```c
//simonMain.c
#include <stdio.h>
#include "supportFiles/leds.h"
#include "supportFiles/globalTimer.h"
#include "supportFiles/interrupts.h"
#include <stdbool.h>
#include <stdint.h>
#include "clockControl.h"
#include "clockDisplay.h"
#include "supportFiles/display.h"
#include "xparameters.h"
#include "globals.h"
#include "simonDisplay.h"
#include "simonButtonHandler.h"
#include "simonFlashSequence.h"
#include "simonVerifySequence.h"
#include "SimonControl.h"
#include "buttons.h"

#define LARGE_PRIME_NUMBER 7919
#define TOTAL_SECONDS 60
#define TIMER_PERIOD 100E-3
#define FLAG_DOWN 0
#define TIMER_CLOCK_FREQUENCY (XPAR_CPU_CORTEXA9_0_CPU_CLK_FREQ_HZ / 2)
#define TIMER_LOAD_VALUE ((TIMER_PERIOD * TIMER_CLOCK_FREQUENCY) - 1.0)

int main()
{
        srand(LARGE_PRIME_NUMBER);
          // Initialize the GPIO LED driver and print out an error message if it fails (argument = true).
           // You need to init the LEDs so that LD4 can function as a heartbeat.
          leds_init(true);
          // Init all interrupts (but does not enable the interrupts at the devices).
          // Prints an error message if an internal failure occurs because the argument = true.
          interrupts_initAll(true);
          interrupts_setPrivateTimerLoadValue(TIMER_LOAD_VALUE);
          u32 privateTimerTicksPerSecond = interrupts_getPrivateTimerTicksPerSecond();
          printf("private timer ticks per second: %ld\n\r", privateTimerTicksPerSecond);
          // Allow the timer to generate interrupts.
          interrupts_enableTimerGlobalInts();
          // Initialization of the clock display is not time-dependent, do it outside of the state machine.
          //TicTacToeDisplay_init();
          display_init();
          display_fillScreen(DISPLAY_BLACK);
          buttons_init();
          // Keep track of your personal interrupt count. Want to make sure that you don't miss any interrupts.
           int32_t personalInterruptCount = 0;
          // Start the private ARM timer running.
          interrupts_startArmPrivateTimer();
          // Enable interrupts at the ARM.
          interrupts_enableArmInts();
          // interrupts_isrInvocationCount() returns the number of times that the timer ISR was invoked.
          // This value is maintained by the timer ISR. Compare this number with your own local
          // interrupt count to determine if you have missed any interrupts.
           while (interrupts_isrInvocationCount() < (TOTAL_SECONDS * privateTimerTicksPerSecond)) {
            if (interrupts_isrFlagGlobal) {  // This is a global flag that is set by the timer interrupt handler.
                  // Count ticks.
                  double duration0;
                personalInterruptCount++;
                 SimonControl_tick();
                verifySequence_tick();
                flashSequence_tick();
                simonbuttonHandler_tick();
                  interrupts_isrFlagGlobal = FLAG_DOWN;
            }
         }
         interrupts_disableArmInts();
         printf("isr invocation count: %ld\n\r", interrupts_isrInvocationCount());
         printf("internal interrupt count: %ld\n\r", personalInterruptCount);
         return 0;

}
```

```h
#ifndef BUTTONHANDLER_H_
#define BUTTONHANDLER_H_

// Get the simon region numbers. See the source code for the region numbering scheme.
uint8_t simonbuttonHandler_getRegionNumber();

// Turn on the state machine. Part of the interlock.
void simonbuttonHandler_enable();

// Turn off the state machine. Part of the interlock.
void simonbuttonHandler_disable();

// Other state machines can call this function to see if the user has stopped touching the pad.
bool simonbuttonHandler_releaseDetected();

// Standard tick function.
void simonbuttonHandler_tick();

// This tests the functionality of the buttonHandler state machine.
// buttonHandler_runTest(int16_t touchCount) runs the test until
// the user has touched the screen touchCount times. It indicates
// that a button was pushed by drawing a large square while
// the button is pressed and then erasing the large square and
// redrawing the button when the user releases their touch.
void simonbuttonHandler_runTest(int16_t touchCount);

#endif /* BUTTONHANDLER_H_ */
```

```c
//simonButtonHandler.c
#include "simonDisplay.h"
#include<stdio.h>
#include<stdint.h>
#include <stdio.h>
#include "supportFiles/leds.h"
#include "supportFiles/globalTimer.h"
#include "supportFiles/interrupts.h"
#include <stdbool.h>
#include <stdint.h>
#include "clockControl.h"
#include "clockDisplay.h"
#include "supportFiles/display.h"
#include "xparameters.h"
#include "supportFiles/utils.h"
#include "simonButtonHandler.h"

#define RUN_TEST_TERMINATION_MESSAGE1 "buttonHandler_runTest()"
#define RUN_TEST_TERMINATION_MESSAGE2 "terminated."
#define RUN_TEST_TEXT_SIZE 2

#define SIMON_BUTTON_WIDTH 60
#define SIMON_DISPLAY_GETREGION -1
#define SIMON_DISPLAY_REGION_0 0
#define SIMON_DISPLAY_REGION_1 1
#define SIMON_DISPLAY_REGION_2 2
#define SIMON_DISPLAY_REGION_3 3
#define TOUCHES_INIT
#define PIXEL_COORD_ZERO 0
#define ONE_MS

#define DISPLAY_WIDTH_HALVES display_width()/2
#define DISPLAY_HEIGHT_HALVES display_height()/2

#define AD_TIMER_EXPIRED_VALUE 1
#define AD_TIMER_INIT 0
#define RUN_TEST_INIT 0
#define BOOL_INIT false
#define ENABLED true
#define DISABLED false

uint16_t touches = TOUCHES_INIT;
uint8_t region = SIMON_DISPLAY_GETREGION;
uint16_t adTimer = AD_TIMER_INIT;

bool enable = false;
bool release = true;
bool erase = false;



//*************************************************************************************************
// simonbuttonHandler(getRegionNumber()
// determines which region of the screen is touched
// converts and returns the touched point into a simon Region Number
//*************************************************************************************************
uint8_t simonbuttonHandler_getRegionNumber()
{
      int16_t x;
      int16_t y;
      uint8_t z;
      display_getTouchedPoint(&x,&y,&z);

      if (x < PIXEL_COORD_ZERO || y < PIXEL_COORD_ZERO)
      {
          return SIMON_DISPLAY_GETREGION;
      }

        if (x < DISPLAY_WIDTH_HALVES)
        {
          if (y < DISPLAY_HEIGHT_HALVES)
          {
                  return SIMON_DISPLAY_REGION_0;
          }

          else
          {
                  return SIMON_DISPLAY_REGION_2;
          }
```

```
        }

        else
        {
            if (y < DISPLAY_HEIGHT_HALVES)
            {
                    return SIMON_DISPLAY_REGION_1;
            }
            else
            {
                    return SIMON_DISPLAY_REGION_3;
          }
         }
}


//*****************************************************************************************************
// simonbuttonHandler_enable()
// enables the button handler state machine
//*****************************************************************************************************
void simonbuttonHandler_enable()
{
      enable = true;
}


//*****************************************************************************************************
// simonbuttonHandler_disable()
// disables the button handler state machine
//*****************************************************************************************************
void simonbuttonHandler_disable()
{
      enable = false;
}


//*****************************************************************************************************
// simonbuttonHandler_releaseDetected()
// detects a release of the touch screen
//*****************************************************************************************************
bool simonbuttonHandler_releaseDetected()
{
      return release;
}

//*****************************************************************************************************
// simonbuttonHandler_runTest(int16_t touchCountArt)
// @param: int16_t touchCountArt
// Runs the button handler until touchCountArg touches are achieved
//*****************************************************************************************************
void simonbuttonHandler_runTest(int16_t touchCountArg)
{
  int16_t touchCount = RUN_TEST_INIT;             // Keep track of the number of touches.
  display_init();                      // Always have to init the display.
  display_fillScreen(DISPLAY_BLACK);  // Clear the display.
  simonDisplay_drawAllButtons();      // Draw the four buttons.
  simonbuttonHandler_enable();
  while (touchCount < touchCountArg) {  // Loop here while touchCount is less than the touchCountArg
    simonbuttonHandler_tick();               // Advance the state machine.
    utils_msDelay(ONE_MS);                     // Wait here for 1 ms.
    if (simonbuttonHandler_releaseDetected()) {  // If a release is detected, then the screen was touched.
      touchCount++;                           // Keep track of the number of touches.
      printf("button released: %d\n\r", simonbuttonHandler_getRegionNumber());  // Get the region number that was
touched.
      simonbuttonHandler_disable();             // Interlocked behavior: handshake with the button handler (now
disabled).
      utils_msDelay(ONE_MS);                     // wait 1 ms.
      simonbuttonHandler_tick();                 // Advance the state machine.
      simonbuttonHandler_enable();               // Interlocked behavior: enable the buttonHandler.
      utils_msDelay(ONE_MS);                     // wait 1 ms.
      simonbuttonHandler_tick();                 // Advance the state machine.
