```c

#include <stdio.h>
#include "simonDisplay.h"
#include "supportFiles/display.h"
#include "buttonHandler.h"
#include "flashSequence.h"
#include "verifySequence_runTest.h"
#include "simonControl.h"
#include "supportFiles/utils.h"

#include "xparameters.h"
#include "supportFiles/leds.h"
#include "supportFiles/globalTimer.h"
#include "supportFiles/interrupts.h"
#include <stdbool.h>
#include <stdint.h>

#define TOTAL_SECONDS 60
// The formula for computing the load value is based upon the formula from 4.1.1 (calculating timer intervals)
// in the Cortex-A9 MPCore Technical Reference Manual 4-2.
// Assuming that the prescaler = 0, the formula for computing the load value based upon the desired period is:
// load-value = (period * timer-clock) - 1
#define TIMER_PERIOD 50.0E-3
#define TIMER_CLOCK_FREQUENCY (XPAR_CPU_CORTEXA9_0_CPU_CLK_FREQ_HZ / 2)
#define TIMER_LOAD_VALUE ((TIMER_PERIOD * TIMER_CLOCK_FREQUENCY) - 1.0)

static uint32_t isr_functionCallCount = 0;

int main()
{
    display_init();
    display_fillScreen(DISPLAY_BLACK);

    interrupts_initAll(true);
    interrupts_setPrivateTimerLoadValue(TIMER_LOAD_VALUE);
    printf("timer load value:%ld\n\r", (int32_t) TIMER_LOAD_VALUE);
    u32 privateTimerTicksPerSecond = interrupts_getPrivateTimerTicksPerSecond();
    printf("private timer ticks per second: %ld\n\r", privateTimerTicksPerSecond);
    interrupts_enableTimerGlobalInts();
    // Initialization of the clock display is not time-dependent, do it outside of the state machine.
    // clockDisplay_init();
    // Start the private ARM timer running.
    interrupts_startArmPrivateTimer();
    // Enable interrupts at the ARM.
    interrupts_enableArmInts();
    // The while-loop just waits until the total number of timer ticks have occurred before proceeding.
    while (interrupts_isrInvocationCount() < (TOTAL_SECONDS * privateTimerTicksPerSecond));
    // All done, now disable interrupts and print out the interrupt counts.
    interrupts_disableArmInts();
    printf("isr invocation count: %ld\n\r", interrupts_isrInvocationCount());
    printf("internal interrupt count: %ld\n\r", isr_functionCallCount);
    return 0;
}
```

```
54 void isr_function() {
55     simonControl_tick();
56     flashSequence_tick();
57     verifySequence_tick();
58     buttonHandler_tick();
59     isr_functionCallCount++;
60 }
61 
62 
```

```
 2  * simonDisplay.h
 7 #ifndef SIMONDISPLAY_H_
 8 #define SIMONDISPLAY_H_
 9
10 #include <stdbool.h>
11 #include <stdint.h>
12
13 // Width, height of the simon "buttons"
14 #define SIMON_DISPLAY_BUTTON_WIDTH 60
15 #define SIMON_DISPLAY_BUTTON_HEIGHT 60
16
17 // WIdth, height of the simon "squares.
18 // Note that the video shows the squares as larger but you
19 // can use this smaller value to make the game easier to implement speed-wise.
20 #define SIMON_DISPLAY_SQUARE_WIDTH  120
21 #define SIMON_DISPLAY_SQUARE_HEIGHT 120
22
23 // Given coordinates from the touch pad, computes the region number.
24
25 // The entire touch-screen is divided into 4 rectangular regions, numbered 0 - 3.
26 // Each region will be drawn with a different color. Colored buttons remind
27 // the user which square is associated with each color. When you press
28 // a region, computeRegionNumber returns the region number that is used
29 // by the other routines.
30 /*
31 |----------|----------|
32 |          |          |
33 |    0     |     1    |
34 |  (RED)   | (YELLOW) |
35 ------------------------
36 |          |          |
37 |     2    |    3     |
38 |  (BLUE)  |  (GREEN) |
39 ------------------------
40 */
41
42 // These are the definitions for the regions.
43 #define SIMON_DISPLAY_REGION_0 0
44 #define SIMON_DISPLAY_REGION_1 1
45 #define SIMON_DISPLAY_REGION_2 2
46 #define SIMON_DISPLAY_REGION_3 3
47
48 int8_t simonDisplay_computeRegionNumber(int16_t x, int16_t y);
49
50 // Draws a colored "button" that the user can touch.
51 // The colored button is centered in the region but does not fill the region.
52 void simonDisplay_drawButton(uint8_t regionNumber);
53
54 // Convenience function that draws all of the buttons.
55 void simonDisplay_drawAllButtons();
56
57 // Convenience function that erases all of the buttons.
58 void simonDisplay_eraseAllButtons();
59
60 // Draws a bigger square that completely fills the region.
61 // If the erase argument is true, it draws the square as black background to "erase"
   it.
62 void simonDisplay_drawSquare(uint8_t regionNo, bool erase);
```

```
63
64 // Runs a brief demonstration of how buttons can be pressed and squares lit up to
   implement the user
65 // interface of the Simon game. The routine will continue to run until the touchCount
   has been reached, e.g.,
66 // the user has touched the pad touchCount times.
67
68 // I used a busy-wait delay (utils_msDelay) that uses a for-loop and just blocks until
   the time has passed.
69 // When you implement the game, you CANNOT use this function as we discussed in class.
   Implement the delay
70 // using the non-blocking state-machine approach discussed in class.
71 void simonDisplay_runTest(uint16_t touchCount);
72
73 #endif /* SIMONDISPLAY_H_ */
74
```

```c
 * simonDisplay.c

#include <stdio.h>
#include "simonDisplay.h"
#include "supportFiles/display.h"
#include "supportFiles/utils.h"

//**********CODE FROM PROFESSOR**********//
#define TOUCH_PANEL_ANALOG_PROCESSING_DELAY_IN_MS 60 // in ms
#define MAX_STR 255
#define TEXT_SIZE 2
#define TEXT_VERTICAL_POSITION 0
#define TEXT_HORIZONTAL_POSITION (DISPLAY_HEIGHT/2)
#define INSTRUCTION_LINE_1 "Touch and release to start the Simon demo."
#define INSTRUCTION_LINE_2 "Demo will terminate after %d touches."
#define DEMO_OVER_MESSAGE_LINE_1 "Simon demo terminated"
#define DEMO_OVER_MESSAGE_LINE_2 "after %d touches."
#define TEXT_VERTICAL_POSITION 0 // Start at the far left.
#define ERASE_THE_SQUARE true  // drawSquare() erases if this is passed in.
#define DRAW_THE_SQUARE false  // drawSquare() draws the square if this is passed in.

//***********MY_VAR**********//
#define DIVIDE_HALF 2 // too cut the screen width and height in half
#define CENTER_BUTTEN_X 50 //offset from the side in the x direction
#define CENTER_BUTTEN_Y 30 //offset from the side in the y direction
#define BOX_HEIGHT 160  // the height of the box that flashes
#define BOX_WIDTH 120       //the width of the bow that flashed
#define HOME_POSITION 0     //the start postion 0,0




int8_t simonDisplay_computeRegionNumber(int16_t x, int16_t y){
    if(x < (DISPLAY_WIDTH/DIVIDE_HALF)){             //see if screen was touched on the left half
        if(y < (DISPLAY_HEIGHT/DIVIDE_HALF)){        //see if screen was touched on the top half
            return SIMON_DISPLAY_REGION_0;           //return button 0
        }
        else{                                       //screen was touched on the the bottom half
            return SIMON_DISPLAY_REGION_2;          //return button 2
        }
    }
    else{                                           //screen was touched on the right half
        if(y < (DISPLAY_HEIGHT/DIVIDE_HALF)){       //see if screen was touched on the top half
            return SIMON_DISPLAY_REGION_1;          //return button 1
        }
        else{                                       //screen was touched on the the bottom half
            return SIMON_DISPLAY_REGION_3;          //return button 3
        }
    }
}

// Draws a colored "button" that the user can touch.
```

```
58 // The colored button is centered in the region but does not fill the region.
