Complementary notes on exercise sessions

Thomas Braipson

December 21, 2025

Foreword

This document contains complementary notes on exercises proposed during tutorials. This document is supposed to help students prepare for the exam, by allowing to better understand what was said during tutorials. This will by no means replace a thorough understanding of what was explained during theoretical lectures. Some solutions presented here may not be unique.

1 Exercise session 3

A circuit is composed of two LEDs and a microcontroller equipped with a timer with tunable frequency. The frequency of this timer can be set to

- 24 kHz,
- 8 kHz,
- 6 kHz, or
- 1 kHz.

LEDs should be respectively toggled at 8 kHz and 3 kHz. How can we program a solution for this, without any busywaiting, if we assume that the instruction clock frequency of the MCU is 1 MHz and that the timer sends an interrupt request at each tick?

```
int cnt1, cnt2 = 0;

void interrupt my_interrupt(void)
{
  cnt1++;
  cnt2++;
  if (cnt1==3)// handle 8kHz led
  {
    !! toggle led1
    cnt1 = 0;
  }
  if (cnt2==8)// handle 3kHz led
  {
    !! toggle led2
    cnt2 = 0;
  }
  !! clear interrupt flags
}

int main()
{
  !! set timer frequency to 24 kHz
```

```
  !! enable interrupts on timer overflow
  while(1);
}
```

The idea behind this solution is to program an interrupt routine that increments two counters at each overflow of the timer. `cnt1` is responsible for deviding the frequency by 3 and `cnt2` is responsible for deviding the frequency by 8.

For the following exercises, we will assume that an 8-bit MCU performs computations. The score of each team is a number between 0 and 99. Therefore, this number can be internally stored as one byte. Yet, the display takes as input the signal to be applied to each segment. How can we convert in constant time the 8-bit value into the 2 times 7 bits used to display it ?

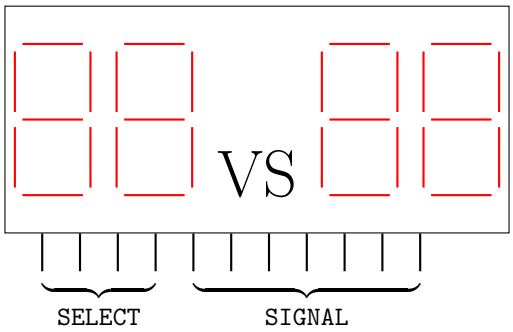

Figure 1: Schematic of the display

Solution for displaying digits:

```
const char table_l[100] = { 0b1111 1100, 0b0110 0000, 0b1101 1010, ...};
const char table_h[100] = { 0b1111 1100, 0b1111 1100, 0b1111 1100, ...};
const char table_id[4]  = { 1, 2, 4, 8};
char SELECT, SIGNAL;
void display(char segment_id, char score)
{
  SELECT = table_id[segment_id];
  if(segment_id % 2) // if segment id is odd, display most significant digit
    SIGNAL = table_h[score];
  else               // if segment id is even, display least significant digit
    SIGNAL = table_l[score];
}
```

The technique used here is called lookup tables. A lookup table consists in an array `a` representing pairs of values $(i, f(i))$ where f is any function and i ranges from 0 to $n-1$, where n is the length of a. To evaluate f in a given index k, one simply reads `a[k]`.

Here, we have three lookup tables, one for computing which segments to turn on the display associated to the least significant digit for each score value. A second one does the same thing for the most significant digit. The last table is used to convert an integer into its unary noation ($k \mapsto 2^k$), this allows to select which segment to select. A team may score a goal at any time (we assume that a delay under 500 ms between two goals cannot happen). Per team, one MCU pin is connected to an external signal responsible for telling that one extra goal has just been scored by the corresponding team. This signal will trigger an interrupt request. The delay between information reception and display has to be as short as possible. It is not permitted to display a number that is neither the current nor the previous score for each team.