    }
  }
  display_fillScreen(DISPLAY_BLACK);                        // clear the screen.
  display_setTextSize(RUN_TEST_TEXT_SIZE);          // Set the text size.
  display_setCursor(PIXEL_COORD_ZERO, DISPLAY_HEIGHT_HALVES);                // Move the cursor to a rough center
point.
  display_println(RUN_TEST_TERMINATION_MESSAGE1);  // Print the termination message on two lines.
  display_println(RUN_TEST_TERMINATION_MESSAGE2);
```

```
}


enum simonButtonHandlerStates {sbh_init, sbh_draw, sbh_ad_timer, sbh_waiting_for_release, sbh_complete} sbh_state;


//*********************************************************************************************
// simonbuttonHandler_tick()
// button handler state machine
//*********************************************************************************************
void simonbuttonHandler_tick()
{

        switch(sbh_state)
        {
            //sbh_init
            //Initializes values and draws the buttons
                 case sbh_init:
                          release = BOOL_INIT;
                          adTimer = AD_TIMER_INIT;
                          region = SIMON_DISPLAY_GETREGION;

                          if(enable == ENABLED)
                          {
                                  simonDisplay_drawAllButtons();
                                  sbh_state = sbh_ad_timer;
                          }
                          break;


            //sbh_ad_timer
            //Debounces the touches
            //If the button handler is not enabled, transitions to completed
                 case sbh_ad_timer:
                     if(!enable)
                          {
                                  sbh_state = sbh_complete;
                          }

                          if(display_isTouched())
                          {
                                  adTimer++;
                          }

                          if(adTimer == AD_TIMER_EXPIRED_VALUE)
                          {
                                  sbh_state = sbh_draw;
                                  display_clearOldTouchData();
                                  adTimer = AD_TIMER_INIT;
                          }
                          break;


            //sbh_draw
            //draws the square in the region of the user's touch
                 case sbh_draw:
                          display_clearOldTouchData();
                          region = simonbuttonHandler_getRegionNumber();
                          simonDisplay_drawSquare(region, erase);
                          erase = ENABLED;
                          release = DISABLED;

                          sbh_state = sbh_waiting_for_release;
                          break;


            //sbh_waiting_for_release
            //waits for the user to release his/her touch
                 case sbh_waiting_for_release:
                          if(!display_isTouched())
                          {
                                  simonDisplay_drawSquare(region, ENABLED);
                                  simonDisplay_drawButton(region);
                                  erase = DISABLED;
                                  release = ENABLED;
                                  sbh_state = sbh_complete;
                          }
                          break;
```

```
            //sbh_complete
            // sets the button handler's completed bool to true
            // returns to init if the button handler is enabled
                 case sbh_complete:
                          erase = DISABLED;

                          if(enable)
                          {
                                   sbh_state = sbh_init;
                          }

                          break;


                 default:
                          sbh_state= sbh_init;
        }
}
```

```
#ifndef GLOBALS_H_
#define GLOBALS_H_

#define GLOBALS_MAX_FLASH_SEQUENCE 1000                 // Make it big so you can use it for a splash screen.

// 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);

// This returns the value of the sequence at the index.
uint8_t globals_getSequenceValue(uint16_t index);

// Retrieve the sequence length.
uint16_t globals_getSequenceLength();

// This is the length of the sequence that you are currently working on.
void globals_setSequenceIterationLength(uint16_t length);

// 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();

#endif /* GLOBALS_H_ */
```

```c
//globals.c
#include <stdio.h>
#include "supportFiles/leds.h"
#include "supportFiles/globalTimer.h"
#include "supportFiles/interrupts.h"
#include <stdbool.h>
#include <stdint.h>
#include "clockControl.h"
#include "clockDisplay.h"
#include "supportFiles/display.h"
#include "xparameters.h"
#include "globals.h"

#define STARTING_SIZE 100

uint16_t SequenceIterationLength;
uint16_t SequenceLength;
int8_t sequencelist[STARTING_SIZE];

//*****************************************************************************************************
//globals_setSequence(const uint8_t sequence[], uint16_t length)
//@param: const uint8_t sequence[], uint16_t length
// Sets the global array of sequences
// This is the length of the complete sequence at maximum length.
//*****************************************************************************************************
void globals_setSequence(const uint8_t sequence[], uint16_t length)
{
	for(int i = 0; i < length; i++)
	{
	  sequencelist[i] = sequence[i];
	}
}


//*****************************************************************************************************
// globals_getSequenceValue(uint16_t index)
// @param: uint16_t index
// This returns the value of the sequence at the index.
//*****************************************************************************************************
uint8_t globals_getSequenceValue(uint16_t index)
{
	return sequencelist[index];
}


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


//*****************************************************************************************************
// globals_setSequenceIterationLength(uint16_t length)
// @param: uint16_t length
// This is the length of the sequence that you are currently working on.
//*****************************************************************************************************
void globals_setSequenceIterationLength(uint16_t length)
{
	SequenceIterationLength = length;
}


//*****************************************************************************************************
// globals_getSequenceIterationLength
// This is the length of the sequence that you are currently working on (not the max length)
// as the player is working through the color sequence
//*****************************************************************************************************
uint16_t globals_getSequenceIterationLength()
{
	return SequenceIterationLength;
}
```

```
#ifndef SIMONDISPLAY_H_
#define SIMONDISPLAY_H_

#include <stdbool.h>
#include <stdint.h>

// Width, height of the simon "buttons"
#define SIMON_DISPLAY_BUTTON_WIDTH 60
#define SIMON_DISPLAY_BUTTON_HEIGHT 60

// Given coordinates from the touch pad, computes the region number.

// The entire touch-screen is divided into 4 rectangular regions, numbered 0 - 3.
// Each region will be drawn with a different color. Colored buttons remind
// the user which square is associated with each color. When you press
// a region, computeRegionNumber returns the region number that is used
// by the other routines.
/*
|----------|----------|
|          |          |
|    0     |     1    |
|  (RED)   | (YELLOW) |
------------------------
|          |          |
|     2    |    3     |
|  (BLUE)  |  (GREEN) |
------------------------
*/

// These are the definitions for the regions.
#define SIMON_DISPLAY_REGION_0 0
#define SIMON_DISPLAY_REGION_1 1
#define SIMON_DISPLAY_REGION_2 2
#define SIMON_DISPLAY_REGION_3 3

int8_t simonDisplay_computeRegionNumber(int16_t x, int16_t y);

void simonDisplay_drawStartScreen(bool erase);

// Draws a colored "button" that the user can touch.
// The colored button is centered in the region but does not fill the region.
void simonDisplay_drawButton(uint8_t regionNumber);

// Convenience function that draws all of the buttons.
void simonDisplay_drawAllButtons();

// Draws a bigger square that completely fills the region.
// If the erase argument is true, it draws the square as black background to "erase" it.
void simonDisplay_drawSquare(uint8_t regionNo, bool erase);

// Runs a brief demonstration of how buttons can be pressed and squares lit up to implement the user
// interface of the Simon game. The routine will continue to run until the touchCount has been reached, e.g.,
// the user has touched the pad touchCount times.

// I used a busy-wait delay (utils_msDelay) that uses a for-loop and just blocks until the time has passed.