59 void simonDisplay_drawButton(uint8_t regionNumber){
60     switch (regionNumber){
61     case SIMON_DISPLAY_REGION_0: // fills button #0 in with red
62         display_fillRect(CENTER_BUTTEN_X, CENTER_BUTTEN_Y, SIMON_DISPLAY_BUTTON_WIDTH,
   SIMON_DISPLAY_BUTTON_HEIGHT, DISPLAY_RED);
63         break;                       // exits the case
64     case SIMON_DISPLAY_REGION_1: // fills button #1 in with yellow
65         display_fillRect((DISPLAY_WIDTH/DIVIDE_HALF)+CENTER_BUTTEN_X, CENTER_BUTTEN_Y,
   SIMON_DISPLAY_BUTTON_WIDTH, SIMON_DISPLAY_BUTTON_HEIGHT, DISPLAY_YELLOW);
66         break;                       // exits the case
67     case SIMON_DISPLAY_REGION_2: // fills button #2 in with blue
68         display_fillRect(CENTER_BUTTEN_X,
   (DISPLAY_HEIGHT/DIVIDE_HALF)+CENTER_BUTTEN_Y, SIMON_DISPLAY_BUTTON_WIDTH,
   SIMON_DISPLAY_BUTTON_HEIGHT, DISPLAY_BLUE);
69         break;                       // exits the case
70     case SIMON_DISPLAY_REGION_3: // fills button #3 in with green
71         display_fillRect((DISPLAY_WIDTH/DIVIDE_HALF)+CENTER_BUTTEN_X,
   (DISPLAY_HEIGHT/DIVIDE_HALF)+CENTER_BUTTEN_Y, SIMON_DISPLAY_BUTTON_WIDTH,
   SIMON_DISPLAY_BUTTON_HEIGHT, DISPLAY_GREEN);
72         break;                       // exits the case
73     }
74 };
75 // Convenience function that draws all of the buttons.
76 void simonDisplay_drawAllButtons(){
77     simonDisplay_drawButton(SIMON_DISPLAY_REGION_0); // draw the button in position #0
78     simonDisplay_drawButton(SIMON_DISPLAY_REGION_1); // draw the button in position #1
79     simonDisplay_drawButton(SIMON_DISPLAY_REGION_2); // draw the button in position #2
80     simonDisplay_drawButton(SIMON_DISPLAY_REGION_3); // draw the button in position #3
81 };
82 // Convenience function that erases all of the buttons.
83 void simonDisplay_eraseAllButtons(){ //uses the same code from simonDisplay_drawButton
   but changes the color to black
84     display_fillRect(CENTER_BUTTEN_X, CENTER_BUTTEN_Y, SIMON_DISPLAY_BUTTON_WIDTH,
   SIMON_DISPLAY_BUTTON_HEIGHT, DISPLAY_BLACK);
85     display_fillRect((DISPLAY_WIDTH/DIVIDE_HALF)+CENTER_BUTTEN_X, CENTER_BUTTEN_Y,
   SIMON_DISPLAY_BUTTON_WIDTH, SIMON_DISPLAY_BUTTON_HEIGHT, DISPLAY_BLACK);
86     display_fillRect(CENTER_BUTTEN_X, (DISPLAY_HEIGHT/DIVIDE_HALF)+CENTER_BUTTEN_Y,
   SIMON_DISPLAY_BUTTON_WIDTH, SIMON_DISPLAY_BUTTON_HEIGHT, DISPLAY_BLACK);
87     display_fillRect((DISPLAY_WIDTH/DIVIDE_HALF)+CENTER_BUTTEN_X,
   (DISPLAY_HEIGHT/DIVIDE_HALF)+CENTER_BUTTEN_Y, SIMON_DISPLAY_BUTTON_WIDTH,
   SIMON_DISPLAY_BUTTON_HEIGHT, DISPLAY_BLACK);
88 };
89
90 // Draws a bigger square that completely fills the region.
91 // If the erase argument is true, it draws the square as black background to "erase"
   it.
92 void simonDisplay_drawSquare(uint8_t regionNo, bool erase){
93     if (erase){
94         switch (regionNo){
95         case SIMON_DISPLAY_REGION_0:   // fills box #0 in with black
96             display_fillRect(HOME_POSITION, HOME_POSITION, BOX_HEIGHT, BOX_WIDTH,
   DISPLAY_BLACK);
97             break;                      // exits the case
98         case SIMON_DISPLAY_REGION_1:   // fills box #1 in with black
99             display_fillRect(BOX_HEIGHT, HOME_POSITION, BOX_HEIGHT, BOX_WIDTH,
   DISPLAY_BLACK);
100            break;                      // exits the case
```

```
101         case SIMON_DISPLAY_REGION_2:   // fills box #2 in with black
102             display_fillRect(0, BOX_WIDTH, BOX_HEIGHT, BOX_WIDTH, DISPLAY_BLACK);
103             break;                      // exits the case
104         case SIMON_DISPLAY_REGION_3:   // fills box #3 in with black
105             display_fillRect(BOX_HEIGHT, BOX_WIDTH,BOX_HEIGHT, BOX_WIDTH,
    DISPLAY_BLACK);
106             break;                      // exits the case
107         }
108     }
109     else{ // if the erase is false then fill the box with the right color
110         switch (regionNo){
111         case SIMON_DISPLAY_REGION_0:   // fills box #0 in with red
112             display_fillRect(HOME_POSITION, HOME_POSITION, BOX_HEIGHT, BOX_WIDTH,
    DISPLAY_RED);
113             break;                      // exits the case
114         case SIMON_DISPLAY_REGION_1:   // fills box #1 in with yellow
115             display_fillRect(BOX_HEIGHT, HOME_POSITION, BOX_HEIGHT, BOX_WIDTH,
    DISPLAY_YELLOW);
116             break;                      // exits the case
117         case SIMON_DISPLAY_REGION_2:   // fills box #2 in with blue
118             display_fillRect(HOME_POSITION, BOX_WIDTH, BOX_HEIGHT, BOX_WIDTH,
    DISPLAY_BLUE);
119             break;                      // exits the case
120         case SIMON_DISPLAY_REGION_3:   // fills box #3 in with green
121             display_fillRect(BOX_HEIGHT, BOX_WIDTH,BOX_HEIGHT, BOX_WIDTH,
    DISPLAY_GREEN);
122             break;                      // exits the case
123         }
124     }
125 };
126
127 // Runs a brief demonstration of how buttons can be pressed and squares lit up to
    implement the user
128 // interface of the Simon game. The routine will continue to run until the touchCount
    has been reached, e.g.,
129 // the user has touched the pad touchCount times.
130
131 // I used a busy-wait delay (utils_msDelay) that uses a for-loop and just blocks until
    the time has passed.
132 // When you implement the game, you CANNOT use this function as we discussed in class.
    Implement the delay
133 // using the non-blocking state-machine approach discussed in class.
134 void simonDisplay_runTest(uint16_t touchCount){
135     display_init();         // Always initialize the display.
136     char str[MAX_STR];      // Enough for some simple printing.
137     uint8_t regionNumber = 0;   // Convenience variable.
138     uint16_t touches = 0;   // Terminate when you receive so many touches.
139     // Write an informational message and wait for the user to touch the LCD.
140     display_fillScreen(DISPLAY_BLACK);              // clear the screen.
141     display_setCursor(TEXT_VERTICAL_POSITION, TEXT_HORIZONTAL_POSITION); // move to
    the middle of the screen.
142     display_setTextSize(TEXT_SIZE);                 // Set the text size for the
    instructions.
143     display_setTextColor(DISPLAY_RED, DISPLAY_BLACK);   // Reasonable text color.
144     sprintf(str, INSTRUCTION_LINE_1);                   // Copy the line to a buffer.
145     display_println(str);                               // Print to the LCD.
146     display_println();                                  // new-line.
147     sprintf(str, INSTRUCTION_LINE_2, touchCount);       // Copy the line to a buffer.
```

```
148     display_println(str);                                   // Print to the LCD.
149     while (!display_isTouched());       // Wait here until the screen is touched.
150     while (display_isTouched());        // Now wait until the touch is released.
151     display_fillScreen(DISPLAY_BLACK);  // Clear the screen.
152     simonDisplay_drawAllButtons();      // Draw all of the buttons.
153     bool touched = false;         // Keep track of when the pad is touched.
154     int16_t x, y;                     // Use these to keep track of coordinates.
155     uint8_t z;                        // This is the relative touch pressure.
156     while (touches < touchCount) {  // Run the loop according to the number of touches
    passed in.
157         if (!display_isTouched() && touched) {          // user has stopped touching
    the pad.
158             simonDisplay_drawSquare(regionNumber, ERASE_THE_SQUARE);  // Erase the
    square.
159             simonDisplay_drawButton(regionNumber);        // DISPLAY_REDraw the
    button.
160             touched = false;                  // Released the touch, set touched to
    false.
161         }
162         else if (display_isTouched() && !touched) {   // User started touching the
    pad.
163             touched = true;                               // Just touched the pad, set
    touched = true.
164             touches++;                                    // Keep track of the number
    of touches.
165             display_clearOldTouchData();                  // Get rid of data from
    previous touches.
166             // Must wait this many milliseconds for the chip to do analog processing.
167             utils_msDelay(TOUCH_PANEL_ANALOG_PROCESSING_DELAY_IN_MS);
168             display_getTouchedPoint(&x, &y, &z);          // After the wait, get the
    touched point.
169             regionNumber = simonDisplay_computeRegionNumber(x, y);// Compute the
    region number, see above.
170             simonDisplay_drawSquare(regionNumber, DRAW_THE_SQUARE);  // Draw the
    square (erase = false).
171         }
172     }
173     // Done with the demo, write an informational message to the user.
174     display_fillScreen(DISPLAY_BLACK);        // clear the screen.
175     // Place the cursor in the middle of the screen.
176     display_setCursor(TEXT_VERTICAL_POSITION, TEXT_HORIZONTAL_POSITION);
177     display_setTextSize(TEXT_SIZE); // Make it readable.
