Problem 3
Example solution with comments and discussion

a) Hardware schematic

---

- System controller to run the microcomputer as a synchronous machine
- Each assembly is executed in a time of \( T \), excepting jump instructions which takes \( \frac{T}{2} \)
- Data_in pins as inputs
- ST, CLK pin as inputs
- Serial_out as EOT as output

---

\[
\begin{array}{cccccccc}
T & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
TRISA & & 1 & 1 & & & & \\
TRISB & & & & 1 & 1 & & \\
TRISC & & 0 & 0 & & & & \\
TRISD & & & & 1 & 1 & & \\
\end{array}
\]

C code

\[
\text{TRISA} = 000000110; \\
\text{TRISB} = 000000011; \\
\text{TRISC} = 0x00; \\
\text{TRISD} = 0b11000000;
\]

(other bit can be configured as inputs or outputs, but it is preferable as output '0')

Other initialisation instructions include GIE = 1 to allow
interrupts from the RB1( INT1) start transmitting and the
selection of the active edge \( \uparrow \) or \( \downarrow \). \( \text{INTEDG}1 = 1; \)
b) The system to be designed is essentially a 4-bit right-shift register

\[ \text{ST} \quad \text{Dat-in} \quad \text{Serial-out} \quad \text{EoT} \]

\[ \text{CLK} \]

ST is the shift order to start transmission
Parallel in \(\rightarrow\) Serial-out

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

```
read_inputs() \rightarrow \text{var. Data-in} \rightarrow \text{output logics (cc2)}
```

```
\text{Interrupt logic INT0} \rightarrow \text{var. CLK-flag}
```

```
\text{Interrupt logic INT1} \rightarrow \text{var. ST-flag}
```

```
\text{state logics (cc1)} \rightarrow \text{var. current state}
```

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

```
\begin{array}{c|c|c|c|c|c|c}
\hline
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
```

Ascii code. For instance, a capital letter to encode each state 'A', 'B', ...
General software organisation from the F.S.M. strategy

1. `init_system()`
   - To set up input & output pins. To enable interrupts.
2. `read_inputs()`
   - To read `Data_in (3..0)` in case of it has to be transmitted.
3. `state_logic()`
   - To start and process the transmission sequence through the different states.
4. `output_logic()`
   - To calculate the output variable depending in which state the F.S.M currently is.
5. `write_outputs()`
   - Transfer the variables to the corresponding bits.

Infinite loop

At any time...

1. `ISR()`
   - To set the variables that depend on edges:
     - `CLK` → `ST`
     - `var_ST_flag`
     - `var_CLK_flag`

---

The `INTIE = 1` all the time to allow the detection of the `ST` edge to start transmissions.

Once the transmission has started, the `INTO` interrupt enable must be enabled (`INTOIE = 1`) to allow detection of the `CLK` edges.

It can be done in different ways; but 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 `CLK` is detected because in this way the `Data_in` is truly "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:

\[
\begin{array}{c}
\var{\text{PORT A}} \\
0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \\
\end{array}
\]

\[
\begin{array}{c}
\var{\text{PORT D}} \\
0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \\
\end{array}
\]

\[
\begin{array}{c}
\var{\text{Mask}} \\
0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \\
\end{array}
\]

\[
\begin{array}{c}
\var{\text{Mask}} \\
0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \\
\end{array}
\]

\[
\begin{array}{c}
\var{\text{AND}} \\
0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \\
\end{array}
\]

\[
\begin{array}{c}
\var{\text{Var. buf1}} = \var{\text{PORT A}} \& \var{\text{OB \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0}}; \\
\var{\text{Var. buf1}} = \var{\text{Var. buf1}} \ll 1; \\
\var{\text{Var. buf2}} = \var{\text{PORT D}} \& \var{\text{OB \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0}}; \\
\var{\text{Var. buf2}} = \var{\text{Var. buf2}} \gg 6; \\
\var{\text{Var. Data_in}} = \var{\text{Var. buf1}} | \var{\text{Var. buf2}}; \\
\end{array}
\]

\[
\begin{array}{c}
\var{\text{Final Result}} \\
0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \\
\end{array}
\]

