

Practical mruby/c firmware development with CRuby

HASUMI Hitoshi @hasumikin

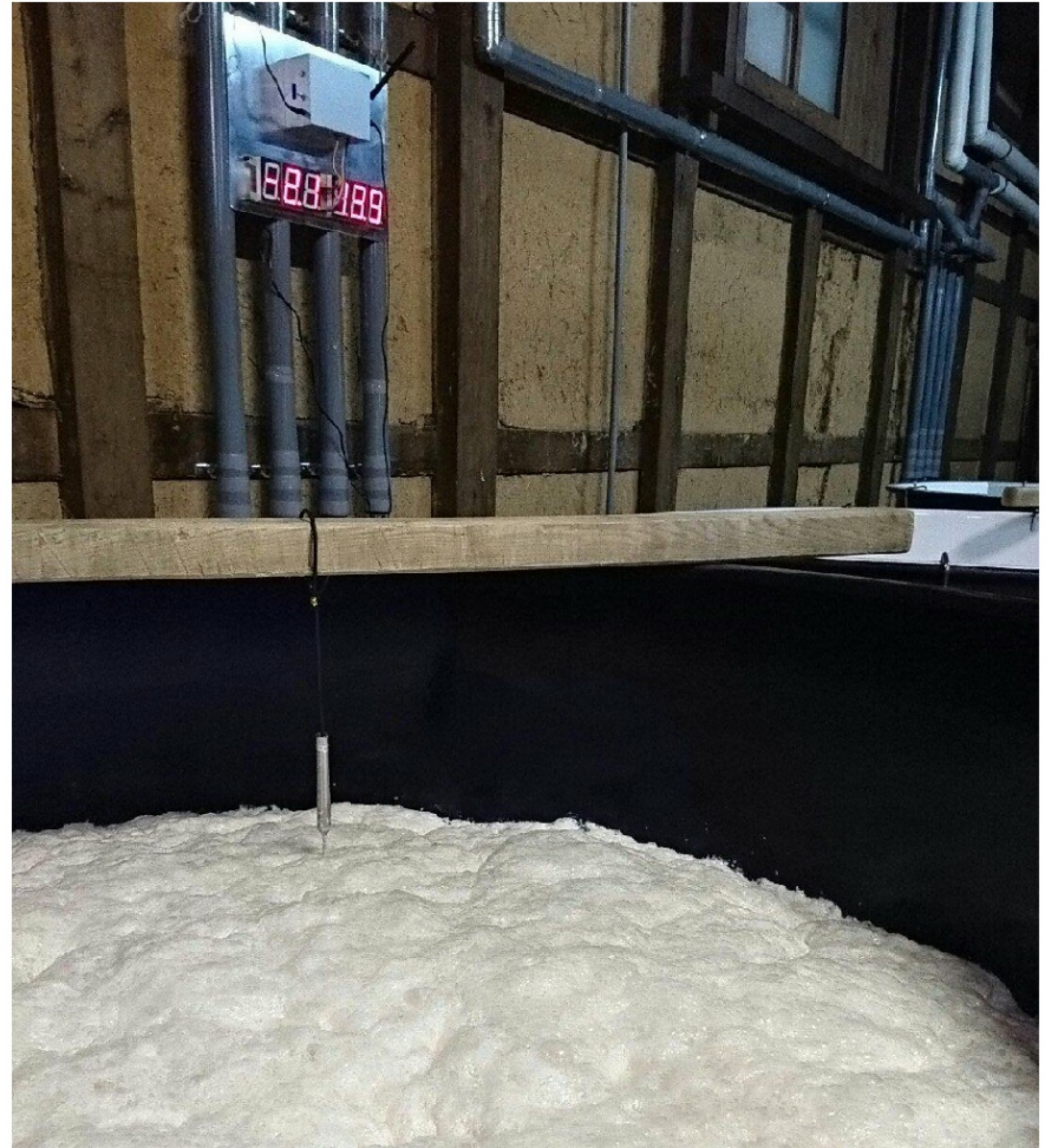
RubyKaigi 2019

April 19, 2019

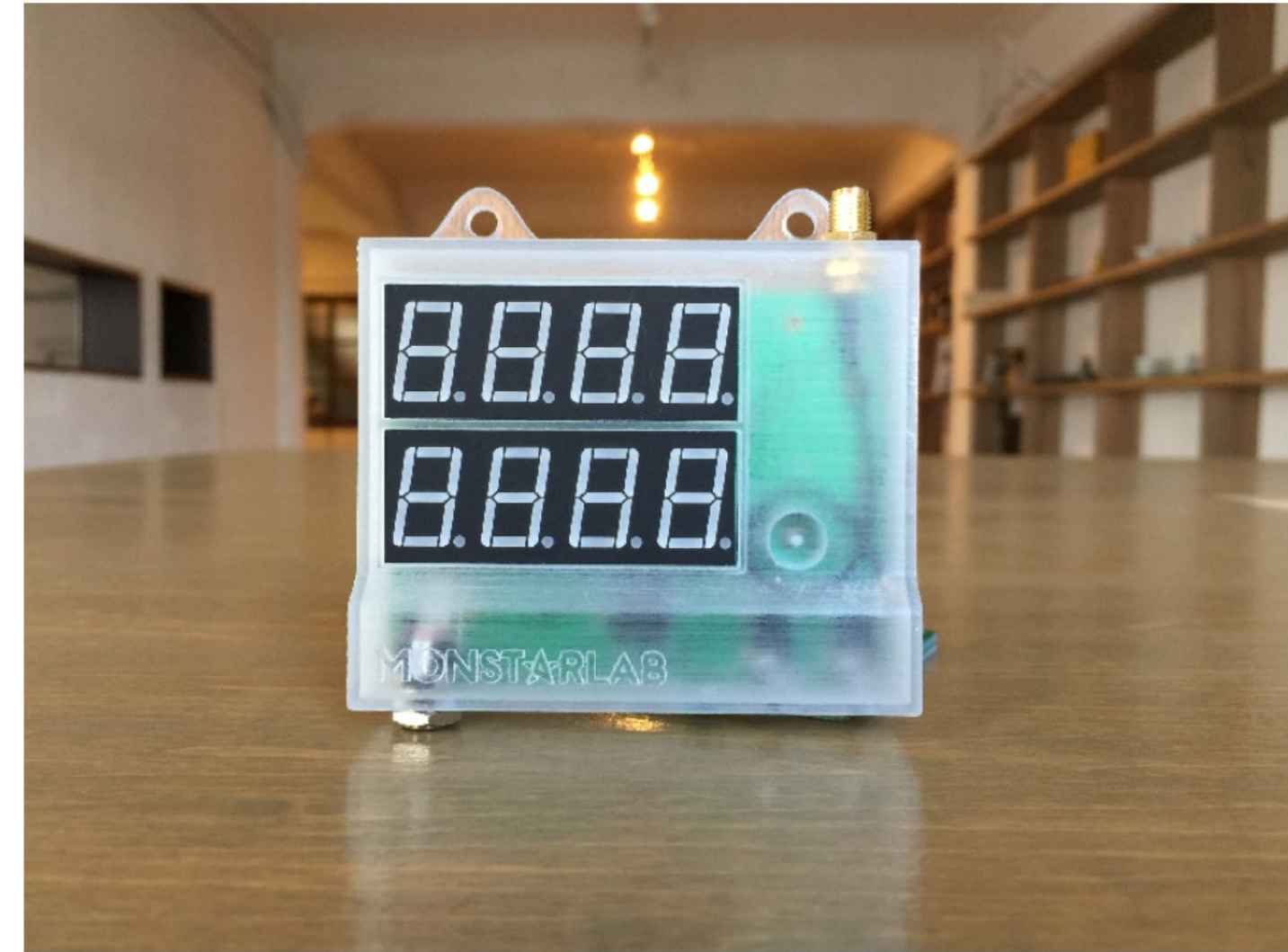
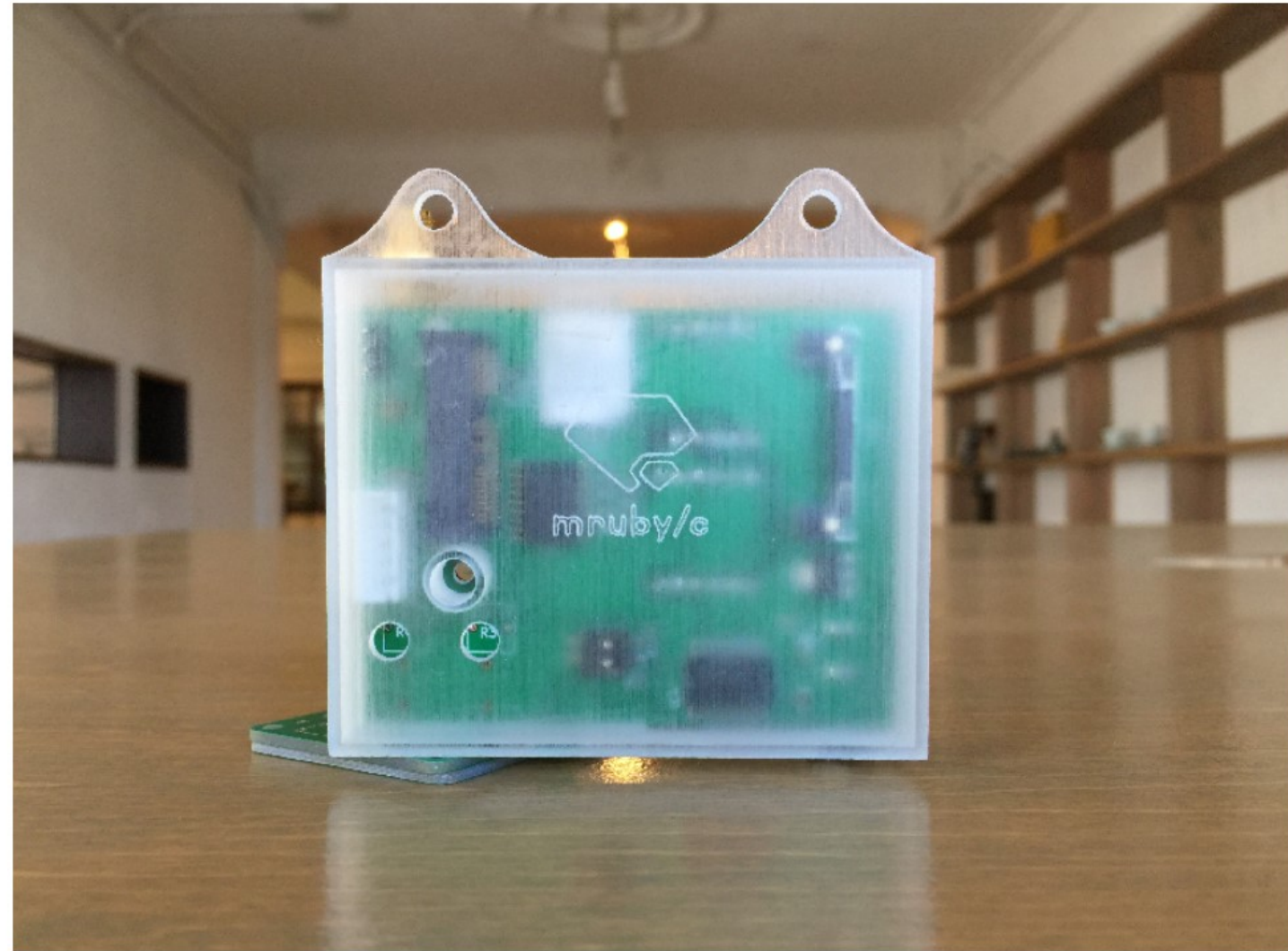
Fukuoka International Congress Center



Sake IoT project

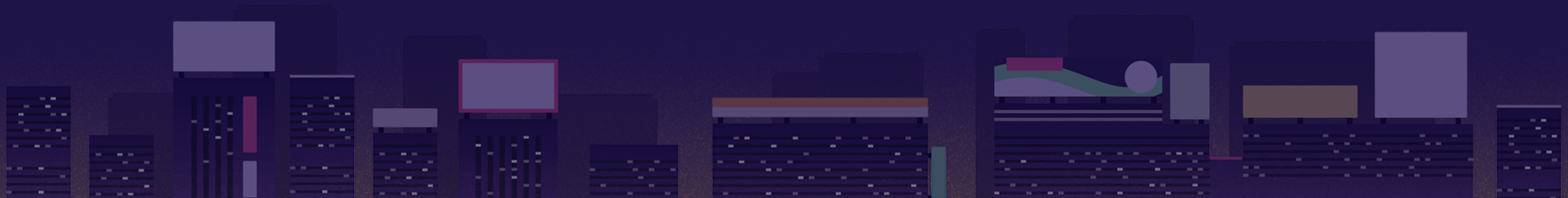


Sake IoT project



what is mruby/c?

