Microprocessor Systems

Final project: **Gate control system.**

Karol Latos

Macrofaculty, group 2, 22 XI 2020

1 The problem

Gate control system The system should enable a gate control. Signals for *motor_open* and *motor_close* operation should be maintained in an active state if the gate motor is to operate. The signals from the limit switches (*open*, *closed*) and photocells (*photocell*) should be taken into account. The gate is controlled by a button (*control*) which should start the opening process of a closed gate, and vice versa. If pressed repeatedly, it should continuously start closing and opening the gate, alternately. The gate should only be opened for a defined time (*open_time*), after which it should close automatically.

2 Implementation

First, a control flow of the program is described by a pseudocode.

```
Program state: STATE (abstract)
One button operating the gate:  CONTROL
Three external sensors:         OPEN, CLOSED (gate mechanism sensors),
                                PHOTOCELL (obstacle detection)
One internal timer:             TIMER (open gate timer)

switch (STATE)
{
case CLOSED:
    if (CONTROL)
        STATE = OPENING

case OPENING:
    if (CONTROL)
        STATE = CLOSING
    if (OPEN)
        STATE = OPEN

case OPEN:
    if ((CONTROL || TIMER) && !PHOTOCELL)
        STATE = CLOSING

case CLOSING:
    if (CONTROL || PHOTOCELL)
        STATE = OPENING
    if (CLOSED)
        STATE = CLOSED
}
```

The implementation begins with defining all the signals and constants, together with the clock (**CLK**).

```
1  CLK         EQU P1.0          // Reference clock input.
2  PHOTOCELL   EQU P1.1          // Photocell sensor.
3  CLOSED      EQU P1.2          // Gate closed sensor.
4  OPEN        EQU P1.3          // Gate open sensor.
5  CONTROL     EQU P1.7          // Gate operating button.
6  MOTOR_CLOSE EQU P3.2          // Motor closing operation (output signal).
7  MOTOR_OPEN  EQU P3.3          // Motor opening operation (output signal).
8  OPEN_TIME   EQU 255           // Gate maximal open time before closing automatically.
9
```

We initialize the ports to low states. **P1** gathers inputs, i.e. signals from lines (1-5), while **P3** sends outputs — lines (6-7).

```
10 DSEG        AT  30
11 TIMER:    DS    1             // Timeout variable.
12
13 CSEG        AT  0
14 RESET:
15 // Initialization.
16     MOV     SP, #7FH
17     MOV     P1, #0
18     MOV     P3, #0
19     SETB    CLOSED
20
```

Now we're left with the task of implementing each state into the system. The default state is `GATE_CLOSED`.

```
21 GATE_CLOSED:
22     CLR     MOTOR_CLOSE
23 // Gate is closed until the button is pressed.
24     JNB     CONTROL, $
25 // Buttons are instantaneously reset to mimic the behaviour of a real button.
26     CLR     CONTROL
27 // Initiate opening.
28     SJMP    GATE_OPENING        // For clarity.
29
```

We stop the closing motion of the motor, as the gate is closed already, which is a stop condition for closing it. Nothing in the flow can change until we press the control button. This button is immediately popped back and the process of opening the gate begins.

```
30 GATE_OPENING:
31     CLR     MOTOR_CLOSE
32     SETB    MOTOR_OPEN
33 // Gate is opening until the button is pressed again (revert to closing) or the gate is opened fully.
34     CLR     CLOSED
35     JNB     CLK, $
36 // Check the control button.
37     JNB     CONTROL, CONT_OPENING
38     CLR     CONTROL
39     SJMP    GATE_CLOSING
40 CONT_OPENING:
41     JNB     OPEN, GATE_OPENING
42 // Set the TIMEOUT.
43     MOV     TIMER, OPEN_TIME
44     SJMP    GATE_OPEN           // For clarity.
45
```

Here again we set the physical state behind the abstract state first. Should the closing motor be running, it is shut down, to allow proper start of the opening motor. This situation may occur when the gate is closing and we press the control button. The order of these operations is specific, as not to have two opposite motors running in the same time, nor one motor trying to run in two directions.