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

Objective:

\[
\begin{array}{cccccc}
6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\text{PORT C}
\end{array}
\]

\[
\begin{array}{ccccccc}
\text{Serial_out} & \text{EoT} & \text{ Serial_out} & \text{EoT} \\
\end{array}
\]

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.

\[
\begin{align*}
\text{var\_buf 1} &= \text{PORT C} \& 0b11\text{\_11011}; \\
\text{var\_buf 2} &= \text{var\_EoT} \ll 2; \\
\text{var\_buf 3} &= \text{var\_Serial\_out} \ll 5; \\
\text{var\_buf 3} &= \text{var\_buf 3} | \text{var\_buf 2}; \\
\text{PORT C} &= \text{var\_buf 3};
\end{align*}
\]

* 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.

\[
\text{PORT C} = (\text{var\_EoT} \ll 2) | (\text{var\_Serial\_out} \ll 5) | (\text{PORT C} \& 0b11\text{\_11011});
\]
e) ISR() is the function for organizing the tasks associated with the interrupt. Any time that an interrupt occurs the main program execution stops, all the register, flags and environment is saved, and the program counter jumps to the interrupt vector where is written the ISR() assembly code.

In this application we have 2 interrupts of the same kind (external) to detect edges in ST and CLK signals (INT1 and INT0).

Later the TMRO can be used as an interrupt source to replace INT0 and save an external circuit by means of an internal peripheral.

---

**Diagram:**

```
start

INT1IF = 1?
  
  Yes
  
  Set the software flag
  
  Var_ST_flag = 1
  INT1IF = 0

  No

  INTOIF = 1?
    
    Yes
    
    INTOE = '1 or '0' depending on the state
    
    INTO1E = 1
    
    ST
    
    Clear INTOIF order
    
    INT1IE = 1
    
    All the time to be able to trigger continuous transmission

  No

  GIE = 1

  Interrupt vector order

edge detection logic

hardware latches RS

Edge detection logic

Clear INTOIF order

INT0IF = 0

S  Q

R

clear the interrupt flag when used
```
f) This the application state diagram to run the FSM

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

(\text{Serial. out = 1}) \quad (\text{EoT} = \emptyset) \quad \text{End T} \xrightarrow{\text{CLK}=1} \text{Idle} \xrightarrow{\text{CLK}=0} \text{ST}=0 \quad \text{ST}=1 \text{ and } \text{CLK}=1 \xrightarrow{\text{CLK}=0} \text{Shut \textit{off}} \xrightarrow{\text{CLK}=0} \text{Data}_0 \xrightarrow{\text{CLK}=0} \text{Data}_1 \xrightarrow{\text{CLK}=1} \text{Data}_2 \xrightarrow{\text{CLK}=1} \text{Data}_3 \xrightarrow{\text{CLK}=0} \text{CLK}=0 \xrightarrow{\text{CLK}=0} \text{CLK}=1 \xrightarrow{\text{CLK}=0} \text{Serial. out = Data. in (3)} \quad (\text{EoT} = \emptyset) \xrightarrow{\text{CLK}=1} \text{Data}_2 \xrightarrow{\text{CLK}=0} \text{Data}_1 \xrightarrow{\text{CLK}=1} \text{Data}_0 \xrightarrow{\text{CLK}=0} \text{Serial. out = Data. in (2)} \quad (\text{EoT} = \emptyset) \xrightarrow{\text{CLK}=1} \text{Data}_1 \xrightarrow{\text{CLK}=0} \text{Data}_0 \xrightarrow{\text{CLK}=0} \text{Serial. out = Data. in (1)} \quad (\text{EoT} = \emptyset) \xrightarrow{\text{CLK}=1} \text{Data}_0 \xrightarrow{\text{CLK}=0} \text{Serial. out = Data. in (0)} \quad (\text{EoT} = \emptyset) \xrightarrow{\text{CLK}=1} \text{Data}_0

It takes 1 CLK period to advance to the next state (150 Hz = 6.6 ms).