178     display_setTextColor(DISPLAY_RED, DISPLAY_BLACK);  // red is foreground color,
    black is background color.
179     sprintf(str, DEMO_OVER_MESSAGE_LINE_1);    // Format a string using sprintf.
180     display_println(str);                      // Print it to the LCD.
181     sprintf(str, DEMO_OVER_MESSAGE_LINE_2, touchCount);  // Format the rest of the
    string.
182     display_println(str);  // Print it to the LCD.
183     };
184
185
186
```

```
1 #ifndef BUTTONHANDLER_H_
2 #define BUTTONHANDLER_H_
3 #include <stdint.h>
4 // Get the simon region numbers. See the source code for the region numbering scheme.
5 uint8_t buttonHandler_getRegionNumber();
6
7 // Turn on the state machine. Part of the interlock.
8 void buttonHandler_enable();
9
10 // Turn off the state machine. Part of the interlock.
11 void buttonHandler_disable();
12
13 // The only thing this function does is return a boolean flag set by the buttonHandler
    state machine. To wit:
14 // Once enabled, the buttonHandler state-machine first waits for a touch. Once a touch
    is detected, the
15 // buttonHandler state-machine computes the region-number for the touched area. Next,
    the buttonHandler
16 // state-machine waits until the player removes their finger. At this point, the
    state-machine should
17 // set a bool flag that indicates the the player has removed their finger. Once the
    buttonHandler()
18 // state-machine is disabled, it should clear this flag.
19 // All buttonHandler_releasedDetected() does is return the value of this flag.
20 // As such, the body of this function should only contain a single line of code.
21 bool buttonHandler_releaseDetected();
22
23 // Standard tick function.
24 void buttonHandler_tick();
25
26 // This tests the functionality of the buttonHandler state machine.
27 // buttonHandler_runTest(int16_t touchCount) runs the test until
28 // the user has touched the screen touchCount times. It indicates
29 // that a button was pushed by drawing a large square while
30 // the button is pressed and then erasing the large square and
31 // redrawing the button when the user releases their touch.
32 void buttonHandler_runTest(int16_t touchCount);
33
34 #endif /* BUTTONHANDLER_H_ */
35
```

```c
 * buttonHandler.c
#include "buttonHandler.h"
#include "simonDisplay.h"
#include "supportFiles/display.h"
#include "supportFiles/utils.h"
#include <stdio.h>

//**********CODE_PROVIDED_BY_PROFESSOR**********//
#define RUN_TEST_TERMINATION_MESSAGE1 "buttonHandler_runTest()"  // Info message.
#define RUN_TEST_TERMINATION_MESSAGE2 "terminated."              // Info message.
#define RUN_TEST_TEXT_SIZE 2                                     // Make text easy to see.
#define RUN_TEST_TICK_PERIOD_IN_MS 100                           // Assume a 100 ms tick period.
#define TEXT_MESSAGE_ORIGIN_X 0                                  // Text is written starting at the right, and
#define TEXT_MESSAGE_ORIGIN_Y (DISPLAY_HEIGHT/2)                 // middle.

//**********MY_VAR**********//
#define ENABLE_FLAG_ON 1                                            // VAR to set the ENABLE FLAG on
#define ENABLE_FLAG_OFF 0                                           // VAR to set the ENABLE FLAG off
#define TRUE 1                                                      // sets true to 1
#define FALSE 0                                                     // sets false to 0
uint8_t buttonEnableFlag = 0;                                       // Initializes the enable flag
uint8_t touchRelease = 0;                                           // initializes the touch_release VAR
uint8_t delayCounter = 0;                                           // counter to make the button_delay_st completely run
uint8_t region;                                                     // VAR to save the region where the screen is being pressed
uint8_t buttonInitFlag = 0;                                         // Flag to signal if the buttons have been printed

enum buttonHandler_st_m {                                           // sets the states of the state machine
    button_int_st,                                                  // state to init
    buttons_print_st,                                               // state to print the buttons to the screen
    button_wait_button_touch_st,                                    // state to wait until the screen is pressed
    button_delay_st,                                                // state to pause for a split second so the state can find region touched and then print the buttons
    button_touch_st,                                                // state for when the screen is touched
    button_touch_release_st,                                        // state for when the screen is release from the touch
    button_end_st                                                   // state to stop the SM until enable flag has been turned off
} buttonHandlerCurrentState = button_int_st;                        // sets the first state to buttons_init_st

uint8_t buttonHandler_getRegionNumber(){                            // function to find the position of the touch on the screen
    int16_t x = 0;                                                  // sets the x coordinate back to 0
```

```
44      int16_t y = 0;                                          // sets the x
  coordinate back to 0
45      uint8_t z;                                              // initializes the
  z coordinate
46      display_getTouchedPoint(&x, &y, &z);                    // finds the new
  coordinates
47      return simonDisplay_computeRegionNumber(x, y);          // then returns
  them
48  }
49
50  void buttonHandler_enable(){                                // Turn on the
  state machine. Part of the interlock.
51      buttonEnableFlag = ENABLE_FLAG_ON;                      // sets the
  buttonsEnableFlag to on
52  }
53
54
55  void buttonHandler_disable(){                               // Turn off the
  state machine. Part of the interlock.
56      buttonEnableFlag = ENABLE_FLAG_OFF;                     // sets the
  buttonsEnableFlag to off
57  }
58
59  // The only thing this function does is return a boolean flag set by the buttonHandler
  state machine. To wit:
60  // Once enabled, the buttonHandler state-machine first waits for a touch. Once a touch
  is detected, the
61  // buttonHandler state-machine computes the region-number for the touched area. Next,
  the buttonHandler
62  // state-machine waits until the player removes their finger. At this point, the
  state-machine should
63  // set a bool flag that indicates the the player has removed their finger. Once the
  buttonHandler()
64  // state-machine is disabled, it should clear this flag.
65  // All buttonHandler_releasedDetected() does is return the value of this flag.
66  // As such, the body of this function should only contain a single line of code.
67  bool buttonHandler_releaseDetected(){                       // function to see
  if the screen has been touch
68      if(touchRelease){                                       // checks if the
  screen has been touched
69          touchRelease = FALSE;                               // initializes the
  touchRelease VAR back to 0
70          return TRUE;                                        // then return
  true
71      }
72      else{                                                   // if the screen
  was not touched
73          return FALSE;                                       // then return
  false
74      }
75  }
76
77  void buttonHandler_tick(){                                  // Standard tick
  function.
78      switch(buttonHandlerCurrentState){                      // set your state
  that your on to buttonHandlerCurrentState
79      case button_int_st:                                     // Moore state
  action for state #1
```

```
 80        break;                                                  // ends case
 81    case buttons_print_st:                                      // Moore state
   action for state #2
 82        simonDisplay_drawAllButtons();                          // prints all the
   buttons to the screen
 83        break;                                                  // ends case
 84    case button_wait_button_touch_st:                           // Moore state
   action for state #3
 85        break;                                                  // ends case
 86    case button_delay_st:                                       // Moore state
   action for state #4
 87        display_clearOldTouchData();                            // clear the old
   data so the SM can read in the new touch data
 88        delayCounter++;                                         // increase the
   delay counter
 89        break;                                                  // ends case
 90    case button_touch_st:                                       // Moore state
   action for state #5
 91        break;                                                  // ends case
 92    case button_touch_release_st:                               // Moore state
   action for state #6
 93        break;                                                  // ends case
 94    case button_end_st:                                         // Moore state
   action for state #7
 95        break;                                                  // ends case
 96    };
 97    switch(buttonHandlerCurrentState){
 98    case button_int_st:                                         // Mealy
   transition state action for state #1
 99        if(buttonEnableFlag && buttonInitFlag){                 // if the flag has
   been raised then enter the state and the buttons have been printed
100            display_clearOldTouchData();                        // clears the old
   data from the screen
101            buttonHandlerCurrentState = button_wait_button_touch_st;// move to next
   state
102        }
103        else if (buttonEnableFlag){                             //if the only the
   enableFlag is raised and the buttons have not been printed then go to the print
   buttons state
104            buttonHandlerCurrentState = buttons_print_st;       // move to next
   state
105        }
106        break;                                                  // ends case
107    case buttons_print_st:                                      // Mealy
   transition state action for state #2
108        buttonInitFlag = TRUE;                                  // set the
   buttonInitFlag to true because you entered the print state
109        buttonHandlerCurrentState = button_wait_button_touch_st;    // move to next
   state
110        break;                                                  // ends case
111    case button_wait_button_touch_st:                           // Mealy
   transition state action for state #3
112        if(display_isTouched()){                                // if the srceen
   is touched then move to the delay state to the touched region can be read
113            buttonHandlerCurrentState = button_delay_st;        // move to next
   state
114        }
115        break;                                                  // ends case
```

```
116     case button_delay_st:                                           // Mealy
    transition state action for state #4
117         if (delayCounter == TRUE){                                  // after the
    little wait to
118             display_clearOldTouchData();                            // clears the old
    data from the screen
119             delayCounter = FALSE;                                       // reset the
    delayCounter for the next time through the SM
120             simonDisplay_drawSquare(buttonHandler_getRegionNumber(), FALSE);// draw
    the square associated with the region that was touched
121             region = buttonHandler_getRegionNumber();               // read the new
    touch data in from the screen to a VAR for later
122             buttonHandlerCurrentState = button_touch_st;            // move to next
    state
123         }
124         break;                                                      // ends case
125     case button_touch_st:                                           // Mealy
    transition state action for state #5
126         if(!display_isTouched()){                                   //when the display
    is released enter the state
127             buttonHandlerCurrentState = button_touch_release_st;    // move to next
    state
128         }
129         break;                                                      // ends case
130     case button_touch_release_st:                                   // Mealy
    transition state action for state #6
131         simonDisplay_drawSquare(region, TRUE);                      //clear the boxes
132         simonDisplay_drawButton(region);                            //then draw the
    buttons
133         touchRelease = TRUE;                                        //set the release
    VAR to one
134         buttonHandlerCurrentState = button_end_st;                  // move to next
    state
135         break;                                                      // ends case
136     case button_end_st:                                             // Mealy
    transition state action for state #7
    //set the fifth state for moore
137         if(!buttonEnableFlag){                                      // wait until the
    the flag is lowered
138             buttonInitFlag = FALSE;                                 // set the
    initFlag off
139             buttonHandlerCurrentState = button_int_st;              // move to next
    state
140         }
141         break;                                                      // ends case
142     }
143 }
144
145 // buttonHandler_runTest(int16_t touchCount) runs the test until
146 // the user has touched the screen touchCount times. It indicates
147 // that a button was pushed by drawing a large square while
148 // the button is pressed and then erasing the large square and
149 // redrawing the button when the user releases their touch.
