```c
//*****************************************************************************
// Copyright (c) 2014 Texas Instruments Incorporated.  All rights reserved.
// Software License Agreement
//
//   Redistribution and use in source and binary forms, with or without
//   modification, are permitted provided that the following conditions
//   are met:
//
//   Redistributions of source code must retain the above copyright
//   notice, this list of conditions and the following disclaimer.
//
//   Redistributions in binary form must reproduce the above copyright
//   notice, this list of conditions and the following disclaimer in the
//   documentation and/or other materials provided with the
//   distribution.
//
//   Neither the name of Texas Instruments Incorporated nor the names of
//   its contributors may be used to endorse or promote products derived
//   from this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// This file was automatically generated by the Tiva C Series PinMux Utility
// Version: 1.0.4
//
//*****************************************************************************
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include "pong.h"
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "inc/hw_gpio.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/gpio.h"
#include "driverlib/ssi.h"
#include "inc/tm4c123gh6pm.h"
#include "driverlib/timer.h"
#include "driverlib/interrupt.h"
#include "driverlib/adc.h"

//*****************************************************************************
#define NUM_SSI_DATA 8

// LED 8x8 Configuration:
// H  G  F  E  D  C  B  A  #
// H0 G0 F0 E0 D0 C0 B0 A0 0
// H1 G1 F1 E1 D1 C1 B1 A1 1
// .  .  .  .  .  .  .  .  .
// .  .  .  .  .  .  .  .  .
// .  .  .  .  .  .  .  .  .
// H7 G7 F7 E7 D7 C7 B7 A7 7

// Array of 8-bit numbers defines LED's on: {A7-0, B7-0, C7-0, D7-0, E7-0, F7-0, G7-0, H7-0}

//_____________________________________________________________________________
//
//** VARIABLE AND ARRAY DECLARATIONS
//______________________________________________________________________________

void gamePlay(int tick);
void updateDisplay(void);
void setGame(void);
void gameOverDisplay(void);

unsigned short paddles[NUM_SSI_DATA] = {0x38,0x00,0x00,0x00,0x00,0x00,0x00,0x38}; // paddles are 3 bits wide 0x38 = 00111000
unsigned short pongBall[NUM_SSI_DATA] = {0x00,0x00,0x00,0x08,0x00,0x00,0x00,0x00};// pong ball is one bit
unsigned short ulDataTx[NUM_SSI_DATA] = {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};// display array sent thru ssi
unsigned short levelsArr[3][NUM_SSI_DATA] = {               // levels select prompt array
{0x00,0x00,0x00,0x20,0x3E,0x24,0x00,0x00},                  // display for '1'
{0x00,0x00,0x00,0x2E,0x2A,0x3A,0x00,0x00},                  // display for '2'
{0x00,0x00,0x00,0x3E,0x2A,0x2A,0x00,0x00}                   // display for '3'
};

unsigned short gameOverArr[2][NUM_SSI_DATA] = {             // sad face towards whoever lost
{0x00,0x00,0x14,0x00,0x1C,0x22,0x00,0x00},
{0x00,0x00,0x22,0x1C,0x00,0x14,0x00,0x00}
};

uint32_t rightPaddleADCValue, leftPaddleADCValue;           // adc values to determine position of left or right paddles
uint32_t adcValuesRight[4],adcValuesLeft[4];
bool score = true;                                          // when a player scores
bool levelSet = false;                                      // used to determine if a level has been selected yet
bool gameOver = false;                                      // used to determine when the game is over
int user2Score = 0;                                         // users scores
int user1Score = 0;
int winner;                                                 // used to determine winner for display
int level= 0;                                               // determine which level to play
int ballSpeed = 12;                                         // variable to control ball update speed
int tick = 0;                                               // variable used to handle update speeds

//_____________________________________________________________________________
//
//** INITIALIZATION FUNCTIONS **
//_____________________________________________________________________________

void PortFunctionInit(void)
{

    volatile uint32_t ui32Loop;

    SYSCTL_RCGC2_R = SYSCTL_RCGC2_GPIOF;                    // Enable the clock of the GPIO port that is used for the on-board LED and switch.

