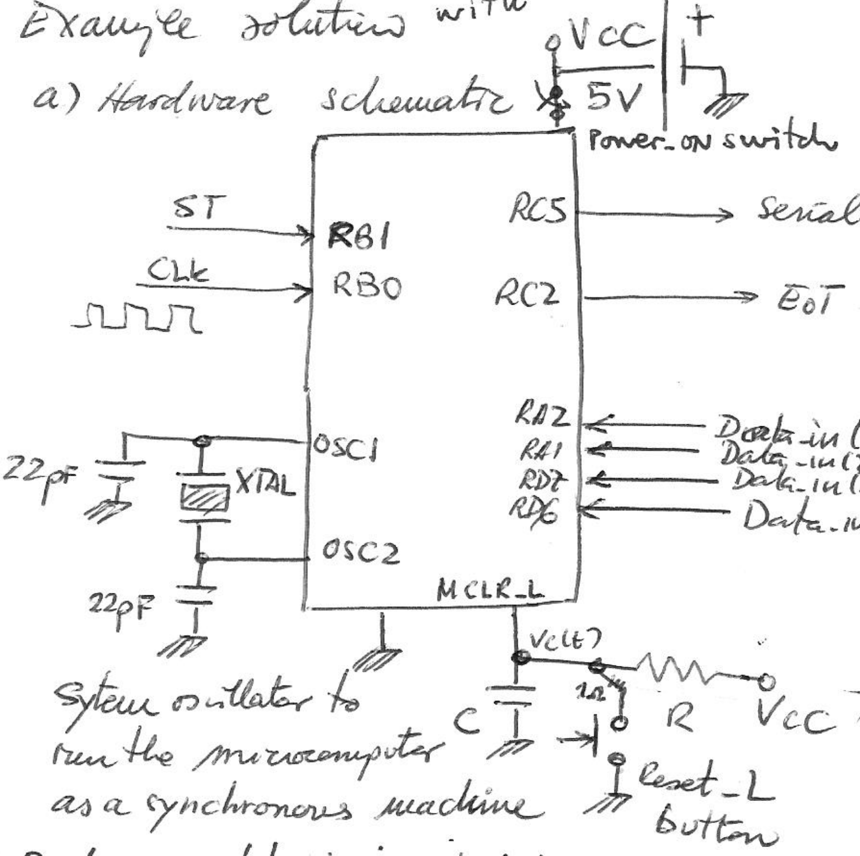
Problem 3

Example solution with comments and discussion

a) Hardware schematic



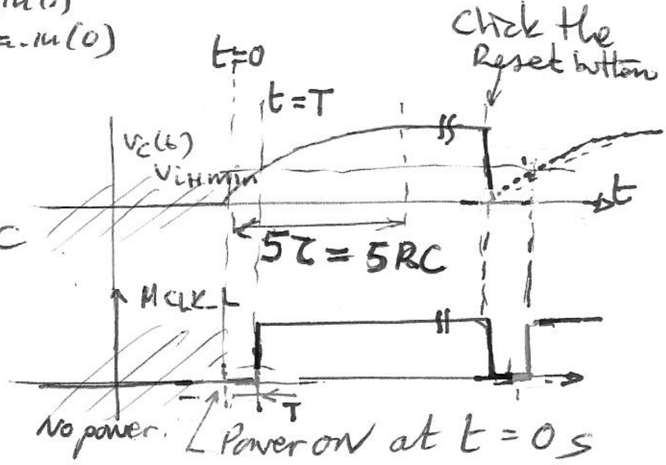
Sytem osillator to run the microcomputer as a synchronous machine

For example:

$T_{Reset} = 50\,ms$

$T_{Reset} \approx RC$; $C = 100nF$

$R = 500k\Omega$



Reset_L signal to initialise the computer

→ Each assembly is executed in a time $\frac{Fosc}{4}$, exceptuating jump instructions which takes $\left(\frac{Fosc}{2}\right)$

- * Data_in pins as inputs
- * ST, CLK pins as inputs
- * Serial_out an EoT as output

→ init_system()

	7	6	5	4	3	2	1	0
TRISA						1	1	
TRISB							1	1
TRISC			Ø			Ø		
TRISD	1	1						

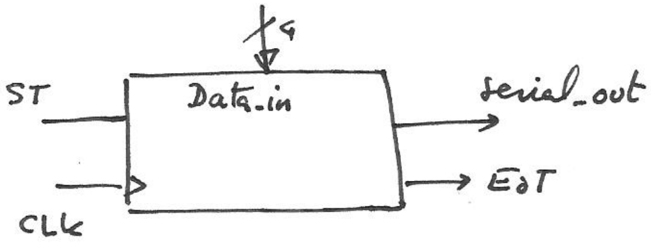
other bit can be configured as inputs or outputs, but it is preferable as outputs 'Ø'

$\vec{\text{C code}}$

```
TRISA = 0bØØØØØ11Ø;
TRISB = 0bØØØØØØ11;
TRISC = 0x00;
TRISD = 0b11ØØØØØØ;
```

→ Other initialisation instructions include GIE = 1 to allow interrupts from the RB1(INT1) Start transmission and the selection of the active edge ↓ or (↑). INTEDG1 = 1;