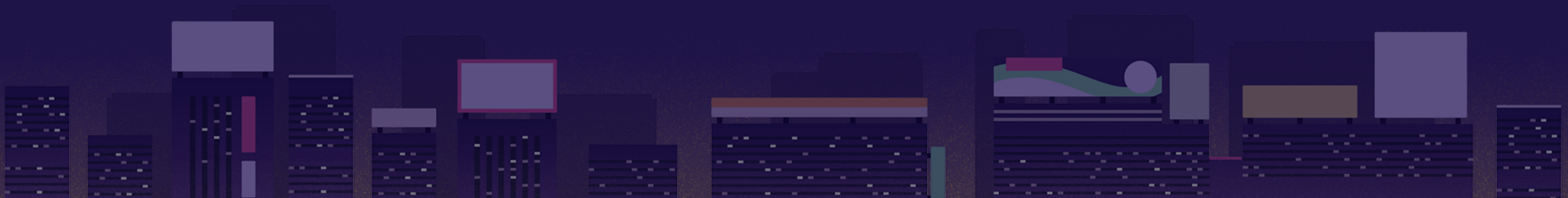
- ⑨ mruby/c is another implementation of mruby
- ⑨ dedicated to microcontroller
- ⑨ `/c` symbolizes
compact, concurrent and capability



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

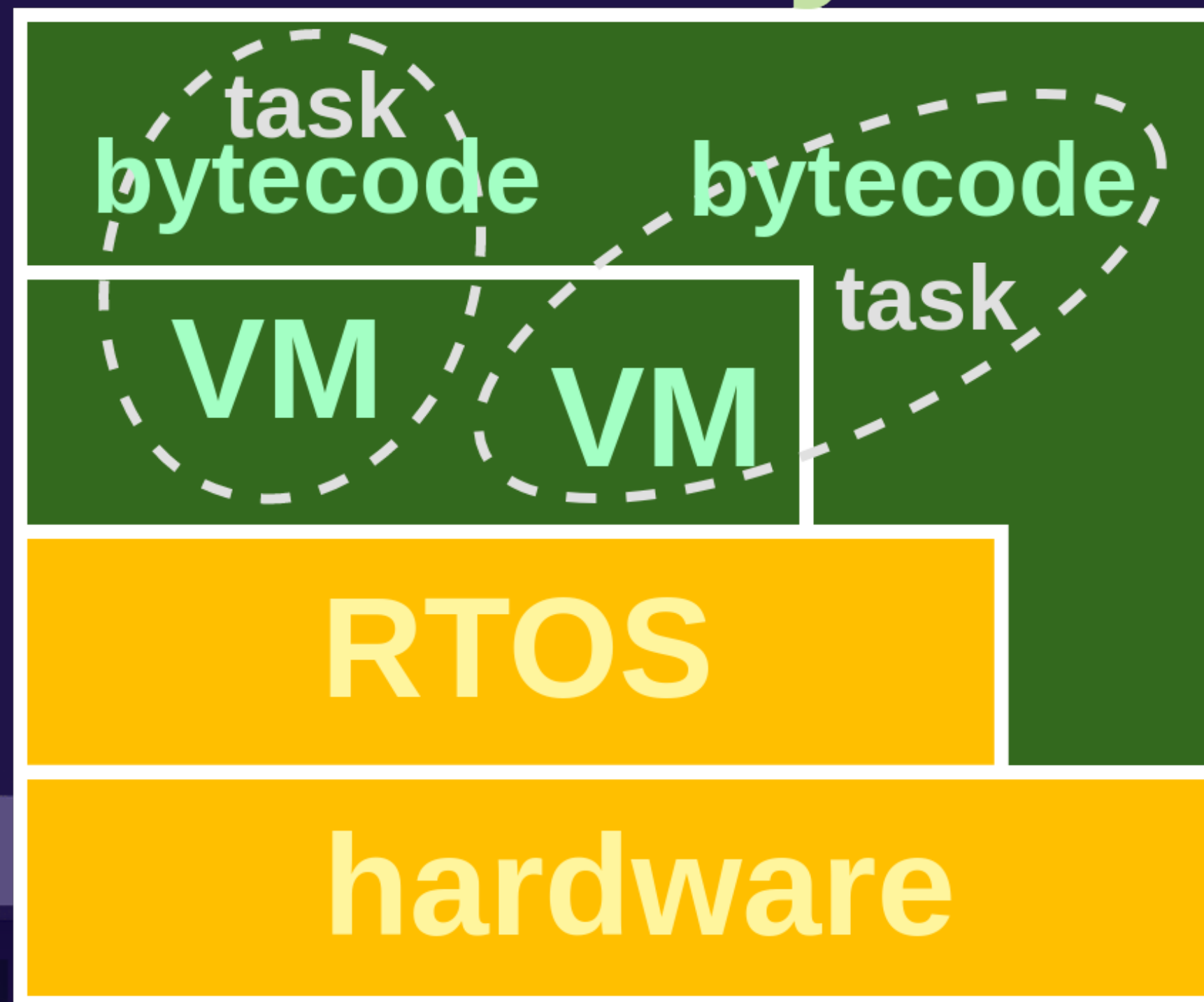
- ◉ sometimes mruby is still too big to run on microcontroller