// When you implement the game, you CANNOT use this function as we discussed in class. Implement the delay
// using the non-blocking state-machine approach discussed in class.
void simonDisplay_runTest(uint16_t touchCount);

void simonDisplay_eraseAllButtons();

void simonDisplay_eraseButton(uint8_t regionNumber);

#endif /* SIMONDISPLAY_H_ */
```

```c
//simonDisplay.c
#include "simonDisplay.h"
#include<stdio.h>
#include<stdint.h>
#include <stdio.h>
#include "supportFiles/leds.h"
#include "supportFiles/globalTimer.h"
#include "supportFiles/interrupts.h"
#include <stdbool.h>
#include <stdint.h>
#include "clockControl.h"
#include "clockDisplay.h"
#include "supportFiles/display.h"
#include "xparameters.h"
#include "supportFiles/utils.h"
#define SIMON_BUTTON_WIDTH 60
#define SIMON_BUTTON_WIDTH_HALVES SIMON_BUTTON_WIDTH_HALVES
#define SIMON_BUTTON_HEIGHT 60
#define SIMON_BUTTON_HEIGHT_HALVES SIMON_BUTTON_HEIGHT/2

#define ONE_FOURTH_HEIGHT display_height()*1/4
#define THREE_FOURTHS_HEIGHT display_height()*3/4
#define ONE_FOURTH_WIDTH display_width()*1/4
#define THREE_FOURTHS_WIDTH display_width()*3/4
#define DISPLAY_WIDTH_HALVES DISPLAY_WIDTH_HALVES
#define DISPLAY_HEIGHT_HALVES DISPLAY_HEIGHT_HALVES
#define PIXEL_COORD_ZERO 0
#define TOUCHES_INIT 0

#define TOUCH_PANEL_ANALOG_PROCESSING_DELAY_IN_MS 60 // in ms
#define MAX_STR 255
#define TEXT_SIZE_2 2
#define TEXT_SIZE_5 5

#define SIMON_REGION_0 0
#define SIMON_REGION_1 1
#define SIMON_REGION_2 2
#define SIMON_REGION_3 3
#define INVALID_REGION -1

#define TOUCH_PANEL_ANALOG_PROCESSING_DELAY_IN_MS 60



//*****************************************************************************************
// simonDisplay_computeRegionNumber(int16_t x, int16_t y)
// @param: int16_t x, int16_t y
// Computes the region of the screen that was touched based on the parameters x and y
//*****************************************************************************************
int8_t simonDisplay_computeRegionNumber(int16_t x, int16_t y)
{
    if(x < PIXEL_COORD_ZERO || y < PIXEL_COORD_ZERO)
    {
        return INVALID_REGION;
    }

    if(x < DISPLAY_WIDTH_HALVES)
    {
        if(y < DISPLAY_HEIGHT_HALVES)
        {
            return SIMON_REGION_0;
        }
        else
        {
            return SIMON_REGION_2;
        }
    }

    else
    {
        if (y < DISPLAY_HEIGHT_HALVES)
        {
            return SIMON_REGION_1;
        }
        else
        {
            return SIMON_REGION_3;
        }
    }
```

```
}



//*****************************************************************************************************
// simonDisplay_drawButton(uint8_t regionNumber)
// @param: uint8_t regionNumber
// Draws a small, colored button in each screen region
//*****************************************************************************************************
void simonDisplay_drawButton(uint8_t regionNumber)
{
    //if the region is invalid, do not draw
      if (regionNumber < PIXEL_COORD_ZERO)
    {
        return;
    }

      switch(regionNumber)
    {

        case 2:
            display_fillRect(ONE_WIDTH_FOURTH-(SIMON_BUTTON_WIDTH_HALVES), (THREE_FOURTHS_HEIGHT)-
(SIMON_BUTTON_WIDTH_HALVES), SIMON_BUTTON_WIDTH, SIMON_BUTTON_HEIGHT, DISPLAY_BLUE);
            break;

        case 0:
            display_fillRect((ONE_FOURTH_WIDTH)-(SIMON_BUTTON_WIDTH_HALVES), (ONE_FOURTH_HEIGHT)-
(SIMON_BUTTON_WIDTH_HALVES), SIMON_BUTTON_WIDTH, SIMON_BUTTON_HEIGHT, DISPLAY_RED);
            break;

        case 1:
            display_fillRect(THREE_FOURTHS_WIDTH-(SIMON_BUTTON_WIDTH_HALVES), ONE_FOURTH_HEIGHT-
(SIMON_BUTTON_WIDTH_HALVES), SIMON_BUTTON_WIDTH, SIMON_BUTTON_HEIGHT, DISPLAY_YELLOW);
            break;

        case 3:
            display_fillRect(THREE_FOURTHS_WIDTH-(SIMON_BUTTON_WIDTH_HALVES), THREE_FOURTHS_HEIGHT-
(SIMON_BUTTON_WIDTH_HALVES), SIMON_BUTTON_WIDTH, SIMON_BUTTON_HEIGHT, DISPLAY_GREEN);
            break;
      }
}


//*****************************************************************************************************
// simonDisplay_drawAllButtons()
// draws a button in each of the four regions
//*****************************************************************************************************
void simonDisplay_drawAllButtons()
{
      simonDisplay_drawButton(SIMON_DISPLAY_REGION_0);
      simonDisplay_drawButton(SIMON_DISPLAY_REGION_1);
      simonDisplay_drawButton(SIMON_DISPLAY_REGION_2);
      simonDisplay_drawButton(SIMON_DISPLAY_REGION_3);
}


//*****************************************************************************************************
// simonDisplay_eraseButton(uint8_t regionNumber)
// @param: uint8_t regionNumber
// erases all four of the buttons
//*****************************************************************************************************
void simonDisplay_eraseButton(uint8_t regionNumber)
{
      // Do nothing if the region number is negative (illegal region, off LCD screen).
      if (regionNumber < PIXEL_COORD_ZERO)
    {
        return;
    }

      switch(regionNumber)
    {

        case 2:
            display_fillRect(ONE_FOURTH_WIDTH-(SIMON_BUTTON_WIDTH_HALVES), (THREE_FOURTHS_HEIGHT)-
(SIMON_BUTTON_WIDTH_HALVES), SIMON_BUTTON_WIDTH, SIMON_BUTTON_HEIGHT, DISPLAY_BLACK);
            break;

        case 0:
            display_fillRect((ONE_FOURTH_WIDTH)-(SIMON_BUTTON_WIDTH_HALVES), (ONE_FOURTH_HEIGHT)-
```

```
(SIMON_BUTTON_WIDTH_HALVES), SIMON_BUTTON_WIDTH, SIMON_BUTTON_HEIGHT, DISPLAY_BLACK);
            break;

        case 1:
            display_fillRect(3*ONE_FOURTH_WIDTH-(SIMON_BUTTON_WIDTH_HALVES), ONE_FOURTH_HEIGHT-
(SIMON_BUTTON_WIDTH_HALVES), SIMON_BUTTON_WIDTH, SIMON_BUTTON_HEIGHT, DISPLAY_BLACK);
            break;

        case 3:
            display_fillRect(3*ONE_FOURTH_WIDTH-(SIMON_BUTTON_WIDTH_HALVES), THREE_FOURTHS_HEIGHT-
(SIMON_BUTTON_WIDTH_HALVES), SIMON_BUTTON_WIDTH, SIMON_BUTTON_HEIGHT, DISPLAY_BLACK);
            break;
      }
}


//*****************************************************************************************
// simonDisplay_eraseAllButtons()
// erases all of the buttons
//*****************************************************************************************
void simonDisplay_eraseAllButtons()
{
      simonDisplay_eraseButton(SIMON_DISPLAY_REGION_0);
      simonDisplay_eraseButton(SIMON_DISPLAY_REGION_1);
      simonDisplay_eraseButton(SIMON_DISPLAY_REGION_2);
      simonDisplay_eraseButton(SIMON_DISPLAY_REGION_3);
}


//*****************************************************************************************
// simonDisplay_drawSquare(uint8_t regionNo, bool erase)
// @param: uint8_t regionNo, bool erase
// draws a full square in the corresponding region
//*****************************************************************************************
void simonDisplay_drawSquare(uint8_t regionNo, bool erase)
{
    // Do nothing if the region number is illegal (off LCD screen).
