Jaden Savoia
Ryan Pryor
Gurnoor Gill

# Detailed Project Documentation

ENCM 511

Group 27

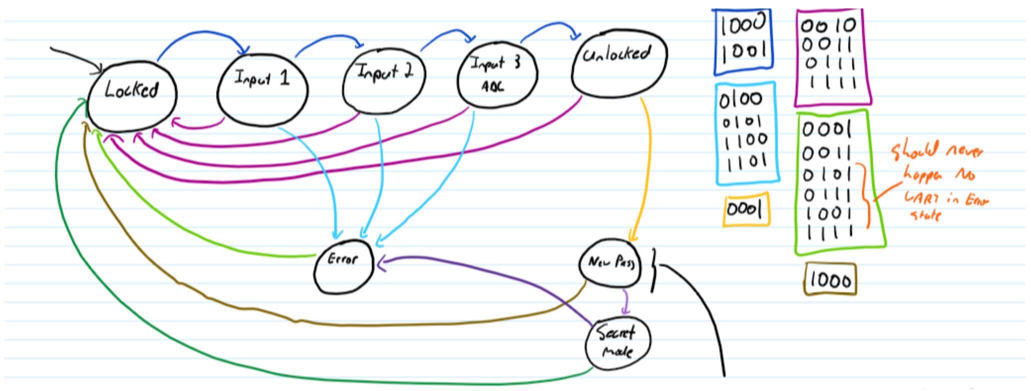
# Files

**NOTE:**
RED LED = RB8
BLUE LED = RB7
GREEN LED = RB9

## main.c

This is the main file for the program. The necessary pragma configurations are called for the function to work, it then goes into the main function of the program using the initialization functions outlined in the Init.c section for the clock, timers, interrupts, IOs, ADC, and UART.

The main loop of the program also contains the main state-machine which is implemented with a switch statement. The overview of the state-machine can be seen in the diagram below



There are two other embedded switch statements in this program, one for setting the new password and the other is for the secret mini game.

## ADC.c & ADC.h

**Global Variables:**

ADC_value → ADC variable, takes in the 10-bit converted value from the ADC sampling
scaled_ADC_value → Scaled ADC value, has a range of 0-9

**Functions**

1. ADC_sample()

This function is for ADC sampling and conversion. It simply turns on the ADC, starts sampling and then once finished turns off the ADC and sets the global variable "ADC_value".

2. ADC_display()

This function uses the bar_visual function and then scales the ADC input between 0-9 (setting the scaled_ADC_value) and displays this scaled value.

3. scale_value(int ten_bit_input)

Scales the 10 bit ADC value to an integer ranging from 0-9 by dividing the ten_bit_input value by 103.

4. bar_visual(uint16_t voltage_value)

This function creates the Bar-Graph display in the terminal based on the ADC voltage_value. The number of stars then displayed is determined by this equation:

[ADC_value * (Number of stars in graph)] / (2^10 - 1)

Which can be seen on line 50 through the simplification of the equation set on num_bars:

```
uint16_t num_bars =  voltage_value * 9 /1023;
```

## <u>Init.c & Init.h</u>

**Global variables:**

None

**Functions**

1. void CLKinit(char clock_freq)

This function is used for our clock initialization. Depending on the input of the “clock_freq” variable it will select the COSC and NOSC bits of OSCCON as well as the clock prescaler. There are 4 options:

1. 8MHz → clock_freq = 8
2. 1MHz → clock_freq = 1
3. 500kHz → clock_freq = 500
4. 31kHz → clock_freq = 31

The function starts by disabling interrupts to allow for the clock change, sets OSCONNH to the chosen value depending on the input to the function which for our default is 1Mhz clock and OSCONNL (clock change bit) to 1 to allow for the change. Runs a while loop till clock has changed, then reenables interrupts.