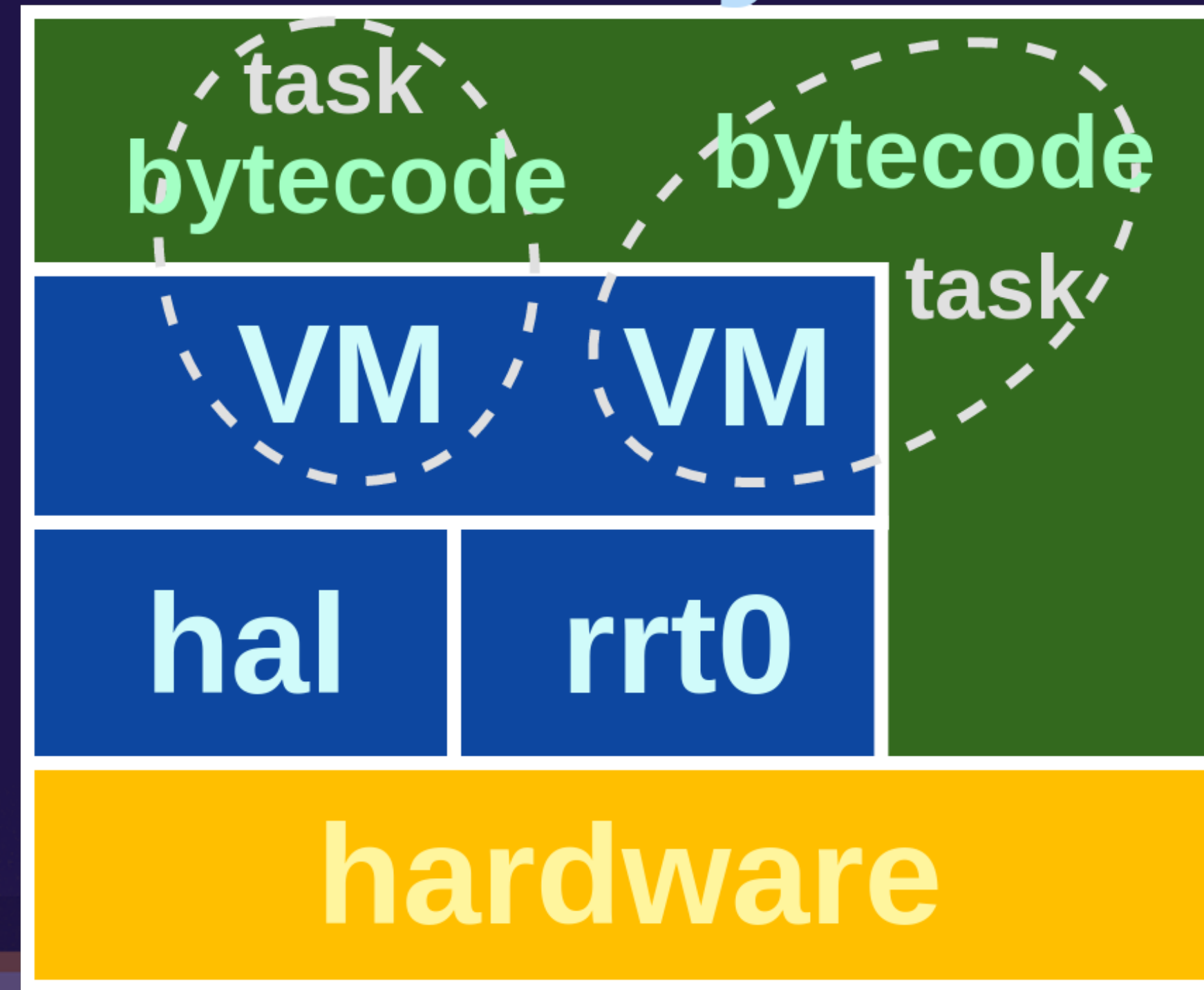
both mruby and mruby/c

- bytecodes are compiled by `mrbc`
virtual machine (VM) executes the bytecode

mruby

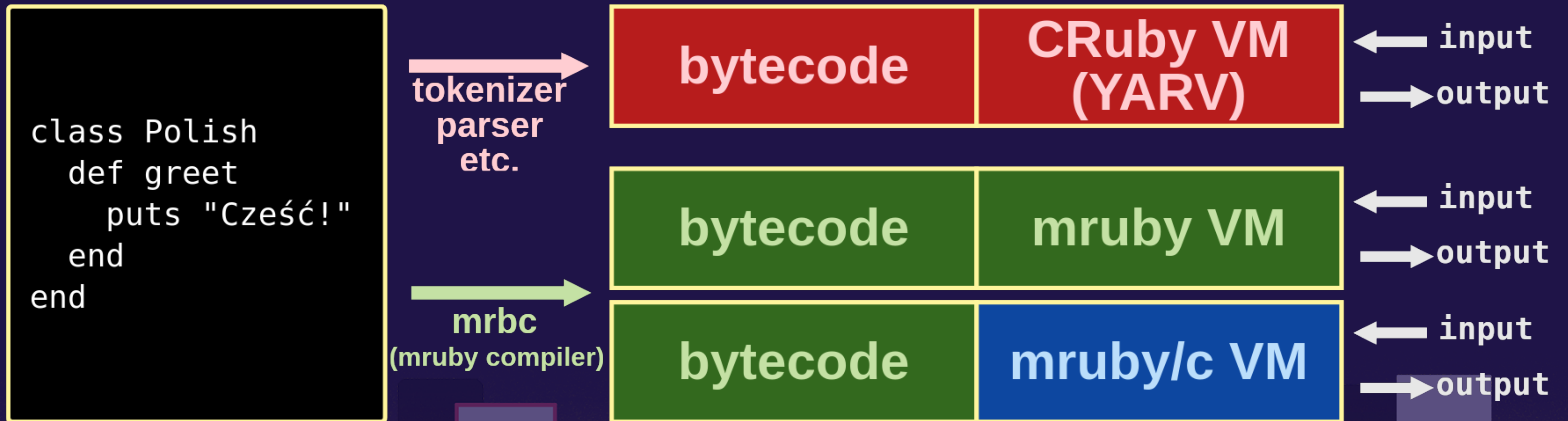


mruby/c



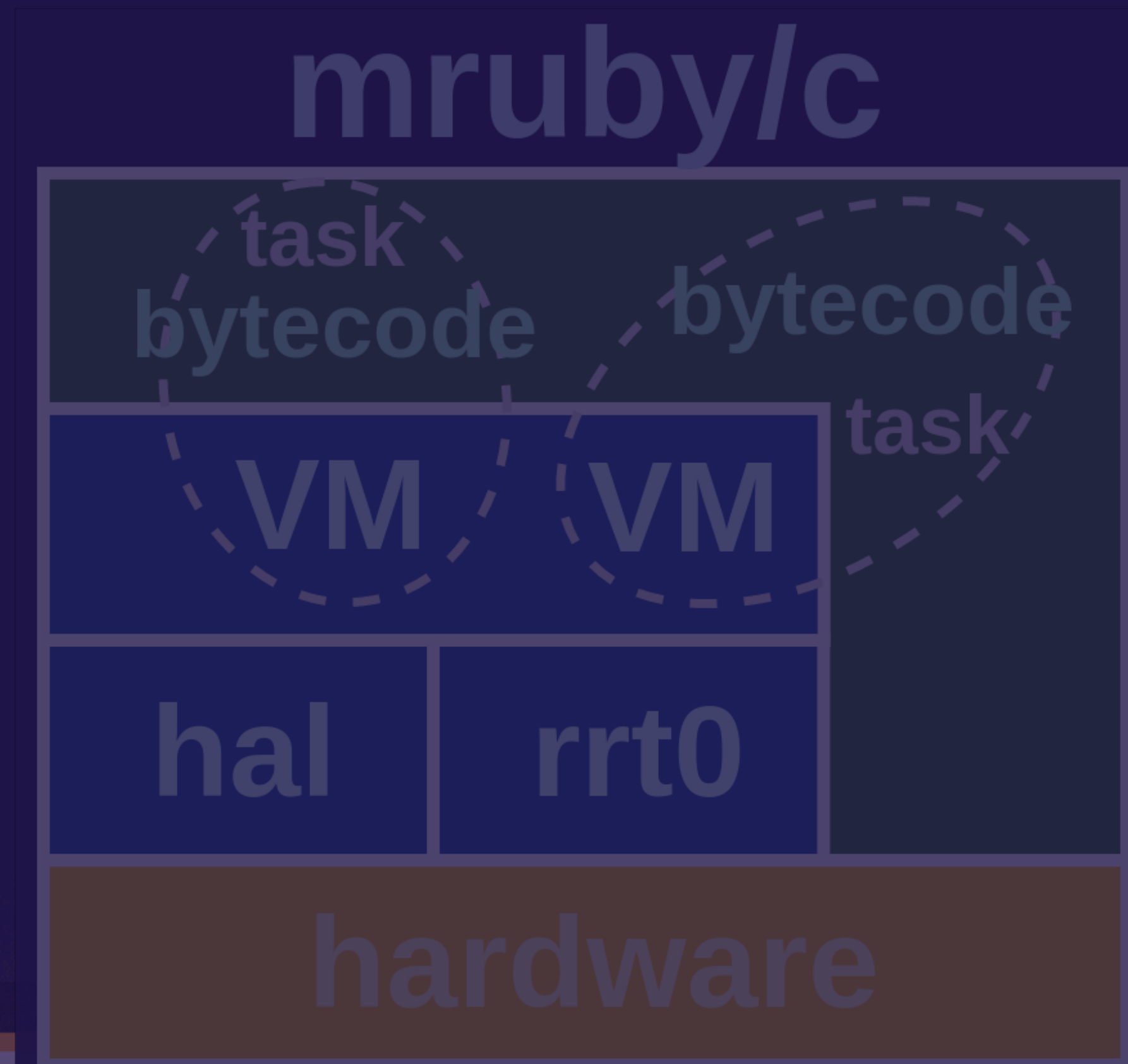
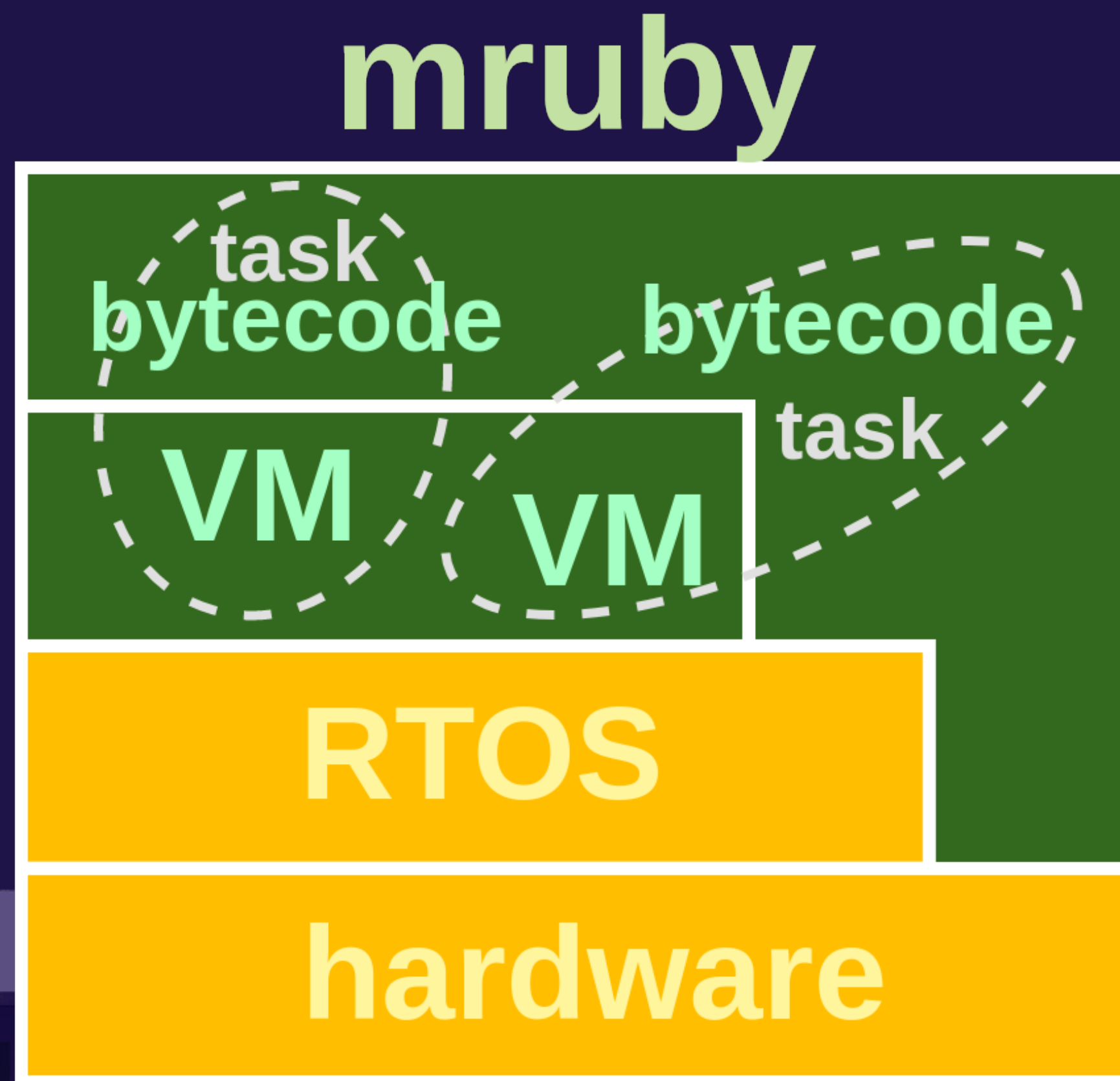
bytecode

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



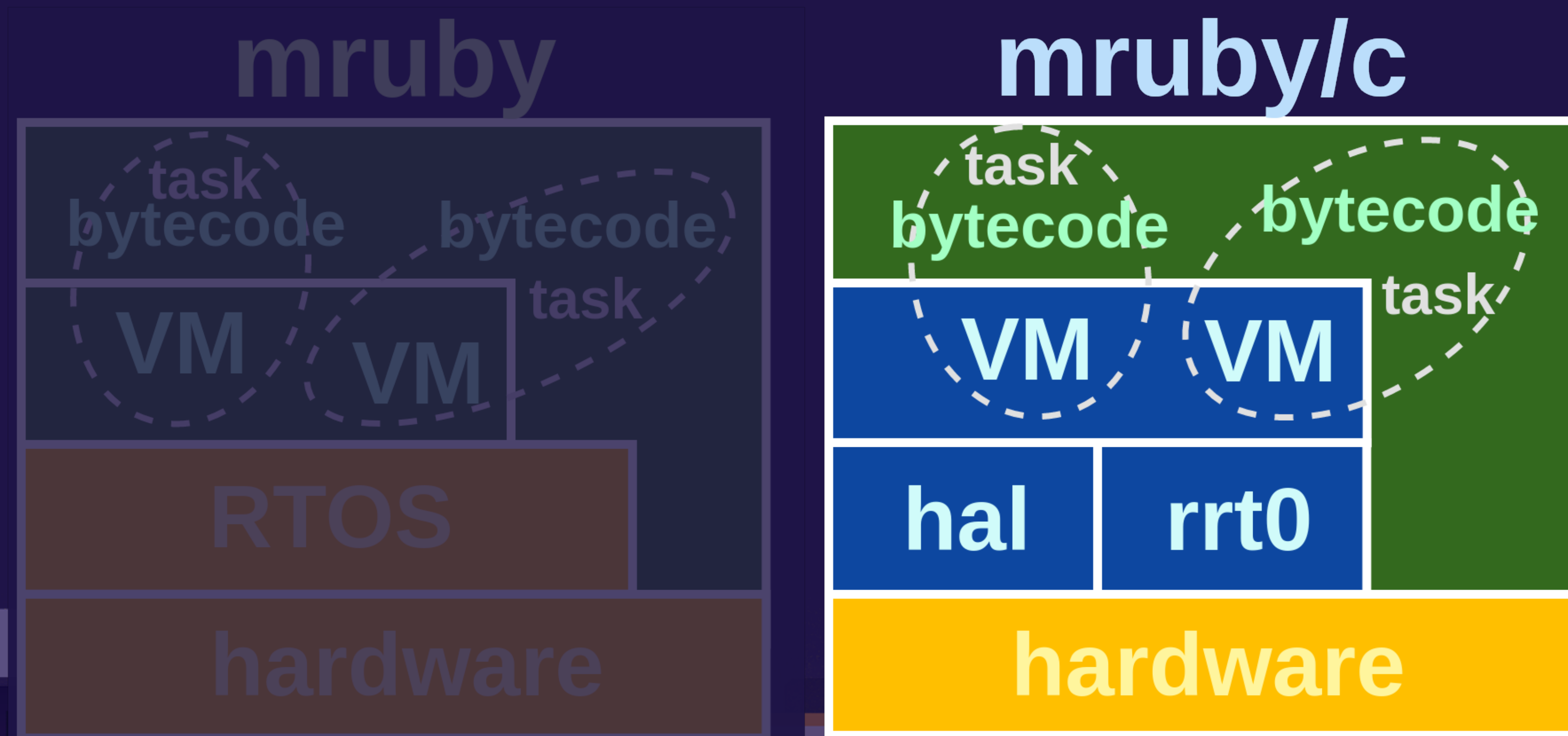
mruby on microcontroller

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



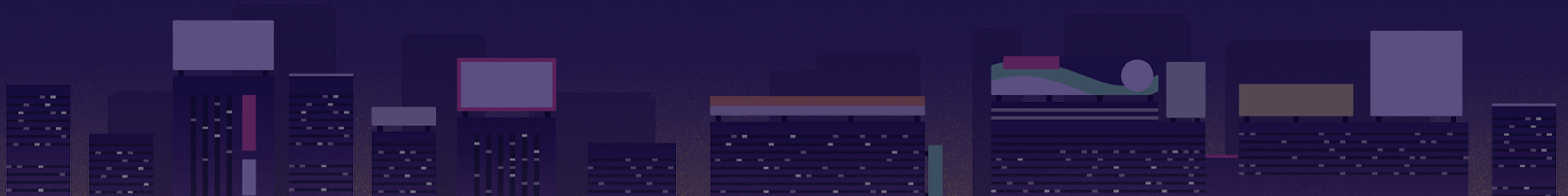
mruby/c on microcontroller

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



mruby/c - virtual machine (VM)