    if (regionNo < PIXEL_COORD_ZERO)
    {
        return;
    }

    switch(regionNo)
    {
        case 0:

            if (!erase)
            {
                display_fillRect(PIXEL_COORD_ZERO, PIXEL_COORD_ZERO, DISPLAY_WIDTH_HALVES, DISPLAY_HEIGHT_HALVES,
DISPLAY_RED);
            }
            else
            {
                display_fillRect(PIXEL_COORD_ZERO, PIXEL_COORD_ZERO, DISPLAY_WIDTH_HALVES, DISPLAY_HEIGHT_HALVES,
DISPLAY_BLACK);
            }
            break;

        case 1:
            if (!erase)
            {
                display_fillRect(DISPLAY_WIDTH_HALVES, PIXEL_COORD_ZERO, DISPLAY_WIDTH_HALVES, DISPLAY_HEIGHT_HALVES,
DISPLAY_YELLOW);
            }
            else
            {
                display_fillRect(DISPLAY_WIDTH_HALVES, PIXEL_COORD_ZERO, DISPLAY_WIDTH_HALVES, DISPLAY_HEIGHT_HALVES,
DISPLAY_BLACK);
            }
            break;

        case 2:
            if (!erase)
            {
                display_fillRect(PIXEL_COORD_ZERO, DISPLAY_HEIGHT_HALVES, DISPLAY_WIDTH_HALVES,
DISPLAY_HEIGHT_HALVES, DISPLAY_BLUE);
            }
            else
            {
```

```
                display_fillRect(PIXEL_COORD_ZERO, DISPLAY_HEIGHT_HALVES, DISPLAY_WIDTH_HALVES,
DISPLAY_HEIGHT_HALVES, DISPLAY_BLACK);
            }
            break;

        case 3:
            if (!erase)
            {
                display_fillRect(DISPLAY_WIDTH_HALVES, DISPLAY_HEIGHT_HALVES, DISPLAY_WIDTH_HALVES,
DISPLAY_HEIGHT_HALVES, DISPLAY_GREEN);
            }
            else
            {
                display_fillRect(DISPLAY_WIDTH_HALVES, DISPLAY_HEIGHT_HALVES, DISPLAY_WIDTH_HALVES,
DISPLAY_HEIGHT_HALVES, DISPLAY_BLACK);
            }
            break;
   }
}


//*****************************************************************************************************
// simonDisplay_runTest(uint16_t touchCount)
// @param: uint16_t touchCount
// Runs a brief demonstration of how buttons can be pressed
//The routine will continue to run until the touchCount has been reached, e.g.,
// the user has touched the pad touchCount times.
//*****************************************************************************************************
void simonDisplay_runTest(uint16_t touchCount) {
  display_init();  // Always initialize the display.
  char str[MAX_STR];   // Enough for some simple printing.
  uint8_t regionNumber;
  uint16_t touches = TOUCHES_INIT;
  // Write an informational message and wait for the user to touch the LCD.
  display_fillScreen(DISPLAY_BLACK);        // clear the screen.
  display_setCursor(PIXEL_COORD_ZERO, DISPLAY_HEIGHT_HALVES); //
  display_setTextSize(TEXT_SIZE_2);
  display_setTextColor(DISPLAY_RED, DISPLAY_BLACK);
  sprintf(str, "Touch and release to start the Simon demo.");
  display_println(str);
  display_println();
  sprintf(str, "Demo will terminate after %d touches.", touchCount);
  display_println(str);
  while (!display_isTouched());       // Wait here until the screen is touched.
  while (display_isTouched());        // Now wait until the touch is released.
  display_fillScreen(DISPLAY_BLACK);  // Clear the screen.
  simonDisplay_drawAllButtons();      // Draw all of the buttons.
  bool touched = false;         // Keep track of when the pad is touched.
  int16_t x, y;                 // Use these to keep track of coordinates.
  uint8_t z;                            // This is the relative touch pressure.
  while (touches < touchCount) {  // Run the loop according to the number of touches passed in.
    if (!display_isTouched() && touched) {         // user has stopped touching the pad.
      simonDisplay_drawSquare(regionNumber, true); // Erase the square.
      simonDisplay_drawButton(regionNumber);          // DISPLAY_REDraw the button.
      touched = false;
                        // Released the touch, set touched to false.
    } else if (display_isTouched() && !touched) {   // User started touching the pad.
      touched = true;                               // Just touched the pad, set touched = true.
      touches++;
                                      // Keep track of the number of touches.
      display_clearOldTouchData();  // Get rid of data from previous touches.
      // Must wait this many milliseconds for the chip to do analog processing.
      utils_msDelay(TOUCH_PANEL_ANALOG_PROCESSING_DELAY_IN_MS);
      display_getTouchedPoint(&x, &y, &z);                 // After the wait, get the touched point.
      regionNumber = simonDisplay_computeRegionNumber(x, y);// Compute the region number.
      simonDisplay_drawSquare(regionNumber, false);    // Draw the square (erase = false).
    }
  }
  // Done with the demo, write an informational message to the user.
  display_fillScreen(DISPLAY_BLACK);        // clear the screen.
  display_setCursor(PIXEL_COORD_ZERO, DISPLAY_HEIGHT_HALVES); // Place the cursor in the middle of the screen.
  display_setTextSize(TEXT_SIZE_2);                   // Make it readable.
  display_setTextColor(DISPLAY_RED, DISPLAY_BLACK);  // red is foreground color, black is background color.
  sprintf(str, "Simon demo terminated");    // Format a string using sprintf.
  display_println(str);                     // Print it to the LCD.
  sprintf(str, "after %d touches.", touchCount);  // Format the rest of the string.
  display_println(str);  // Print it to the LCD.
}
```

```h
#ifndef simonFlashSequence_H_
#define simonFlashSequence_H_

// Turns on the state machine. Part of the interlock.
void flashSequence_enable();

// Turns off the state machine. Part of the interlock.
void flashSequence_disable();

// Other state machines can call this to determine if this state machine is finished.
bool flashSequence_completed();

// Standard tick function.
void flashSequence_tick();

// Tests the flashSequence state machine.
void flashSequence_runTest();

#endif
```

```c
//simonFlashSequence.c
#include "simonFlashSequence.h"
#include "simonDisplay.h"
#include<stdio.h>
#include<stdint.h>
#include <stdio.h>
#include "supportFiles/leds.h"
#include "supportFiles/globalTimer.h"
#include "supportFiles/interrupts.h"
#include <stdbool.h>
#include <stdint.h>
#include "clockControl.h"
#include "clockDisplay.h"
#include "supportFiles/display.h"
#include "xparameters.h"
#include "supportFiles/utils.h"
#include "globals.h"
#include "simonVerifySequence.h"

#define INITFLASHVALUE 10
#define INIT_ZERO 0
#define TWO_SECONDS 2000
#define PIXEL_COORD_ZERO 0
#define ONE 1
#define DISPLAY_HEIGHT_HALVES display_height()/2
#define TEST_SEQUENCE_LENGTH 8    // Just use a short test sequence.
uint8_t flashSequence_testSequence[TEST_SEQUENCE_LENGTH] = {SIMON_DISPLAY_REGION_0,
                                                            SIMON_DISPLAY_REGION_1,
                                                            SIMON_DISPLAY_REGION_2,
                                                            SIMON_DISPLAY_REGION_3,
                                                            SIMON_DISPLAY_REGION_3,
                                                            SIMON_DISPLAY_REGION_2,
                                                            SIMON_DISPLAY_REGION_1,
                                                            SIMON_DISPLAY_REGION_0};

#define INCREMENTING_SEQUENCE_MESSAGE1 "Incrementing Sequence"
#define RUN_TEST_COMPLETE_MESSAGE "Runtest() Complete"
#define MESSAGE_TEXT_SIZE 2
#define FLASH_HOLD_EXP 3
#define F_ENABLED true
#define F_DISABLED false

bool flash_enable = false;
bool flash_completed = false;
bool flash_erase = false;
uint8_t flash_string = INITFLASHVALUE;
uint8_t init_value = INIT_ZERO;
uint16_t flash_sequence_hold = INIT_ZERO;
uint8_t current_sequence_length;
```

```
//*************************************************************************************************
// flashSequence_enable()
// Turns on the state machine. Part of the interlock.
//*************************************************************************************************
void flashSequence_enable()
{
      flash_completed = false;
      flash_enable = true;
}


//*************************************************************************************************
//flashSequence_disable()
// Turns off the state machine. Part of the interlock.
//*************************************************************************************************
void flashSequence_disable()
{
      flash_enable = false;
}


//*************************************************************************************************
// flashSequence_completed()
// Other state machines can call this to determine if this state machine is finished.