150
151 void buttonHandler_runTest(int16_t touchCountArg) {
152     int16_t touchCount = 0;                 // Keep track of the number of touches.
153     display_init();                         // Always have to init the display.
154     display_fillScreen(DISPLAY_BLACK);      // Clear the display.
```

```
155     // Draw all the buttons for the first time so the buttonHandler doesn't need to do
    this in an init state.
156     // Ultimately, simonControl will do this when the game first starts up.
157     simonDisplay_drawAllButtons();
158     buttonHandler_enable();
159     while (touchCount < touchCountArg) {    // Loop here while touchCount is less than
    the touchCountArg
160         buttonHandler_tick();               // Advance the state machine.
161         utils_msDelay(RUN_TEST_TICK_PERIOD_IN_MS);
162         if (buttonHandler_releaseDetected()) {  // If a release is detected, then the
    screen was touched.
163             touchCount++;                       // Keep track of the number of
    touches.
164             // Get the region number that was touched.
165             printf("button released: %d\n\r", buttonHandler_getRegionNumber());
166             // Interlocked behavior: handshake with the button handler (now disabled).
167             buttonHandler_disable();
168             utils_msDelay(RUN_TEST_TICK_PERIOD_IN_MS);
169             buttonHandler_tick();               // Advance the state machine.
170             buttonHandler_enable();             // Interlocked behavior: enable the
    buttonHandler.
171             utils_msDelay(RUN_TEST_TICK_PERIOD_IN_MS);
172             buttonHandler_tick();               // Advance the state machine.
173         }
174     }
175     display_fillScreen(DISPLAY_BLACK);        // clear the screen.
176     display_setTextSize(RUN_TEST_TEXT_SIZE);  // Set the text size.
177     display_setCursor(TEXT_MESSAGE_ORIGIN_X, TEXT_MESSAGE_ORIGIN_Y); // Move the
    cursor to a rough center point.
178     display_println(RUN_TEST_TERMINATION_MESSAGE1); // Print the termination message
    on two lines.
179     display_println(RUN_TEST_TERMINATION_MESSAGE2);
180 }
181
182
```

```
 2  * flashSequence.h
 7
 8
 9 #ifndef FLASHSEQUENCE_H_
10 #define FLASHSEQUENCE_H_
11
12 // Turns on the state machine. Part of the interlock.
13 void flashSequence_enable();
14
15 // Turns off the state machine. Part of the interlock.
16 void flashSequence_disable();
17
18 // Other state machines can call this to determine if this state machine is finished.
19 bool flashSequence_isComplete();
20
21 // Standard tick function.
22 void flashSequence_tick();
23
24 // Tests the flashSequence state machine.
25 void flashSequence_runTest();
26
27 #endif /* FLASHSEQUENCE_H_ */
28
```

```c
 * flashSequence.c

#include "flashSequence.h"
#include "simonDisplay.h"
#include "supportFiles/display.h"
#include "supportFiles/utils.h"
#include "globals.h"
#include <stdio.h>


//**********CODE_FROM_THE_PROFFESOR**********//
// This will set the sequence to a simple sequential pattern.
// It starts by flashing the first color, and then increments the index and flashes the first
// two colors and so forth. Along the way it prints info messages to the LCD screen.
#define TEST_SEQUENCE_LENGTH 8  // Just use a short test sequence.
uint8_t flashSequence_testSequence[TEST_SEQUENCE_LENGTH] = {
    SIMON_DISPLAY_REGION_0,                                     // sets region 0 to 0
    SIMON_DISPLAY_REGION_1,                                     // sets region 1 to 1
    SIMON_DISPLAY_REGION_2,                                     // sets region 2 to 2
    SIMON_DISPLAY_REGION_3,                                     // sets region 3 to 3
    SIMON_DISPLAY_REGION_3,                                     // sets region 3 to 3
    SIMON_DISPLAY_REGION_2,                                     // sets region 2 to 2
    SIMON_DISPLAY_REGION_1,                                     // sets region 1 to 1
    SIMON_DISPLAY_REGION_0};                                    // sets region 0 to 0
#define INCREMENTING_SEQUENCE_MESSAGE1 "Incrementing Sequence"  // Info message.
#define RUN_TEST_COMPLETE_MESSAGE "Runtest() Complete"          // Info message.
#define MESSAGE_TEXT_SIZE 2                                     // Make the text easy to see.
#define TWO_SECONDS_IN_MS 2000                                  // Two second delay.
#define TICK_PERIOD 75                                          // 200 millisecond delay.
#define TEXT_ORIGIN_X 0                                         // Text starts from far left and
#define TEXT_ORIGIN_Y (DISPLAY_HEIGHT/2)                        // middle of screen.

//**********MY_CODE**********//
#define FLASH_ENABLE_FLAG_ON 1                                  // Var for when the FLAG is o
#define FLASH_ENABLE_FLAG_OFF 0                                 // Var for when the enable flag is off
#define TRUE 1                                                  // sets true to 1
#define FALSE 0                                                 // sets false to 0
#define timeDelay 10                                            // time to delay between each box shown
uint8_t flashEnableFlag = 0;                                    // declares the enable flag
uint8_t isCompleteFlag = 0;                                     // declare the complete flag
uint8_t waitCounter = 0;                                        // declare the flash counter
uint8_t series = 0;                                             // Var for what number in the array

enum flashSequence_st_m{                                        // number the state
    init_st,                                                    // start state
    print_st,                                                   // print the button state
```

```
52      wait_st,                                                // wait for the button
  to be printed state
53      delete_st,                                              // delete the button
  state
54      end_st                                                  // end state
55  }flashCurrentState = init_st;                               // set the starting
  state to iniyt_st
56
57  void flashSequence_enable(){                                // Turns on the state
  machine. Part of the interlock.
58      flashEnableFlag = FLASH_ENABLE_FLAG_ON;                 //sets the flag to on
59  }
60
61  void flashSequence_disable(){                               // Turns off the state
  machine. Part of the interlock.
62      flashEnableFlag = FLASH_ENABLE_FLAG_OFF;                //sets the flag to off
63  }
64
65  bool flashSequence_isComplete(){                            // Other state
  machines can call this to determine if this state machine is finished.
66      return isCompleteFlag;
67  }
68
69  void flashSequence_tick(){                                  // Standard tick
  function.