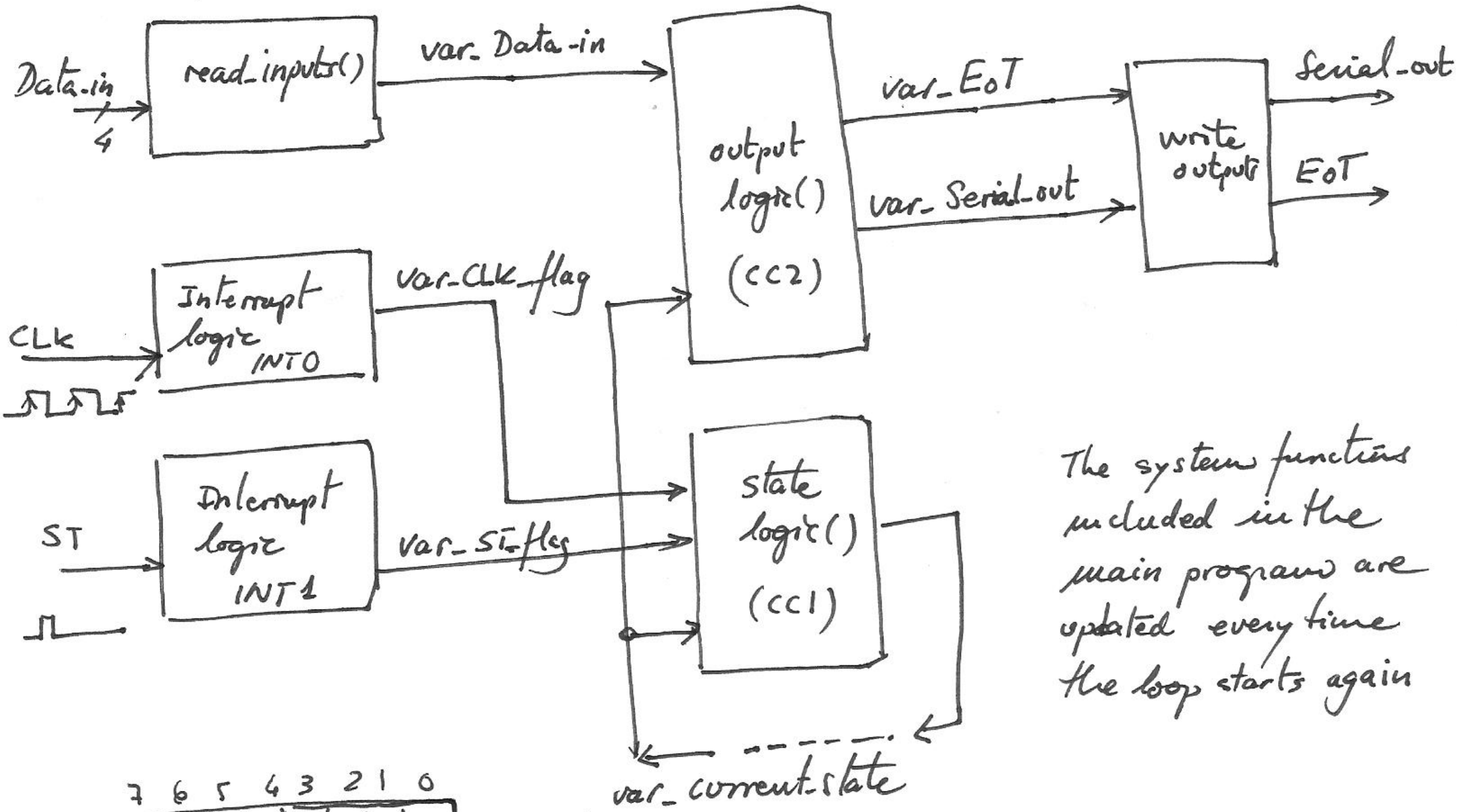
b) The system to be designed is essentially a 4-bit right-shift register



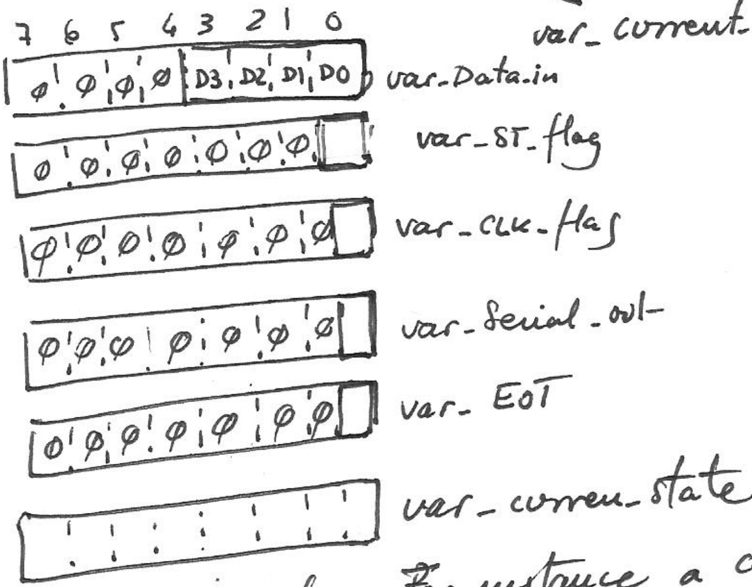
Load the Data-in in a RAM memory variable and transmit one bit at a time (CLK period) plus the start-bit (Ø)

ST is the shift order to start the transmission
Parallel in → serial-out

The typical FSM that can solve the problem looks like this:



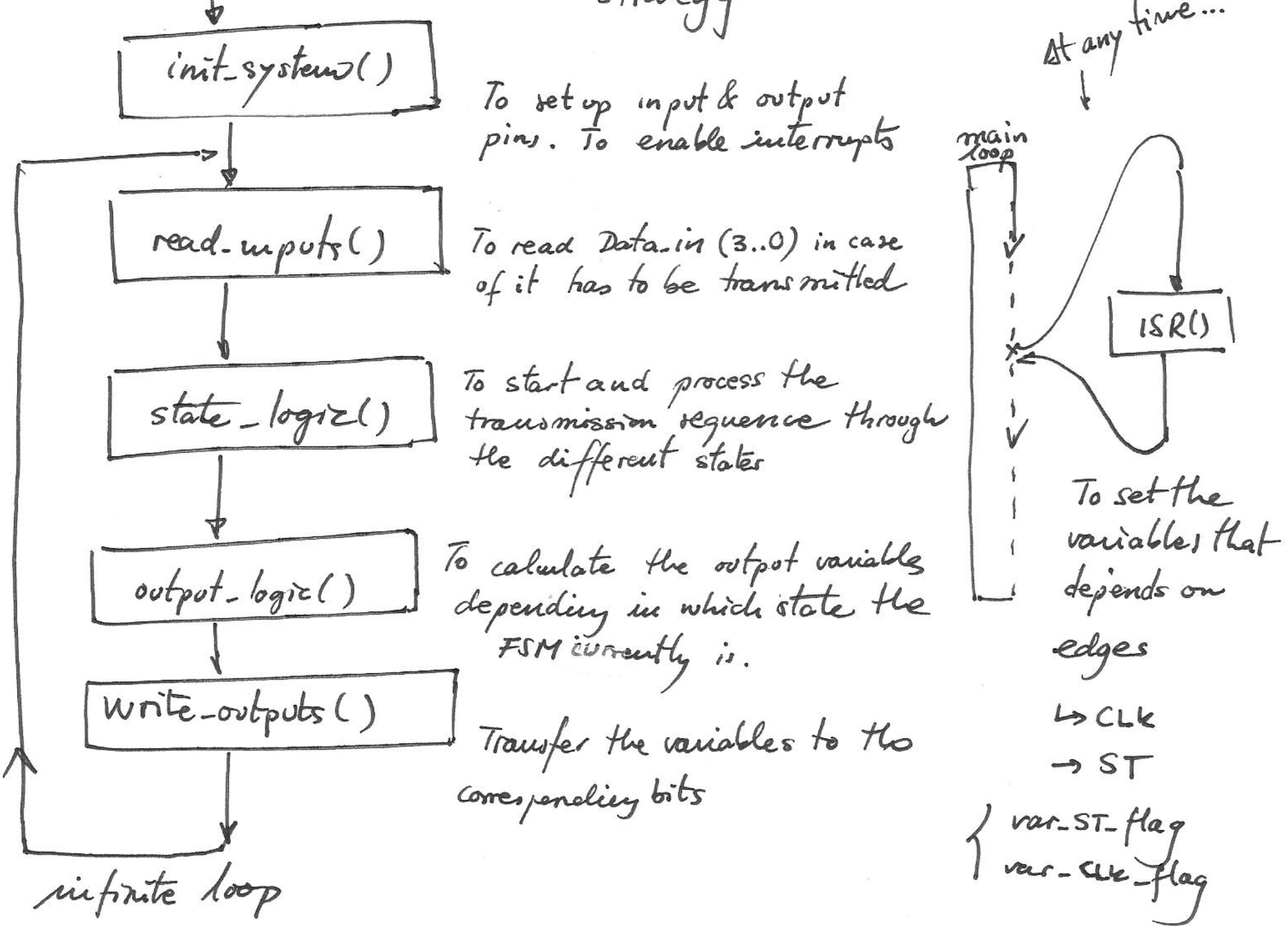
The system functions included in the main program are updated every time the loop starts again



6 bytes of RAM data memory to save the global variables (for easy watching & debugging)

Ascii code. For instance a capital letter to encode each state 'A', 'B', . . .

General software organisation from the F.S.M. strategy



To set up input & output pins. To enable interrupts

To read Data_in (3..0) in case of it has to be transmitted

To start and process the transmission sequence through the different states

To calulate the outpot variables depending in which state the FSM currently is.

Transfer the variables to the corresponding bits

To set the variables that depends on edges

↳ CLK

→ ST

{ var_ST_flag
var_CLK_flag

- The INT1IE = 1 all the time to allow the detection of the ST edge to start transmissions

- Once the transmission has started, the INT0 interrupt enable must be enabled (INT0IE = 1) to allow detections of the CLK edges ↑

- It can be done in different ways, bot polling (reading) the Data_in once the loop starts again is not a bad idea, but it'll be better to read this data once the $\underset{CLK}{\uparrow}$ is detected because in this way the Data_in is trully "sampled" at CLK ↑ as in Chapter II sequential synchronous circuits.