//*************************************************************************************************
bool flashSequence_completed()
{
      return flash_completed;
}

enum FlashSequenceStates{flash_init, flash_getregion, flash_hold, flash_sequence, flash_complete, flash_wait}
flash_state;


//*************************************************************************************************
// flashSequence_tick()
// flash sequence state machine
//*************************************************************************************************
void flashSequence_tick()
{
      switch(flash_state)
      {

        //flash_init
        // sets the current sequence length
        // resets global bools
         case flash_init:
                  current_sequence_length = globals_getSequenceIterationLength();
                  flash_completed = F_DISABLED;
                  if(flash_enable)
                  {
                 flash_state = flash_getregion;
                  }
                  break;


        //flash_getregion
        //draws a square depending on the position of the sequence
         case flash_getregion:
                  simonDisplay_drawSquare(globals_getSequenceValue(init_value), flash_erase);
                  flash_erase = F_ENABLED;

                  flash_state = flash_hold;
            break;


        //flash_hold
        //holds full square on the screen for FLASH_HOLD_EXP time
         case flash_hold:
                  flash_sequence_hold++;
                  if(flash_sequence_hold == FLASH_HOLD_EXP)
                  {
                 init_value++;
                           simonDisplay_drawSquare(globals_getSequenceValue(init_value),flash_erase);
                           flash_erase = F_DISABLED;
                 flash_sequence_hold = INIT_ZERO;
```

```
                        flash_state = flash_sequence;
                }
                break;


        //flash_sequence
        //determines if the sequence length has been reached
         case flash_sequence:
            if(init_value < current_sequence_length)
            {
                flash_state = flash_getregion;
            }

            if(init_value >= current_sequence_length)
            {
                flash_state = flash_complete;
            }
            break;


        //flash_complete
        //reinitialies values and adjusts the completed bool
         case flash_complete:
                 init_value = INIT_ZERO;
                 flash_completed = F_ENABLED;

            if(!flash_enable)
                 {
                        flash_state = flash_init;
                 }

                 break;


         default:
                 flash_state = flash_init;
                 break;

      }
}


//*****************************************************************************************
// flashSequence_printIncrementingMessage()
// Print the incrementing sequence message.
//*****************************************************************************************
void flashSequence_printIncrementingMessage()
{
    display_fillScreen(DISPLAY_BLACK);// Otherwise, tell the user that you are incrementing the sequence.
    display_setCursor(PIXEL_COORD_ZERO, DISPLAY_HEIGHT_HALVES);         // Roughly centered.
    display_println(INCREMENTING_SEQUENCE_MESSAGE1);  // Print the message.
    utils_msDelay(TWO_SECONDS);                              // Hold on for 2 seconds.
    display_fillScreen(DISPLAY_BLACK);                // Clear the screen.
}

//*****************************************************************************************
// flashSequence_runTest()
// Runs the flash sequence state machine test using pre-determined sequences and lengths
//*****************************************************************************************
void flashSequence_runTest()
{
    display_init();       // We are using the display.
    display_fillScreen(DISPLAY_BLACK);     // Clear the display.
    globals_setSequence(flashSequence_testSequence, TEST_SEQUENCE_LENGTH);   // Set the sequence.
    flashSequence_enable();                                 // Enable the flashSequence state machine.
    int16_t sequenceLength = ONE;                          // Start out with a sequence of length 1.
    globals_setSequenceIterationLength(sequenceLength);     // Set the iteration length.
    display_setTextSize(MESSAGE_TEXT_SIZE);                 // Use a standard text size.
    while (1)
    {                   // Run forever unless you break.
        flashSequence_tick();     // tick the state machine.
        utils_msDelay(ONE_MS);    // Provide a 1 ms delay.

        if(flashSequence_completed())// When you are done flashing the sequence.
        {
            flashSequence_disable();  // Interlock by first disabling the state machine.
        }
```

```
        flashSequence_tick();     // tick is necessary to advance the state.
        utils_msDelay(ONE_MS);            // don't really need this here, just for completeness.
        flashSequence_enable();   // Finish the interlock by enabling the state machine.
        utils_msDelay(ONE_MS);    // Wait 1 ms for no good reason.
        sequenceLength++; // Increment the length of the sequence.

        if (sequenceLength > TEST_SEQUENCE_LENGTH) // Stop if you have done the full sequence.
        {
            break;
        }

        flashSequence_printIncrementingMessage(); // Tell the user that you are going to the next step
        globals_setSequenceIterationLength(sequenceLength);// Set the length of the pattern.
    }
  }
    // Let the user know that you are finished.
    display_fillScreen(DISPLAY_BLACK);
    display_setCursor(PIXEL_COORD_ZERO, DISPLAY_HEIGHT_HALVES);
    display_println(RUN_TEST_COMPLETE_MESSAGE);
}
```

```
#ifndef VERIFYSEQUENCE_H_
#define VERIFYSEQUENCE_H_

enum verifySequence_infoMessage_t
{
  user_time_out_e,            // means that the user waited too long to tap a color.
  user_wrong_sequence_e,      // means that the user tapped the wrong color.
  user_correct_sequence_e,    // means that the user tapped the correct sequence.
  user_quit_e                 // means that the user wants to quite.
};


// State machine will run when enabled.
void verifySequence_enable();

// This is part of the interlock. You disable the state-machine and then enable it again.
void verifySequence_disable();

// Used to detect if there has been a time-out error.
bool verifySequence_isTimeOutError();

// Used to detect if the user tapped the incorrect sequence.
bool verifySequence_isUserInputError();

// Used to detect if the verifySequence state machine has finished verifying.
bool verifySequence_isComplete();

// Standard tick function.
void verifySequence_tick();

// Standard runTest function.
void verifySequence_runTest();

void verifySequence_printInfoMessage(verifySequence_infoMessage_t messageType);

void verifySequence_eraseInfoMessage(verifySequence_infoMessage_t messageType);

#endif /* VERIFYSEQUENCE_H_ */
```

```c
//simonVerifySequence.c
#include "simonVerifySequence.h"
#include <stdio.h>
#include "supportFiles/leds.h"
#include "supportFiles/globalTimer.h"
#include "supportFiles/interrupts.h"
#include <stdbool.h>
#include <stdint.h>
#include "clockControl.h"
#include "clockDisplay.h"
#include "supportFiles/display.h"
#include "xparameters.h"
#include "globals.h"
#include "simonDisplay.h"
#include "simonButtonHandler.h"
#include "buttons.h"
#include "supportFiles/utils.h"

#define MESSAGE_X 0
#define MESSAGE_Y (display_width()/4)
#define MESSAGE_TEXT_SIZE 2
#define MESSAGE_STARTING_OVER

#define TIMER_EXPIRED_VALUE 10
#define MESSAGE_WAIT_MS 4000  // Display messages for this long.
#define MAX_TEST_SEQUENCE_LENGTH 4
#define INIT_ZERO 0
#define INVALID_REGION -1
#define BTN0 1
#define V_DISABLED false
#define V_ENABLED true
#define ONE 1
#define ONE_MS 1
#define ZERO 0
uint8_t verifySequence_testSequence[MAX_TEST_SEQUENCE_LENGTH] = {0, 1, 2, 3};

bool verify_enable = false;
bool verify_completed = false;
bool verify_isTimeOutError =false;
bool verify_isUserInputError= false;
uint16_t timer_waiting = INIT_ZERO;
uint8_t simonRegion = INVALID_REGION;
uint16_t verifyValue = INIT_ZERO;
uint16_t verify_current_sequence_length;


//*****************************************************************************************************
// verifySequence_enable()
// enables the verify sequence state machine
//*****************************************************************************************************
void verifySequence_enable()
{
	verify_enable = true;
}


//*****************************************************************************************************
// verifySequence_disable()
// disables the verify sequence state machine, erases the simon buttons
// disables the button handler state machine
//*****************************************************************************************************
void verifySequence_disable()
{
	simonbuttonHandler_disable();
	simonDisplay_eraseAllButtons();
	verify_enable = false;
}

//*****************************************************************************************************
// verifySequence_isTimeOutError
// Used to detect if there has been a time-out error.
//*****************************************************************************************************
bool verifySequence_isTimeOutError()
{
	return verify_isTimeOutError;
}

//*****************************************************************************************************
// verifySequence_isUserInputError()
```

```
// Used to detect if the user tapped the incorrect sequence.
//*********************************************************************************************
bool verifySequence_isUserInputError()
{
      return verify_isUserInputError;
}

//*********************************************************************************************
// verifySequence_isComplete()
// Used to detect if the verifySequence state machine has finished verifying.
