State Machines

Goal: Design a state machine with four unique LED states with an input that cycles them forward and another that cycles backwards. Our application will use two LEDs: one red, one green. The desired configurations will be denoted as follows:

Off State:	O - Both LEDs are off (00)	**Green State**:	G - Only the green LED is on (10)
Red State:	R - Only the red LED is on (01)	**Both State**:	B - Both LEDs are on (11)

Notice that our state is encoded as b_1b_0 where b_1 is the state of the green LED and b_0 for the red LED. The possible bit states are 0 for off and 1 for on. An input from button 1 (B1) or a "Next (N/0)" state message from the client (UARTRx.N ISR) will cycle us forwards: $\{\ O, R, G, B\ \}$ (repeat). While an input from button 2 (B2) or a "Previous (P/1)" state message from the client (UARTRx.P ISR) will cycle us backwards: $\{\ B, G, R, O\ \}$ (repeat).We'll initialize in the off (O) off state.

Simplified State Machine

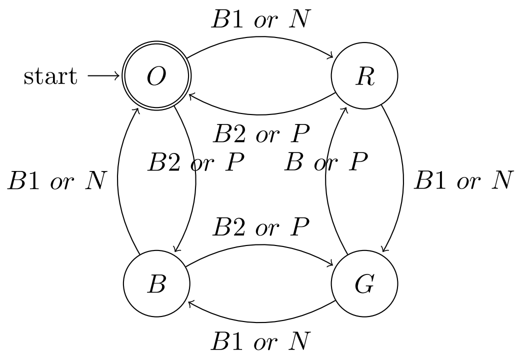

Figure 1: Simplified state machine (Moore).

Refined State Machine

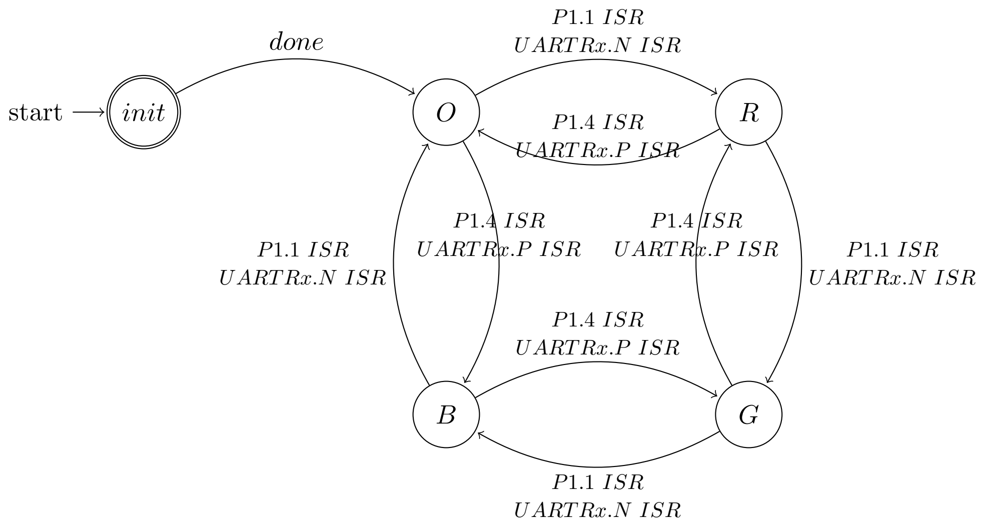

Figure 2: Refined state machine (Moore).

Code and Architecture Description

The application consists of a state machine on the board that broadcasts state changes to a python client.

— **Board Program** —
On the board, we have configured P1.1 and P1.4 as GPIO active-low inputs (with pull-up internal resistors) with interrupts enabled, P1.0 and P2.1 as GPIO active-high outputs for a red and green led, and P1.2 and P1.3 as secondary module function (SEL0, SEL1 as 10) for UART communication — the UCRXIE interrupt request has also been enabled for UART receive events.

The majority of the program logic lies in the `state_controller` function, where state transitions are made based on a given input. The current program state is stored in a static variable. The implemented state machine itself can be seen in Figures 1 and 2 above.

When button 1 (P1.1) is pressed, or an encoded '0' is received in the UART Rx buffer from the client, the corresponding ISR calls the `state_controller` with a `NEXT_STATE=1` input, updating the state machine to the next state and changing the output accordingly. When button 2 (P1.4) is pressed, or an encoded '1' is received in the UART Rx buffer from the client, we call the `state_controller` with a `PREVIOUS_STATE=0` input instead, updating the state machine to the previous state and changing the output accordingly.

For each state transition, a UART transmission is made with the `UART0_putchar` function containing the current state's corresponding encoded char code (i.e char as unsigned byte) - this is the decimal representation of the current state's bianry encoding (off $= 00_2 \Rightarrow$ '0', red $= 01_2 \Rightarrow$ '1', green $= 10_2 \Rightarrow$ '2', and both $= 11_2 \Rightarrow$ '3'). When the client is first started, a unique UART input of an encoded '2' is received, this will make the board transmit the current state without changing state to initialize the client.

— **Client Application** —
The client application consists of a python tkinter GUI application that has two buttons for sending a next state and previous state command, as well as a label for displaying the current state. The application also works in the terminal, with a 1 input for next state and 0 for previous state.

The serial connection with UART is acheived by using the pySerial module. We have created a *UART* class that handles the connection initialization and opening of the UART port. The class itself also has the according methods for serially transmitting state update commands to the board, in addition to two methods that will run in two asynchronous daemon threads (daemon so they exit with main). First, there is the `listen_for_state` method which continuously polls the serial port for any bytes that are received from the board communicating its current state. Once received, they are decoded and passed to the `_update_state` method which updates the client state to match the one on the board and prints out the updated state. The `get_command` thread continuously polls user input (without blocking the main thread) and, once received, transmits the appropriate state update command to the board by calling the `_transmit_char` method.

An *Application* class is responsible for initializing the UART connection instance and threads, in addition to setting up the tkinter window and widgets. On the GUI, the "next state" button simply calls the UART instance's `_next_state` method which transmits a '0' to the board while the "previous state" button calls the UART instance's `_previous_state` method to transmit a '1'. In initialization, the `_get_state` method transmits a '2' to get the current state in cases where the app is started after the board has already changed state. A tkinter label bound to a textvariable (class member) for the current state is used to show the current state: the UART object's `_update_state` method changes this variable accordingly. To run the application, an instance of *Application* is created and its run method is invoked to start the tkinter mainloop.