    ui32Loop = SYSCTL_RCGC2_R;

    GPIO_PORTF_LOCK_R = 0x4C4F434B;                         // Unlock GPIO Port F
    GPIO_PORTF_CR_R |= 0x01;                                // allow changes to PF0

    GPIO_PORTF_DIR_R &= ~0x11;                              // Set the direction of PF4 (SW1) and PF0 (SW2) as input by clearing the bit

    GPIO_PORTF_DIR_R |= 0x04;                               // This pin is used for testing PF2
    GPIO_PORTF_DEN_R |= 0X04;

    GPIO_PORTF_DEN_R |= 0x11;                               // Enable PF4 and PF0 for digital function.

    GPIO_PORTF_PUR_R |= 0x11;                               // Enable pull-up on PF4 and PF0

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);            // port E init ADC CH0, CH1
    GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_3|GPIO_PIN_2);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
    GPIOPinTypeGPIOOutput(GPIO_PORTC_BASE, GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7);
    GPIOPinTypeGPIOOutput(GPIO_PORTA_BASE, GPIO_PIN_2|GPIO_PIN_3|GPIO_PIN_4);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI1);             // Enable SSI1
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);            // Enable GPIO Port D
```

```c
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);            // Enable GPIO Port F

                                                            // Configure muxing and GPIO settings to set SSI functions to pins
    GPIOPinConfigure(GPIO_PD0_SSI1CLK);                     // Configure clock for SPI1
    GPIOPinConfigure(GPIO_PD1_SSI1FSS);                     // Configure frame signal for SPI1
    GPIOPinConfigure(GPIO_PF1_SSI1TX);                      // Configure transmit for SPI1
    GPIOPinTypeSSI(GPIO_PORTD_BASE,GPIO_PIN_0|GPIO_PIN_1);  // Configure pins for use by SSI peripheral
    GPIOPinTypeSSI(GPIO_PORTF_BASE,GPIO_PIN_1);

    // Configure SPI port: (on SSI1, clock source, the mode (0-3), master or slave, bit rate, data width)
    SSIConfigSetExpClk(SSI1_BASE,SysCtlClockGet(),SSI_FRF_MOTO_MODE_0,SSI_MODE_MASTER,10000,16);
    SSIEnable(SSI1_BASE);
}

void ADC0_Init(void)
{

    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);             // activate the clock of ADC0
    SysCtlDelay(2);                                         // insert a few cycles after enabling the peripheral to allow the clock to be fully activated.

    ADCSequenceDisable(ADC0_BASE, 1);                       // disable ADC0 before the configuration is complete

    ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 1);// ADC0 SS3 Step 0, sample from ch0 (PE3), completion of this step will set RIS, only sample of the sequence

    ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_CH0); // ADC0 SS1 Step 0, sample from ain0
    ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_CH0); // ADC0 SS1 Step 1, sample from ain0
    ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_CH0); // ADC0 SS1 Step 2, sample from ain0
    ADCSequenceStepConfigure(ADC0_BASE,1,3,ADC_CTL_CH0|ADC_CTL_IE|ADC_CTL_END); //ADC0 SS1 Step 0, sample from ain0, completion of this step will set RIS, last sample of the sequence

    IntPrioritySet(INT_ADC0SS1, 3);                         // configure ADC0 SS1 interrupt priority as 3
    IntEnable(INT_ADC0SS1);                                 // enable interrupt 33 in NVIC (ADC0 SS1)// data sheet 103
    ADCIntEnableEx(ADC0_BASE, ADC_INT_SS1);                 // arm interrupt of ADC0 SS1

    ADCSequenceEnable(ADC0_BASE, 1);                        // enable ADC0
}
void ADC1_Init(void)
{

    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC1);             // activate the clock of ADC1
    SysCtlDelay(2);                                         // insert a few cycles after enabling the peripheral to allow the clock to be fully activated.