2. TIMERinit()

Sets Timer1 and Timer2 to the settings we require. Notable changes include:

- Timer2 is a 16-bit clock, it uses the internal clock, has a prescaler of 64 and we set continuous mode and disable the gating function.
- Timer1 is a 16-bit clock, uses internal clock, has a prescaler of 8, disables gating function, is set to continuous mode and does not sync with the external clock.

3. INTERRUPTinit()

Nesting interrupts are first enabled. Then we set the settings for Timer1 and Timer2, as well as some change notification interrupts on various buttons.

Some things to note:

- Timer2 has the second lowest priority.
- Timer1 has the lowest priority.
- The following buttons have change notification interrupts: RA4, RB4 and RA2

4. IOinit()

This function starts by disabling analog functionality of all pins. Sets AN5 (pin 8) for analog read functions. Sets the I/O as follows and enables pull ups as follows:
Inputs: RA4, RA2 and RB4
Outputs: RB7, RB8 and RB9
Enabled pull ups for: RA2, RA4, and RB4 (CN0, CN1 and CN30 respectively)

5. ADCinit()

This function runs the initializations for our ADC unit. Starts by setting it to run continuously and setting the ADC output form to INTEGER. Then we specify to allow end sampling and start conversions automatically, and to sample when SAMP bit is equal to 1.

Some things to note about the ADC:

- Reference voltage has been set to VDD (3V3) and the VSS(GND) of the MCU
- The buffer has been set one 16-word buffer
- Uses MUX-A settings
- ADC clock uses system clock, where the clock is equal to $F_{osc}$ / 2 = 500Khz
- Set sampling time to lowest at 64 cycles of ADC clock

## Interrupts.c & Interrupts.h

**Global Variables:**

int CN_flag = 0; → sets interrupt flag to 0, when this is set to 1 indicates interrupt has occurred

**Functions**

1. _T1Interrupt(void)

This ISR is triggered when Timer1 reaches its count value. The ISR starts by clearing the internal interrupt flag for Timer1 then setting a global variable “delay_on” to zero for breaking the debounce loop. Ends the ISR by turning off Timer1 and clearing Timer 1 count.

2. _T2Interrupt(void)

This ISR is triggered when Timer2 reaches its count value. The ISR starts by checking a blink global variable and if true will invert the red LED. The count global variable (initialised in state_machine.c and explained there within this document) will be increased by 1. Ends the ISR by clearing the TIMER2 count and clearing the internal Timer2 interrupt flag.

3. _CNInterrupt(void)

This ISR is triggered when a change to a push button occurs. The following push buttons trigger this ISR:

RA2, RB4, and RA4

If any PB is pressed the CN_flag is set to 1. ISR exits after clearing the internal interrupt flag.

4. _ADC1Interrupt(void)

This ISR was never used/triggered but we kept it in just in case, so not going to explain functionality as that is redundant.

## state_machine.c & state_machine.h

### Macros

Defined for state-machine in header file

```
#define LOCKED_STATE 0
#define INPUT2_STATE 1
#define INPUT3_STATE 2
#define INPUT_NUMBER_STATE 3
#define UNLOCKED_STATE 4
#define ERROR_STATE 5
#define NEW_PASS_STATE 6
#define SECRET_STATE 7
```

### Global Variables:

Button varibales

int PB1 = 0, PB2 = 0, PB3 = 0; → Button variables for indicating if they are pressed or not
int delay_on; → Variable used for while loop for debouncing
int initial_value_PB1, initial_value_PB2, initial_value_PB3; → Used to store initial button press values when debouncing

Password Variables

char set_input1 = 'a', set_input2 = 'b', set_input3 = 'c' ; Variables for storing password inputs (initially set to "abc")
uint16_t set_number = 0; → Variable for holding value of 0-9 for password (initally set as "0")

Timer Variables

uint16_t count = 0; → Used to keep track of red LED blinks for user time-out
char Timer2_blink = 1; → Determines if the timer 2 interrupt will trigger the red LED to blink

Flags and state variables

int state = LOCKED_STATE; → value for determining the state of the state-machine
int correct_input = 0, incorrect_input = 0; → Input check flags for the user input

### Functions

1. void delay_ms(unsigned int ms)

This function sets the PR2 value to set the time between TIMER2 interrupts based off the equation:

PR2 = (desired time in ms) / ([(TIMER2 Prescaler) /(Fosc/2)]*1000)

Also resets the timer count (TMR2) before leaving the function.

