

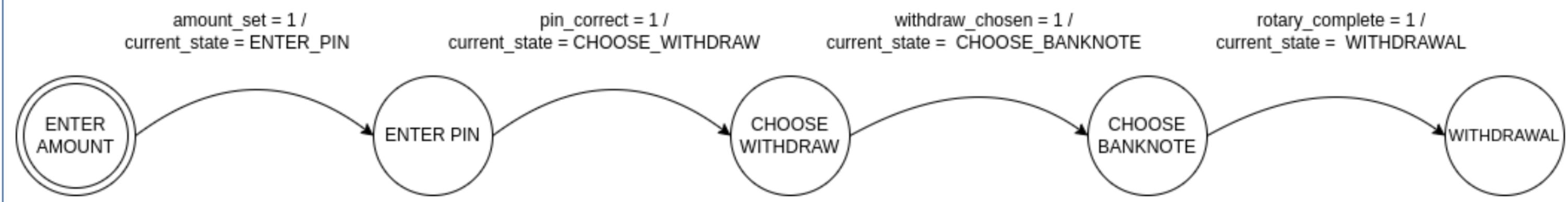
# ATM MACHINE

## EMP Final Assignment

- Daniel  
- Mads  
- Valdemar

### System control FSM

The following finite state machine (FSM) is implemented in the FSM task and serves as the "brain" of the ATM's control system, ensuring that user interactions follow a logical sequence. Each transition is driven by specific flags, set by the corresponding task for that state. This ensures secure and orderly processing of ATM transactions. The design creates a linear and predictable user experience.



The transitions are driven by the flags "amount\_set", "pin\_correct", "withdraw\_chosen", and "rotary\_complete". These are set within the task responsible for operations in the current state. For example, when the system is in the "ENTER AMOUNT" state, the task responsible for receiving the withdrawable amount from the user has control of the system until it sets the "amount\_set" flag. This indicates to the control FSM that the task is finished, which subsequently transitions the system to the "ENTER PIN" state.

### Task control system

The current state from the system control is what decides if a task is allowed to run. Each time a task starts an execution cycle it checks if the current state matches the one where the task is supposed to run. This is implemented using a getState method as shown in the figure below. The getState method works by taking the mutex, protecting the current state and returning the state. If the state is not the one intended for that task, it breaks out of that execution cycle immediately. Otherwise, it allows the execution to continue as normal.

```

state getState()
{
    state outp;
    if (xSemaphoreTake(xStateMutex, portMAX_DELAY) == pdTRUE)
    {
        outp = current_state;
        xSemaphoreGive(xStateMutex);
    }
    return outp;
}
  
```

```

149 void rotary_task(void *pvParameters)
150 {
151     * Input : -
152     * Output : -
153     * Function :
154 }
155 {
156     while (1)
157     {
158         if (getState() == CHOOSE_BANKNOTE)
159         {
160             if (first_run_banknote)
161             {
162                 clr_LCD();
163                 set_cursor(0x0C);
164                 INTBU.banknote_idx[1] = "Choose banknote";
165             }
166         }
167     }
168 }
  
```

The figure to the right shows an if statement in the start of the task that makes sure the current state is correct before proceeding with execution.  
The same approach is used for all five tasks that correspond to a state.