- ⑨ much smaller than mruby's one
 - ⑨ that's why mruby/c runs on smaller RAM
- ⑨ accordingly, mruby/c has **less** functionality than mruby

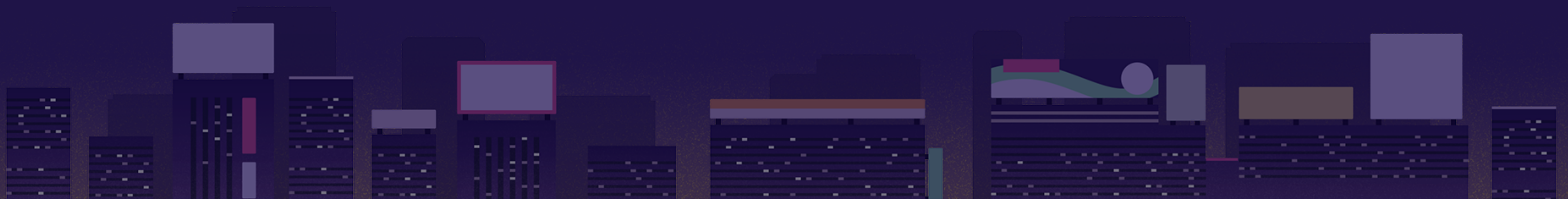


how **less**?



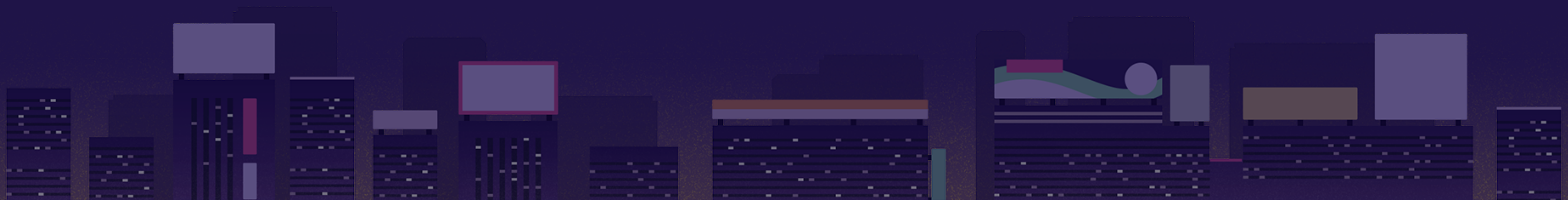
how **less**? - for example

- ◉ 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



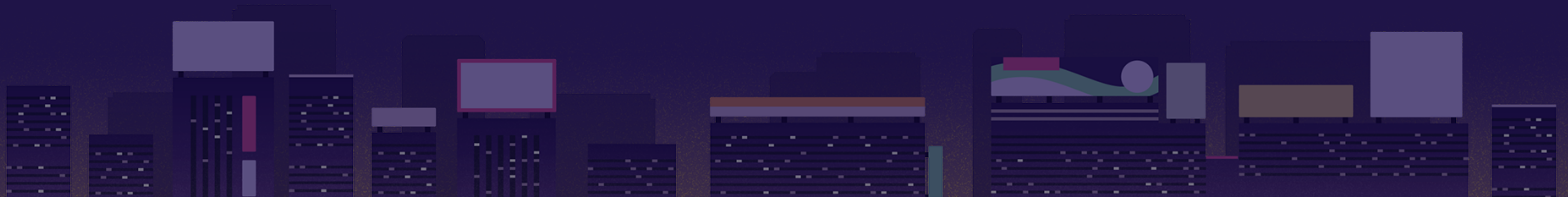
how **less**? - for example

- ◉ mruby/c doesn't have #send, #eval, nor #method_missing
- ◉ moreover, mruby/c neither have your favorite features like TracePoint nor Refinements



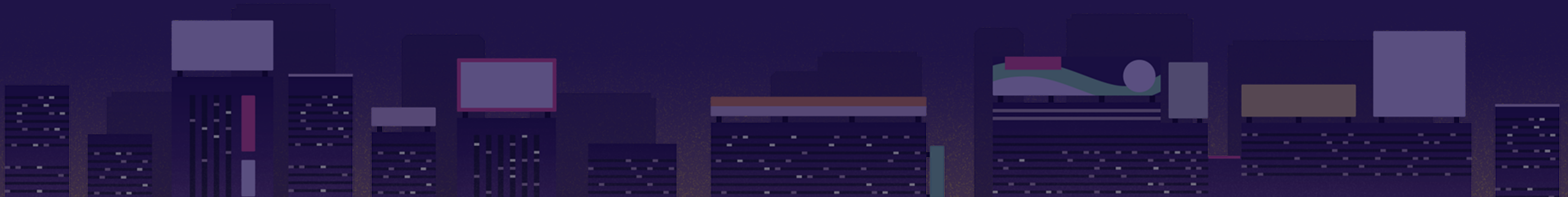
how **less**? - actually

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



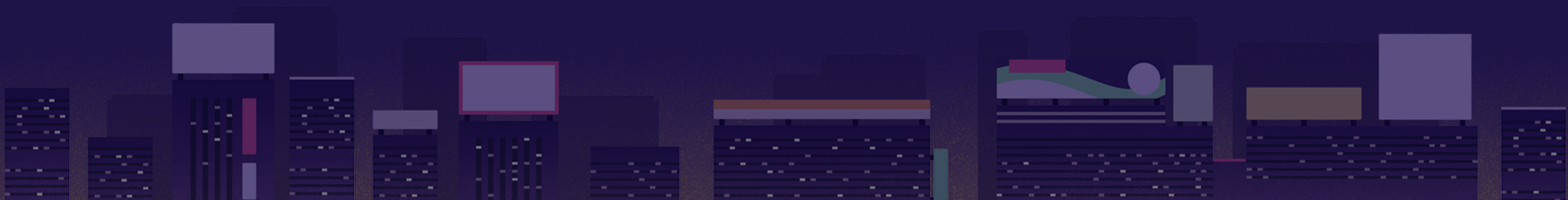
despite the fact,

- ⑨ no problem in practical use of microcontroller
- ⑨ as far as IoT go, mruby/c is enough Ruby as I expect
- ⑨ we can fully develop firmwares with those features of mruby/c



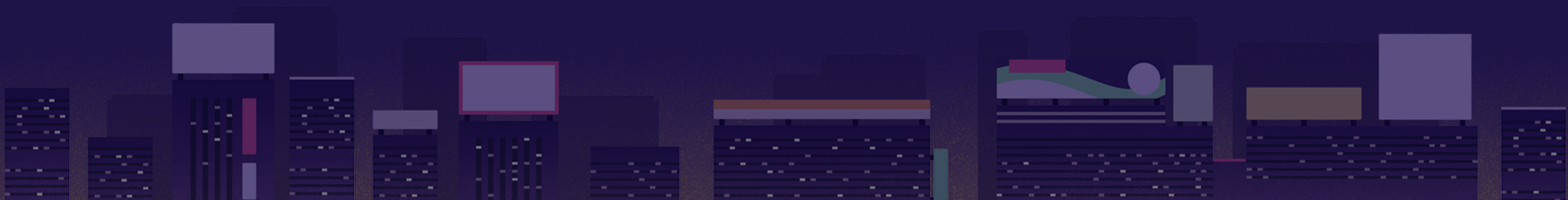
So

というわけで



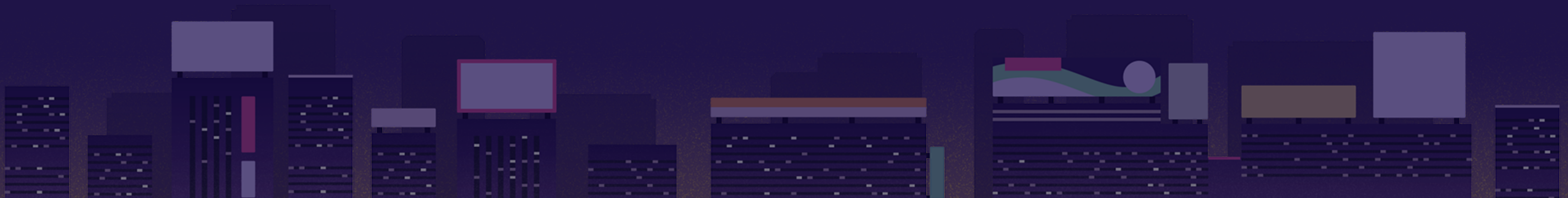
Today's agenda

きょうはこんな話をします



Little more Rubyish

もうちょいRubyっぽくやろう





Matsue.rb

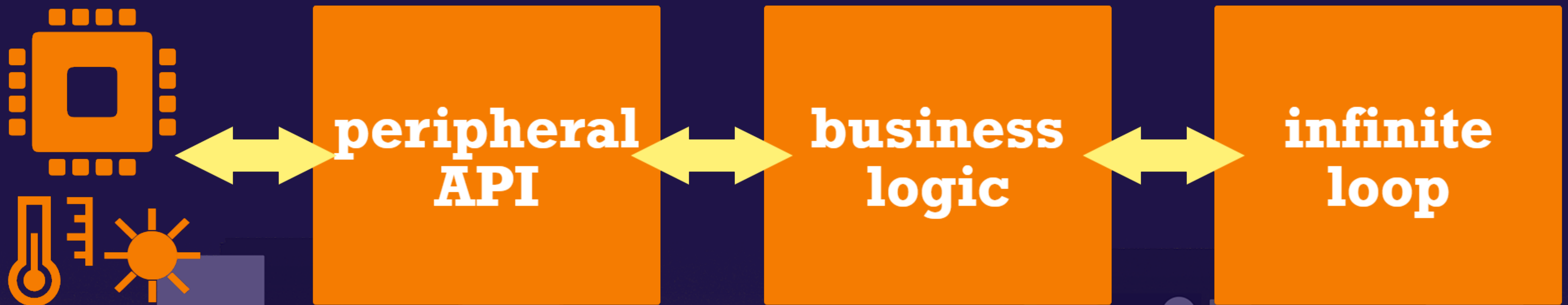
mruby/c firmware is made up of three parts

- ① 1) peripheral API wrapper (C)
- ② 2) business logic (mruby)
- ③ 3) infinite loop (mruby)



mruby/c firmware is made up of three parts

- ① 1) peripheral API wrapper (C)
- ② 2) business logic (mruby) - model
- ③ 3) infinite loop (mruby) - controller