2. void IO_check()

This function calls the debounce function and then sets the PB1, PB2, and PB3 high if the user presses that button

3. void debounce()

This function denounces the button presses using a delay set by timer 1. This delay is set as 40ms by setting PR1 = 2500 at the beginning of the debounce function, it then sets the initial values of the buttons read by the microcontroller, delays for the 40ms and then checks to make sure the read values of the button presses are the same as before the delay. If it is true it returns from the function, if they are not the same we call the button debounce function again until the values before and after the delay are the same.

4. void IO_clear()

This function clears the input variables PB1, PB2, PB3, and ADC_value to zero

5. void Flag_clear()

This function sets the flags used for UART(RXFlag) and IO(CN_flag) checking, as well as the state determining variables (count, correct_input, incorrect_input) to zero

6. void clear_terminal()

This function clears the terminal and returns the cursor to the top right by writing specific ANSI characters to the terminal.

7. void clear_line()

This function clears the current line the cursor is on and returns the cursor to the right of the screen by writing specific ANSI characters to the terminal.

8. void UART_check(char compare_input)

Calls the RecvUartChar() function from the provided UART code and sets the output to a local variable called "UART_input" that is then compared to the given value of "compare_input" when the function was called. If they match it sets the "correct_input" flag, else it sets the "incorrect_input" flag.

9. void return_to_lock_state()

This function clears the terminal and displays a message indicating it is returning to lock state when PB1 is pressed. Used timer 2 and idle() to create 3 second delay before returning.

10. void locked_state()

This is the function that is called when the system is in the locked state. It displays a welcome message, instructions, turns off the LED and timer 2 off when the state is first entered. It then enters a loop that the system stays in until the state variable is changed from "LOCKED_STATE".

This state calls UART_check() while giving the function the "set_input1" variable for the first password character when the UART receive interrupt is triggered and RXflag is set. Then depending on the user input UART_check() will set the "correct_input" or "incorrect_input" flag to either go to the INPUT2_STATE or the ERROR_STATE.

11. void input2_state()

This function is called when the system enters the second input state. It turns the LED blinking and time-out counter on at a rate of 0.5s and displays the instructions for the state. It then enters a loop that the system stays in until the state variable is changed from "INPUT2_STATE".

This state calls UART_check() while giving the function the "set_input2" variable for the second password character when the UART receive interrupt is triggered and RXflag is set. Then depending on the user input UART_check() will set the "correct_input" or "incorrect_input" flag to either go to the INPUT3_STATE or the ERROR_STATE.

12. void input3_state()

This function is called when the system enters the third input state. It turns the LED blinking and time-out counter on at a rate of 0.4s and displays the instructions for the state. It then enters a loop that the system stays in until the state variable is changed from "INPUT3_STATE".

This state calls UART_check() while giving the function the "set_input3" variable for the third password character when the UART receive interrupt is triggered and RXflag is set. Then depending on the user input UART_check() will set the "correct_input" or "incorrect_input" flag to either go to the INPUT_NUMBER_STATE or the ERROR_STATE.

13. void input_number_state()

This function is called when the system enters the fourth input state. It turns the LED blinking and time-out counter on at a rate of 0.3s and displays the instructions for the state. It then enters a loop that the system stays in until the state variable is changed from "INPUT_NUMBER_STATE".

This state samples the ADC at the same rate of the LED blinking (0.3s) while displaying a bar graph and a scaled output of the ADC from 0-9. This is done with the ADC_sample() and the ADC_display() function outlined in the ADC.c section. When a button press occurs it will if it it PB1 it will return to locked state, if it is PB2 it will check the chosen scaled ADC value with "set_number" and if it is correct it will set the "correct_input" flag high else it will set the "incorrect_input" flag. The state logic will then either send it to the "UNLOCKED_STATE" or the "ERROR_STATE".