70      switch(flashCurrentState){
71      case init_st:                                           // Moore state action
  for state #1
72          break;                                              // exit state
73      case print_st:                                          // Moore state action
  for state #2
74          simonDisplay_drawSquare(globals_getSequenceValue(series),FALSE);// print out
  the squares
75          break;                                              // exit state
76      case wait_st:                                           // Moore state action
  for state #3
77          waitCounter++;                                      // increment the
  waitCounter
78          break;                                              // exit state
79      case delete_st:                                         // Moore state action
  for state #4
80          simonDisplay_drawSquare(globals_getSequenceValue(series),TRUE); //delete the
  square that was printed
81          break;                                              // exit state
82      case end_st:                                            // Moore state action
  for state #5
83          isCompleteFlag = TRUE;                              // raise the flag
  saying that the square has been printed and erased
84          break;                                              // exit state
85      }
86      switch(flashCurrentState){
87      case init_st:                                           // Mealy transition
  state action for state #1
88          if(flashEnableFlag){
89              flashCurrentState = print_st;                   // transition to next
  state
90          }
91          break;                                              // exit state
```

```
 92    case print_st:                                          // Mealy transition
   state action for state #2
 93        flashCurrentState = wait_st;                        // transition to next
   state
 94        break;                                              // exit state
 95    case wait_st:                                           // Mealy transition
   state action for state #3
 96        if(waitCounter >= timeDelay){                       // wait till counter
   reaches the timerDelay
 97            waitCounter = FALSE;                            // after the state has
   sat for 10 ticks then reset the counter
 98            flashCurrentState = delete_st;                  // transition to next
   state
 99        }
100        break;                                              // exit state
101    case delete_st:                                         // Mealy transition
   state action for state #4
102        if(series >= globals_getSequenceIterationLength()){ // when the series
   reaches the max iteration length of the array
103            flashCurrentState = end_st;                     // transition to next
   state
104        }
105        else{
106            series++;                                       // increment the spot
   in the series
107            flashCurrentState = print_st;                   // transition to next
   state
108        }
109        break;                                              // exit state
110    case end_st:                                            // Mealy transition
   state action for state #5
111        if(!flashEnableFlag){                               // when the enable
   flag is lowered reset back to begining
112            series = FALSE;                                 // reset the flag
113            isCompleteFlag = FALSE;                         // reset the flag
114            flashCurrentState = init_st;                    // go back the the
   initial state
115        }
116        break;                                              // exit state
117    }
118}
119
120    // Print the incrementing sequence message.
121void flashSequence_printIncrementingMessage() {
122  display_fillScreen(DISPLAY_BLACK);  // Otherwise, tell the user that you are
   incrementing the sequence.
123  display_setCursor(TEXT_ORIGIN_X, TEXT_ORIGIN_Y);// Roughly centered.
124  display_println(INCREMENTING_SEQUENCE_MESSAGE1);// Print the message.
125  utils_msDelay(TWO_SECONDS_IN_MS);   // Hold on for 2 seconds.
126  display_fillScreen(DISPLAY_BLACK);  // Clear the screen.
127}
128
129// Run the test: flash the sequence, one square at a time
130// with helpful information messages.
131void flashSequence_runTest() {
132  display_init();                   // We are using the display.
133  display_fillScreen(DISPLAY_BLACK);    // Clear the display.
134  globals_setSequence(flashSequence_testSequence, TEST_SEQUENCE_LENGTH);    // Set the
```

```
   sequence.
135   flashSequence_enable();             // Enable the flashSequence state machine.
136   int16_t sequenceLength = 1;         // Start out with a sequence of length 1.
137   globals_setSequenceIterationLength(sequenceLength);   // Set the iteration length.
138   display_setTextSize(MESSAGE_TEXT_SIZE); // Use a standard text size.
139   while (1) {                           // Run forever unless you break.
140     flashSequence_tick();             // tick the state machine.
141     utils_msDelay(TICK_PERIOD);   // Provide a 1 ms delay.
142     if (flashSequence_isComplete()) {   // When you are done flashing the sequence.
143       flashSequence_disable();          // Interlock by first disabling the state
   machine.
144       flashSequence_tick();             // tick is necessary to advance the state.
145       utils_msDelay(TICK_PERIOD);       // don't really need this here, just for
   completeness.
146       flashSequence_enable();           // Finish the interlock by enabling the state
   machine.
147       utils_msDelay(TICK_PERIOD);       // Wait 1 ms for no good reason.
148       sequenceLength++;                 // Increment the length of the sequence.
149       if (sequenceLength > TEST_SEQUENCE_LENGTH)  // Stop if you have done the full
   sequence.
150         break;
151       // Tell the user that you are going to the next step in the pattern.
152       flashSequence_printIncrementingMessage();
153       globals_setSequenceIterationLength(sequenceLength);  // Set the length of the
   pattern.
154     }
155   }
156   // Let the user know that you are finished.
157   display_fillScreen(DISPLAY_BLACK);              // Blank the screen.
158   display_setCursor(TEXT_ORIGIN_X, TEXT_ORIGIN_Y);// Set the cursor position.
159   display_println(RUN_TEST_COMPLETE_MESSAGE);     // Print the message.
160 }
161
162
```

```
 2 * globals.h
 7
 8 #ifndef GLOBALS_H_
 9 #define GLOBALS_H_
10 #include <stdint.h>
11 #define GLOBALS_MAX_FLASH_SEQUENCE 1000                  // Make it big so you can use
   it for a splash screen.
12
13 // This is the length of the complete sequence at maximum length.
14 // You must copy the contents of the sequence[] array into the global variable that you
   maintain.
15 // Do not just grab the pointer as this will fail.
16 void globals_setSequence(const uint8_t sequence[], uint16_t length);
17
18 // This returns the value of the sequence at the index.
19 uint8_t globals_getSequenceValue(uint16_t index);
20
21 // Retrieve the sequence length.
22 uint16_t globals_getSequenceLength();
23
24 // This is the length of the sequence that you are currently working on.
25 void globals_setSequenceIterationLength(uint16_t length);
26
27 // This is the length of the sequence that you are currently working on,
28 // not the maximum length but the interim length as
29 // the use works through the pattern one color at a time.
30 uint16_t globals_getSequenceIterationLength();
31
32 #endif /* GLOBALS_H_ */
33
```

```c
 * globals.c

#include "globals.h"
#include "stdio.h"

// The length of the sequence.
// The static keyword means that globals_sequenceLength can only be accessed
// by functions contained in this file.
static uint16_t globals_sequenceLength = 0;  // The length of the sequence.
static uint16_t globals_sequenceIterationLength = 0;
uint16_t seriesArray[GLOBALS_MAX_FLASH_SEQUENCE] = {0};

// This is the length of the sequence that you are currently working on,
// not the maximum length but the interim length as
// the user works through the pattern one color at a time.
void globals_setSequenceIterationLength(uint16_t length) {
    globals_sequenceIterationLength = length - 1;
}


// This is the length of the complete sequence at maximum length.
// You must copy the contents of the sequence[] array into the global variable that you maintain.
// Do not just grab the pointer as this will fail.
void globals_setSequence(const uint8_t sequence[], uint16_t length){
    for (uint16_t i = 0; i <= length; i++){
        seriesArray[i] = sequence[i];
    }
    globals_sequenceLength = length;
}

// This returns the value of the sequence at the index.
uint8_t globals_getSequenceValue(uint16_t index){
    return seriesArray[index];
}

// Retrieve the sequence length.
uint16_t globals_getSequenceLength(){
    return globals_sequenceLength;
}

// This is the length of the sequence that you are currently working on,
// not the maximum length but the interim length as
// the use works through the pattern one color at a time.
uint16_t globals_getSequenceIterationLength(){
    return globals_sequenceIterationLength;
}


// You will need to implement the other functions.


```

```
 2  * verifySequence_runTest.h
 7
 8 #ifndef VERIFYSEQUENCE_H_
 9 #define VERIFYSEQUENCE_H_
10
11 // State machine will run when enabled.
12 void verifySequence_enable();
13
14 // This is part of the interlock. You disable the state-machine and then enable it
   again.
15 void verifySequence_disable();
16
17 // Used to detect if there has been a time-out error.
18 bool verifySequence_isTimeOutError();
19
20 // Used to detect if the user tapped the incorrect sequence.
21 bool verifySequence_isUserInputError();
22
23 // Used to detect if the verifySequence state machine has finished verifying.
24 bool verifySequence_isComplete();
25
26 // Standard tick function.
27 void verifySequence_tick();
28
29 // Standard runTest function.
30 void verifySequence_runTest();
31
32 #endif /* VERIFYSEQUENCE_H_ */
33
```

```c
 * verifySequence_runTest.c

#include "verifySequence_runTest.h"
#include "buttonHandler.h"
#include "simonDisplay.h"
#include "supportFiles/display.h"
#include "supportFiles/utils.h"
#include "globals.h"
#include <stdio.h>
#include <stdint.h>
#include "../Lab2_switch_button/buttons.h"

//**********CODE_FROM_PROF**********//
#define MESSAGE_X 0
//#define MESSAGE_Y (display_width()/4)
#define MESSAGE_Y (display_height()/2)
#define MESSAGE_TEXT_SIZE 2
//#define MESSAGE_STARTING_OVER
#define BUTTON_0 0  // Index for button 0
#define BUTTON_1 1  // Index for button 1
#define BUTTON_2 2  // Index for button 2
#define BUTTON_3 3  // Index for button 3

//**********MY_CODE**********//

#define ENABLE_FLAG_ON 1      //Var for when the FLAG is o
#define ENABLE_FLAG_OFF 0     // Var for when the enable flag is off
#define TIME_OUT_NUM 20
#define TRUE 1                        // sets true to 1
#define FALSE 0                       // sets false to 0
uint8_t verifyEnableFlag = 0;          //  declares the enable flag
uint8_t verifyIsCompleteFlag = 0;   // flag that shows if the SM is complete
uint8_t timeOutErrorFlag = 0;       // flag that show if the game has timed out
uint8_t userInputErrorFlag = 0;     // flag for when the user makes an error
uint8_t timeOut = 0;                // time out VAR
uint8_t indexInArray = 0;           // what index in the array are they



enum verifySequence_st_m{
    init_vs_st,         //state number 1
    enable_vs_st,         //state number 2
    wait_release_vs_st,         //state number 3
    region_vs_st,         //state number 4
    incrament_array_vs_st,  // state number 5
    finish_vs_st         //state number 6
} verifySequenceCurrentState = init_vs_st;

// State machine will run when enabled.
void verifySequence_enable(){
    verifyEnableFlag = ENABLE_FLAG_ON;     //sets the flag to on
}

// This is part of the interlock. You disable the state-machine and then enable it
  again.
void verifySequence_disable(){
    verifyEnableFlag = ENABLE_FLAG_OFF;     //sets the flag to off
```

```
63 }
64
65 // Used to detect if there has been a time-out error.