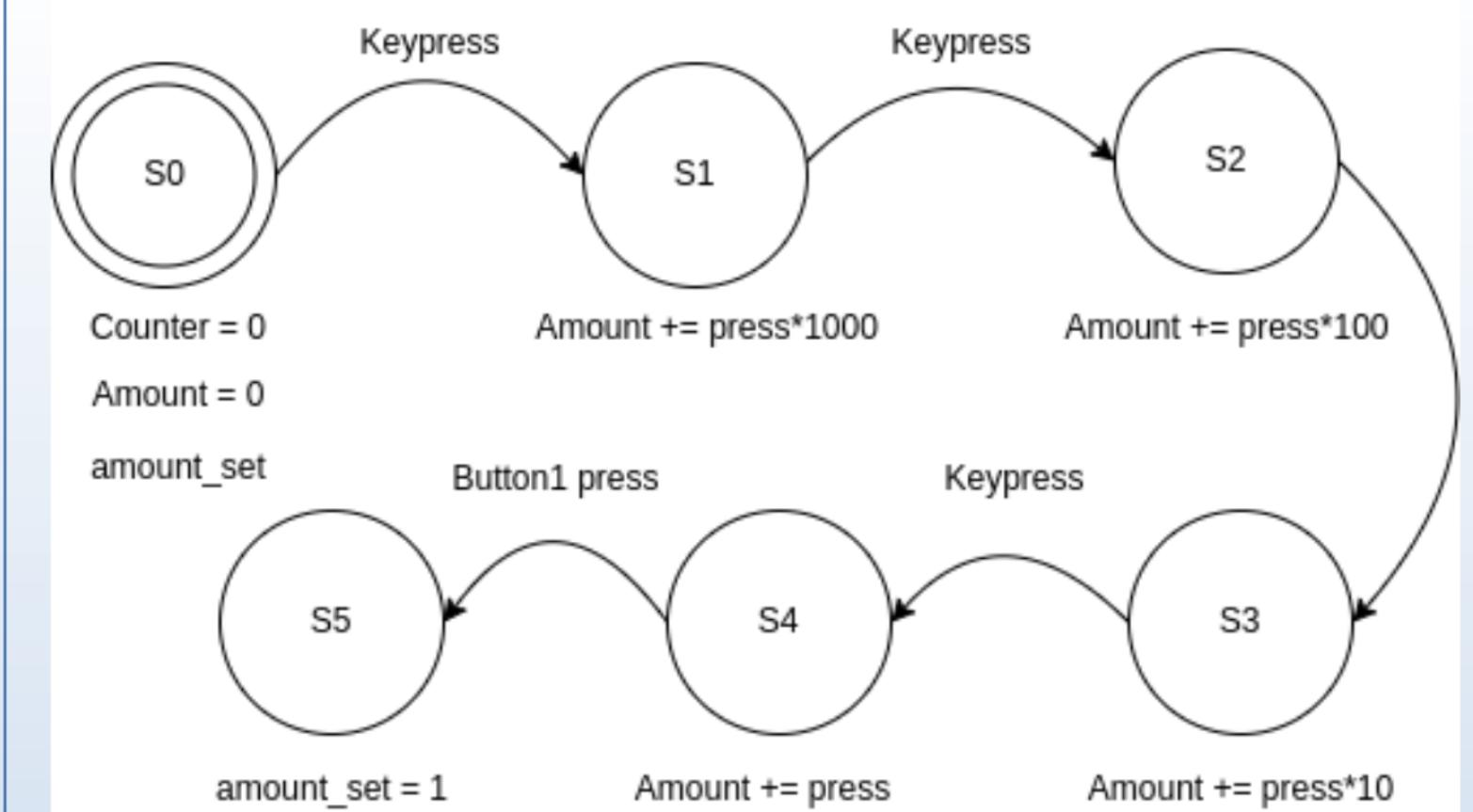
### Enter amount & enter pin tasks

The "enter amount task" and "enter pin" tasks work similar to each other. In each execution cycle they take an input from the keypad queue. It is made sure that it is an appropriate key before entering a switch case. The switch statement is dependent on what the current cycle number is. For all cases the key will be converted from an ASCII character to an integer. Depending on the cycle number, the integer will be multiplied by a thousand, a hundred, ten or one before being summed to the "amount\_value" or "pin\_value". After four runs, the value is stored. If it is the amount task that is running, it starts looking for a button press to set the transition flag.