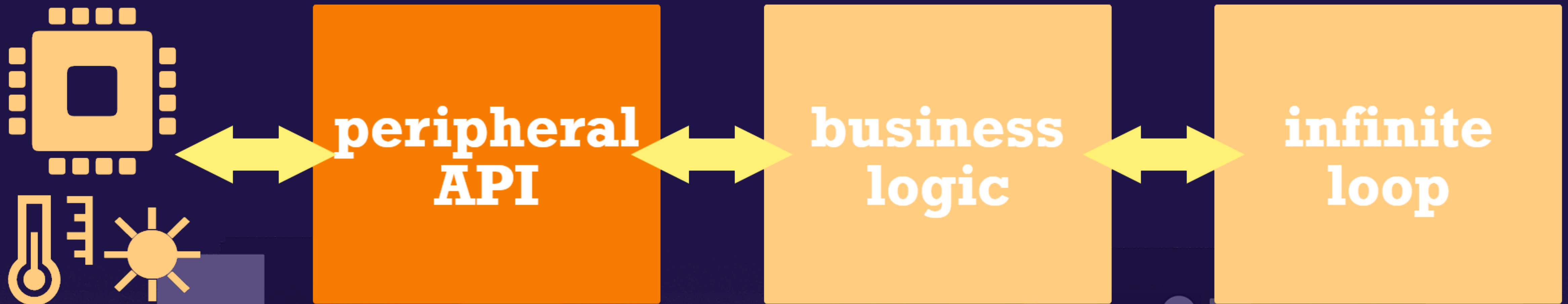
things I want to solve

- ⌚ it's hard to develop if they are tightly coupled
- ⌚ I want to make them being slackly coupled
- ⌚ installing every binary on microcontroller is tiresome



mruby/c firmware is made up of three parts

- ① 1) **peripheral API wrapper (C)**
- ② 2) business logic (mruby)
- ③ 3) infinite loop (mruby)



peripheral API wrapper

🌀 <https://rubykaigi.org/2018>



Hitoshi HASUMI

@hasumon



hammer of Monstar Lab at Shimane

JA

Firmware programming with mruby/c

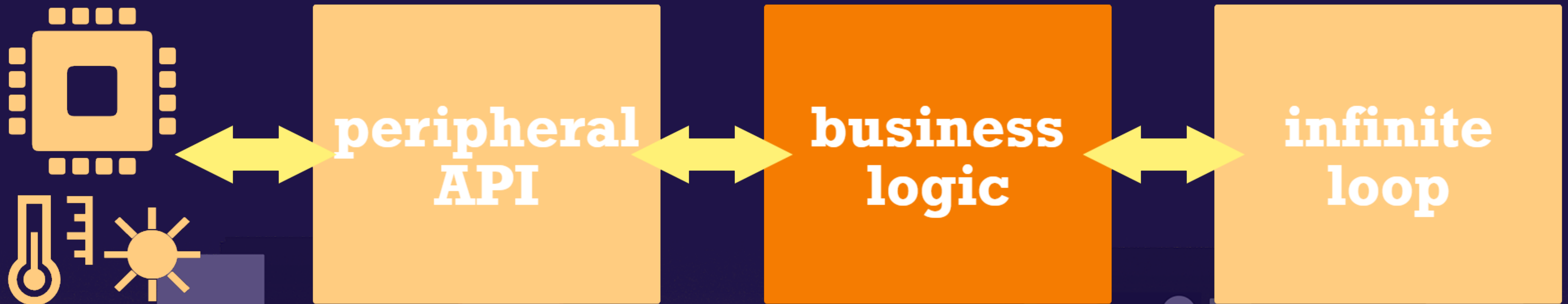
We have a new choice to write firmware for microcomputers(microcontrollers) to introduce mruby/c firmware programming. And besides, my actual IoT project will be described. Since mruby/c is still a young growing tool, you will know there can help it to become better.

Presentation Material

📄 Firmware programming with mruby/c

mruby/c firmware is made up of three parts

- ① 1) peripheral API (C)
- ② 2) **business logic (mruby)**
- ③ 3) infinite loop (mruby)



mruby/c firmware is made up of three parts

```
# infinite loop
foo = Foo.new
while true
  if foo.hoge == 1
    puts "SUCCESS!"
  end
  sleep 1
end
```

```
# business logic
class Foo
  def hoge
    fuga_val = fuga
    c_hoge(fuga_val)
  end
end
```

```
/* peripheral API wrapper */
static void c_hoge(mrb_vm *vm, mrb_value *v, int argc) {
  int result;
  result = peripheral_api_call(GET_INT_ARG(1));
  SET_INT_RETURN(result);
}
```


mruby/c firmware is made up of three parts


```
# infinite loop
```

```
foo = Foo.new
```

```
while true
```

```
  if foo.hoge == 1
```

```
    puts "SUCCESS!"
```

 **keeps waiting
a value**

```
  end  
  sleep 1
```

```
end
```

```
# business logic
```

```
class Foo
```

```
  def hoge
```

```
    fuga_val = fuga
```

```
    c_hoge(fuga_val)
```

```
  end
```

```
end
```

```
/* peripheral API wrapper */
```

```
static void c_hoge(mrb_vm *vm, mrb_value *v, int argc) {
```

```
  int result;
```

```
  result = peripheral_api_call(GET_INT_ARG(1));
```

```
  SET_INT_RETURN(result);
```

```
}
```


mruby/c firmware is made up of three parts

```
# infinite loop
```

```
foo = Foo.new
```

```
while true
```

```
  if foo.hoge == 1
```

```
    puts "SUCCESS!"
```

```
    keeps waiting  
a value
```

```
    sleep 1
```

```
  end
```

```
# business logic
```

```
hits a method
```

```
def hoge
```

```
  fuga_val = fuga
```

```
  c_hoge(fuga_val)
```

```
end
```

```
end
```

```
/* peripheral API wrapper */
```

```
static void c_hoge(mrb_vm *vm, mrb_value *v, int argc) {
```

```
  int result;
```

```
  result = peripheral_api_call(GET_INT_ARG(1));
```

```
  SET_INT_RETURN(result);
```

```
}
```


mruby/c firmware is made up of three parts

```
# infinite loop
```

```
foo = Foo.new
```

```
while true
```

```
  if foo.hoge == 1
```

```
    puts "SUCCESS!"
```

```
    keeps waiting  
a value
```

```
    sleep 1
```

```
  end
```

```
# business logic
```

```
hits a method
```

```
def hoge
```

```
  fuga_val = fuga
```

```
  c_hoge(fuga_val)
```

```
end
```

```
end
```

```
/* peripheral API wrapper */
```

```
static void c_hoge(mrb_vm *vm, mrb_value *v, int argc) {
```

```
forwards to wrapper method
```

```
  result = peripheral_api_call(GET_INT_ARG(1));
```

```
  SET_INT_RETURN(result);
```

```
}
```


mruby/c firmware is made up of three parts

```
# infinite loop
```

```
foo = Foo.new
```

```
while true
```

```
  if foo.hoge == 1
```

```
    puts "SUCCESS!"
```

```
    sleep 1
```

```
  end
```



**keeps waiting
a value**

```
# business logic
```

hits a method

```
def hoge
```

```
  fuga_val = fuga
```

```
  c_hoge(fuga_val)
```

```
end
```

```
end
```

```
/* peripheral API wrapper */
```

```
static void c_hoge(mrb_vm *vm, mrb_value *v, int argc) {
```

forwards to wrapper method

```
  result = peripheral_api_call(GET_INT_ARG(1));
```

```
  SET_INT_RETURN(result);
```

```
}
```

**at last peripheral
library is called**

by the way,

```
# infinite loop
```

```
foo = Foo.new
```

```
while true
```

```
  if foo.hoge == 1
```

```
    puts "SUCCESS!"
```

```
    sleep 1
```

```
  end
```



**keeps waiting
a value**

```
# business logic
```

hits a method

```
def hoge
```

```
  fuga_val = fuga
```

```
  c_hoge(fuga_val)
```

```
end
```

```
end
```

```
/* peripheral API wrapper */
```

```
static void c_hoge(mrb_vm *vm, mrb_value *v, int argc) {
```

```
  int result;
```

```
  result = peripheral_api_call(GET_INT_ARG(1));
```

```
  SET_INT_RETURN(result);
```

```
}
```


fuga?

```
# infinite loop
```

```
foo = Foo.new
```

```
while true
```

```
  if foo.hoge == 1
```

```
    puts "SUCCESS!"
```

```
    sleep 1
```

```
  end
```

**keeps waiting
a value**



```
# business logic
```

hits a method

```
def hoge
```

```
  fuga_val = fuga
```

```
  c_hoge(fuga_val)
```

```
end
```

```
end
```

```
/* peripheral API wrapper */
```

```
static void c_hoge(mrb_vm *vm, mrb_value *v, int argc) {
```

```
  int result;
```

```
  result = peripheral_api_call(GET_INT_ARG(1));
```

```
  SET_INT_RETURN(result);
```

```
}
```


fuga?

```
# infinite loop
```

```
foo = Foo.new
```

```
while true
```

```
  if foo.hoge == 1
```

```
    puts "SUCCESS!"
```

```
    sleep 1
```

```
  end
```

**keeps waiting
a value**



```
# business logic
```

fuga

```
end
```

```
/* peripheral API wrapper */
```

```
static void c_hoge(mrb_vm *vm, mrb_value *v, int argc) {
```

```
  int result;
```

```
  result = peripheral_api_call(GET_INT_ARG(1));
```

```
  SET_INT_RETURN(result);
```

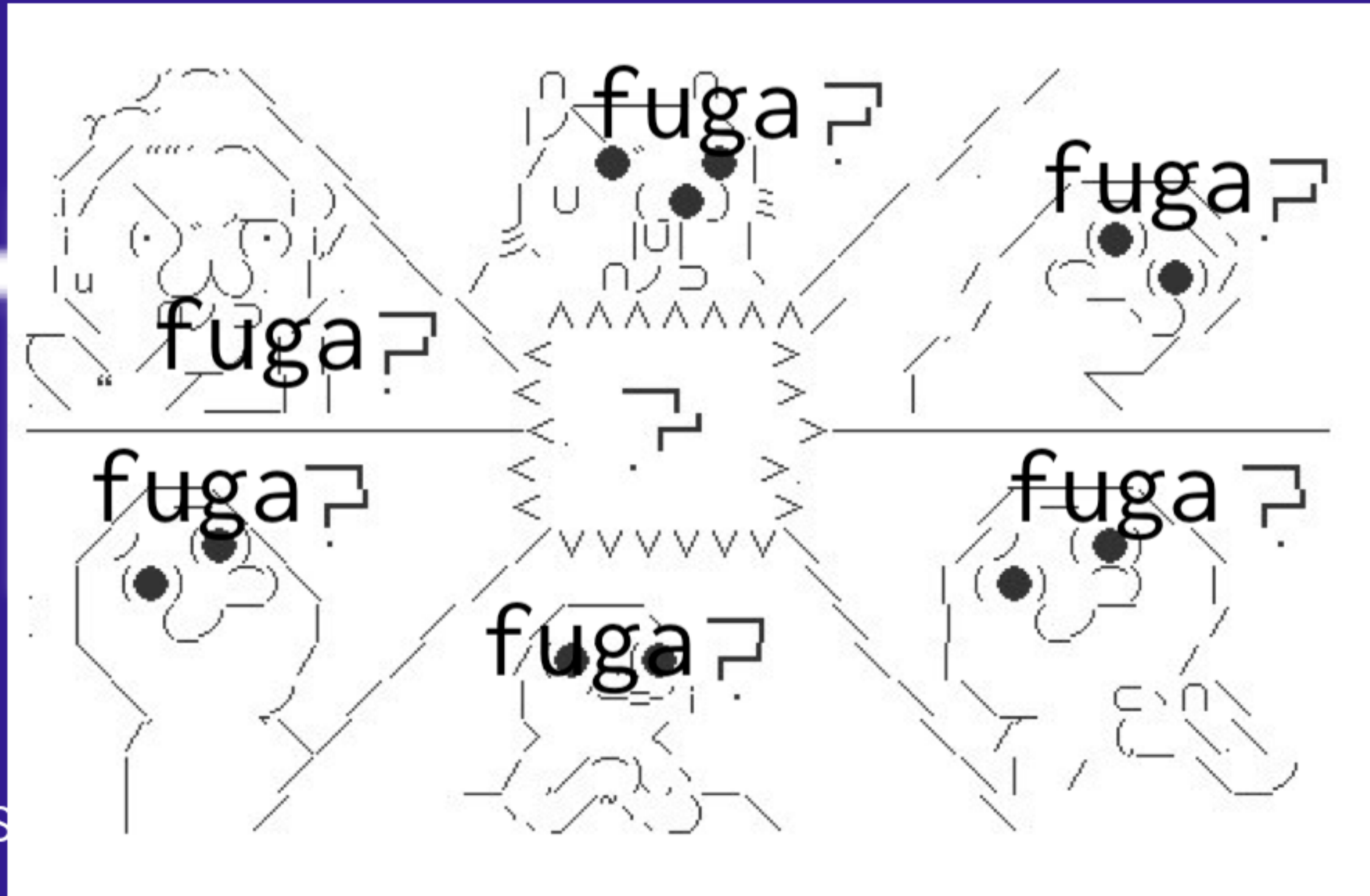
```
}
```


fuga?

fuga

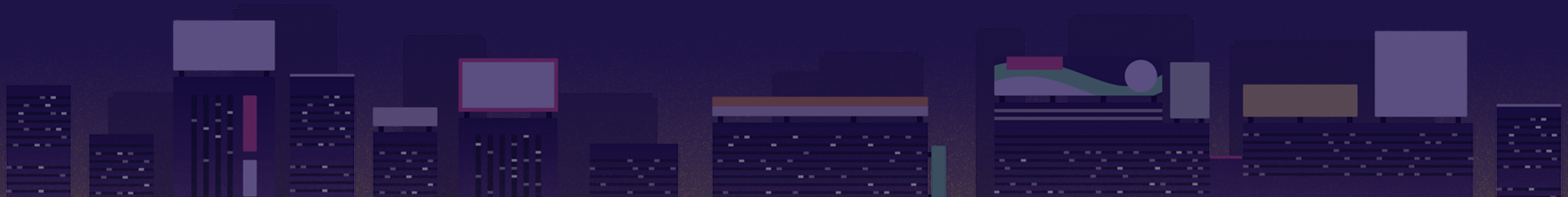
```
SET_INT_RETURN(result);  
}
```


what is fuga?