    ADCSequenceDisable(ADC1_BASE, 1);                       // disable ADC1 before the configuration is complete

    ADCSequenceConfigure(ADC1_BASE, 1, ADC_TRIGGER_PROCESSOR, 2);// ADC1 SS3 Step 0, sample from ch1 (PE2), completion of this step will set RIS, only sample of the sequence

    ADCSequenceStepConfigure(ADC1_BASE, 1, 0, ADC_CTL_CH1); // ADC0 SS1 Step 0, sample from ain1
    ADCSequenceStepConfigure(ADC1_BASE, 1, 1, ADC_CTL_CH1); // ADC0 SS1 Step 1, sample from ain1
    ADCSequenceStepConfigure(ADC1_BASE, 1, 2, ADC_CTL_CH1); // ADC0 SS1 Step 2, sample from ain1
    ADCSequenceStepConfigure(ADC1_BASE,1,3,ADC_CTL_CH1|ADC_CTL_IE|ADC_CTL_END); // ADC0 SS1 Step 0, sample from ain1, completion of this step will set RIS, last sample of the sequence

    IntPrioritySet(INT_ADC1SS1, 4);                         // configure ADC1 SS1 interrupt priority as 4
    IntEnable(INT_ADC1SS1);                                 // enable interrupt 33 in NVIC (ADC1 SS1)// data sheet 103
    ADCIntEnableEx(ADC1_BASE, ADC_INT_SS1);                 // arm interrupt of ADC1 SS1

    ADCSequenceEnable(ADC1_BASE, 1);                        // enable ADC1
}

//Globally enable interrupts
void IntGlobalEnable(void)
{
    __asm("    cpsie   i\n");
```

```c
}

//Globally disable interrupts
void IntGlobalDisable(void)
{
    __asm("    cpsid   i\n");
}

void
Interrupt_Init(void)
{
  NVIC_EN0_R |= 0x40000000;                                 // enable interrupt 30 in NVIC (GPIOF)// DATA SHEET PG 141
  NVIC_PRI7_R &= ~0x00E00000;                               // configure GPIOF interrupt priority as 0
  GPIO_PORTF_IM_R |= 0x11;                                  // arm interrupt on PF0 and PF4
  GPIO_PORTF_IS_R &= ~0x11;                                 // PF0 and PF4 are edge-sensitive
  GPIO_PORTF_IBE_R &= ~0x11;                                // PF0 and PF4 not both edges trigger DATA SHEET 658
  GPIO_PORTF_IEV_R &= ~0x11;                                // PF0 and PF4 falling edge event

  IntGlobalEnable();                                        // globally enable interrupt
}

void Timer1A_Init(unsigned long period)
{
  //
  // Enable Peripheral Clocks
  //
  SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);             // peripheral driver pg 483
  TimerConfigure(TIMER1_BASE, TIMER_CFG_PERIODIC);          // configure for 32-bit timer mode
  TimerLoadSet(TIMER1_BASE, TIMER_A, period -1);            // reload value
  IntPrioritySet(INT_TIMER1A, 0x02);                        // configure Timer1A interrupt priority as 2
  IntEnable(INT_TIMER1A);                                   // enable interrupt in NVIC (Timer1A)
  TimerIntEnable(TIMER1_BASE, TIMER_TIMA_TIMEOUT);          // arm timeout interrupt
  TimerEnable(TIMER1_BASE, TIMER_A);                        // enable timer1A
}

//_____________________________________________________________________________
//
//** HANDLER FUNCTIONS **
//_____________________________________________________________________________

//interrupt handler
void ADC0_Handler(void)
{
    ADCIntClear(ADC0_BASE, 1);
    ADCSequenceDataGet(ADC0_BASE, 1, adcValuesRight);       // usable paddle data is the average of 4 readings from the adc pin
    rightPaddleADCValue = (adcValuesRight[0] + adcValuesRight[1] +
      adcValuesRight[2] + adcValuesRight[3])/4;
}
void ADC1_Handler(void)
{
    ADCIntClear(ADC1_BASE, 1);
    ADCSequenceDataGet(ADC1_BASE, 1, adcValuesLeft);