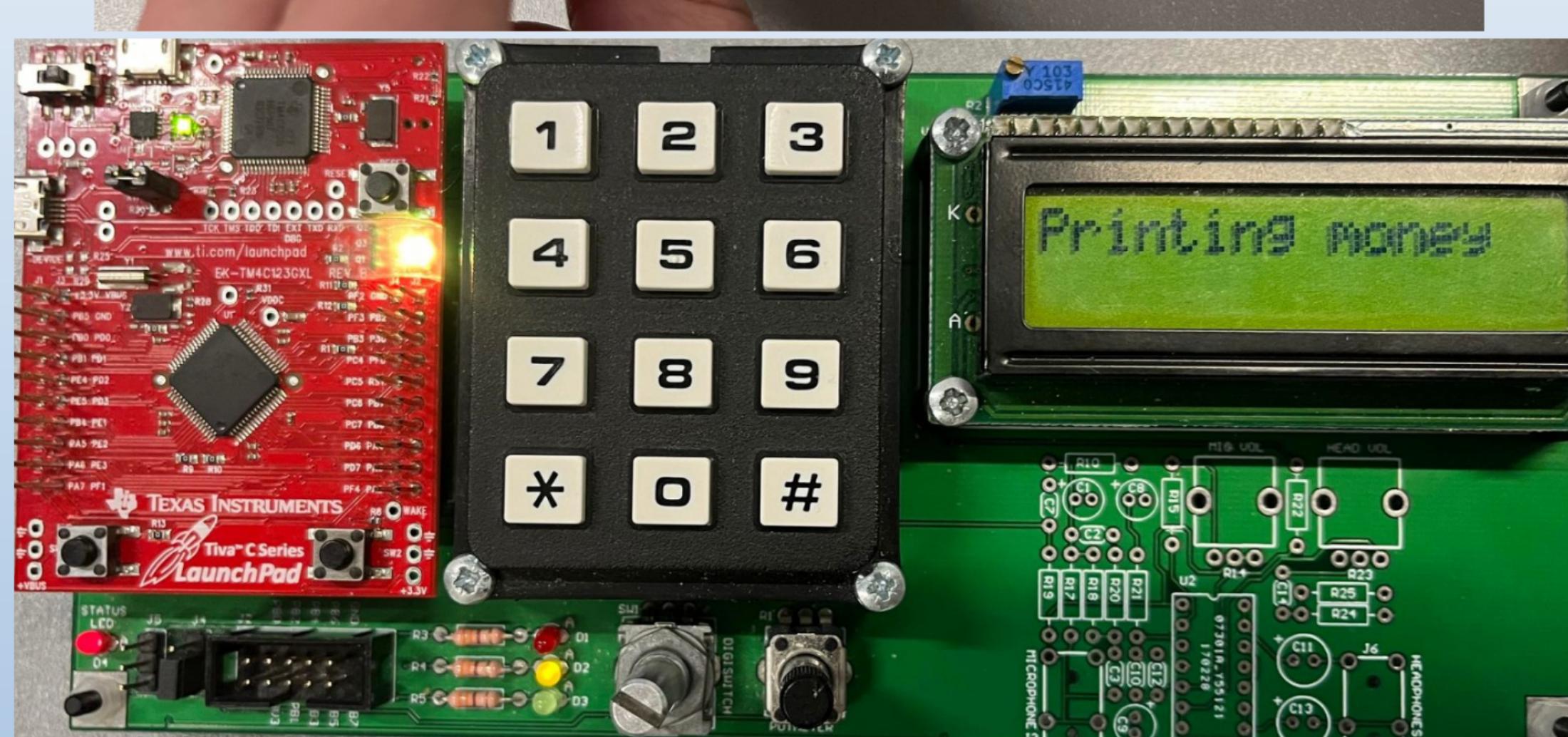
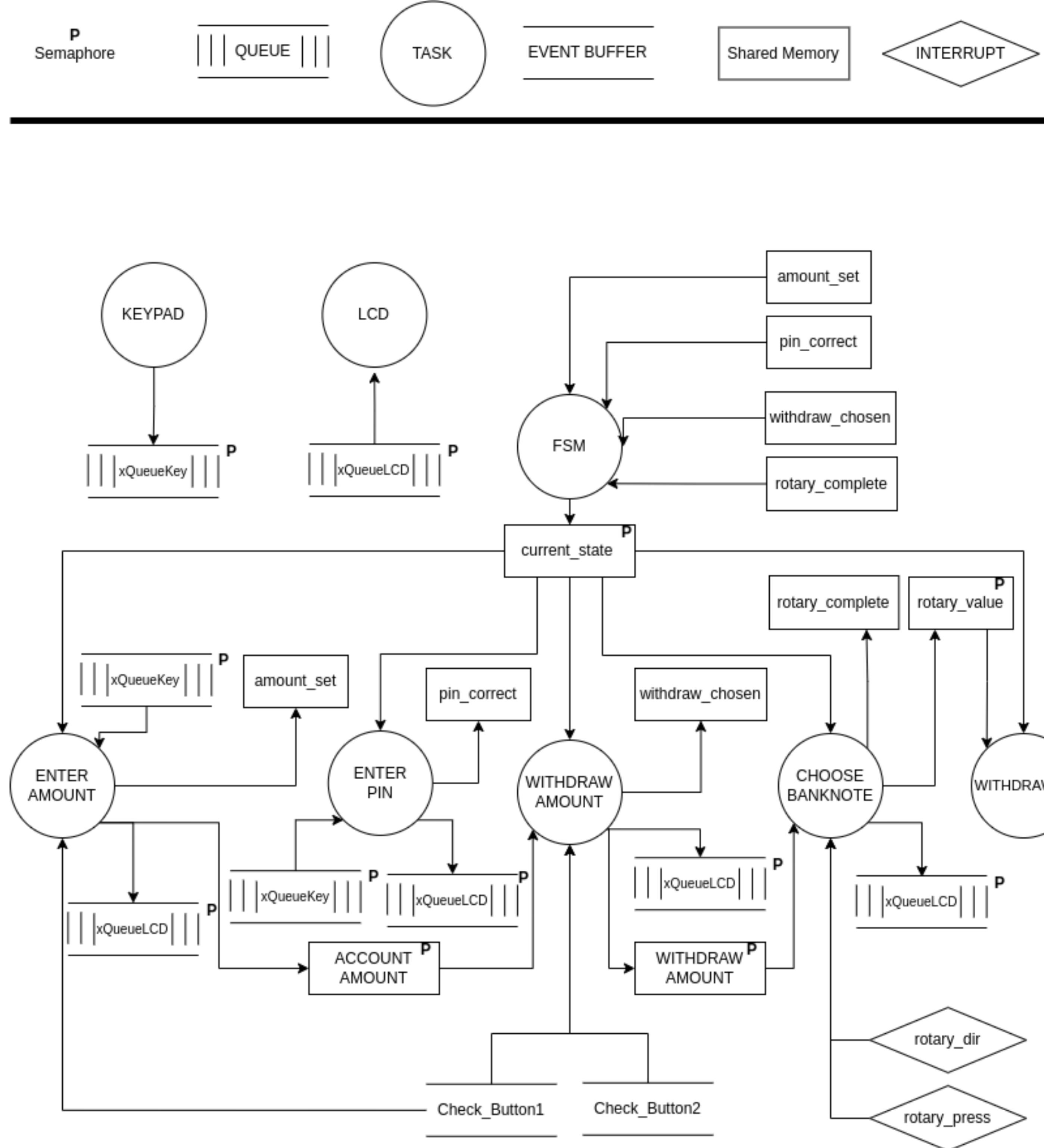
If it is the pin task that is running it checks if the four inputs add up to a number divisible by 8. If so, the pin correct flag is set.

```

181 void enter_amount_task(void *pvParameters)
182 {
183
184     while (1)
185     {
186
187         if (getState() == ENTER_AMOUNT)
188         {
189
190             if (first_run)
191             {
192                 clr_LCD();
193                 INTBU.enter_amount[] = "Enter amount:";
194                 wr_str_LCD(intBU.enter_amount);
195                 move_LCD(0, 1);
196                 vTaskDelay(100 / portTICK_RATE_MS);
197                 first_run = 0;
198             }
199
200             INTBU.key_press = 0;
201
202             if (get_keyboard(&key_press))
203             {
204
205                 if (key_press == '#') || key_press == ' ')
206                 {
207                 }
208                 else
209                 {
210                     if (i < 4)
211                     {
212                         xQueueSend(xQueueLCD, &key_press, portMAX_DELAY);
213                         switch (i)
214                         {
215                             case 0:
216                                 tmp_amount = (key_press - ASCII_TO_INT) * 1000;
217                                 break;
218                             case 1:
219                                 tmp_amount = tmp_amount + (key_press - ASCII_TO_INT) * 100;
220                                 break;
221                             default:
222                                 break;
223                         }
224                     }
225                     i++;
226                 }
227             }
228
229             if (button_pushed() && i > 3)
230             {
231                 amount_set = 1;
232             }
233         }
234     }
235
236     if (button_pushed() && i > 3)
237     {
238         amount_set = 1;
239     }
240 }
241
242
  