14. void unlocked_state()

This state is entered after the correct password has been entered. It turns off timer 2, LED and displays the instructions for the user in this state. It checks for a button press of either PB1 or PB2.

- PB1 = return to locked state
- PB2 = New-password state

15. void error_state()

This is the state entered when a user inputs an incorrect character, number or times out. It displays the instructions for the user, turns off timer 2, and the LED. Then waits for the user to press PB2 to then return back to the locked state.

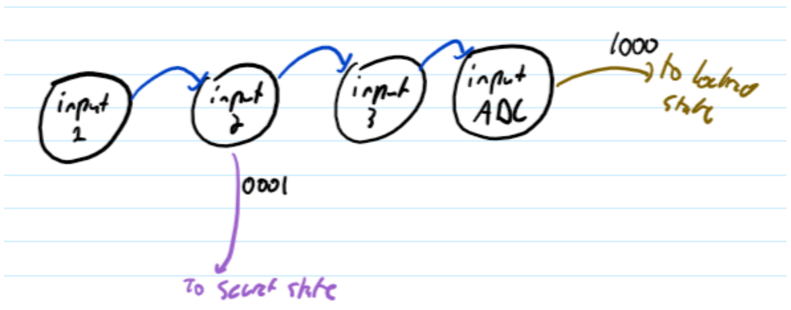
16. void new_pass_state()

This state is entered when the user pressed PB2 in the unlocked state. It calls the "reset_NewPass_StateMachine" outlined in the new_password_StateMachine section, and the "new_pass_state_machine" to enter the state-machine for setting the new password.

17. void secret_state()

This state is entered when the user pressed the PB2 during the second input state in the new password state machine. This state calls "reset_memory_game" function outlined in the memory_game.c section, and the "MemoryGame_StateMachine" to enter the state-machine for the secret mode mini game.

## new_password_StateMachine.c & snew_password_StateMachine.h



### Macros

Defined for new password state-machine in header file
#define INPUT1_STATE 0
#define INPUT2_STATE 1
#define INPUT3_STATE 2
#define INPUT_NUM_STATE 3
#define SET_NEW_PASS_STATE 4
#define PASS_SET_CANCEL_STATE 5

### Global Variables:

Temporary Input Variables
char temp_input1, temp_input2, temp_input3, temp_number;  → Temporary character variables used for holding the new password values until user has finished entering all required password elements

Flag and State Variables
char password_state = INPUT1_STATE, User_input_flag = 0; Variables for keeping track of state and determining state transitions. Initial state variable is set to the input state 1, indicating the first character is to be entered by the user.

### Functions

1. void reset_NewPass_StateMachine()

This function refreshes the new password state to the first element input state (input state 1):
    password_state = INPUT1_STATE;
Also lowers the user input flag indicating no input has been received and waits for input.

2. void new_pass_state_machine()

This is the central state machine function that has one switch case statement inside that controls the states of the new password statemachine. As long as the overall state machine is in the

‘NEW_PASS_STATE’ the switch statement remains active. Once the user has either failed an input, entered an element, or reset this machine, the state machine cycles through to the next state. Overall this machine contains 6 states:

1. INPUT1_STATE
2. INPUT2_STATE
3. INPUT3_STATE
4. INPUT_NUM_STATE
5. SET_NEW_PASS_STATE
6. PASS_SET_CANCEL_STATE

Each of these states contains a dedicated funcion that carries out the IO check for the next input and updates the state transition variables accordingly.

3. void new_character_1()

This is the first input element state (INPUT1_STATE); the structure of the function is as follows: clear the input terminal of the previous entry. If a button has been pushed check which one (ie. PB1 returns to the locked state) and debounce and  update button variables accordingly. If the input is detected (RXFlag) store the received character in temp_input1 and set the user input flag high.

