

REYKJAVÍK UNIVERSITY

MECHATRONICS I

T-411-MECH



INDIVIDUAL ASSIGNMENT 3

02/09/2020

Student:

Giacomo Menchi

Teacher:

Joseph T. Foley

Contents

1	Tasks	1
1.1	Task 1	1
1.2	Task 2	2
1.3	Task 3	2
1.4	Task 4	3
1.5	Task 5	4
1.6	Task 6	8
1.7	Task 7	8
1.8	Task 8	9
1.9	Task 9	9
1.10	Task 10	10
1.11	Video Proof	10

1 Tasks

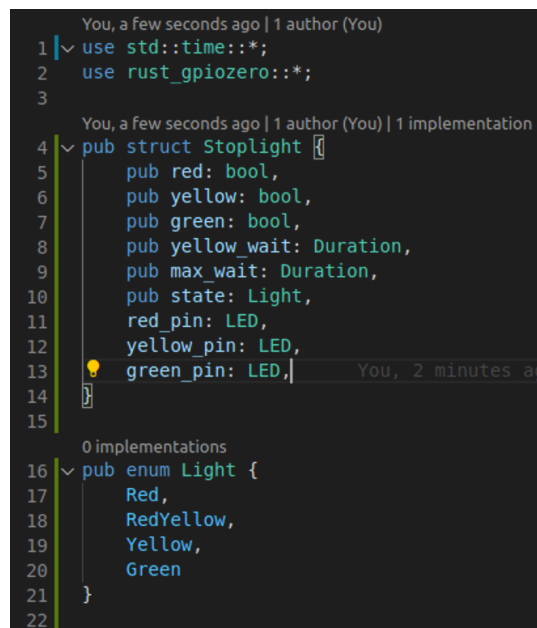
1.1 Task 1

First of all, the definitive version of this stoplight assignment is available on my Github repo:

<https://shorturl.at/jvyKS>

The first task consisted in setting up the Struct containing:

- A boolean value for each light.
- A value for the wait time on a yellow light.
- A value for the maximum waiting time for the light.
- The state of the state machine.
- The pins themselves (as private values).



```
1 | use std::time::*;
2 | use rust_gpiozero::*;
3 |
4 | pub struct Stoplight {
5 |     pub red: bool,
6 |     pub yellow: bool,
7 |     pub green: bool,
8 |     pub yellow_wait: Duration,
9 |     pub max_wait: Duration,
10 |    pub state: Light,
11 |    red_pin: LED,
12 |    yellow_pin: LED,
13 |    green_pin: LED,
14 | }
15 |
16 | pub enum Light {
17 |     Red,
18 |     RedYellow,
19 |     Yellow,
20 |     Green
21 | }
```

Figure 1: Screenshot of the Struct and the state Enum.

1.2 Task 2

The second task had me create a function that initialises the Struct with default values for everything but the maximum waiting time which has to be set using the function's input parameter.

```
28 impl Stoplight{
29     pub fn init(max_wait: Duration) -> Stoplight {
30         Stoplight{
31             red: true,
32             yellow: false,
33             green: false,
34             yellow_wait: Duration::new(secs: 3, nanos: 0),
35             max_wait,
36             state: State::Red,
37             red_pin: LED::new(pin: 4),
38             yellow_pin: LED::new(pin: 6),
39             green_pin: LED::new(pin: 20),
40         }
41     }
42 }
```

Figure 2: Screenshot of the init function.

1.3 Task 3

The third task was creating a function that contains the state machine functionality and have the state variable encode the light state.

```
43 fn state_machine(&mut self) {
44     match self.state {
45         State::Red => {
46             self.state = State::RedYellow;
47             self.yellow_pin.on();
48             thread::park_timeout(dur: self.yellow_wait);
49         }
50         State::RedYellow => {
51             self.state = State::Green;
52             self.red_pin.off();
53             self.yellow_pin.off();
54             self.green_pin.on();
55             thread::park_timeout(dur: self.max_wait);
56         }
57         State::Green => {
58             self.state = State::Yellow;
59             self.green_pin.off();
60             self.yellow_pin.on();
61             thread::park_timeout(dur: self.yellow_wait);
62         }
63         State::Yellow => {
64             self.state = State::Red;
65             self.yellow_pin.off();
66             self.red_pin.on();
67             thread::park_timeout(dur: self.max_wait);
68         }
69     }
70 }
71 }
```

Figure 3: Screenshot of the init function.

To achieve this, I wrote a function that evaluated the current FSM state (by using "match") and made it shift to the following one, by modifying the state variable, turning on/off the related pins and also making the program wait for the right time before proceeding to the next state (the waiting time for the yellow light "yellow_wait" and maximum waiting time "max_wait" exist for this reason).