```



[Full demonstration video link:](#)



### Rotary encoder

The rotary encoder is implemented using interrupts. It is setup to send an interrupt on the rising edge of switch pin A (PA5) and on a press on the digiswitch (PA7).

```

// Interrupt stuff
GPIO_PORTA_ISR_R &= ~PAS; // PAS is edge-sensitive
GPIO_PORTA_IER_R &= PIS; // PAS is not both edges
GPIO_PORTA_IDR_R |= PAS; // Trigger on rising edge for PAS
GPIO_PORTA_ICR_R |= PAS + PA5; // Clear any prior interrupts on PAS
GPIO_PORTA_IM_R |= PA5 + PA5; // Enable interrupt on PAS
NVIC_ENG_R = 1 << 0; // Enable IRQ for GPIO Port A
  
```

The "else if" on l. 127 is responsible for handling interrupts from the digiswitch turning. It works by checking the value of switch pin B to determine if the interrupt came from turning clockwise or counter clockwise. It then uses a semaphore to signal the task that is responsible for keeping track.

```

116 void GPIOA_Handler(void)
117 {
118     if (GPIO_PORTA_RIS_R & PA7) // Check if interrupt was caused by PA7 (rotary press)
119     {
120         rotary_pressed = 1;
121         set_rotary_complete();
122
123         GPIO_PORTA_ICR_R |= PA7; // clear interrupt flag
124     }
125     else if (GPIO_PORTA_RIS_R & PAS && rotary_pressed) // Check if interrupt was caused by PAS
126     {
127         rotary_dir = ((GPIO_PORTA_DATA_R & (1 << 6)) != 0) ? CM : CCW; // If PA6 is high => CW, PA6 is low => CCW
128         xSemaphoreGiveFromISR(xRotaryDirectionSemaphore, NULL);
129
130         GPIO_PORTA_ICR_R |= PAS; // Clear the interrupt flag for PAS
131     }
132 }
  
```

This task takes the semaphore signal and updates the rotary value based on the direction. It cycles through the values 0, 1, and 2 depending on the direction set by the interrupt handler. This rotary value is protected by a mutex and is used as an index in an array consisting of the different banknote options.