If the PB1 flag is high then change states to PASS_SET_CANCEL state. If not, check if the user input flag has gone high and transition to INPUT2_STATE. If the state has changed then the overall loop is broken and we clear all IO flags and input flags and return to the main switch.

4. void new_charcter_2()

Follows the same structure as the new_character_1() function for transitioning to INPUT3_STATE in normal operation. One additional feature of this function includes checking if PB2 has been toggled. If the PB2 flag is high then the overall state machine variable is set to the SECRET_STATE and after clearing all flags (IO/Interrupt) the function returns back to the main switch statement.

5. void new_charcter_3()

Same structure as the new_character_1() function for transitioning to INPUT_NUM_STATE.

6. void new_number()

This function sets the new number for the new password. The structure follows: clear the terminal, enable the ADC and set the ADC sampling to 2 Hz (delay by 500ms). While the ADC is finishing its conversion remains in an idle state (broken by the finish ADC conversion flag). Clear the terminal and if a CNFlag is high then perform an IO check confirming which button was pressed. If PB1 was toggled then the new password state is changed to the reset state. If PB2 has been toggled, the temporary number variable is written with the converted ADC value (ranging from 0-9) and the new password state is transitioned to SET_NEW_PASS_STATE.

7. void set_password()

This state is for transitioning from the old password to the new password. This is done by updating the global password variables with the temporary ones updated using the different states. A delay is added for 3 seconds here and the overall state is returned to the locked state.

8. void canceled_password_change()

This state is used to indicate to the user that the new password change has been terminated. Here the terminal is cleared, the LED is turned off and the system is returned back to the locked state.

## Secret Functionality

The secret functionality of our program is a memory game similar to Simon says. You choose the difficulty mode of:

- Easy - Two LEDs used at blink rate of 0.5s (RB)
- Medium - Three LEDs used at a blink rate of 0.5s (RGB)
- Hard - Three LEDs used at a blink rate of 0.25s (RGB)

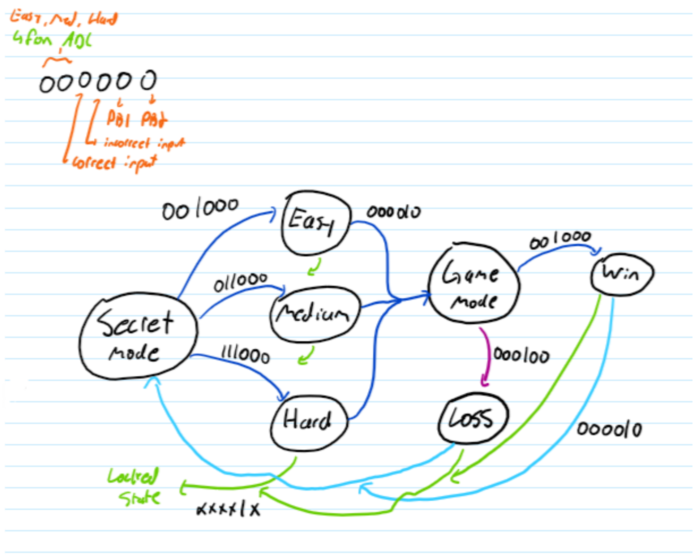
After the user must select the number of stages they want to go through from 3-12 to select the length of the pattern and difficulty further. The LEDs will then blink in a randomised order and the user must then repeat back the pattern by pressing the respective buttons for each LED.

The pattern will increase by 1 each stage until the user inputs a wrong input or the stage limit the user selected is reached. This will then enter into the a win or loss screen where the user can return to the locked state of the program, or return to the difficulty selection.

