ├──── task_1.c
        └──── task_2.c
```

- task_*.c are compliled code from task_*.rb

# how does mruby/c work

```c
/* main.c */
#include "src/task_1.c"
#include "src/task_2.c"
// use 30KB RAM for VMs in this case
#define MEMORY_SIZE (1024*30)
static uint8_t memory_pool[MEMORY_SIZE];
int main(void) {
  mrbc_init(memory_pool, MEMORY_SIZE);
  mrbc_create_task(task_1, 0);
  mrbc_create_task(task_2, 0);
  mrbc_run(); // 2 tasks run concurrently!
  return 0;
  // we will not write `main loop` in main.c
}
```

# how does mruby/c work

- we can run easily multiple VMs with **concurrency** due to **rrt0**
- you might be disappointed to know you have to write C
- yes, we have to write **main.c**
- don't worry, it's almost boilerplate code

# how does mruby/c work

```
~/mrubyc $ tree src -P *.h
src
├── alloc.h
├── c_array.h
...
├── console.h
├── errorcode.h
├── global.h
├── hal_posix
│   └── hal.h
├── hal_psoc5lp
│   └── hal.h
├── hal_esp32
│   └── hal.h
├── load.h
├── mrubyc.h
├── opcode.h
├── rrt0.h
├── static.h
├── symbol.h
├── value.h
├── vm.h
└── vm_config.h
```

# how does mruby/c work

```
~/mrubyc $ tree src -P *.h
src
├── alloc.h
├── c_array.h
...
├── console.h
├── errorcode.h
├── global.h
├── hal_posix
│   └── hal.h # hal for POSIX
├── hal_psoc5lp
│   └── hal.h # hal for PSoC5LP
├── hal_esp32
│   └── hal.h # hal for ESP32
├── load.h
├── mrubyc.h
├── opcode.h
├── rrt0.h
├── static.h
├── symbol.h
├── value.h
├── vm.h
└── vm_config.h
```

# how does mruby/c work

```
~/mrubyc $ tree src -P *.h
src
├── alloc.h
├── c_array.h
...
├── console.h
├── errorcode.h
├── global.h
├── hal_posix
│   └── hal.h
├── hal_psoc5lp
│   └── hal.h
├── hal_esp32
│   └── hal.h
├── load.h
├── mrubyc.h
├── opcode.h
├── rrt0.h      # runtime scheduler
├── static.h
├── symbol.h
├── value.h
├── vm.h
└── vm_config.h
```

# how does mruby/c work

```
~/mrubyc $ tree src -P *.h
src
├── alloc.h
├── c_array.h
...
├── console.h
├── errorcode.h
├── global.h
├── hal_posix
│   └── hal.h
├── hal_psoc5lp
│   └── hal.h
├── hal_esp32
│   └── hal.h
├── load.h
├── mrubyc.h
├── opcode.h
├── rrt0.h
├── static.h
├── symbol.h
├── value.h # this gives you hints about variable
├── vm.h
└── vm_config.h
```

# how does mruby/c work

```
~/mrubyc $ tree src -P *.h
src
├── alloc.h
├── c_array.h
...
├── console.h
├── errorcode.h
├── global.h
├── hal_posix
│   └── hal.h
├── hal_psoc5lp
│   └── hal.h
├── hal_esp32
│   └── hal.h
├── load.h
├── mrubyc.h
├── opcode.h
├── rrt0.h
├── static.h
├── symbol.h
├── value.h
├── vm.h
└── vm_config.h # edit this if needed
```

# debugging

# debugging

- we can neither do step execution nor look into memory to see variables when we use mruby/c in general
- we should prepare a way of debugging before writing app code
- let's go with **old-fashioned 'print debug'**. it'll be almost enough

# debugging

```
/* a part of main.c */
// create serial console with UART for debug print
static void c_debugprint(mrb_vm *vm, mrb_value *v, int argc){
  int total, used, free, fragment;
  mrbc_alloc_statistics(&total, &used, &free, &fragment);
  console_printf(
    "Memory total:%d, used:%d, free:%d, fragment:%d\n",
    total, used, free, fragment);
  unsigned char *key = GET_STRING_ARG(1);
  unsigned char *value = GET_STRING_ARG(2);
  console_printf("%s:%s\n", key, value);
}
int main(void) {
  ...
  mrbc_define_method(0, mrbc_class_object, "debugprint", c_debugprint);
  ...
}
```

# debugging