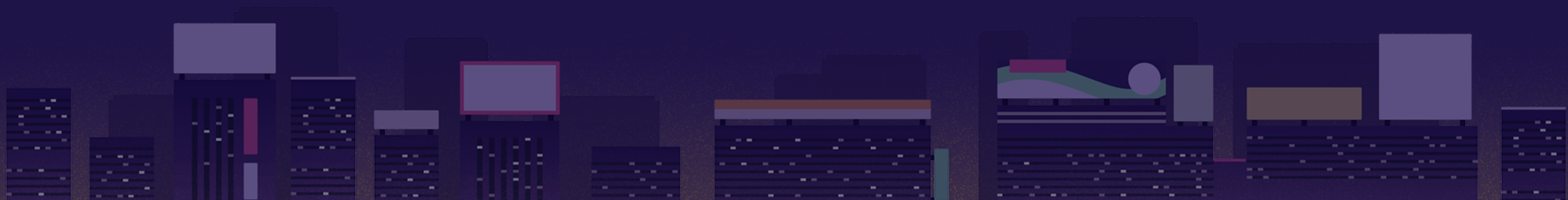


methods still not implemented

- ⌚ we often should write business logic without hitting peripherals
 - ⌚ it will cost a lot in some case
 - ⌚ it is possible the design of peripheral details might not be finished yet
- ⌚ what you expect in this situation?



Stub



Mock

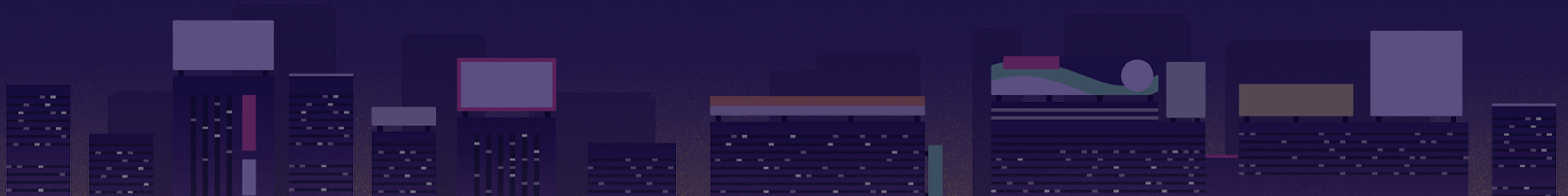


Test Driven Development for Embedded Ruby



when I started to use mruby/c

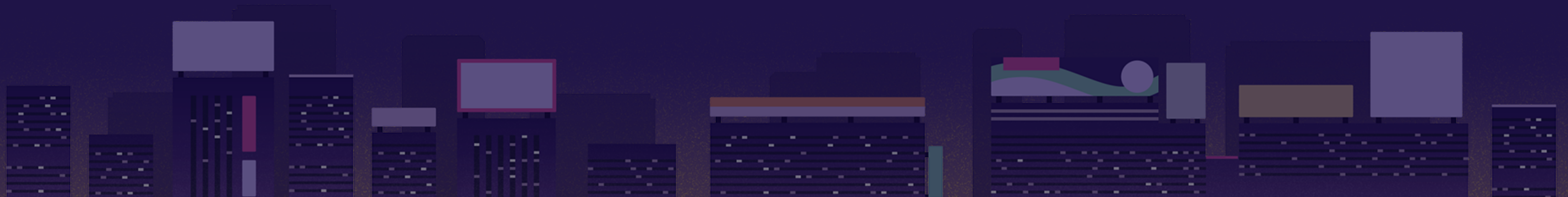
- ⑨ there is no **testing tool**
- ⑨ I had difficulties of writing my application
- ⑨ even mruby/c itself sometimes regressed 🤖



so, why did I use mruby/c?



DESTINO - 運命

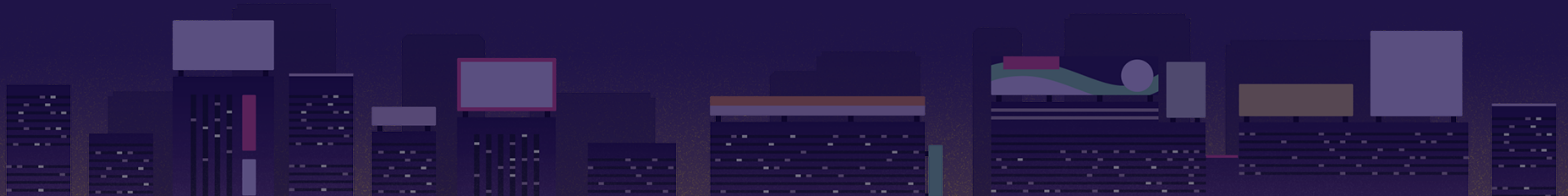


Anyway, I started to create
mrubyc-test.gem



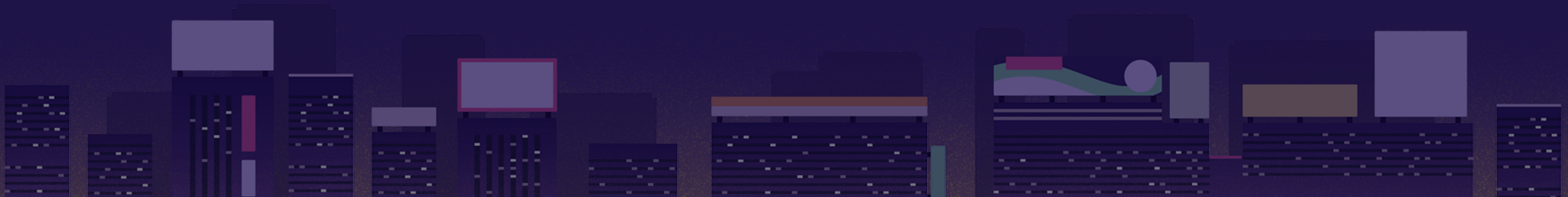
mrubyc-test.gem - the aim

- 🌀 I wanted to go Rubyish
- 🌀 but mruby/c doesn't have enough features to make testing tool as you saw just before



mrubyc-test.gem - designed as

- ⑨ RubyGem, implemented in CRuby instead of mruby
- ⑨ Test::Unit-like API
- ⑨ supports stub and mock
 - ⑨ now you can test your business logic without implementing peripheral functions like **#fuga**



mrubyc-test.gem - stub

```
# app code
class Sample
  attr_accessor :result
  def do_something(arg)
    @result = arg + still_not_defined_method
  end
end

# test code
def stub_case
  sample_obj = Sample.new
  stub(sample_obj).still_not_defined_method { " , so we are nice" }
  sample_obj.do_something("Ruby is nice")
  assert_equal "Ruby is nice, so we are nice", sample_obj.result
end
```

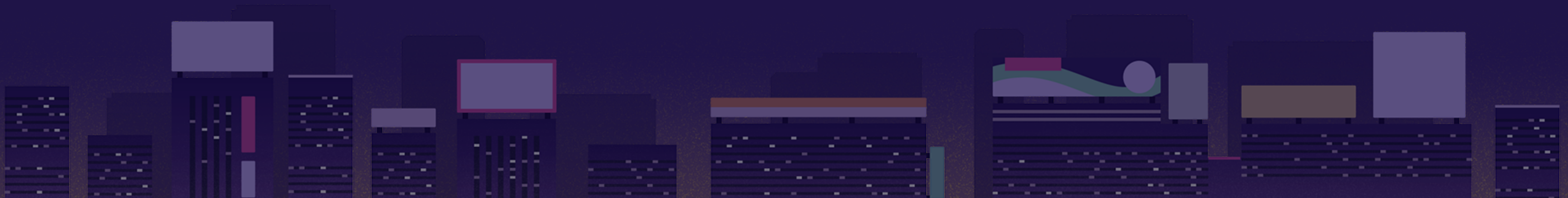

mrubyc-test.gem - mock

```
# app code
class Sample
  def do_other_thing
    to_be_hit()
  end
end

# test code
def mock_case
  sample_obj = Sample.new
  mock(sample_obj).to_be_hit
  sample_obj.do_other_thing
end
```

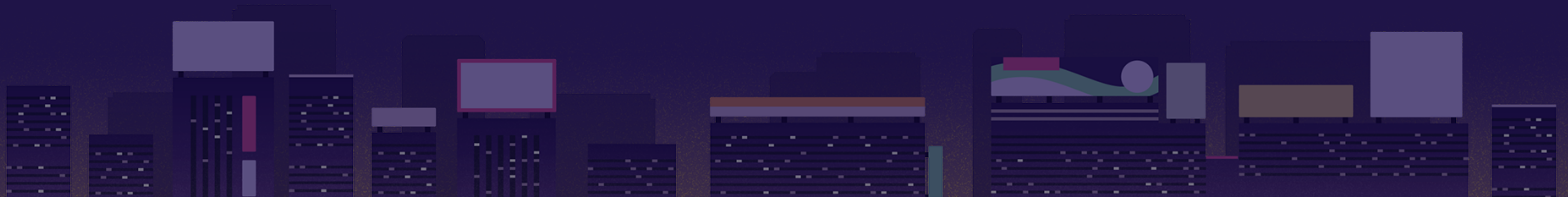
it was my personal tool

github.com/hasumikin/mrubyc-test



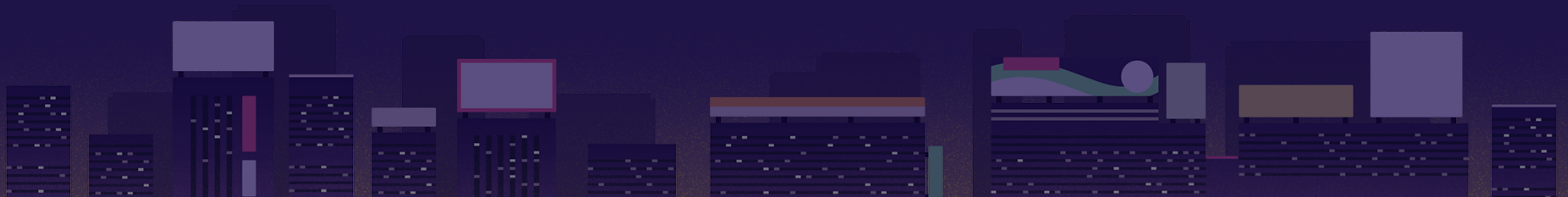
but already abandoned because

~~github.com/hasumikin/mrubyc-test~~



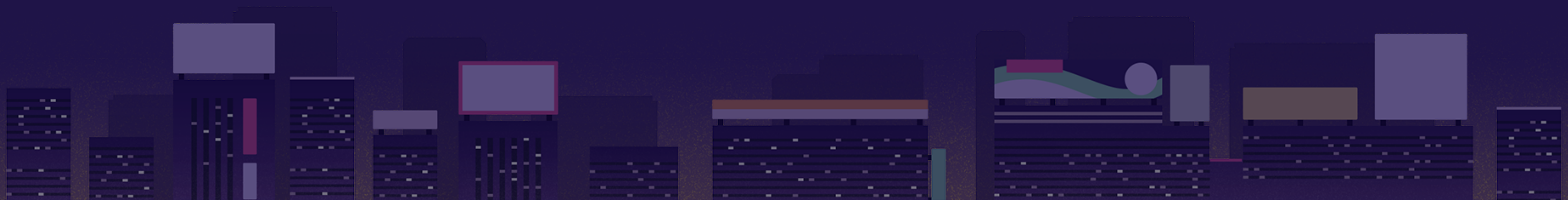
now it's official 🎉

github.com/mruby/mruby-test



mrubyc-test.gem

- 🌀 adopted as the testing tool for mruby/c itself
 - 🌀 so now you can safely send pull request to github.com/mruby/mruby
 - 🌀 you can write mruby/c application with confidence