The following code shows an implementation. What is wrong with it (two things)? How can we circumvent this?

```
char score_t_1, score_t_2 = 0;
void interrupt update_scores(void)
```

```
{
  if(team_1_has_scored)
  {
    team_1_has_scored = 0;
    score_t_1 ++;
  }
  if(team_2_has_scored)
  {
    team_2_has_scored = 0;
    score_t_2 ++;
  }
}
int main()
{
  while(1)
  {
    display(1, score_1);//display first digit of score 1
    display(2, score_1);//display second digit of score 1
    display(3, score_2);//display first digit of score 2
    display(4, score_2);//display second digit of score 2
  }
}
```

The two issues are the following:

- The operation of displaying the two digits of a score is not atomic (even if the function display is atomic itself), since there are two consecutive calls to the function display in a non-atomic block. Therefore, it could happen that a score is updated just between the display of its first and second digits.
- The two global variables used to store scores are not declared `volatile`. Since there is no explicit call of the interrupt routine and it is the only function able to modify these variables, the compiler might try to optimize the code by converting `score_1` and `score_2` into constants equal to `0`. By declaring these variables volatile, ask the compiler not to make any optimisation about these variables.

Here is another implementation. What are the issues now ?

```
int main()
{
  volatile char score_t_1, score_t_2 = 0;
  while(1)
  {
    if(team_1_has_scored)
    {
      team_1_has_scored = 0;
      score_t_1 ++;
    }
    if(team_2_has_scored)
    {
      team_2_has_scored = 0;
      score_t_2 ++;
    }
    display(1, score_1);
    display(2, score_1);
    display(3, score_2);
    display(4, score_2);
  }
}
```

In this implementation, there no longer is any atomicity issue since flags are checked in the beginning of the loop and then displays are updated. Yet, the four displays are not on during the same amount of time. Indeed, display

4 will remain on for a longer amount of time than displays 1, 2, and 3, since it is turned on at the end of the loop and turned off only after testing flags and updating counters.

2 Exercise session 4

We want to program a system responsible for managing sensors on a boat. This system is equipped with six sensors that asynchronously send data to the microcontroller. To tell that a new measurment is available, the corresponding sensor sends an interrupt request and the microcontroller is responsible for fetching the corresponding data. This is assumed to take negligible time. Sensors may send data at any time, with variable frequency. As soon as information coming from a sensor has been fetched by the MCU, it has to be processed (this takes around 100 μs). Then a summary has to be sent every 10 ms. This amounts to copying a few bytes to registers and setting flags, it can be assumed instantaneous. If the measurment obtained by some sensor could be the sign of a life-threatening situation, an alarm should immediately be sounded. This is done by setting a dedicated pin to a high voltage. Moreover, the six sensors monitor different physical quantities and some are better estimators of a hazard than others. We will assume that sensor 1 is the most critical one, just before sensor 2, down to sensor 6.

What software architecture is best suited for this problem?

Give a pseudo-code of this architecture that is precise enough to illustrate the different tasks, their communication objects as well as interrupt routines (if some are needed).

We chose a waiting-queue architecture since

- No task has a period smaller than the execution time of another task. Thus no need for RTOS.
- There is a priority between tasks, they thus cannot be executed in a loop. Thus waiting-queue.

```
/*
global variables to communicate between main code and interrupts
*/

volatile unsigned data[6] // array for saving data from sensors (one cell per sensor)
volatile unsigned summary[6] // array for saving summary from each sensor (one cell per sensor)
volatile bool my_flags[6] // array for remembering which sensors gave measurement results

 /*
Interrupt triggered by sensors. If request came from sensor i,
flags[i] is raised and data is written to reg_i.
*/

void interrupt_gpio()
{
  for(int i = 1; i < 7; i++)
  {
    if(flags[i])
    {
      data[i-1] = reg_i
      my_flags[i] = 1;
      flags[i] = 0;
    }
  }
}
/*
Interrupt triggered by timer
*/
void interrupt_timer()
{
  for (int i = 0; i < 6; i++)
    !! copy summary[i] in appropriate location
```

```
  !! raise flags for sending information
  !! lower interrupt flag of timer
}

/*
each f_i (for i \in {1,...,6}) is similar to f_1
*/
void f_1()
{
  my_flags[0] = 0;
  bool is_enabled = gpio_interrupt_is_enabled(); // check if interrupts from sensors were initially enab
  gpio_interrupt_disable(); // disable interrupts from sensors
  unsigned x = data[0];
  if (is_enabled)
    gpio_interrupt_enable();// restore interrupt enable state
  !! process x
  !! if danger, send high voltage on alarm pin
  is_enabled = timer_interrupt_is_enabled();//check if interrupts from timer were initially enabled
  timer_interrupt_disable(); // disable interrupts from timer
  summary[0] = !! result of processing x
  if(is_enabled)
    timer_interrupt_enable(); //restore interrupt enable state
}

int main()
{
  !! configure timer to send interrupt request every 10 ms and to call interrupt_timer
  !! configure gpio pins to send interrupts and call interrupt_gpio
  !! enable interrupts
  while(1)
  {
    if(my_flags[0])
    {
      f_1();
      continue;
    }
    if((my_flags[1])
    {
      f_2();
      continue;
    }
    if((my_flags[2])
    {
      f_3();
      continue;
    }if((my_flags[3])
    {
      f_4();
      continue;
    }if((my_flags[4])
    {
      f_5();
      continue;
    }if((my_flags[5])
    {
      f_6();
```

```
        continue;
      }
    }
  }
```