```
# mruby
pi = 3.14
debugprint('Pi', pi.to_s)

=> # print in serial console like 'PuTTY' connecting USB
   Memory total:30000, used:20000, free:10000, fragment:3
   Pi:3.14
```

# actual source code

# actual source code

github.com/hasumikin/co2-demo

# actual source code

```
~/co2-demo
 ├─── main.c
 └─── mrblib
       ├─── loops
       │     ├─── master.rb
       │     └─── slave.rb
       └─── models
             ├─── co2.rb
             ├─── led.rb
             └─── thermistor.rb
```

# actual source code

```
# loops/master.rb
$co2 = Co2.new      # Makes it global so that another task
                    # can use it
led = Led.new(19)   # 19 is a pin number which LED connects
while true
  co2 = $co2.concentrate
  if co2 > 2000     # When CO2 reaches fatal level
    5.times do      # Turning LED on and off
      led.turn_on
      sleep 0.1
      led.turn_off
      sleep 0.1
    end
  elsif co2 > 1500 # CO2 reaches warning level
    led.turn_on     # Just keeps turn it on
    sleep 1
  else              # Safe level
    led.turn_off    # Turns off
    sleep 1
  end
end
```

# actual source code

how does **Led#trun_on** work?

# actual source code

```
# models/led.rb
class Led
  def initialize(pin)
    @pin = pin
    gpio_init_output(@pin)
    turn_off
  end
  def turn_on
    gpio_set_level(@pin, 1)
  end
```

# actual source code

```c
/* a part of main.c */
#include "models/led.c"
static void c_gpio_init_output(mrb_vm *vm, mrb_value *v,
                               int argc) {
  int pin = GET_INT_ARG(1);
  gpio_set_direction(pin, GPIO_MOĐE_OUTPUT);
}
static void c_gpio_set_level(mrb_vm *vm, mrb_value *v,
                             int argc){
  int pin = GET_INT_ARG(1);
  int level = GET_INT_ARG(2);
  gpio_set_level(pin, level);
}
int main(void){
  ...
  mrbc_define_method(0, mrbc_class_object, "gpio_init_output",
                     c_gpio_init_output);
  mrbc_define_method(0, mrbc_class_object, "gpio_set_level",
                     c_gpio_set_level);
  ...
}
```

# actual source code

```c
/* a part of main.c */
#include "models/co2.c"
static void c_get_co2(struct VM *vm, mrbc_value v[], int argc){
  uint8_t command[] = { // Command to take CO2
    0xFF, 0x01, 0x86, 0x00, 0x00, 0x00, 0x00, 0x00, 0x79
  };
  uart_write_bytes(uart_num, (const char*)command, 9);
  // ↑Write then ↓Read data
  uint8_t data[10];
  int length = 0;
  ESP_ERROR_CHECK(uart_get_buffered_data_len(uart_num, (size_t*)&length));
  length = uart_read_bytes(uart_num, data, length, 10);
  int i;
  mrb_value array = mrbc_array_new( vm, 9 ); // mrubyc's variable
  for( i = 0; i < 9; i++ ) {
    mrb_value value = mrb_fixnum_value(data[i]);
    mrbc_array_set( &array, i, &value ); // Adding a value to array
  }
  SET_RETURN(array); // Returning the array to mruby
}
int main(void){
  ...
  mrbc_define_method(0, mrbc_class_object, "get_co2", c_get_co2);
  ...
}
```

# actual source code