1.4 Task 4

The fourth task prompted me to create a function that steps the state machine when run and run it in the main code with a fixed time offset.

```
73 fn change_state(&mut self) {  
74     match self.state {  
75         State::Red => {  
76             self.red = true;  
77             self.yellow = false;  
78             self.green = false;  
79         }  
80         State::RedYellow => {  
81             self.red = true;  
82             self.yellow = true;  
83             self.green = false;  
84         }  
85         State::Yellow => {  
86             self.red = false;  
87             self.yellow = true;  
88             self.green = false;  
89         }  
90         State::Green => {  
91             self.red = false;  
92             self.yellow = false;  
93             self.green = true;  
94         }  
95     }  
96 }
```

Figure 4: Screenshot of the change_state function.

This function in the figure 4 changes the state by assigning the correct values to the three booleans that regulate the light colors: it is now called in every state contained inside Figure 3 (not visible in Figure 3 since it was the task 3, hence not yet implemented).

I also added two other functions to make code actually start and work on Raspberry Pi: "run" and "main":

- "run" is contained inside the Stoplight Struct implementation and just loops the state_machine function so that the states will actually change and evolve in time.
- "main" (obviously), which wasn't modified up until now, contains code to boot the "init" function (which initializes the FSM, described in task 2), and to boot the "run" function, which I just explained.

```

232     pub fn run (&mut self){
233         loop{
234             self.state_machine();
235         }
236     }
237 }
238
239 ▶ Run | Debug
239 fn main() {
240     let mut stoplight: Stoplight = Stoplight::init(max_wait: Duration::new(secs: 10, nanos: 0));
241     stoplight.run();
242 }

```

Figure 5: Screenshot of the run and main functions.

1.5 Task 5

Task 5 wanted me to build an Enum to determine if the lights are in the US or the EU and pass this value to the struct when it is constructed to have it change the lights accordingly.

Since I was not so sure about how US and EU stoplights differ from each other, I asked around and was told that the first follow this sequence:

1. Green,
2. Yellow,
3. Red,
4. Yellow,
5. Green.

On the other side, the european ones follow this other sequence (which is also the Icelandic one):

1. Green,
2. Yellow,
3. Red,
4. Red and yellow together,
5. Green.

That said, I went on and implemented the lights in the ways shown above (hoping that the info I was given turns out correct, and if it doesn't, I'm sorry about that).

I started by creating the Struct, as shown below.

```
23  pub enum Region {  
24      EU,  
25      US  
26  }
```

Figure 6: Screenshot of the region enum.

Then, I doubled the two functions written in the steps 3 and 4 and modified one version to have them match either the european or the american model (shown in the pictures below).

```

43  fn state_machine(&mut self, region: Region) {
44      match region {
45          Region::EU => {
46              match self.state {
47                  State::Red => {
48                      self.state=State::RedYellow;
49                      self.change_state(region);
50                      self.yellow_pin.on();
51                      thread::park_timeout(dur: self.yellow_wait);
52                  }
53                  State::RedYellow => {
54                      self.state=State::Green;
55                      self.change_state(region);
56                      self.red_pin.off();
57                      self.yellow_pin.off();
58                      self.green_pin.on();
59                      thread::park_timeout(dur: self.max_wait);
60                  }
61                  State::Green => {
62                      self.state = State::Yellow;
63                      self.change_state(region);
64                      self.green_pin.off();
65                      self.yellow_pin.on();
66                      thread::park_timeout(dur: self.yellow_wait);
67                  }
68                  State::Yellow => {
69                      self.state=State::Red;
70                      self.change_state(region);
71                      self.yellow_pin.off();
72                      self.red_pin.on();
73                      thread::park_timeout(dur: self.max_wait);
74                  }
75              }
76          }
77          Region::US => {
78              match self.state {
79                  State::Red => {
80                      self.state=State::RedYellow;
81                      self.change_state(region);
82                      self.red_pin.off();
83                      self.yellow_pin.on();
84                      thread::park_timeout(dur: self.yellow_wait);
85                  }
86                  State::RedYellow => {
87                      self.state=State::Green;
88                      self.change_state(region);
89                      self.yellow_pin.off();
90                      self.green_pin.on();
91                      thread::park_timeout(dur: self.max_wait);
92                  }
93                  State::Green => {
94                      self.state = State::Yellow;
95                      self.change_state(region);
96                      self.green_pin.off();
97                      self.yellow_pin.on();
98                      thread::park_timeout(dur: self.yellow_wait);
99                  }
100                  State::Yellow => {
101                      self.state=State::Red;
102                      self.change_state(region);
103                      self.yellow_pin.off();
104                      self.red_pin.on();
105                      thread::park_timeout(dur: self.max_wait);
106                  }
107              }
108          }
109      }