66 bool verifySequence_isTimeOutError(){
67     return timeOutErrorFlag;
68 }
69
70 // Used to detect if the user tapped the incorrect sequence.
71 bool verifySequence_isUserInputError(){
72     return userInputErrorFlag;
73 }
74
75 // Used to detect if the verifySequence state machine has finished verifying.
76 bool verifySequence_isComplete(){
77     return verifyIsCompleteFlag;
78 }
79
80 // Standard tick function.
81 void verifySequence_tick(){
82     switch(verifySequenceCurrentState){
83     case init_vs_st:         //state number 1 for moore
84         break;
85     case enable_vs_st:         //state number 2 for moore
86         buttonHandler_enable();         // enable the button handler
87         break;
88     case wait_release_vs_st:         //state number 3 for moore
89         timeOut++;                  // increment the timne out counter
90         break;
91     case region_vs_st:         //state number 4 for moore
92         buttonHandler_disable();            // disable the button handler
93         if(buttonHandler_getRegionNumber() != globals_getSequenceValue(indexInArray)
   ){ // if the area touch doesnt match the value the array has then user made an error
94             userInputErrorFlag = TRUE;
   // raise flag
95         }
96         break;
97     case incrament_array_vs_st:  // state number 5 for moore
98         break;
99     case finish_vs_st:         //state number 6 for moore
100        break;
101    }
102
103    switch(verifySequenceCurrentState){
104    case init_vs_st:         //state number 1 for mealy
105        if(verifyEnableFlag){
106            timeOut = FALSE;                                    // reset the VAR
107            indexInArray = FALSE;                                    // reset the VAR
108            verifyIsCompleteFlag = FALSE;                                    // reset the
   VAR
109            timeOutErrorFlag = FALSE;                                  // reset the VAR
110            userInputErrorFlag = FALSE;                                  // reset the
   VAR
111            verifySequenceCurrentState = enable_vs_st;
112        }
113        break;
114    case enable_vs_st:         //state number 2 for mealy
115        verifySequenceCurrentState = wait_release_vs_st;
116        break;
```

```
117     case wait_release_vs_st:         //state number 3 for mealy
118         if(timeOut == TIME_OUT_NUM){
119             verifyIsCompleteFlag = TRUE;                                    // raise the
    flag that the Verify squence is done
120             timeOutErrorFlag = TRUE;                                        // raise the
    flag that is took to long to press
121             simonDisplay_eraseAllButtons();                                 // erase all
    the buttons from the srceen
122             verifySequenceCurrentState = finish_vs_st;
123         }
124         else if(display_isTouched()){
125             timeOut = FALSE;                                                // if display
    is touch reset the timeout VAR
126         }
127         else if(buttonHandler_releaseDetected()){                           // if the
    screen was released then move to next state
128             verifySequenceCurrentState = region_vs_st;
129         }
130         break;
131     case region_vs_st:                                                       //state
    number 4 for mealy
132         verifySequenceCurrentState = increment_array_vs_st;                 //move to
    next state
133         break;
134     case increment_array_vs_st:                                         // state number 5
    for mealy
135         if (indexInArray == globals_getSequenceIterationLength()){
136             verifyIsCompleteFlag = TRUE;                                // set the
    complete flag to true
137             simonDisplay_eraseAllButtons();                             // erase all the
    buttons
138             verifySequenceCurrentState = finish_vs_st;
139         }
140         else if(userInputErrorFlag){
141             verifyIsCompleteFlag = TRUE;                                // set the
    complete flag to true
142             simonDisplay_eraseAllButtons();                             // erase all the
    buttons
143             verifySequenceCurrentState = finish_vs_st;
144         }
145         else {
146             indexInArray++;                                             // increment the
    nember in the array that the program is checking
147             verifySequenceCurrentState = enable_vs_st;
148         }
149         break;
150     case finish_vs_st:         //state number 6 for mealy
151         if(!verifyEnableFlag){                                          // wait til the
    enable flag is lowered to exit the state machine
152            verifySequenceCurrentState = init_vs_st;
153         }
154         break;
155     }
156
157 }
158
159 // Prints the instructions that the user should follow when
160 // testing the verifySequence state machine.
```

```
161 // Takes an argument that specifies the length of the sequence so that
162 // the instructions are tailored for the length of the sequence.
163 // This assumes a simple incrementing pattern so that it is simple to
164 // instruct the user.
165 void verifySequence_printInstructions(uint8_t length, bool startingOver) {
166     display_fillScreen(DISPLAY_BLACK);              // Clear the screen.
167     display_setTextSize(MESSAGE_TEXT_SIZE);     // Make it readable.
168     display_setCursor(MESSAGE_X, MESSAGE_Y);    // Rough center.
169     if (startingOver) {                                             // Print a message
    if you start over.
170         display_fillScreen(DISPLAY_BLACK);          // Clear the screen if starting
    over.
171         display_setTextColor(DISPLAY_WHITE);        // Print whit text.
172         display_println("Starting Over. ");         // Starting over message.
173     }
174     // Print messages are self-explanatory, no comments needed.
175     // These messages request that the user touch the buttons in a specific sequence.
176     display_println("Tap: ");
177     display_println();
178     switch (length) {
179     case 1:
180         display_println("red");
181         break;
182     case 2:
183         display_println("red, yellow ");
184         break;
185     case 3:
186         display_println("red, yellow, blue ");
187         break;
188     case 4:
189         display_println("red, yellow, blue, green ");
190         break;
191     default:
192         break;
193     }
194     display_println("in that order.");
195     display_println();
196     display_println("hold BTN0 to quit.");
197 }
198
199 // Just clears the screen and draws the four buttons used in Simon.
200 void verifySequence_drawButtons() {
201     display_fillScreen(DISPLAY_BLACK);  // Clear the screen.
202     simonDisplay_drawButton(BUTTON_0);  // Draw the four buttons.
203     simonDisplay_drawButton(BUTTON_1);
204     simonDisplay_drawButton(BUTTON_2);
205     simonDisplay_drawButton(BUTTON_3);
206 }
207
208 // This will set the sequence to a simple sequential pattern.
209 #define MAX_TEST_SEQUENCE_LENGTH 4  // the maximum length of the pattern
210 uint8_t verifySequence_testSequence[MAX_TEST_SEQUENCE_LENGTH] = {0, 1, 2, 3};  // A
    simple pattern.
211 #define MESSAGE_WAIT_MS 4000  // Display messages for this long.
212
213 // Increment the sequence length making sure to skip over 0.
214 // Used to change the sequence length during the test.
215 int16_t incrementSequenceLength(int16_t sequenceLength) {
```

```
216     int16_t value = (sequenceLength + 1) % (MAX_TEST_SEQUENCE_LENGTH+1);
217     if (value == 0) value++;
218     return value;
219 }
220 
221 // Used to select from a variety of informational messages.
222 enum verifySequence_infoMessage_t {
223     user_time_out_e,            // means that the user waited too long to tap a color.
224     user_wrong_sequence_e,      // means that the user tapped the wrong color.
225     user_correct_sequence_e,    // means that the user tapped the correct sequence.
226     user_quit_e                 // means that the user wants to quite.
227 };
228 
229 // Prints out informational messages based upon a message type (see above).
230 void verifySequence_printInfoMessage(verifySequence_infoMessage_t messageType) {
231     // Setup text color, position and clear the screen.
232     display_setTextColor(DISPLAY_WHITE);
233     display_setCursor(MESSAGE_X, MESSAGE_Y);
234     display_fillScreen(DISPLAY_BLACK);
235   switch(messageType) {
236   case user_time_out_e:  // Tell the user that they typed too slowly.
237         display_println("Error:");
238         display_println();
239         display_println("  User tapped sequence");
240         display_println("  too slowly.");
241     break;
242   case user_wrong_sequence_e:  // Tell the user that they tapped the wrong color.
243         display_println("Error: ");
244         display_println();
245         display_println("  User tapped the");
246         display_println("  wrong sequence.");
247     break;
248   case user_correct_sequence_e:  // Tell the user that they were correct.
249         display_println("User tapped");
250         display_println("the correct sequence.");
251     break;
252   case user_quit_e:             // Acknowledge that you are quitting the test.