mrubyc-test.gem - internal

- the gist is creating **test.rb** by `test code generator` implemented in CRuby

```
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
```

test code
generator

test.rb

mruby
compiler

test.c
bytecode

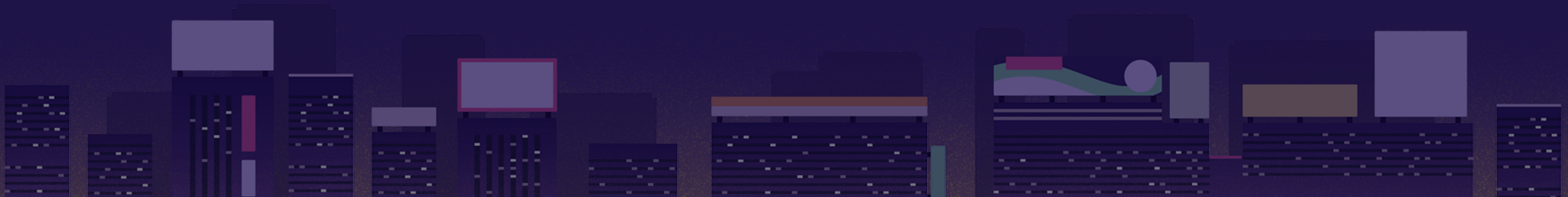
C
compiler

test executable

mruby/c VM

mrubyc-test.gem - making the test.rb

- ⑨ gathers information of test cases by `#method_added`
 - ⑨ I learned this technique from `Test::Unit`
- ⑨ generates stub methods and mock methods
- ⑨ makes all-in-one script: **test.rb**
 - ⑨ all the indispensable mechanism of assertion, stub, mock, app code and test code get together



mrubyc-test.gem - Module#method_added

```
class MrubycTestCase
  def self.method_added(name)
    return false if %i(method_missing setup teardown).include?(name)
    location = caller_locations(1, 1)[0]
    path = location.absolute_path || location.path
    line = location.lineno
    @@added_methods << {
      method_name: name.to_s,
      path:       File.expand_path(path),
      line:       line
    }
  end
end
```


mrubyc-test.gem

```
class SampleTest < MrubycTestCase
  desc "stub test sample"
  def stub_case # hooks #method_added
    sample_obj = Sample.new
    stub(sample_obj).still_not_defined_method {
      ", so we are nice"
    }
  end
end
```

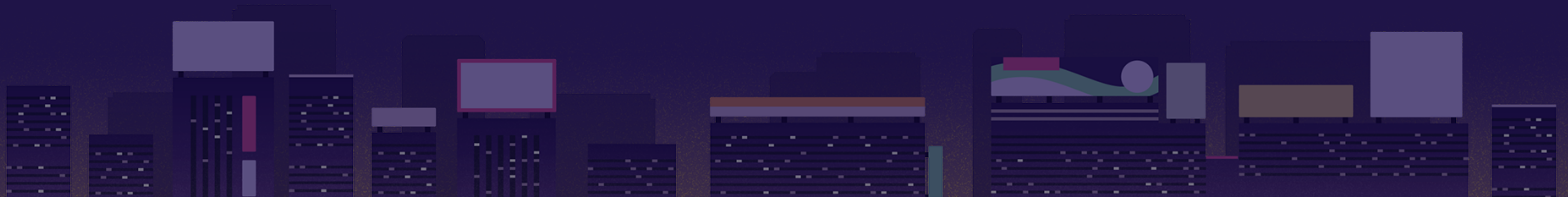
⑨ test code inherits MrubycTestCase to be analyzed

mrubyc-test.gem - BasicObject#method_missing

```
class MrubycTestCase
  def method_missing(method_name, *args)
    case method_name
    when :stub, :mock
      location = caller_locations(1, 1)[0]
      Mrubyc::Test::Generator::Double.new(
        method_name, args[0], location
      )
    end
  end
end
```


mrubyc-test.gem - generated stub method

```
# part of test.rb  
class Sample  
  def still_not_defined_method  
    ", so we are nice"  
  end  
end
```



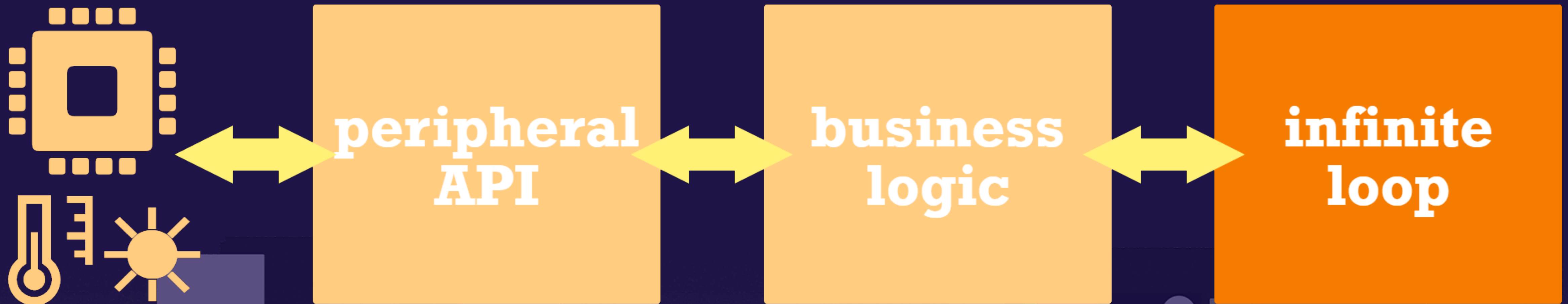
mrubyc-test.gem - template of stub

```
<% test_cases.each do |test_case| -%>
  <% test_case[:stubs].each do |stub| -%>
    class <%= stub[:class_name] %>
      attr_accessor <%= stub[:instance_variables] %>
      def <%= stub[:method_name] %>
        <% if stub[:return_value].is_a?(String) -%>
          "<%= stub[:return_value] %>"
        <% else -%>
          <%= stub[:return_value] %>
        <% end -%>
      end
    end
  end
<% end -%>
```




mruby/c firmware is made up of three parts

- ① 1) peripheral API (C)
- ② 2) business logic (mruby)
- ③ 3) **infinite loop (mruby)**



mruby/c firmware is made up of three parts

```
# infinite loop
```

```
foo = Foo.new
```

```
while true
```

```
  if foo.hoge == 1
```

```
    puts "SUCCESS!"
```

```
    sleep 1
```

```
  end
```



**keeps waiting
a value**

```
# business logic
```

hits a method

```
def hoge
```

```
  fuga_val = fuga
```

```
  c_hoge(fuga_val)
```

```
end
```

```
end
```

```
/* peripheral API wrapper */
```

```
static void c_hoge(mrb_vm *vm, mrb_value *v, int argc) {
```

forwards to wrapper method

```
  result = peripheral_api_call(GET_INT_ARG(1));
```

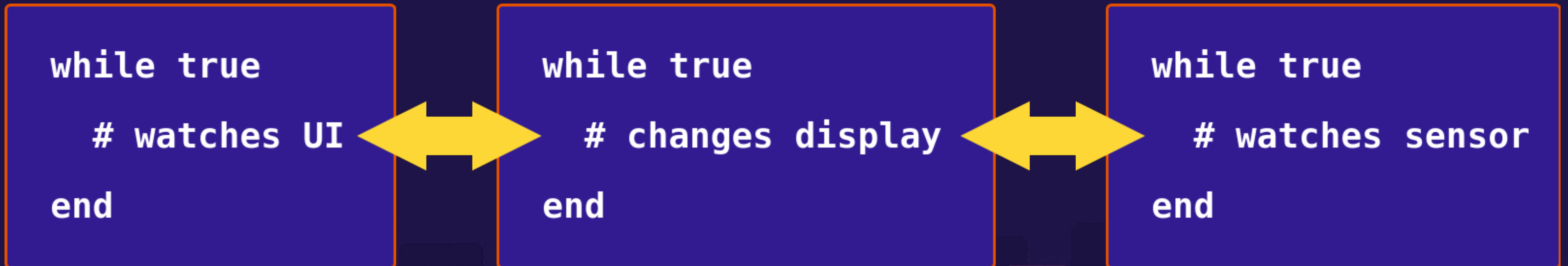
```
  SET_INT_RETURN(result);
```

```
}
```

**at last peripheral
library is called**

we have multiple infinite loops

- firmware programming is essentially thread programming which consists of multiple infinite loops
- they keep watch on status like user input, changing sensor value and BLE/WiFi message, then display some information to indicate internal status



the loops of mruby/c are

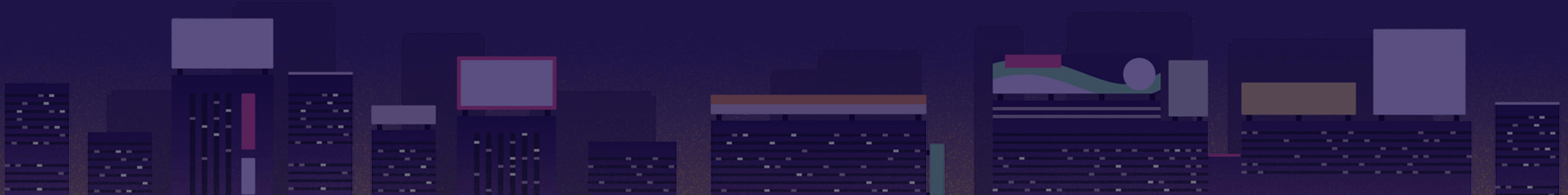
- ⑨ user space threads managed by mruby/c's runtime

```
/* main.c */
#define MEMORY_SIZE (1024 * 40) /* 40KB */
static uint8_t mrubyc_vm_pool[MEMORY_SIZE];
int main(void) {
    mrbc_init(mrubyc_vm_pool, MEMORY_SIZE);
    mrbc_create_task(watch_user_interface, 0);
    mrbc_create_task(change_display, 0);
    mrbc_create_task(watch_sensor_value, 0);
    mrbc_run();
}
```

threads of CRuby

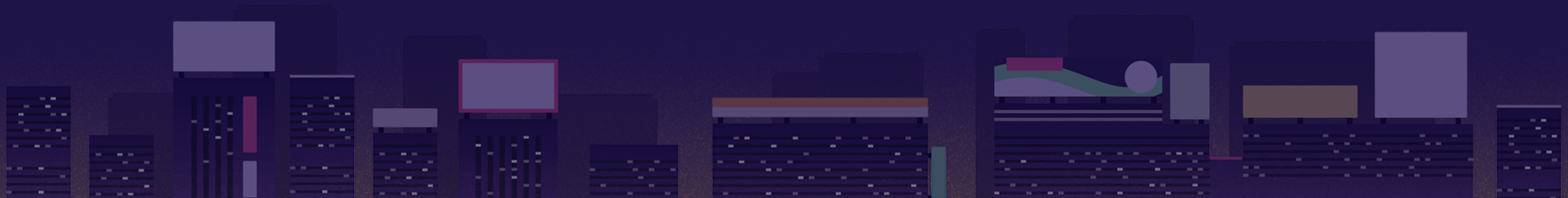
☞ correspond to native threads (with GVL)

```
def start_loops
  threads = []
  threads << Thread.new { watch_user_interface }
  threads << Thread.new { change_display }
  threads << Thread.new { watch_sensor_value }
  threads.each(&:join)
end
```