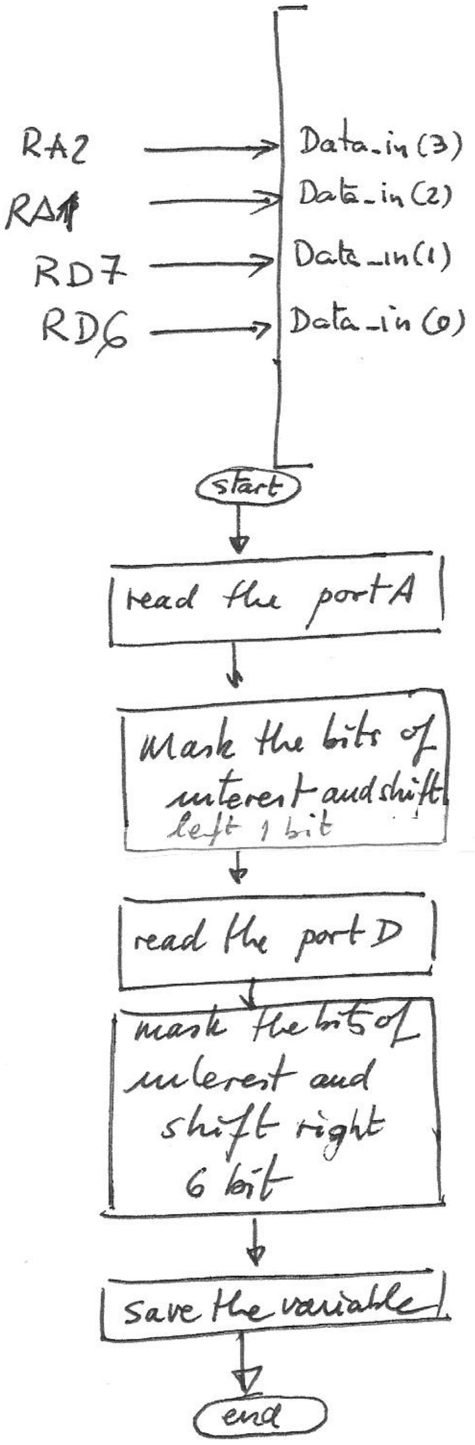
So: read_inputs() when var_CLK_flag is set, skip reading otherwise.

c) read_inputs()

For reading the port bits and masking the bits of interest while rejecting the ones which are not in use:

The objective is

∅	∅	∅	∅	D3	D2	D1	D0	var_Data_in



7	6	5	4	3	2	1	0	
x	x	x	x	x	D3	D2	x	PORTA
∅	∅	∅	∅	∅	1	1	∅	← mask

D1	D0	x	x	x	x	x	x	PORT D
1	1	∅	∅	∅	∅	∅	∅	← mask

AND

```
var_buf1 = PORTA & 0b∅∅∅∅∅11∅;
var_buf1 = var_buf << 1 ;
```

7	6	5	4	3	2	1	0
∅	∅	∅	∅	D3	D2	∅	∅

```
var_buf2 = PORTD & 0b11∅∅∅∅∅∅;
var_buf2 = var_buf2 >> 6;
```

7	6	5	4	3	2	1	0
∅	∅	∅	∅	∅	∅	D1	D0

OR

```
var_Data_in = var_buf1 | var_buf2;
```

→Final result

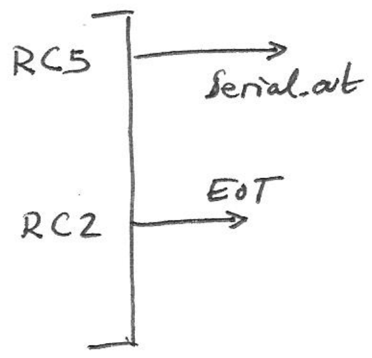
7	6	5	4	3	2	1	0
∅	∅	∅	∅	D3	D2	D1	D0

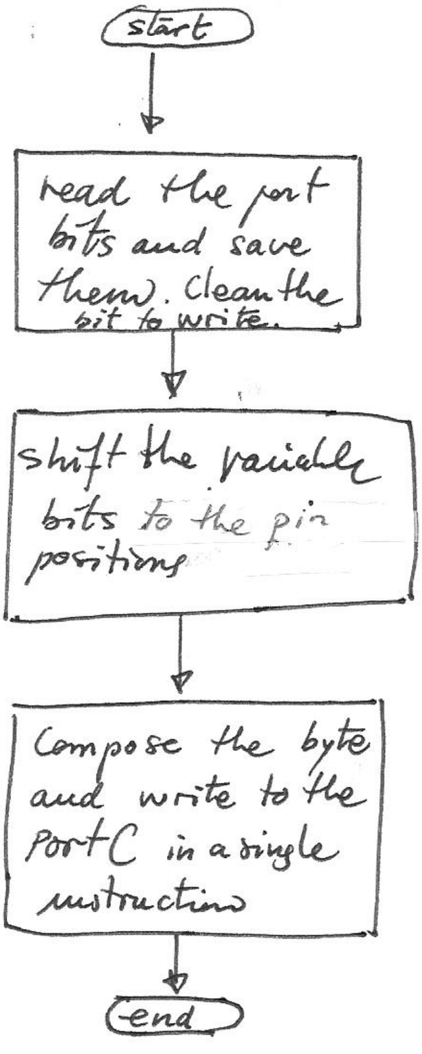
```
if (var_clk_flag)
    read_inputs();
```

↑ sampling when necessary

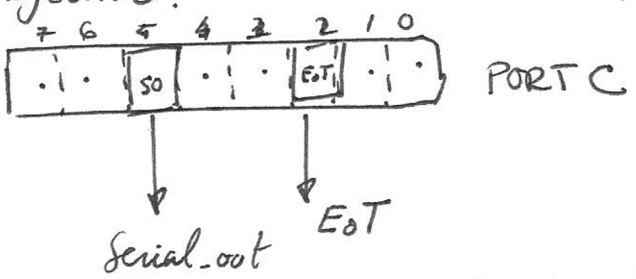
(Which can be debugged using the watch window when developing & testing)

d) Write_outputs()



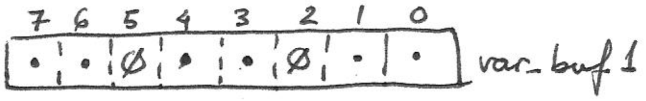

start
read the port bits and save them. Clean the bit to write.
shift the variable bits to the pin positions
Compose the byte and write to the PortC in a single instruction
end