The gate is closed no longer, so the **CLOSED** bit is cleared. In reality, this would be done automatically, as a low signal would be sent from appropriate sensor. When the clock is active, the program checks for a **CONTROL** signal to be provided, in case of reverting the action and closing the gate. Without pushing the button, the gate will continue opening, until the process is complete (limit senor **OPEN** active, here set manually). The timer for the gate is then set to a defined constant value, and the state of the system changes.

```
GATE_OPEN:
    CLR     MOTOR_OPEN
// Gate is open for set amount of time or until the button is pressed, provided there are no obstacles.
    JNB     CLK, $
    JB      CONTROL, CHECK_PHOTOCELL
    DJNZ    TIMER, GATE_OPEN
// When timeout reaches 0, is it reset to start over if there are obstacles.
    MOV     TIMER, OPEN_TIME
// Pressing the operating button, as well as waiting, initiates the photocell check before.
CHECK_PHOTOCELL:
    CLR     CONTROL
    JNB     PHOTOCELL, GATE_CLOSING
    SJMP    GATE_OPEN

```

When the gate is open, the motor operation is finished and the gate can be closed with the operating button or automatically, after the set time passes. However, in order for the gate to be functioning safely and properly, it cannot close if there are obstacles in the way. These obstacles are detected by the photocells and gathered in the **PHOTOCELL** signal. In case of this bit being active, the state reenters itself with the timer reset. Otherwise, the gate starts to close.

```
GATE_CLOSING:
    CLR     MOTOR_OPEN
    SETB    MOTOR_CLOSE
// Gate is closing until the button is pressed or an obstacle is detected, or the gate is closed fully.
    CLR     OPEN
    JNB     CLK, $
// Check the photocell.
    JB      PHOTOCELL, GATE_OPENING
// Check the control button.
    JNB     CONTROL, CONT_CLOSING
    CLR     CONTROL
    SJMP    GATE_OPENING
CONT_CLOSING:
    JNB     CLOSED, GATE_CLOSING
    SJMP    GATE_CLOSED

END

```

The `GATE_CLOSING` state is very similar to the `GATE_OPENING`, just the opposite motor is set and limit sensor **OPEN** is cleared, but we also take obstacles into consideration. On an instance of the **PHOTOCELL** signal being high, the gate reverts to opening, until is it fully open or the obstacle was removed and the button was pressed again. If there are no obstacles, the gate will be closed after the information comes from a limit sensor.

3 Testing

The project was written with a simple `.ini` file, comprising only a simulation of a clock on **P1.0** (signal **CLK**).

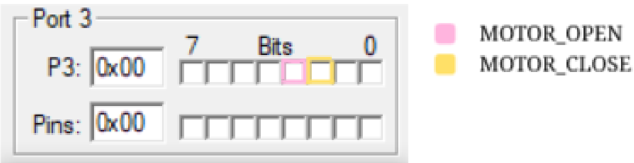
```
signal void clock() {
    while(1) {
        twatch(32);
        P1 = P1 ^ 0x01;
    }
}

clock()
```

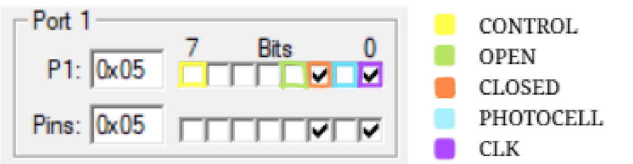
It is recommended to lower the value of **OPEN_TIME** (for example to 2) to follow the program flow on an instruction level. Or rather, the code can be set to run continuously, as we observe the responses on **P1** and **P3**. For this mode, the `GATE_OPENING` state may seem to result in `GATE_CLOSING` state, but the gate is open in between. The maximal time of being open passes really fast during a continuous simulation, so in order to observe the `GATE_OPEN` state, one can activate the **PHOTOCELL** signal, which prevents the gate from closing, as long as it is active.