```ruby
# models/co2.rb
class Co2
  def concentrate
    res = get_co2
    # checks if the sensor works
    if res[0] == 255 && res[1] == 134
      res[2] * 256 + res[3]
    else
      0
    end
  end
end
```

# actual source code

- by the way,
  - C function can return String instead of mruby/c Array
  - `mrbc_array_new` will allocate larger memory than `mrbc_string_new`
- so, you can use String instead of Array if memory becomes short

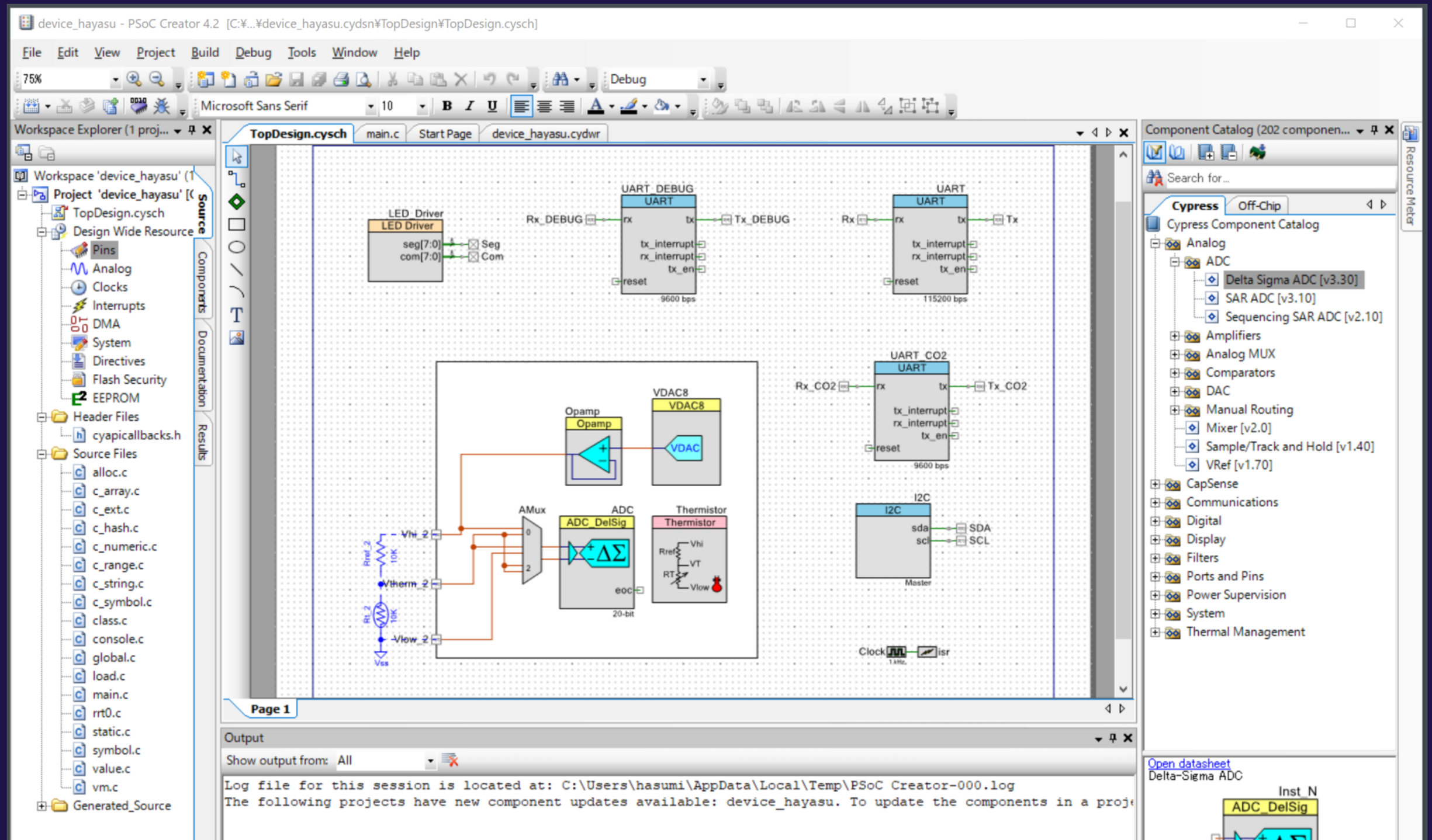
# actual source code