Objective:



Write in a single instruction the bits of interest while preserving the bits not used. In this way the system can be enhanced without rewriting the code

```
var_buf1 = PORTC & 0b11Ø11011;
```

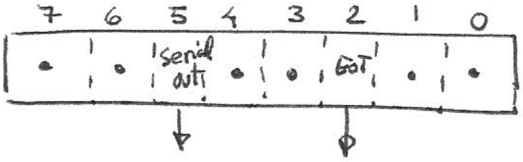


```
var_buf2 = var_EoT << 2;

var_buf3 = var_Serial_out << 5;

var_buf3 = var_buf3 | var_buf2;

PORTC = var_buf3;
```

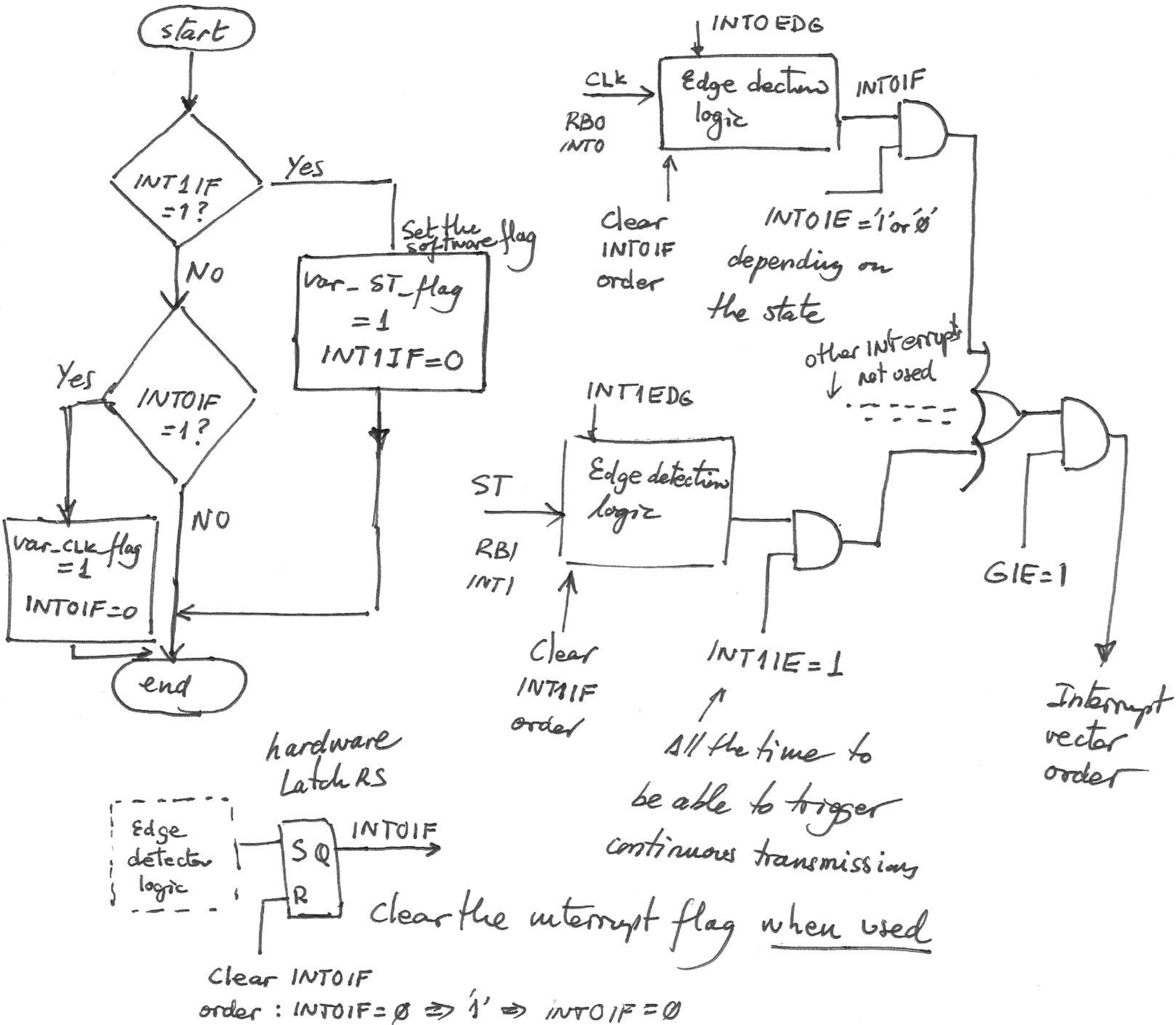


* The code can be more efficient using less memory positions and instructions. But here, what is important is to realise the many operations to perform in a sequence

```
PORTC = (var_EoT << 2) | (var_Serial_out << 5) | (PORTC & 0b11Ø11011);
```