//*********************************************************************************************
bool verifySequence_isComplete()
{
      return verify_completed;
}

enum VerifyStates {verify_init, verify_waiting, verify_wait_release, verify_validate, verify_complete, final_state}
verify_state;


//*********************************************************************************************
// verifySequence_tick()
// verify sequence state machine
//*********************************************************************************************
void verifySequence_tick()
{
      switch(verify_state)
      {

    //verify_init
    //Resets all error cases and sets all timers to zero
    //Gets the current sequence length
    //enables the button handler
      case verify_init:
        verify_isTimeOutError = V_DISABLED;
        verify_isUserInputError = V_DISABLED;
        verify_completed = V_DISABLED;
        timer_waiting = INIT_ZERO;
        verifyValue = INIT_ZERO;

        verify_current_sequence_length = globals_getSequenceIterationLength();

        if(verify_enable)
        {
                simonbuttonHandler_enable();
                verify_state = verify_waiting;
        }
        break;


    //verify_waiting
    //provides the user with time to touch the screen
    //if the user does not touch before TIMER_EXPIRED_VALUE, verify_isTimeOutError occurs
    //disables the button handler after waiting is over
      case verify_waiting:
        timer_waiting++;
        if(timer_waiting == TIMER_EXPIRED_VALUE)
        {
            verify_isTimeOutError = V_ENABLED;
            simonbuttonHandler_disable();

            verify_state = verify_complete;
        }

        if(display_isTouched())
        {
            if(!simonbuttonHandler_releaseDetected() && timer_waiting < TIMER_EXPIRED_VALUE)
            {
                verify_state = verify_wait_release;
            }
        }
        break;


    //verify_wait_release
    //waits for the user to release the touch screen and disables the button handler
      case verify_wait_release:

        if(simonbuttonHandler_releaseDetected())
```

```
        {
            simonRegion = simonbuttonHandler_getRegionNumber();
            timer_waiting = INIT_ZERO;
            simonbuttonHandler_disable();

            verify_state = verify_validate;
        }
        break;


    //verify_validate
    //compares the verifyValue to the sequence value to see if the sequence is over
    //if they are not the same, user input error has occurred
    case verify_validate:

        if(simonRegion == globals_getSequenceValue(verifyValue))
        {
            verifyValue++;
            if(verifyValue == verify_current_sequence_length)
            {
                verifyValue = INIT_ZERO;
                verify_state = verify_complete;
            }

            else
            {
                timer_waiting = INIT_ZERO;
                verify_state = verify_waiting;
                simonbuttonHandler_enable();
            }
        }

        else
        {
            verify_isUserInputError = V_ENABLED;
            verify_state = verify_complete;
        }
        break;


    //verify_complete
    //signifies the state machine has completed
    //restarts if and only if the state machine is enabled
    case verify_complete:

        verify_completed = V_ENABLED;
        timer_waiting = INIT_ZERO;

        if(verify_enable)
        {
            verify_state = verify_init;
        }
        break;


      default:
            verify_state = verify_init;
            break;
      }
}



//*************************************************************************************************
// verifySequence_printInstructions(uint8_t length, bool startingOver)
// Prints the instructions that the user should follow when
// testing the verifySequence state machine.
// Takes an argument that specifies the length of the sequence so that
// the instructions are tailored for the length of the sequence.
// This assumes a simple incrementing pattern so that it is simple to
// instruct the user.
//*************************************************************************************************
void verifySequence_printInstructions(uint8_t length, bool startingOver)
{
    display_fillScreen(DISPLAY_BLACK);         // Clear the screen.
    display_setTextSize(MESSAGE_TEXT_SIZE);   // Make it readable.
    display_setCursor(MESSAGE_X, MESSAGE_Y);  // Rough center.
```

```
 if (startingOver)// Print a message if you start over.
    {
        display_fillScreen(DISPLAY_BLACK);     // Clear the screen if starting over.
        display_setTextColor(DISPLAY_WHITE);    // Print whit text.
        display_println("Starting Over. ");
    }

    display_println("Tap: ");
    display_println();

    switch (length)
    {
        case 1:
            display_println("red");
            break;
        case 2:
            display_println("red, yellow ");
            break;
        case 3:
            display_println("red, yellow, blue ");
            break;
        case 4:
            display_println("red, yellow, blue, green ");
            break;

        default:
            break;
    }

    display_println("in that order.");
    display_println();
    display_println("hold BTN0 to quit.");
}


//*************************************************************************************************
// incrementSequenceLength(int16_t sequenceLength)
// @param: int16_t sequenceLength
// This will set the sequence to a simple sequential pattern.
// Increment the sequence length making sure to skip over 0.
// Used to change the sequence length during the test.
//*************************************************************************************************
int16_t incrementSequenceLength(int16_t sequenceLength) {
  int16_t value = (sequenceLength + ONE) % (MAX_TEST_SEQUENCE_LENGTH + ONE);

    if (value == ZERO)
    {
        value++;
    }
    return value;
}


//*************************************************************************************************
// verifySequence_printInfroMessage(verifySequence_infoMessage_t messageType)
// @param: verifySequence_infoMessage_t messageType
// Prints out informational messages based upon a message type (see above).
//*************************************************************************************************
void verifySequence_printInfoMessage(verifySequence_infoMessage_t messageType)
{
    display_setTextColor(DISPLAY_WHITE);
    display_setCursor(MESSAGE_X, MESSAGE_Y);

    switch(messageType)
    {
        case user_time_out_e:  // Tell the user that they typed too slowly.
            display_println("Error:");
            display_println();
            display_println("  User tapped sequence");
            display_println("  too slowly.");
            break;

        case user_wrong_sequence_e:  // Tell the user that they tapped the wrong color.
            display_println("Error: ");
            display_println();
            display_println("  User tapped the");
            display_println("  wrong sequence.");
            break;
```

```
        case user_correct_sequence_e:  // Tell the user that they were correct.
            display_println("User tapped");
            display_println("the correct sequence.");
            break;

        case user_quit_e:             // Acknowledge that you are quitting the test.
            display_println("quitting runTest().");
            break;

        default:
            break;
  }
}


//*********************************************************************************************
// verifySequence_eraseInfroMessage(verifySequence_infoMessage_t messageType)
// @param: verifySequence_infoMessage_t messageType
// Erases informational messages based upon a message type (see above).
//*********************************************************************************************
void verifySequence_eraseInfoMessage(verifySequence_infoMessage_t messageType)
{
    display_setTextColor(DISPLAY_BLACK);
    display_setCursor(MESSAGE_X, MESSAGE_Y);

    switch(messageType)
    {
        case user_time_out_e:  // Tell the user that they typed too slowly.
            display_println("Error:");
            display_println();
            display_println("  User tapped sequence");
            display_println("  too slowly.");
            break;

        case user_wrong_sequence_e:  // Tell the user that they tapped the wrong color.
            display_println("Error: ");
            display_println();
            display_println("  User tapped the");
            display_println("  wrong sequence.");
            break;

        case user_correct_sequence_e:  // Tell the user that they were correct.
            display_println("User tapped");
            display_println("the correct sequence.");
            break;

        case user_quit_e:             // Acknowledge that you are quitting the test.
            display_println("quitting runTest().");
            break;

        default:
            break;
  }
}


//*********************************************************************************************
// verifySequence_runTest()
// Tests the verifySequence state machine.
// It prints instructions to the touch-screen. The user responds by tapping the
// correct colors to match the sequence.
// Users can test the error conditions by waiting too long to tap a color or
// by tapping an incorrect color.
//*********************************************************************************************
void verifySequence_runTest()
{
    display_init();  // Always must do this.
    buttons_init();  // Need to use the push-button package so user can quit.
    int16_t sequenceLength = ONE;  // Start out with a sequence length of 1.
    verifySequence_printInstructions(sequenceLength, false);  // Tell the user what to do.
    utils_msDelay(MESSAGE_WAIT_MS);  // Give them a few seconds to read the instructions.
    simonDisplay_drawAllButtons()    // Now, draw the buttons.
    // Set the test sequence and it's length.
    globals_setSequence(verifySequence_testSequence, MAX_TEST_SEQUENCE_LENGTH);
    globals_setSequenceIterationLength(sequenceLength);
    // Enable the verifySequence state machine.
    verifySequence_enable();           // Everything is interlocked, so first enable the machine.
    while (!(buttons_read() & BTN0))// Need to hold button until it quits as you might be stuck in a delay.