    leftPaddleADCValue = (adcValuesLeft[0] + adcValuesLeft[1] +
      adcValuesLeft[2] + adcValuesLeft[3])/4;
}

//GPIO interrupt handler
void GPIOPortF_Handler(void)
{
                                                            // debounce by disabling interupt, waiting, then re-enabling it
  NVIC_EN0_R &= ~0x40000000;                                // disable interrupt 30 in NVIC (GPIOF)
  SysCtlDelay(74000);                                       // Delay for a while
  NVIC_EN0_R |= 0x40000000;                                 // re-enable interrupt 30 in NVIC (GPIOF)

  if(GPIO_PORTF_RIS_R&0x10)                                 // SW1 has action
  {
    GPIO_PORTF_ICR_R |= 0x10;                               // acknowledge flag for PF4
                                                            // check for action instead of just
checking for pin status to account for switch bouncing
    if((GPIO_PORTF_DATA_R&0x10)==0x00)                       // SW1 is pressed
    {
      level++;
      if(level > 2)
        level = 0;
    }
  }

  if(GPIO_PORTF_RIS_R&0x01)                                  // SW2 has action
  {
    GPIO_PORTF_ICR_R |= 0x01;                                // acknowledge flag for PF0

    if((GPIO_PORTF_DATA_R&0x01)==0x00)
    {
      levelSet = true;                                       // proceed to play game
    }
  }
}

//interrupt handler for Timer1A
void Timer1A_Handler(void)
{
  if(!gameOver){
  // test one start point
    if(!levelSet){
      setGame();
    }
    else{
      tick++;                                                // increments a count for every frame
      ADCProcessorTrigger(ADC0_BASE, 1);                     // trigger ADC processor every 3 "frames"
      ADCProcessorTrigger(ADC1_BASE, 1);
      //** test two end point: adc function time 452 ns; **
      //** test three start point **
      gamePlay(tick);//
      //** test three end point: gameplay function time  range 3.652 - 9.652 us **
      //** start point test four **
      updateDisplay();
      //** test four end point: display function time 13.6 ms **
      TimerIntClear(TIMER1_BASE, TIMER_TIMA_TIMEOUT);        // acknowledge flag for Timer1A timeout
    }
  //** test one finish point: total time to complete game 13.6 ms **
  //** headroom: 13.4 ms **
  }
  else
    gameOverDisplay();                                       // display frowny face
}

//______________________________________________________________________________
//
//** GAMEPLAY FUNCTIONS **
//______________________________________________________________________________

unsigned short ucIndex,ucReversedNumber;

unsigned char Reverse(unsigned char ucNumber)                // Reverse() takes 8-bit input number and reverses it.
{
    ucReversedNumber = 0;
    //** start of test six **
    for(ucIndex=0; ucIndex<8; ucIndex++)
    {
        ucReversedNumber = ucReversedNumber << 1;
        ucReversedNumber |= ((1 << ucIndex) & ucNumber) >> ucIndex;
    }
    //** end point of test six: reverse function time 1.85 us **
return ucReversedNumber;

}

int initBallPos= 3;                                          // initial ball position in array (pongBall[3])
int ballPos = 3;                                            // initial ball position in array
bool ballGoingLeft = false;
int upDown = -1;                                            // going up = 1, down = -1, staying level = 0

//!!display matrix is formatted so the msb is on bottom, lsb is on top!!