253     display_println("quitting runTest().");
254     break;
255   default:
256     break;
257   }
258 }
259 
260 #define TICK_PERIOD_IN_MS 100
261 // Tests the verifySequence state machine.
262 // It prints instructions to the touch-screen. The user responds by tapping the
263 // correct colors to match the sequence.
264 // Users can test the error conditions by waiting too long to tap a color or
265 // by tapping an incorrect color.
266 void verifySequence_runTest() {
267     display_init();  // Always must do this.
268     buttons_init();  // Need to use the push-button package so user can quit.
269     int16_t sequenceLength = 1;  // Start out with a sequence length of 1.
270     verifySequence_printInstructions(sequenceLength, false);  // Tell the user what to
    do.
271     utils_msDelay(MESSAGE_WAIT_MS);  // Give them a few seconds to read the
    instructions.
```

```
272     verifySequence_drawButtons();    // Now, draw the buttons.
273     // Set the test sequence and it's length.
274     globals_setSequence(verifySequence_testSequence, MAX_TEST_SEQUENCE_LENGTH);
275     globals_setSequenceIterationLength(sequenceLength);
276     // Enable the verifySequence state machine.
277     verifySequence_enable();  // Everything is interlocked, so first enable the
    machine.
278     // Need to hold button until it quits as you might be stuck in a delay.
279     while (!(buttons_read() & BUTTONS_BTN0_MASK)) {
280         // verifySequence uses the buttonHandler state machine so you need to "tick"
    both of them.
281         verifySequence_tick();  // Advance the verifySequence state machine.
282         buttonHandler_tick();   // Advance the buttonHandler state machine.
283         utils_msDelay(TICK_PERIOD_IN_MS);       // Wait for a tick period.
284         // If the verifySequence state machine has finished, check the result,
285         // otherwise just keep ticking both machines.
286         if (verifySequence_isComplete()) {
287             if (verifySequence_isTimeOutError()) {                // Was the user too
    slow?
288                 verifySequence_printInfoMessage(user_time_out_e); // Yes, tell the
    user that they were too slow.
289             } else if (verifySequence_isUserInputError()) {       // Did the user tap
    the wrong color?
290                 verifySequence_printInfoMessage(user_wrong_sequence_e); // Yes, tell
    them so.
291             } else {
292                 verifySequence_printInfoMessage(user_correct_sequence_e); // User was
    correct if you get here.
293             }
294             utils_msDelay(MESSAGE_WAIT_MS);                           // Allow the
    user to read the message.
295             sequenceLength = incrementSequenceLength(sequenceLength);  // Increment
    the sequence.
296             globals_setSequenceIterationLength(sequenceLength);        // Set the
    length for the verifySequence state machine.
297             verifySequence_printInstructions(sequenceLength, true);    // Print the
    instructions.
298             utils_msDelay(MESSAGE_WAIT_MS);                           // Let the user
    read the instructions.
299             verifySequence_drawButtons();                             // Draw the
    buttons.
300             verifySequence_disable();                                 // Interlock:
    first step of handshake.
301             verifySequence_tick();                                    // Advance the
    verifySequence machine.
302             utils_msDelay(TICK_PERIOD_IN_MS);                         // Wait for
    tick period.
303             verifySequence_enable();                                  // Interlock:
    second step of handshake.
304             utils_msDelay(TICK_PERIOD_IN_MS);                         // Wait for
    tick period.
305         }
306     }
307     verifySequence_printInfoMessage(user_quit_e);  // Quitting, print out an
    informational message.
308 }
309
310
```

```
 2  * simonControl.h
 7
 8 #ifndef SIMONCONTROL_H_
 9 #define SIMONCONTROL_H_
10
11
12
13
14 #endif /* SIMONCONTROL_H_ */
15
16 #include <stdbool.h>
17 #include <stdint.h>
18
19 void simonControl_tick(); // include the tick function so main can access it
20
21
```

```c
 * simonControl.c


#include "verifySequence_runTest.h"
#include "flashSequence.h"
#include "buttonHandler.h"
#include "simonDisplay.h"
#include "simonControl.h"
#include "supportFiles/display.h"
#include "supportFiles/utils.h"
#include "globals.h"
#include <stdio.h>
#include <stdint.h>
#include "../Lab2_switch_button/buttons.h"



#define START_MSG "TOUCH TO START"                                  // text for the start message
#define SIMON_MSG "SIMON"                                           // text for the start message
#define YAY_MSG "Yay!"                                              // text for the sequence completed message
#define NEW_L_MSG "Touch for new level"                             // text for the new level message
#define LONG_MSG "Longest Sequence: "                               // text for the longest sequence

#define START_MSG_X 70                                              // coordinates for the start message x position
#define START_MSG_Y 140                                             // coordinates for the start message y position

#define SIMON_MSG_X 80                                              // coordinates for the simon message x position
#define SIMON_MSG_Y 90                                              // coordinates for the simon message y position

#define YAY_MSG_X 100                                               // coordinates for the yay message x position
#define YAY_MSG_Y 100                                               // coordinates for the yay message y position

#define NEW_L_MSG_X 50                                              // coordinates for the new level message x position
#define NEW_L_MSG_Y YAY_MSG_Y                                       // coordinates for the new level message y position

#define LONG_MSG_X 50                                               // coordinates for the longest sequence message x position
#define LONG_MSG_Y YAY_MSG_Y                                        // coordinates for the longest sequence message y position

#define NUM_MSG_X 280                                               // coordinates for the number on the longest sequence message x position
#define NUM_MSG_Y YAY_MSG_Y                                         // coordinates for the number on the longest sequence message y position

```

```
47 #define FALSE 0                                    // define False as
   0 or reset
48 #define TRUE 1                                     // define true as
   1 for raising flags
49
50 #define TEXT_SIZE_L 5                              // larger text
   size
51 #define TEXT_SIZE_S 2                              // small text size
52 #define WAIT_TIME 40                               // time to for
   messages to appear
53 #define WAIT_TIME_S 1                              // time to give an
   extra tick in a state
54 #define INIT_LEVEL 4                               // the starting
   level
55
56
57 uint8_t myArray[GLOBALS_MAX_FLASH_SEQUENCE];       // the array that
   holds the sequence
58 uint8_t currLevel = INIT_LEVEL;                    // the current
   level = the difficulty of the sequence
59 uint8_t currIter = 1;                              // always start
   the level with the first box appearing of the sequence
60 uint8_t randNum = 0;                               // number used to
   generate the random numbers
61
62 //**********COUNTERS**********/
63 uint8_t yayCounter = 0;                            // counter for yay
   message wait
64 uint8_t newLevelCounter = 0;                       // counter for new
   level message wait
65 uint8_t longestSeqCounter = 0;                     // counter for
   longest sequence message wait
66 uint8_t fdCounter = 0;                             // counter for
   flash enable wait
67 uint8_t vdCounter = 0;                             // counter for
   verify enable wait
68
69 //**********FLAGS**********/
70 uint8_t initFlag = 0;                              // flag for when
   the start screen needs to be printed
71 uint8_t nextFlag = 0;                              // flag to send
   the SM on to the next level with out reseting
72
73
74
75
76 enum control_st_m{
77     control_init_st,                               // the
   initializing state
78     control_touch_st,                              // the first touch
   state off of the start screen state
79     control_setIterLength_st,                      // the state that
   sets the iteration length for that level state
80     control_flash_enable_st,                       // the enable
   flash sequence state
81     control_flash_disable_st,                      // the disable
   flash sequence state
82     control_verify_enable,                         // the enable
```

```
   verify sequence state
 83    control_verify_disable,                                  // the disable
   verify sequence state
 84    control_yay_st,                                          // the print yay
   message state
 85    control_new_level_st,                                    // the print new
   level message state
 86    control_longest_sq_st                                    // the print
   longest sequence message state
 87}controlCurrent = control_init_st;                           // initializes the
   first state to the control_init_st
 88
 89void randNumGen(){                                           // function that
   generates a random sequence
 90    srand(randNum);                                          // seeds the
   number generator
 91    for(uint8_t i = 0; i < currLevel; i++){                  // for loop to
   fill the array with random nmbers
 92        myArray[i] = rand() % (INIT_LEVEL);                  // filling the
   array
 93    }
 94}
 95
 96void simonControl_tick(){                                    // Simon control
   state machine
 97    switch(controlCurrent){
 98    case control_init_st:                                    // Moore state
   action for state #1
 99        if(!initFlag){
100            display_setCursor(START_MSG_X, START_MSG_Y);      // set the cursor
   for the text
101            display_setTextColor(DISPLAY_WHITE);             // set the color for
   the text
102            display_setTextSize(TEXT_SIZE_S);                // set the text
   size to small
103            display_println(START_MSG);                      // print the start
   message
104            display_setCursor(SIMON_MSG_X, SIMON_MSG_Y);     // set the cursor
   for the text
105            display_setTextColor(DISPLAY_WHITE);             // set the color for
   the text
106            display_setTextSize(TEXT_SIZE_L);                // set the text
   size to large
107            display_println(SIMON_MSG);                      // print simon on
   the screen
108            if(!initFlag){
109                initFlag = TRUE;                             // raise the flag
   saying that it has been initialized
110            }
111        }
112        randNum++;                                           // get a new
   random number for the sequence generator
113        break;
114    case control_touch_st:                                     // Moore state
   action for state #2
115        break;
116    case control_setIterLength_st:
117        globals_setSequenceIterationLength(currIter);        // set the