    {
```

```
        // verifySequence uses the buttonHandler state machine so you need to "tick" both of them.
        verifySequence_tick();  // Advance the verifySequence state machine.
        simonbuttonHandler_tick();   // Advance the buttonHandler state machine.
        utils_msDelay(ONE_MS);       // Wait 1 ms.
        // If the verifySequence state machine has finished, check the result, otherwise just keep ticking both
machines.
    if (verifySequence_isComplete())
    {
      if (verifySequence_isTimeOutError())
      {                      // Was the user too slow?
        verifySequence_printInfoMessage(user_time_out_e);         // Yes, tell the user that they were too slow.
      }
      else if (verifySequence_isUserInputError())
      {             // Did the user tap the wrong color?
        verifySequence_printInfoMessage(user_wrong_sequence_e);   // Yes, tell them so.
      }
      else
      {
        verifySequence_printInfoMessage(user_correct_sequence_e); // User was correct if you get here.
      }
      utils_msDelay(MESSAGE_WAIT_MS);                             // Allow the user to read the message.
      sequenceLength = incrementSequenceLength(sequenceLength);  // Increment the sequence.
      globals_setSequenceIterationLength(sequenceLength);  // Set the length for the verifySequence state machine.
      verifySequence_printInstructions(sequenceLength, V_ENABLED);    // Print the instructions.
      utils_msDelay(MESSAGE_WAIT_MS);                             // Let the user read the instructions.
      verifySequence_drawButtons();                               // Draw the buttons.
      verifySequence_disable();                                   // Interlock: first step of handshake.
      verifySequence_tick();                                      // Advance the verifySequence machine.
      utils_msDelay(ONE_MS);                                          // Wait for 1 ms.
      verifySequence_enable();                                    // Interlock: second step of handshake.
      utils_msDelay(ONE_MS);                                          // Wait 1 ms.
    }
  }
  verifySequence_printInfoMessage(user_quit_e);  // Quitting, print out an informational message.
}
```

```
#ifndef SIMONCONTROL_H_
#define SIMONCONTROL_H_

#include <stdbool.h>
#include <stdint.h>
#include "globals.h"
#include "simonDisplay.h"
#include "simonButtonHandler.h"
#include "simonFlashSequence.h"
#include "simonVerifySequence.h"
#include <stdio.h>

void SimonControl_tick();

#endif
```

```
//SimonControl.c
#include "SimonControl.h"
#include "supportFiles/display.h"

#define INIT_ZERO 0
#define CONTROL_FOUR 4
#define CONTROL_ONE 1
#define TEXT_SIZE_TWO 2
#define TEXT_SIZE_FOUR 4
```

```
#define NEW_LEVEL_SIZE 25
#define SCORE_MESSAGE_SIZE 20
#define NEW_LEVEL_AD_TIMER_EXPIRED 10
#define FINAL_DELAY_EXPIRED 2
#define DISPLAY_SCORE_TIMER_EXPIRED 40
#define SIMON_AD_TIMER_EXPIRED 1
#define WAIT_FOR_PLAYER_TIMER_EXPIRED 30
#define SEQUENCE_SIZE 50
#define LAST_DELAY_EXPIRED 10
#define WAITING_AD_TIMER 20
#define CURSOR_WIDTH display_width()/3.5
#define CURSOR_HEIGHT display_height()/3
#define DISPLAY_WIDTH_EIGHTS display_width()/8
#define DISPLAY_HEIGHT_HALVES display_height()/2
#define DISPLAY_WIDTH_FIFTHS display_width()/5
#define DISPLAY_HEIGHT_THIRDS display_width()/3
#define NEXT_LINE_WIDTH display_width()/7
#define NEXT_LINE_HEIGHT display_height()/3 * 1.5
#define YAY_TIMER_EXPIRED 5
#define LC_EXPIRED 5

#define SIMON "Simon"
#define PROMPT_TOUCH "touch to start game"
#define CONGRATULATIONS "Congratulations!"
#define YAY "Yay!"

uint8_t simonControl_sequence[SEQUENCE_SIZE];
uint16_t sequence_length = SEQUENCE_SIZE;
int16_t wait_for_player_timer = 0;
int16_t simon_ad_timer = 0;
int16_t display_score_timer = 0;
int16_t final_delay = 0;
int16_t new_level_ad_timer = 0;
int16_t level = 1;
int16_t squares_per_level = 4;
uint16_t length = 1;
uint16_t last_delay = 0;
uint16_t waiting_ad_timer = 0;
uint16_t sc_yay_count = 0;
uint16_t pread = 0;
uint16_t lc = 0;
uint16_t max_length = 0;

bool screen_erase = true;
bool screen_display = false;
bool show_score = false;
char newLevel_message[NEW_LEVEL_SIZE];
char score_message[SCORE_MESSAGE_SIZE];


enum simonControl_States {sc_init, sc_waiting, sc_flash_pause, sc_debounce, sc_ad_timer, sc_yay_state, sc_flash,
sc_wait_touch, sc_verify, sc_new_level, sc_new_level_ad_timer, sc_display_score,sc_score_running, sc_game_over,
sc_last_delay, sc_last_score} sc_state;

//*********************************************************************************************
//@start_message(bool erase)
//@param: bool erase
//Prints a starting message to the touch screen
//If erase is false, the start message will be printed in white
//If erase is true, the start message will be printed over in black
//*********************************************************************************************
void start_message(bool erase)
{
    if(erase == false)
    {
        display_setTextColor(DISPLAY_WHITE);
    }

    else
    {
        display_setTextColor(DISPLAY_BLACK);
    }

    display_setCursor(CURSOR_WIDTH, CURSOR_HEIGHT);
    display_setTextColor(DISPLAY_WHITE);
    display_setTextSize(TEXT_SIZE_FOUR);
    display_println(SIMON);
    display_setCursor(NEXT_LINE_WIDTH, NEXT_LINE_HEIGHT);
    display_setTextSize(TEXT_SIZE_TWO);
```

```
    display_println(PROMPT_TOUCH);
}

//*****************************************************************************************
//@new_level_message(bool erase)
//@param: bool erase
//Prints a starting message to the touch screen
//If erase is false, the new level message will be printed in white
//If erase is true, the new level message will be printed over in black
//*****************************************************************************************
void new_level_message(bool erase)
{
    if(erase == false)
    {
        display_setTextColor(DISPLAY_WHITE);
    }

    else
    {
        display_setTextColor(DISPLAY_BLACK);
    }

    display_setCursor(DISPLAY_WIDTH_FIFTHS, DISPLAY_HEIGHT_THIRDS );
    display_setTextSize(TEXT_SIZE_TWO);
    display_println(CONGRATULATIONS);
    display_setCursor(NEXT_LINE_WIDTH, NEXT_LINE_HEIGHT);
    display_setTextSize(TEXT_SIZE_TWO);
    sprintf(newLevel_message, "touch to start level %2u", level);
    display_println(newLevel_message);
}

//*****************************************************************************************
//@display_yay(bool erase)
//@param: bool erase
//Prints a starting message to the touch screen
//If erase is false, Yay! will be printed in white
//If erase is true, Yay! will be printed over in black
//*****************************************************************************************
void display_yay(bool erase)
{
    if(erase)
    {
        display_setTextColor(DISPLAY_BLACK);
    }
    else
    {
        display_setTextColor(DISPLAY_WHITE);
    }

    display_setCursor(DISPLAY_WIDTH_FIFTHS, DISPLAY_HEIGHT_THIRDS);
    display_setTextSize(TEXT_SIZE_FOUR);
    display_println(YAY);
}

//*****************************************************************************************
//@show_score_message(bool erase)
//@param: bool erase
//Prints a starting message to the touch screen
//If erase is false, the end level score message will be printed in white
//If erase is true, the end level message will be printed over in black
//*****************************************************************************************
void show_score_message(bool erase)
{
    if(erase)
    {
        display_setTextColor(DISPLAY_BLACK);
    }
    else
    {
        display_setTextColor(DISPLAY_WHITE);
    }
    display_setCursor(DISPLAY_WIDTH_EIGHTS, DISPLAY_HEIGHT_HALVES);
    display_setTextSize(TEXT_SIZE_TWO);
    sprintf(score_message, "Longest sequence: %2u", length);
    display_println(score_message);
}

//*****************************************************************************************
//@show_lc(bool erase)
```

```
//@param: bool erase
//Prints a starting message to the touch screen
//If erase is false, the longest count message will be printed in white
//If erase is true, the longest count message will be printed over in black
//*************************************************************************************************
void show_lc(bool erase)
{
    if(erase)
    {
        display_setTextColor(DISPLAY_BLACK);
    }
    else
    {
        display_setTextColor(DISPLAY_WHITE);
    }
    display_setCursor(DISPLAY_WIDTH_EIGHTS, DISPLAY_HEIGHT_HALVES);