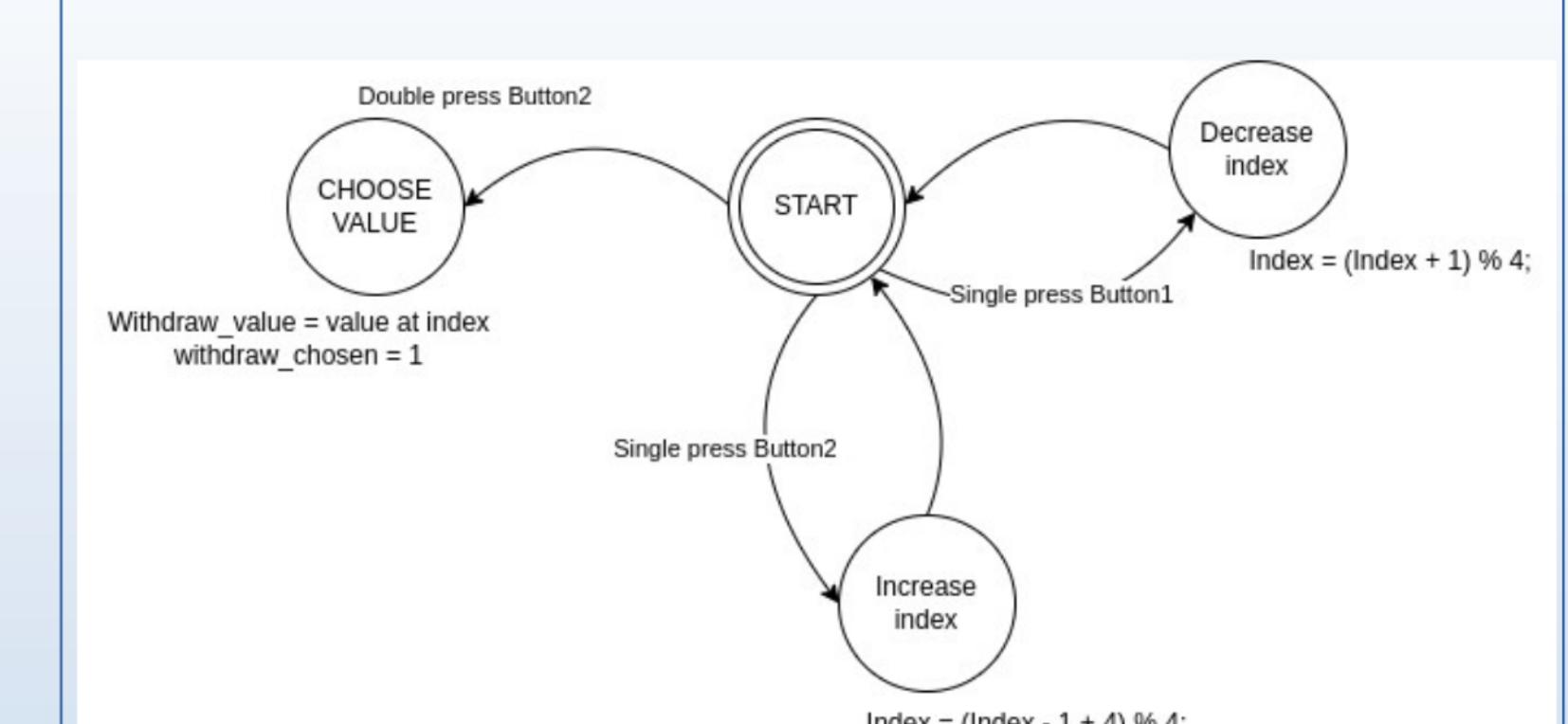
### LED flickering based on ADC

```

201     if (is_final_selected)
202     {
203         switch (final.rot_value)
204         {
205             case 0:
206                 GPIO_PORT7_DATA_R = 0x02;
207                 break;
208             case 1:
209                 GPIO_PORT7_DATA_R = 0x04;
210                 break;
211             case 2:
212                 GPIO_PORT7_DATA_R = 0x08;
213                 break;
214         }
215     }
216     INT6U adc_value = get_adc();
217     INT6U delay_ms = MAX_DELAY_MS - ((adc_value * (MAX_DELAY_MS - MIN_DELAY_MS)) / MAX_ADC_VALUE);
218     vTaskDelay(delay_ms / portTICK_RATE_MS);
219     xSemaphoreGive(xRotaryValueMutex);
220 }
  
```

delay\_ms determines the frequency for which the LEDs flicker. The ADC gives a value from 0-4095. The max. delay is 1000 ms resulting in 1 Hz. The min. delay is 100 ms resulting in 10 Hz. However, in practice this seems to be incorrect. It is suspected that various tasks running in the background may increase the delay in an unpredictable way.

### Choosing amount to withdraw



Choosing the amount to withdraw is done using the two buttons. Button 1 cycles left and button 2 cycles right. They update a value that is used to index into an array of the different withdraw amounts allowed. A double press on button 2 confirms the chosen amount.