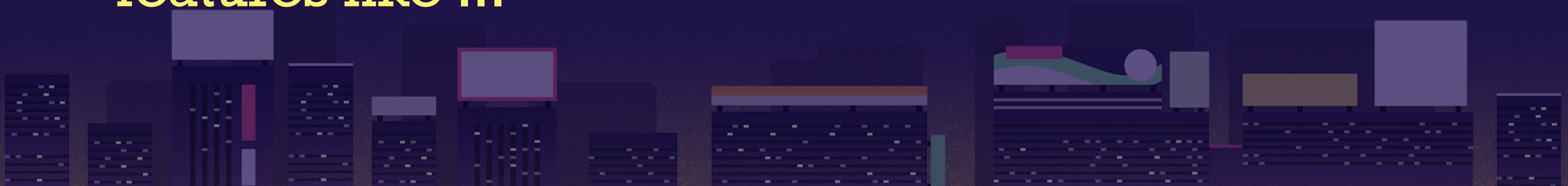
(DEMO)

github.com/hasumikin/mrubyc-debugger



mrubyc-debugger.gem - threads

- ④ mrubyc-debugger runs mruby/c loop script as a CRuby thread
- ④ it simultaneously shows which lines are being executed
- ④ besides, it have to take over the debug print of the script
- ④ in order to do that, we can use your favorite CRuby features like ...



TracePoint



mrubyc-debugger.gem - TracePoint

```
tasks = Dir.glob(File.join(Dir.pwd, "mrubyc_loops_dir", "*.rb"))
TracePoint.new(:c_call, :call, :line) do |tp|
  number = nil
  caller_locations(1, 1).each do |caller_location|
    tasks.each_with_index do |task, i|
      number = i if caller_location.to_s.include?(File.basename(task))
    end
  end
  if number
    @@mutex.lock
    event = {
      method_id:      tp.method_id,
      lineno:         tp.lineno,
      caller_location: caller_location,
      binding:         tp.binding }
    $event_queues[number].push event
    @@mutex.unlock
  end
end
```


Refinements

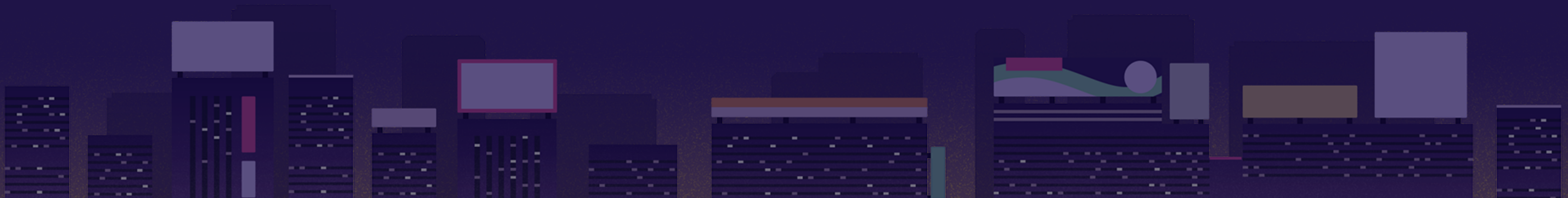


mrubyc-debugger.gem - Refinements

```
module DebugQueue
  refine Kernel do
    def puts(text)
      $debug_queues[Thread.current[:index]] << {
        level: :debug,
        body: text }
    end
  end
end
```

- ◉ assuming mruby/c loops use `#puts` for print debug on serial console,
- ◉ mrubyc-debugger takes it over to print on Curses window

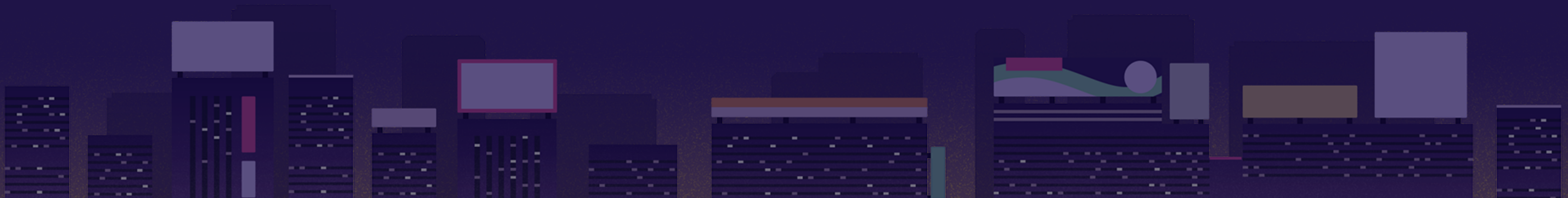
Curses



mrubyc-debugger.gem - Curses

```
include Curses
debug = $debug_queues[i].pop # took over by Refinements
wins[i][:out].addstr " #{debug[:level]} " + debug[:body]
event = $event_queues[i].pop # event info by TracePoint
(1..(wins[i][:src].maxy - 2)).each do |y|
  wins[i][:src].setpos(y, 1)
  if !@srcs[i][y]
    wins[i][:src].addstr ' ' * wins[i][:src].maxx
  else
    # highlight current line
    wins[i][:src].attron(A_REVERSE) if y == event[:lineno]
  end
end
end
vars = {}
event[:tp_binding].local_variables.each do |var|
  vars[var] = event[:tp_binding].local_variable_get(var).inspect
end
```


Binding



mrubyc-debugger.gem - Binding

```
binding.local_variables
```

```
# => [:var_a, :var_b, ...]
```

```
binding.local_variable_get(:var_a)
```

```
# => "foo"
```

```
binding.local_variable_set(:var_a, "bar")
```

```
binding.local_variable_get(:var_a)
```

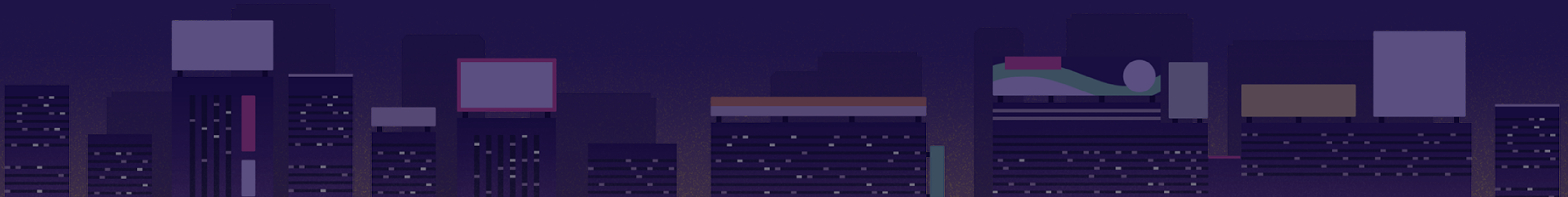
```
# => "bar"
```


conclusion



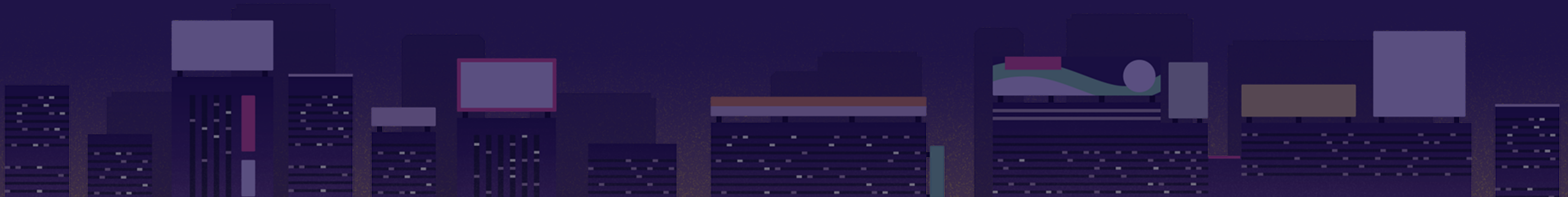
conclusion

- ⑨ at a glance, developing with mruby/c seems to be very restricted due to lack of dynamic features



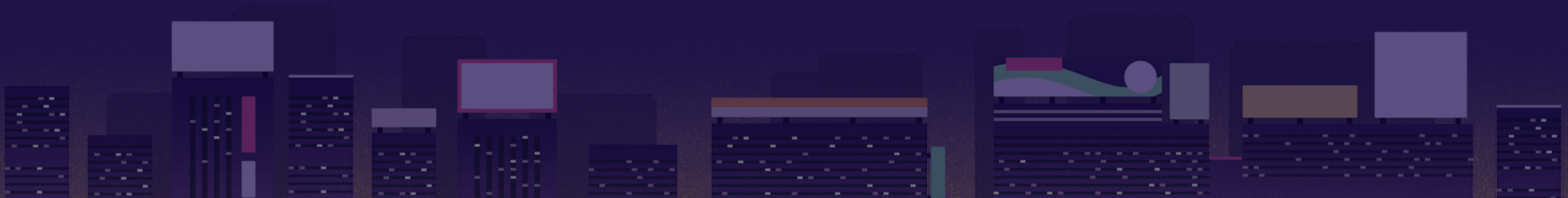
conclusion

- ⑨ at a glance, developing with mruby/c seems to be very restricted due to lack of dynamic features
- ⑨ however, it will be more effective by using the power of CRuby and our own tools



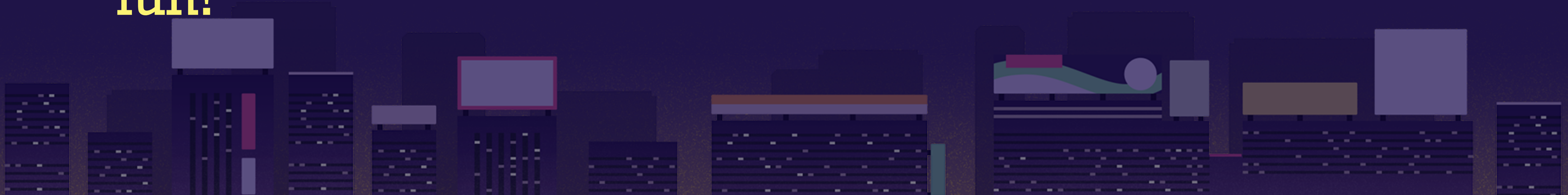
conclusion

- ⑨ at a glance, developing with mruby/c seems to be very restricted due to lack of dynamic features
- ⑨ however, it will be more effective by using the power of CRuby and our own tools
- ⑨ an ecosystem including TDD for embedded Ruby firmware is ready for production



conclusion

- ④ at a glance, developing with mruby/c seems to be very restricted due to lack of dynamic features
- ④ however, it will be more effective by using the power of CRuby and our own tools
- ④ an ecosystem including TDD for embedded Ruby firmware is ready for production
- ④ above all, Rubyish-terminal-based development is fun!



me

- ⑨ HASUMI Hitoshi
@hasumikin
- ⑨ Monstar Lab, inc.
Shimane office
- ⑨ Sake 🍶
Soba 🍜
Coffee ☕



Thank you!