```

```
   iteration length to the current number number in the the array sequence
118         break;
119     case control_flash_enable_st:                                   // Moore state
   action for state #3
120         fdCounter++;                                                // increment the
   counter so the state is active for at least one tick
121         flashSequence_enable();                                       // enable the
   flash sequence SM
122         break;
123     case control_flash_disable_st:                                  // Moore state
   action for state #4
124         fdCounter = FALSE;                                            // reset the
   counter
125         flashSequence_disable();                                      // disable the
   flash sequence SM
126         break;
127     case control_verify_enable:                                     // Moore state
   action for state #5
128         vdCounter++;                                                // increment the
   counter so the state is active for at least one tick
129         verifySequence_enable();                                     // enable the
   verify seq SM
130         break;
131     case control_verify_disable:                                    // Moore state
   action for state #6
132         vdCounter = FALSE;                                            // reset the
   counter
133         verifySequence_disable();                                     // disable the
   verify seq SM
134         break;
135     case control_yay_st:                                            // Moore state
   action for state #7
136         yayCounter++;                                                 // increment
   the counter
137         break;
138     case control_new_level_st:                                      // Moore state
   action for state #8
139         newLevelCounter++;                                                  //
   increment the counter
140         break;
141     case control_longest_sq_st:                                     // Moore state
   action for state #9
142         longestSeqCounter++;                                                //
   increment the counter
143         break;
144     }
145     switch(controlCurrent){
146     case control_init_st:                                           // Mealy
   transition state action for state #1
147         if(nextFlag){
148             randNumGen();                                           // fill the
   array with random numbers
149             globals_setSequence(myArray,currLevel);                 // set the
   sequence
150             controlCurrent = control_setIterLength_st;
151         }
152         else if(display_isTouched()){
153             randNumGen();
```

```
154             globals_setSequence(myArray,currLevel);
155             controlCurrent = control_touch_st;
156         }
157         break;
158     case control_touch_st:                                      // Mealy
    transition state action for state #2
159         if(!display_isTouched()){
160             // print out the start messages
161             display_setCursor(START_MSG_X, START_MSG_Y);            // set the cursor
    for the text
162             display_setTextColor(DISPLAY_BLACK);
    // erase the text
163             display_setTextSize(TEXT_SIZE_S);                       // set the text
    size to small
164             display_println(START_MSG);
165             display_setCursor(SIMON_MSG_X, SIMON_MSG_Y);            // set the cursor
    for the text
166             display_setTextColor(DISPLAY_BLACK);
    // erase the text
167             display_setTextSize(TEXT_SIZE_L);                       // set the text
    size to large
168             display_println(SIMON_MSG);
169             controlCurrent = control_setIterLength_st;
170         }
171         break;
172     case control_setIterLength_st:                              // Mealy
    transition state action for state #3
173         controlCurrent = control_flash_enable_st;
174         break;
175     case control_flash_enable_st:                               // Mealy
    transition state action for state #4
176         if(flashSequence_isComplete()&& (fdCounter > WAIT_TIME_S)){   //if the flash
    SM is done then move to next state
177             controlCurrent = control_flash_disable_st;
178         }
179         break;
180     case control_flash_disable_st:                              // Mealy
    transition state action for state #5
181         controlCurrent = control_verify_enable;
182         break;
183     case control_verify_enable:                                 // Mealy
    transition state action for state #6
184         if(verifySequence_isComplete() && (vdCounter > WAIT_TIME_S)){ //if the verify
    sequence is done move to next state
185             controlCurrent = control_verify_disable;
186         }
187         break;
188     case control_verify_disable:                                // Mealy
    transition state action for state #7
189         if(currIter == currLevel){                                // if the
    level and the iteration match then they won
190             // print yay msg
191             display_setCursor(YAY_MSG_X, YAY_MSG_Y);            // set the cursor for
    the text
192             display_setTextColor(DISPLAY_WHITE);                // set the color for
    the text
193             display_setTextSize(TEXT_SIZE_L);                       // set the text
    size to large
```

```
194             display_println(YAY_MSG);
195             controlCurrent = control_yay_st;
196         }
197         else if(verifySequence_isTimeOutError()|| verifySequence_isUserInputError()){
    //if there was an user error of a time out error go to end
198             //print longest sequence msg
199             display_setCursor(LONG_MSG_X, LONG_MSG_Y);            // set the cursor
    for the text
200             display_setTextColor(DISPLAY_WHITE);                  // set the color for
    the text
201             display_setTextSize(TEXT_SIZE_S);                       // set the text
    size to small
202             display_println(LONG_MSG);
203             display_setCursor(NUM_MSG_X, NUM_MSG_Y);            // set the cursor for
    the text
204             display_println(currLevel);
205             controlCurrent = control_longest_sq_st;
206         }
207         else{
208             currIter++;                                         // increment the
    iteration and go throught the flash and verify again
209             controlCurrent = control_setIterLength_st;
210         }
211         break;
212     case control_yay_st:                                          // Mealy
    transition state action for state #8
213         if(yayCounter >= WAIT_TIME){                            // time to wait for
    the print yay
214             // black yay
215             display_setCursor(YAY_MSG_X, YAY_MSG_Y);            // set the cursor for
    the text
216             display_setTextColor(DISPLAY_BLACK);
    // erase the text
217             display_setTextSize(TEXT_SIZE_L);                       // set the text
    size to large
218             display_println(YAY_MSG);
219             // print new level
220             display_setCursor(NEW_L_MSG_X, NEW_L_MSG_Y);            // set the cursor
    for the text
221             display_setTextColor(DISPLAY_WHITE);                  // set the color for
    the text
222             display_setTextSize(TEXT_SIZE_S);                       // set the text
    size to small
223             display_println(NEW_L_MSG);
224             controlCurrent = control_new_level_st;
225         }
226         break;
227     case control_new_level_st:                                    // Mealy
    transition state action for state #9
228         if(newLevelCounter >= WAIT_TIME){                       // time to wait
    for the new level msg
229             // black new level
230             display_setCursor(NEW_L_MSG_X, NEW_L_MSG_Y);            // set the cursor
    for the text
231             display_setTextColor(DISPLAY_BLACK);
    // erase the text
232             display_setTextSize(TEXT_SIZE_S);                       // set the text
    size to small
```

```
233             display_println(NEW_L_MSG);
234             // print longSQ
235             display_setCursor(LONG_MSG_X, LONG_MSG_Y);             // set the cursor
    for the text
236             display_setTextColor(DISPLAY_WHITE);                   // set the color for
    the text
237             display_setTextSize(TEXT_SIZE_S);                        // set the text
    size to small
238             display_println(LONG_MSG);
239             display_setCursor(NUM_MSG_X, NUM_MSG_Y);             // set the cursor for
    the text
240             display_println(currLevel);
241             controlCurrent = control_longest_sq_st;
242         }
243         if(display_isTouched()){                                     // if screen is
    touch go to harder level
244             // black new level
245             display_setCursor(NEW_L_MSG_X, NEW_L_MSG_Y);             // set the cursor
    for the text
246             display_setTextColor(DISPLAY_BLACK);
    // erase the text
247             display_setTextSize(TEXT_SIZE_S);                        // set the text
    size to small
248             display_println(NEW_L_MSG);
249             // Initialize var
250             currLevel++;                                                 // increment
    level
251             currIter = TRUE;                                             //reset VAR
252             nextFlag = TRUE;                                            //raise the
    next flag
253             yayCounter = FALSE;                                            //reset VAR
254             newLevelCounter = FALSE;                                          //reset
    VAR
255             longestSeqCounter = FALSE;
    //reset VAR
256             controlCurrent = control_init_st;
257         }
258         break;
259     case control_longest_sq_st:                                    // Mealy
    transition state action for state #10
260         if(longestSeqCounter >= WAIT_TIME){
261             // black longSQ
262             display_setCursor(LONG_MSG_X, LONG_MSG_Y);             // set the cursor
    for the text
263             display_setTextColor(DISPLAY_BLACK);
    // erase the text
264             display_setTextSize(TEXT_SIZE_S);                        // set the text
    size to small
265             display_println(LONG_MSG);
266             display_setCursor(NUM_MSG_X, NUM_MSG_Y);             // set the cursor for
    the text
267             display_println(currLevel);
268             // Initialize var
269             currLevel = INIT_LEVEL;                                 //reset VAR
270             currIter = TRUE;                                      //reset VAR
271             initFlag = FALSE;                                      //reset VAR
272             nextFlag = FALSE;                                      //reset VAR
273             yayCounter = FALSE;                                      //reset VAR
```

```
274             newLevelCounter = FALSE;                                //reset VAR
275             longestSeqCounter = FALSE;                               //reset VAR
276             controlCurrent = control_init_st;
277         }
278         break;
279     }
280 }
281
282
```