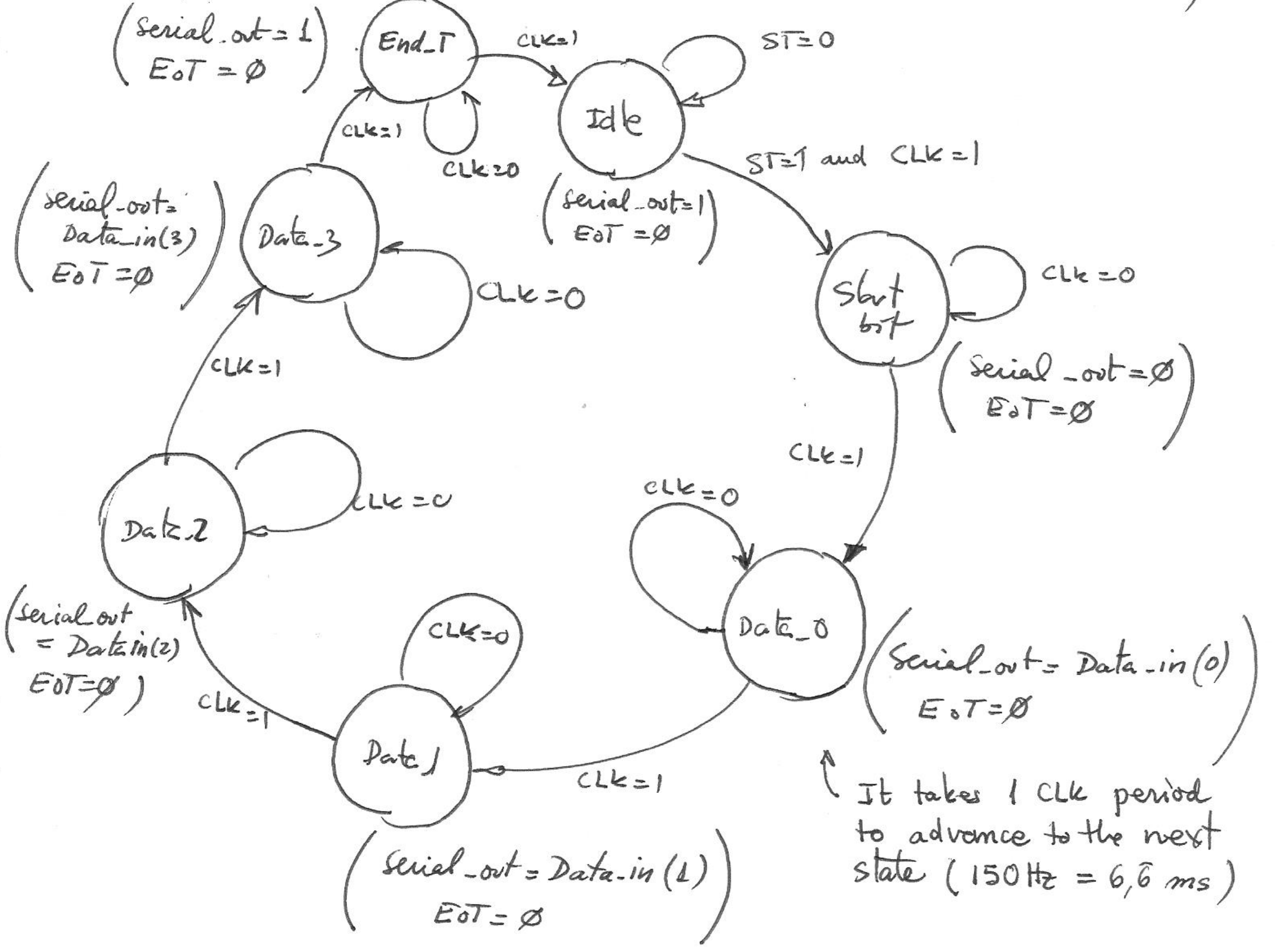
e) ISR() is the function for organising the tasks associated with the interrupts. Any time that an interrupt occurs the main program exeution stops, all the registers, flags and environment is saved, and the program counter jumps to the interrupt vector were is written the ISR() assembly code.

In this application we have 2 interrupts of the same kind (external) to detect edges in ST an CLK signals (INT1 and INT0) Later the TMR0 can be use as an interrupt source to replace INT0 and save an external circuit by means of an internal peripheral.



f) This the application state diagram to run the FSM

Now, if all the prekious concepts are comprehended, let's draw the Key idea to run the transmitter in our standard way.


End_T
Idle
Start bit
Data_0
Data_1
Data_2
Data_3
(Serial_out = 1 EoT = Ø)
(serial_out = 1 EoT = Ø)
(Serial_out = Ø EoT = Ø)
(Serial_out = Data_in(0) EoT = Ø)
(serial_out = Data_in(1) EoT = Ø)
(serial_out = Data_in(2) EOT = Ø)
(serial_out = Data_in(3) EoT = Ø)
CLK=1
ST=0
ST=1 and CLK=1
CLK=0
CLK=1
CLK=0
CLK=1
CLK=0
CLK=1
CLK=0
CLK=1
CLK=0
CLK=1
CLK=0

It takes 1 CLK period to advance to the next state (150Hz = 6,6̄ ms)

NOTE { CLK is var_CLK_flag
ST is var_ST_flag
Data_in is var_Data_in
Serial_out is var_Serial_out
EoT is var_EoT

} RAM memory variables, which are set reading or using interrupts

→ CC2 = output_logic will set this variables

⇒ In this way, the main program runs continuously (measure the time required to run the main program using breakpoints) but each bit is transmitted at 150b/s because it's controlled by the CLK flag

9) From the state diagram we can solve the "combinational circuit" CC1 (state logic) infering the truth table that generates all the 14 arrows (state transitions). See section b)

current_state	ST	CLK	current_state$^+$
Idle	Ø	X	Idle
Idle	1	Ø	Idle
Idle	1	1	start_bit
start_bit	X	Ø	start_bit
start_bit	X	1	Data_0
⋮	⋮	⋮	⋮
End_T	X	Ø	End_T
End_T	X	1	Idle