```

Figure 7: Screenshot of the updated state_machine function.


```

112  ✓ fn change_state(&mut self, region: Region) {
113  ✓     match region {
114  ✓         Region::EU => {
115  ✓             match self.state {
116  ✓                 State::Red => {
117  ✓                     self.red = true;
118  ✓                     self.yellow = false;
119  ✓                     self.green = false;
120  ✓                 }
121  ✓                 State::RedYellow => {
122  ✓                     self.red = true;
123  ✓                     self.yellow = true;
124  ✓                     self.green = false;
125  ✓                 }
126  ✓                 State::Yellow => {
127  ✓                     self.red = false;
128  ✓                     self.yellow = true;
129  ✓                     self.green = false;
130  ✓                 }
131  ✓                 State::Green => {
132  ✓                     self.red = false;
133  ✓                     self.yellow = false;
134  ✓                     self.green = true;
135  ✓                 }
136  ✓             }
137  ✓         }
138  ✓         Region::US => {
139  ✓             match self.state {
140  ✓                 State::Red => {
141  ✓                     self.red = true;
142  ✓                     self.yellow = false;
143  ✓                     self.green = false;
144  ✓                 }
145  ✓                 State::RedYellow => {
146  ✓                     self.red = false;
147  ✓                     self.yellow = true;
148  ✓                     self.green = false;
149  ✓                 }
150  ✓                 State::Yellow => {
151  ✓                     self.red = false;
152  ✓                     self.yellow = true;
153  ✓                     self.green = false;
154  ✓                 }
155  ✓                 State::Green => {
156  ✓                     self.red = false;
157  ✓                     self.yellow = false;
158  ✓                     self.green = true;
159  ✓                 }
160  ✓             }
161  ✓         }
162  ✓     }
163  ✓ }

```

Figure 8: Screenshot of the updated change_state function.

Last but not least, I changed the "run" function to make it set the region, as shown in Figure 9.

```
165     pub fn run (&mut self) {  
166         loop {  
167             self.state_machine(Region::EU);  
168         }  
169     }  
170 }
```

Figure 9: Screenshot of the updated run function.

1.6 Task 6

This task asked to set the timer to be non-blocking and have a walk button skip it and jump to the next state.

I just edited the "run" function and made the button resume the blocked thread, thus skipping the timer.

```
165     pub fn run (&mut self) {  
166         let mut walk_button: Button = Button::new(pin: 19);  
167         let main_thread: Thread = thread::current();  
168         thread::spawn(move || loop {  
169             walk_button.wait_for_press(timeout: None);  
170             main_thread.unpark();  
171         });  
172         loop {  
173             self.state_machine(Region::EU);  
174         }  
175     }
```

Figure 10: Screenshot of the further updated run function.

1.7 Task 7

Task 7 just wanted me to separate the Struct implementation from everything else (which in my case, was just the "main" function, so I moved everything besides it in the "stoplight.rs" file, and edited the main a little bit to make it work again.

```

1  mod stoplight;
2  use stoplight::*;
3  use std::time::Duration;
4
5  ▶ Run | Debug
6  fn main() {
7      let mut stoplight: Stoplight = Stoplight::init(max_wait: Duration::new(secs: 10, nanos: 0));
8      stoplight.run();
9  }

```

Figure 11: Screenshot of the newly organized main file.

1.8 Task 8

- **Ownership** is Rust's alternative to the Garbage Collector or Manual Memory Management which are present in other programming languages: each value has a variable who owns it, this owner has to be unique and the value is no longer available when the owner gets out of scope (assigning an already existing value to another variable means moving its ownership).
- **Borrowing** is referencing to an object without taking its ownership, thus using it after the owner has "lent" it to us.
- **Referencing** is done by using the "&" character before the variable name and allows us to access that variable's value without shifting the ownership; if we want to change a value inside a mutable variable, we can use a mutable reference "&mut", but only once for a certain variable inside a certain scope.

1.9 Task 9

In the code given the programmer is trying to create a mutable reference ("some_string: String") to a variable that is not mutable ("let s = String::from("hello");").

This can be fixed by making the variable mutable, as below:

```

fn main() {
    let mut s = String::from("hello");
    change(&mut s);
}

fn change(some_string: &mut String) {

```

```
        some_string.push_str(", world");
    }
```

1.10 Task 10

In the code given the programmer created two different mutable references to the same variable in the same scope, which, as I said in task 8, is forbidden (this is done to prevent the "data race" phenomenon, which usually causes undefined behavior and, thus, should be avoided).

This can be fixed by creating two different scopes for the two mutable references, as below:

```
let mut s = String::from("hello");
{
    let r1 = &mut s;
}
let r2 = &mut s;
println!("{}", {}, r1, r2);
```

1.11 Video Proof

For a video of the assignment final version in action, I've uploaded a sample on Google Drive (I hope it works)!

<https://shorturl.at/juBM8>