```
# loops/slave.rb
while true
  co2 = $co2.concentrate
  temperature = $thermistor.temperature
  if co2 > 0
    data = "co2=#{co2}&temperature=#{temperature}"
    puts "ĐATASENĐ:#{data}"
    sleep 300
  else
    sleep 3
  end
end
```

# development environment

# development environment

- PSoC Creator for PSoC5LP



# development environment

- the env depends on microcontroller
  - IDE is your env if you use **PSoC5LP**
    - you can code mruby on any text editor
    - IDE is almost mandatory to configure hardware
  - terminal based work is the one if you use **ESP32**

# dev tools for mruby/c

# dev tools for mruby/c

- mrubyc-utils
- mrubyc-test
- mrubyc-debugger

# mrubyc-utils

- github.com/hasumikin/mrubyc-utils
- one-binary tool made with mruby
- helps to install boilerplate of application
- shows mruby/c's classes and methods

# mrubyc-utils

```
your_project $ mrubyc-utils --help
Usage: mrubyc-utils COMMAND [ARGS]

 install         Install mruby/c repo into your local and setup templates.
                 Please run this command at the top directory
                 of your firmware project.
 update          Update mruby/c repo to the newest **master** branch.
 checkout        Checkout specified tag or commit of mruby/c repo.
  -t ¦ --tag       [required] You can specify anything that
                   `git checkout` will accept.
 tag             Show all tags of mruby/c repogitory that you installed.
 classes         Show all the classes that are defined in
                 mruby/c's virtual machine.
 methods         Show all the methods that are available
                 in a class of mruby/c.
  -c ¦ --class     [required] You have to specify class name
 compile         Compile your mruby source into C byte code.
                 This command is for PSoC Creator project. Use make command instead
                 if your project is dedicated to ESP32 or POSIX
  -w ¦ --watch     [optional] Monitoring loop runs and will
                   compile mruby source every time you touched it.

-v ¦ --version   Show version.
-h ¦ --help      Show usage. (this message)

Dependencies:
 git
 mrbc (mruby compiler)
```

# mrubyc-utils

```
your_project $ mrubyc-utils classes
- Array
- FalseClass
- Fixnum
- Float
- Hash
- Math
- Mutex
- NilClass
- Numeric
- Object
- Proc
- Range
- String
- Symbol
- TrueClass
- VM
```

# mrubyc-utils

```
your_project $ mrubyc-utils methods --class=array
Array
- +                     - inspect
- <<                    - join
- []                    - last
- []=                   - length
- at                    - max
- clear                 - min
- collect               - minmax
- collect!              - new
- count                 - pop
- delete_at             - push
- dup                   - shift
- each                  - size
- each_index            - to_s
- each_with_index       - unshift
- empty?                < Object
- first                   - !
- index                   ...
```

# mrubyc-test

- github.com/mrubyc/mrubyc-test
- unit testing framework
- RubyGem implemented with CRuby instead of mruby
- supports stub and mock
- official tool of mruby/c dev team

# mrubyc-test

- gathers information of test cases by CRuby metaprogramming power
- generates stub and mock methods
- makes all-in-one script: **test.rb**

```
class Sample
  def hoge
    "fuga"
  end
end
```

```
class SampleTest < MrubycTestCase
  def some_case
    obj = Sample.new
    assert_equal obj.hoge, "fuga"
  end
end
```