The idea behind this solution is to continuously check if the most important sensor has sent a new measurement and to process it first. Note that without continue statements inside each if, this architecture would have been round Robin with interrupts and each sensor would have been tested one after the other.

3 Exercise session 6

First, we assign priorities according to the RMS strategy: the smaller the period, the higher the priority. This yields $P_3 > P_2 > P_1$.

Then, we make a timeline starting from $t = 0$ and a request for each task, ending in $t = \frac{9}{2}$ and show every request in between.

Then, we assign the processor according to the RMS strategy. Therefore, we give the processor to task 3 for $\frac{1}{2}$ time unit immediately after each request.

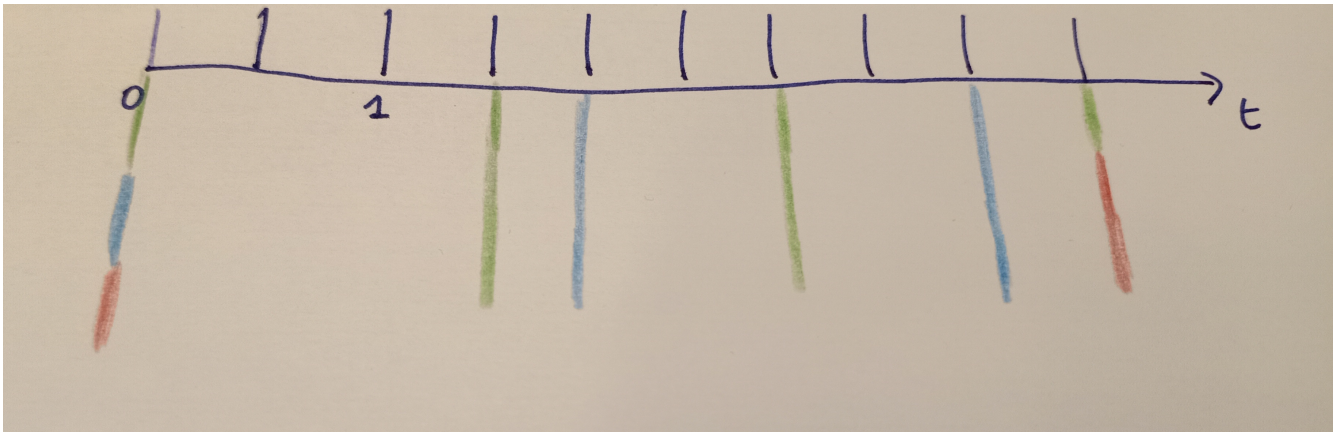

Figure 2: Drawing what we know

Now, we notice that there remain 3 time units for tasks 2 and 1. In that amount of time, one full execution of task 1 has to be performed, as well as two full and one potentially partial executions of task 2.

We distinguish two cases:

- If the there are three full executions of task 2 between $t = 0$ and $t = \frac{9}{2}$: This implies $\alpha \leq \frac{1}{2}$. Let us test the limit case, if $\alpha = \frac{1}{2}$, the time during which the processor is used is equal to $3 \times \frac{1}{2} + 1 = 2.5 < 3$. This case is schedulable but does not fully use the processor since it remains unused during $\frac{1}{2}$ time units.

- Otherwise, $\alpha > \frac{1}{2}$. In this case, task 1 must be fully executed before $t = 4$ (indeed, after that instant and until the next request for this task, the processor will be used by task 2, due to RMS strategy. In that interval, there are $\frac{5}{2}$ slots available and there are two full executions of task 2 and 1 full execution of task 1. Thus, $\frac{5}{2} = 2 \times \alpha + 1$, and we conclude that $\alpha = \frac{3}{4}$.