void goDown(){                                              // function to make the ball go down by binary shifting the value in the array towards msb by 1
  pongBall[ballPos] = pongBall[ballPos] << 1;
}
void goUp(){                                                // function to make the ball go up by binary shifting the value in the array towards lsb by 1
  pongBall[ballPos] = pongBall[ballPos] >> 1;
}

//!!display matrix is formatted so the first entry of the array is the far right of the display!!

void goRight(){                                             // function to make the ball go right, by setting the value of the ball in its current position
                                                            // to the position just before it in the array then clearing the current position and making the new position the current one
  pongBall[ballPos-1] = pongBall[ballPos];                  // position just before current position is now equal to the current position
  pongBall[ballPos] = 0x00;                                 // clear the ball from current position
  ballPos--;                                                // ball position value is now one less than what it was
}

void goLeft(){                                              // function to make the ball go left, by setting the value of the ball in its current position
                                                            // to the position just after it in the array then clearing the current position and making the new position the current one
  pongBall[ballPos+1] = pongBall[ballPos];
  pongBall[ballPos] = 0x00;
  ballPos++;
}

unsigned short pTop, pMiddle, pBottom, ballNextPos;         // values for top, middle, or bottom of paddle and the balls next position

bool checkIntersect(int lr){                                // lr is an int to determine if i'm checking left or right paddle intersect w ball 0 for right, 7 for left
                                                            // checks each of the 6 positions the the paddle could be in and determine which bit is the top, middle, or bottom
  switch (paddles[lr]){                                     // this is to allow ball to change dirction when hit in certain spot of paddle
    case 0x07 : pTop = 0x01; pMiddle = 0x02; pBottom = 0x04; break;
    case 0x0E : pTop = 0x02; pMiddle = 0x04; pBottom = 0x08; break;
    case 0x1C : pTop = 0x04; pMiddle = 0x08; pBottom = 0x10; break;
    case 0x38 : pTop = 0x08; pMiddle = 0x10; pBottom = 0x20; break;
    case 0x70 : pTop = 0x10; pMiddle = 0x20; pBottom = 0x40; break;
    case 0xE0 : pTop = 0x20; pMiddle = 0x40; pBottom = 0x80; break;
  }

  if(pongBall[ballPos] & paddles[lr]){                      // general check intersect value
    if(pongBall[ballPos] == pTop){
      if(upDown == -1)                                      // if coming down on paddle, bounce off middle
        upDown = 0;
      else                                                  // if it hits the top of paddle go up
        upDown = 1;
    }
    else if(pongBall[ballPos]== pBottom){
      if(upDown == 1)                                       // if coming up on paddle, bounce off middle
        upDown = 0;
      else                                                  // if it hits the bottom of paddle go down
        upDown = -1;
    }
    else if(pongBall[ballPos] == pMiddle){
      if(upDown == 0)                                       // if coming up or down on paddle continue in current direction
        upDown = 0;
    }
    return true;
  }                                                         // next bit is to check if were hitting the top or bottom ends of the paddle for Chris
  else{
    if(upDown == 1){                                        // if going up, paddle position is one up from where we check
      if(pongBall[ballPos]!= 0x01){                         // if the ball is not about to hit the top of screen
        ballNextPos = pongBall[ballPos] >> 1;
        if(ballNextPos == pBottom){
          upDown = -1;
          return true;
        }
    }
      else{                                                 // if it is about to hit top of screen
        ballNextPos = pongBall[ballPos] << 1;               // next position is down one
        if(ballNextPos == pTop){
          upDown = 1;
          return true;
        }
      }
    }
    else if(upDown == -1){                                  // if going down paddle position is one down from where we check
      if(pongBall[ballPos] != 0x80){                        // if it's not about to hit the bottom of screen