← the next loop turn

all the transitions in Idle state

ST is var_ST_flag
CLK is var_CLK_flag

once used they have to be reset

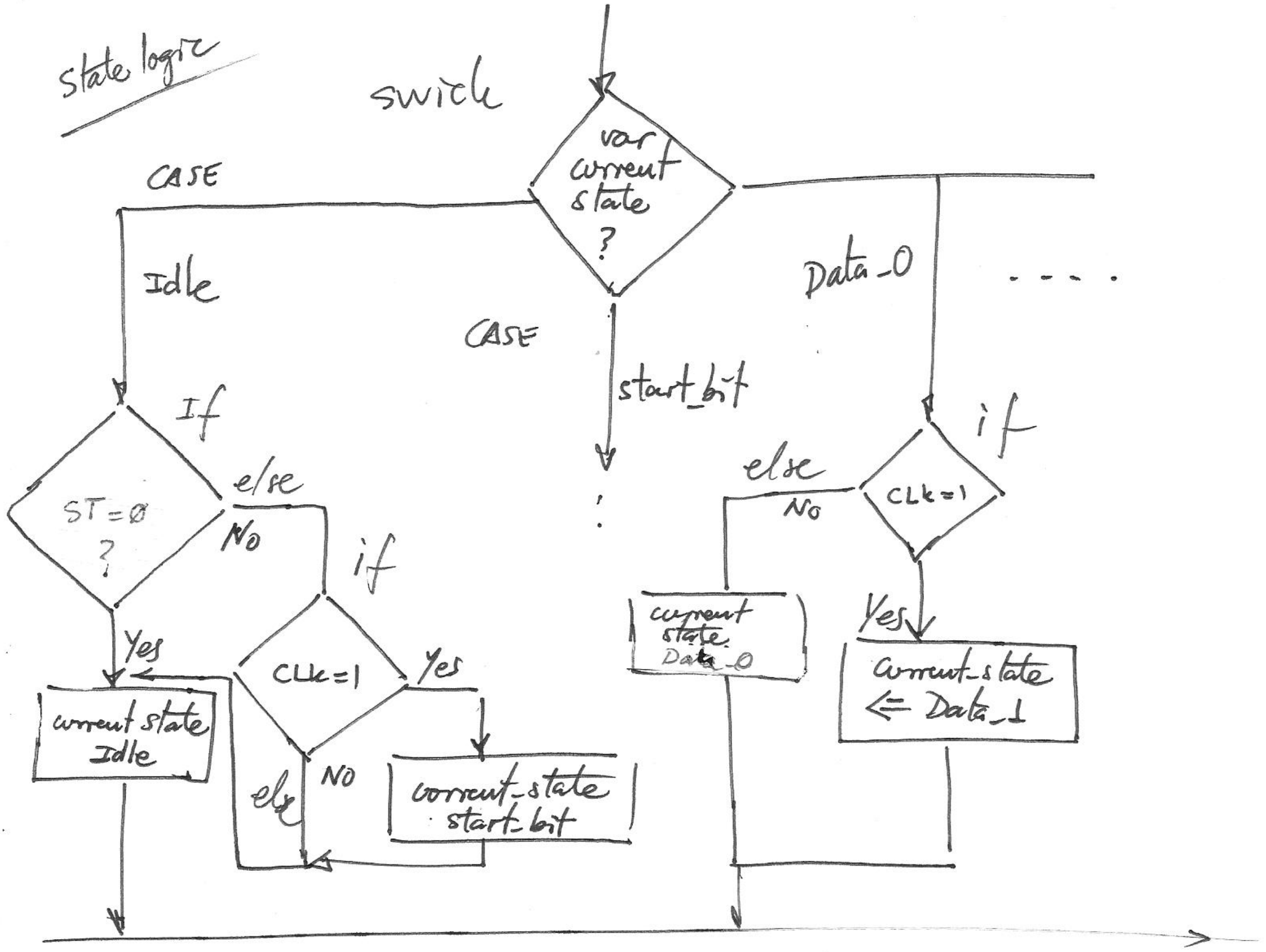
In the same way:

current_state	EoT	Serial out
Idle	Ø	1
Start_bit	Ø	Ø
Data_0	Ø	Data_in(0)
Data_1	Ø	Data_in(1)
Data_2	Ø	Data_in(2)
Data_3	Ø	Data_in(3)
End_T	1	1