Used pins on both ports have been coloured to help with the testing.

Parallel Port 3

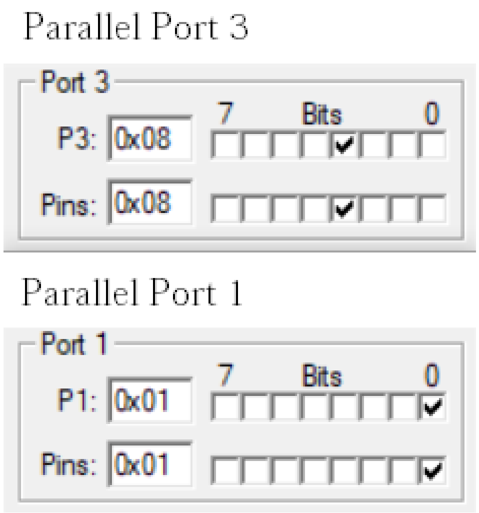


Parallel Port 1

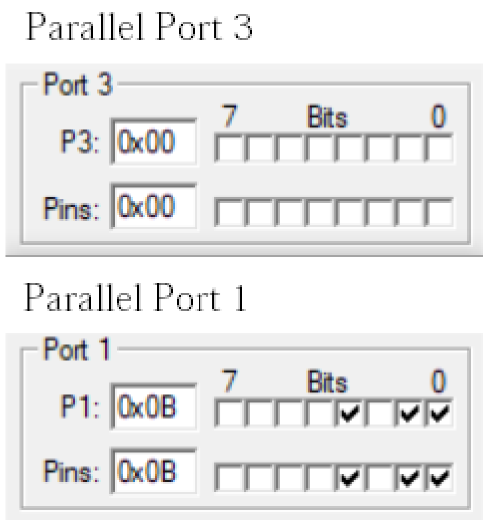


Selected combination of signals is included below as a representation of program states and subconditions.

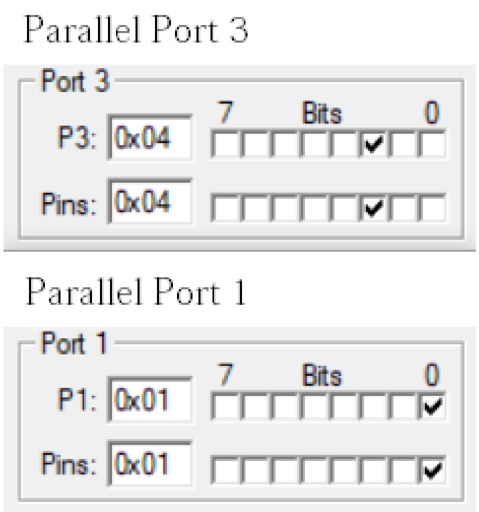
The complete code can be accessed at `https://github.com/rol-x/gate_control_system`.



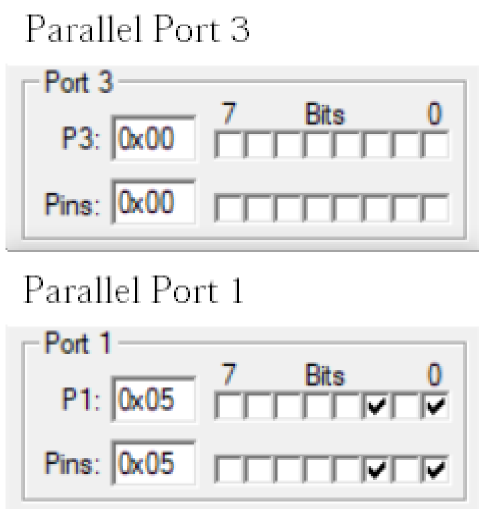
(a) Gate is opening and waiting either for **OPEN** or **CONTROL** signal



(b) Gate is open, with an obstacle in the way



(c) Gate is closing and waiting for either **CLOSED**, **CONTROL** or **PHOTOCELL** signal



(d) Gate is closed