        ballNextPos = pongBall[ballPos] << 1;
        if(ballNextPos == pTop){
          upDown = 1;
          return true;
        }
      }
      else{                                                 // if it is about to hit bottom of screen
        ballNextPos = pongBall[ballPos] >> 1;
        if(ballNextPos == pBottom){                         // next position is up one
          upDown = -1;
          return true;
        }
      }
    }
  }

  return (pongBall[ballPos] & paddles[lr]);                 // returns true if they are about to hit
}

void setBall(){                                             // this function resets the ball to its original position
  pongBall[initBallPos] = 0x08;
  ballPos = initBallPos;                                    // ballPos is the current position of the ball which is now reset to its original position in the array
  upDown = 0;                                               // ball resets just going horizontal
}

void updateBall(){
                                                            // moves the ball to the left
  if(ballGoingLeft){                                        // ball is going towards the left -> user2
    if(ballPos == 7){                                       // ball hasn't been hit by paddle
      user1Score ++;                                        // user scores
      pongBall[ballPos] = 0x00;                             // clear ball from screen
      score = true;                                         // score is true which resets the ball to the middle
      tick = 0;
    }
    else if(ballPos == 6 && checkIntersect(7)){             // when the ball is in front of the paddle, check for hit
        ballGoingLeft = false;                              // if hit, ball now goes right
    }
    else{                                                   // ball has not gotten to position 6, been hit, or scored: continue to go right
      goLeft();
    }
  }
                                                            // moves the ball to the right
  if(!ballGoingLeft){                                       // going towards right
    if(ballPos == 0){                                       // checks for score
      user2Score++;
      pongBall[ballPos] = 0x00;
      score = true;
      tick = 0;
    }
    else if (ballPos == 1 && checkIntersect(0)){            // when the ball is in front of the paddle, check for hit
      ballGoingLeft = true;                                 // ball now goes towards right
      goLeft();                                             // need to tell it to go left from here since it wont check and go left till next cycle
    }
    else{                                                   // continue right
      goRight();
    }
  }
                                                            // moves the ball down
  if(upDown == -1){                                         // if the ball is going down
    if(pongBall[ballPos]==0x80){                            // if the ball is hitting the bottom, bottom is 0x80
      upDown = 1;                                           // start going up
      goUp();
    }
    else
      goDown();                                             // else keep going down
  }
  else if(upDown == 1){                                     // if the ball is going up
    if(pongBall[ballPos] == 0x01){                          // if the ball is hitting the top
      upDown = -1;
      goDown();
    }
    else
      goUp();
  }
}

unsigned long ulindex,ulData;

void setGame(){                                             // used to display numbers for level set

    for(ulindex = 0; ulindex < NUM_SSI_DATA; ulindex++)
    {                                                       // Create 16-bit data word using Reverse function
      ulData = (Reverse(levelsArr[level][ulindex]) << 8) + (1 << ulindex);
      SSIDataPut(SSI1_BASE, ulData);                        // Place data in transmit FIFO buffer using blocking function
      while(SSIBusy(SSI1_BASE))                             // Wait until data has been transmitted
      {
      }
    }
}

void gameOverDisplay(){// used to display numbers for level set

    for(ulindex = 0; ulindex < NUM_SSI_DATA; ulindex++)
    {                                                       // Create 16-bit data word using Reverse function
      ulData = (Reverse(gameOverArr[winner][ulindex]) << 8) + (1 << ulindex);
      SSIDataPut(SSI1_BASE, ulData);                        // Place data in transmit FIFO buffer using blocking function