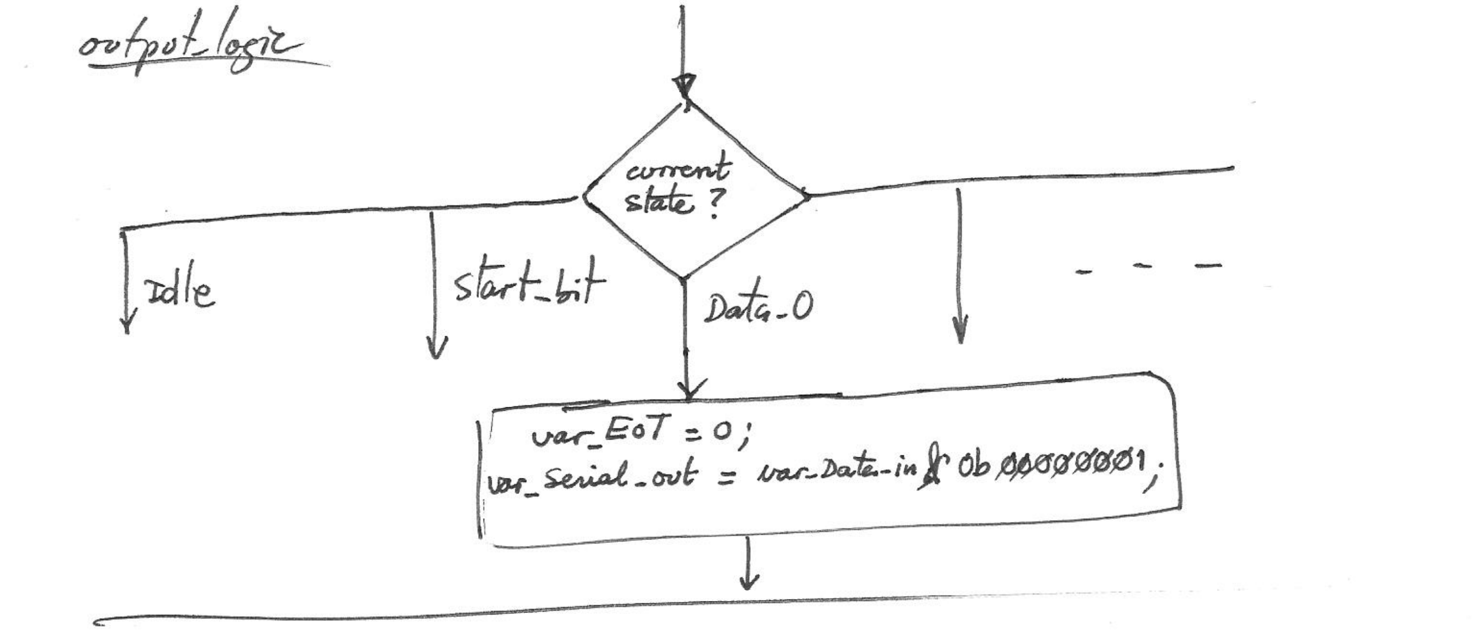
```
var_serial_out = var_Data_in &
                 0bØØØØØØØ1;
var_serial_out = (var_Data_in & 0bØØØØØØ1Ø)
                 >>1;
var_serial_out =
  (var_Data_in & 0bØØØØØ1ØØ) >> 2;
  (var_Data_in & 0bØØØØ1ØØØ ) >> 3;
```

Before translating these truth tables in C which will be done using switch-case instruction, we must have a behaviorral interpretation of the Truth tables, for instance using flow charts

state logic



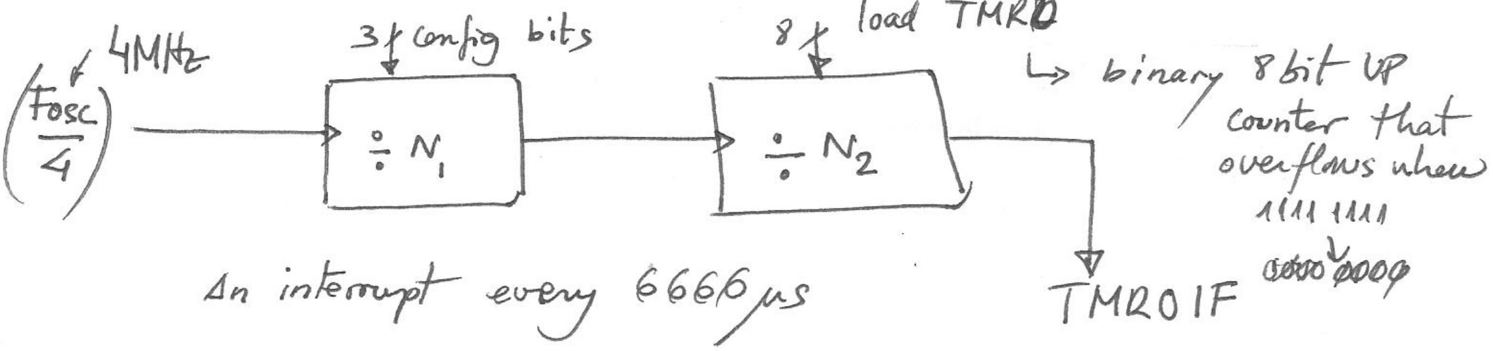
output logic



```
var_EOT = 0;
var_Serial_out = var_Data_in & 0b00000001;
```

h) ⇨ FSM + Datapath (Timer) ⇒ <u>dedicated processor</u>

To save components, for instance an external baud generator (the 150 Hz clock), we can use internal peripherals like TMR0 or TMR2 to perform this task of programming several transmission frequencies. For instance, TMR0 to generate 150 Hz timing period:


4MHz
Fosc/4
3 config bits
÷ N1
8 load TMR0
÷ N2
↳ binary 8 bit UP counter that overflows when 1111 1111 → 0000 0000
TMR0IF

An interrupt every 6666 µs

$$6666\,\mu s = \left(\frac{4}{F_{osc}}\right) \cdot N_1 \cdot N_2 \Rightarrow 6656\,\mu s$$

1µs

N_1	N_2
256	26
128	52
64	104
32	208

Possible values prescaler 2^n values

Load TMR0 = (256 − 208)

which means a transmission frequency of 150.24 b/s (0,16% error)

To obtain more precision we can change the system oscillator to 6MHz and N1 = ÷1 and use TMR0 in 16-bit mode $N_2 = 10000$ (TMR0 = 65536 − 10000)

$$\left(\frac{4}{F_{osc}}\right) \cdot 1 \cdot (10000) = 6666\,\mu s$$

6,6 µs

After this initial <u>specify & plan</u> you can start developing and testing copying and adapting a similar circuit like the Timer_18,5s in P11' to complete the project.