We now complete our graphical simulation to check that this is indeed correct.

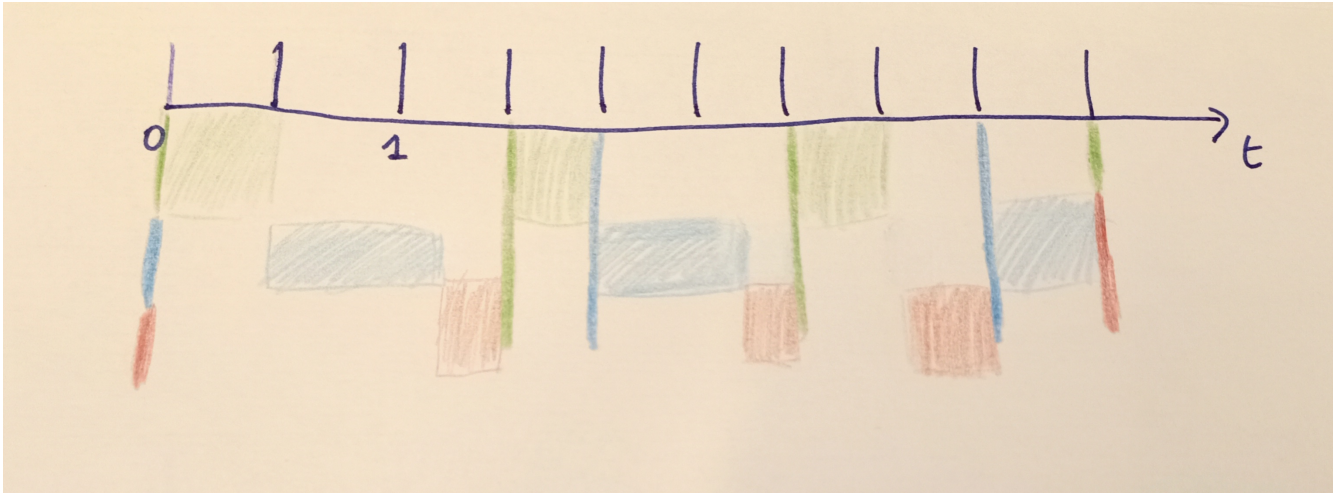

Figure 3: Simulation result

4 Exercise session 7

Consider the following set of periodic tasks $\tau_i = (C_i, T_i)$:

$\{\tau_1 = (1, 7), \tau_2 = (1, 5), \tau_3 = (1, \alpha)\}$, where α is a parameter.

1. Compute the smallest value of α that makes this set of tasks schedulable.
2. Verify your answer with a graphical simulation.

In this case, the unknown is a period and not an execution time. We cannot directly assign priorities since we do not know the value of α. We will test several cases:

1. $\alpha > 7$
2. $5 < \alpha < 7$
3. $\alpha < 5$

1. In this case, $\alpha = 35$ makes the set schedulable since 35 is the least common multiple of 5 and 7. We now look for a value between 35 and 7. We now give the processor to the tasks according to RMS strategy. Between $t = 0$ and $t = 7$, there are:
   - Two requests for task 2 and they are fully executed.
   - One request for task 1 and it is fully executed.
   - One request for task 3 and it is fully executed, for any $\alpha > 7$.

   This leaves 3 time slots unused during this interval.
2. We now know that we can study the system over the interval $0 \leq t \leq 7$. Over this interval, there are
   - One time unit consumed by task 1
   - Between one and two time units consumed by task 2

   This leaves at least 4 time untis for task 3. Since the execution time for task 3 is one time unit, this allows at least four requests for this task during 7 time untis. Thus, $\alpha = \frac{7}{4}$ should work. Let us make a graphical simulation: This is indeed schedulable. Still, we notice that if the fourth request for task 3 happened at $t = 5$, the situation would not change much:
   - In the interval $0 \leq t \leq 5$, the processor would have been given to task 3 for the same amount of time and therefore, this interval would have remained ok.