\text{NOTE:} \quad \text{CLK is var. CLK\_flag} \quad \text{RAM memory variables which are set reading or } \text{ST is var. ST\_flag} \quad \text{using interrupts}\quad \text{Data. in is var. Data. in} \quad \text{Serial. out is var. Serial. out} \quad \text{EoT is var. EoT} \quad \text{CC2 = output logic will set this variables}

⇒ In this way, the main program runs continuously (measure the time required to run the main program with breakpoint) but each bit is transmitted at 1506/s because it's controlled by the CLK flag.
9) From the state diagram we can solve the "combinational input" CCL (state logic) inferring the truth table that generates all the 14 arrows (state transitions). See section 8.

<table>
<thead>
<tr>
<th>current state</th>
<th>ST</th>
<th>CLK</th>
<th>current state +</th>
<th>next loop turn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>0</td>
<td>X</td>
<td>Idle</td>
<td>all the transitions in Idle state</td>
</tr>
<tr>
<td>Idle</td>
<td>1</td>
<td>Ø</td>
<td>Idle</td>
<td></td>
</tr>
<tr>
<td>Idle</td>
<td>1</td>
<td>1</td>
<td>start_bit</td>
<td>ST is var_ST_flag</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>start_bit</th>
<th>ST</th>
<th>CLK</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Ø</td>
<td>Ø</td>
<td>start_bit</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>Data_0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>End_T</td>
<td>X</td>
<td>Ø</td>
<td>End_T</td>
</tr>
<tr>
<td>End_T</td>
<td>X</td>
<td>1</td>
<td>Idle</td>
</tr>
</tbody>
</table>

In the same way:

<table>
<thead>
<tr>
<th>current state</th>
<th>EOT</th>
<th>Serial out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Ø</td>
<td>1</td>
</tr>
<tr>
<td>Start_bit</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>Data_0</td>
<td>Ø</td>
<td>Data_in(0)</td>
</tr>
<tr>
<td>Data_1</td>
<td>Ø</td>
<td>Data_in(1)</td>
</tr>
<tr>
<td>Data_2</td>
<td>Ø</td>
<td>Data_in(2)</td>
</tr>
<tr>
<td>Data_3</td>
<td>Ø</td>
<td>Data_in(3)</td>
</tr>
<tr>
<td>End_T</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

For the Serial out, we have:

\[ \text{var\_Serial\_out} = \text{var\_Data\_in} \& 0b00000001; \]

\[ \text{var\_Serial\_out} = (\text{var\_Data\_in} \& 0b00000010) \gg 1; \]

\[ \text{var\_Serial\_out} = (\text{var\_Data\_in} \& 0b00000100) \gg 2; \]

\[ (\text{var\_Data\_in} \& 0b00001000) \gg 3; \]
Before translating these truth tables in C which will be done using switch-case instructions, we must have a behavioural interpretation of the Truth tables, for instance using flow charts.

```
var_E07 = 0;
var_serial_out = var_data_in &'0b00000001;
```
FSM + Datapath (Timer) \rightarrow \text{dedicated processor}

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

\[
\begin{array}{c}
4 \text{MHz} \\
\frac{f_{\text{osc}}}{4}
\end{array}
\xrightarrow{3 \text{ config bits}}
\begin{array}{c}
\div N_1 \\
\div N_2
\end{array}
\xrightarrow{8 \text{ load TMRO}}
\text{binary 8 bit UP counter that overflows when 1111 1111}
\xrightarrow{1 \text{M overflow}}
\text{TMRO VOF}
\]

An interrupt every 6666 µs

\[
6666 \mu s = \left(\frac{4}{f_{\text{osc}}}\right) \cdot N_1 \cdot N_2 \Rightarrow 6656 \mu s
\]

\[
\begin{array}{c|c|c}
1 \mu s & 256 & 28 \\
69 & 52 & 69 \\
32 & 208 &
\end{array}
\]

Possible values

\[
\begin{array}{c}
\text{prescaler} \\
2^{16} \text{ value}
\end{array}
\xrightarrow{\text{load TMRO}}
(256 - 208)
\]

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 \( N_1 = \frac{1}{4} \) and use TMRO in 16-bit mode \( N_2 = 10000 \)

\[
\left(\frac{4}{f_{\text{osc}}}\right) \cdot 1 \cdot 10000 = 6666 \mu s
\]

6.6 µs

After this initial specify & plan you can start developing and testing copying and adapting a similar circuit like the Timer 185s in P11 to complete the project.