### memory_game.h

**Macros**
SECRET_STATE_ENTER = 0
EASY_STATE = 1
MEDIUM_STATE = 2
HARD_STATE = 3
PLAYING_STATE = 4
LOSS_STATE = 5
WIN_STATE = 6
GAME_CANCEL_STATE = 7

## Memory_game.c



**Global Variables:**

uint8_t randomized_array [12] → creates an array of size 12 for game LED outputs and user checking
int seed = 0 → Seed variable of the random generator
uint8_t number_of_stages = 0 → number of stages that user wants to run through, set by user
uint8_t stage_count = 0 → stage tracker for program
uint8_t game_state = SECRET_STATE_ENTER → state choosing variable for memory games state machine, default setting is ENTER or the start of the game
uint16_t blink_time = 0; → global blink time variable
uint8_t level_scaled_value → value scaled from 1-3 for gamemode selection using ADC

**Functions**

1. void reset_memory_game()

This function resets the memory game to all default values for the next time entering.

The following variables are reset to default state:

number_of_stages = 0;
stage_count = 0;
game_state = SECRET_STATE_ENTER;
blink_time = 0;
level_scaled_value = 0;

2. void randomize_array(uint8_t number_range)

This function randomizes the “randomized_array” global variable with values of either 0-1 or 0-2 dependent upon the users selection of number_range. The srand(seed) function that is called is a function from the standard library that allows the creation of  a random number. Number range is then used to determine a modulus to divide the random number by. For example, if the random number generated has a remainder of  2 after modulo of 3 is done, the number inputted into the randomized array would then be 2. This is done 12 times with a for loop to fill the randomized array.

3. void number_of_stage_display()

This function sets the number of stages from user input, and displays the value on screen. The user can choose between 3-12 stages. This function calls the scale_value() function from the ADC.c and sets that value to the number_of_stages global variable.

The number of stages chosen is then displayed on screen using the XmitUART2() function which will be covered within uart.c of this documentation.

4. void level_stage_display()

This function uses ADC functionality to display which mode you are currently selecting; Easy Mode, Medium Mode or Hard Mode.

The function called to determin the ADC current level is level_select_scale(). If the ADC output is 1, then it is set to Easy Mode, if ADC output is 2 then it is set to Medium Mode, and if the ADC output is 3 then it is set to Hard Mode.

5. uint8_t level_select_scale(int ten_bit_input)

Takes in a ten_bit_input and divides it by 342 and adds 1 and then returns a uint8_t value.

6. Void blink_LED(uint_t LED_select)

Takes in a LED_select value that can be 0, 1 or 2 and depending on this value will turn on the chosen LED 1, 2 or 3 respectively with if else statements on line 104-109. The LED_off_time is used to set Timer1’s PR1 and the program waits until the Timer1 reaches its count value to turn on the chosen LED.

The LED is then turned on after Timer1 reaches its count, then PR1 is reset back to the original blink_time, which is a global variable. The function completes by turning off all LED’s.

7. read_user_input(uint8_t correct_answer)

Whole function exists within a while loop. Within this while loop every time a button is pressed the CNInterrupt is triggered which sets the CN_flag allowing progression within the loop. If there is a double input, clear all flags and IO variables and recursively call back into read_user_input from the top. If single input is given, check to see if the given input is correct through the if else block between 138-147. Break loop with each possible input but if the incorrect input is chosen, set incorrect_input = 1 or high.

8. void two_second_delay()

Starts the function by turning on Timer2, turns on  the red LED and then delays for 2 seconds. Double Idle() call due to the chance of an extra button press by the user after the stage of the current game is complete. Finalizes by turning off the Timer2.

9. void MemoryGame_StateMachine()

This function begins when the state of the whole program enters SECRET_STATE then a switch statement is made for the global variable game_state which is set to default as SECRET_STATE_ENTER. The other possible cases for the switch statement include:

- EASY_STATE
- MEDIUM_STATE
- HARD_STATE
- PLAYING_STATE
- LOSS_STATE
- WIN_STATE
- GAME_CANCEL_STATE

After each case completes its designated function it breaks out and returns to the original main program state machine which will return to this function if the state = SECRET_STATE

10. void secret_state_enter()

Starts this function by resetting the memory game, clearing the terminal and printing out the description of the game and functionality of the buttons while in this state.