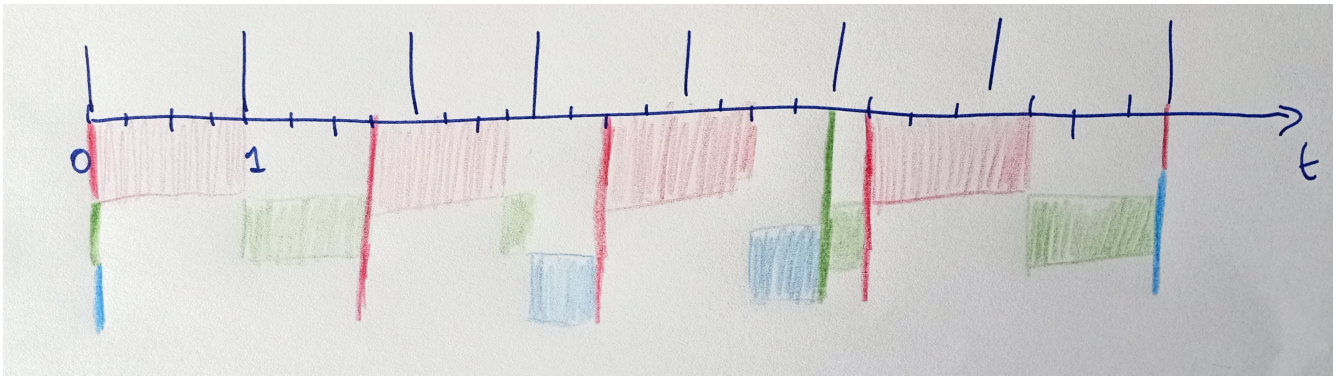

Figure 4: graphical simulation with $\alpha = \frac{7}{4}$

- After $t = 5$, this would have delayed some of the execution of task 2, which is not a problem.

We also notice that if the fourth request for task 4 happened before $t = 5$, this would have no longer been schedulable. Indeed, there would be some execution of task 1 delayed after $t = 5$ but between $t = 5$ and $t = 7$, the processor would have continuously been busy with tasks of higher priority.

We can then compute the value of α such that the fourth request comes at $t = 5$: there are three requests in the interval $0 \leq t < 5$ therefore $3 \times \alpha = 5$ and $\alpha = \frac{5}{3}$. We make another graphical simulation that shows that indeed, this is the smallest possible value for α.

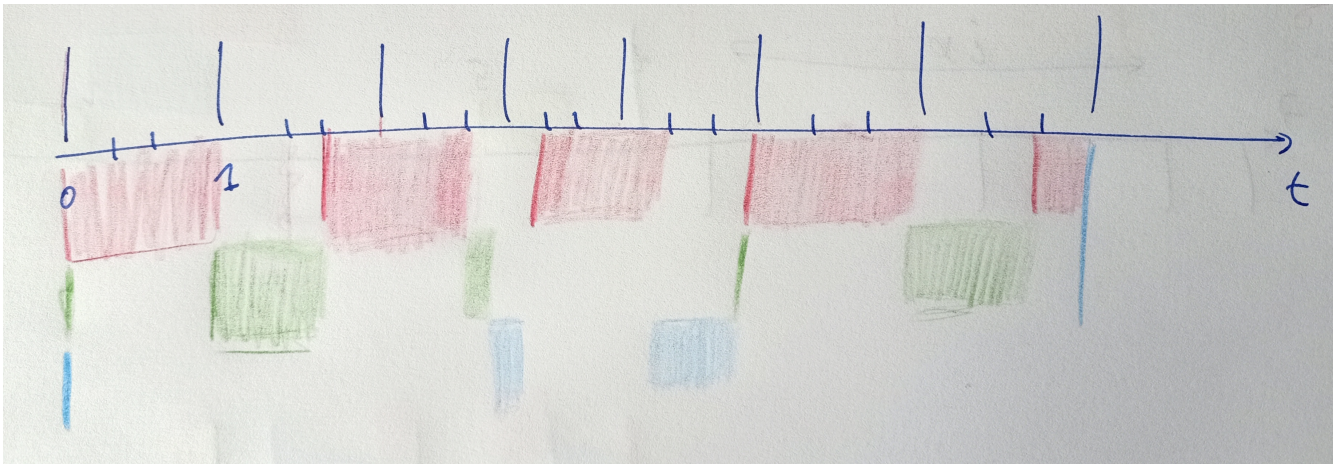

Figure 5: graphical simulation with $\alpha = \frac{5}{3}$

5 Exercise session 10

Variables:

- x the pressure in MPa
- y_1 the contribution to the pressure rate due to the pump
- y_2 the contribution to the pressure rate due to the valve
- x_1 and x_2 timers

The idea of this solution is to have a process modelling the pump and another process modelling the valve, plus a third process modelling the additive behaviour of the joint system.