      while(SSIBusy(SSI1_BASE))                             // Wait until data has been transmitted
      {
      }
    }
}

void updateDisplay(void){
                                                            // for loop to step through data, sending each 16-bit word one at a time
    for(ulindex = 0; ulindex < NUM_SSI_DATA; ulindex++)
    { //** test five start point **
      ulData = (Reverse(ulDataTx[ulindex]) << 8) + (1 << ulindex);// Create 16-bit data word using Reverse function
      //** test seven start point **
      SSIDataPut(SSI1_BASE, ulData);                        // Place data in transmit FIFO buffer using blocking function
      //** test seven end point: SSI FIFO buffer put time 450 ns **
      //** test eight sart point **
      while(SSIBusy(SSI1_BASE))                             // Wait until data has been transmitted
      {
      }
      //** test eight end point: buffer wait time 1.7 ms **
      //** test five end point: single loop run time 1.702 ms **
    }

}
void updateScore(){                                         // function to update scoreboard

  switch(user1Score){
    case 1: GPIO_PORTA_DATA_R |= 0x04; break;
    case 2: GPIO_PORTA_DATA_R |= 0x08; break;
    case 3: GPIO_PORTA_DATA_R |= 0x10; winner = 0;
            gameOver = true; break;                         // when score hits 3, game is over
  }
    switch(user2Score){
    case 1: GPIO_PORTC_DATA_R |= 0x20; break;
    case 2: GPIO_PORTC_DATA_R |= 0x40; break;
    case 3: GPIO_PORTC_DATA_R |= 0x80; winner = 1;
            gameOver = true; break;
  }

}

void gamePlay(int tick){

    if(level<2)                                             // levels one and two have defined ball speeds
      ballSpeed = 12 - (3*level);                           // the higher the level, the more frequent the ball updates
    else                                                    // level three has a variable ball speed that increases as time goes on
      if((tick % 150 == 0) && ballSpeed > 2)
        ballSpeed --;

    if(score){                                              // reset the ball and let it sit for a bit
      setBall();
      ballSpeed = 12;                                       // reset ball speed for level 3
      updateScore();
      if(tick % 75 == 0){                                   // wait a few frames
      score = false;
      ballGoingLeft = !ballGoingLeft;                       // ball goes towards scorer
      }
    }
    else if(tick % ballSpeed == 0){                         // every 'ballspeed' frames, move the ball
      updateBall();
    }

    if(rightPaddleADCValue <= 1022)                         // this checks what position the paddles should be in based on the ADC reading
        paddles[0] = 0x07;
    else if(rightPaddleADCValue > 1022 && rightPaddleADCValue <= 1533)
        paddles[0] = 0x0E;
    else if(rightPaddleADCValue > 1533 && rightPaddleADCValue <= 2044)
        paddles[0] = 0x1C;
    else if(rightPaddleADCValue > 2044 && rightPaddleADCValue <= 2555)
        paddles[0] = 0x38;
    else if(rightPaddleADCValue > 2555 && rightPaddleADCValue <= 3066)
        paddles[0] = 0x70;
    else if(rightPaddleADCValue > 3066)
        paddles[0] = 0xE0;

```

```c
    if(leftPaddleADCValue <= 1022)                           // this checks what position the paddles should be in based on the ADC reading
        paddles[7] = 0x07;
    else if(leftPaddleADCValue > 1022 && leftPaddleADCValue <= 1533)
        paddles[7] = 0x0E;
    else if(leftPaddleADCValue > 1533 && leftPaddleADCValue <= 2044)
        paddles[7] = 0x1C;
    else if(leftPaddleADCValue > 2044 && leftPaddleADCValue <= 2555)
        paddles[7] = 0x38;
    else if(leftPaddleADCValue > 2555 && leftPaddleADCValue <= 3066)
        paddles[7] = 0x70;
    else if(leftPaddleADCValue > 3066)
        paddles[7] = 0xE0;

    for(int i = 0; i < NUM_SSI_DATA; i ++){
      ulDataTx[i] = pongBall[i] | paddles[i];                // combine paddle and ball array to one to send to display
    }

}

int main(void)
{
    SysCtlClockSet(SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ); // 80MHz
    PortFunctionInit();
    Timer1A_Init(SysCtlClockGet()/60);                       // LED array has no latch, data must be continuously streamed in order for static image to appear
    ADC0_Init();                                             // periodic imer used to better control frame rate
    ADC1_Init();
    Interrupt_Init();
    IntMasterEnable();

    GPIO_PORTA_DATA_R &= ~0x1C;                              // start with all scoreboard LEDs off
    GPIO_PORTC_DATA_R &= ~0xE0;

    while(1)
    {

    }
}
```