    display_setTextSize(TEXT_SIZE_TWO);
    sprintf(score_message, "Longest sequence: %2u", max_length);
    display_println(score_message);
}


//*************************************************************************************************
//@screateRandomArray()
// Creates a random number using the rand()
// The array is then set to be the global sequence array
//*************************************************************************************************
void createRandomArray()
{
    for(int i = INIT_ZERO; i < SEQUENCE_SIZE; i++)
    {
        int16_t random = rand() % CONTROL_FOUR;
        simonControl_sequence[i] = random;
    }
    globals_setSequence(simonControl_sequence, sequence_length);
}


//*************************************************************************************************
//@adjustMaxLength(uint16_t num)
//@param: uint16_t num
//Maximizes the max_length integer variable to be printed if the game ends prematurely
//*************************************************************************************************
void adjustMaxLength(uint16_t num)
{
      if(num > max_length)
      {
        max_length = num;
      }
}


//*************************************************************************************************
//@SimonControl_tick()
//The Control State Machine
//*************************************************************************************************
void SimonControl_tick()
{
    switch(sc_state)
    {
        //sc_init
        //Initializes the amount of correct presses to zero
        //Disables the state machines so they do not incorrectly run
        //Displays a start screen
        case sc_init:
        max_length = INIT_ZERO;

            createRandomArray();
            globals_setSequenceIterationLength(length);
            verifySequence_disable();
            flashSequence_disable();

            start_message();
            sc_state = sc_waiting;
            break;


        //sc_waiting
        //Transitions to the ad_timer debouncer when the screen is touched
```

```
        case sc_waiting:
            if(display_isTouched())
            {
                display_clearOldTouchData();
                sc_state = sc_ad_timer;
            }
            break;


        //sc_ad_timer
        //Adds a debounce value for screen presses. Once the value is achieved, it transitions the flash state
        //Else, it returns to the sc_waiting state
        case sc_ad_timer:
            simon_ad_timer++;
            wait_for_player_timer = INIT_ZERO;
            if(display_isTouched() && simon_ad_timer == SIMON_AD_TIMER_EXPIRED)
            {
                 verifySequence_disable();
                start_message(screen_erase);
                flashSequence_enable();
                sc_state = sc_flash;
            }

            else if(!display_isTouched() && simon_ad_timer == SIMON_AD_TIMER_EXPIRED )
            {
                sc_state = sc_waiting;
            }
            break;


        //sc_flash
        //Disables the flashSequence state machine and enables the verifySequence state machine
        case sc_flash:
            simon_ad_timer = INIT_ZERO;
            if(flashSequence_completed())
            {
                flashSequence_disable();
                verifySequence_enable();
                sc_state = sc_verify;
            }
            break;


        //sc_verify
        //Once the verifySequence is complete, a transition will occur based on the conditions of verifyComplete
        //If the user does not press fast enough or presses incorrectly, game over occurs
        //If the user completes a level, the yay state occurs
        //If the user completes a sequence and the level is not over, the next flash sequence occurs
        case sc_verify:
            if(verifySequence_isComplete() && (verifySequence_isTimeOutError() || verifySequence_isUserInputError())
)
            {
                verifySequence_disable();

                sc_state = sc_game_over;
            }

            else if(verifySequence_isComplete() && globals_getSequenceIterationLength() == squares_per_level)
            {
                level++;
                verifySequence_disable();
                display_yay(screen_display);
                squares_per_level++;

                sc_state = sc_yay_state;
            }

            else if(verifySequence_isComplete() && globals_getSequenceIterationLength() != squares_per_level)
            {
                length++;
                 verifySequence_disable();
                flashSequence_enable();
                adjustMaxLength(length);
                globals_setSequenceIterationLength(length);

                sc_state = sc_flash;
            }
            break;
```

```
//sc_yay_state
//Congratulates the user on completing a level
//The yay message is displayed up until sc_yay_count equals YAY_TIMER_EXPIRED
//The state machine transitions to a new level
case sc_yay_state:
 sc_yay_count++;
 if(sc_yay_count == YAY_TIMER_EXPIRED)
 {
         display_yay(screen_erase);

         sc_state = sc_new_level;
 }
 break;


//sc_new_level
//Offers the user a chance to play a new, longer level
case sc_new_level:
 sc_yay_count = INIT_ZERO;
    new_level_message(display_screen);

    sc_state = sc_wait_touch;
    break;


//sc_wait_touch
//If the user accepts a new level before the time expires, the game will continue
//else, the final score state is transitioned to
case sc_wait_touch:
    wait_for_player_timer++;

    if(wait_for_player_timer == WAIT_FOR_PLAYER_TIMER_EXPIRED)
    {
         new_level_message(screen_erase);

        sc_state = sc_display_score;
    }

    if(display_isTouched())
    {
        createRandomArray();

        sc_state = sc_new_level_ad_timer;
    }

    break;


//sc_display_score
//Reinitializes various timer values for a new game
//Shows the score
case sc_display_score:
    wait_for_player_timer = INIT_ZERO;
    display_score_timer = INIT_ZERO;
    squares_per_level = CONTROL_FOUR;
    level = CONTROL_ONE;

    show_score_message(screen_display);

    sc_state = sc_score_running;
    break;


//sc_score_running
//Displays the score for a given amount of time
//Returns to the init state
case sc_score_running:
    display_score_timer++;

    if(display_score_timer == DISPLAY_SCORE_TIMER_EXPIRED)
    {
        length = CONTROL_ONE;
        show_score_message(screen_erase);

        sc_state = sc_init;
    }

    break;
```

```
//sc_new_level_ad_timer
//Debounces the touch for the user to go to a new level
//Once touched, the game will begin in the flash state
case sc_new_level_ad_timer:
    new_level_ad_timer++;
    if(display_isTouched())
    {
        new_level_message(true);
        length = CONTROL_ONE;
        globals_setSequenceIterationLength(length);
        flashSequence_enable();
        new_level_ad_timer=INIT_ZERO;

        sc_state = sc_flash;
    }

    else if(!display_isTouched() && new_level_ad_timer == NEW_LEVEL_AD_TIMER_EXPIRED)
    {
        sc_state = sc_wait_touch;
        new_level_ad_timer=INIT_ZERO;
    }

    break;


//sc_game_over
//Resets values
//Displays the corresponding game over screen for a time FINAL_DELAY_EXPIRED
case sc_game_over:
    final_delay++;

    length = CONTROL_ONE;
    if(final_delay == FINAL_DELAY_EXPIRED || verifySequence_isUserInputError())
    {
        final_delay = INIT_ZERO;
        verifySequence_printInfoMessage(user_wrong_sequence_e);

        sc_state = sc_last_delay;
    }

    if (verifySequence_isTimeOutError())
    {
        final_delay = INIT_ZERO;
        verifySequence_printInfoMessage(user_time_out_e);

        sc_state = sc_last_delay;
    }
    break;


//sc_last_delay
//Holds the game over display on the screen for a time LAST_DELAY_EXPIRED
//Shows the score for the longest count during the game
case sc_last_delay:
 last_delay++;
 if(last_delay == LAST_DELAY_EXPIRED)
 {
        last_delay = INIT_ZERO;
         verifySequence_eraseInfoMessage(user_wrong_sequence_e);
         verifySequence_eraseInfoMessage(user_time_out_e);

         show_lc(display_screen);

         sc_state = sc_last_score;
 }
 break;


//sc_last_score
//Shows up the longest count during the game for time LC_EXPIRED
case sc_last_score:
 lc++;

 if(lc == LC_EXPIRED)
 {
        lc = INIT_ZERO;
         show_lc(screen_erase);
```

```
                sc_state = sc_init;
        }
        break;


        default:
            sc_state = sc_init;
            break;

    }

}
```