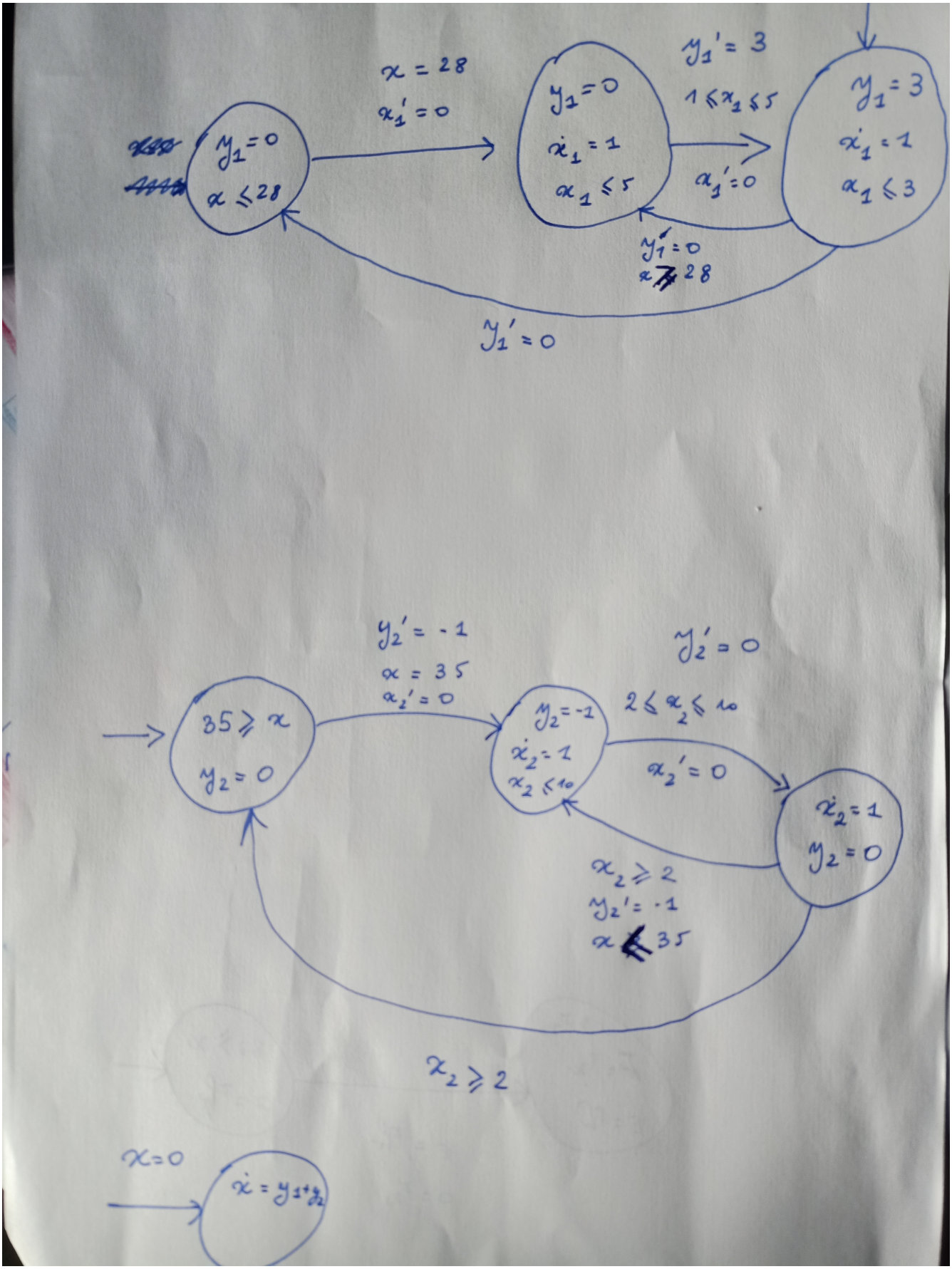

Figure 6: Hybrid system modelling the situation