```
SampleTest#some_case
  assertion : assert_equal
  result    : fuga
Finished
1 examples, 0 failures
```

# mrubyc-debugger

- github.com/hasumikin/mrubyc-debugger
- RubyGem
- debugger for infinite loop

# (anime gif DEMO)

github.com/hasumikin/
mrubyc-debugger

# summary

# summary

- mruby/c is the smallest implementation of Ruby

# summary

- mruby/c is the smallest implementation of Ruby
- we can write firmwares for microcontrollers with mruby/c

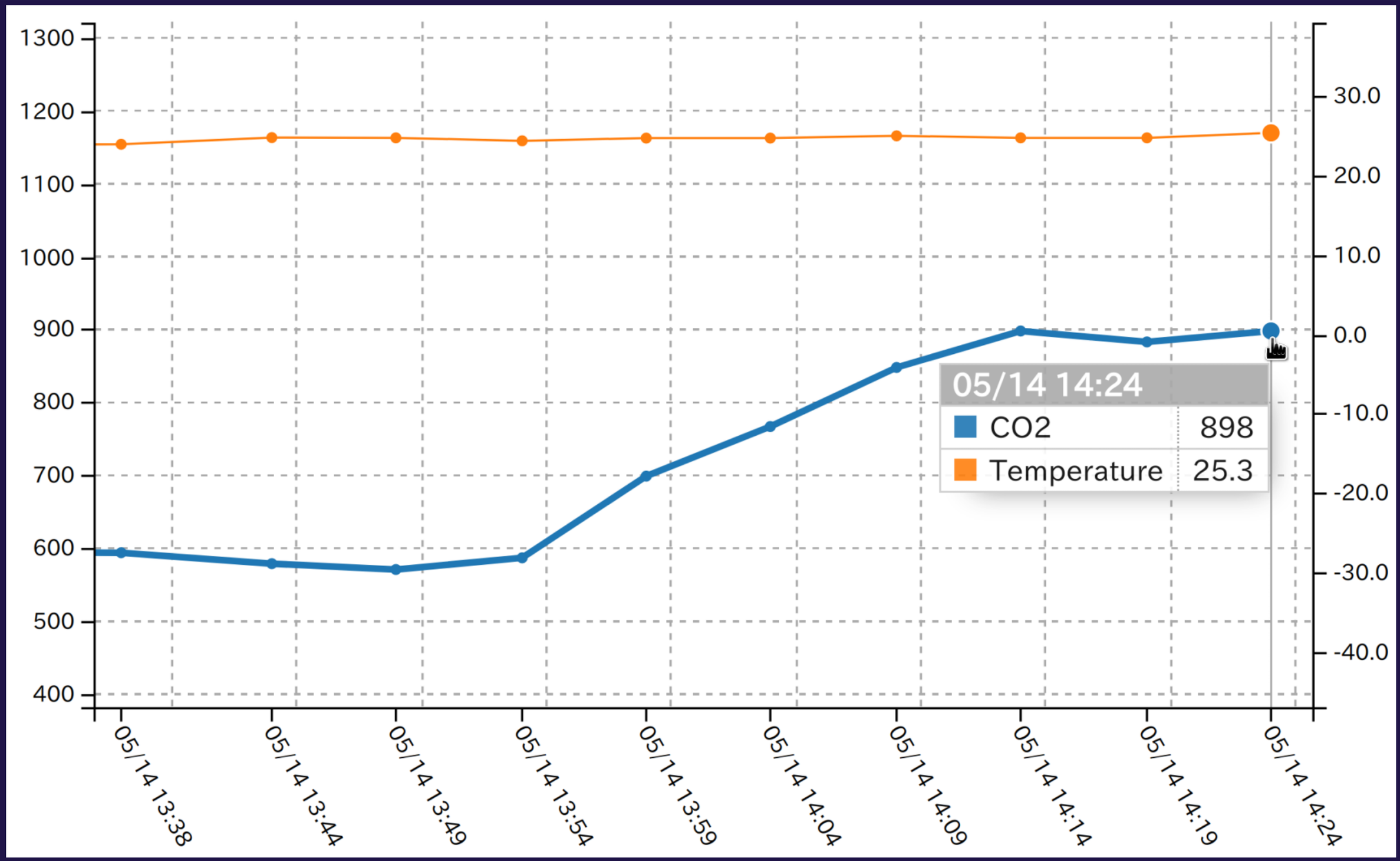
# summary

- mruby/c is the smallest implementation of Ruby
- we can write firmwares for microcontrollers with mruby/c
- it has a short history though, it's ready for production with the Rubyish ecosystem like testing tool

# (DEMO)

$CO_2$

# (added after conference)



# (added after conference)

the ventilation facility
of Browar Lubicz is pretty good 👍

# conclusion

# conclusion

You should refresh air

# thank you!


23BY
加水火入
12
%
吟良雄町にごり火入
生酛純米
旭日
旭日酒造有限会社
島根県出雲市今市町六六
じゅうじあさひ
純米吟醸酒
旭日
じゅうじあさひ
生酛純米
旭日
旭日酒造有限会社
島根県出雲市今市町六六
じゅうじあさひ
五百石
25BY
生酛純米
旭日
純米酒
旭日酒造有限会社
島根県出雲市今市町六六