Begins by turning on Timer1, setting the ADC sampling time with delay_ms() as this just allowed us to minimize code duplication. Turns back on the red LED after the delay function and sets PR1 to 62500, disables the Timer1 Interrupt and turns on Timer1. The disabling of the interrupt allows Timer1 to always be running and constantly changes the seed for the random number generator without causing interruptions in our code.

The function then enters the main state code logic, which is a while loop that constantly samples the ADC, displays the current level of difficulty and clears the line on every change to the level of difficulty (which is controlled by turning the ADC). On button press the CN_flag is triggered which pushes the code into the next if statement.

The following things occur:

- The seed is set to the current Timer1 count
- Turn Timer1 off
- Enable Timer1 interrupts again
- Reset the Timer1 Count
- Call an IOCheck()

Then the State logic section (line 233-240) checks what the current level_scaled_value global variable is set to and which PB was pressed. This is all possible states to be selected:

- ❖ If PB1 was pressed, game was cancelled, return to locked state
- ❖ If PB2 was pressed and level_scaled_value = 1, EASY_STATE selected
- ❖ If PB2 was pressed and level_scaled_value = 2, MEDIUM_STATE selected
- ❖ If PB2 was pressed and level_scaled_value = 3, HARD_STATE selected

Calls an IO_Clear() and Flag_clear() before function completion.

11. void easy_state()

Starts this function by calling a clear_terminal(), displaying your current mode and asks to select the number of stages you would like to play. Then the function calls randomize_array with parameter of int = 2, creating a randomized array of values 0 or 1. Initialization code of this function ends with turning on the red LED again.

Then it enters the while in state code loop, and the following things occur on repeat:

- ADC sample is done to check ADC value
- Displays the current ADC value using number_of_stage_display
- Calls an Idle()
- Clears line() of current output on UART

Then a check occurs on the CN_flag to see if a button was pressed, which calls the IOCheck() to see which button if there was a PB pressed.

Then the state logic is run through as so:
- ❖ If PB1 is pressed CANCEL_GAME_STATE is set to game_state
- ❖ If PB2 is pressed PLAYING_STATE is set to game_state

When exiting the code sets the blink_time time to 0.5s and the global variable "number_of_stages" is already set by the "number_of_stages_display" function.

12. void medium_state()

Medium state follows the same logic as easy_state(), however, just changes the random generation of the array by give the "randomize_array" function a 3 and the text that is displayed in the terminal.

13. void hard_state()

Hard state follows the same logic as medium_state(), however just changes the blink_time to 0.25s and the text displayed in the terminal.

14. void playing_state()

This function is what holds the logic for going through the randomised pattern and the number of stages the user selected. It first displays the instructions with some delay readying the user for play. The code then enters the main loop of the state which has 2 main sections the LED blinking and the user input.

1. LED blinking

This runs a a for loop for the number stored in the global variable "stage_count" calling the "blink_LED" function giving it the value of the location of the for loop counter in the randomised array.

2. User input

This runs a for loop also using the  the global variable "stage_count" calling the "read_user_input" function each time through the loop. After the "read_user_input" function call it checks if an incorrect button press was done and if the incorrect_input flag is set it sets the state to LOSS_STATE.

This cycle is repeated either until an incorrect input or the number of stages the user input has been successfully completed, increasing the number of LED blinks in the pattern by 1 each time.

15. void loss_state()

This state is entered when an incorrect pattern is done when playing the memory game. It displays the instructions for the user and waits for either a PB1 press to return to locked state, or PB2 to go back to the difficulty selection for the mini-game.

16. void win_state()

This state is entered when the user completes all stages successfully when playing the memory game. It displays the instructions for the user and waits for either a PB1 press to return to locked state, or PB2 to go back to the difficulty selection for the mini-game.

17. void game_cancel_state()

This state is entered when the user presses PB1 in either the difficulty selection or number of stages selection and returns the system to the LOCKED_STATE.